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

Peter Oberrauner

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

Peter Oberrauner B.Sc.

Master's Thesis

to achieve the university degree of

Diplom-Ingenieur

Master's Degree Programme: Software Engineering and Management

submitted to

Graz University of Technology

Supervisor

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

Graz, 03 Jan 2022

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

Peter Oberrauner B.Sc.

Masterarbeit

für den akademischen Grad

Diplom-Ingenieur

Masterstudium: Software Engineering and Management

an der

Technischen Universität Graz

Begutachter

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

Graz, 03 Jan 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

Abstract

[TODO: Write Abstract]

keywords:

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

Kurzfassung

[TODO: Translate abstract into german]

Contents

List of Figures

List of Tables

List of Listings

Acknowledgements

[TODO: Write acknoledgements]

Peter Oberrauner
Graz, Austria, 03 Jan 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

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

[TODO: Outline the various chapters]

Chapter 2

Web Technologies

RespVis is a web-based framework. As such, it builds on a stack of technologies which are native to the web. The first sections in this chapter introduce the web's core technologies: HTML 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. 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.

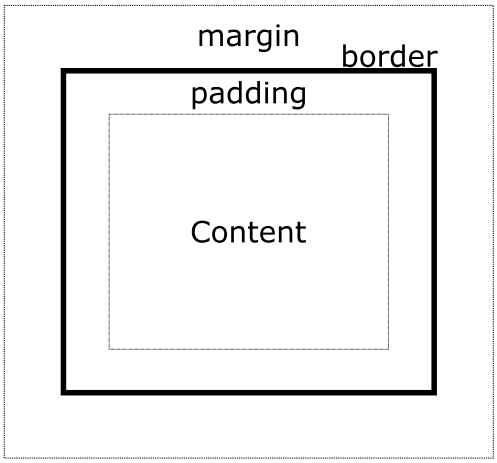


Figure 2.1: The CSS box model defines the properties of boxes which wrap around HTML elements. [Image drawn by the author of this thesis.]

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 `display:flex` property on a container element. The direction of the layout can then be specified using the `flex-direction` property which can be set to either `row` or `column`. The items inside a Flexbox container can have either a fixed or a relative size. When items should be sized relative to the size of their containers, the proportions of how the available space should be divided can be controlled using ratios. These ratios can be set on item elements via the `flex` property.

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

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

illustrated in Figure 2.2. Alignment of items on the cross axis is achieved either by the `align-items` property on the container element or the `align-self` property on the items themselves.

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

2.2.3 CSS Grid Layout

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

Grid layout of elements is enabled by setting the `display:grid` property on their container. The grid in which items shall be laid out is then defined using the `grid-template-rows` and `grid-template-columns` properties. In addition, the `grid-template` property can be used as a shorthand to simultaneously specify both the rows and columns of a grid. Item elements need to specify the cell of the grid into which they shall be positioned. This is done with the `grid-row` and `grid-column` properties, which take the corresponding row and column indices as values. Items can also be configured to span multiple cells by specifying index ranges as the values of those properties.

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

There are also properties which control the layout of items within grid cells and the layout of grid cells themselves. Similar to Flexbox, this can be configured with the `align-items` and `justify-items` properties for laying out within grid cells, and the `align-content` and `justify-content` properties for laying out the grid cells themselves. The latter `*-content` properties only make sense when the cells

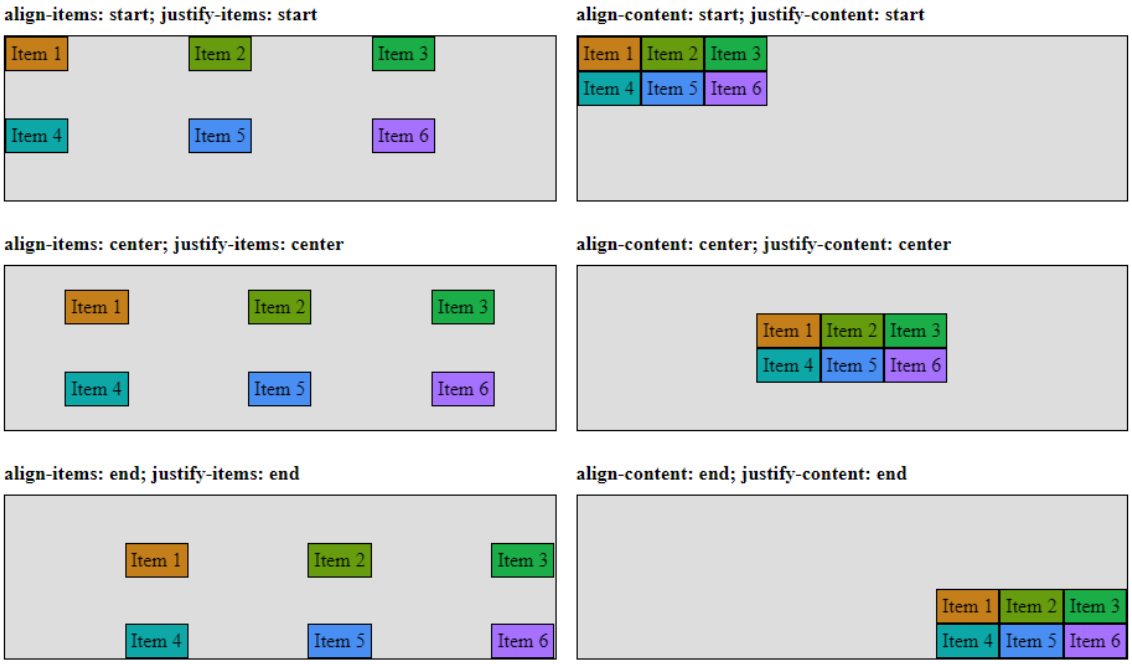


Figure 2.3: The `*-items` properties are used to lay out items within their grid cells, whereas the `*-content` properties are used to lay out the grid cells themselves. [Image created by the author of this thesis.]

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 the 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

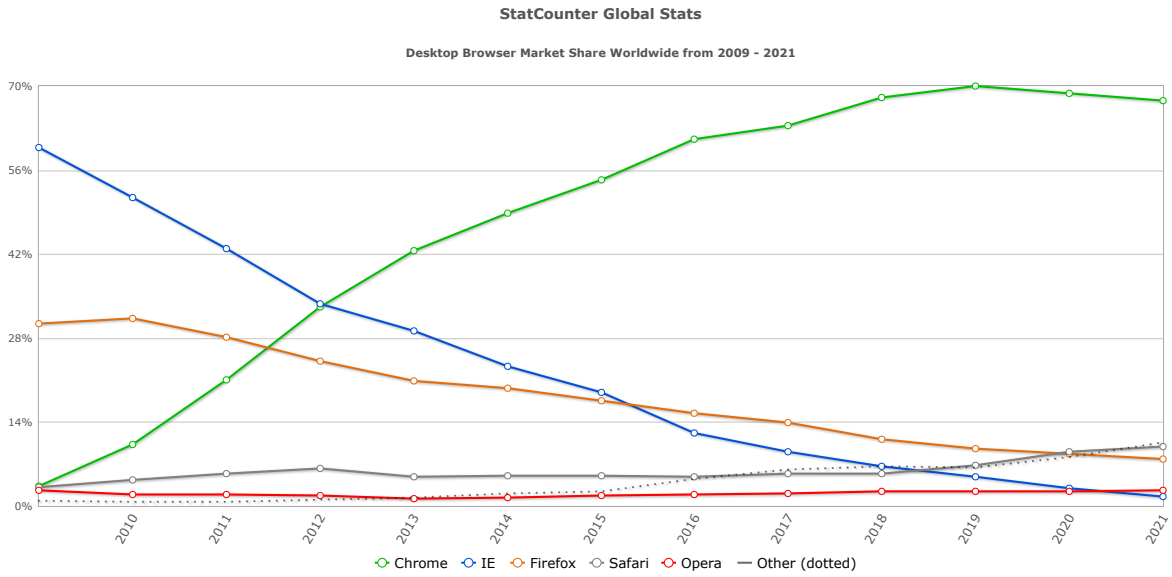


Figure 2.4: Since their release, Firefox and Chrome have contested the monopoly of the Internet Explorer and continuously gained more market share. Recently, Chrome seems to be gaining an increasingly strong position within the market. [Image taken from StatCounter [2021]]

the uncontested monopoly of the Internet Explorer [Routley 2021] 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 latest, widely support version, ECMAScript 6, was released in 2015 [ECMA 2015].

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.

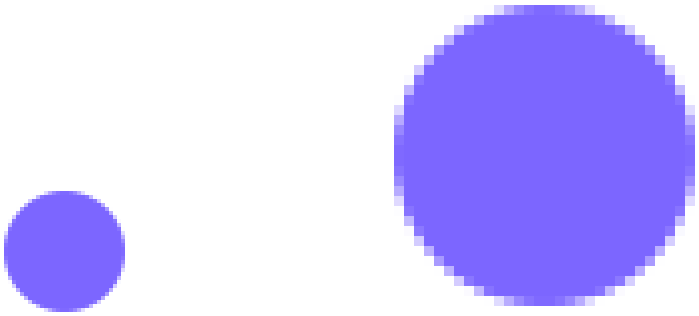
Initially, TypeScript was released by Microsoft in 2012 [Hoban 2012] to extend JavaScript with features which were already present in more mature languages, and whose absence in JavaScript caused difficulties when working on larger codebases. At the time of TypeScript's initial development, it provided 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 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.

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



(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.]

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.

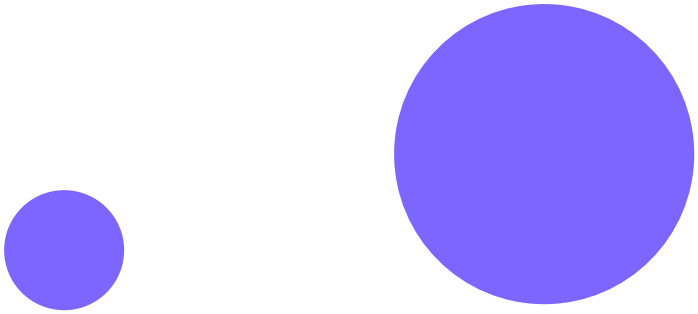
2.5.2 Scalable Vector Graphics (SVG)

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] and the most recent version SVG 1.1 was released in 2011 [Dahlström et al. 2011]. 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 exist which 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.

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
<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.]

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)

[TODO: Link to the correct specification introducing canvas (https://www.w3.org/standards/history/2dcontext)]

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

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml" lang="en" xml:lang="en">
<body>
  <canvas width="64" height="64"></canvas>
  <canvas width="640" height="640"></canvas>
  <script>
    const canvases = Array.from(document.getElementsByTagName('canvas'));
    canvases.forEach((canvas) => {
      const width = canvas.clientWidth;
      const height = canvas.clientHeight;
      const context = canvas.getContext('2d');
      context.fillStyle = '#7c66ff';
      context.beginPath();
      context.arc(width / 2, height / 2, width / 2, 0, Math.PI * 2);
      context.fill();
      context.closePath();
    });
  </script>
</body>
</html>
```

Listing 2.2: A basic HTML document containing two canvases of different sizes which render circles relative to the canvas size. The visual representation of this document is shown in Figure 2.7.

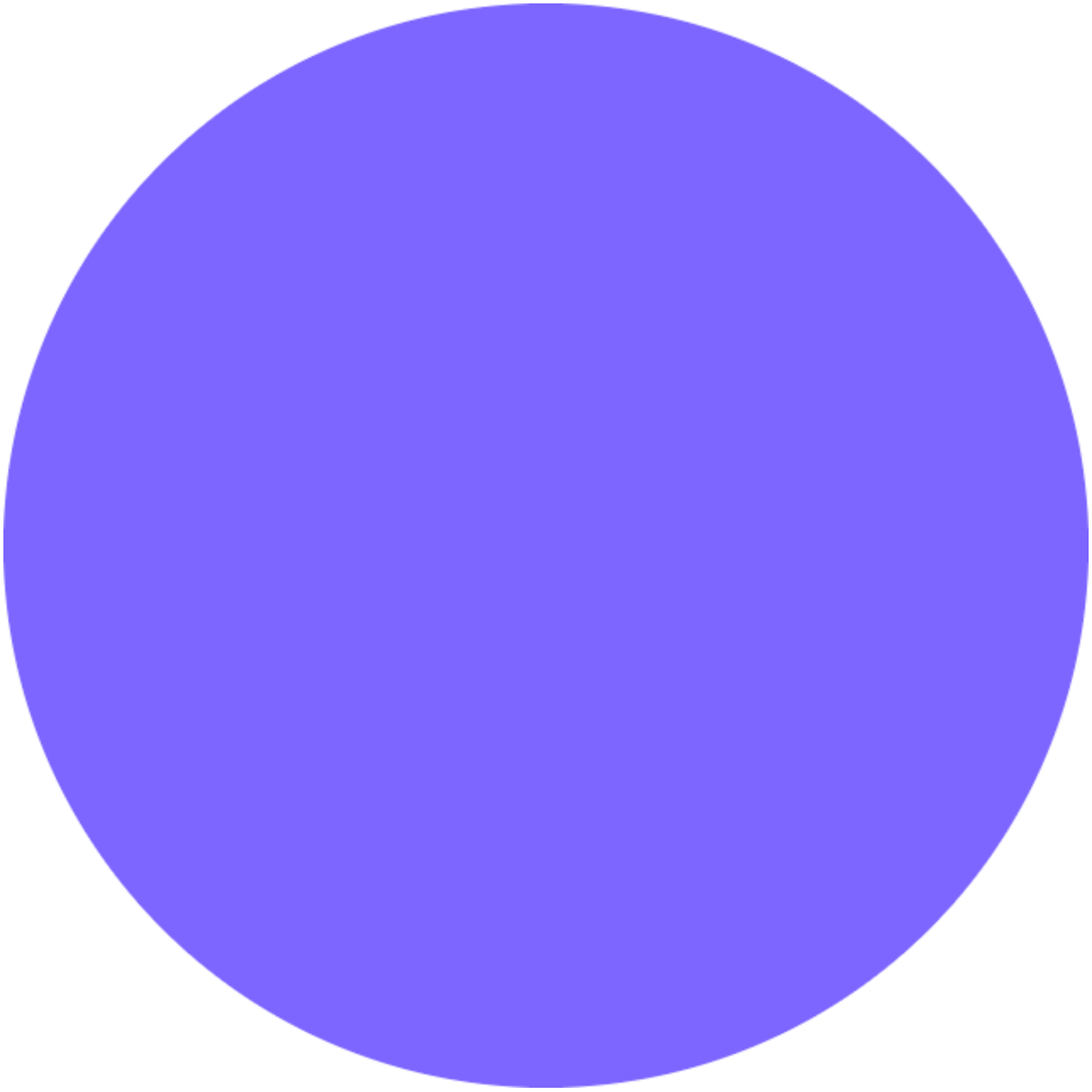
The `2d` rendering context enables platform-independent 2d rendering via a software renderer, whose API is standardized directly in the canvas specification [WHATWG 2021]. An example of an HTML document containing two differently sized canvases into which responsive circles are drawn using a 2d rendering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

[TODO: Change size of circle canvases]

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]. Three-dimensional rendering through WebGL hardware-accelerated and is often much faster than rendering via Canvas 2d or SVG.

(a) 64-pixel canvas.



(b) 640-pixel canvas.

Figure 2.7: Responsive rendering of graphics inside canvas elements has to be implemented manually by calculating everything relative to the canvas' 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 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. Yoga 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

[TODO: Also mention Marcotte's book and not only the article here.]

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 defined by Marcotte [2010], where 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 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.

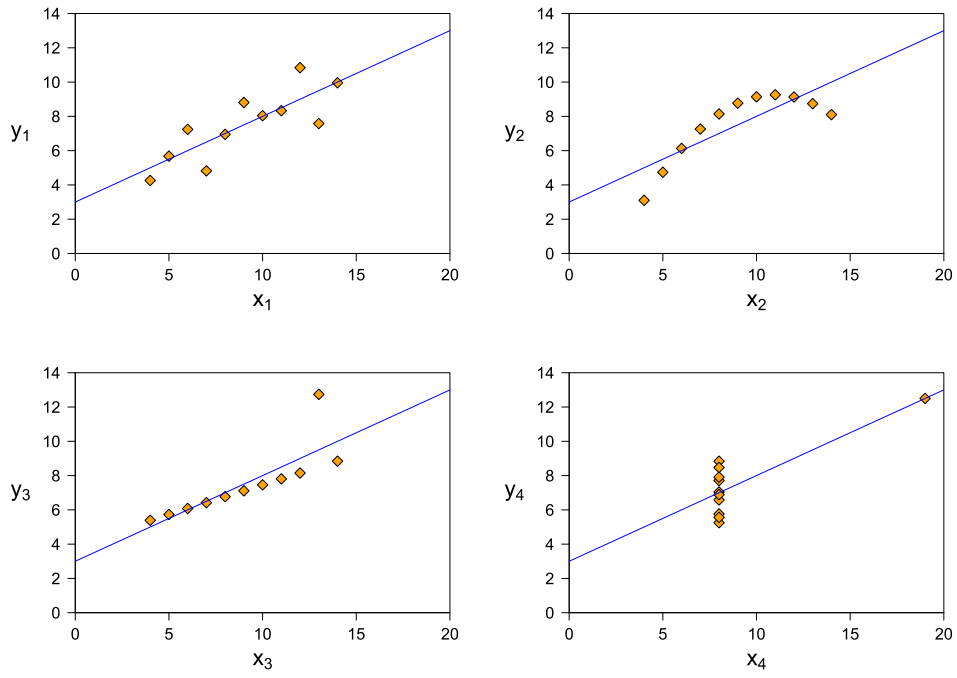


Figure 3.1: When plotted graphically, it is immediately apparent that the four datasets in Anscombe's Quartet are very different. [Image extracted from Andrews [2021]. Used with kind permission by Keith Andrews.]

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 created the chart about the movement of prominent planets [Macrobius 1175] shown in Figure 3.2. Other noteworthy early visualizations include the first occurrence of the principle which Tufte [1983] later called "small multiples" in the 1626 chart by Scheiner [1630] demonstrating sunspot changes shown in Figure 3.3, and the 1644 chart displaying longitudinal distance determinations between Toledo and Rome by Florent [1644] shown in Figure 3.4.

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 3.5), bar charts (Figure 3.6), and area charts (Figure 3.7) to communicate economic factors of England during the eighteenth century. In a related later work [Playfair 1801], he 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] 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 in poor sanitary conditions. One of her contributions to the field of information visualization was the creation of a new

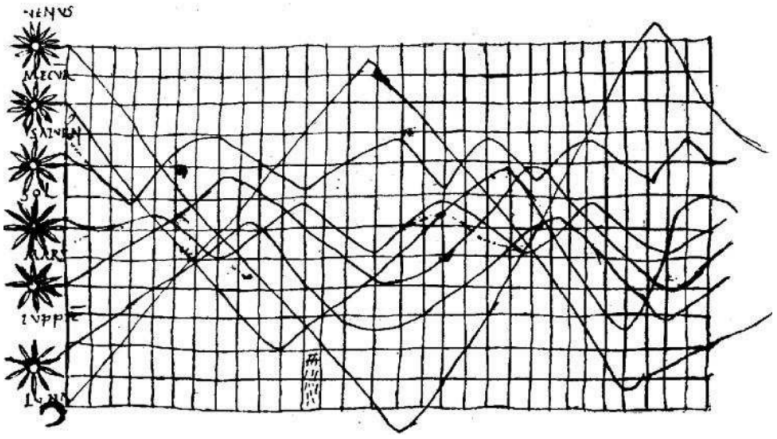


Figure 3.2: A line chart created by an unknown astronomer in the 10th Century, depicting the movements of seven prominent planets. [Image extracted from Friendly [2008]. Original appearance in Macrobius [1175].]

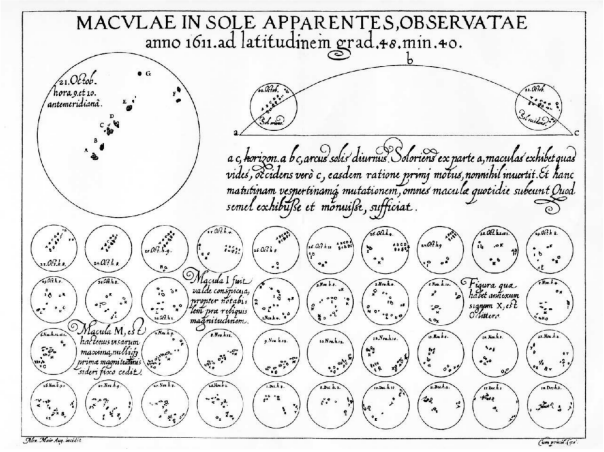


Figure 3.3: The observed changes in sunspots based on recordings of two months of data from 1611. It is the first occurrence of the principle later called "small multiples" by Tufte [1983]. [Image extracted from Friendly [2008]. Original appearance in Scheiner [1630].]

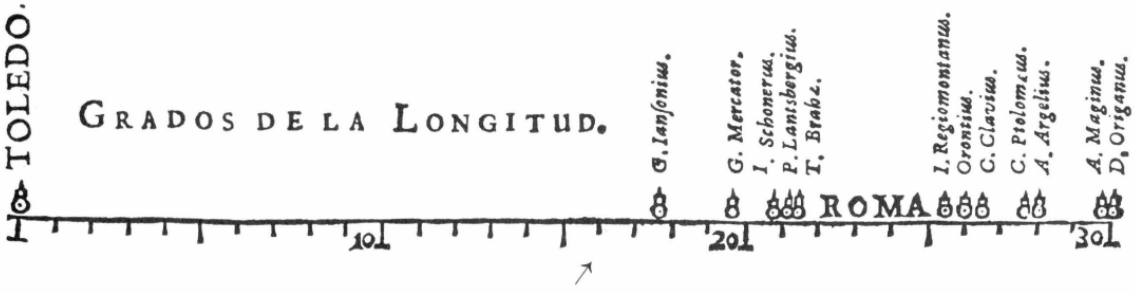


Figure 3.4: A comparison of the twelve known estimates in longitudinal distance between Rome and Toledo by various astronomers. The correct distance is marked by the arrow beneath. It is considered by Tufte [1997, page 15] to be the first visual representation of statistical data. [Image extracted from Friendly [2008]. Original appearance in Florent [1644].]

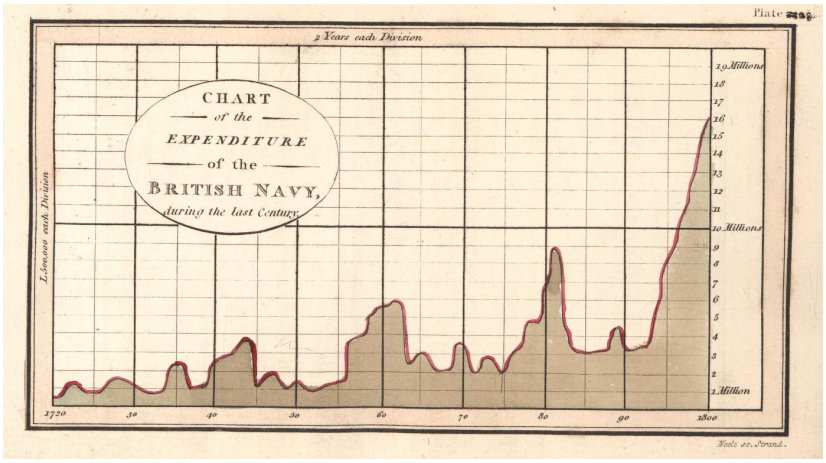


Figure 3.5: Line chart of the expenditure of the British Navy during the 18th Century. It was published in 1786 and is considered to be one of the first occurrences of a line chart containing components found in modern visualizations. [Image extracted from Schoenberg Center for Electronic Text and Image (SCETI). Used under the terms of Creative Commons CC BY 2.5.]

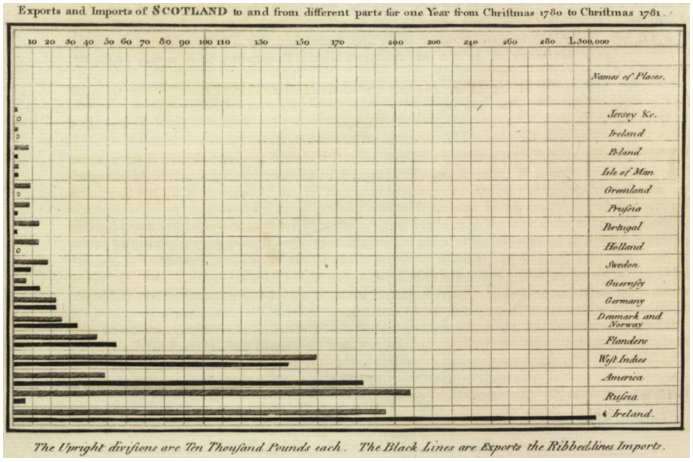


Figure 3.6: Bar chart of England's exports and imports to and from Scotland in 1781. Published in 1786, it is considered to be one of the first occurrences of a bar chart containing most components found in modern visualizations. [Image extracted from Schoenberg Center for Electronic Text and Image (SCETI).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

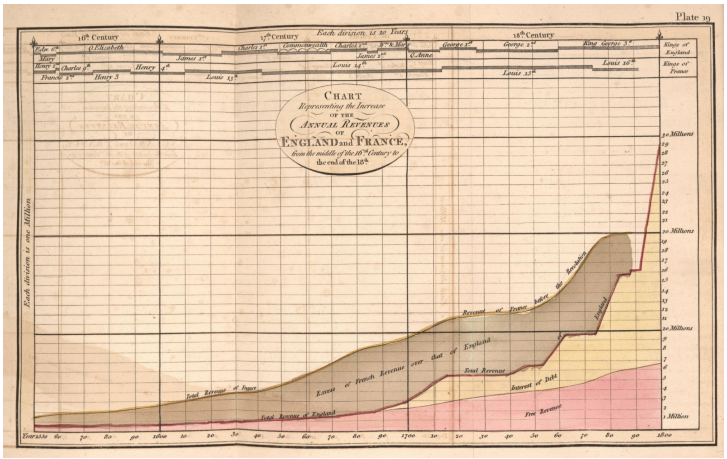


Figure 3.7: Area chart of annual revenues of England and France between 1550 and 1800. Published in 1786, it is considered to be one of the first occurrences of an area chart containing most components found in modern visualizations. [Image extracted from Schoenberg Center for Electronic Text and Image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

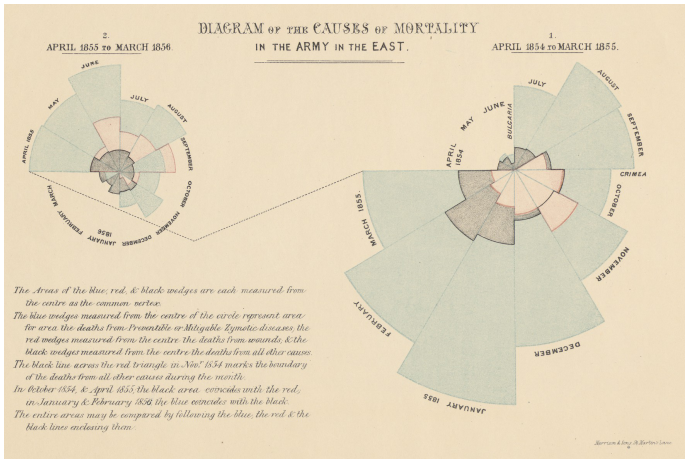


Figure 3.8: One of the polar-area charts created by Florence Nightingale in 1859 to convince people of a need for more sanitary conditions in hospitals. It visualizes the causes of mortality for soldiers during the Crimean War and demonstrates that a large percentage of patients died from preventable diseases linked to unsanitary environments. [Image extracted from Harvard Library. Used under the terms of Creative Commons Attribution 4.0.]

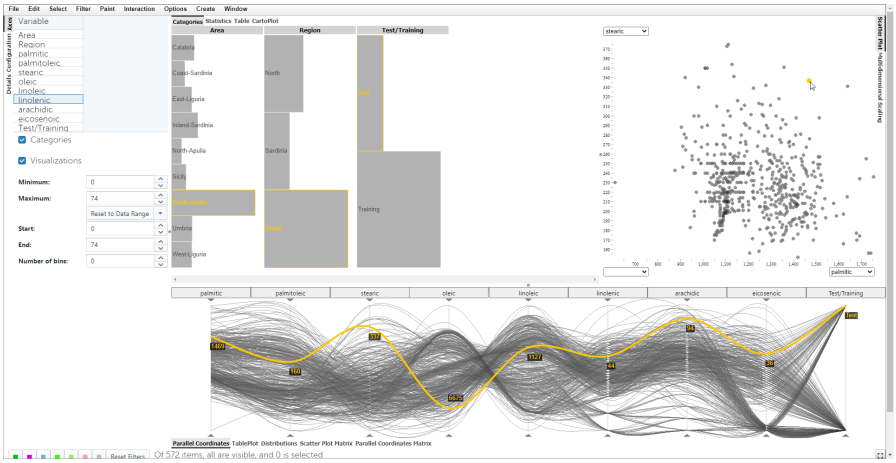


Figure 3.9: High-D by Macrofocus is a visual analytics tool specialized in analyzing multidimensional data. [Screenshot of High-D [Macrofocus 2021] taken by the author of this work.]

type of diagram, called a rose diagram or polar-area chart, shown in Figure 3.8. 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 3.9.

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 Vega-Lite provide a declarative grammar to express the visual and interactive characteristics of a visualization. Template-based visualization libraries provide a higher-level template-based interface which can easily be configured.

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 for 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 `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 Selection of the newly added elements to enable the creation of nested structures. This method chaining syntax is further aided by the `Selection.call` method, which invokes a callback receiving the current Selection as a parameter and returns the original

```
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.

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. Selections 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 the elements which were newly created, the update Selection, containing the elements which merely receive new data, and the exit Selection, containing the elements which are being removed. Each of these Selections can be individually transformed using the `Selection.join` method, which can receive three callbacks 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 Transitions, which wrap a Selection and allow the animation of various element characteristics. Transitions are created using the `Selection.transition` method which creates a Transition wrapping the Selection on which it has been called. The duration of a Transition is defined using the `Transition.duration` method and its easing can be configured using the `Transition.ease` method. It is also possibile to interrupt 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.

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

3.2.2 Vega and Vega-Lite

Vega [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 librarie [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.

```
{
  "$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.

3.2.3 Template-Based Visualization Libraries

Template-based visualization libraries work by providing templates for possible types of visualizations and allowing users to customize them. These types of visualization libraries are easier to use than D3 or Vega because they offer a concise form of configuration which does not require users to have detailed knowledge over the underlying rendering technology or complex, non-standardized domain specific languages. Even though these types of libraries are usually flexible enough to create a huge range of visualizations, at 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, usage popularity (number of downloads), open-source popularity, license, and recent development activity. In terms of rendering technology, most libraries render visualizations as either SVG documents or canvas elements, although some implement a hybrid renderer which can be configured to render as either one of them. Usage popularity was measured

```
{
  "$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.

by the cumulative package downloads from the npm package manager over the previous twelve months. This 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 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 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

```
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.

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 is 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 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.

```
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.

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 designers can make use of CSS-inherent mechanisms to configure responsive styles. Unfortunately, CSS-based styling is only of limited applicability because not all CSS properties affect SVG elements, as described in Section 2.5.2.

To responsively configure other visualization characteristics, such as their type, data, and layout, designers have to resort to configuration mechanisms offered by the libraries. Four libraries (Highcharts, ApexCharts, Chartist, 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, designers have to fall back to more generic mechanisms which are also applicable to other libraries. The mechanisms for responsive configuration in the other libraries are more generic, because they do not offer these configurations as part of their declarative interfaces. This means that developers need to trigger responsive configurations themselves by manually reconfiguring visualizations via their APIs in custom resize or media query event listeners. Nearly all libraries provide

```
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.

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. Korner [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 recent works have surveyed the design space of responsive visualizations and created taxonomies of currently used techniques and recurring patterns [Hoffswell et al. 2020; Adler et al. 2021]. In addition to surveying design patterns, Kim et al. [2021a] also consider issues around "message loss" when reducing graphical complexity.

4.1 Responsive Visualization Patterns

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

Adler et al. [2021] have conducted a survey under close supervision by Keith Andrews [Andrews 2018] identifying nine common patterns which reoccurred in several solutions. Slightly reworded, these patterns are: (1) rotate axis labels, (2) remove axis ticks, (3) modify strings, (4) transpose chart, (5) reposition components, (6) zoom, (7) filter, (8) modify data density and (9) modify chart type. Compared to other

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

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

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

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

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

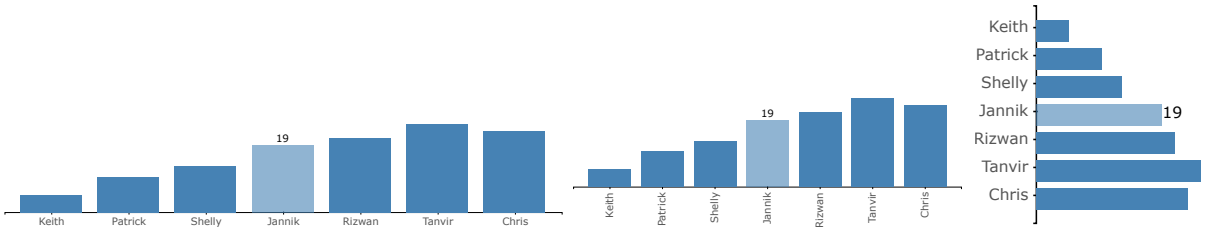
Category	Description
Recompose	Actions which affect the existence of targets. Includes remove, add, replace and aggregate actions.
Rescale	Actions which affect the size of targets. Includes reduce width, simplify labels and elaborate labels actions.
Transpose	Actions which affect the orientation of targets.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

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

4.2 Responsive Visualization Examples

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

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

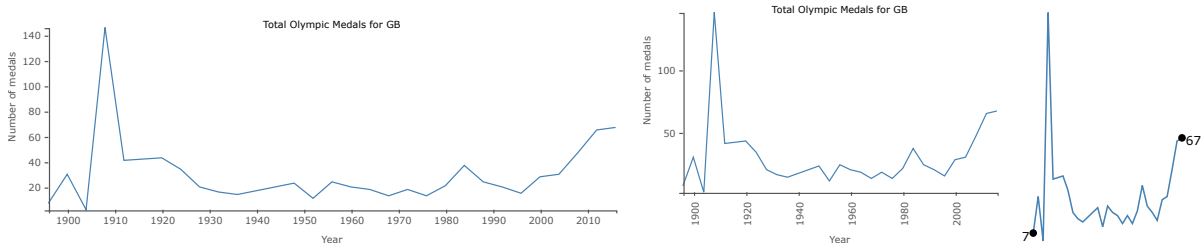


(a) 70rem **(b)** 50rem **(c)** 30rem

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

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

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



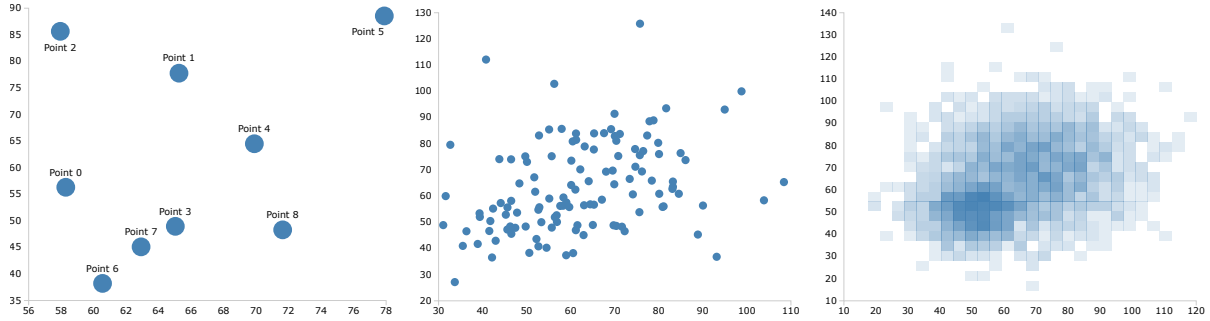
(a) 65rem **(b)** 40rem **(c)** 20rem

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

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

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

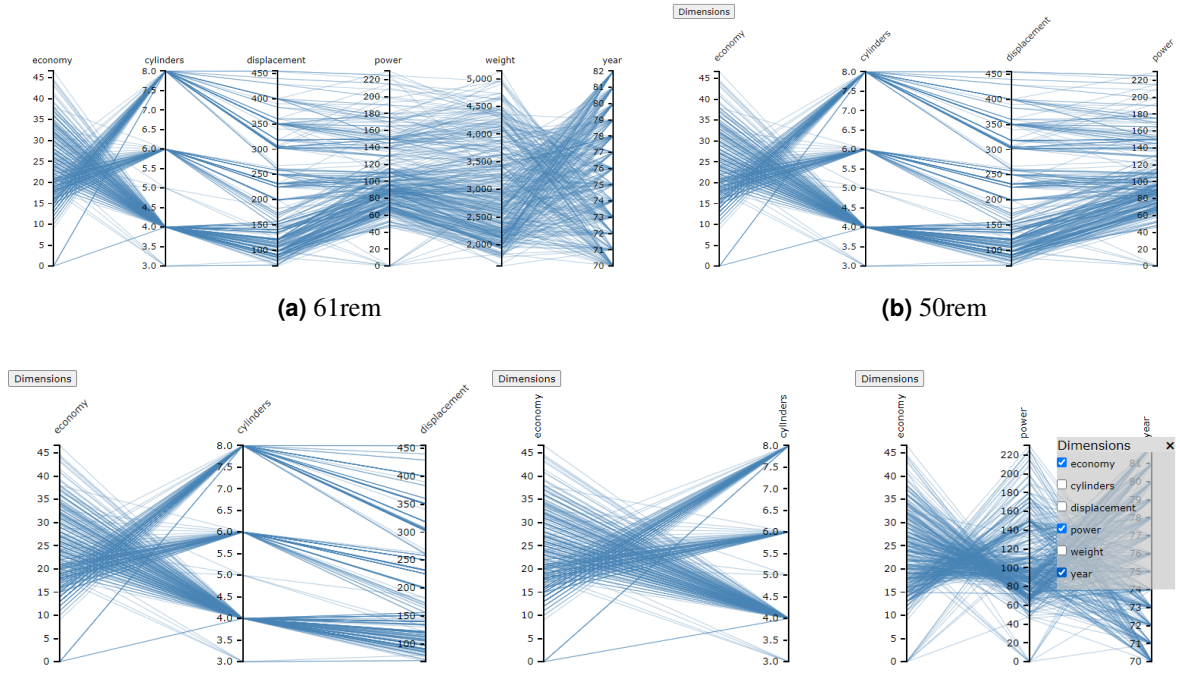
Even though parallel coordinates charts are rarely encountered in non-technical contexts, they are very popular when it comes to visualizing multidimensional data in visual analytics systems [Macrofocus 2021]. In these kinds of charts, multiple dimensions are rendered as parallel axes which are connected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



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

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

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



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

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

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

Chapter 5

The RespVis Library

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

5.1 Design

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

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

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

can only be applied to documents, RespVis does not support rendering to HTML canvas elements because graphics rendered there are not exposed to the document and therefore can not be affected by CSS.

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

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

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

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

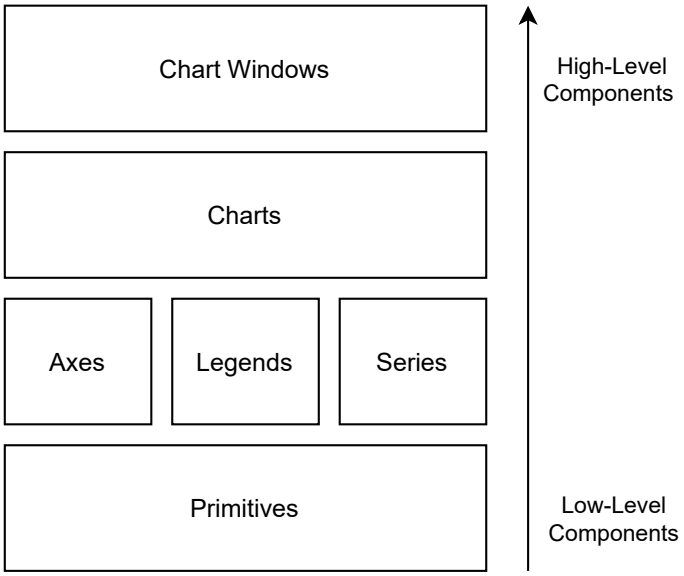


Figure 5.1: This diagram shows the different layers of components of the RespVis library. Layers that are higher in the hierarchy contain increasingly higher-level components that make more assumtions than components in lower layers. The lowest level of components are primitives that are merely rendered SVG elements such as <rect>, <circle>, and <text> elements. Axes, legends and series are composite components that only contain primitives. Charts are composite components that are composed of lower-level primitives or other composite components to form a complete visualization. Chart windows are wrapper components around charts, manage their rendering process and provide a toolbar for them. [Image created by the author of this thesis.]

bugs. These advantages have already been described in Section 2.4.

Sixthly, components in RespVis are structured in layers with different levels of abstraction. Components in higher layers need to make more assumtions about their content than components in lower layers. The bottom-most layer with the lowest level of abstraction consists of visual primitives that are represented by basic SVG elements like <rect>, <circle>, and <text> elements. They do not require any special data to be bound on them and are simply rendered by setting their attributes to the desired values. The layer above primitive elements is formed by composite components that only contain primitive elements. Composite components are usually rendered as <svg> or <g> elements. This layer includes components such as axes, legends, and series and is the lowest layer that is configured using structured data bound on the composite elements. Series components are composite elements that render a series of underlying elements using a data join and the general update pattern. The next higher layer consists of chart components. Charts are composite components that can also include other composite components. In many other visualization libraries, charts are the highest-level component. They are the visual entity that represents a complete visualization and are usually composed of axes, series, and legend components. RespVis contains an additional layer of components above charts. This highest layer is formed by chart window components. These are also composite components, but unlike all previously discussed layers, they are not rendered as SVG elements but as HTML elements. Their purpose is to nest charts into a layouter component, render them in a two-phase rendering process, and provide a toolbar for them. Toolbars are customizable and will hold different tools for different types of charts. The hierarchy of component layers in the RespVis library can be seen in Figure 5.1.

5.2 Naming Conventions

The naming of entities in RespVis follows the same naming conventions that are used in D3 modules. In D3-related modules, it is common that names of entities start with the name of the group an entity belongs

to. These names are then further narrowed down by successively adding more words until the exact entity is described accurately. This convention is not explicitly named, but it is referred to as "top-down naming" in this work. An example of the top-down naming convention can be seen in the `d3-scale` [**D3Scale**] and `d3-axis` [**D3Axis**] 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 the natural english language, using such names can feel odd for the uninitiated. However, the experience of working with APIs following such a naming convention is superior to when they do not. The reason for this is that 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. Due to this superiority and to stay consistent with other D3 modules, the decision has been made that entity names in RespVis should also follow this convention.

The public interface of the library is mostly made up of types and functions. Types are usually written as interfaces and represent the shape of an object. Their names are written in `PascalCase` and adhere to the top-down naming convention. They always start with the group a type belongs to and further words are successively added to distinguish gradually more specialized types. The naming of types can best be demostrated by the different names given to different kinds of bar charts. Normal bar charts are called `ChartBar`, grouped bar charts are called `ChartBarGrouped`, and stacked bar charts are called `ChartBarStacked`.

The API of RespVis is mainly composed of functions. Function names are always written in `camelCase` and also follow the top-down naming convention. They 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, an element to which the data object is bound, and a render function that transforms the data into some form of visual representation. The names of functions that create component data objects are always in the form of `componentNameData`, such as `chartBarData` or `chartBarGroupedData`. Functions that transform bound data into the visual representation of a component are always named in the form of `componentNameRender`, such as `chartBarRender` or `chartBarGroupedRender`.

5.3 Project Setup

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

5.3.1 Directory Structure

The goal of this section is to give an overview over the directory structure of the RespVis project. Roughly summarized, the project contains configuration files for various tools, a `src/` directory containing the source code for the whole library and accompanying examples, a `node_modules` directory containing the

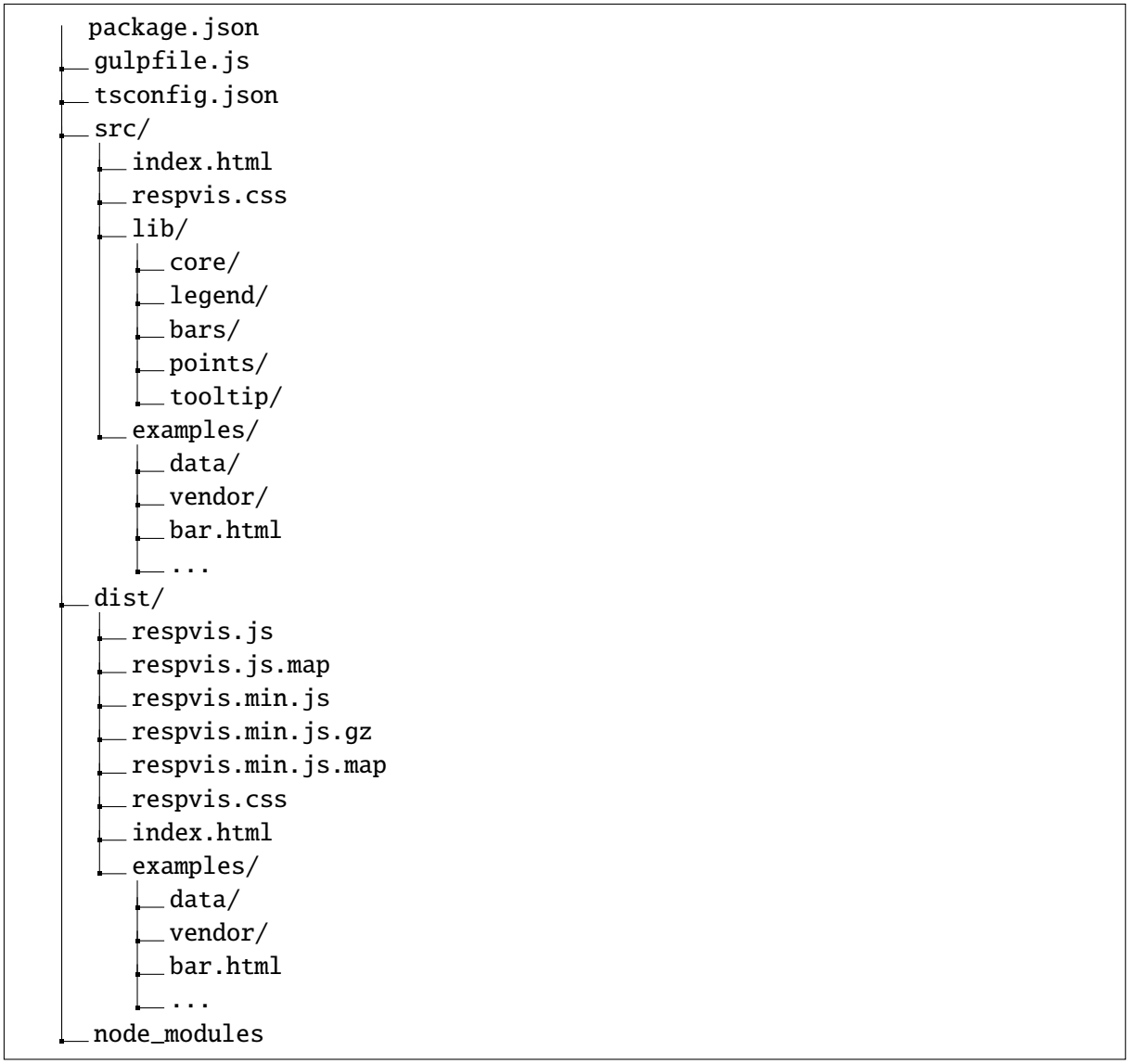


Figure 5.2: The directory structure of RespVis project. Only important files are shown here for readability reasons. [Figure created by the author of this thesis.]

project's cached NodeJS dependencies, and a `dist/` directory containing built versions of the library and examples ready for distribution. The configuration files are only discussed broadly here, as later sections go into more details about the setup of the various tools. A tree visualization of the whole directory structure, including all important files and directories but excluding individual source files, can be seen in Figure 5.2.

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

The `src/` directory at the root of the project contains all the implementation files of the library in the

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

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

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

The code of RespVis is distributed as JavaScript bundles of different formats that can be used depending on the situation. 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 5.3.3. Bundles containing the `.js` extension in their file name contain IIFE source code, whereas bundles containing the `.mjs` extension contain ES module source code. These bundles are also distributed in gradually more 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 [**GZIP**]. Beside these code bundles, the Rollup module bundler has been configured to create 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`. These source maps can be interpreted by developer tools in browsers to map from certain instructions in the bundled JavaScript code to the exact instruction in the original TypeScript code. They are an immense help when developing the library because, without them, debugging in browsers would be virtually impossible. Since RespVis aims to perform all possible styling in CSS, the distribution also contains a `dist/respvis.css` file which contains all the default styles of visualizations created with RespVis. Currently, this file is written manually as a whole in the `src/` directory and merely copied to the `dist/` directory during the build process. In the future, this process should be improved by employing a CSS preprocessing tool such as SASS [**SASS**] so that the CSS can be split into multiple files during development. Beside the bundled library source code and stylesheets, the `dist/` directory also contains usage examples of the library within the `dist/examples/` directory. This directory is identical to the one under `src/examples/` because it is merely being copied to the `dist/` folder during the build process.

5.3.2 NodeJS

NodeJS is a standalone JavaScript runtime built on top of the V8 JavaScript engine [**V8**]. It is an open-source and multi-platform runtime that enables the execution of JavaScript code 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. It can even be set up as a very powerful framework to develop client-side applications, like it has been done in the RespVis library. One of the most important tools in the NodeJS environment is the npm package manager [**npm**]. It was created in 2009 to simplify the sharing of source code modules and the management of the dependencies 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 ones.

RespVis is developed as a npm package. Every npm package is configured via a `package.json` file. This file contains all the necessary meta-data of a package to make it identifiable and provide enough information about what the package contains. It also lists all the dependencies of a package, so they easily be updated and downloaded during the installation process. A package can include normal dependencies and development dependencies. The difference between those two types of dependencies is that normal dependencies of a package will always be installed alongside of it, whereas the development dependencies will only be installed when installing a local package. The `package.json` file that is located in the root directory of the RespVis package can be seen in Listing 5.1.

5.3.3 Rollup

[TODO: Mention tree shaking]

The Rollup module bundler is used to bundle the source code of the RespVis library. Bundling is used to combine code that is written as multiple smaller modules to make it easier to distribute. It also takes care of the necessary transformations to bundle code in different formats, without developers having to worry about the details of how their code will be packaged. All developers have to do is to write their code once in a form that can be parsed by Rollup, and it will perform the necessary actions.

Rollup supports creating bundles in most common module formats like 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. IIFEs have already existed 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 that 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 it has been created. IIFE modules are plain JavaScript and do not require any modern features to be supported by browser. They are simply loaded in web documents like any other JavaScript resource via a `<script>` element. The example in Listing 5.2 was created to demonstrate the IIFE module format.

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

The core package of Rollup is only able to create mostly unmodified bundles from JavaScript source files. Various plugins exist that add frequently-required additional functionality. There are two kinds of

```
{
  "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-array": "^2.12.1",
    "d3-axis": "^2.1.0",
    "d3-brush": "^2.1.0",
    "d3-ease": "^2.0.0",
    "d3-format": "^3.0.1",
    "d3-scale": "^3.3.0",
    "d3-selection": "^2.0.0",
    "d3-transition": "^2.0.0",
    "d3-zoom": "^2.0.0",
    "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 to describe the package and it's dependencies. Keywords and type dependencies have been omitted for readability reasons.

```
// 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 into a function that gets executed immediately after declaring it and returns the public interface of the module. Comments were added to show in which files the individual pieces of code reside. `do-something.js` contains the original code that should be wrapped into an IIFE module, `module.js` contains the code of the IIFE module, and `application.js` demonstrates the usage of the module.

Rollup plugins: bundle plugins, which affect the bundling process, and output plugins, which transform the already bundled code.

The bundle plugins that are used for the bundling of RespVis are the `@rollup/plugin-node-resolve`, `@rollup/plugin-commonjs`, and `@rollup/plugin-typescript` plugins. 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 has been added to interpret them. Lastly, the `@rollup/plugin-typescript` plugin is used to compile TypeScript source files to JavaScript before bundling them. The configuration with which the TypeScript compiler is invoked is taken from the `tsconfig.json` file at the root directory of the project.

The output plugins used during the bundling process are the `rollup-plugin-terser` and `rollup-plugin-gzip` plugins. These plugins do not affect every created bundle. Instead, they are used to selectively transform the contents of specific bundles. The `rollup-plugin-terser` plugin is used to minify the code of created bundles containing the `.min` extension in their file names. Logically, they are equivalent to non-minified bundles, but they are compressed as much as possible to reduce their file size while still containing valid, although unreadable, JavaScript code. Another output plugin used for even stronger minification is the `rollup-plugin-gzip` plugin. This plugin has been applied to all bundles containing the `.gz` extension in their file names. It performs another step of compression on bundles that have already been minified using the `rollup-plugin-terser` plugin, by packing them in the GZIP compression format [**GZIP**].

Another noteworthy thing is that D3 is not included in any of the generated bundles. The reason for this is that RespVis is designed to be an extension of D3 and, most of the time, an application that wishes to use RespVis will already by relying on it. If D3 were to be included in the bundle of RespVis, it would unnecessarily be loaded a second time. This redundancy would cause a needless increase in the

bundle size and loading time of the library. To prevent D3 from being included in the created bundles, all dependencies from 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 5.3.4. The code of the `bundleJS` task can be seen in Listing 5.3. As can be observed in the code, the Rollup API allows the library to be bundled once via the `Rollup.rollup` function that returns the created bundle. This bundle can then be written to the target destination via the `Bundle.write` method, which allows the specification of the target bundle format and the plugins used to transform the code before writing it.

5.3.4 Gulp

[TODO: Add figure of npx gulp –tasks]

Gulp is a task runner that automates workflow processes via a set of named tasks. It is used to automate processes like building the library and serving the examples on a development server. Tasks perform a certain operation that needs to be carried out recurringly. They can perform an atomic operation or represent a composition of other tasks. These composite tasks can execute tasks contained in them in a serial or parallel order. Individual tasks are implemented as JavaScript functions in the Gulp configuration file, `gulpfile.js`, which can be found in the root directory of the project. This approach of favoring code over declarative configuration files means that the person setting up process automation needs to be familiar with JavaScript to do so, in return the possibilities of configuration are endless.

Tasks in the `gulpfile.js` file have been separated into private and public tasks. Private tasks are simply asynchronous functions that perform a certain action that does not necessarily have to be executed by external entities. The private Gulp tasks set up in the RespVis project are the `bundleJS`, `bundleCSS`, `copyExamples`, `cleanDist`, `cleanNodeModules`, and `reloadBrowser` tasks. 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 set up in this project are compositions of other tasks or references to them. The public tasks that are available are the `clean`, `cleanAll`, `build`, and `serve` tasks. The default task that is being executed when no other task is specified via the command-line interface is the `serve` task.

Bundling of the library's source code is implemented in the private `bundleJS` task. It uses the JavaScript API of Rollup to bundle all TypeScript files into IIFE and ES modules of varying levels of minification. This task has already been described in detail in Section 5.3.3, so it wont 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 written as a whole in the `src/` directory and merely copied to the `dist/` directory, but there are plans to build this file from different modules using a CSS preprocessor in the future. Usage of a CSS preprocessor will require an additional bundling step to be performed that compiles individual CSS modules into a merged CSS bundle that can be interpreted by browsers. 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 being developed inside the `src/` directory and they need to be made available as library distributables. Another reason for the copying is that the BrowserSync development server is initialized with the `dist/` directory as its root and every file that should be viewable on this server must reside somewhere in there. The `copyExamples` task is executed during the public `build` and `serve` tasks.

The private `cleanDist` and `cleanNodeModules` tasks are used to respectively delete the `dist/` and `node_modules/` directories. The `cleanDist` task is exported under a different name as the public `clean`

```
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 that bundles the code of the RespVis libary. Bundling is performed once using the `Rollup.rollup` function. After the library has been bundled, it is written multiple times with different configurations using the `Bundle.write` method.

task. This task is necessary because without cleaning the `dist/` directory before every rebuild, files from previous builds that might have disappeared in the meantime would cause littering and confusion. Therefore, this task is being executed as the first step of the `build` task. The public `cleanAll` task is composed of the private `cleanDist` and `cleanNodeModules` tasks. It is only executed manually when developers choose to delete the currently cached dependencies of the project to reinstall them from scratch.

The public `build` task is responsible for building all parts of the project. It is a composite task that 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 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. It has many features like simulated network throttling, interaction synchronization, and file synchronization that enable simultaneuous testing in multiple environments. In the setup of 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 that 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 that is 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 being changed, the `copyExamples` task is executed and the browser is reloaded.

Chapter 6

Modules

The source code of RespVis is structured into modules written in the ES module format. Currently, all these modules are combined into a single, monolithic library bundle during the build process. In the future, each module will be released on its own to allow users to import only the ones they need. The reason for this is that most users will likely only require a subset of all the features included in the library and it would unnecessarily increase the size of their bundles to import all of them. A good example of this is D3, which also separates its considerable amount of features into different modules that can be successively added to a project when the need arises.

At the time of writing, the RespVis library contains five different modules: the core, legend, tooltip, bar, and point modules. Each of these modules contains submodules that have been grouped by thematic similarity. The core module holds the core functionality of the library that all other modules depend on, which includes the layouter, axes, chart and chart window base components, and various utility functions and types. The legend module contains the implementation of a legend component that is mostly meant to describe discrete data dimensions by rendering distinct values as labeled symbols. The tooltip module holds functions to control the showing, placement, and content of tooltips, as well as utility functions that simplify the configuration and initialization of tooltips on series components. The bar module distinguishes between normal, 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.

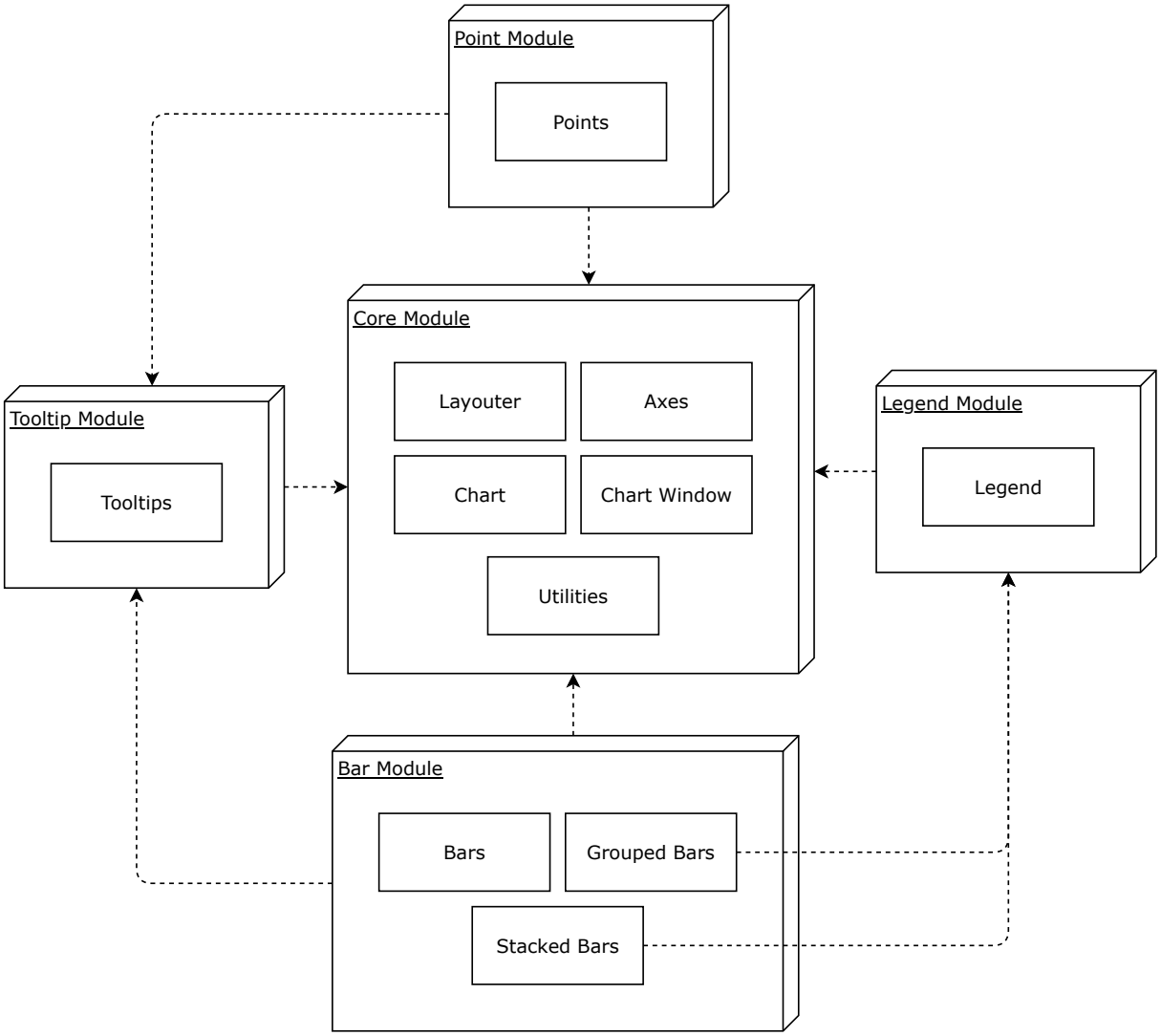


Figure 6.1: This diagram shows the different modules of the RespVis library. It also shows the most important submodules contained in the individual modules. The directional arrows connecting modules indicate dependencies between them. [Image created by the author of this thesis using diagrams.net.]

6.1 Core Module

The core module contains the necessary core functionality of the library. It is the base module that all other modules depend on and includes various utility functions, the layouter, axes, chart base components, and chart window base components. RespVis heavily relies on utility functions to reuse and structure recurring operations. The core module contains utilities to deal with arrays, elements, selections, and texts, as well as geometric utilities that simplify the handling of positions, sizes, rectangles, circles, and paths. The layouter is a custom component that enables controlling the layout of SVG elements with CSS. Axis components have been included in the core module because they are important components that occur in nearly every visualization. Lastly, the chart and chart window components provide base functionalities that simplify the creation of more specialized charts and chart windows. The core module implementation is located in the `src/lib/core/` directory of the project.

6.1.1 Utilities

The utilities provided by RespVis are split on multiple modules that are placed in the `utilities/` directory of the core module. These modules include types and functions that perform array, element, selection and

text operations as well as modules that simplify geometric operations with positions, sizes, rectangles, circles and paths. Utility functions are grouped into modules by the type of entity they operate on. This grouping is also reflected in the names of functions, which all begin with the type of entity a function is associated with.

[TODO: Capitalize "core module", "array utility module", ... ?] Array utilities can be found in the `utilities/array.ts` module. 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 that allows handling of a broad range of situations. However, not everything is covered out of the box and some things require manual implementations, which is why the RespVis library offers additional functions that simplify commonly encountered tasks. The `arrayEquals` function is used to verify equality of two arrays and also works if they contain nested arrays within them. Type guard functions are used to determine the type of a variable at runtime. For this purpose, two different array type guard functions are provided in the array utility module: `arrayIs`, which evaluates to true if the passed parameter is an array, and `arrayIs2D`, which evaluates to true if the passed parameter is a two-dimensional array. The `arrayIs` function is merely a wrapper around the `Array.isArray` method. Theoretically, the `Array.isArray` method could be used directly instead of the `arrayIs` function, but because the `arrayIs2D` function is required, the `arrayIs` function has also been added for consistency reasons. The last function in the array utility module is the `arrayPartition` function. This function 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 utilities module located at `utilities/elements.ts` in the core module contains functions and constants related to document elements. 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 an element's position, it also takes transformations into account. Every element has a set of CSS styles applied on them and the `Window.getComputedStyle` method can be used to access the active style of an element. 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 used during the preparation of a downloadable SVG document to transform styles set in CSS to attributes on the individual elements. If every possible style property on every element would be mapped to an attribute, the resulting SVG document would be unnecessarily bloated 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 properties that are set to default values 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 speed up 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 all names of the presentation attributes listed in the SVG 1.1 specification [Dahlström et al. 2011]. Since only these SVG attributes can be styled via CSS, only these properties need to be taken into account when preparing downloadable SVG documents.

[TODO: When to write `Selection` and when to write Selection? What about plural? `Selections` or `Selection`s?] Selection utilities are implemented in the `utilities/selection.ts` module. They include typing improvements for the D3 `Selection`, `Transition`, and `SelectionOrTransition` interfaces and type guards to distinguish between `Selections` and `Transitions`. The `Selection`, `Transition`, and `SelectionOrTransition` interfaces allow the specification of four different type variables: the type of elements contained in the `Selection` or `Transition`, the type of data that is bound to those elements, the type of the parents of those elements, and the type of data that is 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 interface have been reexported with default types set on all of the type variables. This 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. This is wrong because this method will actually result in a `null` value when reading an attribute that does not exist. To fix this inconsistency and catch potential bugs related to this during compilation, the type declaration of the `Selection.attr` method has been overwritten in the Selection utility module to also include `null` as a possible return value. A less important but nonetheless convenient improvement has been made to the type declaration of the `Selection.dispatch` method. This method allows the dispatching of custom events with certain parameters that control different aspects of how this event is dispatched and the data that may be 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. However, this is not reflected in the type declaration of the function, which requires every parameter to be provided 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` 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 `utilities/text.ts` module. It contains rather basic functionalities that simply 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. These functions respectively set the `data-align-h` and `data-align-v` attribute on a Selection or Transition 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 the `respvis.css` file to declare different `text-anchor` and `dominant-baseline` values that control the alignment of `<text>` elements. Text orientation is set using the `textOrientation` function. This function sets the `data-orientation` attribute on a Selection or Transition to the value specified via the string enum parameter of type `Orientation`. 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 a `<text>` element, in order to rotate it accordingly and position it correctly inside the bounding box calculated by the layouter.

The core module also contains utilities that simplify geometric operations. One of these utilities is the position utility module located at `utilities/position.ts`. This module contains the `Position` interface and various functions to perform operations related to it. The `Position` interface consists of the `x` and `y` number members. Rounding these members is necessary to be able to correctly compare equality of two `Positions` 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 members variables should be rounded to. Equality comparision between two `Position` variables can be performed with the `positionEquals` function, which evaluates to `true` if both `Positions` are equal and `false` if not. To transform a `Position` into its string representation of the form `"x, y"`, the `positionToString` function can be used. Its counterpart, the `positionFromString` function, can be used to transform a string in the correct format into a `Position`. 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 a `SelectionOrTransition` to the values of the `x` and `y` members of a `Position`. Similarly, the `positionToTransformAttr` function can be used to set the

transform attribute of a SelectionOrTransition to a translation representing a Position. The position utility module also contains the positionFromAttrs function, which can be used to create a Position from the x and y attributes of a SelectionOrTransition.

The size utility module is located at utilities/size.ts 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 member variables. The sizeRound function is used to round the member variables of a Size object to a certain number of decimals. To compare two Size objects for equality, the sizeEquals function can be used. 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. Moreover, the sizeToAttrs can be used to set the width and height attributes of a SelectionOrTransition to the values of a Position object and the sizeFromAttrs function can be used to create a new Position object from the values of these attributes.

Utilities for dealing with rectangles can be found in the rectangle utility module, which is located under utilities/rect.ts in 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 number member variables x, y, width, and height. 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. 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 is used in transitions that grow or shrink a <rect> element from or to their center. It creates a minimized version of the passed Rect, which is infinitely small and positioned at the original Rect's center. When declaring a stroke for SVG elements, it is drawn exactly on the outline of an element's silhouette. This means that a stroke will extend outside the original bounds of an element by half the stroke width, which can lead to unwanted artefacts like the stroke of bars in a bar chart overlapping over the chart's axes. To counteract this, the rectFitStroke function is provided to adjust the bounding box of Rect objects to account for a stroke of a certain width around them. Lastly, the rectangle utility module provides functions to calculate specific positions inside of rectangles. The most generic of these functions is the rectPosition functions. This function enables the calculation of a position inside of a rectangle via a two-dimensional parameter that expresses a position as the percental 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 a Rect object's edges. Similarly, The rectTopLeft, rectTopRight, rectBottomRight, rectBottomLeft functions can be used to calculate the corner positions of a rectangle.

The last geometric primitive whose handling is simplified by a RespVis utility module is a circle. The circle utility module can be found at utilities/circle.ts in the core module. It contains the Circle interface, which describes a circle object as a center property of type Position and a radius property of type number. 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 using an angle that defines the direction and an optional parameter that defines the distance from the circle's center as a percentage of the circle's radius. The circle utility module also contains functions to create circles from rectangles. These functions 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 purpose of the path utility module is to provide functions that simplify the creation of path

definitions that can be set on <path> elements. It is located at utilities/path.ts in the core module and only contains a small number of functions. The pathRect function creates a path definition that represents a rectangle. A <path> element with a path definition representing a rectangle can be used instead of a <rect> element. Similarly, the pathCircle function creates a path definition that represents a circle. Such a path definition is used on a <path> element to render a circle instead of using a <circle> element. The reason for using <path> elements rather than more descriptive shape elements is that path elements can change their shape dynamically without replacing elements. Since only the d attribute of a path needs to change when the path's shape is altered, it is also possible to smoothly transition between shapes by interpolating the path definition strings.

6.1.2 Layouter

The Layouter is the most novel contribution of this work. It is a component that is wrapped around an SVG document and allows to configure the layout of elements in this document with CSS. Instead of implementing a custom layout algorithm, the Layouter builds on layout engines integrated in browsers, which have already been summarized in Section 2.6.1. Earlier proof of concept implementations used the FaberJS [FusionCharts 2021] and Yoga [Facebook 2021d] layout engines to calculate layouts, but these implementations were not further pursued because they limited layouting to either Grid or Flexbox-based constraints. Furthermore, the use of already existing 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 the presentation of elements and the possibility of defining different presentations for different contexts via media queries. A large part of the responsive power that CSS offers comes from its ability to change the positioning and layout of elements. As already mentioned in previous chapters, CSS can be used to 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] that allow the use of CSS to style visualizations, none of them offer the possibility to modify the layout of visualizations via CSS. This means that visualization authors have to learn and use custom APIs to position elements, which limits the range of possible layouts to those supported by the individual libraries.

The Layouter distinguishes between laid-out and non-laid-out elements because not every element in a visualization profits from being laid out by it. Laid-out elements are elements whose position and size are being calculated by the Layouter, whereas non-laid-out elements are ignored during the layout process. Theoretically, the layouter could be used to position all elements of a visualization since it is only necessary to determine a good mapping that maps a rectangular bounding box to the desired SVG shape for each element. However, the positioning of elements in a visualization is constrained more strictly than the positioning of elements in typical HTML documents. The content of a visualization is communicated more through visual features such as position, size, and shape 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. These 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. Positioning them via the Layouter would be rather pointless and only cause unnecessary overhead.

The layout process can be seen in Figure 6.2 and consists of three phases that have been implemented in the layouterCompute function:

1. Replication: The structure of the SVG document that shall be laid out must be replicated with HTML elements. These elements are referred to as "layout elements" and they have the same classes

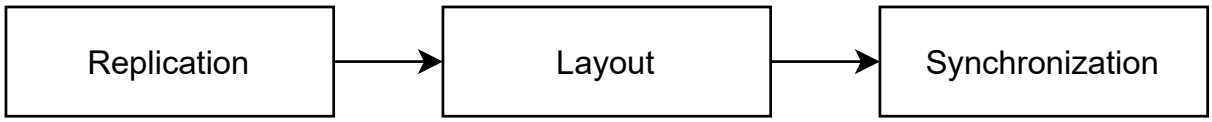


Figure 6.2: This diagram shows the three phases of the layout process of the RespVis Layouter. During the replication phase, the SVG document that shall be laid out is replicated with HTML `<div>` elements. Afterward, these HTML elements are laid out by the browser in the layout phase, and the positions 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.]

and `data-*` attributes as the original SVG elements they are replicating. This replication of the SVG document with HTML documents is necessary because CSS-based positioning can only affect HTML elements.

2. Layout: The replicated layout elements are affected by CSS rules that 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 they will also be applied to the corresponding layout elements.

3. Synchronization: The positions of the layout elements are synchronized with their respective SVG elements. This means that the calculated bounding boxes of layout elements are set as `bounds` attributes on SVG elements, which can be used in subsequent renderings. In addition to that, 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. This replication was implemented using a hierarchical D3 data join in which the original SVG elements are bound as data objects to layout elements. This 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 know which to ignore. For this, the `data-ignore-layout` and `data-ignore-layout-children` attributes have been introduced. Elements that have the `data-ignore-layout` attribute or are children of elements that have the `data-ignore-layout-children` attribute will not be replicated.

To configure the layout of 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 the original SVG element 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 corresponding layout elements. In addition to the classes of the replicated SVG element, the layout class is set on all layout elements. By doing this, it is possible to specifically select the layout element of an SVG element via the same Selector by adding the layout class to the same Selector. If CSS rules of SVG elements use only classes and data attributes in their Selectors, the properties of corresponding layout elements can directly be configured in the same rules. An example of how the replicated layout element tree of an SVG document looks can be seen in Listing 6.1. Furthermore, an example of CSS rules that 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. Since layout elements exist separately from their SVG elements and can not access their content, a manual solution had to be implemented that sets the size of layout elements to the content size of their SVG elements when required. An example of dynamically-sized elements is `<text>` elements because their size is rarely declared in absolute units and usually depends on the size of their text contents. 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 being checked during the replication phase via the `window.getComputedStyle` method. If at least one of these properties is set

```
<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.

to true, the dimensions of the SVG element are calculated with the `Element.getBoundingClientRect` method and respectively set as `width` or `height` properties of the `style` attribute on the corresponding layout element. By doing this, layout elements will have the same size as their svg elements and can be properly used in the calculation of the overall layout.

Layout elements are automatically laid out by the browser during the layout phase of the layout process. Since the layout elements are simply `<div>` elements that have been styled in CSS rules, the browser can position them automatically via its integrated layout engine. This positioning happens as soon as the layout elements have been rendered. After this, 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 iterates over all the layout elements, calculates their bounding boxes, and set 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. This bounding box is then converted to its string representation via the `rectToString` utility function and set as the `bounds` attribute on the corresponding SVG elements. [TODO: Rename bounds attr to data-bounds] The `bounds` attribute can then be deserialized to a `Rect` object whenever the bounding box of an SVG element is needed in subsequent renderings of SVG components. In addition to setting the `bounds` attribute of SVG elements, the Layouter also sets specific attributes on different types of SVG elements that make them fit into their calculated bounding boxes. These attributes can, of course, be overwritten in later renderings, but they represent sensible defaults that express the boundaries of laid-out elements. If the Layouter would not set these default attributes, they would have to be set manually on every laid-out element during the rendering process, which would be less convenient and lead to duplicated code in various places. For SVG elements that can be mapped directly to rectangular areas, such as `<svg>` and `<rect>` elements, the Layouter sets the `x`, `y`, `width`, and `height` attributes to the values of their 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 make them fit into their boundaries. Other SVG elements that are not explicitly sized, such as `<g>` and `<text>` elements, are merely moved to the correct position by setting their `transform` attribute to a translation so that their top-left corners aligns with the top-left corners of their bounding boxes. The Layouter does not automatically recalculate the dimensions of exactly-positioned elements based on the changed dimensions of the composite `<svg>` or `<g>` element containing them. This has to be manually implemented in the render functions of the 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 all elements are known, the visualization needs to be rendered in its final form. This visualization rendering process when using the Layouter consists of three phases and is 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 that affect the layout of a visualization need to be rendered. This mostly includes laid-out container `<svg>` and `<g>` elements that contain exactly-positioned child elements, but it also means that the contents of dynamically-sized elements, such as `<text>` elements and axes, need to be fully rendered in this phase too. The layouting phase is where all the operations of the already-described layout process are being performed. During this phase, the bounding boxes of laid-out elements are calculated and persisted as attributes that can be accessed during the second rendering phase. In the second rendering phase, the desired bounding box position and size of every element is known and can be used to perform a second rendering of the complete visualization. Here, every element affected by the layout, which is every element, is rendered at its final position with its final dimensions. In theory, the first and second rendering phases of components could be implemented in separate functions. However, 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 6.3: This diagram shows 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 as attributes on the SVG elements. During the second render phase, all elements of the visualization are rendered at their final positions with their final dimensions by using the boundary information calculated in the layout phase. [Image created by the author of this thesis using diagrams.net.]

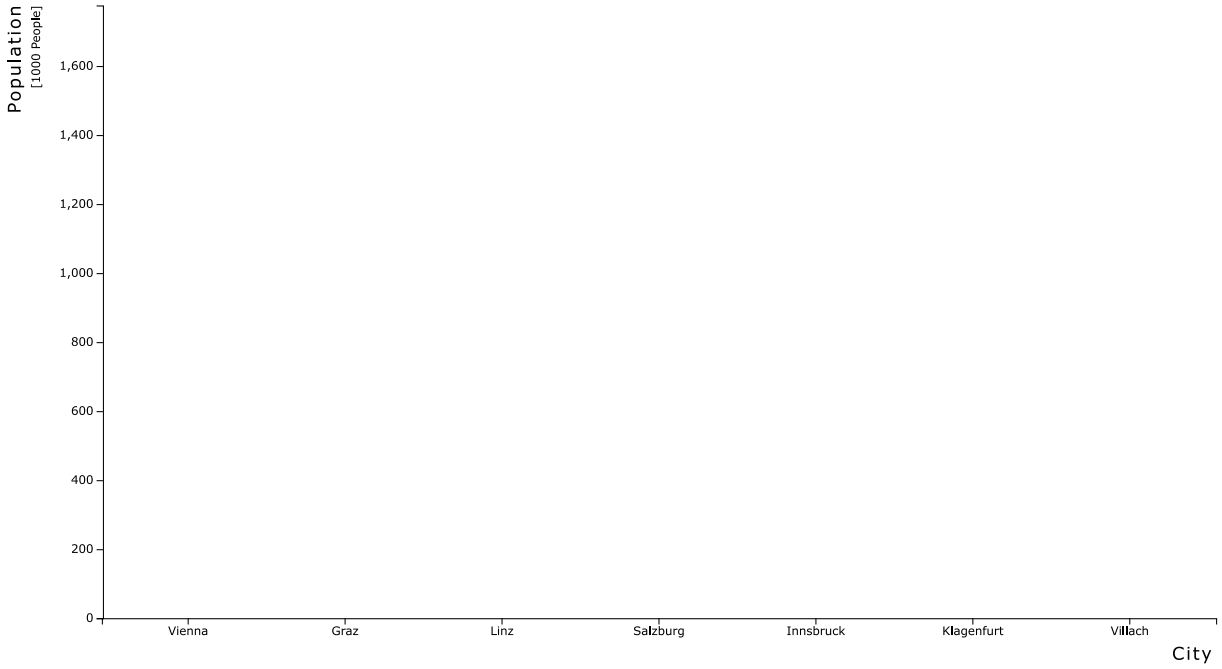


Figure 6.4: This figure shows how a rendered left and bottom axis may look like. The left axis consists of a title, subtitle, and ticks, whereas the bottom axis only consists of ticks and a title. [Image created by the author of this thesis using RespVis and Inkscape.]

6.1.3 Axes

achsen werden verwendet um scales, welche das mapping von abstrakten werten auf zwei-dimensionale positionen representieren, zu visualisieren. Die implementierung aller verfuegbaren Achsen komponenten kann in der `axis.ts` datei im core module gefunden werden. momentan werden von der RespVis library nur kartesische achsen zur verfuegung gestellt da bislang nur kartesische charts implementiert wurden. diese kartesischen achsen komponenten werden per ihrer position relativ zu der draw area einer visualisierung unterschieden und zum aktuellen zeitpunkt wurden nur left and bottom axis components in der RespVis library implementiert. Dies sind die mit abstand am haeufigsten angetroffenen positionierungen von kartesischen achsen und wurden gewaehlt, da mit ihnen die meisten use cases abgedeckt werden koennen. Eine achse besteht aus ticks, welche die eigentliche visualisierung einer scale sind, und aus einem optionalen titel und untertitel, welche gerendert werden koennen um die konfiguration der visualisierten scale zusaetzlich zu beschreiben. Ein Beispiel dafuer wie eine gerenderte left und bottom axis aussehen koennte, kann in Figure 6.4 gesehen werden.

Das `Axis` interface beschreibt die shape eines data objects mit welchem das rendering einer achse konfiguriert werden kann. Es beinhaltet ein `scale` property, welches die scale representiert die visualisiert werden soll, die `title` und `subtitle` string properties, und das `configureAxis` function property, welches verwendet werden kann um die darunterliegende D3 axis zu konfigurieren bevor sie rendered wird. wie

auch die meisten anderen komponenten bestehen auch achsen komponenten hauptsaechlich aus zwei funktionen: einer data creation funktion und einer render funktion. die `axisData` funktion wird verwendet um ein data object in der form des `Axis` interfaces zu erzeugen. sie wird mit einem `Partial<Axis>` object als parameter aufgerufen und liefert ein neues object bei dem alle nicht-gesetzten aber benoetigten properties des parameter objektes mit standardwerten befuellt wurden. die `axisBottomRender` und `axisLeftRender` funktionen werden verwendet um eine left oder bottom axis in einem composite element zu rendern auf welchem eine achse mittels eines gebundenen `Axis` data object konfiguriert wurde. diese beiden render functions verwenden die nicht-exportierte `axisRender` base function um duplizierten code zu vermeiden. alle achsen haben die selbe struktur von elementen und werden so viel wie moeglich ueber CSS positioniert und gestyled. Der root einer achse ist ein CSS Grid container und definiert das layout der title, subtitle und ticks elemente. Die standardkonfiguration einer left axis positioniert diese elemente in einem three-column layout in welchem die title, subtitle und ticks elemente in dieser reihenfolge von links nach rechts plaziert werden. Bei einer bottom axis sieht die standardkonfiguration eine positionierung derselben elemente in einem three-row layout vor in welchem die ticks, title und subtitle elemente in dieser reihenfolge von oben nach unten plaziert werden. Weiters werden die title und subtitle elemente einer left axis mittels der `textOrientation` utility function vertikal orientiert um horizontalen platz zu sparen. Die RespVis achsen komponenten verwenden intern jeweils die `axisBottom` und `axisLeft` funktionen aus dem D3 axis module [**D3Axis**] um die ticks einer achse zu rendern. da diese D3 funktionen attributes verwenden um die einzelnen elemente der ticks einer achse zu positionieren und zu stylen, muessen diese attributes direkt nachdem die ticks gerendert wurden soweit es geht entfernt werden um die konfiguration via CSS zu ermoeglichen.

6.1.4 Chart

charts sind higher-level components die eine vollstaendige visualisierung inklusive achsen, legenden und series representieren. Ein chart wird typischerweise in dem root `<svg>` element eines SVG dokumentes gerendert welches zumindest die `chart` klasse im `class` attribute und den angemessenen SVG namespace im `xmlns` attribute gesetzt hat. Diese attribute koennen in spezifischeren chart komponenten entweder manuel oder ueber die `chartRender` funktion aus der `chart.ts` datei im core module, welche nur diese attribute setzt, gesetzt werden.

wie bereits erwaehnt beinhaltet die RespVis library aktuell nur die implementationen von kartesischen charts welche daten in einem kartesischen koordinatensystems visualisieren. die implementierung der basis funktionen von kartesischen charts befindet sich in der `chart-cartesian.ts` datei im core module. das `ChartCartesian` interface beschreibt ein data object fuer die konfiguration von kartesischen charts. diese data objects beinhalten die `xAxis` und `yAxis` `Axis` properties mit welchen jeweils die X und Y achsen des charts beschrieben werden. das transponieren von achsen ist ein nuetzliches pattern um die responsiveness von visualisierungen zu verbessern und wird mittels des `flipped` boolean properties konfiguriert. wenn das `flipped` property auf `false` gesetzt ist, wird das `xAxis` object fuer die konfiguration der bottom axis und das `yAxis` object fuer die konfiguration der left axis verwendet. wenn es auf `true` gesetzt ist, ist es genau umgekehrt.

die `chartCartesianData` funktion wird verwendet um ein data object in der form des `ChartCartesian` interface zu erzeugen. diese funktion erhaelt ein partielles data object bei welchem nur jene properties gesetzt sind welche fuer den aufrufenden code von interesse sind. alle nicht gesetzten properties werden mit standardwerten befuellt. die standardwerte der `xAxis` und `yAxis` properties werden ueber die `axisData` funktion aus dem axis submodule gesetzt. das `flipped` property wird, falls es nicht ueber den input parameter spezifiziert wird, auf `false` gesetzt.

das rendern von kartesischen charts ist auf zwei funktionen aufgeteilt die seperat voneinander aufgerufen werden muessen. diese aufteilung wurde gemacht weil nicht alle teile eines kartesischen charts zum selben zeitpunkt gerendert werden koennen. die generelle struktur eines charts muss gerendert werden bevor irgendetwas anderes gerendert werden kann da dies das draw area container element

beinhaltet in welches individuelle series componenten gerendert werden muessen. die achsen eines charts benoetigen eine vollstaendig initialisierte scale um korrekt gerendert werden zu koennen. die range einer scale, also der wertebereich in welchen abstrakte werte gemappt werden, haengt allerdings von den dimensionen der draw area boundary ab und wird erst waehrend der render funktion der individuellen series components, welche in die draw area gerendert werden, initialisiert. aus diesem grund duerfen achsen erst nach allen series components gerendert werden um eine vollstaendig initialisierte scale zu gewaehrleisten.

die struktur eines kartesischen charts wird mit der `chartCartesianRender` funktion gerendert. diese fuktion setzt die notwendigen chart attribute und die `chart` klasse mit der `chartRender` funktion. Weiters wird mit dieser funktion die `chart-cartesian` klasse auf den root elementen der Selection gesetzt mit welcher die funktion aufgerufen wurde. die `chartCartesianRender` funktion fuegt fuegt ausserdem noch ein `<svg>` element mit der `draw-area` klasse als child element des component root elementes ein. die draw area ist das container element in welches die individuellen series components eines charts gerendert werden. standardmaessig wird das `overflow` CSS property der draw area auf `visible` gesetzt um eventuell ueberlappende elemente nicht zu clippen. ein `<svg>` element ohne tatsaechlichen content ist nicht in der lage input events abzufangen. das bedeutet dass es zum beispiel nicht moeglich waere mittels scroll events eine zoom interaktion zu steuern wenn der cursor sich ueber der leeren flaeche der draw area befindet. um dem entgegenzuwirken erzeugt die `chartCartesianRender` funktion fuer jede draw area ein transparentes `<rect>` background element welches die gesamte flaeche der draw area fuellt und welches es ermoeglicht input events auch in bereichen zu generieren in welchen sich keine series elemente befinden.

die `chartCartesianAxesRender` funktion wird verwendet um die achsen eines kartesischen charts zu rendern. diese funktion darf nur auf elementen mit einem gebundenen `ChartCartesian` data object und nach der vollstaendigen initialisierung der scales, welche mit den achsen visualisiert werden sollen, aufgerufen werden. meist bedeutet dies fuer die reihenfolge an operationen einer spezifischerer kartesischen chart component, dass zu beginn die struktur des charts mittels der `chartCartesianRender` funktion erzeugt werden muss, gefolgt von dem rendern der gewuenschten series components, und erst danach duerfen die achsen mittels der `chartCartesianAxesRender` funktion gerendert werden. die `chartCartesianAxesRender` function erzeugt zwei `<g>` elemente und rendert eine left und eine bottom axis auf ihnen. je nachdem ob das `flipped` property im gebundenen data object auf `true` oder `false` gesetzt ist, wird das `xAxis` data object fuer die konfiguration der bottom oder left axis verwendet und das `yAxis` data object jeweils fuer die andere achse. nachdem die achsen gerendert wurden wird auf der achse auf welcher das `xAxis` data object gebunden ist die `x-axis` klasse und auf der anderen achse die `y-axis` klasse gesetzt. diese klassen werden gesetzt um die auswahl der konkreten achsen welche die jeweilige konfiguration representieren ueber CSS Selectors zu ermoeglichen.

wie es in der RespVis library ueblich ist, wird alles moegliche an positionierung und styling ueber CSS konfiguriert. die elemente eines kartesischen charts werden ueber ein CSS Grid layout positioniert. standardmaessig wird ein grid erzeugt welcher die `axis-left`, `axis-bottom`, codedraw-area, und `legend` areas definiert. die groesse der meisten zeilen und spalten dieses grids ist auf `auto` gesetzt, was bedeutet dass deren groesse von der groesse ihres inhaltes abhaengt. die ausnahme hiervon sind die zeile und spalte in welchen sich die draw area befindet. deren groesse ist auf `1fr` gesetzt um zu erreichen dass die draw area den gesamten restlichen horizontalen und vertikalen platz einnimmt der nicht von den anderen zeilen und spalten eingenommen wird. die standardkonfiguration der `chart-cartesian` klasse sieht eine positionierung einer eventuellen legende rechts von der draw area vor. um die position der legende zu veraendern kann entweder direkt der grid ueber CSS angepasst werden oder eine der vorkonfigurierten positionen kann ueber das `data-legend-position` attribut aktiviert werden. um das setzten des `data-legend-position` attributes zu vereinfachen kann die `chartLegendPosition` funktion, welche dieses attribut auf den wert eines mitgegebenen `LegendPosition` enum parameters setzt, verwendet werden.

Chapter 7

Examples

7.1 Bar Chart

7.2 Grouped Bar Chart

7.3 Stacked Bar Chart

7.4 Scatterplot

[TODO: Write about application example (newspaper article?)]

Chapter 8

Selected Details of the Implementation

8.1 D3 Select Function Data Modification

8.2 Save as SVG

Chapter 9

Outlook and Future Work

9.1 Outlook

9.2 Ideas for Future Work

9.2.1 Relative Positioning of Series Items

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

9.2.2 Container Queries

Chapter 10

Concluding Remarks

Appendix A

User Guide

Appendix B

Developer Guide

Bibliography

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

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

Amar, Robert, James Eagan, and John Stasko [2005]. *Low-level Components of Analytic Activity in Information Visualization*. 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 19).

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

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 33–34, 36–37, 39).

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

Andrews, Keith [2021]. *Information Visualisation - Course Notes*. Graz University of Technology, Austria. 01 Apr 2021. https://courses.isds.tugraz.at/ivis/ivis.pdf (cited on pages 17–18).

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 17).

Apple [2021]. *WebKit - A fast, open-source web browser engine*. 2021. https://webkit.org/ (cited on page 14).

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).

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

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

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

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 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 23).

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 23).

Bostock, Mike [2021]. *D3.js – Data-Driven Documents*. 2021. https://d3js.org/ (cited on page 23).

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. 01 Apr 2022. https://browsersync.io/ (cited on page 52).

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

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

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

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

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

Chatterjee, Sangit and Aykut Firat [2007]. *Generating Data with Identical Statistics but Dissimilar Graphics: A Follow up to the Anscombe Dataset*. The American Statistician 61.3 (Aug 2007), pages 248–254. doi:10.1198/000313007X220057. https://www.jstor.org/stable/pdf/27643902.pdf (cited on page 17).

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

Chromium [2021]. *Blink - Rendering Engine*. 2021. https://chromium.org/blink/ (cited on page 14).

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 19).

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, 55).

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).

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, 47).

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, 58).

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

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

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

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

Florent, Michael Florent van [1644]. *La Verdadera Longitud por Mar y Tierra*. 1644 (cited on pages 19–20).

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

Friendly, Michael [2008]. *A Brief History of Data Visualization*. In: *Handbook of Data Visualization*. Springer, 2008, pages 15–56. ISBN 3540330364. doi:10.1007/978-3-540-33037-0_2 (cited on pages 20, 22).

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 22).

FusionCharts [2021]. *FaberJS*. GitHub. 21 Oct 2021. https://github.com/fusioncharts/faberjs (cited on pages 14, 58).

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).

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 29, 58).

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

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

Hoffswell, Jane, Wilmot Li, and Zhicheng Liu [2020]. *Techniques for Flexible Responsive Visualization Design*. Proc. 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 33–34).

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

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

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://www.nytimes.com/interactive/2021/07/14/upshot/drug-overdose-deaths.html (cited on page 36).

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

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

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

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

Kunz, Gion [2021]. *Chartist.js*. 2021. http://gionkunz.github.io/chartist-js/ (cited on pages 29, 58).

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 29).

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://www.statista.com/statistics/793628/worldwide-developer-survey-most-used-languages/ (cited on page 7).

Macrobius, Ambrosius Theodosius [1175]. *Commentarii in somnium Scipionis*. De Gruyter, 1175. ISBN 3598715269. doi:10.1515/9783110951899 (cited on pages 19–20).

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

Marcotte, Ethan [2010]. *Responsive Web Design*. A List Apart. 25 May 2010. https://alistapart.com/article/responsive-web-design (cited on page 15).

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 22).

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://www.wsj.com/graphics/french-assembly-2017/ (cited on page 36).

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

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

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

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

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).

Observable [2021]. *Observable: Explore, analyze, and explain data. As a team.* 2021. https://observablehq.com/ (cited on page 23).

OpenJS [2021]. *Node.js*. OpenJS Foundation. 2021. https://nodejs.org (cited on pages 7, 44).

Playfair, William [1786]. *Commercial and Political Atlas: Representing, by Copper-Plate Charts, the Progress of the Commerce, Revenues, Expenditure, and Debts of England, during the Whole of the Eighteenth Century*. Cambridge University Press, 1786. ISBN 0521855543 (cited on page 19).

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 19).

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

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

Raggett, Dave [1997]. *HTML 3.2 Reference Specification*. W3C Recommendation. W3C, 14 Jan 1997. https://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 22).

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

Routley, Nick [2021]. *Internet Browser Market Share (1996–2019)*. https://www.visualcapitalist.com/internet-browser-market-share/ (cited on page 8).

Satyanarayan, Arvind and Jeffrey Heer [2014]. *Lyra: An Interactive Visualization Design Environment*. Computer Graphics Forum 33.3 (12 Jul 2014), pages 351–360. doi:10.1111/cgf.12391 (cited on page 26).

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 26).

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 25–26).

Scheiner, Christoph [1630]. *Rosa Ursina sive Sol ex Admirando Facularum & Macularum Suarum Phoenomeno Varius*. 1630 (cited on pages 19–20).

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

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

Shneiderman, Ben [1996]. *The Eyes Have It: A Task by Data Type Taxonomy for Information Visualizations*. 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 19).

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 14).

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 23).

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 29).

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

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

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

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

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

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

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

Totic, Aleks and Greg Whitworth [2020]. *Resize Observer*. W3C Working Draft. W3C, 11 Feb 2020. https://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 17).

Tufte, Edward Rolf [1983]. *The Visual Display of Quantitative Information*. 1st Edition. Graphics Press, 1983. ISBN 1930824130 (cited on pages 19–20).

Tufte, Edward Rolf [1997]. *Visual Explanations*. Graphics Press, 14 Jan 1997. ISBN 1930824157 (cited on page 20).

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 26).

Wattenberger, Amelia [2019]. *Fullstack D3 and Data Visualization*. Fullstack.io, 29 Jul 2019. ISBN 0991344650. https://newline.co/fullstack-d3 (cited on page 23).

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 19, 25).

Wongsuphasawat, Kanit, Dominik Moritz, Anushka Anand, Jock Mackinlay, Bill Howe, and Jeffrey Heer [2015]. *Voyager: Exploratory Analysis via Faceted Browsing of Visualization Recommendations*. IEEE Transactions on Visualization and Computer Graphics 22.1 (12 Aug 2015), pages 649–658. doi:10.1109/TVCG.2015.2467191. https://www.domoritz.de/papers/2015-Voyager-InfoVis.pdf (cited on page 26).

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 26).

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

Yi, Ji Soo, Youn ah Kang, John Stasko, and Julie A Jacko [2007]. *Toward a Deeper Understanding of the Role of Interaction in Information Visualization*. IEEE Transactions on Visualization and Computer Graphics 13.6 (05 Nov 2007), pages 1224–1231. doi:10.1109/TVCG.2007.70515. https://innovis.cpsc.ucalgary.ca/innovis/uploads/Courses/InformationVisualizationDetails2009/Yi.pdf (cited on page 19).