

**Rethinking RespVis:  
A Responsive Visualization Library with  
Modern CSS**

David Egger



# **Rethinking RespVis: A Responsive Visualization Library with Modern CSS**

David Egger B.Sc.

## **Master's Thesis**

to achieve the university degree of

Diplom-Ingenieur

Master's Degree Programme: Computer Science

submitted to

Graz University of Technology

Supervisor

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

Graz, 11 Dec 2024

© Copyright 2024 by David Egger, except as otherwise noted.

This work is placed under a Creative Commons Attribution 4.0 International (CC BY 4.0) licence.



# **Überarbeitung von RespVis: Eine Bibliothek für Responsive Visualisierungen mittels modernen CSS Methoden**

David Egger B.Sc.

## **Masterarbeit**

für den akademischen Grad

Diplom-Ingenieur

Masterstudium: Informatik

an der

Technischen Universität Graz

Begutachter

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

Graz, 11 Dec 2024

Diese Arbeit ist in englischer Sprache verfasst.

© Copyright 2024 David Egger, sofern nicht anders gekennzeichnet.

Diese Arbeit steht unter der Creative Commons Attribution 4.0 International (CC BY 4.0) Lizenz.



## Abstract

Data visualization presents data in a graphical way, so as to make it easier and more efficient to understand. Responsive web design promotes a set of techniques to make web pages *responsive*, i.e. able to adapt to the characteristics of the end user's device. Charts and visualizations embedded within a web page must themselves also become responsive. In particular, this means adapting to the available display space and to interaction modalities such as keyboard or touch.

RespVis is an open-source library for creating responsive visualizations. It is written in TypeScript with the popular visualization library D3, and is grouped into multiple packages which are publicly available on the npm registry. Chart creators with their own dataset and some experience in web development can use the RespVis API to create responsive bar charts, scatter plots, line charts, and parallel coordinates charts of their data. Chart developers with more experience in web development can customize the standard RespVis charts or create their own new chart types.

The RespVis API makes it easy to apply some common responsive patterns. Chart components such as title, axes, and legend can be positioned via powerful CSS layout techniques like Flexbox and Grid, even though RespVis visualizations are pure SVGs. This is made possible due to a custom layout engine working in the background. During this thesis work, dozens of updates and improvements were made to RespVis, resulting in the current version, RespVis v3.





## Kurzfassung

Data Visualization befasst sich mit der grafischen Darstellung von Daten, die so einfacher und effizienter interpretiert werden können. In Responsive Web Design gibt es empfehlenswerte Muster, um Websites *responsive* zu gestalten, d.h. Websites zu erstellen, die in der Lage sind, sich an die Charakteristiken eines Endgerätes anzupassen. Diagramme und Visualisierungen, die in einer Website eingebettet sind, müssen ebenfalls responsiv sein. Das bedeutet, dass sie sich an den verfügbaren Bildschirmplatz und die vorhandenen Bedienungsmöglichkeiten eines Endgerätes wie Tastatur oder Touchscreen anpassen.

RespVis ist eine quelloffene Bibliothek für das Erstellen von responsiven Visualisierungen. Der Code ist in TypeScript geschrieben, baut auf der populären Visualisierungsbibliothek D3 auf, und ist in mehrere Softwarepakete gegliedert, die öffentlich auf der npm Registratur verfügbar sind. Diagrammautoren mit Erfahrung im Bereich Webentwicklung können mit eigenen Datensätzen die API von RespVis verwenden, um responsive Balkendiagramme, Streudiagramme, Liniendiagramme und Parallele Koordinaten Diagramme zu erstellen. Diagrammautoren mit mehr Erfahrung im Bereich Webentwicklung können die zur Verfügung gestellten RespVis Diagrammtypen individuell anpassen oder eigene, neue Diagrammtypen erstellen.

Die RespVis API erleichtert die Anwendung von einigen gängigen responsiven Mustern. Diagrammkomponenten wie Titel, Achsen und Legenden können mit mächtigen Layout-Techniken wie Flexbox und Grid positioniert werden, obwohl RespVis Diagramme reine SVGs sind. Diese Besonderheit wird durch die spezielle RespVis Layout Engine ermöglicht, die im Hintergrund arbeitet. Während der Erstellung der Masterarbeit wurden dutzende Updates und Verbesserungen am Quellcode durchgeführt, woraus die aktuelle Version, RespVis v3, resultierte.



# Contents

<b>Contents</b>	<b>iv</b>
<b>List of Figures</b>	<b>vi</b>
<b>List of Tables</b>	<b>vii</b>
<b>List of Listings</b>	<b>x</b>
<b>Acknowledgements</b>	<b>xi</b>
<b>Credits</b>	<b>xiii</b>
<b>1 Introduction</b>	<b>1</b>
<b>2 Web Technologies</b>	<b>3</b>
2.1 HyperText Markup Language (HTML)	3
2.2 Cascading Styling Sheets (CSS)	4
2.3 JavaScript	4
2.3.1 Browser Web APIs	4
2.3.2 JavaScript Module Formats	5
2.4 TypeScript	6
2.5 Raster Graphics	6
2.6 Scalable Vector Graphics (SVG)	7
2.7 D3	8
2.8 NodeJS	10
2.9 Rollup	11
2.10 Gulp	11
2.11 Storybook	11
2.12 Responsive Web Design	12
2.12.1 Responsive Design Strategies	13
2.12.2 Modern Responsive Design	14
2.12.3 Avoiding Horizontal Scrolling	14

<b>3</b>	<b>Responsive Visualization</b>	<b>17</b>
3.1	Information Visualization . . . . .	17
3.2	Mobile Visualization . . . . .	18
3.3	Display Properties . . . . .	18
3.4	Responsive Visualization. . . . .	18
3.5	Responsive Visualization Patterns . . . . .	20
3.5.1	Visual Patterns . . . . .	20
3.5.2	Interaction Patterns . . . . .	25
3.5.3	Data Patterns. . . . .	28
<b>4</b>	<b>RespVis v1 and RespVis v2</b>	<b>33</b>
4.1	RespVis v1. . . . .	33
4.2	RespVis v2. . . . .	34
<b>5</b>	<b>RespVis v3</b>	<b>35</b>
5.1	Project Structure. . . . .	37
5.1.1	Package Structure . . . . .	37
5.1.2	Gulp Tasks . . . . .	40
5.1.3	Self-Contained Examples . . . . .	41
5.1.4	Live Documentation . . . . .	41
5.2	Library Design . . . . .	43
5.2.1	Naming Conventions . . . . .	44
5.2.2	Sub-Package Modules . . . . .	45
5.2.3	Component Hierarchy . . . . .	46
5.3	RespVis Core . . . . .	48
5.3.1	Window Modules . . . . .	48
5.3.2	Toolbar Modules . . . . .	50
5.3.3	Layouter Modules . . . . .	53
5.3.4	Chart Modules . . . . .	56
5.3.5	Data Series Modules . . . . .	59
5.3.6	Axis Modules . . . . .	60
5.3.7	Legend Modules . . . . .	61
5.3.8	Marker Primitive Module . . . . .	62
5.3.9	Label Modules . . . . .	63
5.3.10	Element Modules . . . . .	63
5.3.11	Scale Modules . . . . .	63
5.3.12	Categories Module. . . . .	64
5.3.13	Breakpoints Modules. . . . .	64
5.3.14	Responsive Property Modules. . . . .	66
5.3.15	Sequential Color Module . . . . .	67
5.3.16	Zoom Module . . . . .	69
5.3.17	Utilities Modules . . . . .	69

5.4	RespVis Tooltip . . . . .	71
5.5	RespVis Cartesian . . . . .	72
5.6	RespVis Bar . . . . .	76
5.6.1	Bar Chart Modules. . . . .	77
5.6.2	Bar Base Series Modules . . . . .	78
5.6.3	Bar Grouped Series Modules . . . . .	78
5.6.4	Bar Stacked Series Modules . . . . .	79
5.6.5	Bar Module . . . . .	79
5.6.6	Bar Label Module . . . . .	80
5.7	RespVis Point. . . . .	80
5.7.1	Scatter Plot Modules . . . . .	81
5.7.2	Point Series Modules . . . . .	81
5.7.3	Point Module. . . . .	82
5.7.4	Point Label Module . . . . .	82
5.7.5	Radius Modules. . . . .	83
5.8	RespVis Line . . . . .	83
5.8.1	Line Chart Modules . . . . .	83
5.8.2	Line Series Modules . . . . .	84
5.9	RespVis Parcoord . . . . .	84
5.9.1	Parallel Coordinates Chart Modules . . . . .	85
5.9.2	Parallel Coordinates Series Modules . . . . .	86
<b>6</b>	<b>Outlook and Future Work</b>	<b>89</b>
<b>7</b>	<b>Concluding Remarks</b>	<b>91</b>
<b>A</b>	<b>User Guide</b>	<b>93</b>
A.1	PC and Mobile Interactions . . . . .	93
A.2	Toolbar Interactions . . . . .	94
A.2.1	Filter Tool . . . . .	94
A.2.2	Download Tool . . . . .	94
A.2.3	Inspection Tool . . . . .	96
A.2.4	Chart Settings Tool . . . . .	96
A.3	Tooltip Interactions . . . . .	96
A.4	Legend Interactions. . . . .	99
A.5	Zooming . . . . .	99
A.6	Parallel Coordinates Chart Interactions. . . . .	101
<b>B</b>	<b>Chart Creator Guide</b>	<b>103</b>
B.1	RespVis Patterns. . . . .	103
B.1.1	Visual Patterns . . . . .	103
B.1.2	Interaction Patterns . . . . .	105

B.2	Bar Chart . . . . .	106
B.3	Scatter Plot. . . . .	112
B.4	Line Chart . . . . .	118
B.5	Parallel Coordinates Chart . . . . .	125
<b>C</b>	<b>Chart Developer Guide</b>	<b>133</b>
C.1	Creating New Charts . . . . .	133
C.2	Customizing Standard Chart Types . . . . .	136
<b>D</b>	<b>Maintainer Guide</b>	<b>139</b>
D.1	Releasing . . . . .	139
D.2	Importing and Exporting . . . . .	140
D.2.1	Importing and Exporting TypeScript . . . . .	140
D.2.2	Importing CSS . . . . .	140
	<b>Bibliography</b>	<b>143</b>

# List of Figures

2.1	Scaling Raster Graphics and Vector Graphics . . . . .	7
2.2	SVG Created with D3. . . . .	9
2.3	Responsive Breakpoint Diagram . . . . .	13
3.1	Responsive Line Chart . . . . .	19
3.2	V1: Scaling Entire Chart Down . . . . .	21
3.3	V2: Repositioning Element Labels in Scatter Plot . . . . .	22
3.4	V3: Using Tooltips Instead of Element Labels . . . . .	22
3.5	V4: Rotating Axis Tick Labels . . . . .	23
3.6	V5: Shortening Labels and Titles . . . . .	23
3.7	V6: Scaling Labels Between Minimum and Maximum Size . . . . .	24
3.8	V9: Rotating Chart 90° . . . . .	26
3.9	I1: Providing a Menu . . . . .	27
3.10	I1: Providing a Toolbar . . . . .	27
3.11	I2: Filtering Dimensions and Records . . . . .	28
3.12	I3: Supporting Zooming. . . . .	28
3.13	D1: Data Generalization. . . . .	29
3.14	D1: Data Generalization. . . . .	30
3.15	D2: Data Aggregation . . . . .	31
5.1	Sub-Package Dependencies . . . . .	40
5.2	Live Documentation . . . . .	42
5.3	Respvis Component Hierarchy . . . . .	48
5.4	Toolbar . . . . .	50
5.5	Statically Positioned Toolbar . . . . .	51
5.6	Filter Menu . . . . .	51
5.7	Download Modal . . . . .	52
5.8	Inspection Tool . . . . .	53
5.9	Layout Phases . . . . .	55
5.10	Chart Layout . . . . .	57
5.11	Padding Wrapper Layout . . . . .	58
5.12	Flipping Axes . . . . .	61

5.13	Rotating Axis Labels . . . . .	62
5.14	Shifting Breakpoints . . . . .	66
5.15	Breakpoint Property Interpolation . . . . .	68
5.16	Sequential Color Encoding . . . . .	69
5.17	Data Series Tooltip. . . . .	72
5.18	Cartesian Padding Wrapper Layout . . . . .	74
5.19	Origin Line . . . . .	74
5.20	Grid Lines. . . . .	75
5.21	Inverted Cartesian Axis . . . . .	76
5.22	Bar Chart . . . . .	78
5.23	Grouped Bar Chart. . . . .	79
5.24	Stacked Bar Chart . . . . .	80
5.25	Point Label Position Strategies . . . . .	82
5.26	Responsive Bubble Radii . . . . .	83
5.27	Multi-Series Line Chart . . . . .	85
5.28	Parallel Coordinates Chart . . . . .	86
A.1	Toolbar . . . . .	94
A.2	Filter Menu . . . . .	95
A.3	Download Modal . . . . .	95
A.4	Toolbar Tooltip . . . . .	97
A.5	Inspection Tool . . . . .	97
A.6	Data Series Tooltip. . . . .	98
A.7	Legend Highlighting . . . . .	98
A.8	Legend Filtering . . . . .	99
A.9	Zooming in Cartesian Charts . . . . .	100
A.10	Zooming in Parallel Coordinates Charts . . . . .	100
A.11	Parallel Coordinates Chart Equidistant Axes . . . . .	101
A.12	Parallel Coordinates Chart With No Equidistant Axes . . . . .	101
A.13	Parallel Coordinates Chart Range Sliders. . . . .	102
A.14	Parallel Coordinates Chart Axis Inversion . . . . .	102
B.1	Grouped Bar Chart. . . . .	107
B.2	Scatter Plot . . . . .	113
B.3	Multi-Series Line Chart . . . . .	119
B.4	Parallel Coordinates Chart . . . . .	126
C.1	Axis Chart. . . . .	136
C.2	Customizing a Bar Chart . . . . .	137



# List of Tables

5.1	RespVis v3 Naming Conventions . . . . .	45
-----	---	----



# List of Listings

2.1	D3 Selection API . . . . .	9
2.2	Media Query Example . . . . .	12
5.1	Top-Level File and Directory Structure . . . . .	38
5.2	Sub-Package File and Directory Structure . . . . .	39
5.3	Self-Contained Examples File and Directory Structure . . . . .	42
5.4	Live Documentation File and Directory Structure. . . . .	43
5.5	Chart Module Validation Logic . . . . .	47
5.6	Core Directory Structure . . . . .	49
5.7	Chart Layout CSS . . . . .	57
5.8	Padding Wrapper Layout CSS. . . . .	58
5.9	Padding CSS Variables . . . . .	59
5.10	Component Breakpoints Argument . . . . .	65
5.11	Component Breakpoints in Style Container Queries . . . . .	65
5.12	Component Breakpoints in Size Container Queries . . . . .	66
5.13	Title Argument as Responsive Value . . . . .	67
5.14	Tick Orientation Argument as Breakpoint Property . . . . .	68
5.15	Tooltip Directory Structure. . . . .	71
5.16	Cartesian Directory Structure . . . . .	73
5.17	Cartesian Padding Wrapper Layout CSS . . . . .	73
5.18	Bar Directory Structure . . . . .	77
5.19	Point Directory Structure . . . . .	81
5.20	Line Directory Structure. . . . .	84
5.21	Parallel Coordinates Directory Structure . . . . .	85
B.1	Grouped Bar Chart: HTML . . . . .	108
B.2	Grouped Bar Chart: CSS . . . . .	109
B.3	Grouped Bar Chart: TypeScript . . . . .	110
B.4	Scatter Plot: HTML . . . . .	112
B.5	Scatter Plot: CSS . . . . .	114
B.6	Scatter Plot: TypeScript . . . . .	115
B.7	Multi-Series Line Chart: HTML. . . . .	120
B.8	Multi-Series Line Chart: CSS. . . . .	121
B.9	Multi-Series Line Chart: TypeScript . . . . .	122
B.10	Parallel Coordinates Chart: HTML. . . . .	127
B.11	Parallel Coordinates Chart: CSS. . . . .	128
B.12	Parallel Coordinates Chart: TypeScript . . . . .	130

C.1 Axis Chart Class . . . . . 134  
C.2 Axis Chart Validation. . . . . 135  
C.3 Custom Bar Chart Class . . . . . 137

# Acknowledgements

Major thanks go to my advisor, Keith Andrews, who always had an ear open for my many questions of both technical and literal nature. Without him, this work only would have been an inferior version of the actual work.

I am also indebted to Graz University of Technology, for the invaluable knowledge I could gather through many years of studying. Thanks go to my colleagues, who always supported and accompanied me during this time.

Lastly, I want to thank my friends, my family, and my girlfriend, from the deepest of my heart for their endless support, and the patience I received from them. Completing this work would have been a lost cause without their persistent care.

David Egger  
Graz, Austria, 11 Dec 2024



# Credits

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

- The thesis was written using Keith Andrews' skeleton thesis [Andrews 2021].
- Figure 2.3 is used with kind permission of Keith Andrews, Graz University of Technology.
- Figure 3.1 is used with kind permission of Keith Andrews, Graz University of Technology.





# Chapter 1

## Introduction

The field of data visualization seeks to present data and information visually, so as to facilitate its rapid assimilation and understanding [Andrews 2024]. Early figures in the field, like William Playfair and Florence Nightingale, produced visual representations of data on paper [Infogram 2016]. Today, charts and visualizations are largely viewed online on the web. Smartphones and tablets are also increasingly displacing desktops and laptops as the viewing platform of choice. In the fourth quarter of 2023, already 54.67% of all web page views worldwide were requested from mobile devices [Statista 2023b].

Responsive web design refers to a set of techniques, which are used to create web pages and applications capable of adapting to the characteristics of the end user’s device [Marcotte 2014]. These days, all web design is responsive web design. Data visualizations capable of adapting to the different requirements of desktops, tablets, smartphones, and other devices, are called *responsive visualizations* [Andrews 2018b].

This thesis presents RespVis v3 [Egger and Oberrauner 2024a; Egger and Oberrauner 2024b; Egger 2024j], the current version of RespVis, an open-source library for creating responsive visualizations. RespVis v3 is written in TypeScript [Microsoft 2024b] and is built on top of D3 [Bostock 2024h], a powerful low-level JavaScript visualization library. D3 works by injecting SVG or HTML nodes into the DOM, binding data to these nodes, and defining the appearance of the nodes based on the bound data. RespVis v3 provides an intuitive, uniform API for the creation of complete charts, but also allows chart creators to access the underlying functionality. This includes overriding and explicitly customizing the render routines of the provided charts.

Since the source code of RespVis v3 is written in TypeScript, library users can rely on full type support when importing modules from the official RespVis v3 packages. RespVis v3 provides multiple packages, which are publicly available on the npm registry [NPM 2023], from where they can conveniently be installed using a NodeJS package manager.

While the API of RespVis v3 is more extensive than in previous versions, the custom layout mechanism introduced in RespVis v1 remains a core feature. It allows chart creators to apply powerful CSS layout techniques to elements contained in an `<svg>` element, which is generally not possible. Recommended good practice is to use the custom layout mechanism to position and size higher-level chart components like titles, legends, and axes via CSS. In contrast, elements in the drawing area should not use the custom layout mechanism, since these elements typically must be positioned at exact coordinates depending on the underlying data. The custom layout mechanism was slightly improved in RespVis v3, such that alternating between standard SVG layout and the custom layout mechanism became possible at lower nesting levels. This feature proved useful for positioning the titles of axes where they are contained inside the drawing area of a chart, for example with a parallel coordinates chart.

A core feature introduced with RespVis v3 is the layout breakpoints API. The concise definition of layout breakpoints and their reusability in CSS via style container queries makes them a powerful tool and opens up new possibilities for applying responsive patterns to RespVis visualizations.

To avoid any confusion, some frequently occurring terminology is clarified at this point. RespVis identifies four specific kinds of user. A *chart end user* (or simply *end user*) is a person who does not necessarily know anything about the technical details of RespVis, but views and explores charts created with RespVis. A *chart creator* (or *chart author*) is a user with some web developer experience, who has some data and wishes to create a responsive chart with that data using the RespVis API. A *chart developer* is a more experienced web developer, who wishes to customize the standard RespVis charts or create their own new charts. Finally, a *maintainer* is a custodian of the RespVis library itself.

In terms of datasets, RespVis typically deals with tabular (spreadsheet) data in the form of *records* (or *data points*), each represented by a row in the table, and *dimensions* (or *variables*), each represented by a column in the table. A *data series* is a set of data points which belong together. In RespVis v3, the data points in a data series can be specifically grouped into categories. For example, a multi-line chart with measurements from three regions would be handled as one data series with three categories and one polyline would be drawn for each category.

The rest of this thesis is structured as follows. Chapter 2 discusses some of the modern web technologies used in the implementation of RespVis, beginning with the core technologies of the web, HTML, CSS, and JavaScript, progressing to more specific tools like TypeScript, NodeJS, Gulp, and Storybook, and ending with an explanation of responsive web design. Chapter 3 continues with an overview of information and data visualization, and specific discussion of responsive data visualization.

Chapter 4 describes the previous versions of RespVis, which were used as a starting point for this work. They are referred to as RespVis v1 [Oberrauner 2022b; Oberrauner 2022a] and RespVis v2 [Egger and Oberrauner 2023a]. Chapter 5 contains the main body of the thesis, describing the dozens of updates and improvements made to RespVis during this thesis work to produce RespVis v3. It includes detailed explanations of the project structure, library design, and all available RespVis packages. The thesis concludes with an outlook and ideas for future work in Chapter 6.

The thesis has four appendices. Appendix A contains a User Guide for end users viewing and exploring responsive charts created with RespVis v3. Appendix B contains a Chart Creator Guide for chart creators wishing to use the standard charts provided by RespVis v3 with their own datasets. It describes how to apply various responsive patterns, and includes a complete example of each of the four chart types available in RespVis v3. Appendix C contains a Chart Developer Guide for developers wishing to modify existing charts or create their own charts. Finally, Appendix D contains a Maintainer Guide for subsequent maintainers and contributors, who might work on RespVis in the future.

## Chapter 2

# Web Technologies

RespVis is a library conceived to be used on the web. As such, it makes use of the core technologies of the web: HTML, SVG, CSS, and JavaScript. RespVis is actually implemented in TypeScript, a strictly typed superset of JavaScript. RespVis is heavily dependent on D3, the widely used JavaScript library for data visualization. Visualizations on the web are a form of web graphics. The official vector graphics standard of the web is Scalable Vector Graphics (SVG). The purpose of RespVis is the creation of responsive visualizations, which are rendered as SVGs.

Many technologies are used as developer dependencies to improve the development workflow when dealing with the source code, the curated set of self-contained examples, and the live documentation. NodeJS and npm are used for dependency management and running the tools Gulp, Rollup, and Storybook. Gulp is used for automating and orchestrating tasks like the bundling of the source code via the module bundler Rollup. Storybook is used for creating RespVis' live documentation.

### 2.1 HyperText Markup Language (HTML)

HTML is a markup language used to describe the content and structure of a web page. Some HTML elements are capable of containing other elements, leading to a tree hierarchy of elements. It was introduced in 1993 by Tim Berners-Lee and originally designed to organize scientific information. The core functionality provided by HTML is its capability of marking up structured text and connecting documents over a digital channel using links (HyperText) [Berners-Lee 1999; Gillies and Cailliau 2000; Vujovic 2024].

HTML was soon adopted, and its potential was recognized by various companies and institutions. This eventually led to the foundation of the World Wide Web Consortium (W3C) in October 1994 [W3C 2024b] and the subsequent publication of the HTML 2.0 standard as RFC 1866 in November 1995 [Berners-Lee and Connolly 1995].

A major step forward was the specification of HTML5 in October 2014 [W3C 2014], which significantly extended and improved the features of HTML. The last versioned specification was HTML 5.2 in December 2017 [W3C 2017]. In May 2019, the W3C passed control over the HTML standard to the Web Hypertext Application Technology Working Group (WHATWG) [WHATWG 2024b]. HTML is now a living standard, which is continuously updated [WHATWG 2024a].

HTML is parsed and interpreted by web browsers, which create an in-memory representation of the parsed markup called the Document Object Model (DOM) [MDN 2023d]. Since HTML5, web browsers are capable of parsing SVG elements as well as HTML elements and adding them to the DOM. Browsers provide Application Programming Interfaces (APIs) allowing DOM elements to be manipulated via JavaScript, as discussed in Section 2.3.

## 2.2 Cascading Styling Sheets (CSS)

CSS is a declarative stylesheet language, proposed in 1994 by Håkon Wium Lie, and officially standardized in 1996 by the W3C as CSS 1 [W3C 1996]. The motivation of Lie was to take the existing idea of separating structure from presentation by introducing a separate language for the presentation aspect [Bos 2016]. In CSS, the presentation of the markup can be controlled by specifying CSS rules. A CSS rule consists of two parts: a *selector* specifying to which elements a rule should be applied, and a set of *declarations* (wrapped in curly braces) to be applied to the selected elements. Each declaration consists of a property and a value. Depending on which values are assigned to which properties, the presentation of an element changes [MDN 2024k]. A core feature of CSS is its capability to apply a sophisticated cascading algorithm, to detect which CSS rules have precedence over others [MDN 2024c].

CSS 2 was defined in May 1998 [W3C 1998], followed by CSS 2.1 in June 2011 [W3C 2011a]. Rather than being a single, monolithic standard, CSS 3 was divided into numerous separate documents (modules), which are progressed and standardized independently, and are collected into annual “snapshots” [W3C 2024a].

The most common ways of adding CSS to a web site are the addition of `<link>` elements referencing external stylesheets, and the usage of inline styles specified directly in the markup. The parsing and interpretation of CSS by modern browsers is invoked during the creation of the DOM. When the HTML parser of the browser comes across an element linking to an external stylesheet or an element with an inline style, the creation of the CSS Object Model (CSSOM) is initiated. The CSSOM is the result of parsing and interpreting all CSS contained in a document. The styles are applied by constructing the Render Tree out of the DOM and CSSOM. The Render Tree represents all actually visible elements of a loaded document and their current properties [Irish and Garsiel 2011].

## 2.3 JavaScript

Before JavaScript was conceived, no interactivity could be added to web sites. Browsers could only interpret HTML for displaying content and handle simple events like form submissions in a limited manner. The original prototype of JavaScript was developed by Brendan Eich at Netscape, under the name LiveScript, to provide a solution for these limitations for the Netscape Navigator 2 web browser. Soon, other companies began to implement their own versions of JavaScript to add interactivity to their browsers [Manik 2020].

It became clear that the language needed to be standardized such that web developers could be sure their client-side code works in all browsers. This was the birth of the ECMAScript [Ecma 2024], standardized by the then European Computer Manufacturers Association (today known as Ecma International) in 1997. This language specification solely concerns the core constructs of the JavaScript language like language syntax, data types, objects, and arrays.

JavaScript was designed to be executed in the runtime environment provided by web browsers like Google Chrome, Firefox, and Safari. More recently, server-side runtime environments have been developed for JavaScript, such as NodeJS, Deno, and Bun. A JavaScript runtime environment has a JavaScript engine to interpret and execute JavaScript code, and all JavaScript engines strive to comply with the current ECMAScript version published annually by Ecma International.

### 2.3.1 Browser Web APIs

Depending on the runtime environment being used, additional APIs are made available to JavaScript. Inside a web browser, these are called Web APIs. Organizations like the W3C [W3C 2024b] and WHATWG [WHATWG 2024b] publish API standards recommended to be implemented by browsers, but browser vendors choose whether to implement a particular API or not, depending on various factors like usefulness, demand, and implementation difficulty.

A list of browser Web APIs is maintained by MDN [2023f]. Among the most important is the DOM-API, which is used to access and manipulate DOM nodes, as discussed in Section 2.1. The DOM-API is a collection of interfaces describing all existing element types in the DOM. Accessing properties and functions defined by these interfaces in JavaScript is reliably supported by browsers. The DOM-API comprises the HTML-DOM-API [MDN 2024i] and the SVG-DOM-API [MDN 2024j], which are collections of interfaces describing all elements in HTML documents and SVG documents respectively.

### 2.3.2 JavaScript Module Formats

In the beginning, JavaScript was conceived as a scripting language for executing simple tasks at the client-side of web sites. However, over the years, more functionality was included in the language, and it became capable of executing complex procedures. With the release of NodeJS [OpenJS 2024b], JavaScript became a viable alternative for backend (server-side) applications, making it possible to create full stack applications entirely in JavaScript. Today, JavaScript is the most widely used programming language in the world [Statista 2023a]. Apart from the web, it is also commonly used in game development, desktop applications, and data visualization, to name only a few.

Due to the massive growth of the language, there was high demand among developers for code modularization. This led to the development of standard formats for collecting JavaScript code into modules. There are now five principal formats for JavaScript modules:

- *Immediately Invoked Function Expressions (IIFE)*: The IIFE module pattern was conceived to solve the problem of global scope pollution. In the early days of JavaScript, there was no standard syntax for modularization. In IIFE, all code in the module is wrapped in an anonymous function, which is called immediately and saved into one global variable. This variable represents and holds the API of the module, and can be called later by other scripts in a controlled manner [Dixin 2024].
- *CommonJS (CJS)*: CJS is a module pattern founded under the lead of Kevin Dangoor in 2009, with the goal of easing the organization of code in large-scale projects, especially server-side [Theekshana 2024]. It was quickly adopted by NodeJS, which was released later in the same year. For systems applying the pattern, each JavaScript file is interpreted as CJS module. Each module must provide a `module.exports` property, declaring the properties visible to other modules. Modules can import other modules using the `require` function. CJS applies synchronous module loading, which comes with a guarantee of dependencies being always loaded before usage, but may also lead to performance issues, especially when loading dependencies conditionally or in loops.
- *Asynchronous Module Definition (AMD)*: AMD is a module pattern designed to be used on the client-side, where asynchronous module loading is required for performance reasons. Its API specification defines a single function `define` for defining a dependency array and the API of a module as factory function. Loading modules should be achieved by using the reserved `require` keyword [AMD 2016]. The most prominent implementation of the AMD pattern is the module loader RequireJS [RequireJS 2024].
- *Universal Module Definition (UMD)*: UMD emerged from the need for libraries to be independently usable in all used module environments. The idea is to make use of an IIFE, which checks for the current environment (AMD, CJS, or as browser globals) and, based on that, defines the module in an appropriate way [UMD 2024].
- *ECMAScript Modules (ESM)*: With the proliferation of different module patterns, it was high time for a modularization system to be standardized as part of the native JavaScript syntax [MDN 2024d]. This was achieved with the introduction of ESM as part of ECMAScript2015 [Ecma 2015]. ESM modules exactly define their dependencies to other modules using the `import` keyword and their API to other modules using the `export` keyword. Today, all major browsers support ESM modules. They can be included in HTML as `<script>` elements with their `type` attribute set to `module`.

## 2.4 TypeScript

In its early days, JavaScript was used mainly to add a small amount of interactivity to web sites. For this reason the language was invented to be dynamically typed. In dynamically typed languages, variable types are checked at runtime, meaning there is no need to explicitly define a variable's type in its definition. Furthermore, the variable type can change dynamically multiple times, which is possible due to type re-checking at runtime. The advantages of a dynamically typed language like JavaScript are great flexibility and concise and easy-to-write code [BairesDev 2024].

As explained in Section 2.3, JavaScript evolved dramatically over the years, becoming a general purpose language usable for creating large-scale frontend and backend applications. However, the language is not well-suited for implementing these kinds of applications due to its dynamic typing. This led to the invention of TypeScript, a statically-typed superset of JavaScript, which was first released in 2012 by Microsoft [Jackson 2012].

TypeScript essentially provides two tools: the TypeScript compiler `tsc`, and the TypeScript standalone server `tsserver` [Vanderkam 2024, page 27]. The compiler, which can be configured by creating a `tsconfig.json` file, is used to compile TypeScript into JavaScript, which is subsequently executed in a runtime environment. This means any production code written in TypeScript is compiled beforehand and executed as plain and pure JavaScript, effectively leading to no additional computational cost when choosing TypeScript over JavaScript. One of the main features of TypeScript is its easy migration to JavaScript applications, since all syntactically valid JavaScript programs are also syntactically valid TypeScript programs. The rules specified in the `tsconfig.json` file determine which code should be included in the type-checking process, and how strict the type-checking process should be. The major benefit of type-checking is that errors can be caught at compile time, leading to better quality code.

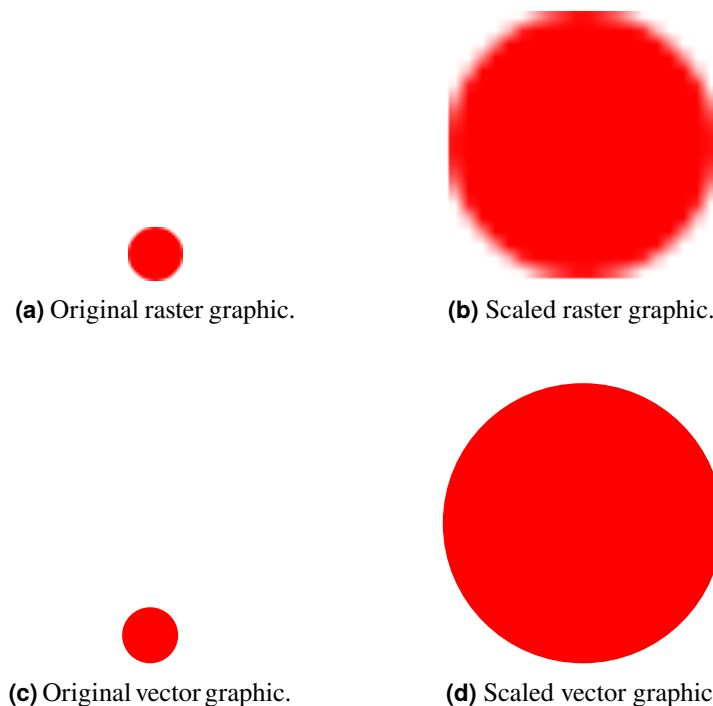
The second main reason to use TypeScript are the language services provided by the TypeScript standalone server [Vanderkam 2024, page 27]. Using these services greatly improves the workflow when developing large-scale applications. The services include autocompletion, error checking, enhanced refactoring, and code navigation. A TypeScript developer makes use of these services by using an Integrated Development Environment (IDE) configured to communicate with the TypeScript standalone server.

## 2.5 Raster Graphics

In the early days of HTML, text was the only content of web pages. Soon, the web community began to realize how useful it would be to embed multimedia like images, audio, and videos into web pages. This resulted in the invention of the `<img>` element, which was first released in the Mosaic web browser in 1993, and later officially standardized as part of HTML2.0 in 1995 [Hoffmann 2017]. This new element made it possible to embed raster graphics next to the text of a web page, at first in the form of GIF images.

A raster graphic stores the information of an image by defining a two-dimensional grid, where each grid cell contains color information and is called a *pixel*. The quality of a raster graphic is determined by its resolution and color depth [Pickle 2023]. Having a high resolution, i.e. many pixels, and deep color depth results in smooth color transitions, enhancing the overall quality of an image. However, with increasing quality, the file size increases too. This can pose a problem, especially in web development where files should be as small as possible to speed up loading times.

To address this issue, different compressed raster graphic formats were conceived. The most prominent ones are Joint Photographic Experts Group (JPEG) and Portable Network Graphics (PNG). JPEG is the most used type on the web, since it provides a good trade-off between quality and file size. The downside of JPEG are the artifacts caused by its lossy compression algorithm, which accumulate if the image is re-edited. PNG images, in contrast, are saved using a lossless compression algorithm, but with significantly larger file sizes [Weinreb 2019]. Recently, more advanced image formats with even better compression



**Figure 2.1:** Scaling up a raster graphic leads to aliasing effects and a blurred appearance, while an equivalent, scaled up vector graphic remains crisp. [Images created by the author of this thesis.]

and quality have started to emerge and be supported by web browsers, including WebP, AVIF, and JPEG XL [Osmani 2021].

## 2.6 Scalable Vector Graphics (SVG)

In contrast to raster graphics, vector graphics are not defined by a static raster of pixels, but use mathematical equations to define the shapes and colors of a graphic in a two-dimensional coordinate system. This comes with two major advantages compared to raster graphics: smaller file size, and automatic crisp scalability, independent of the graphic’s resolution. Figure 2.1 demonstrates the negative effects of scaling up a low-resolution raster graphic compared to the equivalent scaled up vector graphic.

Using vector graphics on the web became possible with SVG. SVG is an XML-based markup language standard for vector graphics, which has been continuously improved since the late 1990s by the SVG working group [W3C 2010]. Since SVGs are XML-based, they can be edited with both text editors, and graphical drawing editors such as Adobe Illustrator and Inkscape.

Like HTML, SVG is parsed by browsers to create and maintain a DOM from the parsed SVG elements. SVG is specifically designed to work well with other web standards like HTML, CSS, and JavaScript. For example, complete SVG documents can be embedded into HTML documents without additional adjustments. The concept of XML namespaces avoids possible naming collisions between elements of different XML dialects, such as the `<title>` element which exists in both the HTML and SVG specifications [MDN 2024f].

The appearance and position of SVG elements can be defined in three ways. The first method is to set the attributes of SVG elements directly in the markup, which exhibits the lowest specificity. The second method is to specify attribute values via CSS. This approach is only applicable to so-called *presentational attributes*, a subset of all SVG attributes, and has a higher specificity than the first approach. The third

method is to use inline styles, which like the second approach is only applicable to a subset of SVG attributes, but has the highest specificity.

SVGs can be easily made interactive. This is achieved by combining the SVG-DOM-API, a dedicated API for manipulating SVG elements, with the possibility of assigning event listeners to SVG elements. Both of these features are supported by all major browsers.

The first SVG standard was SVG 1.0 in 2001 [W3C 2001]. SVG 1.1 was specified in 2003 [W3C 2003], and its second edition SVG 1.1 (Second Edition) in 2011 [W3C 2011b]. Modern web browsers have broad support for SVG 1.1 (Second Edition). SVG 2 has been a W3C Candidate Recommendation since October 2018 [W3C 2018], but has only partially been implemented in modern web browsers, and doubts persist whether it will ever become a full W3C Recommendation with wide browser support [Coyier 2016].

The various advantages of SVGs make it the perfect medium for creating responsive visualizations on the web. They can be scaled without loss of quality, have a small file size, can easily be made interactive using Web APIs, live directly in the DOM, and allow the exact positioning of elements.

## 2.7 D3

D3 [Bostock et al. 2011; Bostock 2024h] is a powerful, low-level JavaScript visualization library. It differs from traditional charting libraries, since no API for creating complete charts is provided. Instead, recipes show how to bind data to marks, which are composed to form various chart types. D3 also provides a wide range of utility functions for many use cases. The library follows a poly-repository architecture by grouping related utility functions into separate repositories. The code of each repository is provided as a package on the npm registry. This section describes the D3 packages used by RespVis v3. The book by Wattenberger [2019] is an excellent and extensive resource for a deep dive about creating visualizations with D3.

D3 enables the manipulation of DOM elements via its Selection API, which is part of the `d3-selection` package [Bostock 2024e]. Selections of DOM elements can be created using the functions `Selection.select`, `Selection.selectAll`, `Selection.selectChild`, and `Selection.selectChildren` and passing selector strings as arguments. The `Selection.data` function can be used in multiple ways to bind data to selections. A common practice is to pass an array of data objects, where each object is bound to an element. The mapping between data objects and elements can be specified as a key function. By default, the array index of a data object is used as the key. The `Selection.data` function can be called repeatedly to update the underlying data. By subsequently calling `Selection.join` on a selection, new elements can be created for previously non-existent data, and old elements, whose data was removed in the data update, can be removed. This procedure, called a D3 data join, results in intermediate enter, update, and exit selections and a join selection, containing the elements existing in the DOM after the data update.

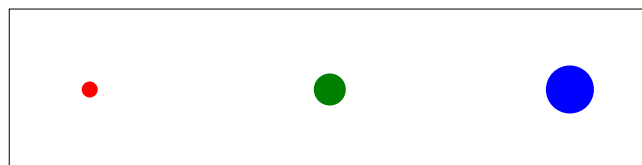
Selections also provide modification functions like `Selection.classed`, `Selection.attr`, and `Selection.style` to modify elements of a selection. In RespVis v3, selections are omnipresent in render routines. Listing 2.1 demonstrates how the DOM can be manipulated via D3 selections. The resulting SVG is shown in Figure 2.2. As can be seen, D3's API exploits the advantages of method chaining to enable developers to write more concise code. The `Selection.call` method is used to call any function expecting a selection as input argument, without interrupting method chaining.

D3's `d3-transition` package [Bostock 2024f] provides the API for interacting with D3 transitions. Transitions can be created by calling the `Selection.transition` function. A transition can be used similarly to a selection. The difference is that modification functions, which conduct changes to the DOM, are animated rather than being applied instantly. Furthermore, certain methods provided by selections like `Selection.data` and `Selection.join` are not available for transitions. In RespVis v3, transitions are used to animate DOM changes of labels and markers in the drawing area.



```
1 export function renderCircles() {
2   const width = 400
3   const height = 100
4   const offset = 100
5   const circles = [{
6     size: 5,
7     color: 'red',
8   }, {
9     size: 10,
10    color: 'green',
11  }, {
12    size: 15,
13    color: 'blue',
14  }]
15
16  function printDimension(selection: Selection) {
17    console.log(selection)
18  }
19
20  const circleS = d3.selectAll('.svg-wrapper')
21    .data([null])
22    .join('svg')
23    .attr('viewBox', `0 0 ${width} ${height}`)
24    .classed('svg-wrapper', 'wrapper')
25    .call(printDimension)
26    .selectAll('.circle')
27    .data(circles)
28    .join('circle')
29    .attr('fill', d => d.color)
30    .attr('cx', (d, i) => offset / 2 + i * ((width - offset) / (circles.length - 1))
31      )
32    .attr('cy', `${height / 2}`)
33    .attr('r', d => d.size)
34 }
```

**Listing 2.1:** Using D3's Selection API to create a simple SVG containing three colored circles. The output is shown in Figure 2.2.



**Figure 2.2:** An SVG containing three colored circles. The graphic was produced by the code in Listing 2.1. [Image created with D3 [Bostock 2024h] by the author of this thesis.]

The `d3-dispatch` package [Bostock 2024b] allows for registering and dispatching named callbacks. In contrast to DOM events, which involve the event queue and asynchronous execution of event listeners, the callbacks registered by `d3.dispatch` and `Selection.dispatch` are executed immediately when dispatching them. In `RespVis v3`, the `D3`'s dispatch system is mainly used to manually trigger re-renders.

The `d3-scale` package [Bostock 2024d] provides various functions for creating `Scale` objects, which map one domain of values to another. In most cases, scales are used for mapping input data to a visual representation. The mapping between data values and screen position is the most frequent use case of scales in `RespVis v3`. However, scales are also used for mapping a numerical dimension to a color range, for instance to color-code data points in scatter plots.

The `d3-axis` package [Bostock 2024a] provides convenient utility functions for creating axis components. The functions `axisTop`, `axisBottom`, `axisLeft`, and `axisRight` are used to create `Axis` objects with different orientation configurations. The functions expect a scale to be passed as an argument for the creation of an axis object, which can further be configured to customize the desired visualization of axis ticks. Axis objects are also functions. When calling an axis object as a function and passing a selection of `<g>` elements, the `<g>` elements are populated with all visual elements of the axis. In `RespVis v3`, the utility functions of the `d3-axis` package are heavily used, since all provided chart types contain axes.

The `d3-drag` [Bostock 2024c] and `d3-zoom` [Bostock 2024g] packages provide a convenient way for adding drag and zoom interactions to selections. The functions `d3.drag` and `d3.zoom` create drag and zoom behavior objects respectively, which are also functions. To handle drag and zoom events, event listeners can be directly attached to these behavior objects. The first argument of the event listeners are event objects, which provide useful data for event handling. The behavior objects are finally attached to selections by making use of the `Selection.call` function. `RespVis v3` makes use of the `d3-drag` package in the render routine of the parallel coordinates chart to enable dragging and dropping axes and interaction elements. The `d3-zoom` package is used in the render routines of all charts to enable zooming into axes with a numerical domain.

## 2.8 NodeJS

NodeJS [OpenJS 2024b] is a server-side runtime environment for JavaScript. With the advent of NodeJS, JavaScript became a viable alternative for backend (server-side) systems, making it possible to create full stack applications entirely in JavaScript. The first version of NodeJS adopted the CJS module pattern natively for code modularization. Nowadays, developers can freely choose between CJS and ESM. For more information about module formats, see Section 2.3.2.

Backend systems must be able to handle multiple requests concurrently. This is typically achieved by creating multiple threads, which was not possible in JavaScript for a long time. JavaScript is an asynchronous, event-driven language requiring an event loop to function properly. The event loop manages asynchronous operations and must be provided by runtime environments like NodeJS. NodeJS exploits the asynchronous nature of JavaScript to solve the concurrency problem by providing an API for loading off computational costly I/O tasks to the system kernel [Ostrowski 2023]. Since version 12, NodeJS is capable of running multiple threads in parallel using *worker threads*. These threads can be spawned by the main thread to offload computationally costly non-I/O tasks. Other features of the NodeJS runtime environment include built-in APIs for HTTP and file operations.

NodeJS comes with a default package manager called Node Package Manager (npm) [NPM 2024a], which manages the installation and sharing of dependencies for NodeJS applications. Today, npm still requires and runs on top of NodeJS, but its usage is no longer limited to NodeJS applications. The official npm registry [NPM 2023] comprised over 2.1 million packages in 2022 [OpenJS 2024a], making it the world's largest package system. It contains packages for countless use cases like CLI tooling, frontend libraries, backend service libraries, JavaScript utility libraries, and testing libraries.

Over recent years, competing package managers for the node ecosystem were released. The most prominent ones are Yarn [Yarn 2024] and pnpm [pnpm 2024]. These package managers are used similarly to npm, but may achieve better results in terms of performance, security, and disk efficiency.

## 2.9 Rollup

Rollup [Rollup 2024] is a JavaScript module bundler. The purpose of module bundlers is to convert complete JavaScript codebases, which may consist of an arbitrary number of files and have an arbitrary number of third-party dependencies, into single files called *bundles* [Gardón 2022]. This comes with many advantages.

First, bundlers greatly improve the performance of web applications. Complex JavaScript applications should be structured and organized in meaningful named directories and files to enhance the development workflow. However, using many files in production requires browsers to send many requests, one request per file to be precise. Furthermore, the transmitted files will unnecessarily contain unused or dead code, increasing the load time of the scripts. Bundlers solve these problems by creating a dependency graph of a project, traversing these dependencies, and including only the actually needed functionality in a single bundle, which can be sent to clients via a single request. Rollup especially excels in dead code elimination, which is also called *tree shaking*. It is noteworthy, that tree shaking only works for ESM modules, which are discussed in Section 2.3.2.

The second feature coming with module bundlers is extensive control over different output formats. In RespVis v3, Rollup and its plugins are instructed to produce multiple bundles in different module formats. Furthermore, all bundles are created once with third-party dependencies included, and once without third-party dependencies. This is especially useful for large libraries, since it allows library users to choose which parts of the library they need for their projects.

## 2.10 Gulp

Gulp [Gulp 2024] is a task runner, which automates complex procedures by defining public and private tasks in plain JavaScript or TypeScript, in a file called `gulpfile.js` or `gulpfile.ts` (the Gulpfile). Tasks are declared as asynchronous functions and can be composed to be executed in both serial or parallel. Private tasks can be imported by the Gulpfile, which makes it possible to outsource private tasks to separate files to achieve a higher degree of readability. A public task has to be exported from the main Gulpfile and can be invoked via the command line using the `gulp` and `gulp-cli` packages. Gulp uses NodeJS as its runtime environment. Typical use cases of Gulp include the efficient automation of file operations, orchestration of bundling tools like Rollup, and setting up development servers for live editing of code.

## 2.11 Storybook

Storybook [Storybook 2024a] is an open-source frontend tool for developing, testing, and documenting UI components. This is achieved by implementing so-called *stories*, where each story describes exactly one state of a UI component. Storybook can be installed using a node package manager, as discussed in Section 2.8. Storybook can be used in projects using popular frameworks like React, Angular, and Vue, but also works with plain HTML and JavaScript. The Storybook application itself is built using React [Meta 2024] and can be served locally via command line or built as static files, which can be served from a web server without further adjustment. A Storybook application is structured in two UI components: the Manager and the Preview [Storybook 2024b]. While the Manager component is responsible for rendering the UI components inherent to the Storybook application, the Preview component is an isolated iframe, rendering the stories defined in the project.

```
1 @media only screen and (max-width: 40rem) {  
2   // styling for narrow screens  
3 }  
4  
5 @media only screen and (min-width: 40rem) and (max-width: 60rem) {  
6   // styling for medium screens  
7 }  
8  
9 @media only screen and (min-width: 60rem) {  
10  // styling for wide screens  
11 }
```

**Listing 2.2:** Media queries for three viewport sizes, with breakpoints at 40 and 60 rem.

By default, Storybook finds all stories inside a project automatically by searching for files with the ending `*.stories.ts`. Storybook also provides the possibility to include Markdown JSX (MDX) files [MDX 2024], which are markdown files capable of interpreting JavaScript XML (JSX). JSX was introduced by Meta for writing components in XML-like syntax in JavaScript [Meta 2022]. Using MDX files in Storybook makes it possible to write documentation with embedded React components. This allows for great flexibility when composing documentation out of texts and interactive elements like charts.

## 2.12 Responsive Web Design

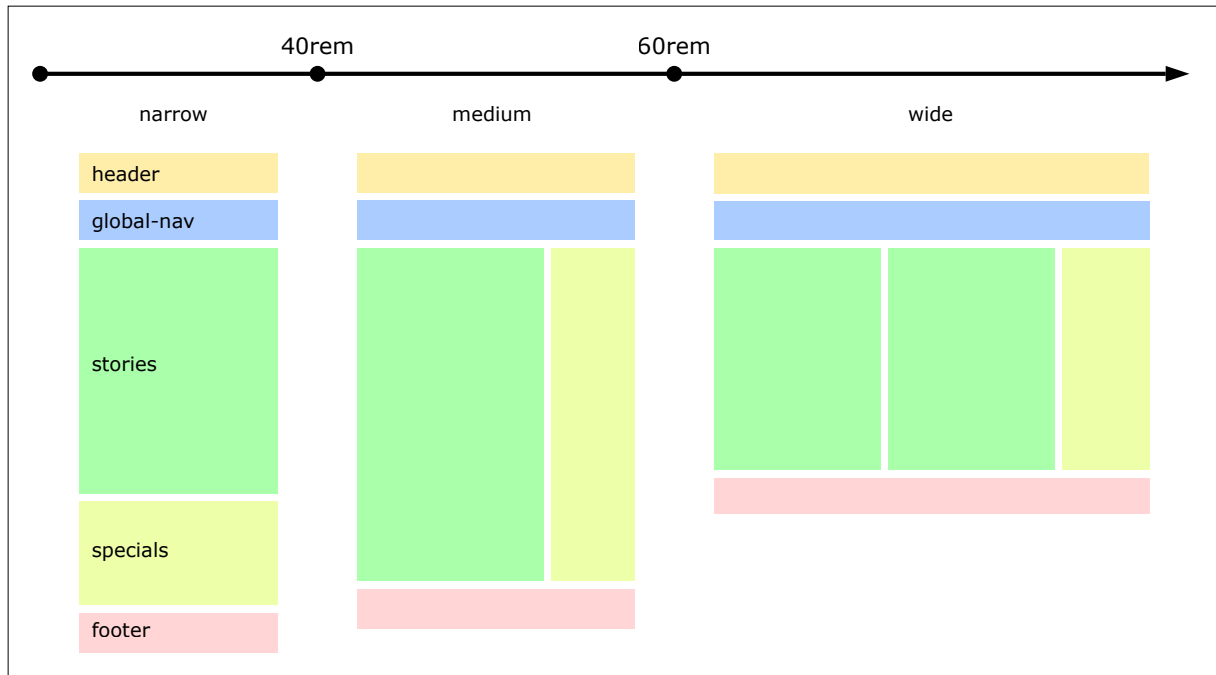
In 2010, Marcotte [2010] published his article entitled “Responsive Web Design”, which was the catalyst for this new design approach to web sites and applications. The intention was to design only once for all kinds of devices, and to serve the same code from the same URL to all devices. A year later, Marcotte [2011] expanded on the article with a book of the same title.

Marcotte described three core technical concepts for responsive web design:

- *Flexible Grids*: The width and height of container elements should be defined in relative CSS units such as % and em, rather than absolute units such as px. Font sizes should also be specified in relative units such as em or rem, depending on whether the font sizes should cascade or not.
- *Flexible Images*: Images and other media elements should adapt their size to the available space, for example by specifying `max-width: 100%`;
- *Media Queries*: Media queries were introduced in CSS3 and allow selected styles to be applied if certain conditions are met. In most cases, such conditions query the current viewport width, although other factors like aspect ratio, screen orientation, or screen resolution can also be queried. Listing 2.2 shows media queries for three layout widths.

Media queries with viewport widths are often used to define layout breakpoints, at which the screen layout changes. For example, as shown in Figure 2.3, setting layout breakpoints at `40rem` and `60rem` defines three layout widths: narrow, medium, and wide. This could be used to implement a one-column layout for narrow screens (`<40rem`), a two-column layout for medium width screens (between `40rem` and `60rem`), and a three-column layout for wide screens (`>60rem`).

For a web page or application, the *viewport* is the area of the page or application visible to the user [W3Schools 2023]. The viewport depends on the display properties of the device and the screen real estate used by the web browser, as well as on the size of the browser window. Paddings and margins sometimes have to be reduced on narrower screens, and font sizes may have to be adjusted to the current viewport too. Some elements may have to be reordered or removed, depending on the initial structure



**Figure 2.3:** A responsive breakpoint diagram. Setting layout breakpoints at, say, 40 rem and 60 rem provides for three different layout widths: narrow, medium, and wide. The layout scales smoothly between breakpoints and changes at a breakpoint. [Used with kind permission of Keith Andrews.]

of the user interface. Wider screens may also pose problems, like elements becoming too large or lines of text becoming too long. This is especially problematic for paragraphs of text, since lines longer than around 75 characters are harder to read [Rendle 2019].

### 2.12.1 Responsive Design Strategies

In the early days of the web, developers designed applications only for desktops and laptops. Therefore, it is not surprising that when web browsers became available for mobile devices, the first strategy was to adapt a design for a desktop device to also fit a mobile device. This design strategy is known as *desktop-first* design.

As the number of mobile devices out-shipped the number of desktop and laptop devices in 2010, a rethinking of the design strategy started. Web designers began to design their applications for narrower screens first, later ensuring that the design responded well to make use of the extra space available on wider devices. This strategy became known as *mobile-first* design [Wroblewski 2011].

The increasing usage of the mobile web led to a strategy which went even further, with the objective to solely design for mobile devices, so-called *mobile-only* design. This strategy can make sense for projects which are clearly dominated by mobile devices, like location-based services, but may be detrimental to user experience [Budiu and Pernice 2016].

Perhaps the best strategy is to define a number of logical layout widths (say narrow, medium, and wide) using layout breakpoints like those in Figure 2.3, and to design for all of them in parallel. A good name for it would be *everything-in-parallel* design.

### 2.12.2 Modern Responsive Design

Over ten years have passed since the term responsive web design was first used [Marcotte 2010]. Naturally, the web continued to evolve and new tools for creating responsive web applications were created. In his online article, Shadeed [2023] discusses the current state of responsive web design in 2023. The core message is to reduce the number of media queries and instead use more recent CSS layout techniques like Flexbox [MDN 2023a] and Grid [MDN 2023b], and viewport units [MDN 2023c]:

- The Flexbox property `flex-wrap` can be used to make elements fill up the available space, and realign themselves automatically into additional rows if space becomes too narrow.
- CSS Grid layout allows elements to be placed in a 2d grid, with sizing according to a variety of criteria, such as auto-resizing for columns, for example:

```
.grid {
  grid-template-columns: repeat(auto-fit, minmax(10rem, 1fr));
}
```

Here, the columns will always have a minimum width of `10rem`. If the grid container grows enough to fit another column without breaking the `10rem` constraint, the number of columns will automatically increase. A more detailed explanation is given by Soueidan [2017].

Another possibility is to explicitly assign template areas for specific named elements, like this:

```
grid-template-areas: "a a a"
                    "b c c"
                    "b c c";
```

- With the introduction of CSS comparison functions, another tool for increasing the fluidity of layouts became available. For example, the `clamp` function allows font sizing to fluidly change between a minimum and maximum size:

```
h1 {
  font-size: clamp(2rem, 2rem + 0.5vw, 3rem);
}
```

- Size container queries are similar to media queries, but refer to the size of the *parent* element (container), rather than the viewport. This is often what is wanted, so size container queries can be expected to replace media queries for responsive layout. Container query units (`cqw`, `cqh`) work like viewport units (`vw`, `vh`), but are relative to the corresponding parent container.
- Style container queries allow the current styling of a container element to be queried, so as to conditionally apply styling to its contents.

At the time of writing, all the above CSS features are supported by all modern web browsers, according to Deveria [2024], except for style container queries which are still being standardized.

### 2.12.3 Avoiding Horizontal Scrolling

In his blog post, Juviler [2021] explains why it is generally a bad idea to encounter horizontal scrolling in a web application:

- Since the beginning of the web, the convention was and is to scroll vertically through a web application, not horizontally. Changing this rule would result in a higher cognitive workload for the user.
- Many users might simply not notice that horizontal scrolling was possible (discoverability).

- Vertical scrolling is easy to do with a mouse wheel, horizontal scrolling, on the other hand, is not. Similarly, users of mobile phones have a higher range of motion to scroll vertically than horizontally.

For these reasons, it is generally advisable to avoid horizontal scrolling in web development and to deal with issues of fitting content into narrower widths in other ways.





## Chapter 3

# Responsive Visualization

In essence, responsive visualization addresses the process of designing visualizations capable of adapting to different device characteristics, both in terms of screen space and interaction modalities. Designing for smaller displays, such as those on mobile devices, can be particularly challenging.

### 3.1 Information Visualization

The field of visualization covers all types of visualization, which seek to present abstract information in a human-readable, visual way by offloading cognitive work to the human visual perception system, which is able to process patterns and differences in certain visual attributes unconsciously in parallel, instead of consciously in series. These attributes include, for example, color hue, color intensity, orientation, size, shape, focus and blur, and many more [Andrews 2024, Chapter 2; Ware 2021, Chapter 2]. The visual representation is, however, only half the story. The provision of interactivity is equally important to enable a viewer or analyst to explore the visualization.

The broader field of visualization can be broken down into three main subfields: scientific visualization (SciVis), geographic visualization (GeoVis), and information visualization (InfoVis). Whereas the visual representation in SciVis and GeoVis is largely *given* by the underlying concrete objects and coordinates data, information visualization deals with abstract information structures, necessitating that an appropriate visual representation be carefully *chosen* by the visualization author or the analyst. In recent years, the term data visualization (DataVis) has become popular; it can be thought of as encompassing both InfoVis and GeoVis.

The abstract information structures and spaces covered by information visualization include [Andrews 2024, Chapter 1]:

- *Linear*: Sequentially ordered information like table rows, lines of code, and lists.
- *Hierarchical*: Tree-like information structures like folders and subfolders, library catalogs, etc.
- *Networks*: Graph structures of nodes connected by links, for example social networks, web pages, and underground maps.
- *Multidimensional*: Tabular (spreadsheet) data comprising rows of records and columns of dimensions.
- *Feature Spaces*: Features extracted from a collection of objects to represent them in a high-dimensional space.

Some of the most prominent visualization types come from classic statistical charts such as line chart, scatter plot, and bar chart. These charts are easy to read, independent of a person's experience with

information visualization. Other charts, like parallel coordinates [Inselberg 2009], are more complex and often used by experienced analysts to explore larger multidimensional datasets.

## 3.2 Mobile Visualization

Horak et al. [2021, pages 37-40] explain in detail the factors which must be considered when designing for mobile devices such as phones, tablets, and smartwatches:

- *Usage factors*: Usage factors describe the impact of a user's posture when using a device as well as the position of the device itself. This is very different from desktop devices, which are operated typically by a person sitting in front of the device, resulting in fixed posture and position of the device.
- *Environmental factors*: Environmental factors represent surrounding influences, which may have impact on the user's perception and/or interaction capabilities. A noisy bus ride may prevent a user from catching audio messages or force the user to use one hand to hold on. Lighting conditions can influence how a visualization is perceived.
- *Data factors*: Data factors embody the difficulty to render visualizations for large datasets on devices with limited available screen sizes and/or computational power.
- *Human factors*: Human factors stand for the individual motivation, background knowledge, attention span, goals, and subjective preferences a user may have. Depending on those factors, completely different aspects of a visualization may be of interest to the user.
- *Device factors*: Device factors include screen size and interaction modalities. Issues like the fat-finger-problem [Horak et al. 2021, page 38] on touch devices have to be resolved for a visualization to become truly responsive.

Many of these factors are also relevant to responsive visualization.

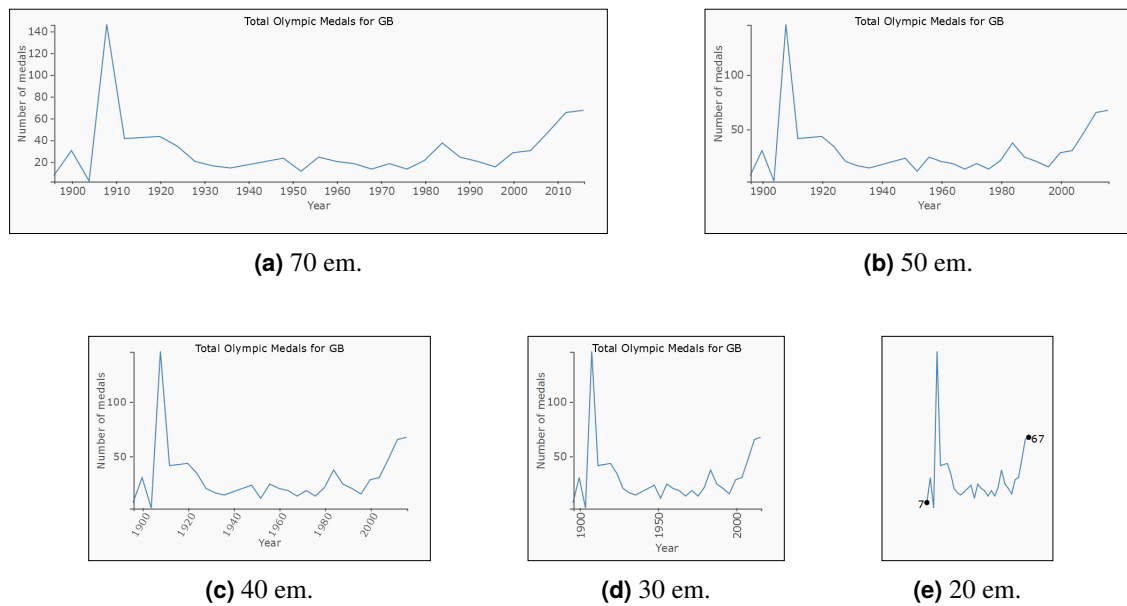
## 3.3 Display Properties

The size of a screen can be measured in two ways, either in number of pixels or in physical size. When referring to the physical size of a screen, the correct term is *physical size* or *screen size*, which is typically stated in centimeters or inches. When referring to the number of pixels, the correct term is *display resolution*, or simply *resolution* [Christensson 2019]. Typically, resolution is stated as *width*×*height*, where width is the number of horizontal pixels and height is the number of vertical pixels, for example 1920×1080 (Full HD).

*Pixel density* is a measure of detail, which sets the number of pixels in relation to the physical space. The corresponding unit is *pixels per inch* (ppi). The *aspect ratio* defines the relation between width and height of a screen, for example 16:9.

## 3.4 Responsive Visualization

Responsive visualization seeks to design visualizations capable of adapting to the characteristics of the display device, including display size, orientation, and interaction modalities. A common strategy is to first design a visualization for wider viewports and then apply responsive transformations to adapt the visualization to also fit narrower ones [Kim et al. 2021]. Such transformations often include reducing or removing non-essential information in order for the visualization to take up less space. However, the author has to be careful not to change the original wider version in a way that communicates a different message. An example of a responsive line chart can be seen in Figure 3.1.



**Figure 3.1:** A responsive line chart at various widths. As the chart adapts to smaller widths, tick marks are thinned out and tick labels are rotated. Finally, the chart becomes a sparkline. [Extracted from Figure 1 of Andrews [2018b]. Used with kind permission of Keith Andrews.]

The simplest approach to creating responsive visualizations is to simply scale the whole visualization down for smaller screen sizes. However, if downscaling is excessive, this approach becomes problematic as certain components become unreadable. This means simply shrinking the size of a chart alone will not be sufficient to create a good visualization for a smaller viewport. As Kim et al. [2021] explain, the key challenge when scaling down visualizations is the density-message trade-off. They distinguish three types of challenge in this context:

- *Graphical Density Challenges:* Visualizations often include many smaller elements like axis ticks, points, lines, labels, etc. When a visualization shrinks, these elements can only shrink to a certain extent. Otherwise, users would not be able to read labels or differentiate between similar-looking elements like data points with different radii. However, if elements do not shrink with the visualization, this will inevitably lead to overlapping elements. A typical approach for avoiding such scenarios is to thin out elements in a way that preserves the original message, but does not show too many elements.
- *Layout Challenges:* Visualizations are often built from a set of components. If enough width is available, it makes sense to place the legend, say, to the right of the chart. If less width is available, it might make sense to place the legend above or below the chart.
- *Interaction Complexity Challenges:* For the majority of interaction modalities on desktop devices, equivalent interactions exist for mobile devices. However, some interactions, like hovering or navigating by tabbing (pressing the Tab key), are only available when using a mouse or keyboard, respectively [Korduba et al. 2022, page 4]. Additional factors like the precision difference between a finger and a mouse pointer must also be taken into account when designing responsive visualizations.

An alternative approach is to pre-render multiple separate visualizations, applicable for specific viewport widths. This is typically achieved by first creating a base visualization and then applying responsive transformations at fixed-width breakpoints. The breakpoints should be chosen carefully to provide appealing visualizations for as many devices as possible. This method comes with the advantage of having stable, fixed visualizations for specified screen sizes, easing the integration of visualizations. Certain

visualizations can be prepared as raster graphics, if necessary due to performance constraints. Recently developed tools like Hoffswell’s visualization system [Hoffswell et al. 2020] support this approach. Modifications to one visualization can be propagated down to the others, and a simultaneous preview of all visualizations helps avoid inconsistencies. While using multiple pre-rendered visualizations has advantages, it breaks the principle of shipping a single codebase for all kinds of devices.

As discussed in Andrews [2018b, page 2], simply scaling a visualization down for smaller sizes does not create a truly responsive visualization. Instead, a responsive visualization is a single visualization capable of changing its layout structure at specific breakpoints and adapting its content automatically and fluidly in between breakpoints. Furthermore, support for a variety of input modalities should be provided, such as touch (tap, swipe, pinch zoom), keyboard, and mouse. When applying these concepts, the content of a visualization always perfectly fits the available space. Furthermore, transitioning between different visualization sizes can be controlled in a fine-grained manner by adapting only those entities and properties of a visualization, which need to be changed. The disadvantage of creating a single responsive visualization is the complex process behind it. Not all transformations are applied simultaneously and may influence each other. Furthermore, breakpoints will still be necessary to switch between different layouts at specific viewport or container widths.

### 3.5 Responsive Visualization Patterns

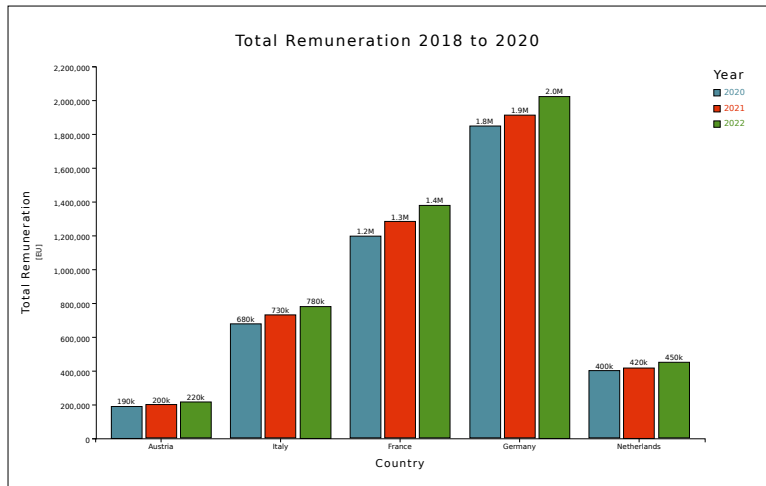
Kim et al. [2021] list 76 design strategies for transforming wider visualizations into narrower ones. Each strategy includes a target element and an executed action. Targets are grouped into five types: data, encoding, interaction, narrative, and references/layout, while actions are split into five kinds: recomposition, rescaling, transposition, reposition, and compensation.

Previously, Egger [2024a] curated a set of 16 tried-and-tested responsive visualization patterns; these are repeated here. The patterns are divided into three groups: *visual patterns* (10), *interaction patterns* (3) and *data patterns* (3). These patterns are generic, best practice examples, which can theoretically be applied by any arbitrary visualization system. However, since the web is the primary medium for consuming visualizations, all the patterns assume web-based visualizations. Practical examples of the patterns can be seen at the showcase web sites of Andrews [2018a] and Egger and Oberbauer [2024a].

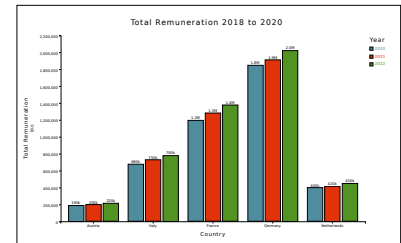
#### 3.5.1 Visual Patterns

Visual patterns are applied directly to visualizations in order to change the state and appearance of a visualization’s components to maximize the user experience depending on the available space:

- V1: Scaling Entire Chart Down
- V2: Repositioning Element Labels
- V3: Using Tooltips Instead of Element Labels
- V4: Rotating Axis Tick Labels
- V5: Shortening Labels and Titles
- V6: Scaling Labels Between Minimum and Maximum Size
- V7: Scaling Down Visual Elements
- V8: Hiding Elements and Labels
- V9: Rotating Chart 90°
- V10: Using a Different Chart



(a) Wide.



(b) Narrow: Entire chart simply scaled down.

**Figure 3.2:** V1: A chart can become unreadable when the entire chart is simply scaled down. [Images created with RespVis [Egger and Oberrauner 2024a] by the author of this thesis.]

### 3.5.1.1 V1: Scaling Entire Chart Down

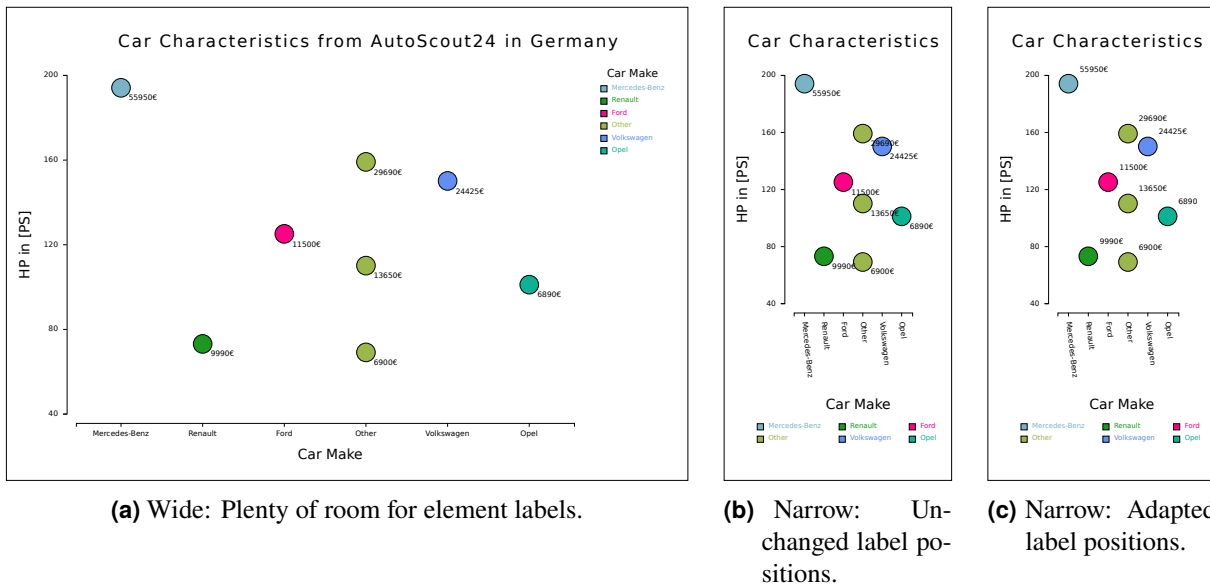
The most straightforward visual transformation is to scale the whole chart down. However, if downscaling is too excessive, this approach becomes problematic, as certain components become unreadable. While there is no fixed rule about a minimum font size, there is some consensus around the root font size being 16px or 12pt [PennState 2023]. The WCAG documentation also states that text should be scalable up to 200% of its original size [WCAG 2023]. Text labels in particular should not be scaled down below this tolerance. Similar issues arise when downscaling markers for data points; they can become too small to distinguish. Another problem occurs when downscaling a visualization in just one direction. This can lead to an unacceptable degree of distortion. As Andrews [2018b] states, it is not enough to just scale down a visualization to make it responsive. If it were, it would be the only pattern needed for achieving responsiveness. Figure 3.2 illustrates the problems which occur when only scaling down the entire visualization.

### 3.5.1.2 V2: Repositioning Element Labels

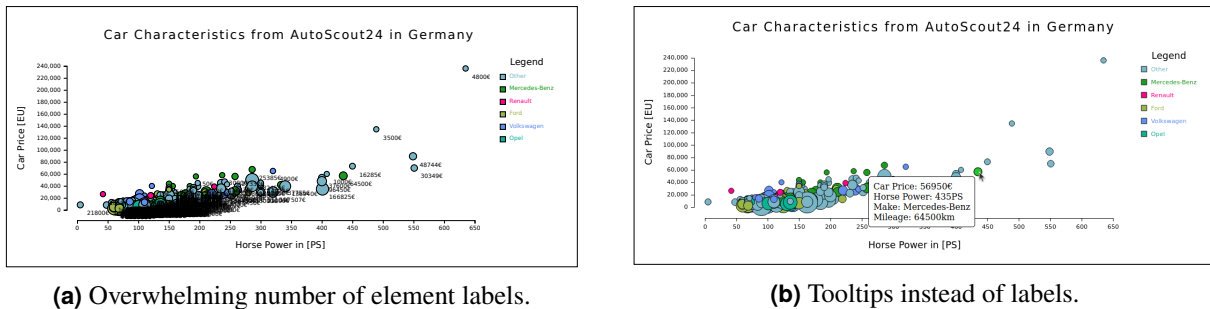
Repositioning of labels can help avoid intersections and clutter. In web-based visualizations, this can be achieved using media queries or container queries. Both of these are discussed in Section 2.12. Figure 3.3 illustrates how repositioning element labels can help avoid clutter in a visualization.

### 3.5.1.3 V3: Using Tooltips Instead of Element Labels

If a visualization contains many data points, but has only limited space, a helpful pattern is to hide the labels of all elements and display an element’s label upon hover or selection. In web-based visualizations, the CSS hover selector can be used to display a tooltip while hovering over an element with a pointer device. For touch devices, a single touch can be used to toggle the display of a tooltip. The disadvantage of this pattern is that it is not immediately obvious to users (discoverability). A practical example of the usage of tooltips can be seen in Figure 3.4.



**Figure 3.3:** V2: A scatter plot containing element labels. Clutter can lead to labels overlapping elements at narrow widths. Repositioning the element labels resolves the issue. [Images created with RespVis [Egger and Oberrauer 2024a] by the author of this thesis.]

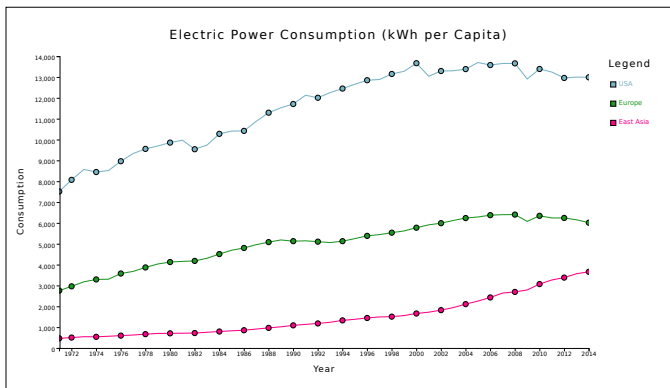


**Figure 3.4:** V3: If a scatter plot contains many data points in a narrow space, it makes sense to use tooltips instead of element labels. [Images created with RespVis [Egger and Oberrauer 2024a] by the author of this thesis.]

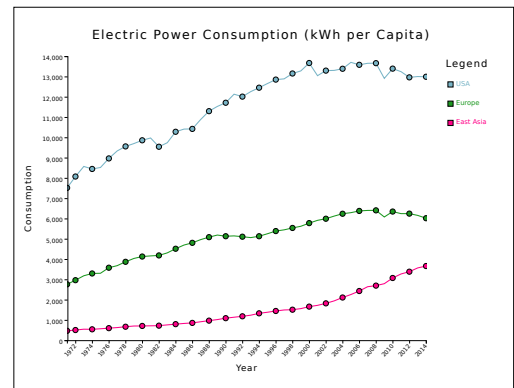
### 3.5.1.4 V4: Rotating Axis Tick Labels

A useful method for avoiding overlapping axis labels is to rotate them. This is especially useful for x-axis tick labels. As the available horizontal space decreases, this approach helps preserve more of the original axis label information (rather than thinning out or shortening the axis labels). The rotation of y-axis tick labels can also be considered, but is less useful.

For web-based visualizations, a possible way to achieve rotating tick labels is to use a combination of JavaScript and CSS. A JavaScript algorithm and event listeners are necessary to detect the currently appropriate angles of labels, while the styling of the labels themselves can be achieved via the CSS properties `rotate` or `transform: rotate()`. The advantage of this pattern is the preservation of the original axis information. On the downside, the axis labels are harder to read, since the natural reading direction is not retained. A practical example of a chart making use of rotating x-axis tick labels can be seen in Figure 3.5.

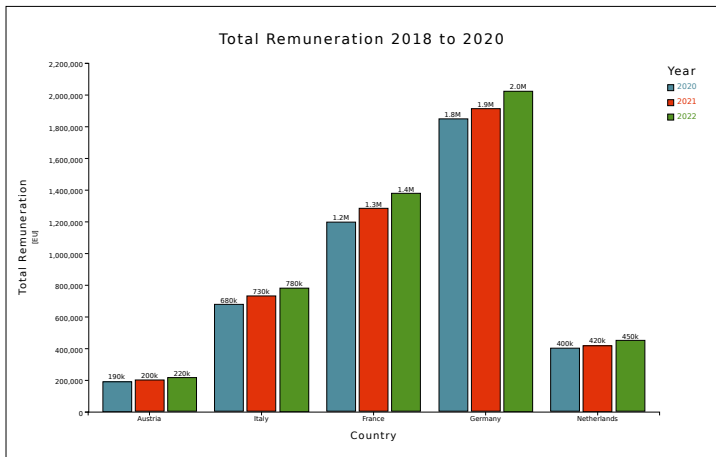


(a) Wide: Room for horizontal x-axis labels.

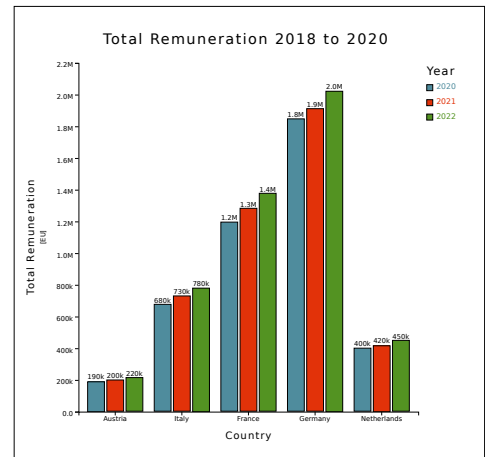


(b) Narrow: Rotated x-axis labels.

**Figure 3.5:** V4: A multi-line chart which avoids overlapping x-axis tick labels at narrow widths by rotating the labels. [Images created with RespVis [Egger and Oberrauner 2024a] by the author of this thesis.]



(a) Wide: Original y-axis labels.



(b) Narrow: Shortened y-axis labels.

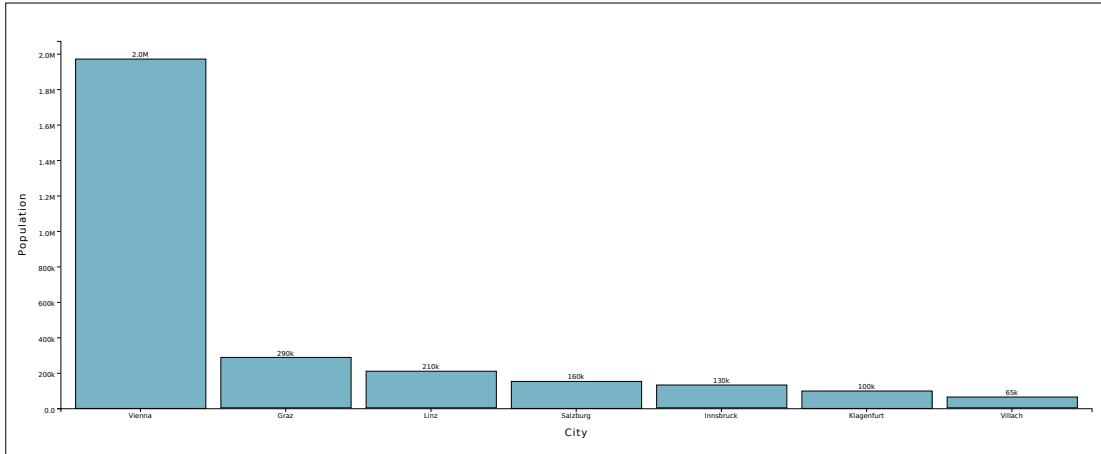
**Figure 3.6:** V5: A grouped bar chart with label shortening applied to the y-axis tick labels. [Images created with RespVis [Egger and Oberrauner 2024a] by the author of this thesis.]

### 3.5.1.5 V5: Shortening Labels and Titles

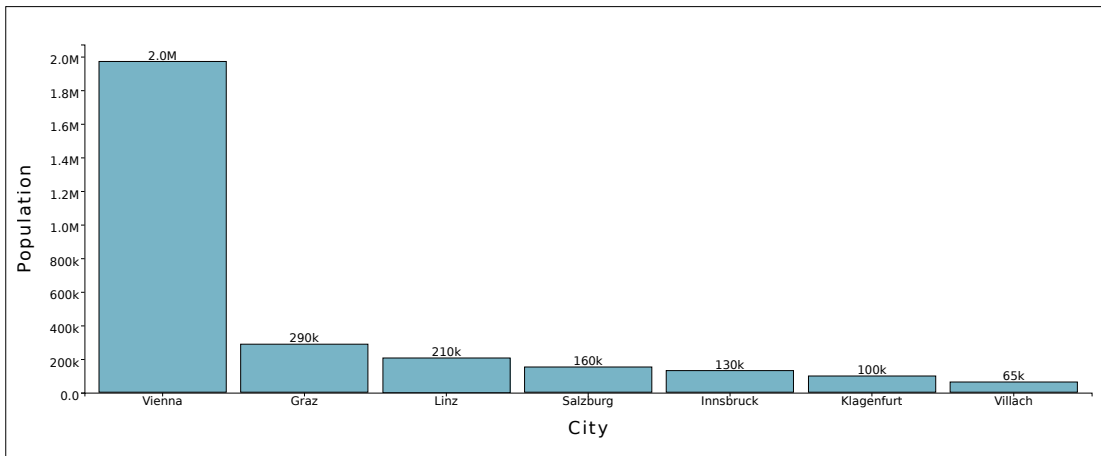
A common technique for avoiding clutter and overlaps is to have different formats for labels and different lengths for titles for different space requirements. Numbers, for example, can be shown in full length if enough space is available (e.g. 2,200,000), but shortened to well-known abbreviated forms (e.g. 2.2M) when space is limited. Similar strategies can be applied to other types of text such as dates, organization names, and geographic locations. However, when using shortened labels in a visualization, any resulting information loss, such as that caused by the rounding of numbers, should also be considered. Shortened texts should still be understandable, so as not to confuse users. Figure 3.6 demonstrates how the technique can be applied in practice.

### 3.5.1.6 V6: Scaling Labels Between Minimum and Maximum Size

Having different font sizes for different space requirements can be a useful method not only for increasing the readability of labels, but also improving the overall aesthetics of a visualization. Having larger font



(a) Fixed-size (small) unscaled labels.



(b) Scaled labels.

**Figure 3.7:** V6: A bar chart with scaled labels. When a visualization has the available space, it makes sense to enlarge the font size of the labels and titles. [Images created with RespVis [Egger and Oberrauner 2024a] by the author of this thesis.]

sizes where space allows looks much better than choosing the easier, safe option of always having the same smaller font size, as can be seen in Figure 3.7.

For web-based visualizations, there are two approaches to implement this transformation. The first relies on media or container queries and defines fixed font sizes for different breakpoints. The other uses CSS comparison functions to fluidly transition between smaller and larger font sizes. The second approach can also be combined with container query units to relate the font sizes to the size of the visualization container. This method relies heavily on modern CSS techniques, which are discussed in Section 2.12.2.

### 3.5.1.7 V7: Scaling Down Visual Elements

Scaling is a transformation applicable at multiple levels and in multiple variants. One form of scaling is scaling down selected visual elements in the chart in one or both directions. A bar element in a bar chart, for example, can be scaled in both horizontal and vertical directions without problems. An example can be seen in Figure 3.6. The width of the bars of the grouped bar chart becomes smoothly narrower as less space is available.



The same holds for lines in a line chart, since these simply have to update their thickness and target points. A marker for a data point is more complicated, since it must retain its aspect ratio during scaling to avoid distortion. The same holds for any other marker elements. Other types of visualization, like pie charts, chord diagrams, and maps, must retain their aspect ratio when scaling down.

The advantages of this technique are that much space can be saved without information loss and that smooth transitions via event listeners appear very natural. On the downside, the pattern is only applicable to visualizations with a manageable number of elements, since otherwise elements are already quite small even at larger widths. Another disadvantage is that line elements in line charts appear steeper at narrower widths and flatter at larger widths, affecting the perception of the original message.

#### **3.5.1.8 V8: Hiding Elements and Labels**

One possibility to adapt a visualization to narrower widths is to remove some elements or labels completely. When applying this technique, care must be taken to not alter the original message of the visualization. The advantage of this pattern is that an arbitrary amount of space can be saved by removing enough elements. However, this comes at the cost of information loss with respect to all the removed elements. When removing whole categories or dimensions, it is advisable to offer interactive possibilities, so the user can choose which dimensions or categories are of interest. This is described in more detail in Section 3.5.2.2. A practical example of hiding labels can be seen in Figure 3.4, which demonstrates how visible labels can be replaced with tooltips.

#### **3.5.1.9 V9: Rotating Chart 90°**

Transposing or rotating a chart by 90° can be a convenient way to align the dimension which requires more space vertically rather than horizontally. Vertical scrolling is much more acceptable than horizontal scrolling. An example of a rotated grouped bar chart can be seen in Figure 3.8. The advantage of this pattern is that no information is lost by the transformation process. The main disadvantage is the major change of the visualization which may affect other ongoing transformations.

#### **3.5.1.10 V10: Using a Different Chart**

At some point, the best option may be to completely swap a visualization for a different one. This technique is the last resort, when other techniques either lead to unacceptable information loss, alteration of the original message, distortion of the visualization, or unreadable elements or labels. The advantage of this technique is that it provides a solution where all others do not. On the downside, maintaining two different visualizations is more effort.

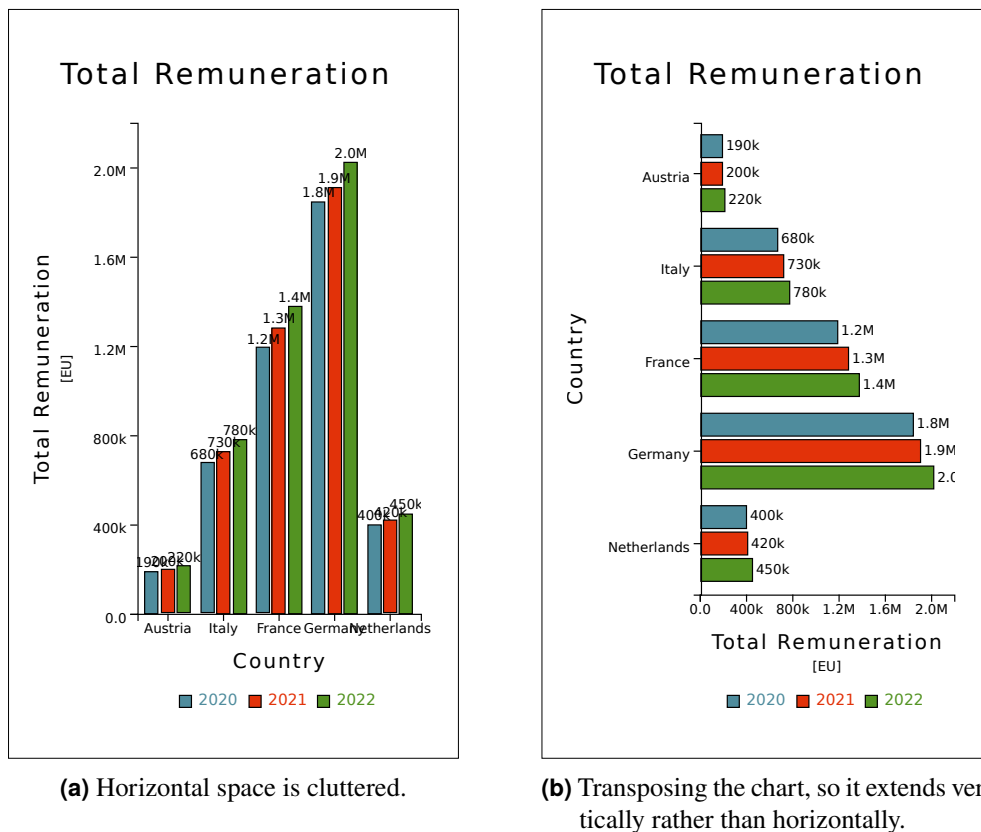
### **3.5.2 Interaction Patterns**

Interaction patterns support responsiveness by providing interactive functionality such as zooming and filtering:

- I1: Providing a Toolbar or Menu
- I2: Filtering Dimensions and Records
- I3: Supporting Zooming

#### **3.5.2.1 I1: Providing a Toolbar or Menu**

Interactivity bound to visual elements, such as hovering or a right-click context menu, suffers from poor discoverability. The user has to know such actions are possible or discover them by trial and error. A toolbar or menu, on the other hand, is visible to users, and its features can be explored. Typical actions



**Figure 3.8:** V9: Rotating a grouped bar chart by  $90^\circ$  to make better use of vertical space. [Images created with RespVis [Egger and Oberrauner 2024a] by the author of this thesis.]

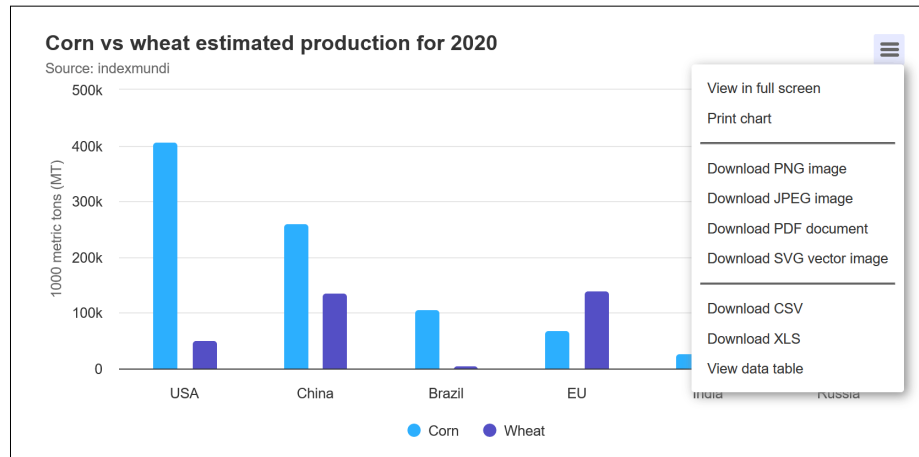
provided by toolbars or menus include being able to download a chart as SVG, download the data as CSV, view the chart in full screen, view the data as a table, and show and hide specific records and dimensions in the data.

The advantages of this pattern are the theoretically unlimited interaction options that can be added to a visualization and the high likelihood of the toolbar being discovered by the user. Disadvantages of the pattern include the space needed for the additional control elements and the effort for the user to find them if they are hidden by default. A practical example of a menu from Highcharts is shown in Figure 3.9. The toolbar provided by Plotly.js can be seen in Figure 3.10.

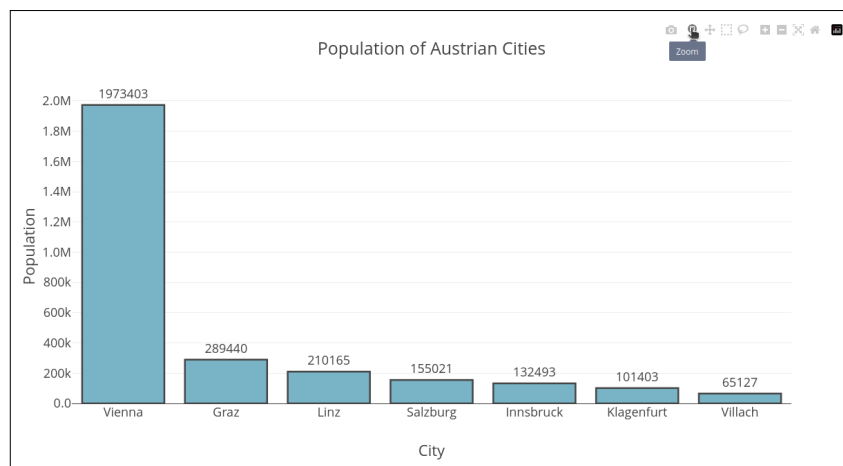
### 3.5.2.2 I2: Filtering Dimensions and Records

If space requirements are very tight, there is often no other solution than removing information from a visualization. However, when doing so it is a good idea to empower the user to choose which dimensions or records should be shown or hidden. The user may not be able to see all the data at once, but still has access to all information if necessary.

Possible interaction elements for the filtering of data can be the legend of a chart, the elements themselves, or separate control elements such as dropdown menus. Figure 3.11 shows the filtering of records from a grouped bar chart.



**Figure 3.9:** I1: The menu provided by Highcharts. [Image created with Highcharts [Highsoft 2023] by Keith Andrews and used with kind permission.]

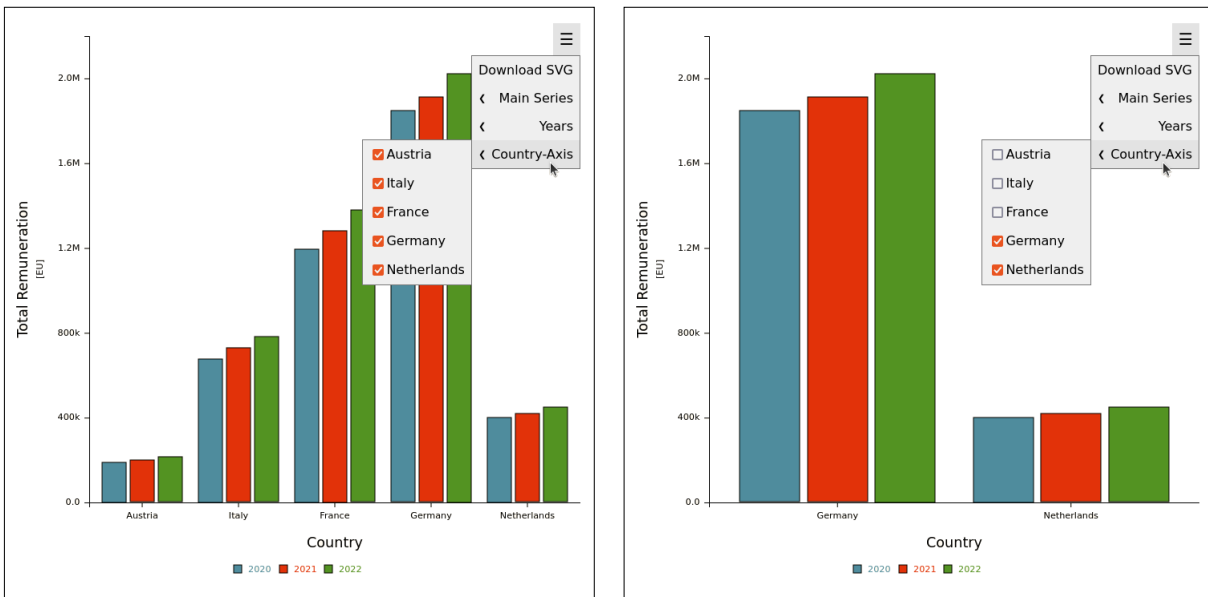


**Figure 3.10:** I1: The toolbar provided by Plotly.js is in the top right. [Image created with Plotly.js [Plotly 2023] by the author of this survey.]

### 3.5.2.3 I3: Supporting Zooming

Zooming is a crucial tool for overcoming the problems of limited resolutions and narrow screens. The standard approach, *geometric zoom*, allows a user to control the magnification of a visualization, and thereby trade the space needed for irrelevant information for more space for areas of interest [InfoVis:Wiki 2006]. The scatter plot example presented by Egger and Oberrauner [2023b] demonstrates perfectly how this technique can be combined with grabbing and panning to make data points more accessible. Figure 3.12 shows how zooming can be used to solve some of the problems associated with dense data and intersecting elements.

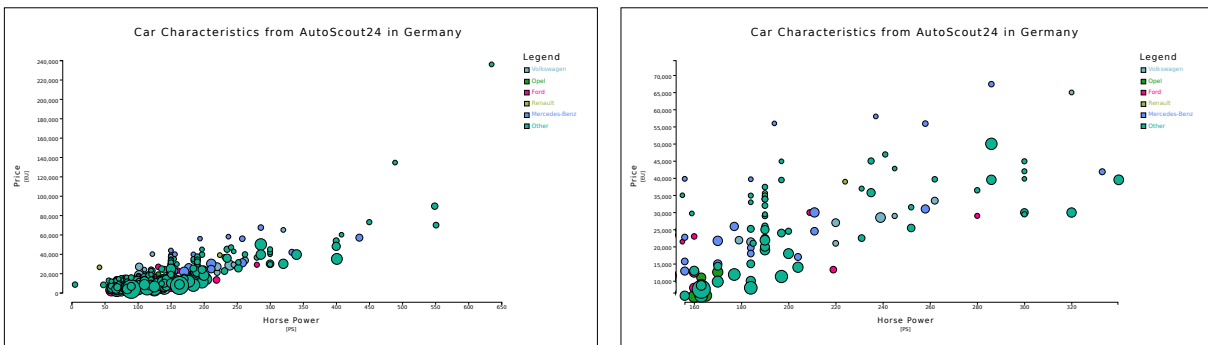
In *fisheye zoom*, the focus area is magnified and the surrounding context area is reduced, like using a magnifying glass to view the chart or visualization. Instead of removing the context completely, it is distorted so as to take up less space. For cartesian visualizations, which have perpendicular axes such as *x* and *y*, *cartesian zoom* can be used. This technique divides the chart into a grid of cells and distorts the size of the cells such that more interesting cells are enlarged to show more detail, while surrounding cells are made smaller. Fisheye zoom and cartesian zoom are described in detail by Sarkar and Brown [1992], interactive examples can be explored in the responsive scatter plot example by Andrews [2018a].



(a) All five countries are visible.

(b) Only two of five countries are visible.

**Figure 3.11: I2:** A grouped bar chart where countries or years can be filtered with a control menu. [Images created with RespVis [Egger and Oberrauner 2024a] by the author of this thesis.]



(a) Unzoomed.

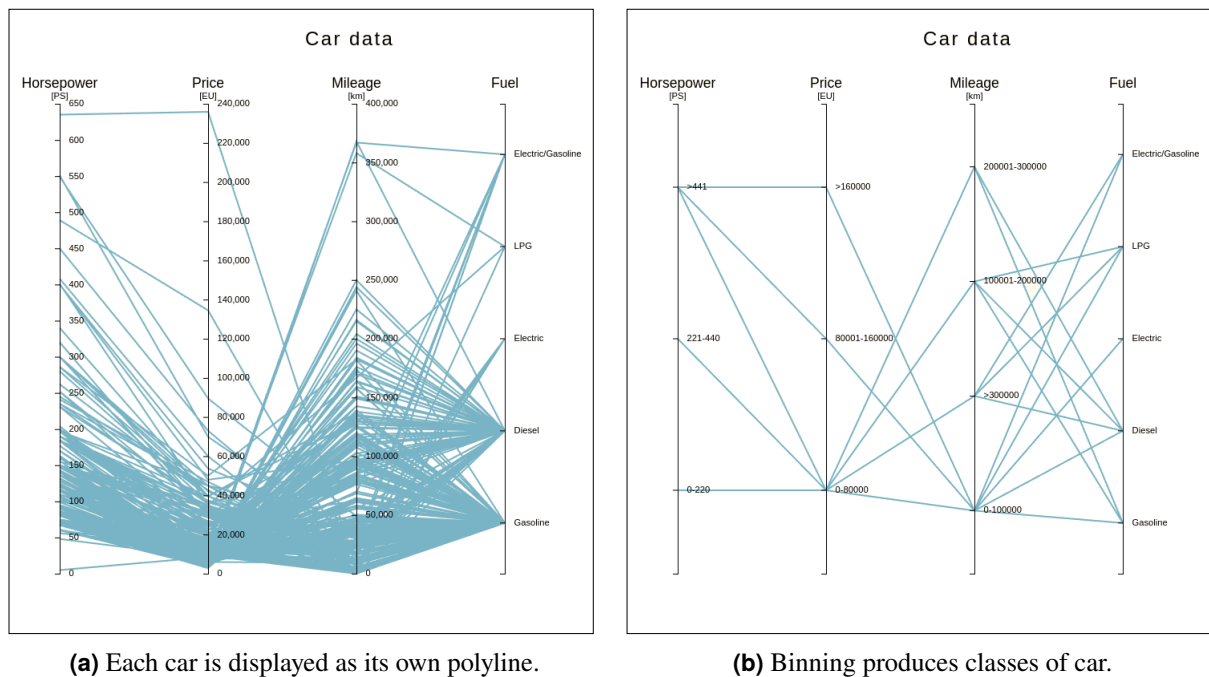
(b) Geometric zooming.

**Figure 3.12: I3:** A scatter plot with 500 data points representing cars. There are many overlapping data points. To enable the user to inspect all points, geometric zooming is supported. [Images created with RespVis [Egger and Oberrauner 2024a] by the author of this thesis.]

Another approach is called *semantic zoom* [Bederson and Hollan 1995]. This form of zooming does not simply change the sizes of items, but considers which items to display and how to display them. To apply the technique, a form of data structuring is needed to create different levels of detail. Depending on the current zoom factor, elements can be removed, split into sub-elements, or change size or shape [InfoVis:Wiki 2014].

### 3.5.3 Data Patterns

The applicability of visual patterns is highly dependent on the size of the dataset and the chosen visualization type. In many cases, it is necessary to group and transform the original data to obtain new datasets and statistics, which can be visualized more easily. Such data patterns include:



**Figure 3.13:** D1: A parallel coordinates chart with binning applied to the first three dimensions. [Images created with RespVis [Egger and Oberrauner 2024a] by the author of this thesis.]

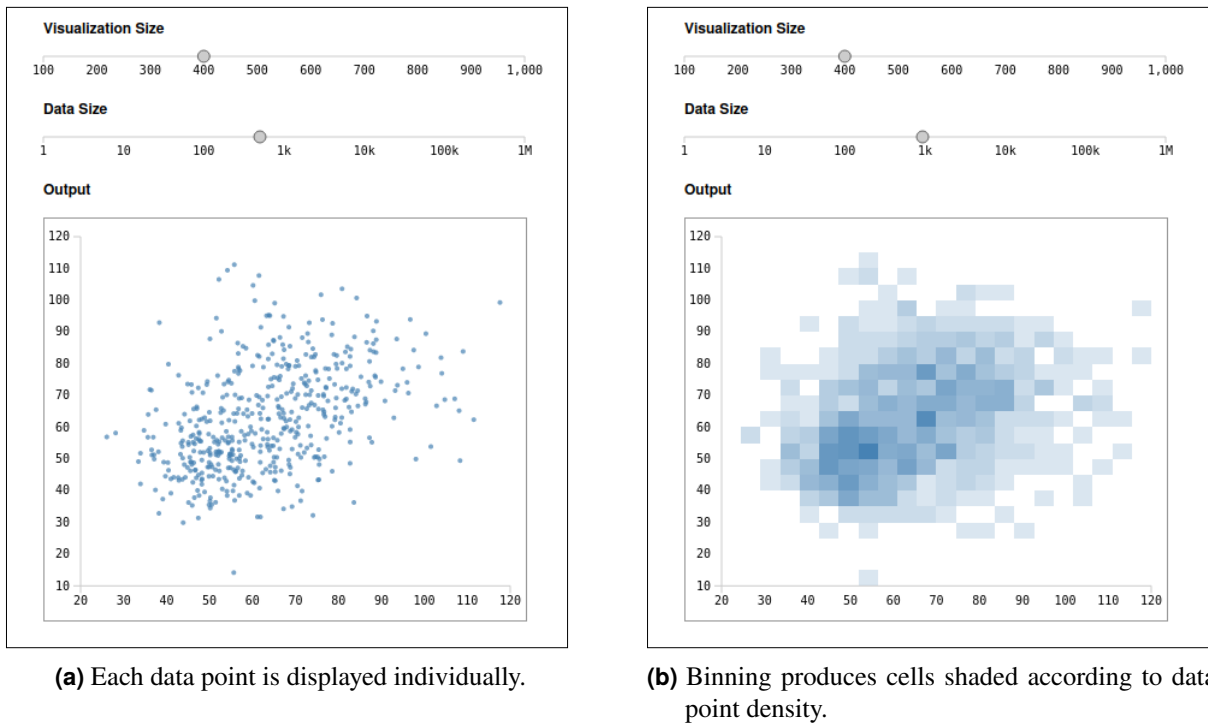
- D1: Data Generalization
- D2: Data Aggregation
- D3: Data Clustering
- D4: Data Sampling

### 3.5.3.1 D1: Data Generalization

Data generalization combines many data points into manageable groups. For example, individual ages can be binned into age ranges [Satori 2022]. The approach allows information to be presented in a more compact way, consuming less space and not overwhelming the user with too many data points.

An example can be seen in Figure 3.13, which shows a parallel coordinates chart about used cars in Germany. In Figure 3.13a, each car is displayed as its own polyline. After binning the records along the first three dimensions, classes of cars have been created, allowing Figure 3.13b to show a less cluttered view on the data.

Rabinowitz [2014] shows another way of how data generalization can be used to create responsive visualizations. He created an interactive online prototype of a scatter plot, where the user can select how much space is available to the visualization and how many data points should be included. The visualization transforms the scatter plot into a heatmap if the density of the data points exceeds a certain threshold. The resulting heatmap is a generalized version of the scatter plot, in that it bins data points into a grid with the cell coloring indicating the data point density, as shown in Figure 3.14.



**Figure 3.14:** D1: A scatter plot transforms into a heatmap if a certain threshold of data point density is exceeded. [Images created with Rabinowitz' prototype [Rabinowitz 2014] by the author of this survey.]

### 3.5.3.2 D2: Data Aggregation

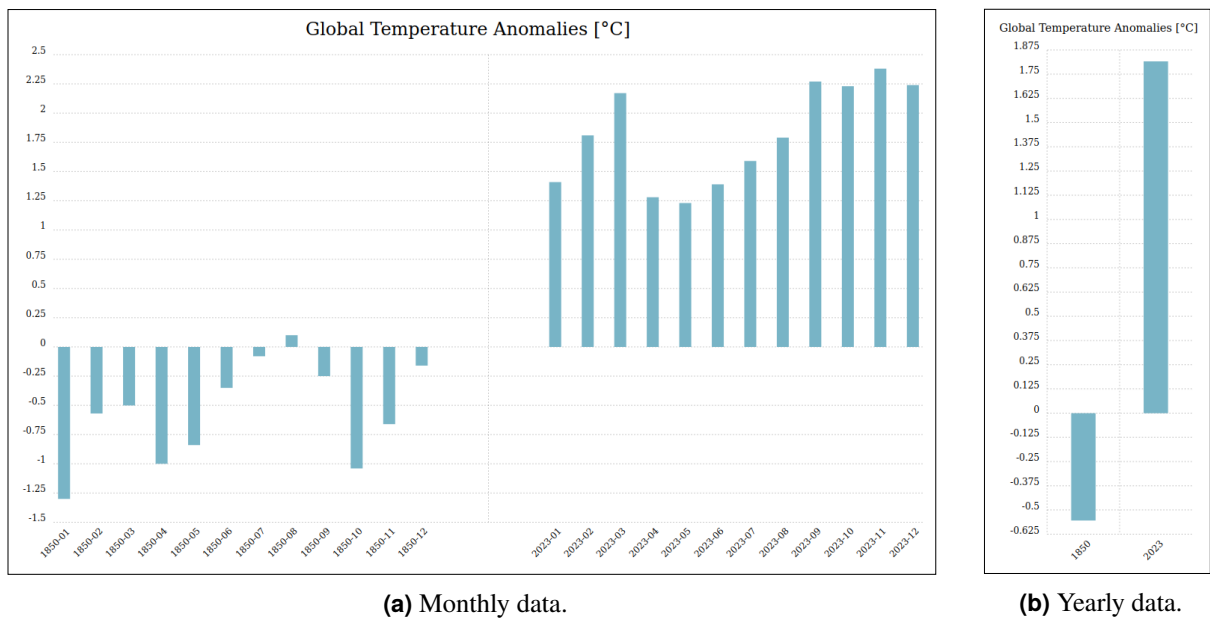
Data aggregation is a method used to summarize the information from a collection of data points into one or multiple useful statistics, such as average or sum [Zanini 2023]. When creating responsive visualizations, it allows the presentation of large amounts of underlying data as single visualization elements.

Figure 3.15 shows the global surface temperature anomalies of the years 1850 and 2023, i.e. the difference in global surface temperature in those years compared to the average temperature in the 100 years from 1901 to 2000 [NCEI 2024]. The bar chart in Figure 3.15a shows the data for 1850 and 2023 in monthly intervals. The bar chart in Figure 3.15b aggregates the monthly data into two bars, one for each of the two years. This version of the chart is much more compact, but the original message of a significant rise in average global surface temperature between the years 1850 and 2023 is preserved.

### 3.5.3.3 D3: Data Clustering

Clustering groups objects (records) based on their similarity. The most common clustering algorithms include hierarchical clustering, k-means clustering, model-based clustering, density-based clustering, and fuzzy clustering [Sarah 2023].

For responsive visualizations, clustering algorithms enable new options for dealing with large datasets, since they allow the abstraction of similar data points within a dataset. One example is the use of agglomerative hierarchical clustering to replace overlapping data points with a cluster element, with interactions such as displaying cluster information and toggling presentation as a cluster versus individual data points.



**Figure 3.15:** D2: Two versions of a bar chart with different time aggregation intervals. [Images created with Chartist [Kunz 2017] by the author of this survey.]

### 3.5.3.4 D4: Data Sampling

Data sampling is a technique typically used in statistical analysis to identify patterns and trends in a population by extracting, processing, and analyzing a representative sample of an overall population [Egnyte 2022]. It can also be used to improve the responsiveness and performance of visualizations by avoiding overplotting. Probabilistic sampling has the objective of creating samples which represent the overall population as accurately as possible. Different types of probabilistic sampling include random sampling, stratified sampling, cluster sampling, and systematic sampling. Non-probabilistic sampling techniques are less rigorously representative and include convenience sampling, quota sampling, snowball sampling, and purposive sampling.

A good example of how data sampling can be used to create responsive visualizations can be seen in Figure 3.12. The scatter plots show a random sample of 500 cars from the original 46,405 cars in the dataset [Ander 2021].





## Chapter 4

# RespVis v1 and RespVis v2

The original vision for RespVis v1 was to create an open-source library for creating responsive visualizations as an extension of the well-known D3 visualization library [Bostock 2024h]. The main focus was to reduce the effort necessary for users to create responsive charts. RespVis v2 included improvements in the development workflow, deployment, and first drafts of features now part of the current RespVis v3. RespVis v2 was presented at the 27<sup>th</sup> International Conference on Information Visualisation (IV 2023) [Andrews et al. 2023].

### 4.1 RespVis v1

The two main contributions of RespVis v1 were the implementation of example patterns for responsive visualization and the inclusion of a novel layout engine. The layout engine allows components of an SVG chart to be positioned responsively using standard CSS layout mechanisms like Flexbox and Grid.

With RespVis v1, a chart creator can choose from a collection of simple visualization patterns up to more complex visualizations like complete charts [Oberrauner 2022b]. RespVis v1 provides the functionality to create the following complete charts:

- Bar chart.
- Grouped bar chart.
- Stacked bar chart.
- Single-series line chart.
- Multi-series line chart.
- Point chart (scatter plot).

RespVis v1 is available in a separate GitHub repository [Oberrauner 2022a], and includes at least one example of each mentioned chart type. The necessary HTML code, styles, and scripts to render one chart example are all included in the same file. In addition to the HTML files, the directory `data/` contains various example datasets, and the directory `vendor/` contains the D3 library, which is a dependency of RespVis. The RespVis v1 library is essentially distributed as two monolithic files: `respvis.js` and `respvis.css`.

The task runner Gulp, discussed in Section 2.10, was already used in RespVis v1 to automate common tasks and workflows for developers working on the library. Gulp uses the bundling tool Rollup, discussed in Section 2.9, to bundle all TypeScript functions into a single JavaScript library file. Gulp also handles the removal, renaming, and movement of files, and is used to set up file watchers to provide a smooth development experience.

Following the convention of D3, only functions were used in RespVis v1; no classes were used. The intention was to separate data from code and provide the functionality to create small, reusable, components for the composition of complete responsive charts. While this approach was a good starting point for RespVis, it came with many practical drawbacks, which are discussed in Section 5.2.

## 4.2 RespVis v2

The contributions of RespVis v2 included improvements to the development environment with Gulp, the addition of a deployment workflow for the chart examples, improvements to the library, and improvements to the chart examples.

With RespVis v2, the chart examples became self-contained, meaning a single example chart could be deployed by simply pushing it to a web server without further adjustment. This was achieved by adapting the directory structure of the project and improving the existing Gulpfile. In RespVis v1, all Gulp tasks were located in the single main Gulpfile `gulpfile.js`. With RespVis v2, all private tasks were moved to separate files to improve readability. Furthermore, tasks were introduced for generating all required libraries and data dependencies into the chart example directories when serving or building the chart examples.

To make the self-contained chart examples broadly accessible, it was decided to deploy them to a publicly available web site. In fact, two versions of the RespVis v2 chart examples were automatically deployed: the current stable version and the current development version. Since the RespVis repository is hosted on GitHub, GitHub Actions were implemented to achieve this task. The cloud-computing platform Netlify [Netlify 2024] was chosen to host the example chart sites, since it is free and provides sufficient resources.

RespVis v2 made four major contributions to the library:

1. Four new features were added to the scatter plot: multiple categories, pinch zoom, bubble charts, and sequential color encoding.
2. A first prototype of the parallel coordinates chart was implemented.
3. A first cut of how to define breakpoints and responsive properties for RespVis visualizations was implemented.
4. A first version of rotating x-axis labels was implemented.

All of these features have been improved significantly with RespVis v3.

Another major part of RespVis v2 was the improvement and maintenance of the existing chart examples. The following were performed for each of the example charts:

- Refactoring of example code.
- Maintenance of stylesheets.
- Adapting and swapping out datasets, and specifying their sources.
- Data cleaning for some datasets.
- Introduction of container queries.
- Adapting chart title and subtitle for various chart sizes.

## Chapter 5

# RespVis v3

RespVis is an open-source visualization library for creating responsive visualizations as SVGs [Egger 2024j]. The library is built on top of D3 [Bostock 2024h], a powerful low-level JavaScript visualization library, which differs from traditional charting libraries, since no API for creating complete visualizations is provided. D3 draws to the web page by dynamically injecting SVG nodes into the DOM (SVG-DOM). A visualization is composed in D3 by assembling a collection of marks and modifying them through channels to respond to data. As well as drawing, D3 provides a wide range of utility functions, for example to read and write CSV files, construct appropriate axis tick labels, and format strings from dates and times.

This way of creating charts comes with great power and flexibility, but also has disadvantages. D3 can be very challenging to learn, especially for beginners in the field of data visualization. Furthermore, the creation of complete visualizations with D3 can require much time and effort, since everything must be composed of the basic marks [SciChart 2024].

For this reason, RespVis v3 was designed to ease the process of authoring responsive charts, providing a better developer experience for chart creators, independent of their individual level of expertise. In contrast to earlier versions of RespVis, high-level visualization components can be created simply via class instantiation and passing single configuration objects, hiding away the complexity of D3's API, and taking care of setting up RespVis' layout and render mechanisms. The layout mechanism is a core feature of RespVis and allows chart creators to apply powerful CSS layout techniques to SVG elements, which is generally not possible.

The main improvements made to RespVis in RespVis v3 include:

- The library was refactored and restructured to make code reusable and understandable. The goal was to provide a clear, uniform API, including the definition of strict type arguments to guide chart creators during the development of charts.
- RespVis charts were improved to provide a single method for defining all executed render routines. Chart creators can override this method to completely change the desired render behavior, without changing the expected arguments and the validation process.
- CSS variables for spacing, transition times, colors, and font sizing were added.
- All inner `<svg>` elements were removed and replaced with `<g>` elements, except for imported SVG interaction elements.
- The layout of charts was changed to include padding containers and a `<clipPath>` element, which are controlled using dedicated CSS Variables. One can make use of these CSS variables to solve the problem of overflowing content.

- Existing conflicts of the RespVis layouter mechanism with CSS-only layout changes on hovering were resolved by letting the Layouter recompute the layout on hover interactions.
- The Layouter component was improved to allow alternating between the SVG standard layout system and RespVis' novel CSS layout mechanism within nested SVG elements.
- A uniform way of handling the alignment of SVG `<text>` elements was introduced. The new approach makes it easy to avoid inconsistent spacing, since `<text>` elements and their replicated counterparts occupy the same place and their baselines are aligned.
- Layout breakpoints and CSS layout variables were introduced, to improve the reusability of charts.
- The axes of cartesian charts were improved to support top and right orientations.
- The rotation of axis labels was improved to work for all axis positions (top, bottom, left, right) and both dimensions (width, height) out of the box.
- TypeScript mixins were used to extract reusable functionality shared by groups of charts.
- Category filtering was added for all charts, categorical axes, and categorical legend entries.
- Numerical filtering was added for all charts and numerical axes.
- Zooming was added for all charts and numerical axes.
- Zooming in cartesian charts was improved to allow for zooming into a single axis, in case of one numerical and one categorical axis.
- Flipping was added to all charts. Additional flip options are available for axes to give precise control over axes in different orientations.
- A unified approach of creating labels for marker primitives was added.
- The origin line and a configurable grid were added to cartesian charts.
- Support for time scales was added.
- Dragging and dropping was established for axes in parallel coordinates charts and additional interactive filter elements were added, supporting filtering of data records.
- Cursor icons were added for parallel coordinates charts. The Gulp `genSVGDataURI` task was introduced to easily convert all SVG icons in a directory into data URIs, which can be used directly in CSS.
- RespVis' `Toolbar` component was completely redesigned to provide a better user experience. It provides additional tools and new options for previously existing tools.
- All JavaScript Gulp files were rewritten to TypeScript.
- Separate modes for starting Gulp in development or production were introduced.
- A Gulp task was added for compiling the TypeScript files used in the self-contained examples.
- RespVis v2 previously used SCSS to manipulate and merge CSS. Since most of the functionality provided by SCSS is now provided directly in CSS, SCSS was removed. A new Gulp step was added for merging multiple CSS files to a single one, maintaining readability during development, but providing single CSS files for library users.
- The Gulp `build` task was improved to create both standalone and dependency-based bundles.

- The Gulp `build` task was improved to create multiple sub-packages, which are published on the npm registry.
- Extensive documentation was added using Storybook. Custom documentation components were added to provide uniformly structured live documentation with both code and interactive examples.

## 5.1 Project Structure

RespVis v3 comes with an increased number of files and directories, which made it necessary to adapt the project structure accordingly. The current top-level directory structure of the project can be seen in Listing 5.1.

The top-level directory `respvis/` contains configuration and metadata files. The `README.md` file contains information about RespVis in general, lists the contributors of the project, and provides links to more detailed documentation. The `CHANGELOG.md` file lists all notable changes to the project. The `LICENSE` file contains the current license of the project, which is an MIT license. The `package.json` [NPM 2024b] contains important metadata about the project, like the name, current version, and dependencies. This metadata is interpreted by package managers of the Node ecosystem like npm [NPM 2024a] to install dependencies, or to publish a project in the npm registry [NPM 2023]. The `tsconfig.json` file configures how TypeScript is used in the project; it includes options for the TypeScript compiler and which files should be included in the compilation process. The `gulpfile.ts` file contains all public gulp tasks of the project. These tasks include bundling the RespVis library and developing locally via a live server. Private utility tasks are defined in the `gulp-tasks/` directory.

The directories located in the `src/` directory contain the source code of RespVis' sub-packages (`packages/`), the assets used in the sub-packages (`assets/`), the self-contained examples (`examples/`), the data used in the self-contained examples (`data/`), the external libraries used in the self-contained examples (`libs/`), and the source code of the live documentation (`storybook/`). The `declarations/` directory contains additional type rules for the D3 selections and transitions APIs and specifies how non-TypeScript files are imported via Gulp based on their file endings. The `dist/` and `package/` directories are created as a result of executing the Gulp `build` task and contain the compiled version of the self-contained examples and the full monolithic package of RespVis respectively.

### 5.1.1 Package Structure

In the original design of RespVis v1, the whole source code of the library was kept in a single repository, with one CSS file containing the default styles, one `package.json` file at the root directory and multiple top level directories containing semantically related TypeScript code of the library, written in ES module format. The Gulp `build` task discussed in Section 5.1.2 created one bundle of the whole library, forcing chart creators to unnecessarily bloat their bundle sizes when importing the library.

To save chart creators from this disadvantage, the top-level directories were intended to represent sub-packages in future versions of RespVis. The reasoning behind this design choice was that in most cases only parts of RespVis' functionality need to be imported by library users [Oberrauner 2022b, page 51].

In RespVis, a sub-package is part of the whole RespVis library, but contains its own `package.json` file, TypeScript code, and CSS styles. The `build` task generates a separate bundle for each sub-package, making each sub-package capable of being published on the npm registry and imported independently of other sub-packages. The sub-packages are:

- `respvis-core` [Egger 2024e]: Provides core functionality of RespVis, which is always necessary when creating a visualization with RespVis. It also includes the custom layout mechanism.
- `respvis-tooltip` [Egger 2024i]: Provides the functionality to create tooltip components.

```

respvis/
├── declarations/
├── dist/ (generated)
├── gulp-tasks/
├── package/ (generated)
├── src/
│   ├── assets/
│   ├── data/
│   ├── examples/
│   ├── libs/
│   ├── packages/
│   │   ├── respvis-bar/
│   │   ├── respvis-cartesian/
│   │   ├── respvis-core/
│   │   ├── respvis-line/
│   │   ├── respvis-parcoord/
│   │   ├── respvis-point/
│   │   └── respvis-tooltip/
│   └── storybook/
├── CHANGELOG.md
├── gulpfile.ts
├── LICENSE
├── package.json
├── README.md
└── tsconfig.json

```

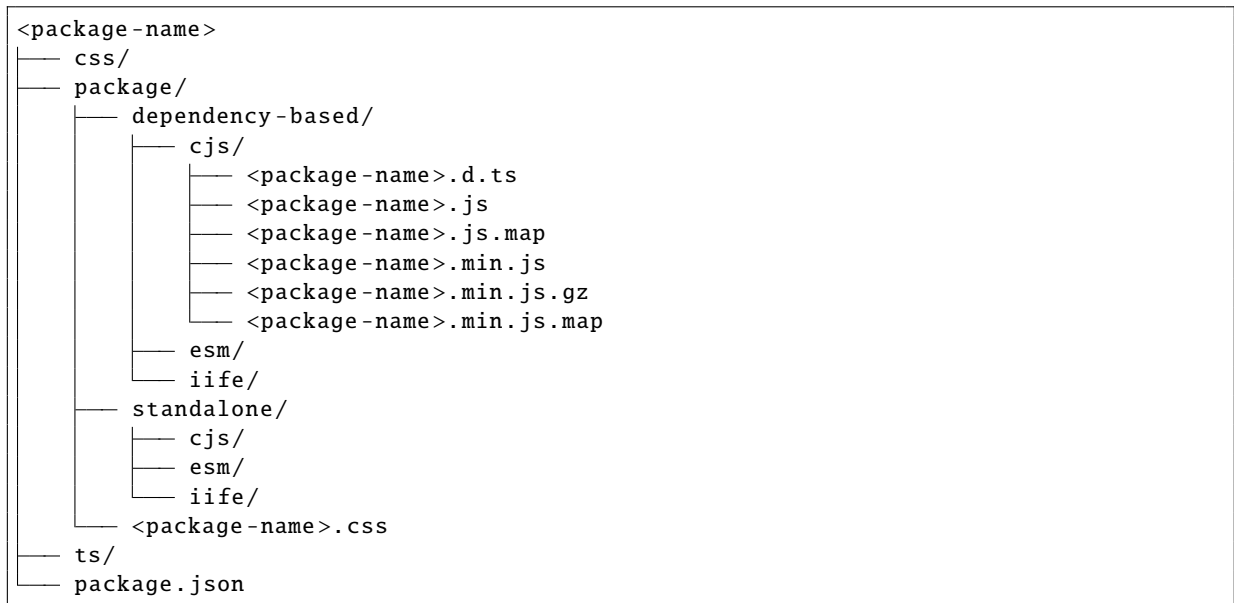
**Listing 5.1:** The top-level file and directory structure of the RespVis project.

- `respvis-cartesian` [Egger 2024d]: Provides the functionality to create cartesian components.
- `respvis-bar` [Egger 2024c]: Provides the functionality to create bar components.
- `respvis-point` [Egger 2024h]: Provides the functionality to create point components.
- `respvis-line` [Egger 2024f]: Provides the functionality to create line components.
- `respvis-parcoord` [Egger 2024g]: Provides the functionality to create parallel coordinates components.

To allow for the whole functionality of RespVis to be imported at once, the `build` task also creates a full monolithic bundle of RespVis, which can be published as a package called `respvis` using the top-level `package.json` file [Egger 2024b].

Introducing sub-packages would have been possible either by splitting up the library across multiple repositories, or by reworking the existing structure to conform to a modern monorepo architecture. The author of this thesis decided for the latter, since there is no large team working on RespVis, but typically only one or two students at a time. Therefore, using a monorepo approach instead of splitting up related code sections into their own repositories saves future developers much effort. The main advantages are the easy management of a single repository and consistency across the project [Woltmann 2024]. The monorepo approach made it possible to restructure the semantically related code directories from RespVis v1 into fully-fledged sub-packages located in the `respvis/src/packages/` directory.

The file and directory structure of a RespVis v3 sub-package is shown in Listing 5.2. The directories `css/` and `ts/` contain the styles and TypeScript source code of the sub-package. The `package.json` file contains metadata like the version and name of the package, and which files should be included when publishing. The `package/` directory is generated by the bundling process and contains the two subdirectories `dependency-based/` and `standalone/` and a stylesheet `<package-name>.css`, as follows:

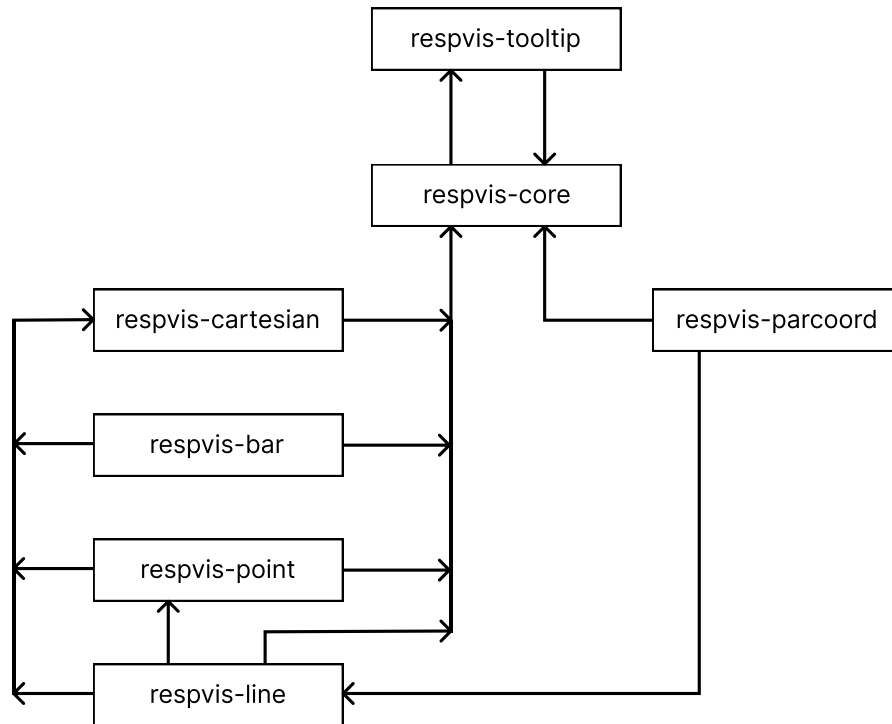


**Listing 5.2:** The file and directory structure of a RespVis v3 sub-package.

- Bundles located in the `dependency-based/` directory solely contain code and type information of the corresponding RespVis package. Therefore, when importing code from the dependency-based directory, one must make sure to install the peer dependencies of the package too. As explained in the npm docs [NPM 2024b], as of npm v7 peer dependencies are installed by default.
- The `standalone/` directory contains only bundles having all necessary dependencies included. It is not necessary to worry about peer-dependencies when importing from a standalone bundle. On the other hand, such bundles are substantially larger than their dependency-based counterparts.
- The `<package-name>.css` stylesheet includes all CSS code required to be imported if the corresponding dependency-based package is included in a project. If a standalone bundle is imported, this file should be ignored. The full stylesheet from the `respvis` package should be imported instead.

Generated bundles are not only grouped by dependency type, but are also available in three different module formats: ESM, IIFE, and CJS. The module formats are explained in Section 2.3.2. While ESM is the de facto standard nowadays, the CJS and IIFE bundles are included in the package for backward compatibility. When importing from a package, the `package.json` acts as a resolver. By default, it resolves to the dependency-based bundle in ESM format.

The internal dependencies between RespVis' sub-packages are shown in Figure 5.1. As can be seen, all sub-packages are dependent on `respvis-core`, which serves as the fundamental package. The sub-packages `respvis-bar`, `respvis-point` and `respvis-line` are dependent on `respvis-cartesian`, which provides the base functionality for creating series and charts defined by having two dimensions for values and scales. The `respvis-line` sub-package is dependent on `respvis-point`. This avoids duplication of similar render functions, since a Line Series can be seen as a polyline, i.e. a series of points connected by line segments. The `respvis-parcoord` sub-package is dependent on `respvis-line`, since one record of a parallel coordinates chart is visualized as a polyline connecting once on each axis. The only external dependency of RespVis is the visualization library D3 [Bostock 2024h].



**Figure 5.1:** The internal dependencies between RespVis sub-packages.[Image created by the author of this thesis.]

### 5.1.2 Gulp Tasks

In RespVis, the task runner Gulp (discussed in Section 2.10) is used to automate repeatable tasks like bundling the library, merging style sheets, and building the self-contained examples. In previous versions, all Gulp tasks were located in a single file called `gulpfile.js`. With RespVis v3, this file was changed to be a TypeScript file `gulpfile.ts`, which defines six public tasks. Private tasks, which can not directly be invoked via the command line, were moved into TypeScript files located in the `gulp-tasks/` directory, as can be seen in Listing 5.1. The public tasks can be invoked either by running Gulp via `npx` or by running the equivalent scripts defined in `package.json`:

- `npx gulp clean`: The `clean` task removes existing `package/` and `dist/` directories to enable a clean rebuild of the project.
- `npx gulp cleanExampleDeps`: The `cleanExampleDeps` task removes generated dependencies from the `src/examples/` directory.
- `npx gulp cleanAll`: The `cleanAll` task executes the `clean` task, `cleanExampleDeps` task, and additionally removes the `node_modules/` directory and the file `package-lock.json/` in order to enable a clean rebuild of the project including the re-installation of dependencies.
- `npx gulp build`: The `build` task first executes the `clean` task, then builds the whole library and copies the self-contained examples into the freshly created `dist/` directory. The `build` task bundles all sub-packages contained in RespVis, in addition to building the monolithic package. To build the library, two private tasks `bundleJs` and `bundleLibCSS` are executed in parallel.

The `bundleJs` task uses Rollup to generate bundles of all RespVis sub-packages and the monolithic package. The `bundleLibCSS` task is responsible for merging all style sheets located in a sub-package to a single style sheet. Both tasks are executed for all sub-packages and, again, also for the monolithic package. The files generated from the two private tasks are written to freshly generated `package/`



directories, which are ready to be published on npm. A generated `package/` directory is located at the top-level directory of the corresponding sub-package, or at the root directory for the monolithic bundle. After the execution of the two tasks, required dependencies are generated into `src/examples/`, so that the examples can be compiled and built.

- `npx gulp serve --dev`: The `serve` task generates only the monolithic package to save time on rebuilds. It additionally executes a private task called `watcher`, which has two responsibilities: First, the `browser-sync` package is used to initialize a live server serving the `respvis/dist/` directory. Then, file watchers are initiated, which automatically update `dist/` if changes are made in the `src/` directory and subsequently notify browsers to reload the page. The `serve` task can be invoked in production (`--prod`) or developer (`--dev`) mode. The default mode is production. The non-secret environment variables for production and development mode can be found in the files `.env.prod` and `.env.dev` respectively. The `serve` task only bundles the standalone form of `RespVis`, since this package is the only one needed for the self-contained examples. Omitting the bundling of the other packages saves much time during live development.
- `npx gulp genSVGDataURI`: The `genSVGDataURI` task converts SVG files for icons into data URIs which are placed in a text file `svg-uri-mapping.txt` inside a newly generated directory `gulp-util-generated/`. They can then be conveniently copied into a style sheet to define cursor shapes. The task has to be run manually by the developer; it is not run automatically by the build process.

### 5.1.3 Self-Contained Examples

The `RespVis` repository contains a curated set of self-contained examples. The source code for these examples is located in the directory `src/examples/`. When executing the `build` task via Gulp, as explained in Section 5.1.2, a new directory `dist/` is generated containing the built version of the examples. These examples are insofar special, as they are fully self-contained, meaning a single example can be deployed by simply pushing it onto a web server without further adjustment.

All chart examples, except the article example, follow the same file structure, which is illustrated in Listing 5.3. The `<chart>.css` file contains the code for styling the appearance of an example. The `<chart>.html` file contains the markup of an example. The `<chart>.ts` file provides a function responsible for rendering the desired chart. The function is called by a script defined in `<chart>.html`. Since `RespVis` is written in TypeScript, full type support is available when working on the chart render function. The Gulp `build` task compiles `<chart>.ts` to `<chart>.js`. The `<chart>/data/` directory contains the dataset required by the example along with a text file describing the source of the dataset. The `<chart>/libs/` directory contains the libraries required by the example. At the time of writing, this always includes the current full `RespVis` bundle and the `d3-7.6.0` bundle.

The article example differs slightly in structure, because it contains multiple charts and has additional layout rules, but it is also provided as a self-contained example, ready to deploy.

### 5.1.4 Live Documentation

The documentation introduced with `RespVis v3` is a major part of the new version. The frontend tool `Storybook`, which is discussed in Section 2.11, is used to create the documentation. To view the documentation, one can run `Storybook` locally or visit the hosted documentation of the latest version `RespVis` [Egger 2024j]. The documentation is called *live documentation*, because it provides many interactive chart examples, which provide the code necessary for rendering and can be interacted with. Part of the live documentation is shown in Figure 5.2.

All documentation files, are located in the `src/storybook/` directory. The most important files and directories are shown in Listing 5.4. The file `main.ts` contains the main configurations for `Storybook`, including static file paths, locations of stories, active addons, and the chosen build tool and its plugins.

```

<chart>
├── data/
│   ├── <dataset>.js (generated)
│   └── source.txt (generated)
├── libs/
│   ├── d3-7.6.0/ (generated)
│   └── respvis/ (generated)
├── <chart>.css
├── <chart>.html
└── <chart>.ts (<chart>.js)

```

**Listing 5.3:** The file and directory structure of RespVis' self-contained examples.

[How to create a Chart \(Advanced Setup\)](#)

[How to create a Chart \(Basic Setup\)](#)

[Basic](#)

[Labeled](#)

[With Date Values](#)

[With Responsive Title](#)

[With Responsive Chart](#)

## Improving Basic Chart

The `basic` chart misses fundamental information. A chart viewer will not be able to understand the underlying data at all. To change this, the chart must be titled and labeled accordingly:

```

...
const args: LineChartUserArgs = {
  series: {
    x: {
      values: years
    },
    y: {
      values: students
    }
  },
  title: 'Students Registered at TU Graz',
  subTitle: '',
  x: {
    title: 'Year',
    subTitle: '[2012 to 2021]',
  },
  y: {
    title: 'Students',
    subTitle: '[Winter Semester]',
  }
}
...

```

[Copy](#)

### Labeled

[JS Code](#) [Controls](#)

**Figure 5.2:** The RespVis live documentation includes code and interactive examples. [Screenshot taken by the author of this thesis.]

```
storybook/  
├── plugins/  
│   └── vite-plugin-svg-raw.ts  
├── static-assets/  
│   ├── png/  
│   │   ├── respvis-logo-light.png  
│   │   └── respvis-logo.png  
│   └── storybook-reset.css  
├── stories/  
│   ├── contributing/  
│   ├── top-level-mdx/  
│   ├── using-respvis/  
│   └── util/  
├── main.ts  
├── manager-head.html  
├── manager.ts  
├── preview.ts  
├── tsconfig.json  
└── vite-env.d.ts
```

**Listing 5.4:** The files and directories in RespVis’ live documentation, created using Storybook [Storybook 2024a].

The files `manager-head.html` and `manager.ts` are used to adjust the theme of the documentation and hide Storybook-specific settings, which have nothing to do with the RespVis library. The `preview.ts` file is used for importing global documentation example styles and to specify the order of items in the sidebar. The `tsconfig.json` extends the `tsconfig.json` of the root directory, with small adjustments in the configuration to get Storybook to work properly with TypeScript.

The live documentation is structured into three parts. The first part consists of top-level Markdown documents located in the `stories/top-level-mdx/` directory. The second part, located in the `stories/using-respvis/` directory, contains documentation, live demos, and guides about the usage of the RespVis library from the perspective of a chart creator. The third part, located in the directory `stories/contributing/`, is addressed to developers keen to learn more about the implementation details of RespVis and how to contribute.

One of the difficulties of setting up the documentation was how to avoid massive duplication when creating many similar chart examples. Although duplication could not be avoided entirely, it could at least be drastically reduced by creating reusable utilities. These utilities are located in the `stories/util/` directory.

## 5.2 Library Design

The RespVis v3 library is contained in the `src/packages/` directory and distributed among the sub-packages discussed in Section 5.1.1. The main objectives behind RespVis v3 were to improve readability and reusability of the code, provide a better, clearer API for chart creators, and to introduce new features for authoring responsive visualizations. To achieve these goals it was necessary to change RespVis’ structure.

The original idea for RespVis v1 was strongly inspired by D3. Therefore, no classes were used in RespVis v1, only functions. The intention was to separate data from code and provide the functionality to create reusable, small components for the composition of complete, responsive charts. While this approach was a good starting point for the next versions of RespVis, it came with a number of practical drawbacks.

First, creating charts with only functions results in much duplicated code, which is hard to maintain and keep consistent. Instead, it is advisable to use inheritance for this use case, an object-oriented mechanism to avoid duplication and share fundamental chart behavior across all chart types.

Furthermore, strictly decoupling data from functionality leads to heavy usage of if conditions and ternary operators in functions, which tremendously reduces the readability of the code. Instead, polymorphism should be used to create objects similar in structure but differing in behavior. The behavior of these objects can be invoked at runtime without knowing the internals of the objects, dramatically reducing the need for if conditions and ternary operators. Polymorphism helps also when aiming for a strict API, which is a main principle of RespVis v3. In RespVis v1, a chart creator is only loosely restricted in the choice of the passed arguments, since all arguments are optional. This leads to ambiguity about how a chart creator is expected to use RespVis' API. RespVis v3's API, on the contrary, exactly defines the allowed types for arguments and clearly describes which are optional and which are required. Polymorphic objects are used to wrap different types of data input. These objects can be used at runtime through a defined, shared interface.

Another problem with a pure functional approach is the increased difficulty in maintaining the state of a chart. With a growing number of chart features and interaction possibilities like filtering, zooming, and inversion there was an urgent need to establish a defined manner to update the chart state.

All of these problems eventually led to the decision to introduce object-oriented concepts and classes in RespVis v3. For further reading about the topic, see the discussion about object-oriented programming and functional programming written by Melkonyan [2023].

### 5.2.1 Naming Conventions

In contrast to the top-down naming convention applied by Oberrauner [2022b, page 40], RespVis v3 follows a different approach. Entities are named such that composed names sound natural and generally adhere to English conventions. The reasoning is that a person will have less difficulty trying to understand the internals of RespVis if the code is written in a way that seems natural to humans. As a concrete example, the function:

```
function chartCartesianAxesRender(
  selection: ChartCartesianSelection): void {...}
```

in RespVis v1 was replaced by:

```
function renderCartesianAxes<T extends CartesianChartSelection>(
  chartS: T) {...}
```

in RespVis v3. Both functions accomplish the same task. When looking at the current version, a reader immediately understands what the function accomplishes. It *renders cartesian axes*. From the parameter name, one can derive that the function needs a chart selection, where the axes will be rendered. Although the second function is a generic function, which is generally longer, it is shorter than its previous counterpart, because it is less repetitive and avoids redundant expressions like a void return value.

A developer who is unfamiliar with the internals of RespVis and looks at the previous version will begin to read the function and question whether the function is about cartesian charts or cartesian axes. Then, the person will reach the Render part and reread the whole expression, hopefully drawing the right conclusion. In the worst case, the developer does not understand the function naming and must look into the content of the functions to understand it. To avoid such scenarios, RespVis v3 changed its naming conventions. While there are always exceptions to the rule, in most cases the concepts shown in Table 5.1 are applied. These concepts are adapted from the work of Anichiti [2021].

For example, entities should have meaningful, natural, and descriptive names. The example in Table 5.1 shows the interfaces for a bar chart in RespVis v3 vs. RespVis v1. In RespVis v3, the user input

Rule	Good	Bad
Repetitive naming should be avoided.	<pre>type Component = {   title: string,   description: string }</pre>	<pre>type Component = {   componentTitle: string,   componentDescription: string }</pre>
Functions should always start with verbs.	<code>renderLegend()</code>	<code>legendRender()</code>
Entities should have meaningful, natural, and descriptive names.	<pre>interface BarChartUserArgs interface BarChartArgs interface BarChartData class BarChart</pre>	<code>interface ChartWindowBar</code>
Files and directories should be named using kebab case [MDN 2024e].	<code>label-series.ts</code>	<code>label_series.ts</code>
Functions and variables should be named using camel case [MDN 2024e].	<code>renderLegend()</code>	<code>RenderLegend()</code>
Classes, types, and interfaces should be named using pascal case [MDN 2024e].	<code>LegendUserArgs()</code>	<code>legendUserArgs()</code>

**Table 5.1:** Naming conventions in RespVis v3.

(`BarChartUserArgs`), function input (`BarChartArgs`), resulting data (`BarChartData`), and instantiable class (`BarChart`) are precisely defined, while in RespVis v1 there exists only one interface, `ChartWindowBar`, which is used to define all data input and output. This introduced ambiguity to the data input. Furthermore, the name `ChartWindowBar` does not adhere to English conventions and is difficult to read and understand.

## 5.2.2 Sub-Package Modules

All TypeScript modules of a sub-package are located in the `src/packages/<package-name>/ts/` directory, which itself may contain up to four directories: `constants/`, `data/`, `render/`, and `utilities/`, corresponding to the four types of aggregated module in RespVis:

- *Constant modules:* Constant modules provide constants, default values, error messages, and basic type definitions, which are used throughout the entire library.
- *Data modules:* Data modules expose clear and understandable interfaces for chart creators and provide the functionality to validate input in a safe and controlled manner, typically using factory functions or classes. The resulting validated data objects reliably provide information during the render phase of a chart. As well as creation, data modules may provide additional utilities to manipulate or retrieve data from these objects.
- *Render modules:* Render modules interpret data objects and subsequently render corresponding components. Render modules may, like data modules, include factory functions or classes to create corresponding data objects from the input of chart creators. If a render module does not provide such functionality, it is conceived for internal use and provides at least a type definition of the required

arguments for the render function.

- *Utility modules*: Utility modules provide reusable functions and types, which can be used in the entire code base. The use cases of utility modules comprise easing interactions with the DOM, D3, or JavaScript data structures, applying TypeScript mixins, providing geometrical types and functionality, and defining recurring mathematical formulas.

All RespVis packages include `render/` directories. The `respvis-core` and `respvis-point` packages have `data/` directories. The `respvis-core` package is the only package to include a `constants/` directory and a `utilities/` directory.

Each of the `constants/`, `data/`, `render/`, and `utilities/` directories implements a TypeScript aggregated module [MDN 2024d], by defining an `index.ts` file at the top-level. All exports are defined in this `index.ts` file, making it possible to import functionality from assorted files in a directory the same way as it is done with single files.

Most data modules and render modules validate user arguments during the validation phase of a chart, implementing an important concept to increase the developer experience of chart creators. They provide three type definitions (interfaces) and a factory function or class constructor to handle validation of the arguments. The three type definitions were conceived to exactly define the allowed user input at chart creation, the allowed function arguments of validation functions, and the resulting data objects used during the render phase of a chart:

- `<module-name>UserArgs`: Specifies the structure of the arguments passed by a chart creator.
- `<module-name>Args`: Extends the first type with additional arguments required by the validation function. The first and second type may be equal if no additional arguments are required by the validation function.
- `<module-name>` (or `<module-name>Data`): Specifies the output of the validation function or constructor. If a related class with the name `<module-name>` already exists, the suffix `Data` is appended. An example of this convention can be seen in Listing 5.5, which contains the validation logic for the chart module.

### 5.2.3 Component Hierarchy

In essence, a RespVis component is a cohesive unit of one or more elements rendered during the render phase of a chart. All code related to a component is located in a corresponding render module. Composite components provide dedicated render functions, which may include calls to render functions of other components. A primitive component, in contrast, does not provide a render function, but provides a type definition used in the render routines of composite components to create a series of primitive components by applying D3 data joins.

In most cases, RespVis components are bound to data objects via D3. These data objects originate from the input data passed by a chart creator at the instantiation of a chart. All charts expect two arguments: first, a D3 selection of a single empty HTML element, typically a `<div>` element, which later becomes the `Window` component, and second, a configuration object for adjusting the content and render process of the chart. A configuration object is a nested JavaScript object containing options for all configurable components of a chart. All passed options are validated in the chart constructor by calling the corresponding validation functions. This is called the validation phase of a chart.

The outcome of the validation phase is a single validated data object, which represents the current state of a chart and its underlying components and is bound to the `Window` component. The component hierarchy of RespVis v3 is illustrated in Figure 5.3. Once a chart transitions into the render phase, data is propagated down from top to bottom of the layout hierarchy in a controlled manner. Direct children of the `Window` component are the `Toolbar` component, which is bound to dedicated parts of the validated data,

```

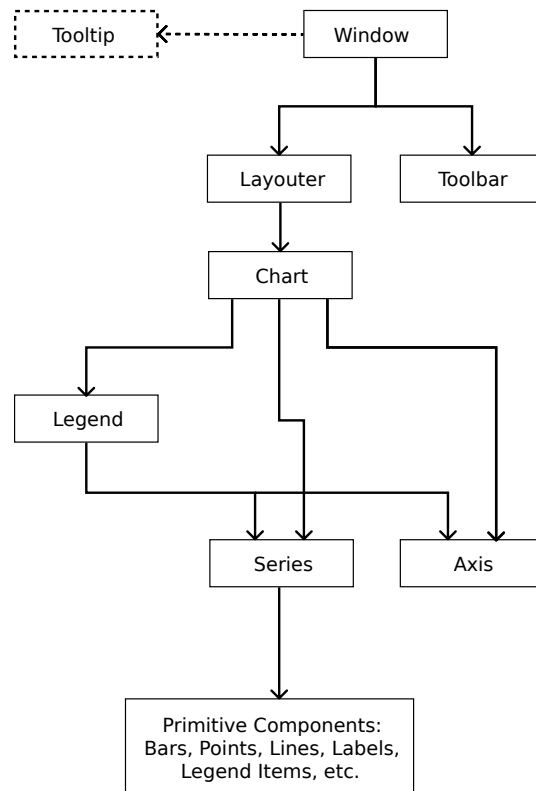
1 import {RenderArgs} from "../renderer";
2 import {
3   RespVal,
4   RespValUserArgs,
5   validateRespVal
6 } from "../../data/responsive-value/responsive-value";
7 import {WindowArgs} from "../../window";
8 import {LayoutBreakpoints} from "../../data/layout-breakpoints";
9 import {
10   LayoutBreakpointsUserArgs
11 } from "../../data/layout-breakpoints/layout-breakpoints";
12
13 export type ChartDataUserArgs = Pick<WindowArgs, 'tooltip'> & {
14   breakpoints?: LayoutBreakpointsUserArgs
15   title?: RespValUserArgs<string>
16   subTitle?: RespValUserArgs<string>
17 }
18
19 export type ChartDataArgs = ChartDataUserArgs & RenderArgs
20
21 export type ChartData =
22   Required<Omit<ChartDataArgs, 'breakpoints' | 'tooltip' | 'title' | 'subTitle'>>
23   & {
24     breakpoints: LayoutBreakpoints,
25     title: RespVal<string>
26     subTitle: RespVal<string>
27   }
28
29 export function validateChart(args: ChartDataArgs): ChartData {
30   return {
31     renderer: args.renderer,
32     breakpoints: new LayoutBreakpoints(args.breakpoints),
33     title: validateRespVal(args.title || ''),
34     subTitle: validateRespVal(args.subTitle || ''),
35   }
36 }

```

**Listing 5.5:** The file `validate-chart.ts` contains the validation logic for the chart module. The declaration and export of `ChartDataUserArgs` acts as a contract between chart creator and library. `ChartDataArgs` defines what must be passed to the validation function. `ChartData` defines the data object returned by the validation function, `validateChart`.

and the `Layouter` component, which needs no data assigned to it. The `Layouter` component applies `RespVis`' layouter mechanism in the background.

The `Chart` component is a direct child of the `Layouter` component and is bound to the same object as the `Window` component, i.e. the data object representing the current state of a chart. It is a composite component, with the outermost element being an `<svg>` element containing all SVG elements of the chart. The children of a `Chart` component depend on the chart type and underlying data, and can consist of one or more `Series` components, zero or more `Axis` components, and zero or one `Legend` components. Primitive components, consisting of bars, points, lines, labels, and legend items are low-level components, which cannot contain any other components and, therefore, form the leaves of the hierarchy. The `Tooltip` component is the only component, which is not a descendant of the `Window` component. It is rendered as a child of the document's `<body>` element. However, the rendering of the `Tooltip` component is invoked in the render routine of the `Window` component.



**Figure 5.3:** The component hierarchy in RespVis v3. Components are rendered from top to bottom, beginning with the `Window` component and ending with the `Primitive` components. Outgoing lines indicate the inclusion of lower components. The `Tooltip` component is drawn with dashed lines, since it is not a direct child of the `Window` component. [Image created by the author of this thesis.]

## 5.3 RespVis Core

The `respvis-core` package is a collection of the fundamental modules, components, classes, functions, types, and constants used throughout the RespVis code base. Its directory structure is shown in Listing 5.6. The package provides the following aggregated modules:

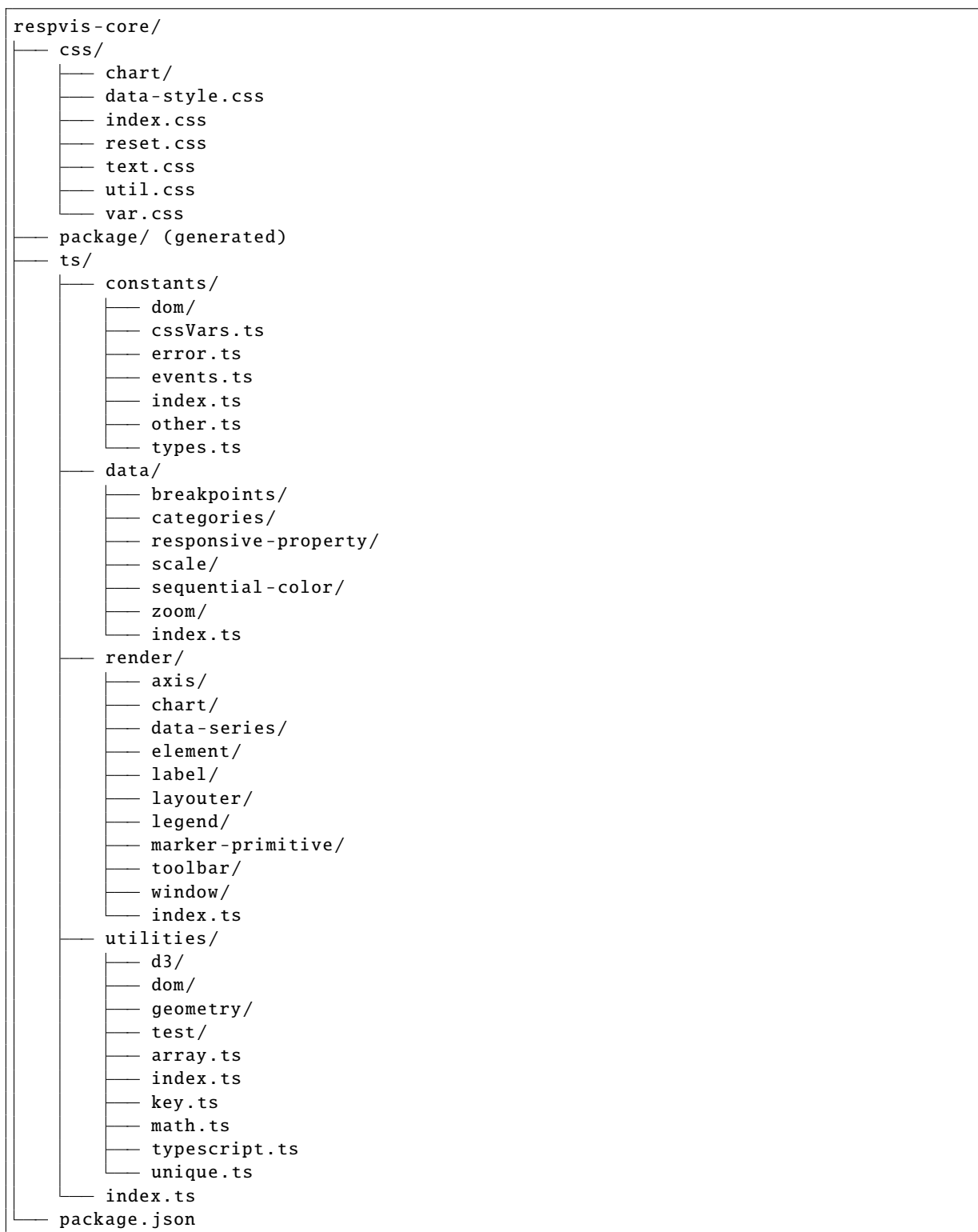
- *Render modules:* `window/`, `toolbar/`, `layouter/`, `chart/`, `data-series/`, `axis/`, `legend/`, `marker-primitive/`, `label/`, and `element/`.
- *Data modules:* `scale/`, `categories/`, `breakpoints/`, `responsive-property/`, `sequential-color/`, `zoom/`.
- *Utility modules:* `d3/`, `dom/`, `geometry/`, `test/`, `array.ts`, `key.ts`, `math.ts`, `typescript.ts`, and `unique.ts`.
- *Constant modules:* `dom/`, `cssVars.ts`, `error.ts`, `events.ts`, `index.ts`, `other.ts`, and `types.ts`.

These are listed and described in top-to-bottom order corresponding to the component hierarchy of Figure 5.3, rather than in alphabetical order.

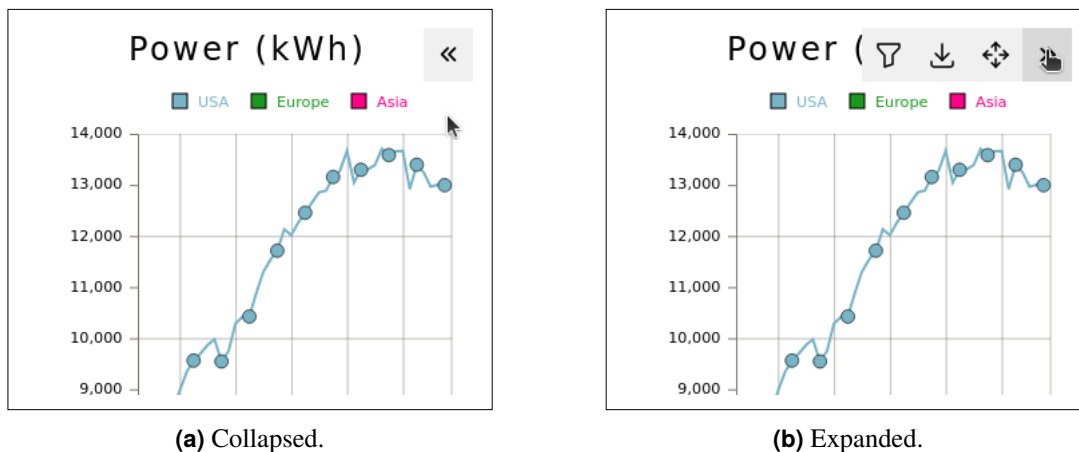
### 5.3.1 Window Modules

The `Window` modules, located in the `window/` directory shown in Listing 5.6, are responsible for the validation of window user arguments and rendering of the `Window` component, the outermost layer of a RespVis chart. The contained render routine `renderWindow` expects a selection of an empty single HTML element, already bound to a `Window` data object, as its argument. This element officiates as a single wrapper





**Listing 5.6:** The file and directory structure of the respvis-core sub-package.



**Figure 5.4:** An absolutely positioned RespVis Toolbar may overlap the chart when expanded. [Screenshots taken by the author of this thesis.]

for a complete self-contained RespVis chart. The render routine attaches a chart-specific class name to the window element and also maintains its layout CSS variables, which are discussed in Section 5.3.13. Furthermore, the function checks if tooltips are active for the current chart, and conditionally calls the render routine of the Tooltip component.

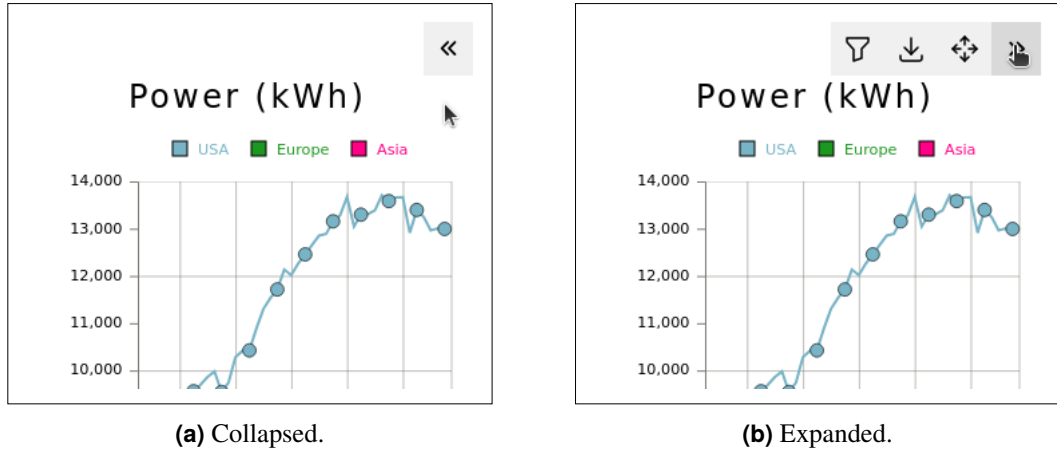
The Window component uses CSS Grid to lay out its child components Toolbar and Layouter, which are both rendered as `<div>` elements. The first row of the grid is reserved for the Toolbar, while the second row is reserved for the Layouter. The Toolbar is positioned absolutely by default. If the Toolbar overlaps with other elements of the chart, a chart creator can override its position to be static, which is discussed in Section 5.3.2.

### 5.3.2 Toolbar Modules

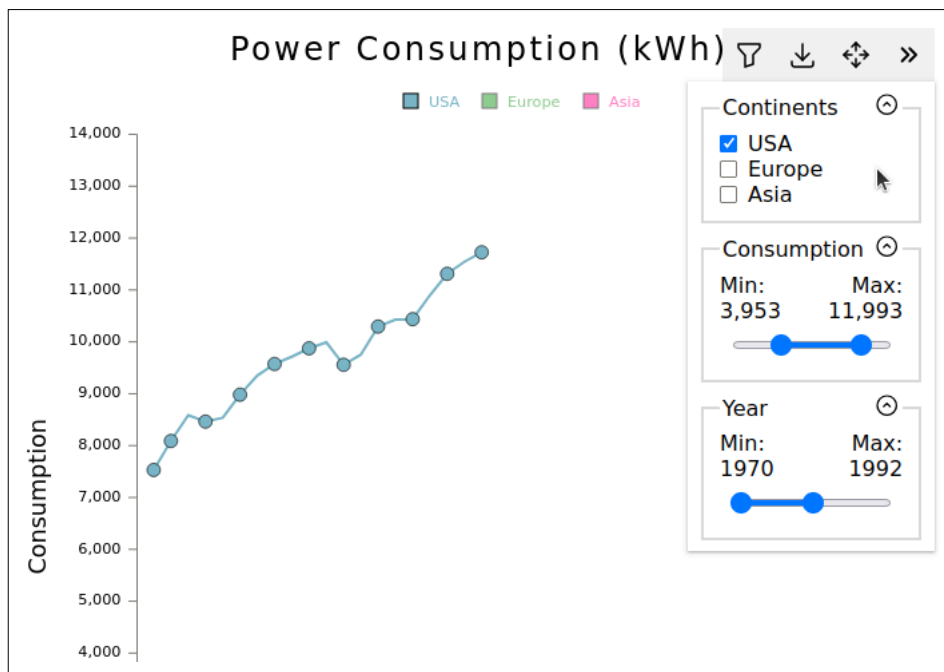
The Toolbar modules are located in the `toolbar/` directory shown in Listing 5.6. Toolbars are commonly used to make visualizations more interactive by providing a set of tools selectable from a bar. If implemented wisely, a toolbar may be of tremendous value, since it theoretically allows adding unlimited interaction options to a visualization. In RespVis v3, an optional Toolbar is included in the creation process of a visualization by default. The Toolbar is located at the top right of the chart. A chart creator can position the Toolbar either absolutely, meaning it is removed from the document flow and may overlap the chart when expanded, or statically, meaning it is allocated a grid row of its own and does not overlap the chart when expanded. This can be seen in Figures 5.4 and 5.5.

When collapsed, only a single button for expanding the Toolbar is visible. When expanded, the Toolbar displays three or four buttons, depending on the chart type, with icons indicating the corresponding tool. When a user hovers over an icon, a Tooltip with the corresponding tool name is displayed. All tools are activated by clicking their respective button.

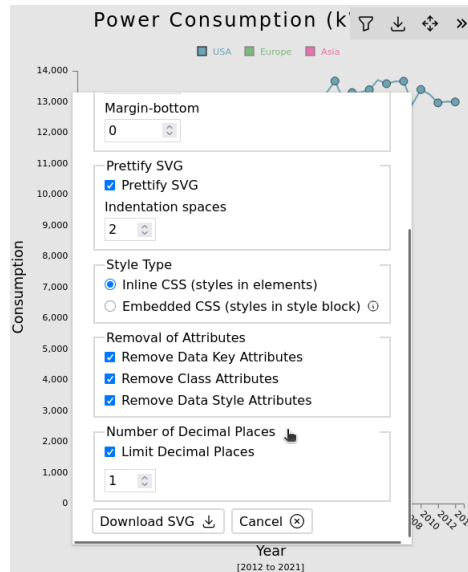
The first tool, from left to right, is the Filter Tool. If activated, the Filter Menu slides in and docks beneath the Toolbar, as can be seen in Figure 5.6. The Filter Menu contains multiple fieldsets, which give control of the active filtering settings. A fieldset can be expanded and collapsed by clicking on its caption. Each fieldset provides filter options for one dimension. This dimension may be either categorical, numerical, or temporal. For a categorical dimension, a fieldset contains a series of checkboxes giving control of each category. For numerical and temporal dimensions, the fieldset contains a double-edged range slider, allowing an end user to specify the active range of values for the corresponding dimension. The double-edged range slider is manipulated by dragging and dropping handles for minimum and maximum values.



**Figure 5.5:** A statically positioned RespVis Toolbar is allocated a grid row of its own and does not overlap the chart when expanded. [Screenshots taken by the author of this thesis.]



**Figure 5.6:** The Filter Menu with active configurations for a categorical (Continents), numerical (Consumption), and temporal (Year) dimension. [Screenshot taken by the author of this thesis.]



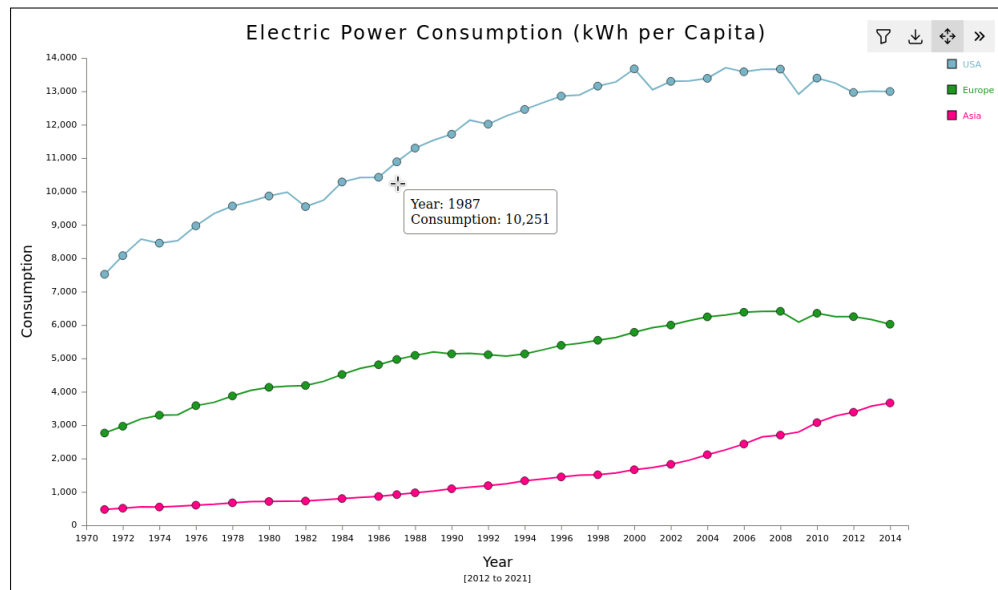
**Figure 5.7:** The scrollable Download Modal for configuring the download process. [Screenshot taken by the author of this thesis.]

To ensure a consistent presentation of the value labels, RespVis makes use of the `D3Axis.tickFormat` function provided by D3, which formats filter labels exactly like axis labels. This is especially useful for time dimensions, since their values are internally represented as JavaScript Date objects, which have various possible output formats.

The second tool is the Download Tool. It enables an end user to download the current state of a chart as a static SVG file. If activated, a modal dialog window pops up in the center of the page. Figure 5.7 shows the modal containing five fieldsets for specifying the desired download options. The first fieldset allows margins to be added to the downloaded version of the chart. The second fieldset contains download options regarding prettification. An end user can check or uncheck if prettification is applied to the downloaded SVG file. If unchecked, the option will not have any effect on the downloaded SVG file. If checked, an end user can specify a desired number of indentation spaces.

The third fieldset allows choosing between two approaches of including styles. In the first approach, non-default styles are included as inline attributes and inline styles. In the second approach, relevant style rules are filtered from all active style sheets and included in `<style>` elements. This involves complex processing and filtering of active CSS rules and, in some cases, the modification of nested CSS selectors. For these reasons, there is no guarantee of flawless results. When comparing both approaches, the inline style approach comes with the advantages of easier implementation, a higher success rate, and working fine for SVGs with a smaller number of elements. A drawback of generated inline styles is that they can not simply be edited after generation, since they are applied separately for all elements. Another disadvantage of the approach is a larger file size for charts containing many elements. The `<style>` element approach, on the other hand, has a lower success rate, but all styles are included in one place and can be edited after the creation of an SVG file. Furthermore, downloaded charts with many elements have smaller file sizes when using the `<style>` element approach, since style rules need to be specified only once.

The fourth fieldset contains removal options for RespVis-specific attributes. The fifth fieldset allows the maximum number of decimal places used in the static SVG to be set. Setting a fixed limit of one or even zero decimal places generally does not affect the appearance of a static SVG negatively, while leading to much smaller file sizes. At the bottom of the modal, there are two buttons, one for canceling and one for confirming the download. After confirmation, RespVis first creates a deep clone of the



**Figure 5.8:** The Inspection Tool is used to visualize the Inspection Tooltip, which contains information about the dimensions values at the current coordinates of the pointing device. [Screenshot taken by the author of this thesis.]

existing chart using the `Node.cloneNode` function. Then, the `<svg>` tag of the cloned element is modified by replacing its `x`, `y`, `width`, and `height` attributes with a `viewbox` attribute. This ensures the downloaded SVG scales automatically with the viewport, which is the desired default for static SVGs. The margin values specified earlier in the process are taken into account when calculating the bounds of the freshly created `viewbox` attribute. The cloned chart node is further processed by applying the previously specified download options. Finally, the SVG of the cloned node is packed into a blob object, which is automatically downloaded as an SVG file.

The third tool is the Inspection Tool. When activated, a Tooltip is displayed when hovering over any part of the drawing area. The Tooltip displays information about the dimension values at the exact coordinates of the currently used pointer device. Figure 5.8 showcases the use of the Inspection Tool in a multi-line chart.

The fourth tool is currently only available for parallel coordinates charts. When activated, a modal pops up in the center of the page, allowing a chart viewer to change chart-specific settings.

### 5.3.3 Layouter Modules

Powerful CSS layout techniques like CSS Flexbox and CSS Grid are only applicable to HTML elements. They are not applicable to elements in SVG namespaces. There are good reasons that styling works differently in the HTML and SVG namespaces. In the HTML namespace, elements are laid out according to the CSS Box Model [MDN 2024h]. All elements in an HTML document are represented as boxes. These boxes consist of four well-defined areas: margin, border, padding, and content. Powerful CSS layout modes like Flexbox and Grid can be applied to these boxes. In many cases, the default positioning of a suitable layout mode already achieves appealing results.

In the SVG namespace, on the other hand, a coordinate system is used to precisely define and position elements [MDN 2023e]. This layout technique makes sense if elements have complex shapes or are constrained to specific positions and sizes, which is the case for charts composed of many components. It would make no sense to lay out such elements via the CSS techniques applied in HTML namespaces. Instead, JavaScript and libraries like D3 should be used to take care of this task. However, an SVG tree may consist of many elements. This leads to situations where laying out certain parts of the tree

could be tremendously simplified by making use of the CSS Box Model and layout modes of the HTML namespace, rather than using the precise but potentially verbose coordinate system.

For this reason, RespVis allows certain parts of an SVG to be laid out with the CSS layout techniques of the HTML namespace. Indeed, it is possible to alternate between CSS layout and SVG layout in the same SVG chart tree. To make this possible, RespVis applies a non-trivial, custom layout mechanism under the hood. The `Layouter` modules, which are located in the `ts/render/layout/` directory of Listing 5.6, contain the code necessary for setting up this mechanism. The mechanism produces two node trees in the DOM:

1. A tree consisting of the displayed SVG elements themselves, the original SVG node tree, which is pure SVG in the SVG namespace.
2. A replication of the SVG node tree, where all elements are replaced by invisible `<div>` elements. This replicated node tree is a pure HTML node tree in the HTML namespace, which can be laid out with standard CSS layout techniques.

The `Layouter` component serves as a wrapper for these two trees. The replicated node tree can be laid out with powerful CSS layout mechanisms like Flexbox and Grid. The elements of the original node tree subsequently adapt their size and position accordingly, making it possible to control the size and position of elements inside an SVG using CSS layout techniques.

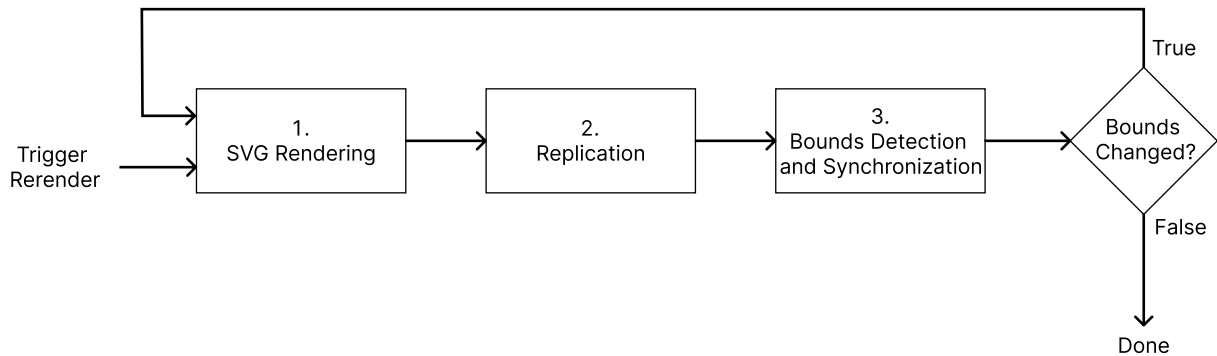
### 5.3.3.1 Layout Components

The creation and maintenance of the replicated node tree is the tricky part of the layout mechanism. To make it work, layout elements are assigned certain class names:

- `layouter`: An empty `<div>` element serving as a wrapper element for both the SVG node tree and the replicated node tree. RespVis' layout mechanism is applied by calling the function `layouterCompute` and passing a selection of a single `layouter` element.
- `layout`: All elements of the replicated HTML node tree are marked as layout elements, since their only purpose is to make it possible to lay out their SVG twin elements.
- `layout-container`: To switch from SVG standard layout to RespVis' custom layout, an element must be classed as a `layout-container`. All descendant elements of the `layout-container` element are then laid out with the custom layout mechanism. The mechanism is interrupted by an element exhibiting either a `data-ignore-layout` attribute or a `data-ignore-layout-children` attribute (which then applies to any direct descendants).
- `layout-container-positioner`: When SVG `layout-container` elements are replicated as `<div>` elements, the `<div>` elements must be positioned such that their position matches with the position of their SVG twin elements. This is achieved by adding an additional wrapper `<div>` element with a class name of `layout-container-positioner`.

In addition to class names, dedicated attributes can be attached to layout elements to influence their layout behavior:

- `data-ignore-layout`: To interrupt RespVis' custom layout and switch to SVG standard layout one can attach the `data-ignore-layout` attribute to an SVG element. These elements do not have an HTML twin element and stop the propagation of RespVis' custom layout for all descendant elements.
- `data-ignore-layout-children`: To interrupt RespVis' custom layout and switch to SVG standard layout one can attach the `data-ignore-layout-children` attribute to an SVG element. These elements *do* have an HTML twin element, but their *child* elements do not, and stop the propagation of RespVis' custom layout for all descendant elements.



**Figure 5.9:** The three layout phases of RespVis v3. Phases are processed repeatedly from left to right in sequential order until no bounds of any element change. [Image created by the author of this thesis.]

### 5.3.3.2 Layout Phases

Layouting proceeds in three phases, as shown in Figure 5.9:

1. *SVG Rendering*: The standard SVG node tree is created and rendered.
2. *Replication*: The replicated HTML node tree of `<div>` elements is created with the method `layouterCompute`. The browser positions and sizes the `<div>` elements in the replicated HTML node tree according to whatever CSS layout has been specified.
3. *Bounds Detection and Synchronization*: Now that the position and size of each `<div>` element has been determined, a custom attribute called `bounds` is applied to its corresponding SVG element, holding its `x`, `y`, `width`, and `height` in normalized SVG coordinates.

Depending on the type of SVG element, the bounds values are applied to it. A `<rect>` element has its `x`, `y`, `width`, and `height` attributes set accordingly. A `<circle>` element has its `cx`, `cy`, and `r` attributes set appropriately. Synchronizing the size and position of a `<text>` element is more complicated, but proceeds similarly. For `<g>` elements, and all other SVG elements, the `transform` attribute is used.

If the bounds of any SVG element have changed, the process starts again with SVG Rendering.

The second phase, Replication, comes with many important checks. SVG elements with the attribute `data-ignore-layout` or `data-ignore-layout-children` pause layout propagation, which is restarted if a child element has the class `layout-container`. However, replication of the SVG node tree must continue if any child is a `layout-container`, since the intermediate elements will be needed to be able to restart custom CSS layout at a deeper level if required.

At a transition from SVG layouting to custom CSS layouting, the `layout-container` SVG element is placed at the position of its parent element in the SVG node tree. This is achieved by inserting an additional `layout-container-positioner` wrapper `<div>` element into the replicated HTML node tree. This wrapper element is positioned exactly at the said location by setting its CSS property `position` to `fixed` and calculating the position of its parent in the SVG node tree.

### 5.3.3.3 Synchronization of Bounds

The synchronization of bounds between original and replicated elements is accomplished by maintaining a `bounds` attribute, containing the layout information of the replicated elements in the form of "`<x>,<y>,<width>,<height>`". This information can be accessed beginning with the second render of a chart. In addition to the `bounds` attribute, different actions are executed depending on the element type:

- `<svg>`, `<rect>`: The rectangular shape of these elements allows a 1:1 synchronization with the position and size of the replicated layout element by setting the `x`, `y`, `width`, and `height` attributes.

- `<circle>`, `<ellipse>`: The required values for synchronizing with the layout of the replicated layout element can easily be calculated. They are set to the attributes `cx`, `cy`, and `r`.
- `<g>` and other: The `transform` attribute is used to synchronize with the position of the replicated layout element.

In addition to the above-listed elements, also `<text>` elements are also sometimes laid out by RespVis. The positioning of `<text>` elements is more complex, since they come with additional presentational attributes for alignment:

- `text-anchor` (`start` | `middle` | `end`): An attribute controlling the alignment in writing direction in relation to the initial text position (derived from `x` and `y` attributes) [MDN 2024g].
- `dominant-baseline` (`auto` | `central` | `hanging` | ...): An attribute controlling the position of the baseline. The baseline is an invisible line upon which the characters of text sit. The `dominant-baseline` attribute, together with the `x` and `y` attributes, is responsible for the alignment of text vertical to the writing direction [Angelica 2024; MDN 2024b].

These attributes are necessary to exactly position text elements in SVGs, since there is no box model taking care of positioning text automatically. However, the RespVis custom layouter must lay out SVG `<text>` elements exactly like ordinary text in the HTML namespace. To achieve this behavior, the center of a `<text>` element must match with the center of its corresponding layout element.

The difficulty thereby lies in positioning the baseline. The `<text>` element's baseline position can be controlled by the `dominant-baseline` attribute, but this alone is not sufficient, since there is no available option leading to the desired result. The solution applied in the RespVis layout mechanism is to first set the `dominant-baseline` attribute of the `<text>` element to `central`, and additionally set its `y` attribute to half of its replicated layout element's height.

To additionally ensure alignment in the horizontal direction, the `text-anchor` attribute of the `<text>` element is set to `start`. Then, its `x` attribute is set to half of the difference between the replicated layout element's width and the text element's width.

With this strategy, SVG `<text>` elements laid out by the RespVis custom layouter are aligned exactly like text in the HTML namespace. This makes it possible to apply useful transformations to the elements without risking undesired side effects. One example is the rotation of y-axis titles around their centers by setting their `transform-origin` to `center`. Another advantage of this strategy is that it is automatically and uniformly applied for all replicated `<text>` elements, which saves chart developers from struggling to choose appropriate text alignment attributes.

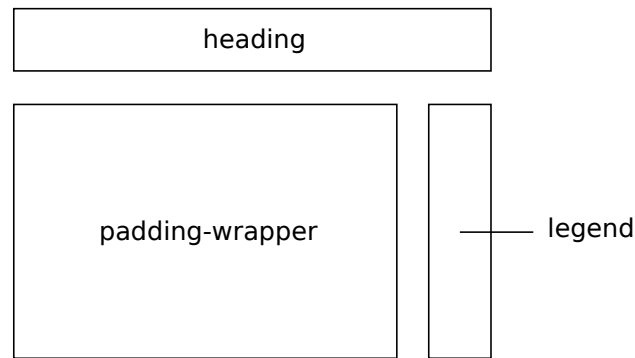
### 5.3.4 Chart Modules

The Chart modules, located in the `ts/render/chart/` directory shown in Listing 5.6, provide a convenient way of using RespVis to the fullest by making the initialization of a chart straightforward, as follows:

```
const window = document.querySelector('#chart-wrapper')
const chart = new <ConcreteChart extends Chart>(window, args)
chart.buildChart()
```

The code first selects the existing empty `<div>` element with id `chart-wrapper`. Then, a new chart instance is created by passing the empty `<div>` element and the desired arguments to the constructor of a class, which extends the `Chart` class. The constructor validates the passed arguments and throws an error message indicating the exact problem if there are serious issues with the arguments. If the arguments have only minor issues, RespVis makes small adjustments and continues without throwing an error. After validation, the data is attached to the originally empty `<div>` element, which now becomes the `Window` component of the chart.





**Figure 5.10:** Typical layout structure of a RespVis chart and its child elements. [Image created by the author of this thesis.]

```

1 .chart {
2   display: grid;
3   grid-template: auto 1fr / 1fr auto;
4   grid-template-areas:
5     'header header'
6     'padding-wrapper legend';
7 }

```

**Listing 5.7:** CSS for laying out the components of a chart in a typical arrangement.

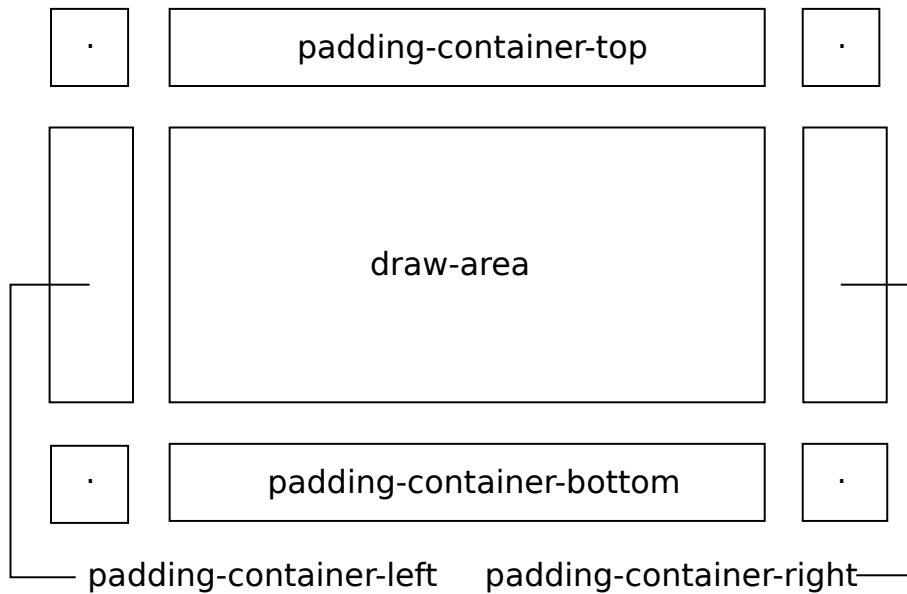
Finally, by calling `chart.buildChart()` all necessary actions are executed to set up the layout structure of the chart. This includes:

- The first render of the chart including rendering the Window, Layouter, and Chart components.
- The first execution of the custom layout mechanism.
- The installation of all necessary listeners responsible for triggering layout updates and re-renderings of the chart. These listeners are invoked if:
  - the width and/or height of the chart change.
  - an interaction with the chart requires an immediate re-rendering (such as filtering data).
  - a pointing device is hovering over the chart and is not conflicting with any ongoing D3 transitions. A re-rendering of a chart becomes thereby necessary, since CSS styles can be conditionally applied on hovering, eventually leading to changing bounds of arbitrary elements inside a chart.

#### 5.3.4.1 Chart Layout

The Chart component comes with the functionality to handle the layout mechanisms discussed in Section 5.3.3. This means the layout of a chart can be controlled via CSS. For this reason, RespVis comes with default styles to provide good-looking charts out of the box. If a chart creator desires to change the layout in general, or via media or container queries, the chart creator can conveniently do so by overriding the default CSS of RespVis. The chart element contains three child elements: `heading`, `padding-wrapper`, and `legend`. A typical arrangement is shown in Figure 5.10. The CSS responsible for it is shown in Listing 5.7, making use of a CSS Grid.

The chart itself is contained within the `padding-wrapper`. The layout of the `padding-wrapper` is shown



**Figure 5.11:** Layout structure of the padding wrapper and its child elements. [Image created by the author of this thesis.]

```

1 .padding-wrapper {
2   display: grid;
3   grid-template: auto 1fr auto / auto 1fr auto;
4   grid-template-areas:
5     '. padding-container-top .'
6     'padding-container-left draw-area padding-container-right'
7     '. padding-container-bottom .';
8 }

```

**Listing 5.8:** CSS for laying out a padding wrapper.

in Figure 5.11. Listing 5.8 shows the styles responsible for laying out the padding-wrapper. As can be seen, the padding wrapper’s child elements are laid out with a CSS Grid. The drawing area of the SVG node tree is a `<g>` element containing the graphical elements of the chart. Depending on the chart type and data, it may contain many elements, which are constrained to specific locations and sizes. There is no sense laying out these elements with CSS techniques like Flexbox or Grid. Instead, this is the point where the custom layout of RespVis stops and JavaScript and D3 take care of positioning and resizing elements in standard SVG layout.

Since `<g>` elements are only intended for grouping and do not have bounds themselves, the drawing area needs a background `<rect>` element with the same size and position of the drawing area layout element. The bounds of the background element will always match with the bounds of the drawing area layout element.

Nonetheless, since the custom layout stops at the drawing area, there may be cases of elements overflowing the bounds of the drawing area layout. Imagine a scatter plot with labeled points. If a point is located at the edge of the drawing area, its corresponding label may partially lie outside the drawing area’s bounds and may potentially intersect with elements of the heading or the legend.

For this reason, the drawing area contains a `<clipPath>` element. It limits the visible area of the drawing area and cuts off any overflowing elements. This solves the problem of overflowing content intersecting

```
1 .padding-container--left {
2   width: var(--chart-padding-left, 0);
3 }
4
5 .padding-container--top {
6   height: var(--chart-padding-top, 0);
7 }
8
9 .padding-container--right {
10  width: var(--chart-padding-right, 0);
11 }
12
13 .padding-container--bottom {
14  height: var(--chart-padding-bottom, 0);
15 }
```

**Listing 5.9:** The padding around a chart is defined by CSS variables, which can easily be overridden by chart creators.

with other layout areas of the chart. However, this approach comes with a new issue. Overflowing elements would simply be cut off, and become partly or completely invisible. This is where the padding wrapper and the padding containers come into play. The padding containers reserve space in the grid for overflowing content of the drawing area. The `<clipPath>` element of the drawing area respects the sizes of the padding containers when calculating the bounds of the drawing area's visible content. The size of the padding containers can be adjusted via CSS variables, as shown in Listing 5.9.

#### 5.3.4.2 Extending the Chart Class

The abstract `Chart` class of the `respvis-core` package cannot be instantiated on its own. Instead, it is intended to be used as a base class to create derived classes representing more advanced charts. The `Chart` class provides the abstract `Chart.renderContent` method, which must be implemented by a derived chart with a custom render function. The complex re-rendering and layouting tasks, on the other hand, are abstracted away. Furthermore, the `Chart` class implements the `Renderer` interface, which provides useful getter functions for accessing its components as D3 selections. Derived classes can reliably use these functions, which must be part of all `RespVis` charts. If a chart developer decides to construct a custom version of the base class, it must implement the `Renderer` interface too to work properly.

`RespVis` makes use of TypeScript mixins, to take advantage of multi-inheritance and behavioral composition. The `Chart` modules include the `DataSetChartMixin`, which is mixed into all advanced charts of `RespVis`. The `DataSetChartMixin` ensures that it can only be mixed into chart classes providing validated series objects by defining abstract properties which must be implemented by such charts. If a chart class uses the `DataSetChartMixin`, it will be able to invoke methods for creating `Legends`, `Toolbars`, `Data Series`, `Data Series Tooltips`, `Data Series Labels`, and highlighting of `Data Series`.

#### 5.3.5 Data Series Modules

A data series in `RespVis v3` is a set of data points which belong together, and can be specifically grouped into categories. For example, a multi-line chart with measurements from three regions would be handled as one data series with three categories. The purpose of `Data Series` modules is to provide structured solutions for validating data values passed by a chart creator and rendering the validated data as `Data Series` components in a chart. The `Data Series` modules are located in the `ts/render/data-series/` directory of Listing 5.6. Instead of providing a concrete data series type, the modules define the `DataSetSeries` interface, which all data series implementations must adhere to. Among its properties are `originalData`

and `renderData`, which are both of type `DataSeriesData`. The former represents the stable series state, while the latter is a transformed version with the current filtering, zooming, and inversion effects applied to it during each render cycle. The reason for introducing a separate property to apply these effects was that changing the original scale objects led to bugs in earlier versions. The current implementation, in contrast, avoids changing the original scale.

Objects of type `DataSeriesData` are created by calling the validation function `validateDataSeriesArgs`, which is intended to be called by concrete Data Series modules. All modules defining implementations of the `DataSeries` interface must provide the functionality to:

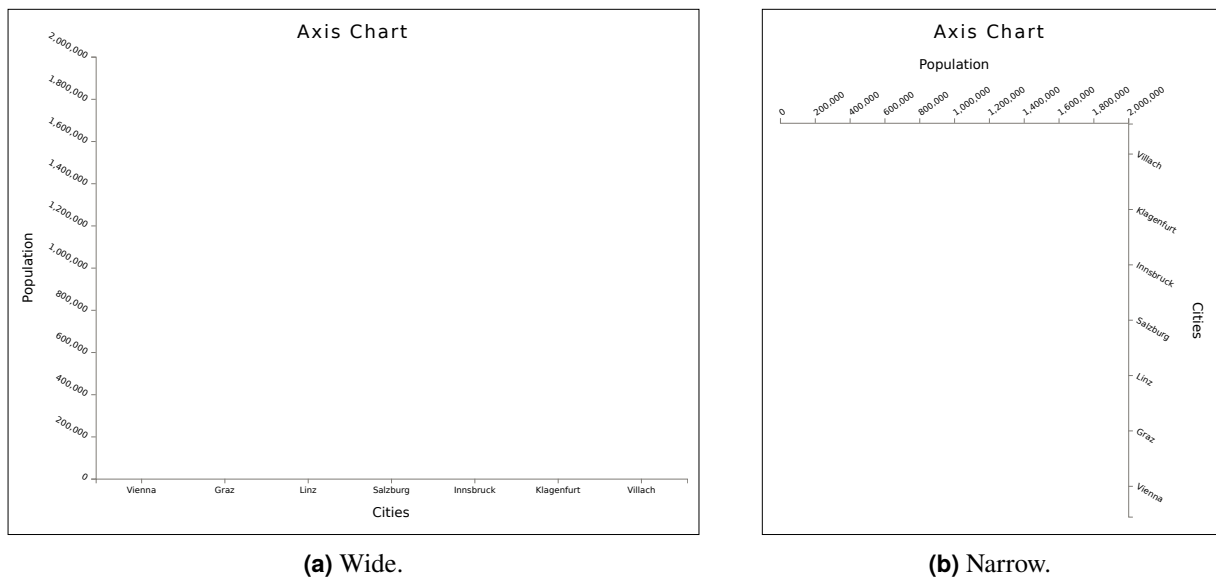
- create marker primitives conforming to axes and scales of the series.
- categorize marker primitives and represent their categories visually using categorical color encoding.
- maintain and manipulate a map of the currently active categories (filtering).
- apply sequential color encoding.
- support using a Data Series Tooltip.
- alter the state of the `renderData` property by applying zooming.
- maintain a responsive state object, which updates all responsive properties before rendering the Data Series. The responsive properties maintained by all types of data series are: the current flip state, drawing area dimensions, and drawing area scale ranges (inverted and non-inverted).

### 5.3.6 Axis Modules

Axis components act as reference elements [Kirk 2019, page 12] to indicate the mapping between data values and their spatial positions in a chart. The Axis modules are located in the `ts/render/axis/` directory of Listing 5.6, and provide two types of axes. A further two axis types are provided by the `respvis-cartesian` and `respvis-parcoord` packages.

The first axis type of `respvis-core` is the `BaseAxis`, which contains functionality shared by all other axis types. A validated `BaseAxis` contains the properties:

- `renderer`, `series`, and `scaledValues`, referencing the associated chart, `DataSeries`, and `ScaledValues` objects.
- `title` and `subtitle`.
- `configureAxis`, referencing a callback for making adjustments to the `D3Axis` object, which is generated during the render phase. If no callback is provided, the property defaults to an empty function.
- `breakpoints`, referencing a `ComponentBreakpoint` object holding breakpoint information for the width and height of the Axis component.
- `horizontalLayout` and `verticalLayout`, defining the orientation of an Axis based on the current orientation of its associated Data Series. Valid values are `bottom` and `top` for a horizontal axis layout, and `left` and `right` for a vertical axis layout. The default values are `left` and `bottom`. The chart in Figure 5.12 demonstrates how an Axis can use the properties when being flipped. The example contains two Axes. The population Axis is left-aligned for the wide version of the chart, and is top-aligned for the narrow version. The cities Axis, on the other hand, is bottom-aligned for the wide version of the chart, and is right-aligned for the narrow version.
- `tickOrientation` and `tickOrientationFlipped`, defining the desired orientation of tick labels in degrees for specified layout breakpoint widths. The tick orientation between two defined layout



**Figure 5.12:** Flipping Axes when transitioning between wide and narrow screens. [Images created with RespVis [Egger and Oberrauner 2024a] by the author of this thesis.]

breakpoint widths is linearly interpolated. Figure 5.13 demonstrates how the `tickOrientation` property is used to rotate the tick labels of both horizontal and vertical axes. The example chart in Figure 5.12 demonstrates the result of defining two separate tick orientations for flipping axes.

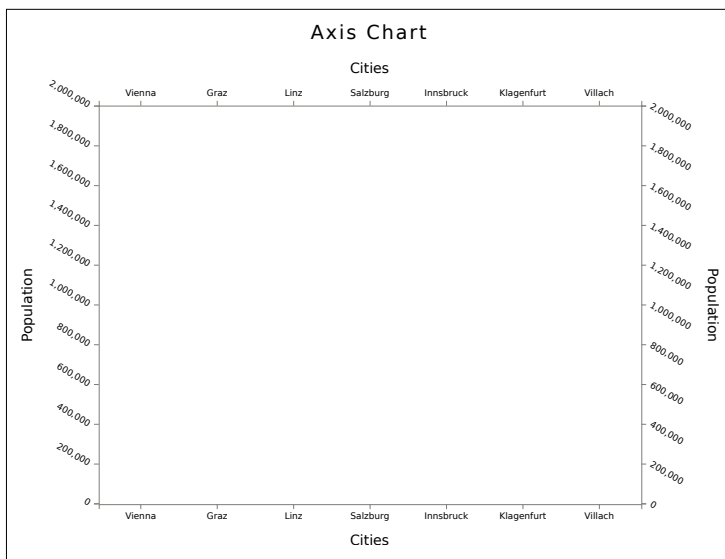
The second axis type provided by `respvis-core`, `Axis`, is a union type of the discussed `BaseAxis`, the `KeyedAxis` defined in `respvis-parcoord`, and the `CartesianAxis` defined in `respvis-cartesian`. Using type narrowing in TypeScript makes it possible to determine the real underlying type, which simplifies the initiation of additional steps on certain occasions, like the filtering of inactive axes for parallel coordinates charts.

### 5.3.7 Legend Modules

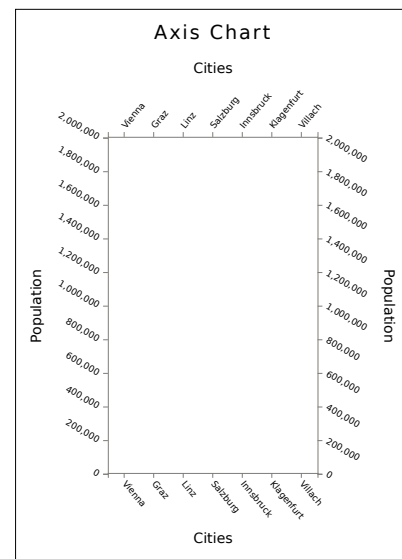
Legends are used to provide visual explanations of the colors, shapes, and sizes used in a chart [Kirk 2019, page 12]. The Legend modules of RespVis are located in the `ts/render/legend/` directory of Listing 5.6. They currently support two types of Color Legend and one Size Legend.

The first Color Legend is the Category Legend, describing the mapping of colors and categories of a Data Series. This Legend consists of colored pairs of Symbols and Labels, where each pair describes one category. Symbols are `<path>` elements with their shapes specified by callback functions, while Labels display the name of a category. By default, the order of items corresponds to the order of categories and can be reversed by a chart creator. The functionality to highlight Marker Primitives when hovering over a Legend Item with a matching category is located in the Legend Highlighting module and is built into all charts by default. The Legend Items can also be operated via a pointer device to filter out desired categories. Inactive categories are displayed half opaque. The Category Legend is only rendered if categories for Data Series are passed by a chart creator. Figure 5.26 shows the Category Legend of a Scatter Plot.

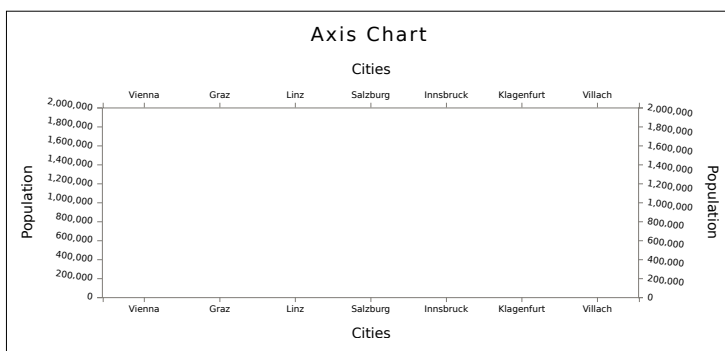
The second Color Legend is the Sequential Color Legend, which describes the sequential color encoding of a Data Series. It is only rendered if a chart creator passes the `color` argument, making it possible to add an additional numerical dimension to a visualization, which is independent of spatial scaling. It contains a `<rect>` element filled with a gradient color and an `Axis` for displaying the mapping of colors to their corresponding domain values. Currently, the `Axis` can only be positioned at the bottom of the `<rect>`



(a) Wide and tall.



(b) Narrow and tall.



(c) Wide and short.

**Figure 5.13:** Rotating axis labels when transitioning to narrower widths or shorter heights. [Images created with RespVis [Egger and Oberrauner 2024a] by the author of this thesis.]

element, but in future versions it will be possible to position it also at the left, right, or top. An example of a Sequential Color Legend can be seen in Figure 5.16.

Currently, there exists only one type of size legend in RespVis, the Bubble Radius Legend. It is only rendered in Scatter Plots if a chart creator passes a `bubble radii` argument, transforming the standard Scatter Plot into a Bubble Plot. It exhibits the same structure as the Sequential Color Legend but contains two circles instead of the gradient rectangle. The hollow, uncolored circles show the difference between bubbles with the minimum and maximum radii, as can be seen in Figure 5.26

### 5.3.8 Marker Primitive Module

A marker primitive is the visual representation of a single data point. The Marker Primitive module is located in the `ts/render/marker-primitive/` directory of Listing 5.6. It defines the `MarkerPrimitive` interface, which concrete implementations must adhere to. The interface requires marker primitives to have the following properties:

- `category`, an optional property defining the category of the marker primitive.

- `categoryFormatted`, an optional property defining the display value of the marker primitive's category.
- `styleClass`, defining the value of the style class the marker primitive belongs to. Marker primitives sharing style classes can be selected and styled collectively via CSS.
- `key`, defining the composite unique key of the marker primitive.
- `getLabel`, a method for retrieving a new label data object, with positioning adapted to the specific requirements of the marker primitive.
- `polarity`, an optional property indicating the polarity of the marker primitive.

Currently, there exist two classes in `RespVis` implementing the `MarkerPrimitive` interface: the `Bar` class of the `respvis-bar` package and the `Point` class of the `respvis-point` package. The render functions of these two marker primitives are located in the aggregated modules of their respective packages.

### 5.3.9 Label Modules

Labels are used to facilitate the comprehension of visualizations by providing additional textual information to accompany visual components. Labels are traditionally used to describe axes and legends, but can also be assigned directly to data points [DVS 2024]. The `Label` modules, located in the `ts/render/label/` directory of Listing 5.6, are dedicated modules for the latter type of labels, those assigned to data points. These modules define the `Label` interface and provide a function for rendering a series of labels.

Labels implement the `Position` interface and, therefore, contain `x` and `y` properties for defining an exact position. Furthermore, labels contain a `text` property defining their content and a `marker` reference. The reasoning behind the `marker` reference is that a label should only exist for describing a corresponding marker primitive. Therefore, the `VisualPrimitive` interface requires the implementation of a `getLabel` method, which creates a label for a specific marker primitive instance. The render function for a `Label Series` component then calls the `getLabel` method of all passed marker primitives and renders each retrieved label as a `<text>` element.

### 5.3.10 Element Modules

The `Element` modules, located in the `ts/render/element/` directory of Listing 5.6, contain useful functions for rendering simple elements not coupled to any data. The `svg.ts` file provides the `renderSVGs` function, which renders an arbitrary number of complete SVGs specified as a string array. The `bg-svg-only.ts` file contains the two functions `renderBgSVGOnlyByBox` and `renderBgSVGOnlyByRect`, which both create a background `<rect>` element around an argument, typically to provide interactivity. The first is based on the bounding box of a reference element, while the second is passed the position and extent of the rectangle as an argument.

### 5.3.11 Scale Modules

D3 provides many types of scales. The purpose of the `Scale` modules, which are located in the `ts/data/scale/` directory of Listing 5.6, is to restrict which combinations of values and scale types are specifiable by chart creators, and to provide constructs for handling scales and their corresponding domain values. The restriction of types of input values for scales was conceived to guide chart creators and immediately display error messages, if scale arguments do not match the expected type.

The most used constructs in `Scale` modules are of type `ScaledValuesSpatial`. These constructs are used to map data series values to their corresponding positions, sizes, and shapes in the drawing area. Furthermore, they provide the scales used for the creation and update process of the corresponding `Axes`. Currently, there are three kinds of spatial scaled values: numerical, temporal, and categorical.

Numerically and temporally scaled values behave similarly, and can be filtered by specifying a valid range of numbers inside the domain. They are also capable of applying zooming by rescaling. Categorically scaled values on the other hand provide a different way of filtering by controlling the filter state of each category. Zooming is not applicable to categorically scaled values. Another type of scaled values is defined by the `ScaledValuesSequential` interface, which is used to apply sequential color encoding to a Data Series.

Moreover, the Scale modules provide a helper class for applying cumulative aggregation. The `ScaledValuesCumulativeAggregator` requires a Data Series having one numerical and one categorical dimension. The aggregator calculates the cumulative sum of all records by iterating over all domain values, taking into account the order of categories. Currently, the only practical use case of the aggregator is the Stacked Bar Chart. In future versions it might be reused to add stacked area charts or waterfall charts to RespVis.

### 5.3.12 Categories Module

The Categories module is located in the `ts/data/categories/` directory of Listing 5.6. The module provides a clear definition of the required user arguments for the categorization of Data Series. If categories are included, each record of a series must be assigned a corresponding category. To ensure fast access during the render phase, the module contains a factory function, which returns validated Categories objects. These objects contain the properties `categoryArray`, which is an ordered array of Category objects, and a Category mapping with the original category strings as keys. A Category object has the properties:

- `key`, enabling the filtering of categories via the Legend or the Filter Menu of the Toolbar.
- `styleClass`, for the visualization of category differences in the Legend and marker primitives.
- `order`, representing the category order related to the other categories in the same category array. The exact value depends on the first appearance of the category in the originally passed category array.
- `formatValue`, the result of applying the formatting specified by a chart creator to the original category value.
- `value`, the original category value.

### 5.3.13 Breakpoints Modules

For a visualization to be responsive, it must be capable of adapting its layout and content depending on the available space. In responsive web design, a *breakpoint* is a specific width value (e.g. `20rem`) where a layout may change. The ranges between breakpoints are called *layout widths* (e.g. narrow, medium, and wide, if two breakpoints are defined). Breakpoints are typically defined using media or container queries in CSS. However, for visualizations, there are times when the currently active layout width must be available in JavaScript too. One example is the shortening of tick labels for narrow layout widths. This cannot really be achieved in CSS alone, but has to be done in JavaScript.

To make the current layout width information accessible from both JavaScript and CSS, RespVis introduces a novel approach to defining breakpoints for components, using the Breakpoints modules located in the `ts/data/breakpoints/` directory of Listing 5.6. Listing 5.10 shows how `ComponentBreakpoints` can be defined. Breakpoints can be specified for both width and height at the same time. Currently, component breakpoints can be assigned to Axis and Chart components. In future versions, component breakpoints will also be specifiable for the Legend component and the drawing area.

When defining component breakpoints, RespVis takes the specified values, validates them, and defines CSS variables to hold the current layout width. During each render cycle, the `Breakpoints.updateLayoutCSSVars` method updates the CSS variables of assigned elements in the DOM. For the example given in Listing 5.10, there are three breakpoints and hence four layout widths, resulting in the following layout width indices:



```

1 breakpoints: {
2   width: {
3     values: [20, 30, 50],
4     unit: 'rem'
5   }
6 }

```

**Listing 5.10:** An argument passed by a chart creator to define component breakpoints at widths 20rem, 30rem, and 50rem.

```

1 .window-rv {
2   container-type: inline-size;
3   @container style(--layout-width: 0) or style(--layout-width: 1)
4     or style(--layout-width: 2) {
5     .chart {
6       grid-template: auto 1fr auto / 1fr;
7       grid-template-areas:
8         'header'
9         'padding-wrapper'
10        'legend';
11     }
12   }
13 }

```

**Listing 5.11:** Practical usage of component breakpoints in style container queries to rearrange CSS Grid layout for specific layout widths.

1. `--layout-width = 0`: chart width  $\leq 20\text{rem}$ .
2. `--layout-width = 1`: chart width  $> 20\text{rem}$  and  $\leq 30\text{rem}$ .
3. `--layout-width = 2`: chart width  $> 30\text{rem}$  and  $\leq 50\text{rem}$ .
4. `--layout-width = 3`: chart width  $> 50\text{rem}$ .

Since CSS variables are inherited, the current layout width is available to all nested child elements of a chart. It is possible to set separate breakpoints for an `Axis` component within a chart, overriding any chart breakpoints which would otherwise apply to nested child elements like an `Axis`.

For browsers which support style container queries, the `--layout-width` variable can be used inside a style container query to adapt the layout for various widths, as shown in Listing 5.11. This approach helps avoid bugs, since breakpoints are only defined once. At the time of writing, Firefox does not yet support style container queries [Deveria 2024], so best practice to support all modern web browsers currently involves hard-coding the breakpoint values within a size container query, as shown in Listing 5.12. This approach necessitates duplication of breakpoint values, and hence is more error-prone when changes are made.

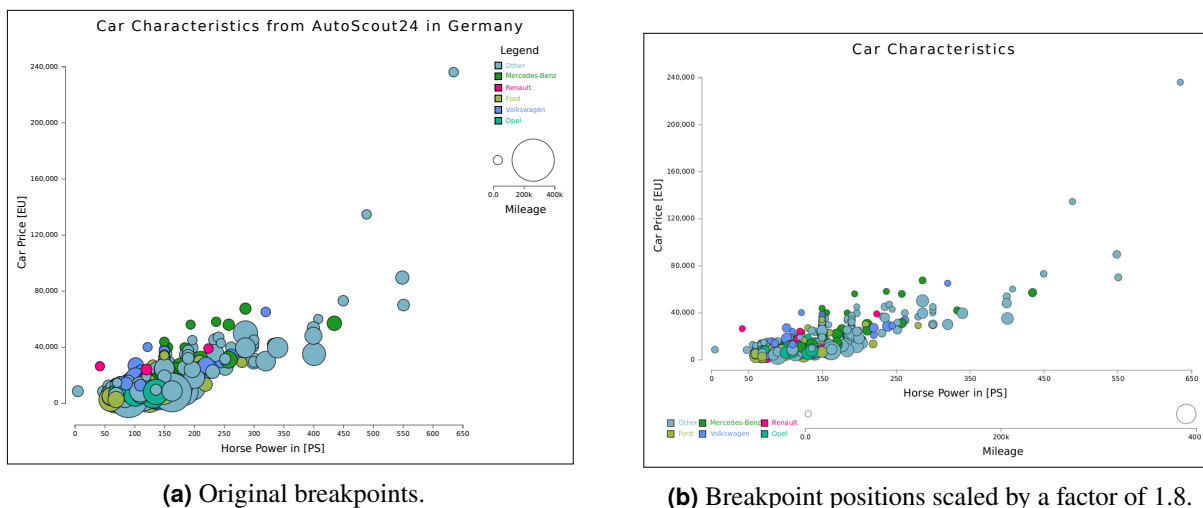
Another advantage of storing the current state of layout width in CSS variables is the capability of scaling and shifting all responsive transformations of a visualization at once. RespVis again makes use of CSS variables to provide this feature. Given the example breakpoints defined previously in Listing 5.10, and defining the CSS variables `--layout-width-factor: 1.8`; and `--layout-width-offset-factor: 0`;, the original breakpoint widths are scaled, and the final breakpoint positions are adapted accordingly. For the given example, the following width breakpoints emerge:

```

1  .window-rv {
2    container-type: inline-size;
3    @container (width < 50rem) {
4      .chart {
5        grid-template: auto 1fr auto / 1fr;
6        grid-template-areas:
7          'header'
8          'padding-wrapper'
9          'legend';
10     }
11  }
12 }

```

**Listing 5.12:** Practical usage of hard-coded breakpoint values within a size container query to rearrange CSS Grid layout for specific layout widths.



**(a)** Original breakpoints.

**(b)** Breakpoint positions scaled by a factor of 1.8.

**Figure 5.14:** Shifting the positions of all breakpoints by changing only a single CSS variable. [Images created with RespVis [Egger and Oberrauner 2024a] by the author of this thesis.]

1. breakpoint 0:  $20\text{rem} * 1.8 + 0 = 36\text{rem}$ .
2. breakpoint 1:  $30\text{rem} * 1.8 + 0 = 54\text{rem}$ .
3. breakpoint 2:  $50\text{rem} * 1.8 + 0 = 90\text{rem}$ .

Figure 5.14 demonstrates the difference with two Scatter Plots, which only differ in the above described CSS variables. As can be seen, the scaled breakpoint positions affect the title content, x-axis tick count, bubble radius, and legend position. The RespVis library automatically accounts for the adapted positions of layout breakpoints. For style sheets to automatically consider the changes too, the use of style container queries is necessary.

### 5.3.14 Responsive Property Modules

The Responsive Property modules, located in the `ts/data/responsive-property/` directory of Listing 5.6, consist of generic types and classes, which are used to extend selected chart arguments and provide responsive functionality. If an argument is declared as a responsive property, a chart creator is still able to ignore the responsive capabilities and pass a static argument in the chart creation process. However,

```
1 title: {  
2   dependentOn: 'width',  
3   scope: 'chart',  
4   mapping: {  
5     0: 'Short Title',  
6     1: 'A Medium Length Title',  
7     3: 'A Really Long Title with Extra Bells'  
8   }  
9 }
```

**Listing 5.13:** An argument passed by a chart creator to define a responsive chart title based on layout widths. Note that layout width 2 has been deliberately omitted.

the real benefits of responsive properties come into play when they are specified in their responsive form.

There are two types of responsive property. The first is the `ResponsiveValue`, which is assigned to the arguments `title`, `subtitle`, `configureAxis`, and `flipped`. A classic use case for this type is the creation of responsive chart titles. Listing 5.13 shows an example of a responsive chart title argument. The `dependentOn` property defines which layout dimension should be queried (width or height). The `scope` property is optional and defines which component breakpoints should be used. Finally, the `mapping` property maps available layout widths to the respective values of the responsive property. Layout widths are sorted from narrow to wide in ascending order. If no property value is specified for a layout width, as is the case for layout width 2 in the given example, the current property value is derived from the first valid previous layout width. The property for layout width 0 is required to be always defined. Given the same chart breakpoints of `20rem`, `30rem`, and `50rem` previously defined in Listing 5.10, the following chart titles will be used:

1. `--layout-width = 0: 'Short Title'`
2. `--layout-width = 1: 'A Medium Length Title'`
3. `--layout-width = 2: 'A Medium Length Title'`
4. `--layout-width = 3: 'A Really Long Title with Extra Bells'`

The second responsive property type is `BreakpointProperty`, which is assigned to the arguments `tickOrientation`, `tickOrientationFlipped`, and `radii`. Breakpoint properties are specified similarly to responsive values. Breakpoint properties define `breakpointValues` properties, which map breakpoints to values. Since breakpoints define exact values rather than ranges, the current value of a responsive property is interpolated between the defined breakpoint values. This allows for the continuous change of a responsive property.

A classic use case for this type is the specification of axis tick rotations. Listing 5.14 shows an example of a responsive `tickOrientation` argument. If no value is specified for a breakpoint index, as is the case for breakpoint 1 here, the breakpoint is ignored for the interpolation process. Before the first breakpoint, and after the last breakpoint, the corresponding value is simply propagated and returned as the current active value. The property for breakpoint 0 is required to always be defined. Given the same chart breakpoints of `20rem`, `30rem`, and `50rem` previously defined in Listing 5.10, Figure 5.15 shows how RespVis computes the current tick orientation for an Axis.

### 5.3.15 Sequential Color Module

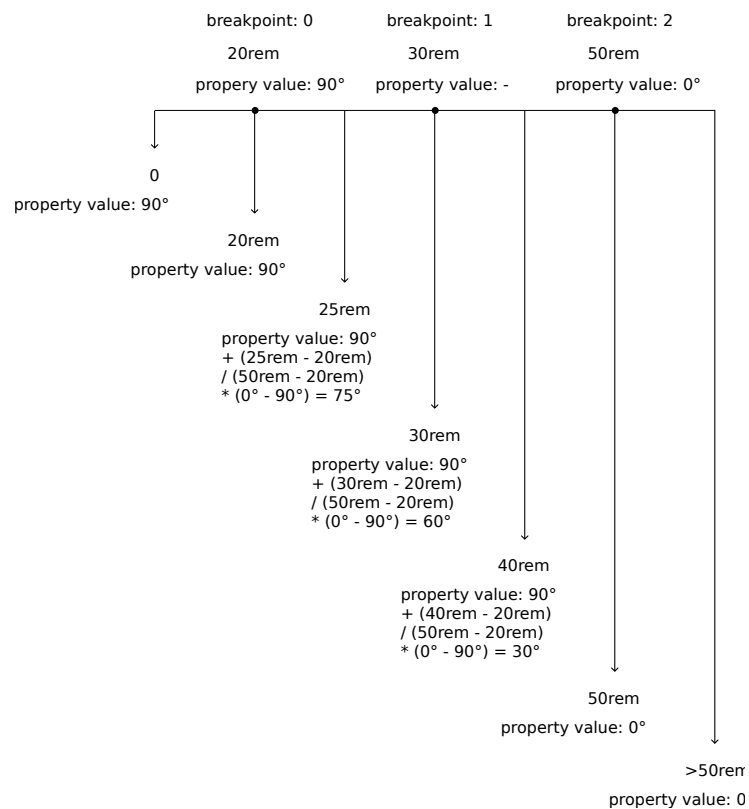
The Sequential Color module, located in the `ts/data/sequential-color/` directory of Listing 5.6, contains the interfaces required for adding a continuous color encoding to a Data Series. It is based on D3's

```

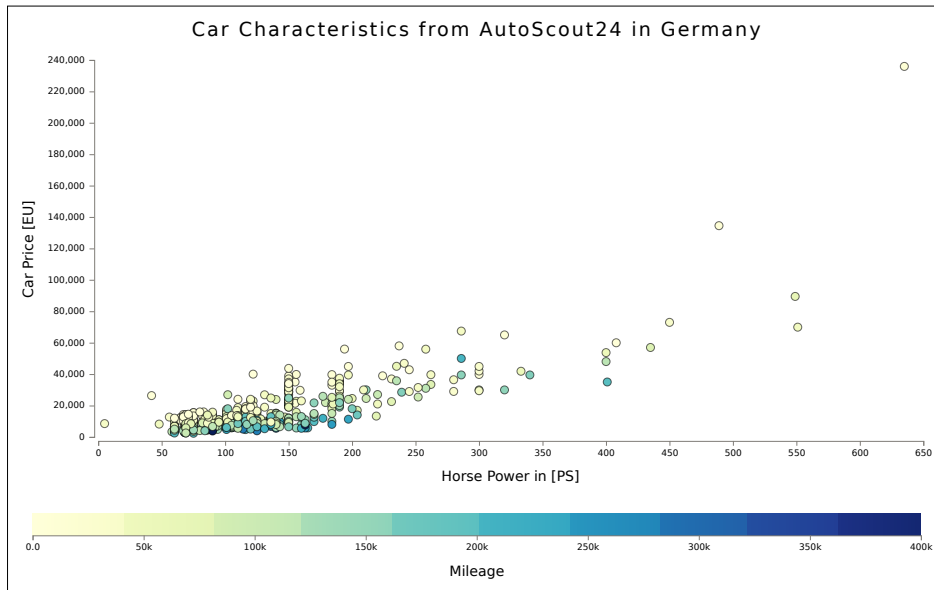
1 tickOrientation: {
2   dependentOn: 'width',
3   scope: 'chart',
4   breakpointValues: {
5     0: 90,
6     2: 0
7   }
8 }

```

**Listing 5.14:** Argument passed by a chart creator to define a responsive, interpolated tick orientation for an Axis, based on breakpoints.



**Figure 5.15:** Sampled tick orientations interpolated between defined breakpoint values. [Image created by the author of this thesis.]



**Figure 5.16:** Scatter Plot with sequential color encoding. [Image created with RespVis [Egger and Oberrauner 2024a] by the author of this thesis.]

`scaleSequential` function. The enclosed factory function validates user arguments and produces objects containing the domain values, a continuous scale for mapping domain values to their respective color counterparts, and a `BaseAxis` object to use in the `Legend` component to visually explain the mapping of values to colors. If a continuous color dimension is included, each record of a `Data Series` must be assigned a corresponding color, otherwise `RespVis` throws an error with a dedicated error message. Figure 5.16 shows how continuous color encoding is applied to a `Scatter Plot`. Continuous color encoding is not reserved for point markers only, but can also be applied to all other marker primitives.

### 5.3.16 Zoom Module

Zooming is a crucial tool for overcoming the problems of limited resolutions and narrow screens. The standard approach, geometric zoom, allows an end user to control the magnification of a visualization, and thereby trade the space needed for currently less relevant information for more space for areas of interest [InfoVis:Wiki 2006].

The `Zoom` module, located in the `ts/data/zoom/` directory of Listing 5.6, contains the interfaces required to enable zooming for `Data Series` components. The enclosed factory function requires the maximal scale factors for zooming in and out respectively and returns objects which additionally contain a `D3 zoom` behavior and the current zoom transform state. The details of `D3 zoom` operations are discussed in Section 2.7.

All concrete `RespVis` data series types are capable of applying zooming (except `Stacked Bar Series`). Zooming can only be applied to scales with a numeric domain (numerically or temporally scaled values), with different `Data Series` types leading to different zooming effects. For example, while `Point Series` change only positions, their size remains unchanged. `Bar Series` are the opposite as they do not change positions, but adapt their size to fit the zoomed state of their scales.

### 5.3.17 Utilities Modules

The utilities modules are located in the `ts/utilities/` directory of Listing 5.6. The `D3` modules, located in subdirectory `d3/`, contain utility functions and types for interacting with constructs defined in the `D3` visualization library. The `drag.ts` module exports the functions `relatedDragWayToSelection` and

`relateDragWayToSelectionByDiff` for calculating the covered drag path relative to a reference element. Furthermore, it includes the `attachActiveCursorLocking` function, which is used to set up event listeners, which take care of locking cursor appearances during drag and drop interactions with a chart. The `formats.ts` module exports the `formatWithDecimalZero` function, which enforces D3 format functions to always include values of 0 unformatted. The `re-exports.ts` module exports D3 functionality, which is frequently used by chart creators. Including these exports means the standalone bundle contains parts of D3, while the dependency-based bundles remain unchanged. The decision to re-export parts of D3 completely was made because of a negligible increase in bundle size compared to a much friendlier chart creation experience. The `selection.ts` module provides utility functions for dealing with D3 selections. This includes mapping, applying a class list, retrieving a set of selected elements, retrieving computed CSS variables, and creating a selector and a class string from a class list.

The `text.ts` module contains the `positionSVGTextToLayoutCenter` function, which is used in the layouting phase to align the positions of SVG `<text>` elements with their layout element counterparts, as explained in Section 5.3.3.3. The `throttle.ts` module contains a `throttle` function, which is used to prevent runtime issues with frequently occurring events. More advanced throttling functionality can be achieved using the `ThrottleScheduled` class, which does not stall a function call completely, but schedules it to be executed after a specified delay. The `transition.ts` module provides functions for adding special classes to elements in selections or transitions. Three functions, `addD3TransitionClass`, `addD3TransitionClassForSelection`, and `removeD3TransitionClassSelection`, are used to indicate if elements currently undergo a D3 transition. Another three functions, `addCSSTransitionEnterClass`, `addCSSTransitionExitClass`, and `cancelExitClassOnUpdate` are used to assign special class names, which can be used to apply simple CSS transitions.

The `Dom` modules contain useful functions for accessing and manipulating elements in the DOM. The `detectClassChange` function in `detect-mutation.ts` is used to assign a callback function if the class of an element changes. The `element.ts` module provides two functions for determining the absolute and relative bounds of an element, and another function for detecting all non-default computed styles of an element for a given list of properties. The `unit.ts` module contains the `cssLengthInPx` function, which converts values given in the units `px`, `rem`, `em`, or `%` into `px`.

All basic geometric types and constructs of `RespVis` are located in the `Geometry` modules in subdirectory `geometry/`. The `Position` module defines the `Position` type and its utility functions. It is heavily used by the `Shapes` modules, which are also part of the `geometry` modules. The `Shapes` modules define types representing graphical SVG elements and utility functions for manipulating, positioning, and querying attributes of these elements. The `Angle` module provides utility functions for dealing with angles. Currently, only a single function is provided, for normalizing a given angle.

The `Array` module defines the namespace `RVArray`, which contains additional functions for advanced use cases of the JavaScript built-in `Array` object. Chart creators pass data in form of arrays containing domain values. One use case is the equalization of related domain arrays by shortening the longer one with the `RVArray.equalizeLengths` function. To ensure passed arrays are correctly typed, there are type guards for validating arrays of numbers, strings, `Dates` and objects containing a `valueOf` property of type `number` (number-like objects). Another use case is the calculation of a number array's sum with the `RVArray.sum` function. The `RVArray.mapToRanks` function calculates the rank of each element of a number array and returns a new array of ranks. One usage of the function is the decoupling of the visual order of `Axes` in `Parallel Coordinates Charts` from their original order on drag and drop interactions. The elimination of duplicated values from an array becomes possible with the `Array.clearDuplicatedValues` function. A practical use case of the function is the establishment of an ordered category array from all category values passed by a chart creator.

The `Key` module defines the `Key` class, which is used to define the keys of marker primitives. The class provides useful methods for retrieving the data series key, categorical keys, and individual key of a marker primitive. Moreover, the `Key` module provides utility functions for splitting composite keys and merging

```
respvis-tooltip/  
├── css/  
│   └── index.css  
├── package/ (generated)  
├── ts/  
│   ├── render/  
│   │   ├── data-series-tooltip.ts  
│   │   ├── index.ts  
│   │   ├── movable-cross-tooltip.ts  
│   │   ├── render-tooltip.ts  
│   │   └── tooltip.ts  
│   └── index.ts  
└── package.json
```

**Listing 5.15:** The file and directory structure of the `respvis-tooltip` sub-package.

and combining sub keys. The `getActiveKeys` function returns all active keys of a given `ActiveKeyMap` object. The current `RespVis` key schema defines:

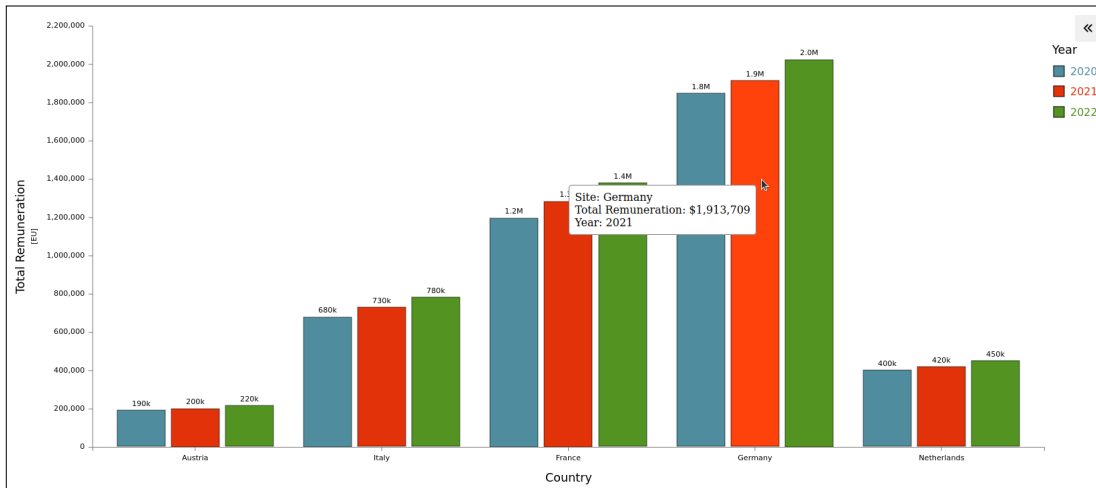
- `s-<index>`: A series key.
- `a-<index>`: An Axis key.
- `ac-<index>`: A Sequential Color Axis key.
- `ar-<index>`: A Radii Axis key.
- `s-<index>-c-<index>`: A category key of a Data Series independent of an Axis.
- `a-<index>-c-<index>`: A category key of a Data Series visually presented as an Axis.

The `Test`, `Math`, `Typescript`, and `Unique` modules comprise the remaining utility modules. The `Test` module contains the `measureFunctionPerformance` function, which is not exported, but can be used for internal performance tests. The `Math` module is conceived to provide useful math utility functions. Currently, it provides only the `calcLimited` function, which limits a calculation to avoid underflows and overflows. The `Typescript` module contains the `applyMixins` function which enables the usage of TypeScript mixins, and is described in the official TypeScript documentation [Microsoft 2024a]. The `Unique` module contains the function `uniqueId` for generating unique identifiers, and avoiding naming collisions for certain elements, which is useful in case multiple `RespVis` charts are included in a page.

## 5.4 RespVis Tooltip

The `respvis-tooltip` package provides the functionality to render and manipulate the `Tooltip` component. Its directory structure is shown in Listing 5.15. The `Tooltip` component is the only component rendered outside the `Window` component. The render function, which is only called if at least one chart with active Tooltips is created, attaches the `Tooltip` component to the document's `<body>` element. It is shared by all `RespVis` charts of a web site. Only one `Tooltip` component is needed, since an end user can only inspect one chart at once. Furthermore, the render function attaches a `pointermove` event listener at the document's window to consistently check if the tooltip state of any chart must be changed, since each chart maintains a dedicated `Tooltip` object.

The `Tooltip` object is instantiated by calling the constructor of the `Tooltip` class and holds configuration properties. Moreover, it provides the two methods `isVisible`, which checks the current visibility state for all included tooltip types, and `applyPositionStrategy`, which makes it possible to have multiple position



**Figure 5.17:** The Data Series Tooltip is used to display information about the currently hovered marker primitive. [Screenshot taken by the author of this thesis.]

strategies for different types of Tooltip components. Currently, two strategies of positioning are supported. In the first one, the sticky strategy, the Tooltip is aligned diagonally with an adjustable offset relative to the current mouse position. Furthermore, the Tooltip is automatically placed such that the available space around the mouse is optimally used. The second strategy does not position the Tooltip at all and leaves this task to a chart creator.

Currently, two types of tooltip information are supported. The first is the Inspection Tooltip, which is activated via the Toolbar as discussed in Section 5.3.2. The second is the Data Series Tooltip, which can be applied to any selection of elements bound to `DataSeries` objects containing the optional property `markerTooltipGenerator`. This property is a callback function passed by a chart creator as an argument at the creation time of a chart. The callback accepts two arguments: the first is the currently hovered element, and the second is the data object of the corresponding marker primitive. With this context, the callback returns markup code as a string, which is inserted into the Tooltip component as soon as it is activated. Figure 5.17 shows the Data Series Tooltip for a Grouped Bar Chart.

## 5.5 RespVis Cartesian

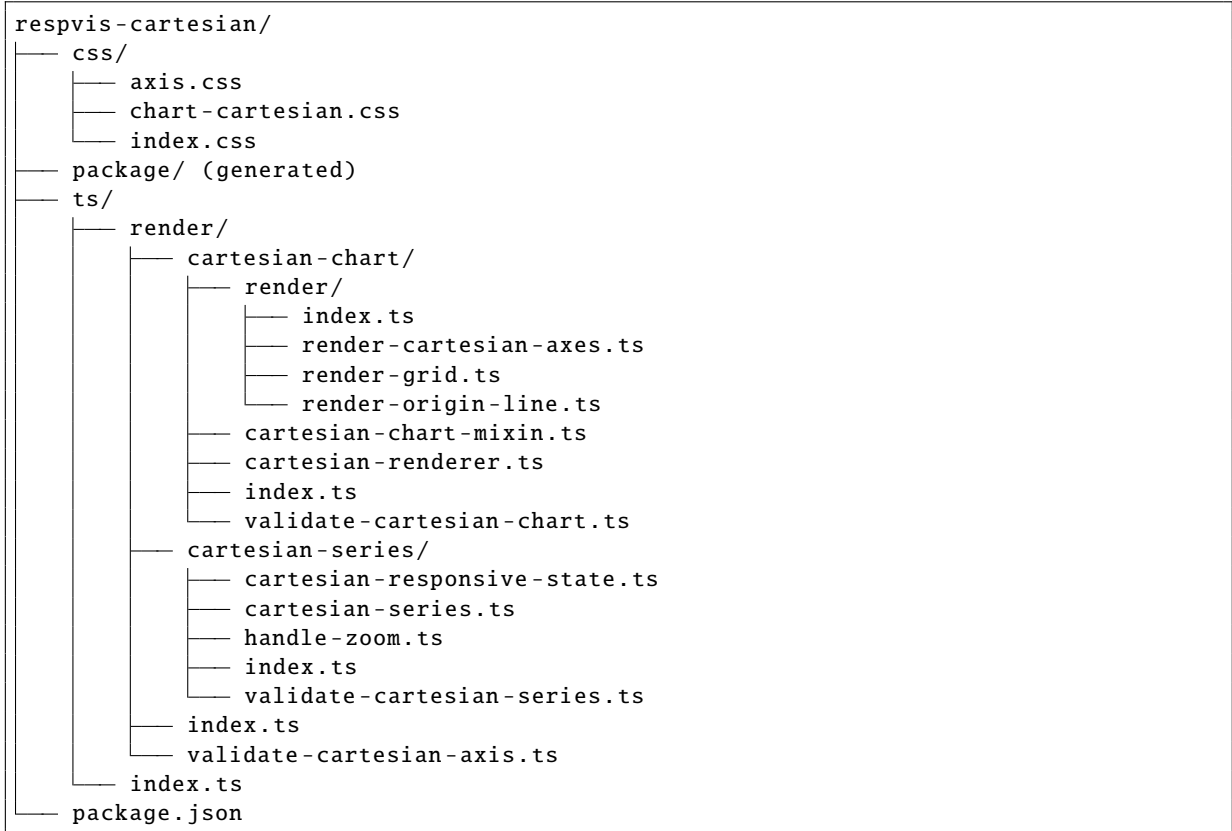
In cartesian charts, components are positioned based on a cartesian coordinate system [Kirk 2019, page 291]. Currently, `respvis-cartesian` only supports two-dimensional coordinates with a horizontal (x) and a vertical (y) dimension. The directory structure of the package is shown in Listing 5.16.

The `respvis-cartesian` package does not contain any chart class which can be instantiated. Instead, it provides the interfaces which the cartesian chart classes `BarChart`, `LineChart` and `ScatterPlot` must adhere to, and a factory function for creating valid `CartesianChartData` objects. The `CartesianChartData` interface extends the `SeriesChartData` interface and further declares the properties:

- `series`, the `CartesianSeries` object.
- `x`, a `CartesianAxis` object representing the horizontal Axis of a chart, assuming that the associated Data Series is not flipped.
- `y`, a `CartesianAxis` object representing the vertical Axis of a chart, assuming that the associated Data Series is not flipped.

`CartesianChartData` objects are used in the cartesian render functions. These functions can be invoked by all classes making use of the `CartesianChartMixin` class. The first render function is the





**Listing 5.16:** The file and directory structure of the respvis-cartesian sub-package.

```

1 .padding-wrapper {
2   display: grid;
3   grid-template: auto auto 1fr auto auto / auto auto 1fr auto auto;
4   grid-template-areas:
5     ' . . axis-top . .'
6     ' . . padding-container-top . .'
7     'axis-left padding-container-left draw-area padding-container-right axis-right'
8     ' . . padding-container-bottom . .'
9     ' . . axis-bottom . .';
10 }

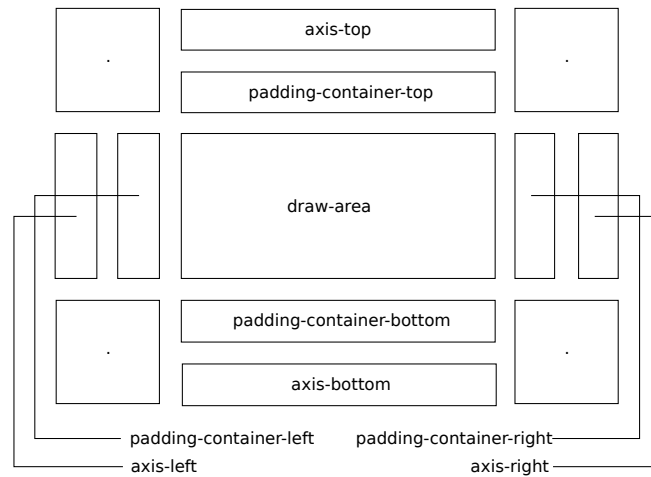
```

**Listing 5.17:** CSS for laying out the padding wrapper in cartesian charts.

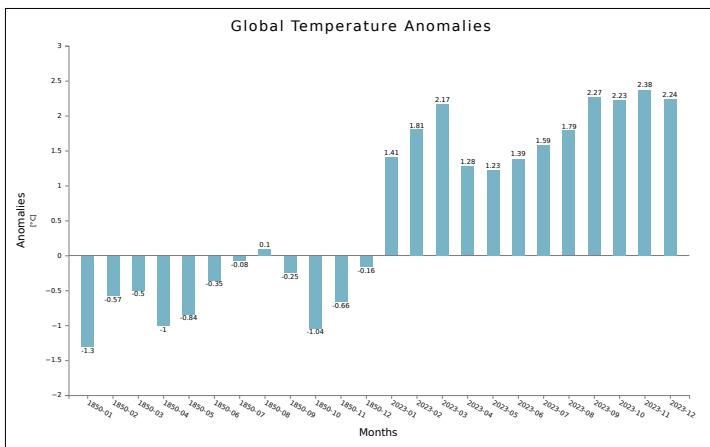
`renderCartesianAxis` function. Invoking this function leads to a change in the standard layout structure discussed in Section 5.3.4.1, since Cartesian Axes are added to the padding wrapper. The adapted layout structure is illustrated in Figure 5.18 and the styles responsible for the layout can be seen in Listing 5.17.

A call to the second render function, `renderOriginLine`, places horizontal or vertical lines at the origin of a chart's Axes, given that the domain of the underlying data values includes both positive and negative values. Figure 5.19 shows a Bar Chart with bars representing values, which are either positive or negative. Letting the bars rest on an origin line helps in understanding the bipolar nature of the bars at first glance.

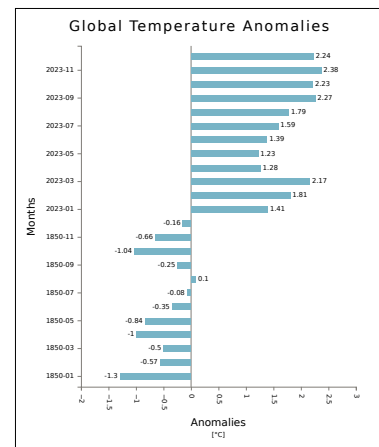
The third render function, `renderGrid`, is responsible for adding grid lines to a chart. Grid lines, in general, should assist an end user in the interpretation of a chart. They come with the benefits of visually



**Figure 5.18:** The layout structure of the padding wrapper and its child elements in cartesian charts. [Image created by the author of this thesis.]



(a) Horizontal origin line.



(b) Vertical origin line.

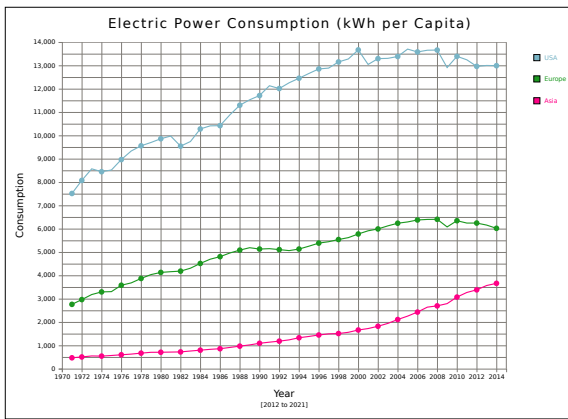
**Figure 5.19:** The origin line indicates the bipolarity of bar markers. [Images created with RespVis [Egger and Oberrauner 2024a] by the author of this thesis.]

aligning data elements far away from axes, highlighting differences between data elements far away from axes, and dividing a chart into specific sections, which could optionally be further analyzed with the help of interaction tools [Choudhury 2014].

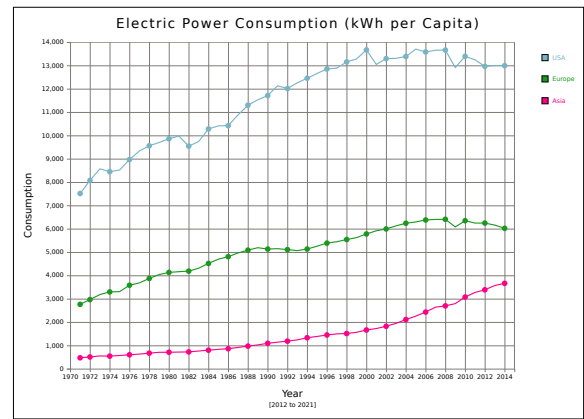
To function properly, the `renderGrid` function is dependent on the `CartesianAxis` objects representing the horizontal and vertical Axes of a cartesian chart. These objects provide, in addition to the `BaseAxis` properties, a reference to their corresponding `CartesianSeries` and optional configuration properties for the distance between grid lines (`gridLineFactor`), and the inversion state of the Axis (`inverted`).

The `gridLineFactor` is specifiable by a chart creator and can be chosen as any positive number  $> 0.01$ . If a chosen value differs from this restriction, `undefined` will be assigned to the property. There are four possible scenarios depending on the value of the property as illustrated in Figure 5.20:

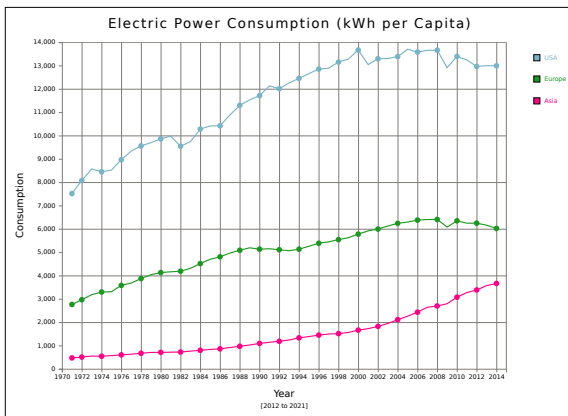
- `gridLineFactor < 1`: Grid lines are created at the position of each axis tick and, furthermore, in between two ticks according to the factor. A factor of 0.5 would, for example, lead to one additional line exactly between each two ticks.



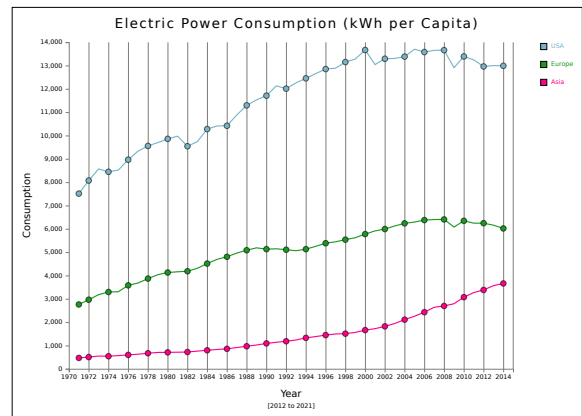
(a) Grid line factor < 1.



(b) Grid line factor = 1.



(c) Grid line factor > 1.



(d) Grid line factor undefined.

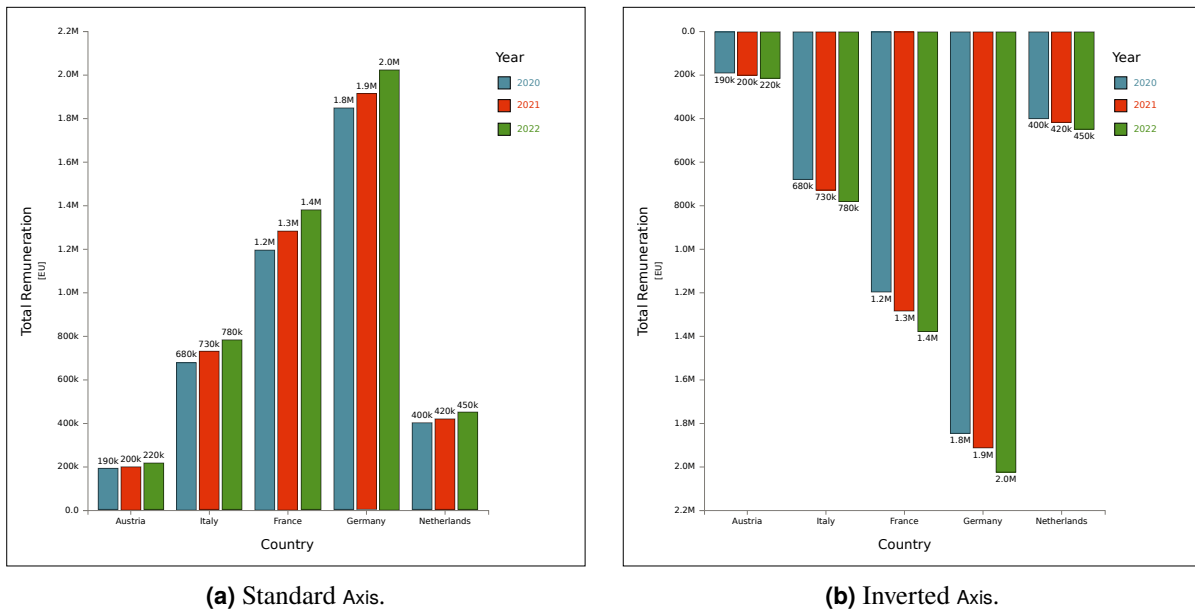
**Figure 5.20:** The appearance of grid lines depends on the chosen grid line factor. Here, the grid line factor of the Y-Axis is varied. [Images created with RespVis [Egger and Oberrauner 2024a] by the author of this thesis.]

- `gridLineFactor = 1`: Grid lines are created at the position of each axis tick.
- `gridLineFactor > 1`: First, the two neighboring axis ticks with the greatest distance are determined. Then, the distance is multiplied with the `gridLineFactor`, resulting in the grid line distance. Lastly, grid lines are created in intervals of the calculated grid line distance, starting at the first of the two originally retrieved axis ticks, in both directions until no further grid line can be accommodated.
- `gridLineFactor = undefined`: No grid lines are created.

The boolean property `inverted` may be specified by a chart creator and can be chosen to be a responsive property, which are discussed in Section 5.3.14. If the `inverted` property is true, the mapping between the domain and range of the underlying axis scale is inverted, as can be seen in Figure 5.21.

Another essential part of the package is the abstract `CartesianSeries` class. The purpose of the class is to serve as a blueprint for other classes, which hold the data necessary for rendering `Cartesian Series` components. It implements the `DataSet` interface discussed in Section 5.3.5 and is inherent to the derived classes `BarBaseSeries`, `PointSeries` and `LineSeries`. The data properties held by `CartesianSeries` objects must be of type `CartesianSeriesData`, which defines:

- an `x` property, the scaled values object representing the horizontally scaled values of the Data Series,



**Figure 5.21:** A Grouped Bar Chart with a standard Y-Axis compared to an inverted Y-Axis. [Images created with RespVis [Egger and Oberrauner 2024a] by the author of this thesis.]

assuming the series is not flipped.

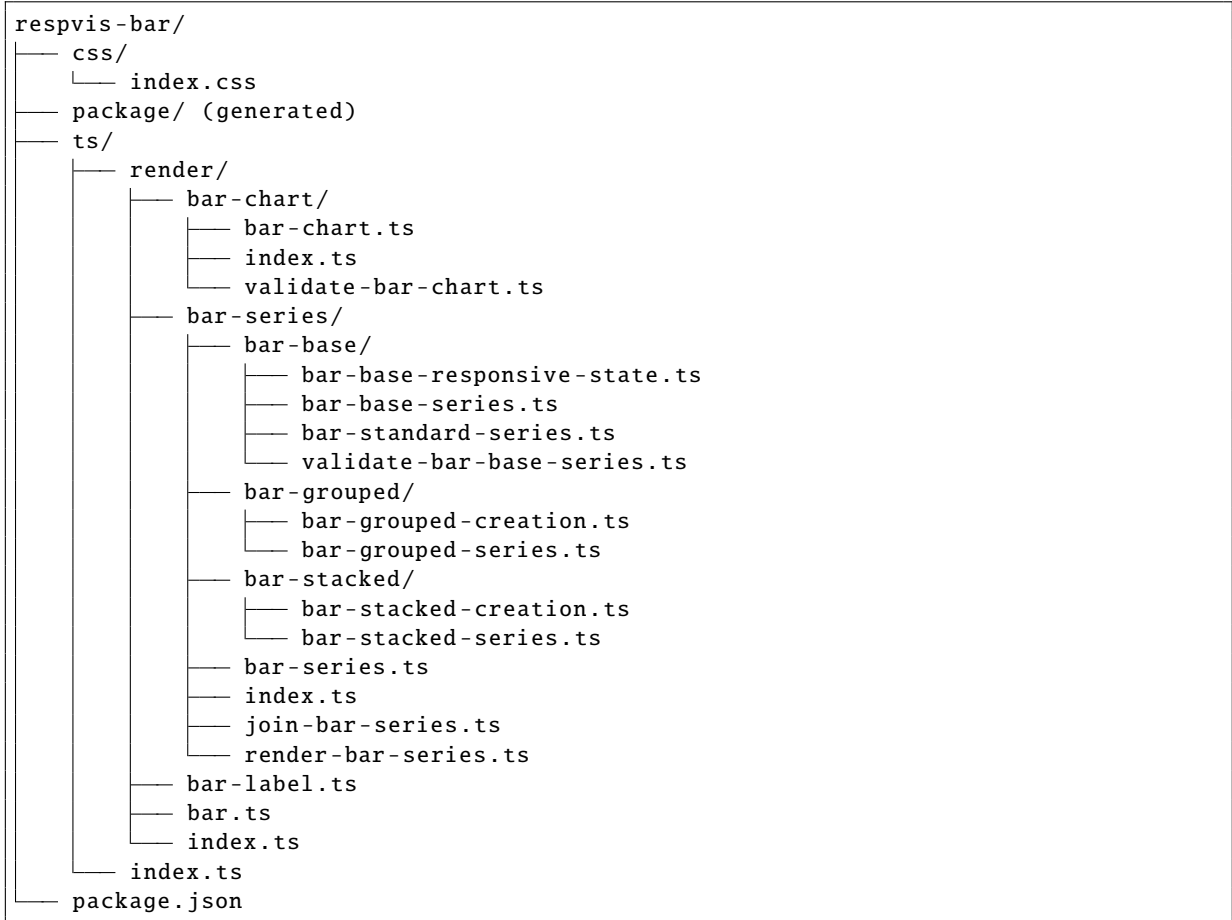
- a `y` property, the scaled values object representing the vertically scaled values of the Data Series, assuming the series is not flipped.
- an optional `zoom` property, representing the current zoom state of the Data Series.

The `CartesianSeries` class furthermore requires derived classes to hold a `responsiveState` property of type `CartesianResponsiveState`, which provides:

- an update function for maintaining the Data Series' `x` and `y` scales and setting up zoom event listeners, which take care of maintaining the zoom object.
- helper functions for retrieving the current horizontally and vertically scaled values, depending on the flip state of the Data Series.

## 5.6 RespVis Bar

Bar charts are used to compare numerical variables of dataset items by visualizing categorized bars in a cartesian coordinate system. The bar length represents the magnitude of the numerical variable, which is compared. The bar position represents its categorical value. Bars always rest on the categorical axis. If the categorical variable is displayed along the horizontal axis, the chart can be referred to as a column chart. If the other way around, the chart can be referred to as a row chart [Kirk 2019, pages 140–141, 159]. Both chart types are supported by `respvis-bar`, RespVis' sub-package for creating bar charts. Furthermore, the creation of `Grouped Bar Series` and `Stacked Bar Series`, which are more complex variations of `Standard Bar Series`, is possible using `respvis-bar`. The directory structure of the package is shown in Listing 5.18.



**Listing 5.18:** The file and directory structure of the `respvis-bar` sub-package.

### 5.6.1 Bar Chart Modules

The Bar Chart modules, located in the `ts/render/bar-chart/` directory of Listing 5.18, enable the creation of Bar Charts by providing the `BarChart` class. The `BarChartUserArgs` interface acts as a contract between `RespVis` and a chart creator, and the `validateBarChart` function returns a validated `BarChartData` object in the validation phase of the chart.

The optional `BarChartUserArgs.series.type` property must be specified by a chart creator to choose the desired Bar Series variation. If the property is omitted, by default a Standard Bar Series will be created. As soon as the property is specified, a good Integrated Development Environment (IDE) is able to narrow down the desired bar series type, supporting a chart creator by revealing additional parameter options based on the chosen type.

The render routine of a Bar Chart is defined in the `renderContent` method. This method invokes the render functions of the incorporated mixins `DataSeriesChartMixin` and `CartesianChartMixin` to render all cartesian chart components. Calling the function `renderBarSeries` completes the render process by deriving all Bar data objects from the underlying `BarSeries`, binding the data objects to `<rect>` elements, and subsequently calling `joinBarSeries`. The `joinBarSeries` function ultimately takes care of updating the enter, update, and exit selections of a Bar Series by using the bound data objects to perform the necessary DOM manipulation operations on the `<rect>` elements.



**Figure 5.22:** Wide and narrow versions of a Bar Chart. [Images created with RespVis [Egger and Oberrauner 2024a] by the author of this thesis.]

## 5.6.2 Bar Base Series Modules

The Bar Base Series modules, located in the `ts/render/bar-series/bar-base/` directory of Listing 5.18, define the `BarBaseSeries` class, a super class of `CartesianSeries` and the abstract base class of all concrete bar series classes. The main purpose of `BarBaseSeries` is to act as a template for its concrete implementations. It enforces categorical x-axis values and provides a `getBars` method for deriving all Bar marker primitive objects from the data records.

Since each Bar Series variation has a different strategy for calculating the lengths and positions of its bars, each variation must provide a `getRect` method for retrieving `Rect` shape objects for the creation of its bars. This is enforced by the abstract `BarBaseSeries.getRect` method, which all descendants must implement.

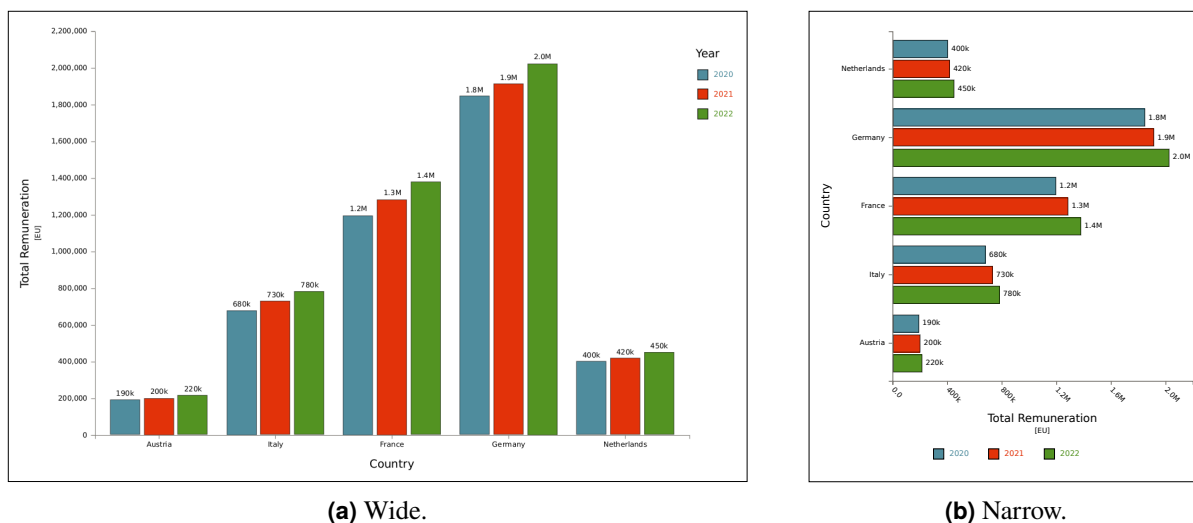
Another class defined in the Bar Base Series modules is `BarBaseResponsiveState`. It is a derived class of `CartesianSeriesResponsiveState` and provides the `getBarBaseRect` method, which takes the index of a data record and returns the currently valid corresponding `Rect` shape object. This is achieved using a band scale to determine the width and position of a bar along its categorical Axis. The width and position of a bar along its numerical Axis is determined using a numerical scale. The `getBarBaseRect` method takes into account the responsively changing flip state of the Bar Series and applies different strategies for vertically and horizontally aligned bars.

`BarStandardSeries` represents an instantiable version of `BarBaseSeries` and directly calls `getBarBaseRect` to create bar markers. Figure 5.22 illustrates how a Standard Bar Chart can be rendered as a Column Chart for wide spaces and as a Row Chart for narrow spaces.

## 5.6.3 Bar Grouped Series Modules

A grouped bar chart requires data records to provide a secondary categorical value. Bars with the same primary category but different secondary category form a group and are located directly next to each other, which enables local comparison of the bars. Bars typically exhibit color coding based on the secondary category [Kirk 2019, pages 140, 141, 159].

The Bar Grouped Series module, located in the `ts/render/bar-series/bar-grouped/` directory of Listing 5.18, defines the `BarGroupedSeries` class, which is a concrete class derived from `BarBaseSeries`. It



**Figure 5.23:** Wide and narrow versions of a Grouped Bar Chart. [Images created with RespVis [Egger and Oberrauner 2024a] by the author of this thesis.]

only differs by its strategy for creating bars. Instead of one bar using the whole width provided by the corresponding band scale, the available space is split up between bars of the same primary but differing secondary category. This is achieved using another band scale responsible for splitting up the available space provided by the first band scale. The local position of the bars is thereby determined by the order of the secondary category. The position and width along the numerical axis stay the same as for a Standard Bar Series. A Grouped Bar Chart example is shown in Figure 5.23.

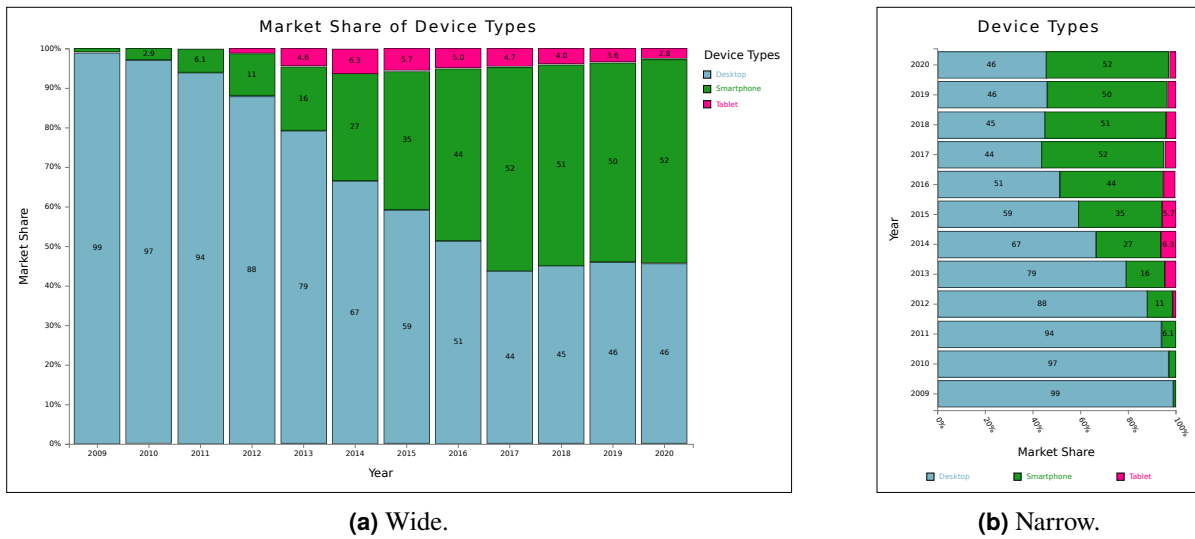
### 5.6.4 Bar Stacked Series Modules

A stacked bar chart requires data records to provide a secondary categorical value. Bars with the same primary but different secondary category are stacked on top of each other, representing parts of a whole bar. Bars typically exhibit color coding based on their secondary category [Kirk 2019, pages 140, 141, 159].

The Bar Stacked Series module, located in the `ts/render/bar-series/bar-stacked/` directory of Listing 5.18, defines the `BarStackedSeries` class, which is a concrete class derived from `BarBaseSeries`. Instead of one bar using the whole height, which is provided by the corresponding numerical scale, the available space is split up between bars of the same primary but differing secondary category. This is achieved by calculating the cumulative sums of the numerical values of data records with the same primary but different secondary category. Calculating cumulative sums is an important detail in the summation process, since the start position of a bar along the numerical axis is defined by the cumulative sum of all previous bars with the same primary category. An example of a Stacked Bar Chart is shown in Figure 5.24.

### 5.6.5 Bar Module

The Bar module, defined in the `ts/render/bar.ts` file of Listing 5.18, contains the definition of the marker primitive class `Bar`. Bar objects are designed to be bound to `<rect>` element selections upon which the D3 data join function `joinBarSeries` can be performed. Furthermore, bar markers hold additional data relevant for the creation of bar tooltips and bar labels.



**Figure 5.24:** Wide and narrow versions of a Stacked Bar Chart. [Images created with RespVis [Egger and Oberrauner 2024a] by the author of this thesis.]

### 5.6.6 Bar Label Module

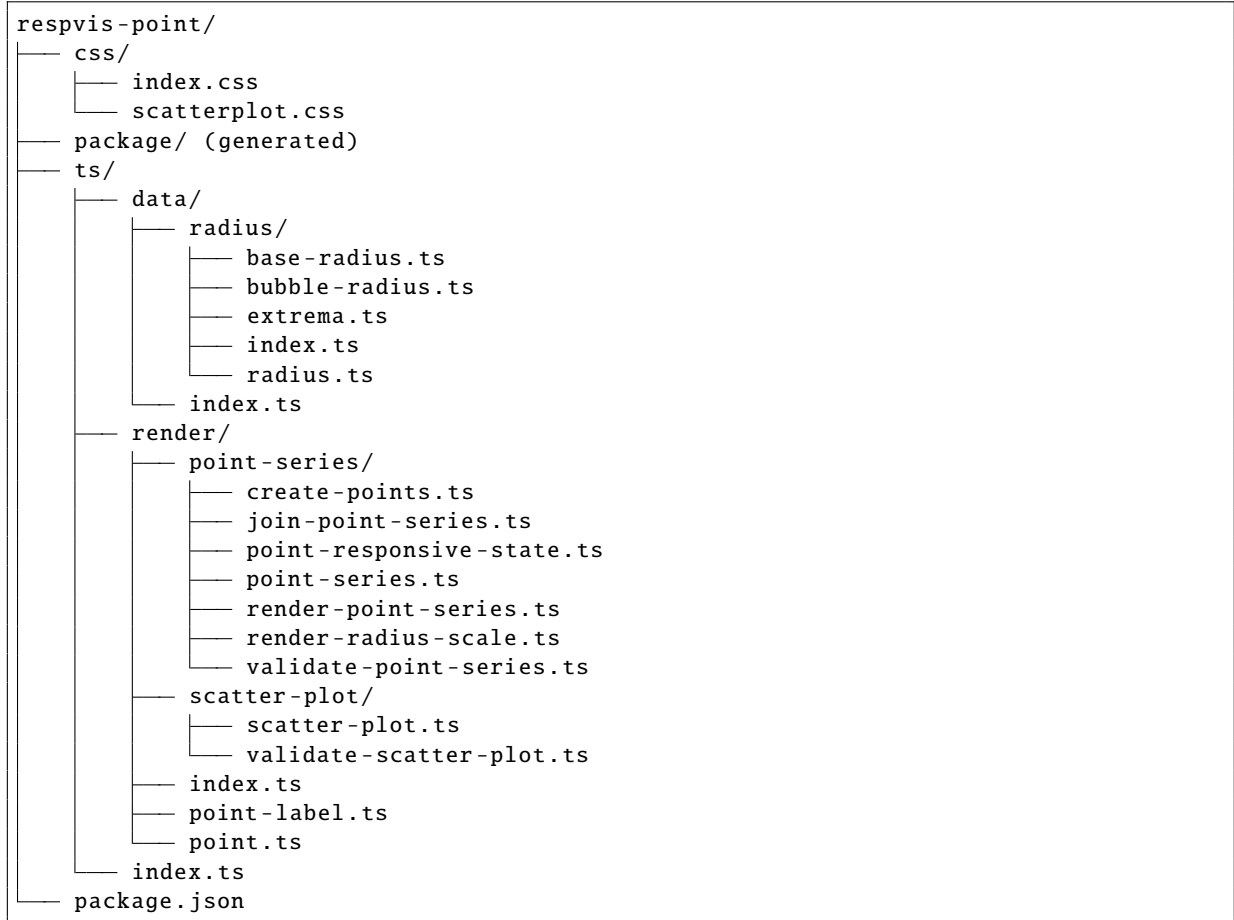
The purpose of the Bar Label module, which is defined in the `ts/render/bar-label.ts` file of Listing 5.18, is to declare interfaces for the creation and use of bar labels. This includes an interface for the user arguments, allowing library users to specify the desired labels, format function, and position strategies. Furthermore, an interface for valid bar label objects is defined. Labels generated by the `Bar.getLabel` method must adhere to this structure to make it possible to map bar markers to bar labels.

The four available position strategies for bar labels are center, positive, negative, and dynamic. Strategy center places the bar label at the center of each bar. If a chart creator chooses the positive or negative strategy, the bar label will be located at the positive or negative end of each bar, respectively. The underlying algorithm automatically takes into account if the Bar Series is flipped or scales are inverted. The dynamic position strategy extends the two previous approaches by automatically detecting if a bar represents a positive or negative value and positions the label at the appropriate end. Figure 5.19 demonstrates the dynamic position strategy for a Standard Bar Chart with positive and negative values.

## 5.7 RespVis Point

Scatter plots visualize the relationship between two numerical data dimensions by plotting points in a cartesian coordinate system. They may support the categorization of data records by mapping distinct color attributes to the points. Another option of using colors in scatter plots is to introduce an additional dimension by applying sequential color encoding. Also the size of the points can be encoded to add a further dimension. This variation of a scatter plot is called a bubble chart. Scatter plots with one categorical and one numerical dimension are called dot plots and serve the purpose of representing comparable distributions [Kirk 2019, pages 152, 166–177]. All the discussed scatter plot variations are supported by `respvis-point`, RespVis' sub-package for creating scatter plots. The directory structure of the package is shown in Listing 5.19.





**Listing 5.19:** The file and directory structure of the respvis-point sub-package.

### 5.7.1 Scatter Plot Modules

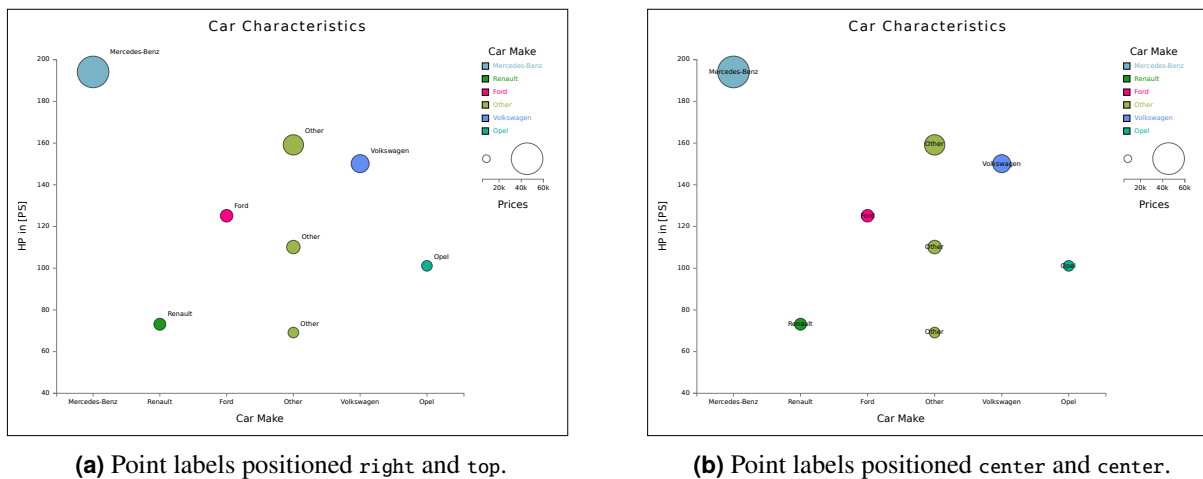
The Scatter Plot modules, located in the `ts/render/scatter-plot/` directory of Listing 5.19, enable the creation of charts with all the discussed features by providing the `ScatterPlot` class. The `ScatterPlotUserArgs` interface acts as a contract between `RespVis` and a chart creator when creating a new `ScatterPlot`. The `validateScatterPlot` function returns a validated `ScatterPlotData` object in the validation phase of the chart.

The render routine of the Scatter Plot is defined in the `renderContent` method. This method invokes the render functions of the incorporated mixins `DataSeriesChartMixin` and `CartesianChartMixin` to render all cartesian chart components. Calling the function `renderScatterPlotContent` completes the render process by deriving all `Point` data objects from the underlying `PointSeries`, binding the data objects to `<circle>` elements, and subsequently calling the `joinPointSeries` function.

### 5.7.2 Point Series Modules

Point Series are the visualization of related data records as points in a chart. The Point Series module, located in the `point-series/` directory shown in Listing 5.19, provides concrete implementations for the abstract constructs defined in the `Cartesian Series` modules. Apart from the characteristics of all `Cartesian Series`, a `Point Series` additionally provides:

- a `PointSeriesData.radii` property, defining the potentially responsive radii of the `Point Series`.



(a) Point labels positioned right and top.

(b) Point labels positioned center and center.

**Figure 5.25:** Different point label position strategies are provided to chart creators. [Images created with RespVis [Egger and Oberrauner 2024a] by the author of this thesis.]

- a `PointSeries.responsiveState` property for maintaining responsive properties specific to `PointSeries`. Currently, this solely includes updating and interpolating the `radii` property.
- a `createPoints` function for creating an array of point markers from a `PointSeries` object. The function takes into account the current filter, zoom, and category state and returns all relevant points, either as multiple arrays grouped by categories, or as a single array.
- a `joinPointSeries` function taking care about updating the enter, update, and exit selections of a `PointSeries` using the bound data objects to perform the necessary DOM manipulation operations on the `<circle>` elements.
- a `renderRadiusScale` function for adding a radius scale to the `Legend` in case a chart creator specified a bubble radius. If the bubble radius is responsive, the corresponding `Size Legend` adapts automatically.

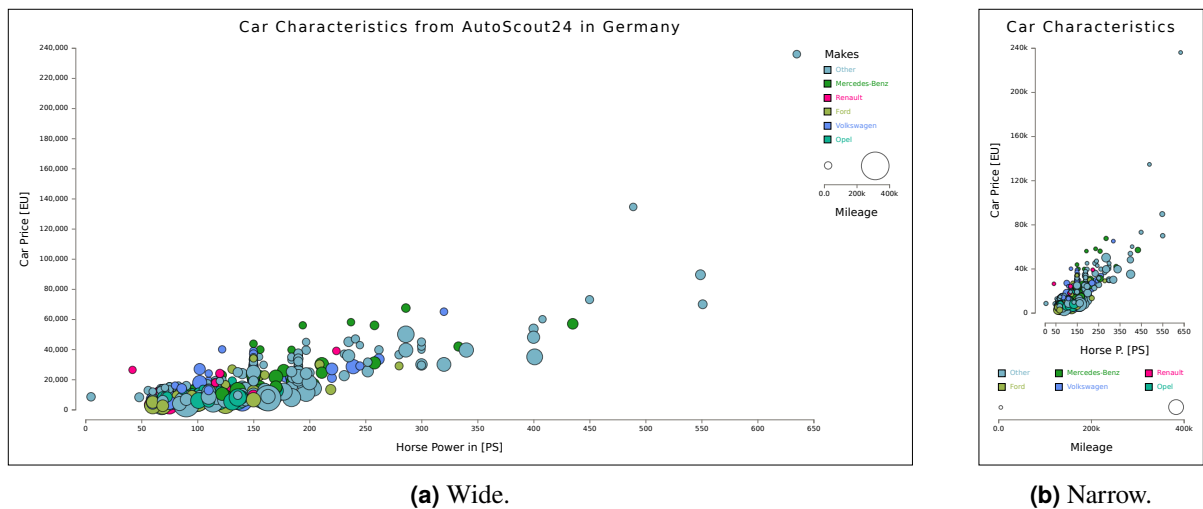
### 5.7.3 Point Module

The `Point` module, defined in the `ts/render/point.ts` file of Listing 5.19, contains the definition of the marker primitive class `Point`. `Point` objects are designed to be bound to `<circle>` element selections upon which the D3 data join function `joinPointSeries` can be performed. Furthermore, point markers hold additional data relevant for the creation of point tooltips and point labels. RespVis v3 only supports circles as markers for data points. In future, other kinds of markers could be supported, including triangles, squares, crosses, and potentially custom shapes.

### 5.7.4 Point Label Module

The purpose of the `Point Label` module, which is defined in the `ts/render/point-label.ts` file of Listing 5.19, is to declare interfaces for the creation and use of point labels. This includes an interface for the user arguments, allowing chart creators to specify the desired labels, format function, and position strategies. Furthermore, an interface for valid point label objects is defined. Labels generated by the `Point.getLabel` method must adhere to this structure to make it possible to map point markers to point labels.

The available horizontal position strategies for point labels are `left`, `center`, and `right`, and the available vertical position strategies are `top`, `center`, and `bottom`. Horizontal and vertical position strategies can be mixed arbitrarily. Figure 5.25 illustrates two possible combinations.



**Figure 5.26:** The radii of data points in a Bubble Chart can be responsively linearly interpolated when transitioning between wide and narrow spaces. [Images created with RespVis [Egger and Oberrauner 2024a] by the author of this thesis.]

### 5.7.5 Radius Modules

The Radius Modules, located in the `ts/data/radius/` directory of Listing 5.19, contain the definition of two radius types used for the creation of Point Series. The first one is the `BaseRadius` type, which expects radii to be specified as numbers, effectively leading to the same radius for all points. Since `BaseRadius` is a breakpoint property, the specification of different radii for different breakpoints is possible, leading to a responsive point radius based on the respective layout width. The radius length in between the breakpoints is linearly interpolated.

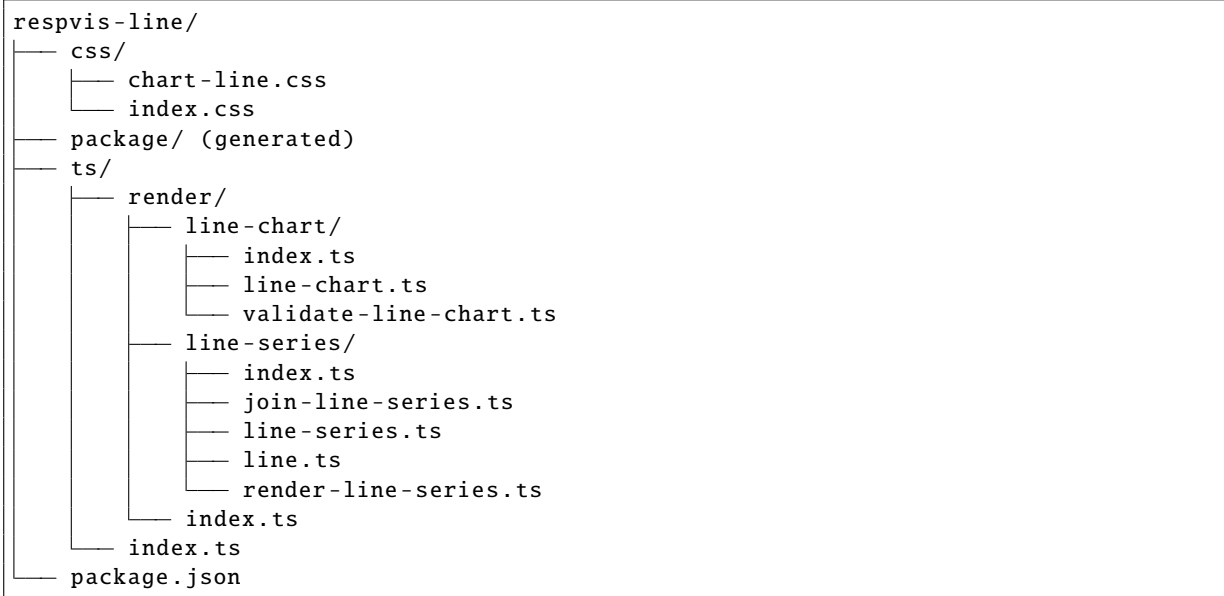
The second radius type is `BubbleRadius`, which requires the definition of a corresponding Axis and extrema. Like `BaseRadius`, `BubbleRadius` is a breakpoint property and enables the creation of responsive, linearly interpolated bubble radii, which can be seen in Figure 5.26. Since bubble radius domains are numerical, they can also be filtered in the Filter Menu of the Toolbar.

## 5.8 RespVis Line

Line charts are typically used to visualize the trend of a numerical variable (y-axis) over a temporal variable (x-axis) in a cartesian coordinate system. Another option is to use categorical data for the x-axis which results in the chart communicating a similar message to a bar chart (comparison of categorized items by a numerical dimension). In line charts, data is visualized by plotting related data points as markers and connecting them with a polyline, resulting in a related sequence of values. Including multiple independent line sequences is typically achieved via categorical color encoding [Kirk 2019, page 171]. All the discussed line chart variations are supported by `respvis-line`, RespVis' sub-package for creating line charts. The directory structure of the package is shown in Listing 5.20.

### 5.8.1 Line Chart Modules

The Line Chart modules, located in the `ts/render/line-chart/` directory of Listing 5.20, enable the creation of Line Charts by defining the `LineChart` class. The `LineChartUserArgs` interface acts as a contract between RespVis and a chart creator when instantiating a new `LineChart`. The `validateLineChart` function returns a validated `LineChartData` object in the validation phase of the chart. The render routine of the Line Chart is defined in the `renderContent` method. This method invokes the render functions of the incorporated



**Listing 5.20:** The file and directory structure of the `respvis-line` sub-package.

mixins `DataSetChartMixin` and `CartesianChartMixin` to render all cartesian chart components. Calling the function `renderLineChartContent` completes the render process by rendering a `Line Series` component. Figure 5.27 illustrates how a `Multi-Series Line Chart` is rendered for wide and narrow spaces.

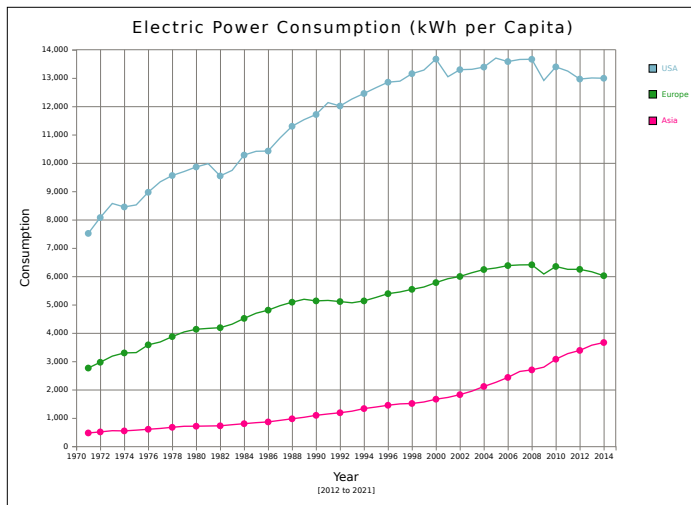
### 5.8.2 Line Series Modules

The `Line Series` Modules, located in the `ts/render/line-series/` directory of Listing 5.20, define the `LineSeries` class, which extends the `PointSeries` class. A `Line Series` component consists of two series components. The first one contains the point markers, where markers of the same category are grouped in dedicated `<g>` elements. The creation of the markers is achieved reusing the `createPoints` function located in the `respvis-point` package. The second component contains polylines of all categories, with one polyline connecting all markers of one category. The creation of the second component is conducted in the functions `renderLineSeriesLines` and `joinLineSeries`. The `renderLineSeriesLines` function is responsible for deriving `Line` objects from the previously created `Point` objects by extracting positions, keys, and style classes and passing the generated data objects to the second function. Finally, the `joinLineSeries` takes care of performing a D3 data join of the created `Line` objects with `<path>` elements and setting all attributes of the created elements.

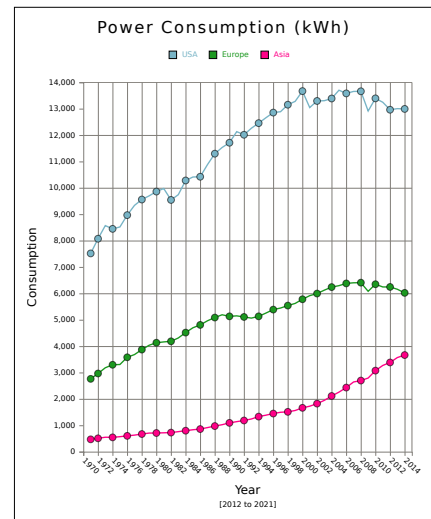
## 5.9 RespVis Parcoord

Parallel coordinate charts are useful tools for visualizing multivariate data [Ribecca 2024; Inselberg 2009]. Each dimension of a dataset is represented by a dedicated axis, scaled according to the domain values of the variable. The axes are aligned in parallel, either all horizontally or all vertically, such that every two neighboring axes can be connected by straight line segments. A data record is represented by a polyline, i.e. multiple, connected, straight line segments, touching each axis once at the corresponding data value.

Parallel coordinates charts tend to become cluttered and confusing if too many records are included. Interactivity can be used to ameliorate this. Filter interactions can be used to reduce the number of displayed data records to overcome the problem. Another way of coping with overplotting would be to provide interactive zoom for an axis, to focus in on a particular range of interest.



(a) Wide.



(b) Narrow.

**Figure 5.27:** Wide and narrow versions of a Multi-Series Line Chart. [Images created with RespVis [Egger and Oberrauner 2024a] by the author of this thesis.]

```

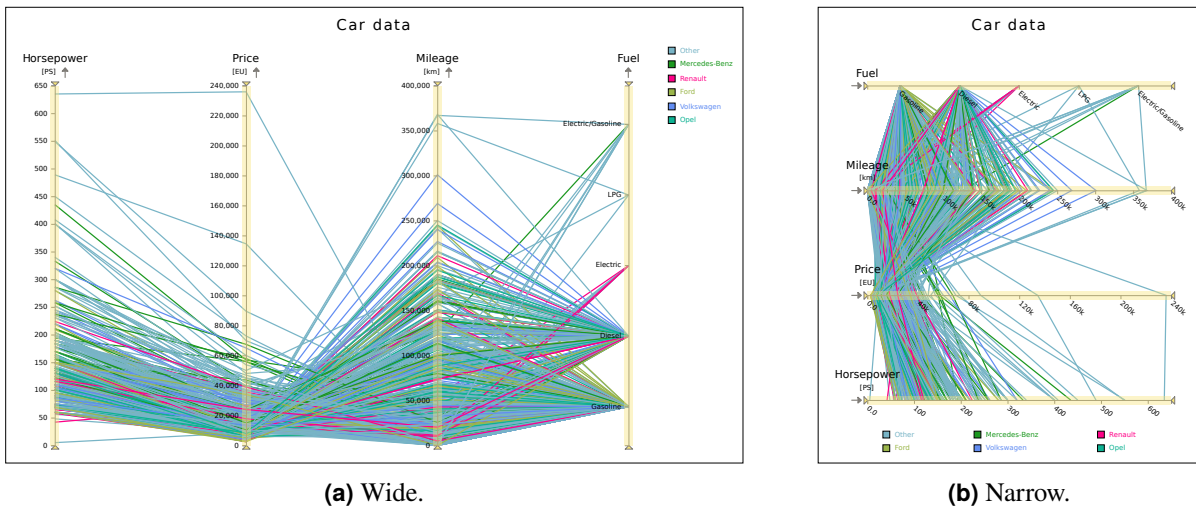
respvis-parcoord/
├── css/
│   ├── chart-parcoord.css
│   └── index.css
├── package/ (generated)
├── ts/
│   ├── render/
│   │   ├── parcoord-chart/
│   │   ├── parcoord-series/
│   │   ├── index.ts
│   │   └── validate-keyed-axis.ts
│   └── index.ts
└── package.json
    
```

**Listing 5.21:** The file and directory structure of the respvis-parcoord sub-package.

If too many dimensions are contained in a dataset, it is useful to be able to (temporarily) hide one or more dimensions. To compare two dimensions, they must be next to one another in the visualization. Hence, a chart end user must be able to reposition axes within the visualization; typically by drag-and-drop. To visually confirm a suspected correlation, it is often useful for the end user to be able to invert an axis. All the discussed parallel coordinates features are supported by respvis-parcoord, RespVis' sub-package for creating parallel coordinates charts. The directory structure of the package is shown in Listing 5.21.

### 5.9.1 Parallel Coordinates Chart Modules

The purpose of the Parallel Coordinates Chart modules, which are located in the ts/render/parcoord-chart/ directory of Listing 5.21, is to provide a comfortable way for chart creators of all experience levels to create customized Parallel Coordinates Charts. This is achieved by instantiating the ParcoordChart class and passing a desired configuration of the chart in form of a ParcoordUserArgs typed object along with an empty container element. The ParcoordUserArgs interface acts thereby as a contract between RespVis and



**Figure 5.28:** Wide and narrow versions of a Parallel Coordinates Chart. [Images created with RespVis [Egger and Oberrauner 2024a] by the author of this thesis.]

a chart creator. The constructor of the `ParcoordChart` class conducts the validation of the user arguments by calling the factory functions `validateParcoordChart` and `validateWindow` imported from the `Window` modules. The validated data objects are merged and the resulting object, representing the state of the chart, is bound to the `Window` component of the chart, which completes the validation phase.

The `ParcoordChart` class inherits functionality for all types of charts from the `chart` class, which is discussed in Section 5.3.4. This includes getter functions for frequently used selections, methods for setting up event listeners triggering layout updates and re-renders, and the `buildChart` method, which must be called by a chart creator to start the render phase of a chart. The `ParcoordChart` class further incorporates the `DataSeriesChartMixin`, discussed in Section 5.3.4.2, to have access to methods for creating a `Legend` and a `Toolbar` component, setting up filtering, and then setting up highlighting listeners for its contained polylines (each implemented as a `Line Series` component). The methods are called in the render phase of the chart, concretely in the `renderContent` method, which also calls the render routines of the `Parallel Coordinates Series`. Once the render phase finishes, the initially empty container element contains a fully rendered `Parallel Coordinates Chart`. An example of a `Parallel Coordinates Chart` in wide and narrow versions is shown in Figure 5.28.

### 5.9.2 Parallel Coordinates Series Modules

The central part of the `Parallel Coordinates Series` modules, which are located in the `ts/render/parcoord-series/` directory of Listing 5.21, is the `ParcoordSeries` class. The main purpose of the class is to hold the data necessary for rendering a composite series component, consisting of a `Line Series`, where each line represents a data record, and an `Axis Series`, where each `Axis` represents one data variable. By implementing the `DataSeries` interface, which is discussed in Section 5.3.5, a `ParcoordSeries` object adheres to the general structure of all data series types. This includes maintaining two data properties of type `ParcoordSeriesData`, `originalData` and `renderData`, where the former represents the stable series state, and the latter is changed during each render cycle by applying filtering, zoom, and inversion to the original data. The `ParcoordSeries` class also contains a `responsiveState` property, which is responsible for updating and accessing responsive properties of the data.

The render routine of a `Parallel Coordinate Series` is defined in the `renderParcoordAxisSeries` function. The order of rendering matters, since the `Axes` should not be occluded by the lines. First, the `Line Series` component is rendered. This is achieved by transforming the data stored in `ParcoordSeriesData.axes` into

an array of `Line` objects, where one line connects all `Axes` from left to right. Since `Axes` can be moved by drag and drop, the visual and original order may differ. To ensure only neighboring `Axes` are connected by lines, `Axes` are iterated in visual order when calculating the line positions. The transformation process further ensures filtering is applied for data records, by not creating lines for records containing a filtered value. The render process of the `Line Series` is completed by performing a data join upon the created `Line` objects.

Next, the `Axes Series` component is rendered. The `ParcoordSeriesData.axes` property holds an array of `KeyedAxis` typed objects, upon which a data join is performed with `<g>` elements. `Axes` of this type are currently only part of the `respvis-parcoord` package and provide additional properties and methods for filtering the `Axis` itself via the `Filter Menu` and setting an active value range. After the data join, the content of the `Axes` is rendered using the render functionality provided by the `Axis` modules, which are discussed in Section 5.3.6. After the `Axes` are rendered, they are enhanced to provide the additional zoom, drag and drop, filtering, and inversion interactions. This is accomplished either by enhancing existing elements of the `Axis` components, or by the addition of interaction elements.

Zoom and pan is enabled by adding a corresponding event listener for the outermost element of an `Axis`, making it possible to apply the interactions for the whole area of the component. If a zoom interaction event is fired, a corresponding `ZoomTransform` object is part of the event parameters. The object represents the current zoom state of an `Axis` and is stored in the `ParcoordSeries.zooms` property, which stores the current zoom state of all `Axes` as an array. With zooming, `Axes` can be modified to show specific domain ranges of interest. With panning, the displayed domain range can be changed without changing the current zoom factor.

Dragging and dropping an `Axis` is enabled by adding a corresponding event listener to the `<g>` element containing the title and subtitle elements. If a drag event is fired, the `Axis` position is calculated based on the distance of the pointer position to the edges of the drawing area. Depending on whether the `Parallel Coordinates Series` is flipped or not, the `Axis` moves only horizontally or vertically. The relative distance to the top or left edge of the drawing area is updated in the `ParcoordSeriesData.axesPercentageScale` property, which is an ordinal scale holding the positions of all `Axes` as percentages relative to the drawing area bounds. If two `Axes` have the same position because of drag and drop, the position of the resting `Axis` is adjusted slightly such that two `Axes` never have the exact same position. By default, when a drop end event is fired, the space between all `Axes` is equally distributed and the `axesPercentageScale` property is updated accordingly. This behavior can be turned off by disabling the checkbox with the label `EquidistantAxes` in the `Chart Tool Menu`, which is described in Section 5.3.2. If deactivated, the distance between `Axes` remains unchanged when dropping an `Axis`.

Additional elements are included to provide filter interactions. Each `Axis` is equipped with a double-edged range slider comprised of two value limiters (triangles represented by `<path>` elements) and an enclosed range (a rectangle represented by a `<rect>` element), making it possible to control the range of active domain values on that `Axis`. Each limiter and the enclosed range can be moved by dragging. Position updates during dragging are achieved by event listeners. To make the control surface slightly larger than the graphical representation, additional background `<rect>` elements are included around each value limiter and the enclosed range to increase the area for detecting drag events.

Inversion of an `Axis` is possible by clicking its inversion arrow, which is implemented as a nested `<svg>` element containing a `<path>`. As for the limiter elements, a background element is included to increase the area for click events. Upon interaction, the arrow rotates and changes the corresponding `Axis` entry in the `ParcoordSeriesData.axesInverted` property, which is a boolean array. Hover interactions are indicated by dedicated cursor icons, which appear when hovering over corresponding interaction areas.





## Chapter 6

# Outlook and Future Work

Nowadays, data is available in multiple forms, and is collected more rapidly than ever before, leading to an enormous amount of data being collected and stored all over the world [Sethi 2024]. To make sense out of this vast amount of data, it must be presented in a human-readable way, which makes data visualization more important than ever. The web, which constantly advancing, is the main medium for the creation and distribution of data visualizations. Nowadays, modern CSS provides powerful layout techniques like size and style container queries, and CSS subgrid. SVG is evolving much more slowly than CSS. SVG 2.0 has been a Candidate Recommendation since 2018 [W3C 2018]. Some browsers have implemented parts of SVG 2.0, but support is currently patchy. A core feature coming with SVG 2.0 will be the adoption of the attributes `cx`, `cy`, `r`, `rx`, `ry`, `x`, and `y` as geometry attributes, which will make it possible to style them via CSS.

There are many concrete features, which could be implemented in future versions of RespVis beyond RespVis v3:

- Extending RespVis' chart API with the possibility to mix different types of series in one chart.
- In the case of overlapping elements, bringing currently highlighted elements to the foreground.
- Improving RespVis' current implementation of optional text wrapping, which is currently only applied to chart titles.
- Extending RespVis to provide an attribution feature.
- Extending RespVis' inspection tool to render a vertical and a horizontal line for cartesian charts. The lines should start at the horizontal and vertical axis respectively, and end at the current pointer position. The lines will give a visual reference for aligning and comparing marker primitives.
- Adding other kinds of markers for data points, such as triangles, squares, crosses, and potentially custom shapes.
- Extending RespVis' scatter plot API to support optional hierarchical clustering for intersecting markers. The cluster information can be displayed in a separate window at the side or bottom (mobile) of a web site. A button can be provided for manually splitting clusters up into original points or merging points into clusters.
- Extending the RespVis API to handle multiple category arrays for axes. Categorical axes can switch their domain by swiping, as implemented in the scatter plot example of Andrews [2018a].
- Extending the RespVis API to handle multiple category arrays for data series. The currently used category can be specified when navigating to the toolbar settings.

- Extending the RespVis toolbar to provide a fisheye view tool, as implemented in the scatter plot example of Andrews [2018a].
- Displaying explanations about charts in general, chart interactions, and custom descriptions of a chart creator. These explanations can be displayed in a separate window at the side or bottom (mobile) of a web site. The visibility of the window can be toggled by a small button in the toolbar.
- Extending RespVis' line chart to support auto interpolation when hovering over a line segment to estimate values between two data points.
- Extending RespVis with CSS Variables indicating the current transition level between two breakpoints in percent from 0 to 1. This is useful for interpolating e.g. font sizes between two breakpoints in a controlled way.
- Extending RespVis' toolbar with an option for toggling the visibility of either categorical or sequential color encoding (if both are defined by the chart creator).
- Improving RespVis' default styling of the toolbar on mobile devices. A viable option would be to let tools slide in vertically instead of horizontally.
- Extending RespVis' download tool by providing an option to remove interaction elements like the range sliders used in the parallel coordinates chart.
- Adding the option to download a chart as a PNG file, in addition to SVG.

Since the implementation of RespVis v3 mainly focused on refactoring and documentation, improving and unifying the existing API, and adding many features to existing charts, the only new chart type is the parallel coordinates chart. There is plenty of room for the implementation of further new chart types like pie charts, area charts, heatmaps, and many more. The accomplishments of RespVis v3, including adding the first non-cartesian chart type with parallel coordinates charts, and providing reusable code templates for RespVis charts, will make the implementation of such charts much easier. Also, the creation of packages for new charts is now more conveniently achievable, without further major restructuring or refactoring.

Regarding the usage of modern CSS technologies, future versions may be able to make use of the `@scope` selector, which allows for selecting elements in DOM subtrees without increasing specificity [MDN 2024a]. The `@scope` selector will be useful for making RespVis' default CSS easily overridable. CSS style container queries, once they are supported by all major browsers (at time of writing, support is still missing in Firefox), will greatly ease the process of writing CSS for chart creators. They are discussed in Section 2.12.2.

RespVis' layout mechanism may or may not be improved in future versions of RespVis. While the current mechanism works perfectly fine, it is rather complex and comes with computational costs, as explained by Oberrauner [2022b, page 91]. In certain cases, it also makes writing CSS more verbose, since certain styles need to be applied to the SVG elements, and others to the replicated HTML elements.

## Chapter 7

# Concluding Remarks

This thesis presented RespVis v3, an open-source JavaScript library for creating responsive visualizations [Egger and Oberrauner 2024b]. Many major changes and improvements were made in the course of this work. First, all charts now have a consistent, structured calling interface with fully typed arguments, so chart creators can more easily instantiate a particular chart. A core component of RespVis is its custom layouter, which enables the use of powerful CSS layout techniques like CSS Flexbox and CSS Grid in SVG namespaces. The custom layout mechanism was extended to support alternating between SVG standard layout and RespVis' custom CSS layout within a chart. Responsive properties were introduced to allow chart titles, tick label orientation, and other elements of a chart to be specified in a responsive manner. The toolbar was restyled and toolbar interactions for all charts were improved and extended to support zooming, numerical filtering, and data inspection.

A significant improvement was the introduction of layout breakpoints within charts, which are specified as JavaScript objects and passed as an argument at chart creation, and are made available as CSS variables. Facilities were added to enable axis positioning and flipping, as well as the rotation of axis tick labels. Furthermore, downloading a chart to an SVG file was made much more configurable, with options to add margins, prettify the output, specify the number of decimal places, and choose how styling is embedded, among others. In terms of project improvements, RespVis v3 was split into modular sub-packages published separately, as well as still being available as a single monolithic package. Significant effort was also put into creating a comprehensive set of live documentation with Storybook.

Chapter 1 of the thesis introduced RespVis and responsive visualizations in general. Chapter 2 described the web technologies the library is built upon, and Chapter 3 discussed the field of responsive visualization. Chapter 4 covered previous versions of RespVis which served as a starting point for this work. RespVis v3, the current version of RespVis, was described in detail in Chapter 5. Finally, Chapter 6 presented some concrete ideas for future work. Four appendices contain a User Guide, a Chart Creator Guide, a Chart Developer Guide, and a Maintainer Guide.



# Appendix A

## User Guide

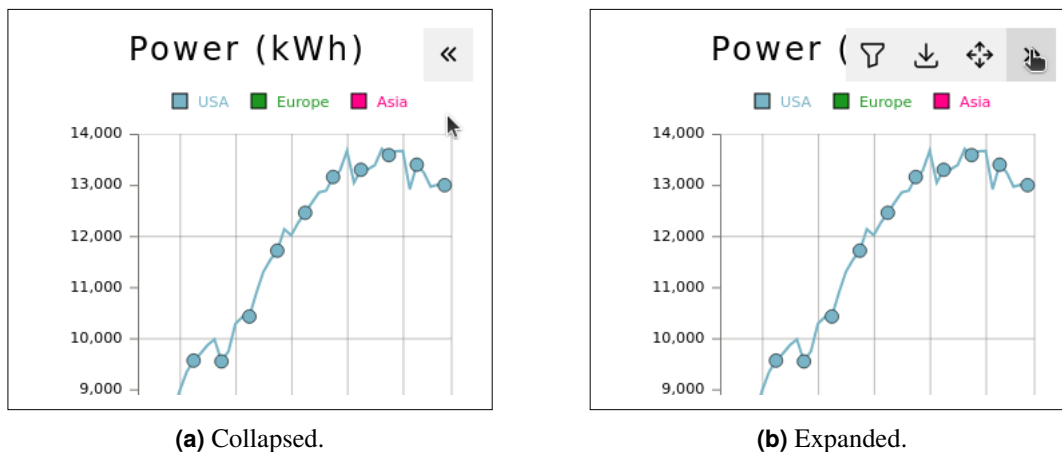
This guide presents detailed information about the usage of RespVis visualizations from the perspective of a *chart end user*. An end user is a person who does not necessarily know anything about the technical details of RespVis, but views and interacts with visualizations created with RespVis. The main focus of this guide lies on the interactions provided by RespVis, and how they can be used by an end user to explore a RespVis chart.

### A.1 PC and Mobile Interactions

Charts created with RespVis are inherently responsive. The shape and format of a chart can change according to how much space is available for the chart. The possible interactions can also vary depending on the type of device. Interactions on PCs (laptops and desktops) are sometimes different to interactions on mobile devices (tablets and smartphones):



- *Hover Interactions*: An entity is *hovered* by moving the pointer of a device above the entity. In PC environments, the position of the pointer is typically changed using a mouse or trackpad device. Hover interactions are typically not available on mobile devices. This limits the applicability and discoverability of hover interactions like displaying Tooltips or highlighting Data Series to a certain extent.
- *Click or Tap Interactions*: An entity is *clicked* by moving the pointer of a device above the entity and conducting a click interaction. In PC environments, this is typically achieved with mouse devices, by pressing and immediately releasing the left, middle, or right mouse key. On mobile devices, tapping an entity is typically considered equivalent to a single left-click.
- *Zoom Interactions*: On a PC device, zooming is often achieved by scrolling the mouse wheel up (zoom in) or down (zoom out). On a mobile device, zooming is typically achieved by pinching with two fingers on the touch screen: moving the two fingers apart zooms in, moving them towards each other zooms out.
- *Drag-and-Drop Interactions*: On a PC device, an entity is dragged by left-clicking it without releasing the mouse key and moving it. The entity is dropped when the mouse key is released. On a mobile device, an entity is dragged by tapping it without lifting the finger and then moving it. The entity is dropped when the finger is lifted.

For convenience, whenever this guide explains an interaction using the terms hover, click, zoom, or drag-and-drop, the corresponding interactions for both kinds of devices are referred to. Differences between PC and mobile devices are discussed explicitly.




**Figure A.1:** The two states of the RespVis Toolbar. [Screenshots taken by the author of this thesis.]


## A.2 Toolbar Interactions

Toolbars make visualizations more interactive by providing a set of tools, selectable from a bar. The RespVis Toolbar is located at the top right of a chart and can be expanded and collapsed by clicking the Toolbar Button,  or  respectively, as shown in Figure A.1. When expanded, the Toolbar displays three or four buttons, depending on the chart type, with icons indicating the corresponding tool. When a user hovers over a button, a Tooltip with the corresponding tool name is displayed. All tools are activated by clicking their corresponding button. From left to right, the tools are: Filter Tool, Download Tool, Inspection Tool, and for some charts the Chart Tool.

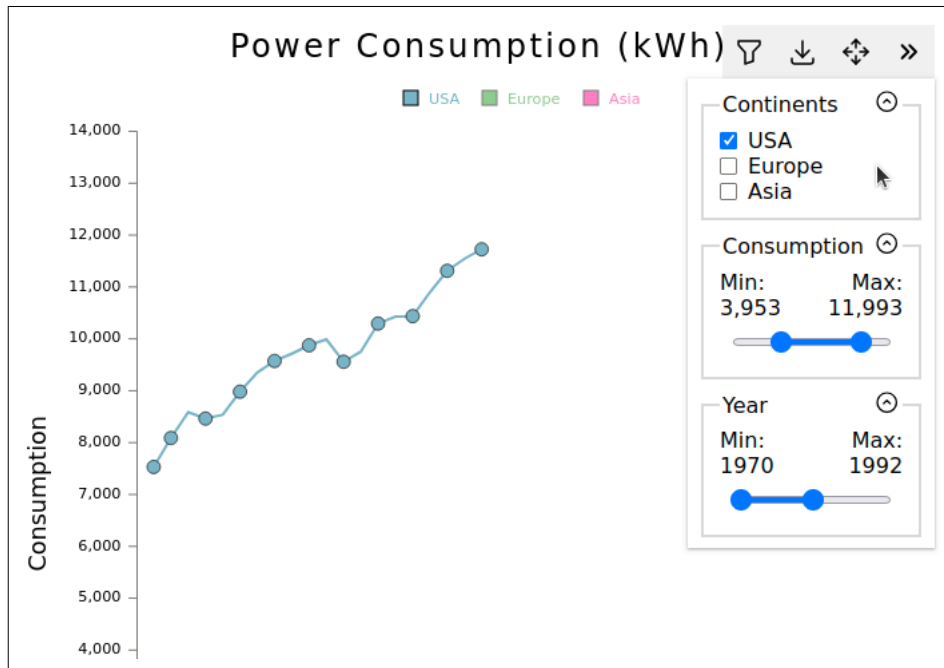
### A.2.1 Filter Tool

The possibility to interactively filter data can be very useful to locate data of interest in a visualization. The Filter Tool is activated with the Filter Button . When activated, the Filter Menu slides in and docks at the bottom of the Toolbar. The Filter Menu contains multiple fields giving control of the active filter settings. A field can be expanded and collapsed by clicking on its caption. Each field provides filter options for one data dimension. This dimension may be either categorical, numerical, or temporal. For a categorical dimension, a field contains a series of checkboxes giving control of each category. For numerical and temporal dimensions, the field contains a double range input, allowing an end user to specify the active range of values for the corresponding dimension. The double range slider is manipulated by dragging and dropping handles to set minimum and maximum values. Figure A.2 shows a Filter Menu for a multi-line chart containing all types of fields.

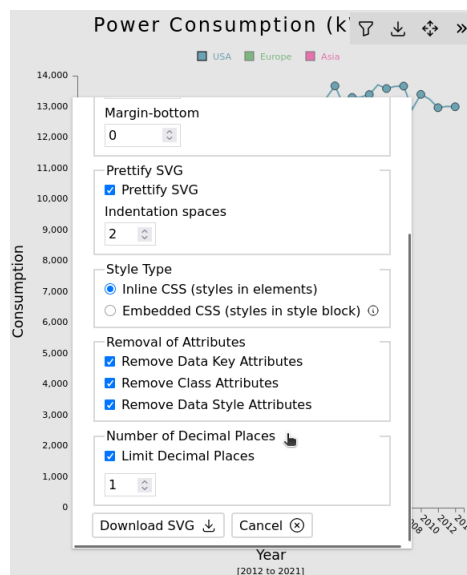
### A.2.2 Download Tool

The possibility to download and save a visualization can be of much value to an end user. RespVis supports this feature with the Download Tool. It enables an end user to download the current state of a chart as a static SVG (Scalable Vector Graphic) file. SVGs can be freely scaled without loss of quality. The Download Tool is activated with the Download Button . When the Download Tool is activated, a modal dialog pops up in the center of the page. Figure A.3 shows the modal with five fields for specifying the desired download options.

The first field allows margins to be added to the downloaded version of the chart. The second field contains download options regarding prettification. An end user can check or uncheck whether prettification is applied to the downloaded SVG. If unchecked, the option will not have any effect on the



**Figure A.2:** The Filter Menu for a multi-line chart with configurations for a categorical dimension (Continents), a numerical dimension (Consumption), and a temporal dimension (Year). [Screenshot taken by the author of this thesis.]



**Figure A.3:** The scrollable Download Modal with options for configuring the downloaded SVG file. [Screenshot taken by the author of this thesis.]

downloaded SVG. If checked, an end user can specify a desired number of indentation spaces for the code of the downloaded SVG.


The third field allows choosing between two approaches of including styles. The only important thing for an end user with no technical experience is that there are two options here. Choosing the `InlineCSS` option will always result in correct files, but may lead to large file sizes for complex visualizations. Choosing the `EmbeddedCSS` option does not guarantee flawless results, but leads to smaller file sizes for complex visualizations. In doubt, an end user should probably choose the first option, `Inline CSS`.

The fourth field contains options for the removal of `RespVis`-specific attributes, which are not of importance for an end user. Checking the options reduces the file size of the downloaded SVG.


The fifth field allows changing the number of decimal places used in the static SVG file. Checking the option and setting the number field to, say, 1 may be the best choice, since this reduces the file size of the downloaded SVG without generally affecting the appearance of the chart.

At the bottom of the modal, there are two buttons for canceling or confirming the download. If the cancel button is clicked, the download modal closes without downloading the chart. If the confirm button is clicked, the download modal closes and the chart is downloaded according to the specified download options.

### A.2.3 Inspection Tool

The possibility to inspect data at specific coordinates of a visualization can be useful to end users when interpreting data. `RespVis` supports this feature by providing the `Inspection Tool`. The `Inspection Tool` is activated with the `Inspection Button` . When the `Inspection Tool` is activated, the `Inspection Tooltip` is displayed. It is discussed in Section A.3.

### A.2.4 Chart Settings Tool

Custom options based on the chart type are supported by the `Chart Settings Tool`. It is only shown for charts which need to display additional options specific to their chart type. The `Chart Settings Tool` is activated with the `Chart Settings Button` . When activated, a modal pops up in the center of the page, providing chart-specific settings. Currently, only `Parallel Coordinates Charts`, discussed in Section A.6, make use of this tool.

## A.3 Tooltip Interactions

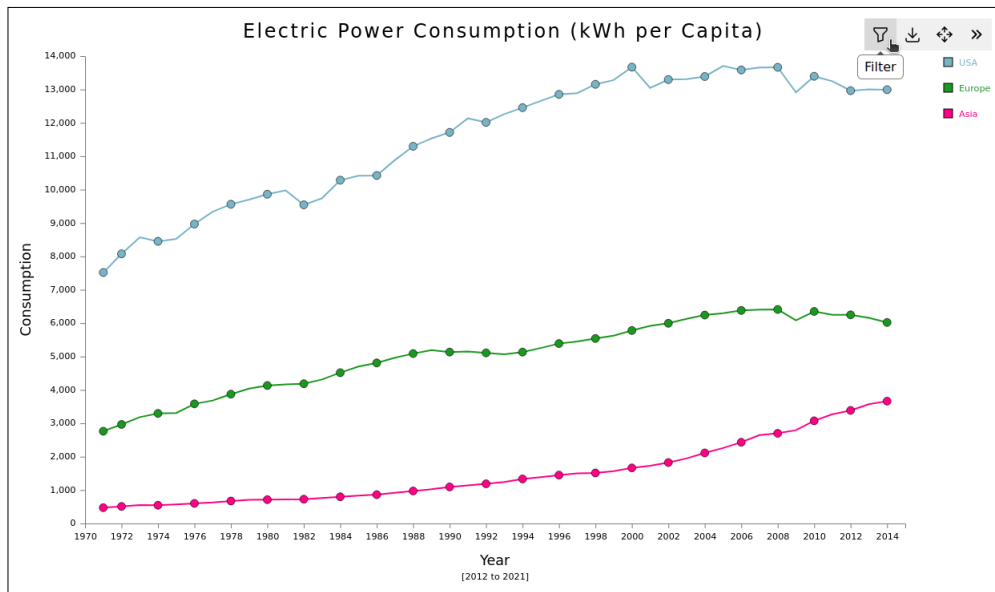
Tooltips are user interface (UI) elements used to provide additional contextual information about an entity. Typically, they are visually represented by small text boxes, which appear when hovering over certain elements. In `RespVis` there are currently three types of tooltips: `Toolbar Tooltip`, `Inspection Tooltip`, and `Data Series Tooltip`.

A `Toolbar Tooltip` appears when hovering over a tool in the `Toolbar`, as can be seen in Figure A.4. This `Tooltip` displays the name of the currently hovered tool.

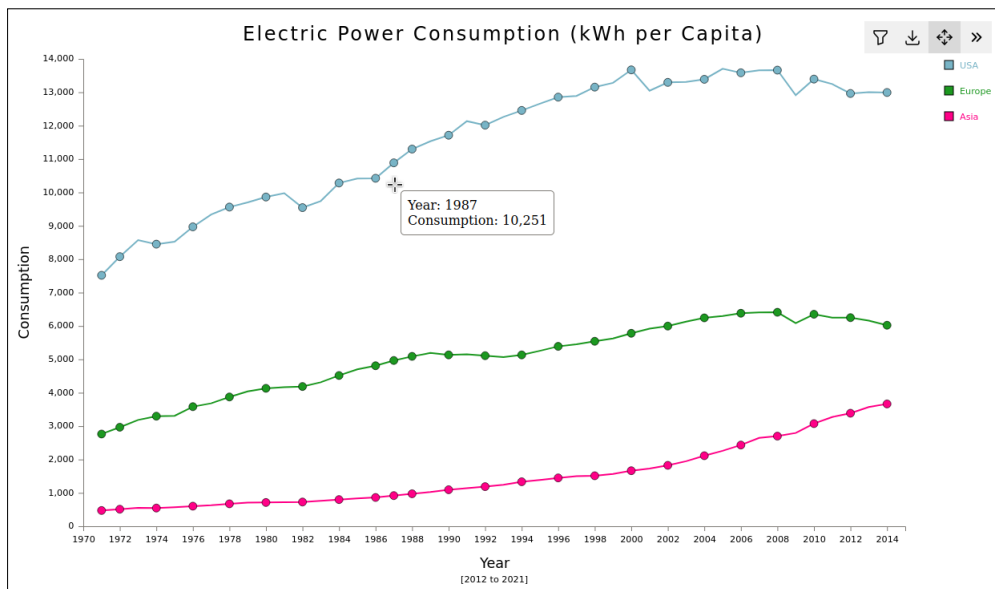
The `Inspection Tooltip` of a chart is activated and deactivated by clicking on the `Inspection Tool` in the `Toolbar`. If activated, the `Inspection Tooltip` displays information about the dimension values at the exact coordinates of the pointer. Figure A.5 shows an active `Inspection Tooltip` for a `Multi-Series Line` chart.

The `Data Series Tooltip` appears when hovering over a data marker, currently bars in bar charts and points in line charts and scatter plots. The content of a `Data Series Tooltip` is specified by the chart creator and may vary from chart to chart. In general, the `Data Series Tooltip` holds information about the currently hovered data point marker. Figure A.6 shows an example of a `Data Series Tooltip` when hovering over an element of a grouped bar chart.

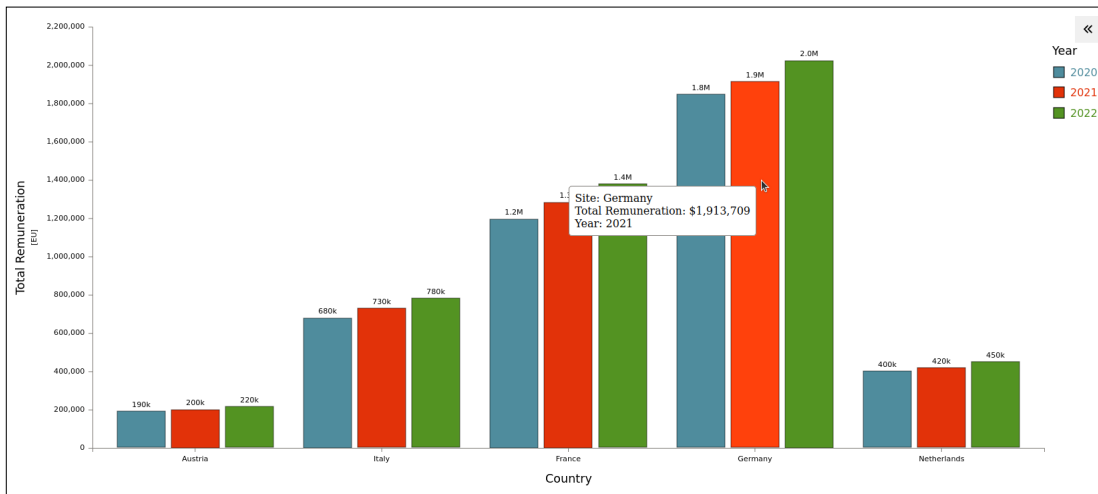




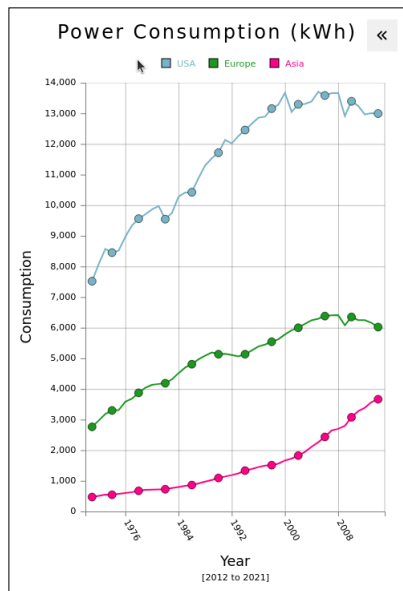
**Figure A.4:** The Toolbar Tooltip is used to display the name of the currently hovered tool in the Toolbar. [Screenshot taken by the author of this thesis.]



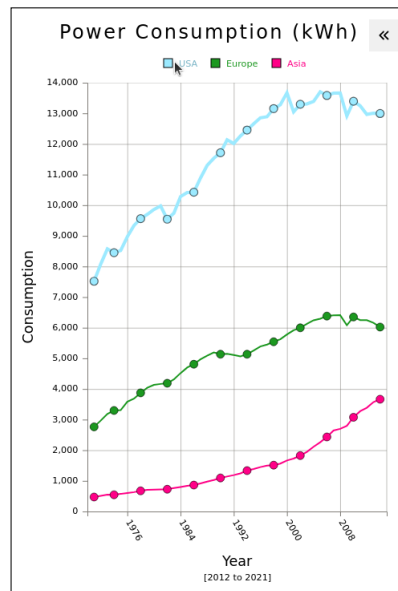
**Figure A.5:** The Inspection Tool provides the Inspection Tooltip, which display information about the dimension values at the current coordinates of the pointing device. [Screenshot taken by the author of this thesis.]



**Figure A.6:** The Data Series Tooltip is used to display information about the currently hovered data point marker. [Screenshot taken by the author of this thesis.]

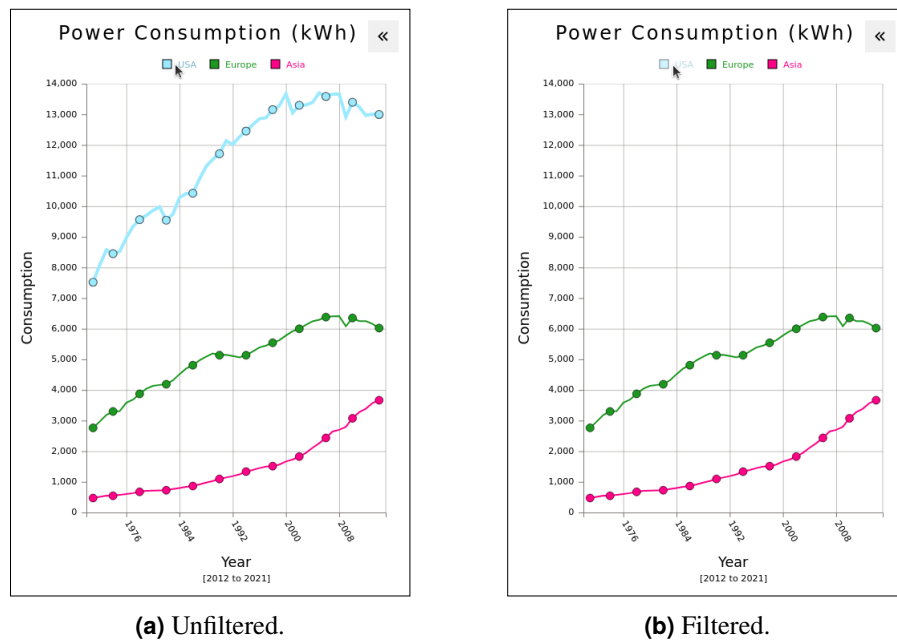


(a) Not highlighted.



(b) Highlighted.

**Figure A.7:** Hovering over a legend category entry highlights the corresponding marker primitives in a RespVis chart. Here, the data points in the USA category are highlighted. [Screenshots taken by the author of this thesis.]



**Figure A.8:** Clicking on a legend category entry filters out the corresponding marker primitives.  
 [Screenshots taken by the author of this thesis.]

## A.4 Legend Interactions

Legends are UI elements providing visual explanations of the colors, shapes, and sizes used in a chart [Kirk 2019, page 12]. In general, an end user interacts with a legend in a passive way by reading and comprehending it. RespVis provides two active legend interactions: data highlighting, and data filtering. Data highlighting is activated by hovering over displayed category items in a Legend. This leads marker elements (data points) with the same category to be highlighted in the chart, as can be seen in Figure A.7. Categorical filtering can be turned on and off by clicking category items in a Legend. This changes the current filter state of a category, leading to marker elements in the category to be filtered out (not displayed) or in, as can be seen in Figure A.8. Inactive categories are displayed half opaque in the Legend.

## A.5 Zooming

Zooming is a crucial tool for overcoming the problems of limited resolution and narrow screens and can be applied in multiple variations. In RespVis zooming allows an end user to control the magnification of special areas of interest [InfoVis:Wiki 2006]. It can be applied to all chart types, except Stacked Bar Charts. In cartesian charts, zooming can be applied in the whole coordinate system of a chart. The position of the pointer determines the center of the zoom, and thus which area is zoomed in or out, as shown in Figure A.9.

In Parallel Coordinates Charts, zoom and pan can be applied to Axes. With zooming, Axes can be modified to show specific domain ranges of interest. With panning, the displayed domain range can be changed without changing the current zoom factor. Zooming into an Axis also affects the line segments connecting the Axis to its neighboring Axes. If an Axis is zoomed into, all connecting lines falling out of the displayed axis range are hidden, as shown in Figure A.10. Zooming only applies to numerical data dimensions. Cartesian charts with only one numerical dimension will zoom only in one direction. The same holds true for Parallel Coordinates Charts, where zooming on a categorical Axis has no effect. The final decision whether zoom interactions are supported at all is up to the author of a chart.

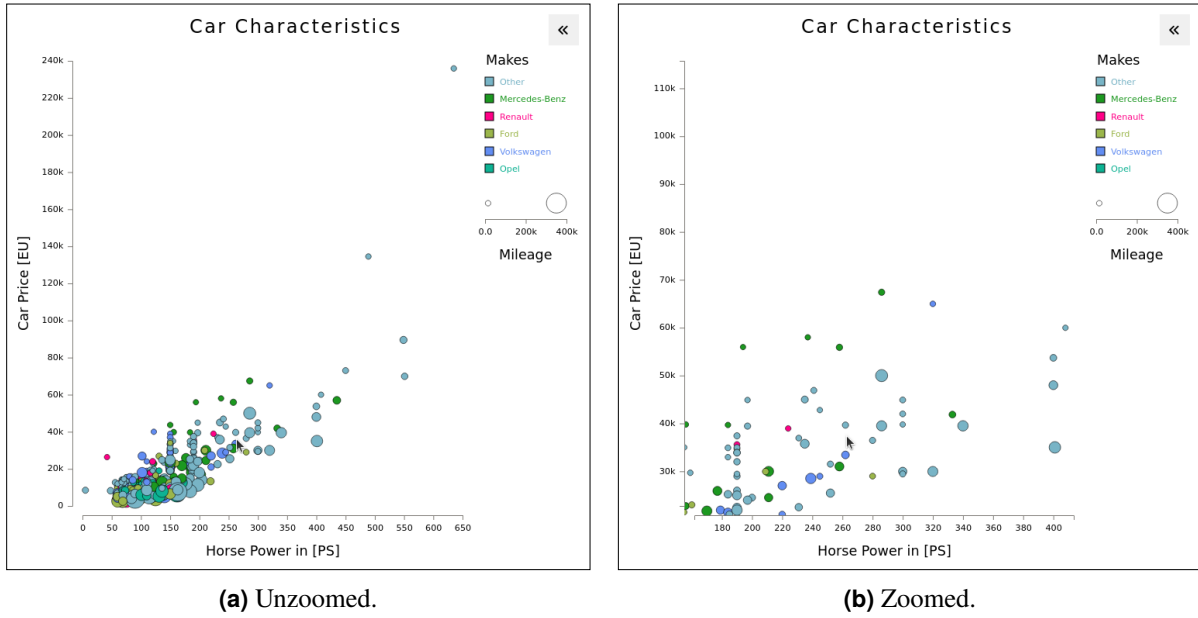


Figure A.9: A Cartesian Chart is zoomed into. [Screenshots taken by the author of this thesis.]

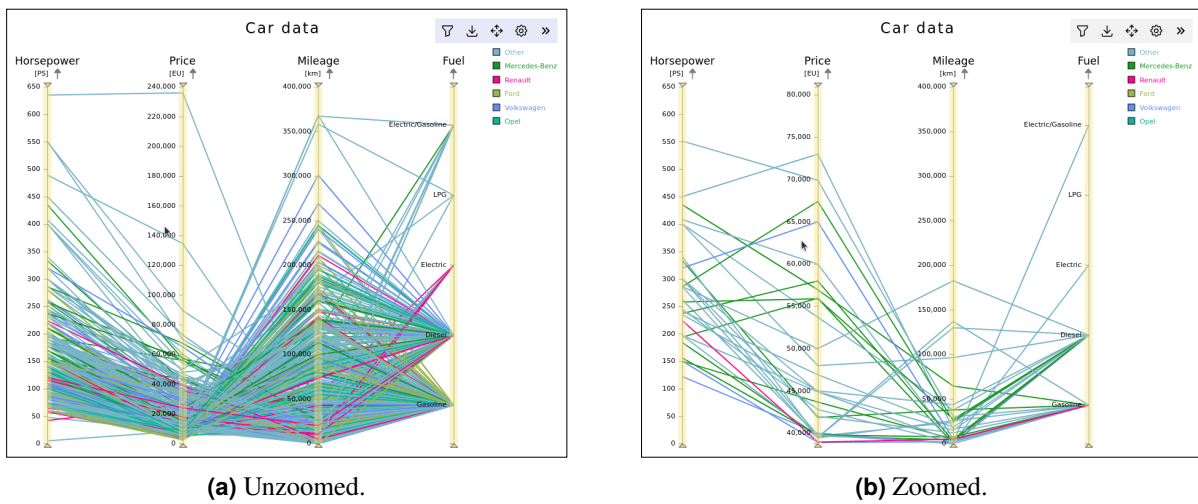
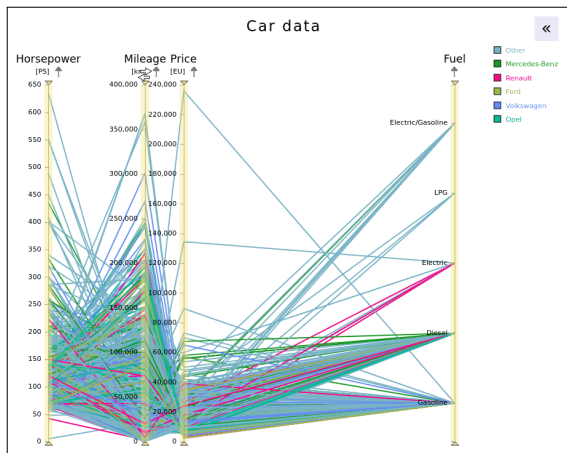
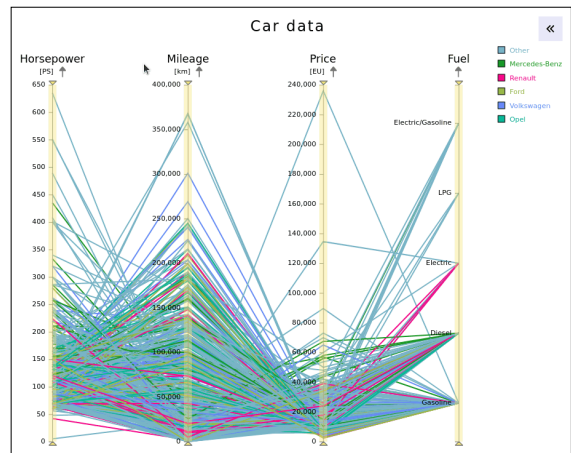


Figure A.10: The Price Axis of a Parallel Coordinates Chart is rescaled by zooming and panning. [Screenshots taken by the author of this thesis.]

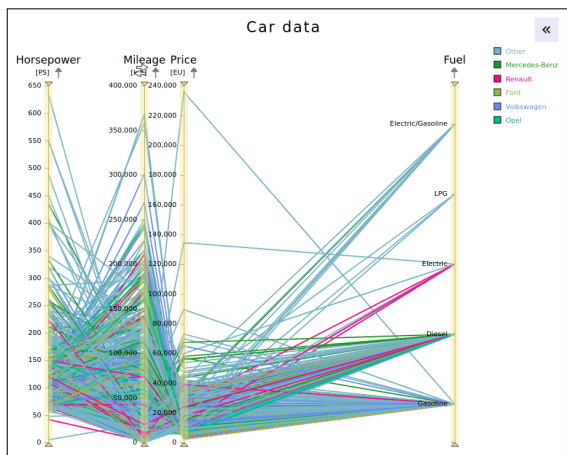


(a) Dragging Axis.

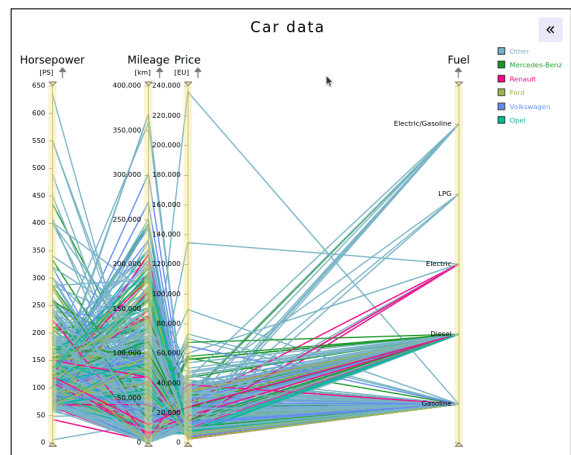


(b) Equidistant Axes after dropping.

**Figure A.11:** Equidistant Axes of a Parallel Coordinates Chart after dragging and dropping the Mileage Axis. [Screenshots taken by the author of this thesis.]



(a) Dragging Axis.

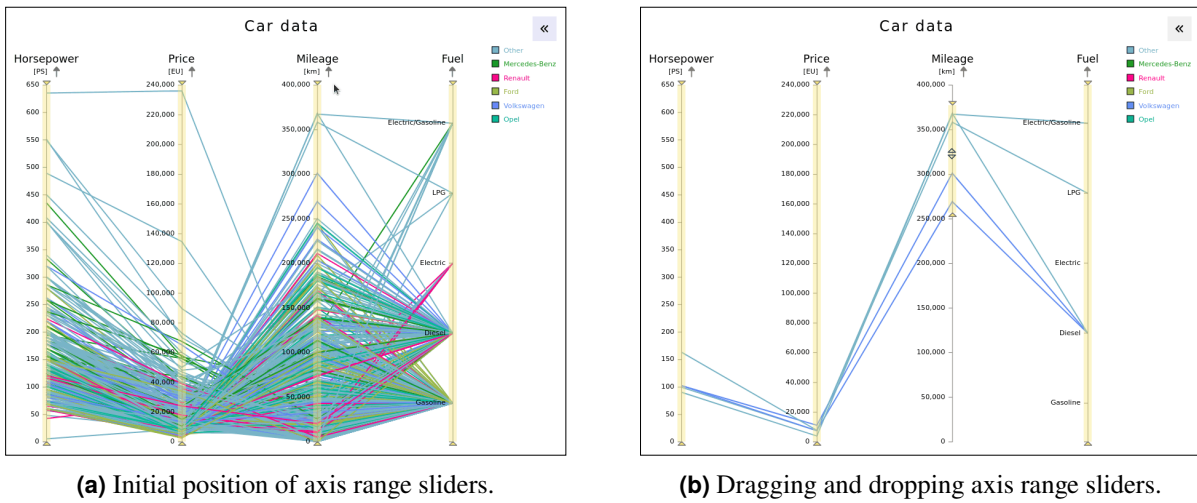


(b) Dropping Axis does not change axis position.

**Figure A.12:** Unchanged axis positions of a Parallel Coordinates Chart after dragging and dropping the Mileage Axis. [Screenshots taken by the author of this thesis.]

## A.6 Parallel Coordinates Chart Interactions

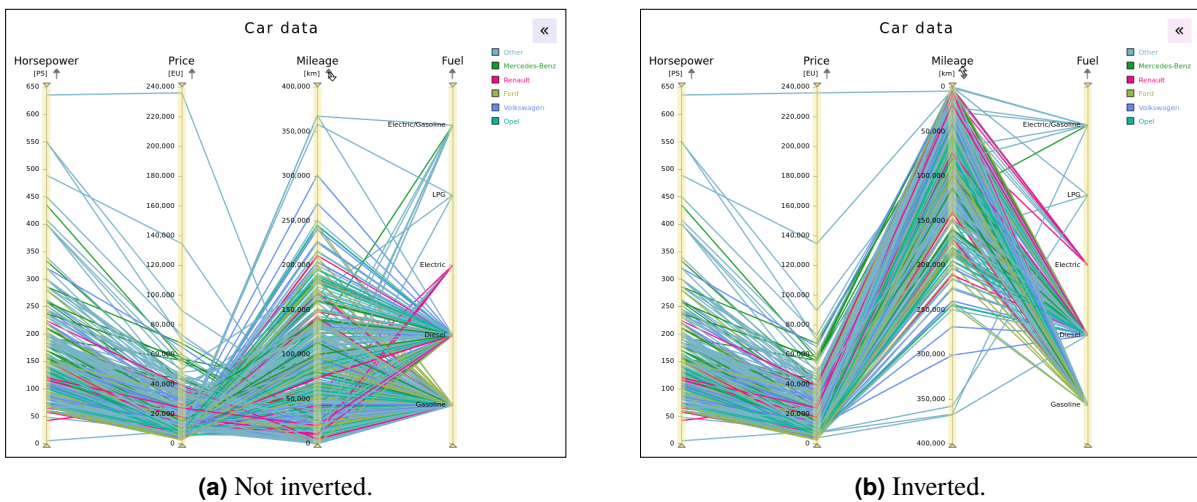
RespVis Parallel Coordinates Charts come with built-in interactivity to make their exploration easier: reordering axes, inverting axes, and filtering records. An Axis can be repositioned by dragging and dropping the title or subtitle element of the corresponding Axis. Depending on whether the chart content is flipped or not, the Axis moves only horizontally or vertically. An Axis can only be positioned inside the drawing area of a chart. When dragging an Axis and dropping it outside the drawing area, the Axis is positioned at the nearest edge of the drawing area. If two Axes have the same position because of drag and drop, the position of the resting Axis is adjusted slightly, such that two Axes never have the exact same position. By default, when an Axis is dropped, the space between all Axes is equally distributed, as shown in Figure A.11. This behavior can be turned off by disabling the checkbox with the label *EquidistantAxes* in the Chart Tool Menu, which is described in Section A.2.4. If deactivated, the distance between Axes remains unchanged when dropping an Axis, as can be seen in Figure A.12.



(a) Initial position of axis range sliders.

(b) Dragging and dropping axis range sliders.

**Figure A.13:** Filtering records of a Parallel Coordinates Chart by adjusting the double-edged range slider of an Axis. [Screenshots taken by the author of this thesis.]



(a) Not inverted.

(b) Inverted.

**Figure A.14:** Inversion of an Axis in a Parallel Coordinates Chart by clicking the Axis' inversion button, here for the Mileage axis. [Screenshots taken by the author of this thesis.]

Additional elements are included to provide filter interactions. Each Axis is equipped with a double-edged range slider comprising two value limiters (triangles) and an enclosed range (rectangle), making it possible to control the allowed range of domain values on that Axis. Each limiter and the enclosed range can be moved by dragging, as shown in Figure A.13. Inversion of an Axis is possible by clicking its inversion arrow. Upon interaction, the arrow rotates and the corresponding Axis is inverted, as can be seen in Figure A.14. Dedicated cursor icons appear when hovering over corresponding interaction areas, as an additional aid for desktop users. Hover cursors are typically not displayed on mobile devices.

## Appendix B

# Chart Creator Guide

This guide explains how to create charts in RespVis v3, and is addressed to *chart creators*. In the context of this thesis, a chart creator is someone who uses the RespVis API to create responsive visualizations. While a good understand of RespVis' technical details is advantageous for creating custom charts, it is not a requirement for making use of the charts already provided as part of the RespVis API. These are bar charts, scatter plots, line charts, and parallel coordinates charts.

### B.1 RespVis Patterns

A curated set of 16 responsive visualization patterns form the basis of RespVis v3 [Egger 2024a]. These patterns are partly based on the more abstract work of Kim et al. [2021], who list 76 design strategies for transforming wider visualizations into narrower ones. Ten of the patterns are adopted in the practical chart examples in this guide. The patterns are divided into two groups: seven visual patterns and three interaction patterns.

#### B.1.1 Visual Patterns

Visual patterns change the state and appearance of a visualization's components, to maximize the user experience depending on the available space:

- V1: Repositioning Element Labels
- V2: Using Tooltips Instead of Element Labels
- V3: Rotating Axis Tick Labels
- V4: Shortening Labels and Titles
- V5: Scaling Down Visual Elements
- V6: Hiding Elements and Labels
- V7: Rotating Chart 90°

##### B.1.1.1 V1: Repositioning Element Labels

Repositioning of labels can help avoid intersections and clutter. In web-based visualizations, this can be achieved using CSS media queries or container queries.

### **B.1.1.2 V2: Using Tooltips Instead of Element Labels**

If a visualization contains many data points, but has only limited space, a helpful pattern is to hide the labels of all elements and display an element's label upon hover or selection. In web-based visualizations, the CSS hover selector can be used to display a tooltip while hovering over an element with a pointer device. For touch devices, a single touch can be used to toggle the display of a tooltip. The disadvantage of this pattern is that it is not immediately obvious to users (discoverability).

### **B.1.1.3 V3: Rotating Axis Tick Labels**

An especially useful method for avoiding intersections and clutter of axis labels is to rotate the x-axis tick labels. As the available horizontal space decreases, this approach helps preserve more of the original axis label information (rather than thinning out or shortening the axis labels). The rotation of y-axis tick labels can also be considered, but is typically less useful.

For web-based visualizations, a possible way to achieve rotating tick labels is to use a combination of JavaScript and CSS. A JavaScript algorithm and event listeners are necessary to detect the currently appropriate angles of labels, while the styling of the labels themselves can be achieved via the CSS properties `rotate` or `transform: rotate()`. The advantage of this pattern is the preservation of the original axis information. On the downside, the axis labels are harder to read, since the natural reading direction is not retained.

### **B.1.1.4 V4: Shortening Labels and Titles**

A common technique for avoiding clutter and overlaps is to have different formats for labels for different space requirements. Numbers, for example, can be shown in full length if enough space is available (e.g. 2,200,000), but shortened to well-known abbreviated forms (e.g. 2.2M) when space is limited. Similar strategies can be applied to other types of text such as dates, organization names, and geographic locations. However, when using shortened labels in a visualization, any resulting information loss, such as that caused by the rounding of numbers, should also be considered. Shortened texts should use well-known and understandable formats, so as not to confuse users.

### **B.1.1.5 V5: Scaling Down Visual Elements**

Selected visual elements can be scaled down in size. This approach varies for different kinds of element. A bar element, for example, can be scaled in both horizontal and vertical directions without problems. The same holds for lines in a line chart, since these simply have to update their thickness and target points. A marker for a data point is more complicated, since it must retain its aspect ratio during scaling to avoid distortion.

Freely scaling down a selection of elements is applicable only to visualizations with a variable aspect ratio. Other types of visualization like pie charts, chord diagrams, and maps can only apply this kind of transformation to their elements when retaining their original aspect ratio.

The advantages of this technique are that much space can be saved without information loss, and smooth transitions via event listeners appear very natural. On the downside, the pattern is only applicable to visualizations with a manageable number of elements, since otherwise elements are already quite small even at larger widths. Another disadvantage is that line elements in line charts appear steeper at narrower widths and flatter at larger widths, affecting the perception of the original message.

### **B.1.1.6 V6: Hiding Elements and Labels**

One possibility to adapt a visualization to narrower widths is to remove some elements or labels completely. When applying this technique, care must be taken to not alter the original message of the visualization.



The advantage of this pattern is that an arbitrary amount of space can be saved by removing enough elements. However, this comes at the cost of information loss with respect to all the removed elements. When removing whole categories or dimensions, it is advisable to offer interactive possibilities, so the user can choose which dimensions or categories are of interest.

#### **B.1.1.7 V7: Rotating Chart 90°**

Transposing or rotating a chart by 90° can be a convenient way to align the dimension which requires more space vertically rather than horizontally. Even if vertical scrolling is required, it is much more acceptable than horizontal scrolling. The advantage of this pattern is that no information is lost by the transformation process. The main disadvantage is the major change of the visualization which may affect other ongoing transformations.

### **B.1.2 Interaction Patterns**

Interaction patterns support responsiveness by providing interactive functionality such as zooming and filtering:

- I1: Supporting Toolbar and Menus
- I2: Filtering Dimensions and Records
- I3: Supporting Zooming

By default, the patterns I1 and I2 are part of any RespVis visualization, while pattern I3 can optionally be implemented for all visualizations except Stacked Bar Charts.

#### **B.1.2.1 I1: Providing a Toolbar or Menu**

Interactivity bound to visual elements, such as hovering or a right-click context menu, suffers from poor discoverability. The user has to know such actions are possible or discover them by trial and error. A toolbar or menu, on the other hand, is visible to users, and its features can be explored. Typical actions provided by toolbars or menus include being able to download a chart as SVG, download the data as CSV, view the chart in full screen, view the data as a table, and show and hide specific records and dimensions in the data.

The advantages of this pattern are the theoretically unlimited interaction options that can be added to a visualization and the high likelihood of the toolbar being discovered by the user. Disadvantages of the pattern include the space needed for the additional control elements and the effort for the user to find them if they are hidden by default.

#### **B.1.2.2 I2: Filtering Dimensions and Records**

If space requirements are very tight, there is often no other solution than removing information from a visualization. However, when doing so it is a good idea to empower the user to choose which dimensions or records should be shown or hidden. The user may not be able to see all the data at once, but still has access to all information if necessary. Possible interaction elements for the filtering of data can be the legend of a chart, the elements themselves, or separate control elements such as dropdown menu.

#### **B.1.2.3 I3: Supporting Zooming**

Zooming is a crucial tool for overcoming the problems of limited resolutions and narrow screens. The standard approach, geometric zooming, allows a user to control the magnification of a visualization, and thereby trade the space needed for irrelevant information for more space for areas of interest.

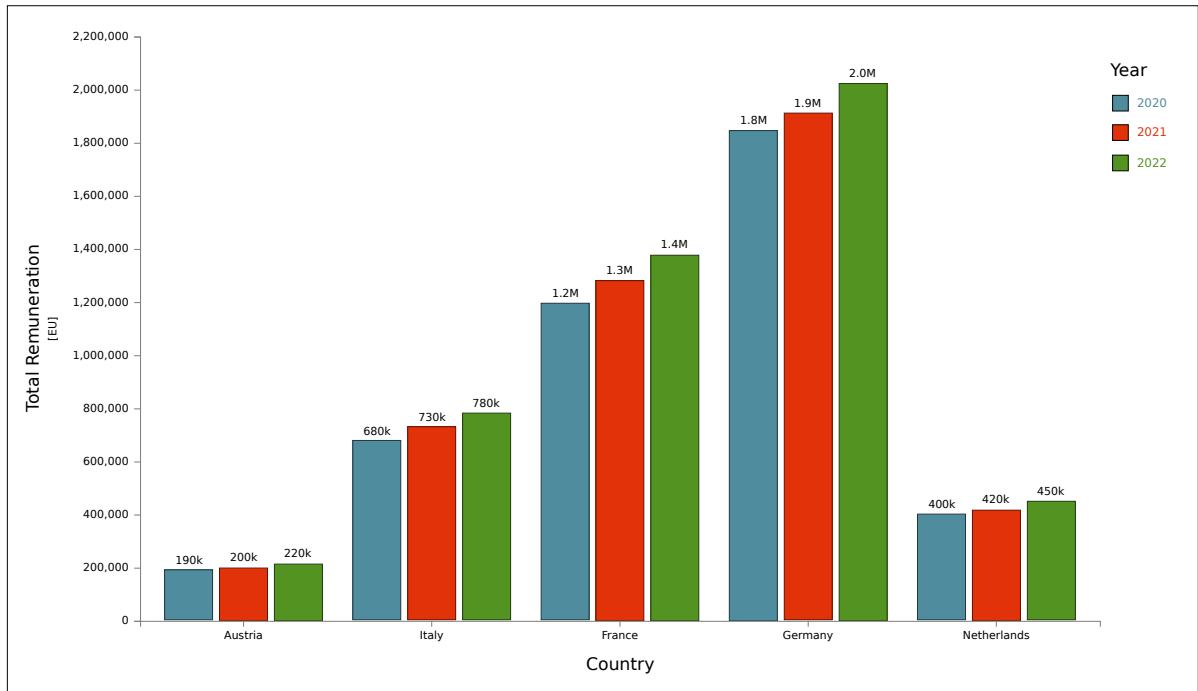
## B.2 Bar Chart

Bar charts are used as a means to compare a numerical variable for a collection of data records by visualizing bars in a cartesian coordinate system. The bar length represents the magnitude of the numerical variable, and the bar position represents the value of its categorical variable. Bars always rest on the categorical axis. If the categorical variable is displayed along the horizontal axis, the chart can be referred to as a column chart. If the other way around, the chart can be referred to as a row chart [Kirk 2019, pages 140–141, 159]. Since RespVis enables flipping all types of Data Series, both bar chart types are supported. RespVis also supports the creation of Grouped Bar Charts and Stacked Bar Charts, which are more complex variations of standard bar charts.

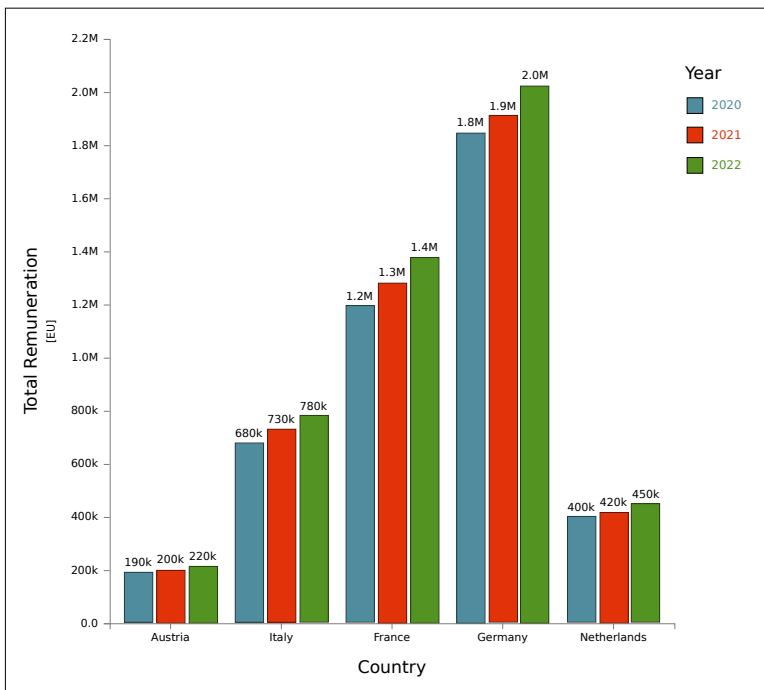
An implementation of a responsive Grouped Bar Chart can be seen in Figure B.1. It is implemented with the HTML markup shown in Listing B.1, the CSS shown in Listing B.2, and the TypeScript code shown in Listing B.3. The presented Grouped Bar Chart applies the following visual responsive patterns:

- *V1 Repositioning Element Labels*: RespVis comes with built-in position strategies for element labels. In the given example, a position strategy of `dynamic` is specified (Listing B.3, line 58). This results in labels being automatically positioned by RespVis. For Bar Charts, this is often already sufficient. In case of overlapping labels, a chart creator could reposition the labels manually using CSS.
- *V3 Rotating Axis Tick Labels*: When transitioning from a wide to a narrow view, the chart is flipped, meaning that X-Axis and Y-Axis switch places. To avoid intersecting labels of a horizontally positioned Y-Axis, the tick labels are rotated (Listing B.3, lines 82–86).
- *V4 Shortening Labels and Titles*: When transitioning from a wide to a narrow view, the labels of the Y-Axis (Listing B.3, lines 89–94) are shortened to avoid intersecting or cropped labels.
- *V5 Scaling Down Visual Elements*: When transitioning from a wide to a narrow view, the drawing area, the Axes, and the bar elements are scaled down automatically by RespVis during re-rendering.
- *V6 Hiding Elements and Labels*: When transitioning from a wide to a narrow view, the ticks of the flipped Y-Axis are thinned out to avoid intersecting or cropped labels. (Listing B.2, line 51).
- *V7 Rotating Chart 90°*: When transitioning from a wide to a narrow view, the chart is transposed, since the chart will now have more space vertically than horizontally (Listing B.2, lines 29–52), (Listing B.3, lines 44–47). This pattern is commonly applied for bar charts, because it prevents bar labels from intersecting with bar elements, leaves more space for the categorical axis, and ensures a minimum bar width.

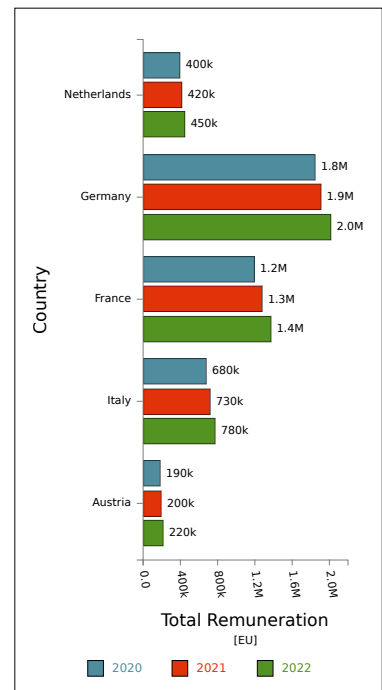
In addition, the presented Grouped Bar Chart also implements all three interactive patterns.



(a) Wide.



(b) Medium.



(c) Narrow.

**Figure B.1:** Wide, medium, and narrow versions of a Grouped Bar Chart. [Images created with RespVis [Egger and Oberrauner 2024a] by the author of this thesis.]

```
1 <!DOCTYPE html>
2 <html xmlns="http://www.w3.org/1999/xhtml" lang="en" xml:lang="en">
3 <head>
4   <title>RespVis - Grouped Bar Chart</title>
5   <meta name="viewport" content="width=device-width, initial-scale=1.0"/>
6   <meta charset="UTF-8"/>
7   <link rel="stylesheet" href="./libs/respvis/respvis.css"/>
8   <link rel="stylesheet" href="./grouped-barchart.css"/>
9 </head>
10
11 <body>
12 <h1>Grouped Bar Chart</h1>
13 <noscript>Please enable JavaScript!</noscript>
14 <div id="chart"> </div>
15
16 <script type="module">
17   import {renderGroupedBarChart} from './grouped-barchart.ts'
18   renderGroupedBarChart('#chart')
19 </script>
20 </body>
21 </html>
```

**Listing B.1:** HTML markup for a responsive Grouped Bar Chart.

```

1 #chart {
2   width: 100%;
3   height: 75vh;
4   min-height: 25rem;
5   container-type: inline-size;
6   .window-rv {
7     /* categorical color encoding can be changed as desired */
8     [data-style=categorical-0] { --color-categorical-0: #4f8c9d; }
9     [data-style=categorical-1] { --color-categorical-1: #e23209; }
10    [data-style=categorical-2] { --color-categorical-2: #539322; }
11  }
12  .legend {
13    margin-top: 1.5rem;
14    margin-right: 0.3rem;
15    .title { margin-left: 0.6rem; }
16    .items {
17      .legend-item {
18        width: 5rem;
19        height: 1.5rem;
20        gap: 0.5rem;
21        justify-content: flex-end;
22        &.highlight { font-size: calc(1.4 * var(--font-size-legend-label)); }
23      }
24      align-items: flex-end;
25    }
26  }
27
28  /* for narrow screen widths */
29  @container (width < 45rem) {
30    .window-rv {
31      /* increase size of right padding container so bar label fits */
32      --chart-padding-right: calc(3rem - clamp(0rem, 8vw, 3rem));
33    }
34    .chart {
35      /* move legend below chart */
36      grid-template: auto 1fr auto / 1fr;
37      grid-template-areas: 'header' 'padding-wrapper' 'legend';
38    }
39    .legend {
40      margin-top: 0.5rem;
41      margin-right: 0;
42    }
43    .legend .title { display: none; }
44    .legend .items {
45      width: 100%;
46      flex-direction: row;
47      justify-content: space-evenly;
48    }
49    /* thin out y-axis ticks to avoid intersecting tick labels
50     (since y-axis is flipped for narrow widths) */
51    .axis-y .tick:nth-of-type(2n) { display: none; }
52  }
53 }

```

Listing B.2: CSS for a responsive grouped bar chart.

```

1 import {BarChart, BarChartUserArgs, layouterCompute}
2   from './libs/respvis/respvis.js';
3 import {compensations, sites, years} from './data/compensation-employees.js';
4 import * as d3 from './libs/d3-7.6.0/d3.js'
5
6 /**
7  * Render Function of a Grouped Bar Chart
8  * @param selector The selector for querying the wrapper of the Grouped Bar Chart
9  */
10 export function renderGroupedBarChart(selector: string) {
11   // horizontal breakpoints for x-axis (y-axis when flipped).
12   // must be specified in ascending order and have same unit
13   const axisBreakPointsWidth = {
14     values: [10, 30, 50],
15     unit: 'rem'
16   } as const
17
18   // using BarChartUserArgs provides type support
19   const barChartArgs: BarChartUserArgs = {
20     breakpoints: {
21       width: {
22         values: [20, 45, 50], // chart breakpoints
23         unit: 'rem'
24       }
25     },
26
27     series: {
28       type: 'grouped',
29       x: { values: sites }, // array of strings for x values
30       y: { values: compensations }, // array of numbers for y values
31       categories: {
32         title: 'Years', // used to label the category
33         values: years // array of strings for categories
34       },
35
36       // callback function returning tooltip when bar hovered
37       markerTooltipGenerator: ((e, d) => {
38         return `Site: ${d.xValue}<br/>
39           Total Remuneration: ${d3.format(',')(d.yValue)}<br/>
40           Year: ${d.categoryFormatted ?? ''}<br/>`;
41       }),
42
43       // at which layout widths bar chart is flipped
44       flipped: {
45         dependentOn: 'width',
46         mapping: {0: true, 2: false} // flipped when < 45 rem
47       },
48
49       // maximum scale factors for zooming in and out
50       zoom: {
51         in: 20,
52         out: 1
53       },
54
55       // content and position of bar labels
56       labels: {
57         values: compensations.map(comp => d3.format('.2s')(comp)),
58         offset: 6, positionStrategy: 'dynamic'
59       }
60     },
61   },

```

**Listing B.3:** TypeScript code for a responsive Grouped Bar Chart.

```

61
62     legend: {
63         title: 'Year'
64     },
65
66     x: {
67         title: 'Country',
68         breakpoints: {
69             width: axisBreakPointsWidth,
70         },
71     },
72
73     y: {
74         title: 'Total Remuneration',
75         subTitle: '[EU]',
76         breakpoints: {
77             width: axisBreakPointsWidth
78         },
79
80         // orientation of y-axis tick labels, if y-axis is flipped
81         // rotation is interpolated between 0° (wide) and 90° (narrow)
82         tickOrientationFlipped: {
83             scope: 'self',
84             dependentOn: 'width',
85             breakpointValues: {0: 90, 2: 0}
86         },
87
88         // tick formatting for different chart widths
89         configureAxis: {
90             dependentOn: 'width',
91             scope: 'chart',
92             mapping: {0: (axis) => axis.tickFormat(d3.format('.2s')),
93                 3: (axis) => axis.tickFormat()}
94         },
95     }
96 }
97
98 // append empty div for chart window and create new chart instance
99 const chartWindow = d3.select(selector).append('div')
100 const renderer = new BarChart(chartWindow, barChartArgs)
101 renderer.buildChart()
102
103 const itemHover = () => {
104     layouterCompute(renderer.layouterS) // legend items grow on hover
105     renderer.render() // recompute layout
106 }
107
108 chartWindow.selectAll('.legend-item')
109     .on('mouseenter', itemHover)
110     .on('mouseleave', itemHover)
111 }

```

**Listing B.3** (cont.): TypeScript code for a responsive Grouped Bar Chart.

```

1 <!DOCTYPE html>
2 <html xmlns="http://www.w3.org/1999/xhtml" lang="en" xml:lang="en">
3 <head>
4   <title>RespVis - Scatterplot</title>
5   <meta name="viewport" content="width=device-width, initial-scale=1.0"/>
6   <meta charset="UTF-8"/>
7   <link rel="stylesheet" href="./libs/respvis/respvis.css" type="text/css"/>
8   <link rel="stylesheet" href="scatter-plot.css" type="text/css"/>
9 </head>
10 <body>
11 <h1>Scatterplot</h1>
12 <noscript>Please enable JavaScript!</noscript>
13 <div id="chart"> </div>
14
15 <script type="module">
16   import {createScatterplot} from './scatterplot.ts';
17   createScatterplot('#chart')
18 </script>
19 </body>
20 </html>

```

**Listing B.4:** HTML markup for a responsive Scatter Plot.

### B.3 Scatter Plot

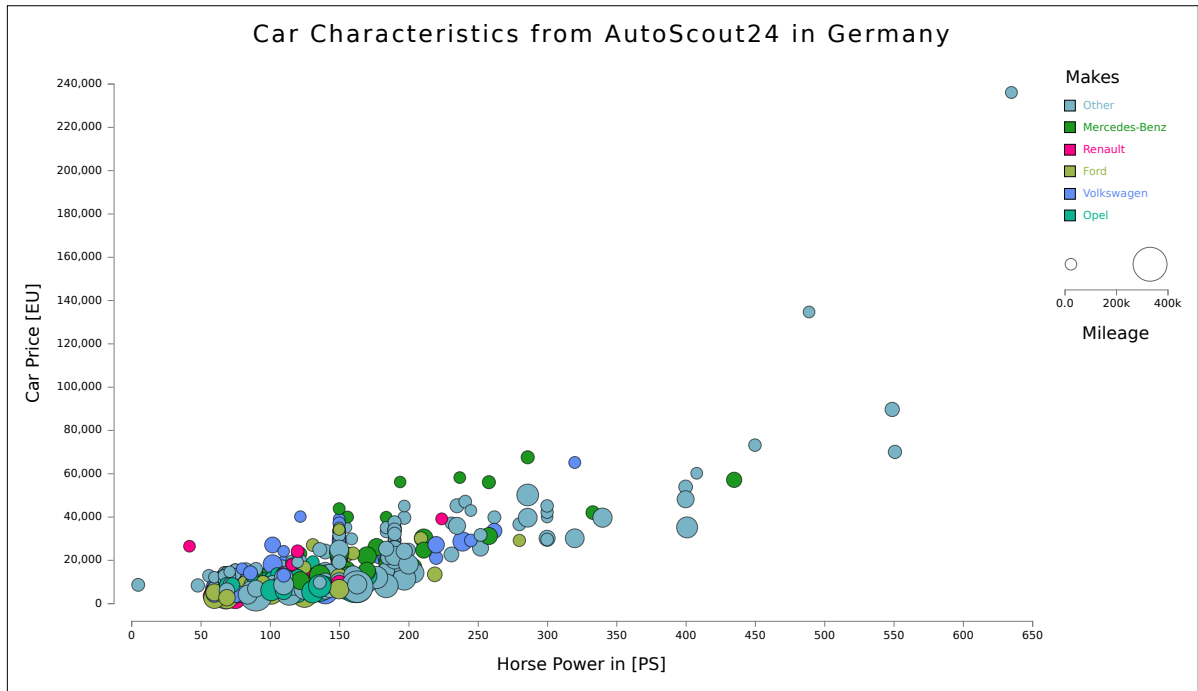
Scatter plots visualize the relation between two numerical data dimensions by plotting points in a cartesian coordinate system. They may also support the categorization of data records by mapping distinct color attributes to points. Another option of using colors in scatter plots is to introduce an additional dimension using sequential color encoding. Also, the size of the points can be encoded to add another dimension. This variation of a scatter plot is called a bubble chart [Kirk 2019, pages 166–167].

An implementation of a responsive Scatter Plot can be seen in Figure B.2. It is implemented with the HTML markup shown in Listing B.4, the CSS shown in Listing B.5, and the TypeScript code shown in Listing B.6. The presented Scatter Plot applies the following visual responsive patterns:

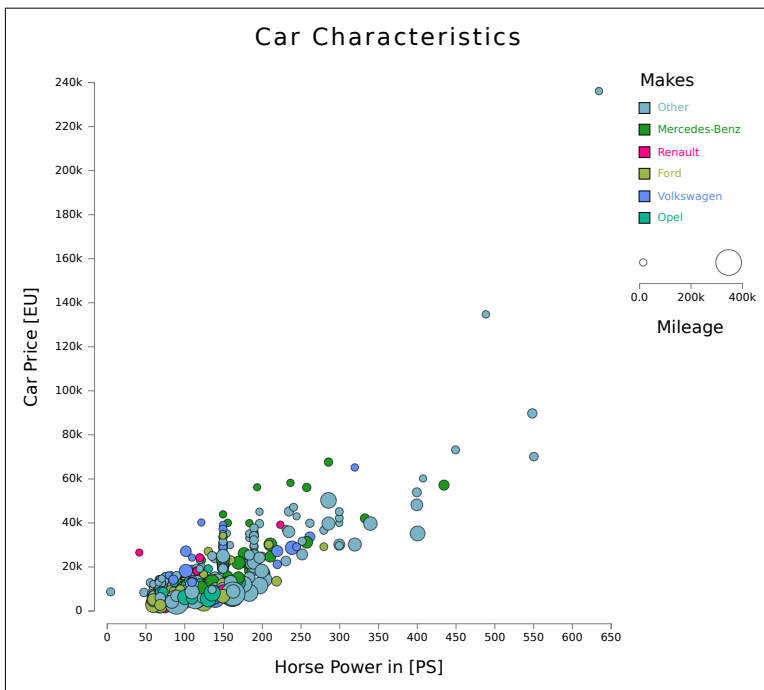
- *V2 Using Tooltips Instead of Element Labels:* Since there are many point elements in the Scatter Plot, element labels were avoided as they would pollute the visualization with an overwhelming number of labels. Instead, a Data Series Tooltip (Listing B.6, lines 62–67) is configured to be displayed when hovering over elements.
- *V4 Shortening Labels and Titles:* When transitioning from a wide to a narrow view, the title of the Scatter Plot (Listing B.6, lines 22–26), the title of the X-Axis (Listing B.6, lines 77–81), the labels of the X-Axis (Listing B.6, lines 92–104), and the labels of the Y-Axis (Listing B.6, lines 111–118) are shortened to avoid cluttered or cropped text.
- *V5 Scaling Down Visual Elements:* When transitioning from a wide to a narrow view, the drawing area and the Axes are scaled down automatically by RespVis during re-rendering. The size of the point elements is configured to smoothly transition between defined breakpoints (Listing B.6, lines 51–58).
- *V6 Hiding Elements and Labels:* When transitioning from a wide to a narrow view, both Axes thin out their axis ticks to avoid cluttered text.

In addition to the listed visual patterns, the presented Scatter Plot also implements all discussed interactive patterns. Figure B.2 (c) shows the mobile view of a Scatter Plot, which is zoomed into.

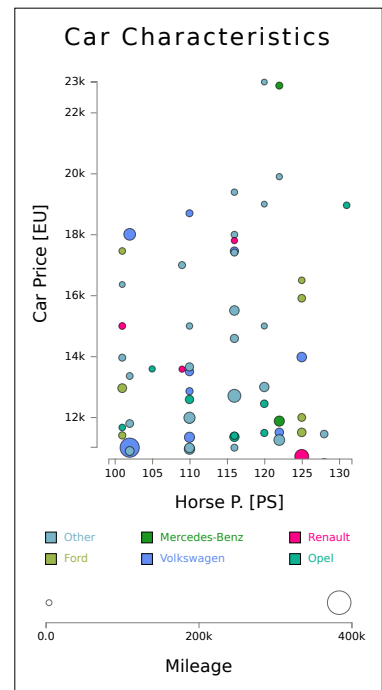




(a) Wide.



(b) Medium.



(c) Narrow and zoomed.

**Figure B.2:** Wide, medium, and narrow versions of a Scatter Plot. The narrow version has been zoomed in. [Images created with RespVis [Egger and Oberrauner 2024a] by the author of this thesis.]

```
1 #chart {
2   width: 100%;
3   height: 75vh;
4   min-height: 25rem;
5
6   .window-rv {
7     container-type: inline-size;
8
9     /* for narrow screen widths */
10    @container (width < 40rem) {
11      .chart {
12        /* move legend below chart */
13        grid-template: auto 1fr auto / 1fr;
14        grid-template-areas: 'header' 'padding-wrapper' 'legend';
15        --chart-padding-right: 1rem;
16      }
17
18      .legend {
19        margin-top: 1rem;
20        justify-content: center;
21        & > .title { display: none; }
22
23        .legend__categories {
24          width: 100%;
25        }
26        .items {
27          display: grid;
28          width: 100%;
29          justify-content: space-between;
30          grid-template-columns: auto auto auto;
31        }
32      }
33      /* thin out y-axis ticks to avoid intersecting tick labels
34       (since y-axis is flipped for narrow widths) */
35      .axis-y .tick {
36        display: none;
37
38        &:nth-of-type(2n + 1), &:first-of-type, &:last-of-type {
39          display: block;
40        }
41      }
42    }
43  }
44 }
```

**Listing B.5:** CSS for a responsive Scatter Plot.

```
1 import {formatWithDecimalZero, Point, ScatterPlot, ScatterPlotUserArgs}
2   from './libs/respvis/respvis.js';
3 import * as d3 from './libs/d3-7.6.0/d3.js'
4 import {getTopMakesData} from "./data/sold-cars-germany.js";
5
6 /**
7  * Render Function of a Scatter Plot
8  * @param selector The selector for querying the wrapper of the Scatter Plot
9  */
10 export function createScatterplot(selector: string) {
11   const {mileages, horsePower, prices, makes} = getTopMakesData(5)
12
13   // using ScatterPlotUserArgs provides type support
14   const data: ScatterPlotUserArgs = {
15     breakpoints: {
16       width: {
17         values: [40, 60, 90], // chart breakpoints
18         unit: 'rem'
19       }
20     },
21
22     title: {
23       dependentOn: 'width',
24       mapping: {0: 'Car Characteristics',
25                2: 'Car Characteristics from AutoScout24 in Germany'}
26     },
27
28     series: {
29       x: {
30         values: horsePower, // array of numbers for x values
31       },
32       y: {
33         values: prices, // array of numbers for y values
34       },
35       categories: {
36         title: 'Makes', // used to label the category
37         values: makes // array of strings for categories
38       },
39
40       radii: {
41         values: mileages, // array of numbers for radii
42         axis: { // radii axis in legend
43           title: 'Mileage',
44           horizontalLayout: 'bottom',
45           configureAxis: (axis => {
46             axis.ticks(2)
47             axis.tickFormat(d3.format('.2s'))
48           })
49       },
```

**Listing B.6:** TypeScript code for a responsive Scatter Plot.

```

50
51     extrema: { // radii sizes are interpolated
52         dependentOn: 'width',
53         breakpointValues: {
54             0: {minimum: 3, maximum: 12},
55             1: {minimum: 5, maximum: 15},
56             2: {minimum: 7, maximum: 20},
57         },
58     },
59 },
60
61 // callback function returning tooltip when point hovered
62 markerTooltipGenerator: ((e, d: Point) => {
63     return 'Car Price: ${d.yValue}€<br/>
64           Horse Power: ${d.xValue}PS<br/>
65           Make: ${d.categoryFormatted ?? ''}<br/>
66           Mileage: ${d.radiusValue}km<br/>'
67 }),
68
69 // maximum scale factors for zooming in and out
70 zoom: {
71     in: 20,
72     out: 1
73 },
74 },
75
76 x: {
77     title: {
78         dependentOn: 'width',
79         scope: 'self',
80         mapping: {0: 'HP in [PS]', 1: 'Horse P. [PS]', 2: 'Horse Power in [PS]'}
81     },
82
83     breakpoints: {
84         width: {
85             values: [10, 30, 50],
86             unit: 'rem'
87         }
88     },
89
90 // tick formatting for different axis widths
91 // number of ticks is reduced via D3 ticks function
92 configureAxis: {
93     dependentOn: 'width',
94     scope: 'self',
95     mapping: {
96         0: (axis) => {
97             axis.tickFormat(d3.format('.3d'))
98             axis.ticks(7)
99         },
100        2: (axis) => {
101            axis.tickFormat(d3.format('.3d'))
102        }
103    }
104 }
105 },

```

**Listing B.6** (cont.): TypeScript code for a responsive Scatter Plot.

```
106
107   y: {
108     title: 'Car Price [EU]',
109
110     // tick formatting for different chart widths
111     configureAxis: {
112       dependentOn: 'width',
113       scope: 'chart',
114       mapping: {
115         0: (axis) => axis.tickFormat(formatWithDecimalZero(d3.format('.2s'))),
116         2: (axis) => axis.tickFormat(formatWithDecimalZero(d3.format(', ')))
117       }
118     }
119   },
120
121   legend: {
122     title: {
123       dependentOn: 'width',
124       scope: 'chart',
125       mapping: {0: '', 1: 'Makes'}
126     },
127   }
128 }
129
130 // append empty div for chart window and create new chart instance
131 const chartWindow = d3.select(selector).append('div')
132 const renderer = new ScatterPlot(chartWindow, data)
133 renderer.buildChart()
134 }
```

**Listing B.6** (cont.): TypeScript code for a responsive Scatter Plot.

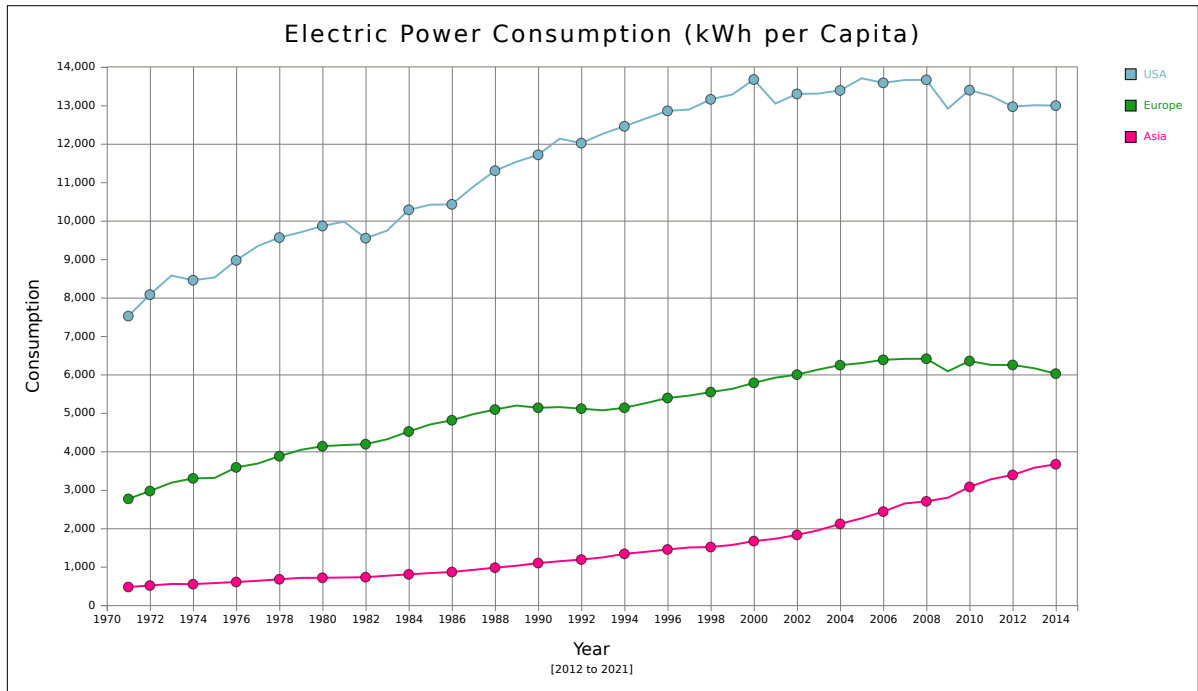
## B.4 Line Chart

Line charts are typically used to visualize the trend of a numerical variable (y-axis) over a temporal variable (x-axis) in a cartesian coordinate system. Another option is to use categorical data for the x-axis which results in the chart communicating a similar message like a bar chart (comparison of categorized items by a numerical dimension). Data is visualized in line charts by plotting related data points as markers and connecting them with a polyline, resulting in a related sequence of values. Multiple independent line sequences are typically distinguished via categorical color encoding [Kirk 2019, page 171].

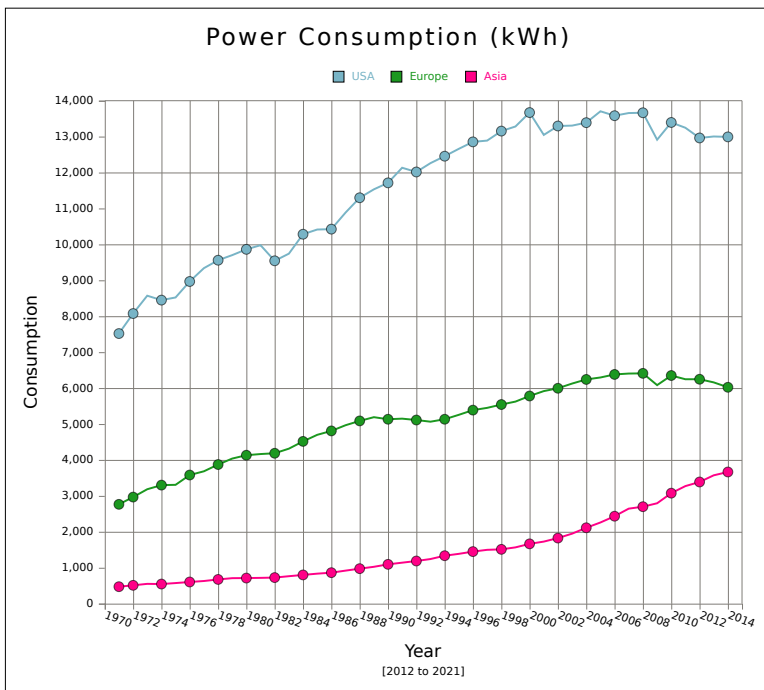
An implementation of a responsive Multi-Series Line Chart can be seen in Figure B.3. The presented Line Chart is implemented by providing the HTML markup shown in Listing B.7, the CSS shown in Listing B.8, and the TypeScript code shown in Listing B.9. The presented Line Chart applies the following visual responsive patterns:

- *V2 Using Tooltips Instead of Element Labels*: Since there are many point elements in the Line Chart, element labels were avoided as they would intersect with each other. The simpler solution was to configure a Data Series Tooltip to be displayed when hovering over elements (Listing B.9, lines 39–41).
- *V3 Rotating Axis Tick Labels*: When transitioning from a wide to a narrow view, the tick labels of the X-Axis are rotated (Listing B.9, lines 69–73).
- *V4 Shortening Labels and Titles*: When transitioning from a wide to a narrow view, the title of the Line Chart is shortened to fit into the narrower available space (Listing B.9, lines 50–54).
- *V5 Scaling Down Visual Elements*: When transitioning from a wide to a narrow view, the drawing area, and the Axes are scaled down automatically by RespVis during re-rendering.
- *V6 Hiding Elements and Labels*: When transitioning from a wide to a narrow view, the ticks of the X-Axis are thinned out to avoid intersecting or cropped labels (Listing B.9, lines 76–79). Also, point elements representing data points are thinned out to avoid overplotting (Listing B.8, line 16, lines 22–24, lines 50–52).

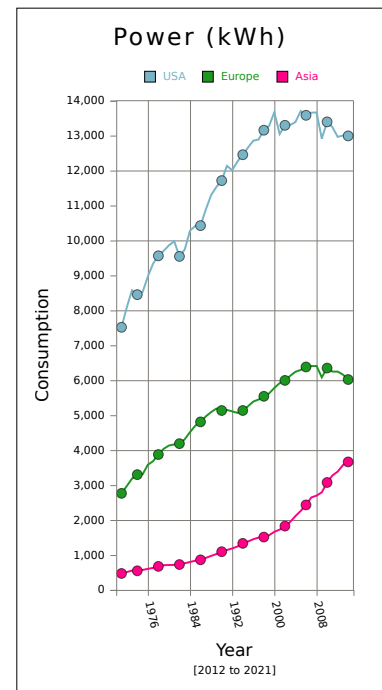
In addition to the listed visual patterns, the presented Multi-Series Line Chart also implements all discussed interactive patterns.



(a) Wide.



(b) Medium.



(c) Narrow.

**Figure B.3:** Wide, medium, and narrow versions of a Multi-Series Line Chart. [Images created with RespVis [Egger and Oberrauner 2024a] by the author of this thesis.]

```
1 <!DOCTYPE html>
2 <html xmlns="http://www.w3.org/1999/xhtml" lang="en" xml:lang="en">
3 <head>
4   <title>RespVis - Multi-Line Chart</title>
5   <meta name="viewport" content="width=device-width, initial-scale=1.0"/>
6   <meta charset="UTF-8"/>
7   <link rel="stylesheet" href="./libs/respvis/respvis.css"/>
8   <link rel="stylesheet" href="multi-line-chart.css"/>
9 </head>
10
11 <body>
12 <h1>Multi-Line Chart</h1>
13 <noscript>Please enable JavaScript!</noscript>
14 <div id="chart"> </div>
15
16 <script type="module">
17   import {renderMultiLineChart} from "./multi-line-chart.ts"
18   renderMultiLineChart('#chart')
19 </script>
20 </body>
21 </html>
```

**Listing B.7:** HTML markup for a responsive Multi-Series Line Chart.



```
1 #chart {
2   width: 100%;
3   height: 75vh;
4   min-height: 25rem;
5   container-type: inline-size;
6   .window-rv {
7     --chart-padding-right: 1rem;
8   }
9
10  .legend {
11    margin-left: 2rem;
12    .items { gap: 1rem; }
13  }
14
15  /* hide x-axis ticks by default */
16  .point:not(.inspect-nearest), .axis-x .tick { display: none; }
17
18  /* for average and wide screen widths */
19  @container (width >= 30rem) {
20    .point-category {
21      /* show first, last, and each 2nd data point */
22      .point:first-of-type, .point:last-of-type, .point:nth-of-type(2n) {
23        display: block;
24      }
25    }
26    /* show all x-axis ticks */
27    .axis-x .tick { display: block; }
28  }
29
30  /* for narrow to average screen widths */
31  @container (width <= 50rem) {
32    .chart {
33      /* move legend on top of chart */
34      grid-template: auto auto 1fr / 1fr;
35      grid-template-areas: 'header' 'legend' 'padding-wrapper';
36    }
37
38    .legend {
39      width: 100%;
40      margin-bottom: 1rem;
41    }
42
43    .legend .items { flex-direction: row; }
44  }
45
46  /* for narrow screen widths */
47  @container (width < 30rem) {
48    .point-category {
49      /* show first, last, and each 4th data point */
50      .point:first-of-type, .point:last-of-type, .point:nth-of-type(4n) {
51        display: block;
52      }
53    }
54    /* show each 4th x-axis tick */
55    .axis-x .tick:nth-of-type(4n) { display: block; }
56  }
57 }
```

Listing B.8: CSS for a responsive Multi-Series Line Chart.

```

1 import {LineChart, LineChartUserArgs, select, selectAll, timeFormat, timeYear}
2   from './libs/respvis/respvis.js';
3 import {mapPowerConsumptionData} from './data/electric-power-consumption.js'
4
5 /**
6  * Render Function of a Multi-Series Line Chart
7  * @param selector The selector for querying the wrapper of the Multi-Series
8  * Line Chart
9  */
10 export const renderMultiLineChart = (selector: string) => {
11   const {yUSA, yEurope, yAsia, yearsJSDateFormat} = mapPowerConsumptionData()
12
13   // using LineChartUserArgs provides type support
14   const data: LineChartUserArgs = {
15     breakpoints: {
16       width: {
17         values: [25, 30, 50],
18         unit: 'rem'
19       }
20     },
21
22     series: {
23       x: { // array of Date objects for x values
24         values: [...yearsJSDateFormat, ...yearsJSDateFormat, ...yearsJSDateFormat]
25       },
26       y: { // array of numbers for y values
27         values: [...yUSA, ...yEurope, ...yAsia]
28       },
29       categories: {
30         title: 'Continents', // used to label the category
31         values: [ // array of strings for categories
32           ...yUSA.map(() => 'USA'),
33           ...yEurope.map(() => 'Europe'),
34           ...yAsia.map(() => 'Asia')
35         ],
36       },
37
38       // callback function returning tooltip when data points hovered
39       markerTooltipGenerator: (_, point) =>
40         `Year: ${point.xValue as Date}.getFullYear()}
41         <br/>Pow. Consumption: ${point.yValue}kWh`,
42
43       // maximum scale factors for zooming in and out
44       zoom: {
45         in: 20,
46         out: 1
47       }
48     },
49
50     title: {
51       dependentOn: 'width',
52       mapping: {0: 'Power (kWh)', 1: 'Power Consumption (kWh)',
53         3: 'Electric Power Consumption (kWh per Capita)'}
54     },

```

**Listing B.9:** TypeScript code for a responsive Multi-Series Line Chart.

```
55
56   x: {
57     title: 'Year',
58     subTitle: '[2012 to 2021]',
59
60     breakpoints: {
61       width: {
62         values: [10, 30, 50],
63         unit: 'rem'
64       }
65     },
66
67     // orientation of x-axis tick labels
68     // rotation is interpolated between 0° (wide) and 90° (narrow)
69     tickOrientation: {
70       dependentOn: 'width',
71       scope: 'self',
72       breakpointValues: {0: 90, 2: 0},
73     },
74
75     // tick formatting for temporal axis
76     configureAxis: (axis) => {
77       axis.ticks(timeYear.every(2))
78       axis.tickFormat(timeFormat('%Y'))
79     },
80     gridLineFactor: 1           // vertical grid line each x-axis tick
81   },
82
83   y: {
84     title: 'Consumption',
85     breakpoints: {
86       width: {
87         values: [10, 30, 50],
88         unit: 'rem'
89       }
90     },
91
92     // orientation of y-axis tick labels, if y-axis is flipped
93     // rotation is interpolated between 0° (wide) and 90° (narrow)
94     tickOrientationFlipped: {
95       dependentOn: 'width',
96       scope: 'self',
97       breakpointValues: {0: 90, 2: 0},
98     },
99     gridLineFactor: 2           // horizontal grid line each 2nd y-axis tick
100   },
```

**Listing B.9** (cont.): TypeScript code for a responsive Multi-Series Line Chart.

```
101
102 // assign classnames to currently inspected data points (inspection tool)
103 tooltip: {
104   onInspectMove: (info) => {
105     const pointS = selectAll('.point.element')
106     const nearestPointS = pointS.filter((d) =>
107       d.xValue === info.horizontalNearestRealValue)
108     nearestPointS.classed('inspect-nearest', true)
109     const otherPointS = pointS.filter((d) =>
110       d.xValue !== info.horizontalNearestRealValue)
111     otherPointS.classed('inspect-nearest', false)
112   }
113 }
114 }
115
116 // append empty div for chart window and create new chart instance
117 const chartWindow = select(selector).append('div')
118 const renderer = new LineChart(chartWindow, data)
119 renderer.buildChart()
120 }
```

**Listing B.9** (cont.): TypeScript code for a responsive Multi-Series Line Chart.

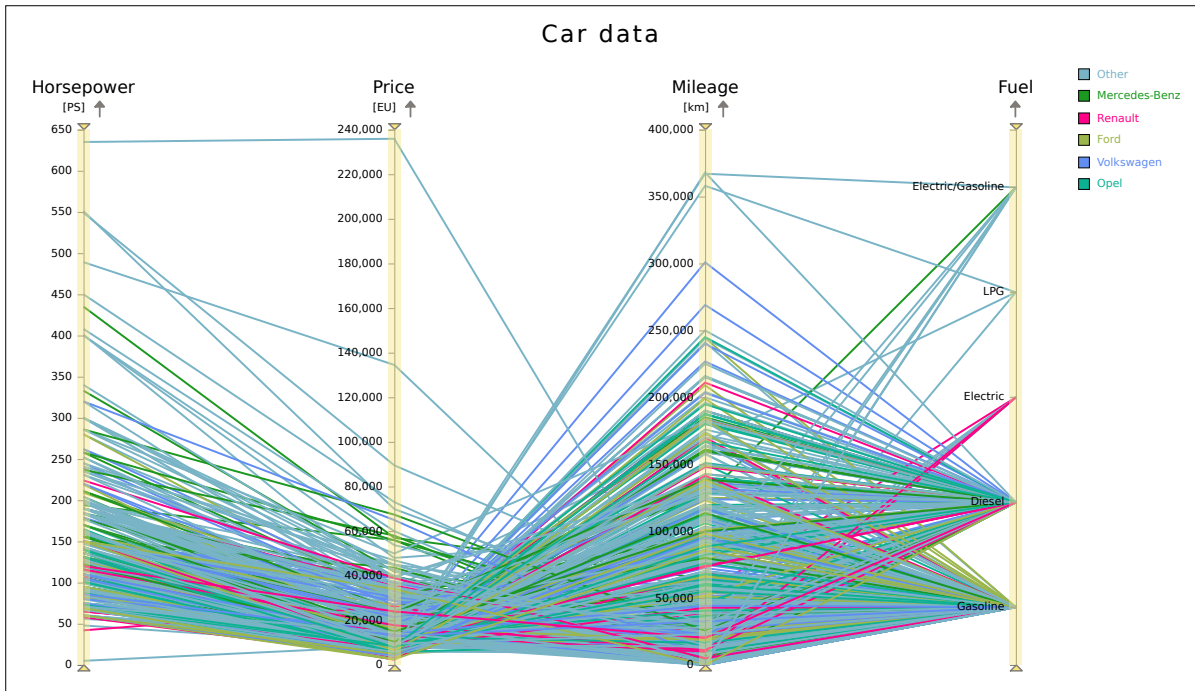
## B.5 Parallel Coordinates Chart

Parallel coordinate charts are useful tools for visualizing multivariate data [Ribecca 2024; Inselberg 2009]. Each dimension of a dataset is represented by a dedicated axis, scaled according to the domain values of the variable. The axes are aligned in parallel, either all horizontally or all vertically, such that every two neighboring axes can be connected by straight line segments. A data record is represented by a polyline, i.e. multiple, connected, straight line segments, touching each axis once at the corresponding data value.

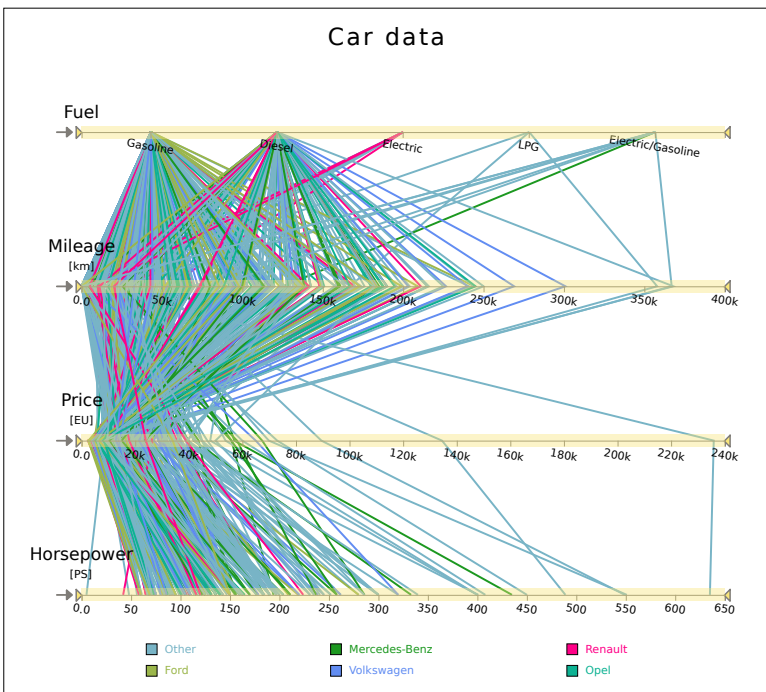
An implementation of a Parallel Coordinates Chart can be seen in Figure B.4. The presented Parallel Coordinates Chart is implemented by providing the HTML markup shown in Listing B.10, the CSS shown in Listing B.11, and the TypeScript code shown in Listing B.12. The presented Parallel Coordinates Chart applies the following visual responsive patterns:

- *V3 Rotating Axis Tick Labels*: When transitioning from a wide to a narrow view, the chart is flipped, meaning that all Axes are displayed horizontally rather than vertically. Once the chart is flipped, the tick labels of all Axes are rotated continuously to avoid intersecting labels of the horizontally positioned Axes (Listing B.12, lines 29–32, line 60, line 78, line 96 and line 107).
- *V4 Shortening Labels and Titles*: When transitioning from a wide to a narrow view, the labels of the Price Axis and Mileage Axis are shortened to avoid intersecting axis labels (Listing B.12, lines 16–25, 77, and 95).
- *V5 Scaling Down Visual Elements*: When transitioning from a wide to a narrow view, the drawing area and the Axes are scaled down automatically by RespVis during re-rendering.
- *V6 Hiding Elements and Labels*: When transitioning from a wide to a narrow view, the ticks of the Horsepower Axis and Price Axis are thinned out to avoid cluttered text (Listing B.11, lines 57-62).
- *V7 Rotating Chart 90°*: When transitioning from a wide to a narrow view, the chart is transposed, since the chart provides more space vertically than horizontally (Listing B.11, lines 118–121), (Listing B.12, lines 24–58). This pattern is commonly applied for Parallel Coordinates Charts, since it helps to add more space between the chart's Axes and, therefore, prevents axis intersections. Having enough space between Axes is an important matter for responsive Parallel Coordinates Charts, since Axes can be interacted with pointer devices and intersecting Axes would be a major issue in this context.

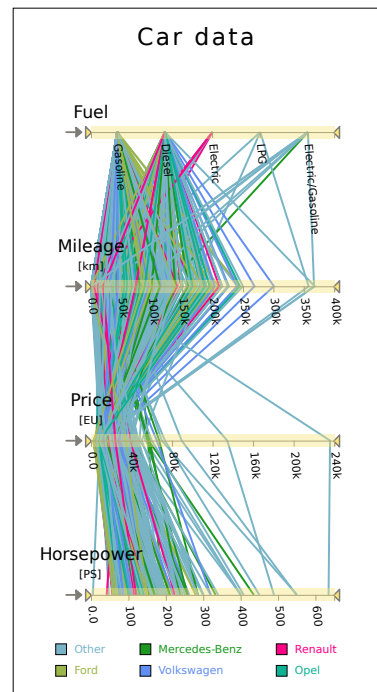
In addition to the listed visual patterns, the presented Parallel Coordinates Chart also implements all discussed interactive patterns.



(a) Wide.



(b) Medium.



(c) Narrow.

**Figure B.4:** Wide, medium, and narrow versions of a Parallel Coordinates Chart. [Images created with RespVis [Egger and Oberrauner 2024a] by the author of this thesis.]

```
1 <!DOCTYPE html>
2 <html xmlns="http://www.w3.org/1999/xhtml" lang="en" xml:lang="en">
3 <head>
4   <title>RespVis - Parallel Coordinates Chart</title>
5   <meta name="viewport" content="width=device-width, initial-scale=1.0"/>
6   <meta charset="UTF-8"/>
7   <link rel="stylesheet" href="./libs/respvis/respvis.css"/>
8   <link rel="stylesheet" href="parcoord-chart.css"/>
9 </head>
10 <body>
11 <h1>Parallel Coordinates Chart</h1>
12 <noscript>Please enable JavaScript!</noscript>
13 <div id="chart"> </div>
14
15 <script type="module">
16   import {renderParcoord} from "./parcoord-chart.ts";
17   renderParcoord('#chart')
18 </script>
19 </body>
20 </html>
```

**Listing B.10:** HTML markup for a responsive Parallel Coordinates Chart.

```

1 #chart {
2   width: 100%;
3   height: 75vh;
4   min-height: 25rem;
5   container-type: inline-size;
6
7   /* increase transition time of record lines */
8   .window-rv { --transition-time-ms: 750;}
9
10  .chart-parcoord {
11    /* increase size of padding containers for axis titles to fit */
12    --chart-padding-left: 4rem;
13    --chart-padding-right: 4rem;
14    --chart-padding-top: 4rem;
15    --chart-padding-bottom: 0.8rem;
16
17    /* prevent flicker on highlighting with certain category color */
18    path.line.animated.highlight[data-key~="s-0-c-0"] {
19      filter: unset;
20    }
21  }
22
23  /* for average to narrow screen widths */
24  @container (width < 50rem) {
25    .chart-parcoord {
26      /* adapt size of padding containers for flipped chart */
27      --chart-padding-right: 2rem;
28      --chart-padding-bottom: 3rem;
29
30      /* move legend below chart */
31      grid-template: auto 1fr auto / 1fr;
32      grid-template-areas: 'header' 'padding-wrapper' 'legend';
33    }
34
35    .legend {
36      width: 100%;
37      flex-direction: row;
38      justify-content: center;
39
40      .title { display: none; }
41
42      .legend__categories { width: 100%; }
43
44      .items {
45        display: grid;
46        width: 100%;
47        justify-content: space-evenly;
48        grid-template-columns: auto auto auto;
49      }
50    }
51  }

```

**Listing B.11:** CSS for a responsive Parallel Coordinates Chart.



```
52
53 /* for narrow screen widths */
54 @container (width < 40rem) {
55   /* thin out axis ticks of axes a-0 ('Horsepower') and a-1 ('Price')
56   (since axes are flipped for narrow widths) */
57   .axis.axis-sequence {
58     &[data-key="a-0"] .tick, &[data-key="a-1"] .tick {
59       &:nth-of-type(2n) {
60         display: none;
61       }
62     }
63   }
64 }
65 }
```

**Listing B.11** (cont.): CSS for a responsive Parallel Coordinates Chart.

```

1 import {formatWithDecimalZero, ParcoordChart, ParcoordChartUserArgs}
2   from './libs/respvis/respvis.js';
3 import * as d3 from './libs/d3-7.6.0/d3.js'
4 import {getTopMakesData} from "./data/sold-cars-germany.js";
5
6 /**
7  * Render Function of a Parallel Coordinates Chart
8  * @param selector The selector for querying the wrapper of the
9  * Parallel Coordinates Chart
10 */
11 export function renderParcoord(selector: string) {
12   const sampleSize = 500
13   const {horsePower, prices, mileages, makes, fuel} = getTopMakesData(5)
14
15   // tick formatting for different chart widths of 'Mileage' and 'Price' axes
16   const sharedAxisConfig = {
17     dependentOn: 'width',
18     scope: 'chart',
19     mapping: {
20       0: (axis: d3.Axis<d3.AxisDomain>) =>
21         axis.tickFormat(formatWithDecimalZero(d3.format('.2s'))),
22       3: (() => {
23         })
24     }
25   } as const
26
27   // tick label orientation of all axes, if axes are flipped
28   // rotation is interpolated between 0° (wide) and 90° (narrow)
29   const sharedTickOrientationFlipped = {
30     dependentOn: 'width',
31     breakpointValues: {0: 90, 2: 0}
32   } as const
33
34   // using ParcoordChartUserArgs provides type support
35   const data: ParcoordChartUserArgs = {
36     title: 'Car data',
37     breakpoints: {
38       width: {
39         values: [20, 30, 50],
40         unit: 'rem'
41       }
42     },

```

**Listing B.12:** TypeScript code for a responsive Parallel Coordinates Chart.

```
43
44   series: {
45     dimensions: [
46       // 'Horsepower' axis
47       {
48         // array of numbers for 'Horsepower' axis
49         scaledValues: {values: horsepower.slice(0, sampleSize)},
50
51         // maximum scale factors for zooming in and out
52         zoom: {
53           in: 10,
54           out: 1
55         },
56
57         axis: {
58           title: "Horsepower",
59           subTitle: "[PS]",
60           tickOrientationFlipped: sharedTickOrientationFlipped
61         }
62       },
63
64       // 'Price' axis
65       {
66         // array of numbers for 'Price' axis
67         scaledValues: {values: prices.slice(0, sampleSize)},
68
69         zoom: {
70           in: 20,
71           out: 1
72         },
73
74         axis: {
75           title: "Price",
76           subTitle: "[EU]",
77           configureAxis: sharedAxisConfig,
78           tickOrientationFlipped: sharedTickOrientationFlipped
79         }
80       },
81
82       // 'Mileage' axis
83       {
84         // array of numbers for 'Mileage' axis
85         scaledValues: {values: mileages.slice(0, sampleSize)},
86
87         zoom: {
88           in: 20,
89           out: 1
90         },
91
92         axis: {
93           title: "Mileage",
94           subTitle: "[km]",
95           configureAxis: sharedAxisConfig,
96           tickOrientationFlipped: sharedTickOrientationFlipped
97         }
98       },
99     ],
100   },
```

**Listing B.12** (cont.): TypeScript code for a responsive Parallel Coordinates Chart.

```
99
100     // 'Fuel' axis
101     {
102         // array of strings for 'Fuel' axis
103         scaledValues: {values: fuel.slice(0, sampleSize)},
104
105         axis: {
106             title: "Fuel",
107             tickOrientationFlipped: sharedTickOrientationFlipped
108         }
109     },
110 ],
111
112 categories: {
113     title: 'Makes', // used to label the category
114     values: makes.slice(0, sampleSize) // array of strings for categories
115 },
116
117 // at which layout widths parallel coordinates chart is flipped
118 flipped: {
119     mapping: {0: true, 3: false},
120     dependentOn: 'width'
121 }
122 },
123 }
124
125 // append empty div for chart window and create new chart instance
126 const chartWindow = d3.select(selector).append('div')
127 const renderer = new ParcoordChart(chartWindow, data)
128 renderer.buildChart()
129 }
```

**Listing B.12** (cont.): TypeScript code for a responsive Parallel Coordinates Chart.

## Appendix C

# Chart Developer Guide

This guide explains how to create entirely new charts in RespVis v3. They can either be derived from RespVis' empty base chart, or by customizing the standard chart types provided by RespVis. This guide is addressed to *chart developers*, who are expected to have a solid understanding of the fundamental web technologies JavaScript, HTML, CSS, and SVG.

### C.1 Creating New Charts

The recommended way of creating new charts is to create new chart classes by extending the existing chart class provided by RespVis. The abstract chart class takes care of setting up complex re-rendering and layouting tasks, such that chart developers can focus on defining the interfaces, data validation, and render routines of their new charts.

Listings C.1 and C.2 show a practical example of a custom chart implementation called Axis Chart, which simply consists of Axis components. The underlying AxisChart class, shown in Listing C.1, is structured like the standard chart types of RespVis and expects a selection of a single HTML <div> element and a chart-specific input object. The validation function and the accepted types for chart input, validation function input, and validation function output are defined in Listing C.2

The abstract methods of the Chart class, `renderContent` and `revalidate`, must be implemented by concrete derived classes like AxisChart. The `renderContent` method must contain the render routines of the chart. In case of the AxisChart, this includes rendering all Axes passed as part of the chart arguments, updating the scales and orientation of these Axes, and rendering the Toolbar. The `revalidate` method must contain the logic for validating the chart-specific input object and binding the validated output to the window selection. The method is public and can be called after chart instantiation to replace the underlying data of the chart. An example of the discussed Axis Chart can be seen in Figure C.1.

```

1 import {
2   Axis, Chart, rectFromString, renderAxisLayout,
3   renderToolbar, validateWindow, Window, Selection
4 } from "respvis";
5 import {AxisChartUserArgs, AxisChartValid, validateAxisChart}
6 } from "./validate-axis-chart";
7
8 type WindowSelection = Selection<HTMLDivElement, Window & AxisChartValid>
9 type ChartSelection = Selection<SVGSVGElement, Window & AxisChartValid>
10
11 export class AxisChart extends Chart {
12   constructor(windowSelection: Selection<HTMLDivElement>,
13     args: AxisChartUserArgs ) {
14     super()
15     this._windowS = windowSelection as WindowSelection
16     this.revalidate(args)
17   }
18   _windowS: WindowSelection
19   get windowS(): WindowSelection { return this._windowS }
20   _chartS?: ChartSelection
21   get chartS(): ChartSelection {
22     return ((this._chartS && !this._chartS.empty()) ? this._chartS :
23       this.layouterS.selectAll('svg.chart')) as ChartSelection
24   }
25
26   protected renderContent() {
27     const {width, height} = rectFromString(this.drawAreaS.attr('bounds')
28       || '0, 0, 600, 400')
29     const {axes, series} = this.chartS.datum()
30     this.chartS.classed('chart-cartesian', true)
31
32     series.responsiveState.update()
33     const flipped = series.responsiveState.currentlyFlipped
34
35     axes.forEach((axis, index) => {
36       const orientation = flipped ? (axis.standardOrientation === 'horizontal' ?
37         'vertical' : 'horizontal')
38         : axis.standardOrientation
39
40       if (orientation === 'horizontal') axis.scaledValues.scale.range([0, width])
41       else axis.scaledValues.scale.range([height, 0])
42
43       this.paddingWrapperS.selectAll<SVGElement, Axis>(`.axis.axis-${index}`)
44         .data([axis])
45         .join('g')
46         .classed(`axis-${index}`, true)
47         .call(s => renderAxisLayout(s, orientation))
48     })
49     renderToolbar(this._windowS, {renderer: this, getAxes: () => axes,
50       getSeries: () => [series]})
51   }
52
53   revalidate(args: AxisChartUserArgs) {
54     const initialWindowData = validateWindow({...args, type: 'cartesian',
55       renderer: this})
56     const chartData = validateAxisChart({...args, renderer: this})
57     this.windowS.datum({...initialWindowData, ...chartData})
58   }
59 }

```

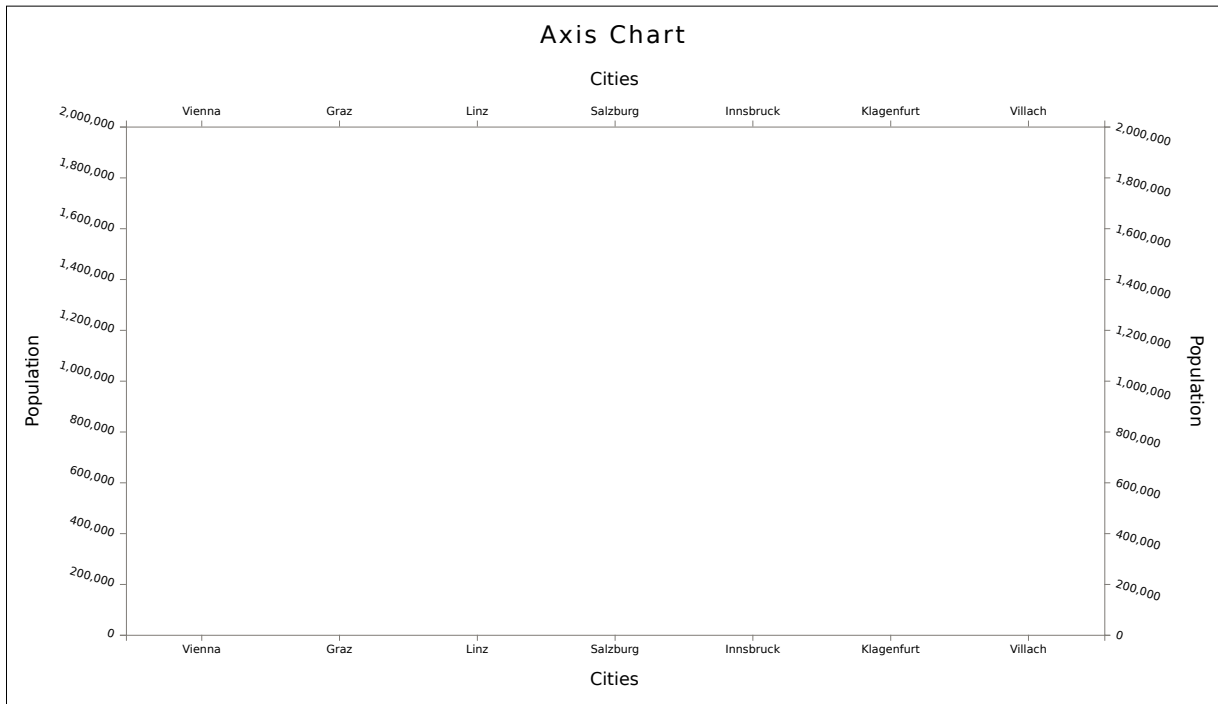
**Listing C.1:** TypeScript code for creating a new simple chart only containing Axes, called an Axis Chart.

```

1 import {
2   BaseAxis,
3   BaseAxisUserArgs,
4   ChartData,
5   ChartDataArgs,
6   ChartDataUserArgs,
7   Orientation,
8   Orientations,
9   ResponsiveValueUserArgs,
10  ScaledValuesSpatialDomain,
11  ScaledValuesSpatialUserArgs,
12  validateBaseAxis,
13  validateChart,
14  validateScaledValuesSpatial,
15  EmptySeries
16 } from "respvis";
17
18 export type AxisChartAxisUserArgs = BaseAxisUserArgs & {
19   vals: ScaledValuesSpatialUserArgs<ScaledValuesSpatialDomain>
20   standardOrientation?: Orientation
21 }
22
23 type AxisChartAxisValid = BaseAxis & {
24   standardOrientation: Orientation
25 }
26
27 export type AxisChartUserArgs = ChartDataUserArgs & {
28   axes: AxisChartAxisUserArgs[]
29   flipped?: ResponsiveValueUserArgs<boolean>
30 }
31
32 type AxisChartArgs = ChartDataArgs & AxisChartUserArgs
33
34 export type AxisChartValid = ChartData & {
35   axes: AxisChartAxisValid[],
36   series: EmptySeries
37 }
38
39 export function validateAxisChart(args: AxisChartArgs): AxisChartValid {
40   const series = new EmptySeries({key: 's-0', renderer: args.renderer,
41     flipped: args.flipped})
42   return {
43     ...validateChart(args),
44     series,
45     axes: args.axes.map((axis, index) => {
46       const scaledValues = validateScaledValuesSpatial(axis.vals, `a-${index}`)
47       const standardOrientation = (axis.standardOrientation &&
48         Orientations.includes(axis.standardOrientation)) ?
49         axis.standardOrientation : 'horizontal'
50       return {
51         ...validateBaseAxis({...axis, series, scaledValues,
52           renderer: args.renderer}), standardOrientation
53       }
54     })
55   }
56 }

```

**Listing C.2:** TypeScript code for the validation logic of an Axis Chart.



**Figure C.1:** An Axis Chart with four Axes. [Images created with RespVis [Egger and Oberrauner 2024a] by the author of this thesis.]

## C.2 Customizing Standard Chart Types

The RespVis v3 API includes classes for creating four types of charts out of the box: Bar Charts, Scatter Plots, Line Charts, and Parallel Coordinates Charts. These standard chart types can be customized by extending their classes with subclasses and overriding methods and properties. There are two reasons why a chart developer may want to customize one of the standard RespVis charts.

The first is that standard charts have a fixed render routine, meaning that the render routines of chart components like Legend, Toolbar, and cartesian components are always called. Furthermore, features like highlighting elements on hover, or whole Data Series being highlighted when a corresponding legend item is hovered, are always active by default. While there are other methods for turning some of these features off, a simple solution is to create a subclass of the corresponding standard chart class and customize its `renderContent` method. This gives chart developers full control of the content of a chart. Listing C.3 shows how the `BarChart` class can be customized to remove highlighted elements and Tooltips. Figure C.2 demonstrates the difference between a standard Bar Chart and the customized version when hovering over a bar element.

The second reason for customizing a standard chart may be the desire to create a new chart variant. The standard chart classes come with useful render methods, which can also be used by custom chart implementations. However, if a new chart variant greatly deviates from the base chart type, it may be advisable to create a completely new chart, as discussed in Section C.1.

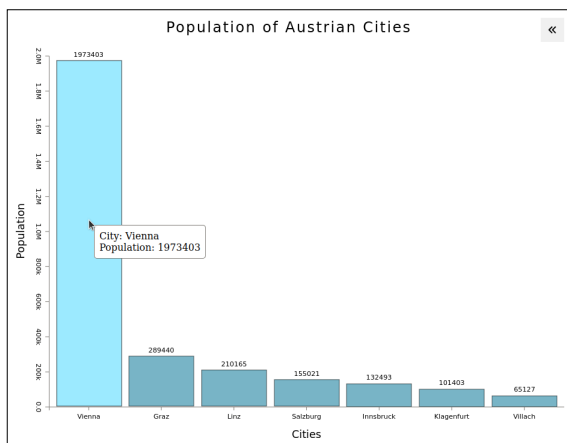


```

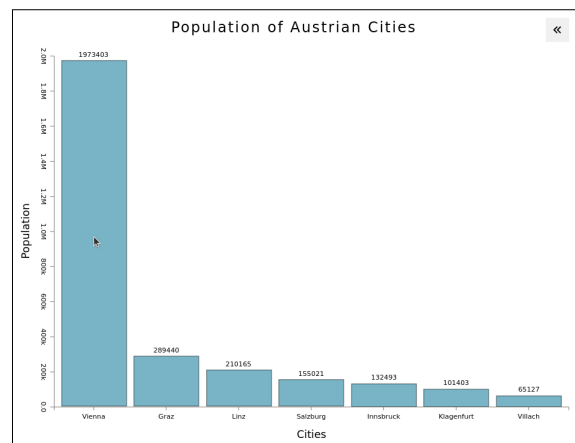
1 //Example for custom chart with no highlight and tooltips
2 class BarChartCustom extends BarChart {
3   renderContent() {
4     this.renderSeriesChartComponents()
5     const series = this.chart$.datum().series.cloneToRenderData()
6       .applyZoom().applyFilter()
7     const {seriesS, barS} = renderBarSeries(this.drawArea$, [series])
8     const bars = barS.data()
9
10    seriesS.call((s) => this.addSeriesLabels(s))
11    this.drawArea$.selectAll('.series-label')
12      .attr( 'layout-strategy', bars[0]?.labelData?.positionStrategy ?? null)
13
14    this.renderCartesianComponents()
15    this.addFilterListener()
16  }
17 }

```

**Listing C.3:** TypeScript code for creating a custom Bar Chart with no highlighted elements and no Tooltips.



(a) Standard Bar Chart.



(b) Customized Bar Chart.

**Figure C.2:** A Bar Chart supports element highlighting on hover and a Data Series Tooltip by default. This can be changed when creating a custom version of the chart. [Screenshots taken by the author of this thesis.]



# Appendix D

## Maintainer Guide

This guide explains how to contribute to RespVis v3 and is addressed to developers working with the internals of RespVis.

### D.1 Releasing

RespVis contains multiple sub-packages and is publicly available at the npm registry. All sub-packages follows semantic versioning. The following checklist details the steps necessary to prepare a new release of RespVis:

- Checking out the `develop` branch in the RespVis repository.
- Raise the version of all sub-packages and the monolithic package by changing the `version` entry in the corresponding `package.json` files. They all must have the same version.
- Raise the version of all RespVis peer-dependencies in all RespVis sub-packages. They must match with the currently released version number of the sub-packages and the monolithic package.
- Make sure all packages follow the import/export policy discussed in Section D.2.
- Commit all conducted versioning and import/export changes as release commit to the `develop` branch and subsequently merging the `develop` branch into the `master` branch by doing a fast-forward merge (a merge without creating a merge commit).
- Check out the `master` branch.
- Create a new tag. It must be named: `v<major>.<minor>.<patch>` and match with the versions of the `package.json` files.
- Push the created tag.
- Build all sub-packages and the monolithic package by running `npx gulp build` or `npm run build`. The execution of these public Gulp tasks can take a couple of minutes, depending on the used device.
- Publish all packages on the public npm registry. This is achieved by first changing the working directory to the project root directory `respvis/` and then invoking the following npm commands:

```
npm publish --dry-run --workspaces npm publish --dry-run npm publish
--workspaces npm publish
```

The first command can be used to test if the correct files are published on the actual execution of the `publish` command for all sub-packages. The second command tests the same for the monolithic

package. The third command publishes all sub-packages. The fourth command publishes the monolithic package.

- Create a release from the new tag on GitHub.
- Zip the `respvis.js`, `respvis.js.map`, `respvis.min.js`, `respvis.min.js.map`, `respvis.d.ts` files located in the `respvis/package/standalone/esm/` directory and the `respvis/package/respvis.css` file as a `package.zip` file and attach this file to the created release on GitHub.

## D.2 Importing and Exporting

Each RespVis sub-package contains CSS and TypeScript source files. Importing functionality from one RespVis sub-package to another requires a clearly defined import/export policy, since each sub-package is publicly available on the npm registry and provides the bundled code and CSS of a specific part of the whole RespVis library.

### D.2.1 Importing and Exporting TypeScript

A bundler must be able to differentiate between importing dependencies from another module in the same sub-package, and importing dependencies from other sub-packages. Otherwise, a bundler would also include code from other sub-packages when creating the bundle of a sub-package, bloating it up unnecessarily. RespVis makes use of path aliases defined in the `tsconfig.json` to avoid this form of double bundling, and to enable importing functionality between sub-packages.

It is important to understand the bundling process. Each sub-package of the RespVis library has an entry file `index.ts` located in the directory `respvis/src/packages/<sub-package>/ts/`. The bundler will check everything exported from this entry file and include it in the final bundle. To gain more fine-grained control, additional `index.ts` files exist for all nested directories. The entry file `index.ts` exports all desired functionality of a sub-package by exporting all nested `index.ts` files. There is also another entry file located in the directory `respvis/src/packages/`, which exports the entry files of all sub-packages. This entry file is used to create the monolithic bundle, which contains the full functionality of RespVis.

The following import and export scenarios use `respvis-bar` as the dependent package and `respvis-core` as the providing package:

1. A file in `respvis-core` imports from another file in `respvis-core`. In this case, using relative paths is advisable, since using the path alias may lead to problems and slowdowns in the bundling process.
2. A file in `respvis-bar` imports functionality from `respvis-core`. In this case an import must be declared via path alias as `import {<functionality>} from 'respvis-core'`. When following this approach, the bundler recognizes the imported functionality as an external dependency coming from another sub-package, and does not include the imported code in the generated bundle.
3. An `index.ts` reexports a named type from a module. For this case, it is not enough to export in this form: `export {<type-name>} from "<path-to-file>"`. Instead, type exports must be used: `export type {<type-name>} from "<path-to-file>"`. This is necessary, because the live documentation is built using Vite, which has different requirements for its transpiling process.

### D.2.2 Importing CSS

The bundling process for CSS is simpler than for TypeScript, because there is no need to import dependencies from other sub-packages. Instead, all CSS files of a sub-package must be included in its bundle. Each sub-package contains an entry file `index.css` in the directory `respvis/src/packages/<sub-package>/css/`. Other CSS files can be imported using import statements in the form of

`@import '<relative-path>';`. In the `build` task, Gulp substitutes import statements with the real CSS code and generates a single CSS file, which is placed into the generated package directory. This procedure is conducted for all sub-packages. There is also another entry file located in the directory `respvis/src/packages/`, which imports the entry files of all sub-packages. This entry file is used to create the monolithic CSS file, which contains all styles provided by RespVis.



# Bibliography

- AMD [2016]. *Asynchronous Module Definition (AMD) API*. 09 Feb 2016. <https://github.com/amdjs/amdjs-api> (cited on page 5).
- Ander [2021]. *Germany Cars Dataset*. 2021. <https://kaggle.com/datasets/ander289386/cars-germany?resource=download> (cited on page 31).
- Andrews, Keith [2018a]. *Responsive Data Visualisation*. 2018. <https://projects.isds.tugraz.at/respvis> (cited on pages 20, 27, 89–90).
- Andrews, Keith [2018b]. *Responsive Visualisation*. CHI 2018 Workshop on Data Visualization on Mobile Devices (MobileVis 2018) (Montréal, Québec, Canada). 21 Apr 2018. [https://mobilevis.github.io/assets/mobilevis2018\\_paper\\_4.pdf](https://mobilevis.github.io/assets/mobilevis2018_paper_4.pdf) (cited on pages 1, 19–21).
- Andrews, Keith [2021]. *Writing a Thesis: Guidelines for Writing a Master’s Thesis in Computer Science*. Graz University of Technology, Austria. 10 Nov 2021. <https://ftp.isds.tugraz.at/pub/keith/thesis/> (cited on page xiii).
- Andrews, Keith [2024]. *Information Visualisation Course Notes*. 08 Mar 2024. <https://courses.isds.tugraz.at/ivis/ivis.pdf> (cited on pages 1, 17).
- Andrews, Keith, David Egger, and Peter Oberrauner [2023]. *RespVis: A D3 Extension for Responsive SVG Charts*. Proc. 27<sup>th</sup> International Conference Information Visualisation (IV 2023) (Tampere, Finland). 25 Jul 2023, pages 19–22. doi:10.1109/IV60283.2023.00014. <https://ftp.isds.tugraz.at/pub/papers/andrews-iv2023-respvis.pdf> (cited on page 33).
- Angelica, Laura [2024]. *The Complete JavaScript Module Bundlers Guide*. 06 Jun 2024. <https://mockit.t.wondershare.com/dictionary/what-is-baseline.html> (cited on page 56).
- Anichiti, Andreea [2021]. *TypeScript Coding Standards*. 11 May 2021. <https://gist.github.com/anichitiandreea/e1d466022d772ea22db56399a7af576b> (cited on page 44).
- BairesDev [2024]. *Static vs Dynamic Typing: A Detailed Comparison*. BairesDev Editorial Team, 09 Feb 2024. <https://bairesdev.com/blog/static-vs-dynamic-typing> (cited on page 6).
- Bederson, Benjamin B. and James D. Hollan [1995]. *Pad++: A Zoomable Graphical Interface System*. Conference Companion, ACM Conference on Human Factors in Computing Systems (CHI 1995) (Denver, Colorado, USA). 07 May 1995, pages 23–24. doi:10.1145/223355.223394. [https://cs.umd.edu/~bederson/images/pubs\\_pdfs/p23-bederson.pdf](https://cs.umd.edu/~bederson/images/pubs_pdfs/p23-bederson.pdf) (cited on page 28).
- Berners-Lee, Tim [1999]. *Weaving the Web*. Harper, 22 Sep 1999. 240 pages. ISBN 0062515861 (cited on page 3).
- Berners-Lee, Tim and Daniel W. Connolly [1995]. *Hypertext Markup Language - 2.0*. IETF, Nov 1995. <https://datatracker.ietf.org/doc/html/rfc1866> (cited on page 3).
- Bos, Bert [2016]. *A Brief History of CSS until 2016*. World Wide Web Consortium, 17 Dec 2016. <https://w3.org/Style/CSS20/history.html> (cited on page 4).

- Bostock, Michael, Vadim Ogievetsky, and Jeffrey Heer [2011]. *D3: Data-Driven Documents*. IEEE Transactions on Visualization and Computer Graphics 17.12 (23 Oct 2011), pages 2301–2309. doi:10.1109/TVCG.2011.185. <https://idl.cs.washington.edu/files/2011-D3-InfoVis.pdf> (cited on page 8).
- Bostock, Mike [2024a]. *D3-Axis*. 27 Aug 2024. <https://d3js.org/d3-axis> (cited on page 10).
- Bostock, Mike [2024b]. *D3-Dispatch*. 27 Aug 2024. <https://d3js.org/d3-dispatch> (cited on page 10).
- Bostock, Mike [2024c]. *D3-Drag*. 27 Aug 2024. <https://d3js.org/d3-drag> (cited on page 10).
- Bostock, Mike [2024d]. *D3-Scale*. 27 Aug 2024. <https://d3js.org/d3-scale> (cited on page 10).
- Bostock, Mike [2024e]. *D3-Selection*. 27 Aug 2024. <https://d3js.org/d3-selection> (cited on page 8).
- Bostock, Mike [2024f]. *D3-Transition*. 27 Aug 2024. <https://d3js.org/d3-transition> (cited on page 8).
- Bostock, Mike [2024g]. *D3-Zoom*. 27 Aug 2024. <https://d3js.org/d3-zoom> (cited on page 10).
- Bostock, Mike [2024h]. *D3: The JavaScript Library for Bespoke Data Visualization*. 27 Aug 2024. <https://d3js.org/> (cited on pages 1, 8–9, 33, 35, 39).
- Budiu, Raluca and Kara Pernice [2016]. *Mobile First Is NOT Mobile Only*. 24 Jul 2016. <https://nngroup.com/articles/mobile-first-not-mobile-only> (cited on page 13).
- Choudhury, Shilpi [2014]. *Grid Lines: Chart Junk or Visual Aids?* Data Visualization Standards, 19 Jun 2014. <https://fusioncharts.com/blog/grid-lines-chart-junk-or-visual-aids/> (cited on page 74).
- Christensson, Per [2019]. *Resolution Definition*. 26 Aug 2019. <https://techterms.com/definition/resolution> (cited on page 18).
- Coyier, Chris [2016]. *The SVG 2 Conundrum*. 11 Nov 2016. <https://css-tricks.com/svg-2-conundrum/> (cited on page 8).
- Deveria, Alexis [2024]. *Can I Use*. 07 Apr 2024. <https://caniuse.com/css-container-queries-style> (cited on pages 14, 65).
- Dixin [2024]. *Understanding (all) JavaScript Module Formats and Tools*. 09 Feb 2024. <https://weblogs.asp.net/dixin/understanding-all-javascript-module-formats-and-tools> (cited on page 5).
- DVS [2024]. *Labels*. U.S. Census Bureau, Data Visualization Standards, 11 Dec 2024. <https://xdgov.github.io/data-design-standards/components/labels> (cited on page 63).
- Ecma [2015]. *ECMAScript 2015 Language Specification*. 6<sup>th</sup> Edition. Ecma International. 01 Jun 2015. [https://ecma-international.org/wp-content/uploads/ECMA-262\\_6th\\_edition\\_june\\_2015.pdf](https://ecma-international.org/wp-content/uploads/ECMA-262_6th_edition_june_2015.pdf) (cited on page 5).
- Ecma [2024]. *ECMAScript 2024 Language Specification*. 15<sup>th</sup> Edition. Ecma International. 01 Jun 2024. <https://ecma-international.org/publications-and-standards/standards/ecma-262> (cited on page 4).
- Egger, David [2024a]. *Responsive Visualization Patterns and Tools*. Survey Paper. 706.424 Seminar/Project Interactive and Visual Information Systems SS 2023. Graz University of Technology, Austria, 26 Apr 2024. 46 pages. <https://ftp.isds.tugraz.at/pub/surveys/egger-2024-04-26-survey-respvis-patterns-tools.pdf> (cited on pages 20, 103).
- Egger, David [2024b]. *respvis*. 28 Nov 2024. <https://npmjs.com/package/respvis> (cited on page 38).
- Egger, David [2024c]. *respvis-bar*. 28 Nov 2024. <https://npmjs.com/package/respvis-bar> (cited on page 38).
- Egger, David [2024d]. *respvis-cartesian*. 28 Nov 2024. <https://npmjs.com/package/respvis-cartesian> (cited on page 38).



- Egger, David [2024e]. *respvis-core*. 28 Nov 2024. <https://npmjs.com/package/respvis-core> (cited on page 37).
- Egger, David [2024f]. *respvis-line*. 28 Nov 2024. <https://npmjs.com/package/respvis-line> (cited on page 38).
- Egger, David [2024g]. *respvis-parcoord*. 28 Nov 2024. <https://npmjs.com/package/respvis-parcoord> (cited on page 38).
- Egger, David [2024h]. *respvis-point*. 28 Nov 2024. <https://npmjs.com/package/respvis-point> (cited on page 38).
- Egger, David [2024i]. *respvis-tooltip*. 28 Nov 2024. <https://npmjs.com/package/respvis-tooltip> (cited on page 37).
- Egger, David [2024j]. *RespVis: Responsive Visualisations*. 28 Nov 2024. <https://respvis-docs.netlify.app/> (cited on pages 1, 35, 41).
- Egger, David and Peter Oberrauner [2023a]. *RespVis v2 Release*. 25 Jul 2023. <https://github.com/tugraz-isds/respvis/releases/tag/v2.0.0> (cited on page 2).
- Egger, David and Peter Oberrauner [2023b]. *Respvis-V2 Release Demo*. 24 Apr 2023. <https://respvis.netlify.app/> (cited on page 27).
- Egger, David and Peter Oberrauner [2024a]. *RespVis v3 Release Demo*. 28 Nov 2024. <https://respvis.netlify.app/> (cited on pages 1, 20–24, 26, 28–29, 61–62, 66, 69, 74–76, 78–80, 82–83, 85–86, 107, 113, 119, 126, 136).
- Egger, David and Peter Oberrauner [2024b]. *RespVis v3 Repository*. 24 Aug 2024. <https://github.com/tugraz-isds/respvis> (cited on pages 1, 91).
- Egnyte [2022]. *Data Sampling*. 19 Apr 2022. <https://egnyte.com/guides/life-sciences/data-sampling> (cited on page 31).
- Gardón, Diego Salinas [2022]. *The Complete JavaScript Module Bundlers Guide*. 21 Mar 2022. <https://snipcart.com/blog/javascript-module-bundler> (cited on page 11).
- Gillies, James and Robert Cailliau [2000]. *How The Web Was Born: The Story of the World Wide Web*. Oxford University Press, 07 Dec 2000. 392 pages. ISBN 0192862073 (cited on page 3).
- Gulp [2024]. *Gulp*. 22 Aug 2024. <https://gulpjs.com> (cited on page 11).
- Highsoft [2023]. *Highcharts*. 30 Aug 2023. <https://highcharts.com/> (cited on page 27).
- Hoffmann, Jay [2017]. *The Origin of the IMG Tag*. 07 Mar 2017. <https://thehistoryoftheweb.com/the-origin-of-the-img-tag> (cited on page 6).
- Hoffswell, Jane, Wilmot Li, and Zhicheng Liu [2020]. *Techniques for Flexible Responsive Visualization Design*. Proc. ACM Conference on Human Factors in Computing Systems (CHI 2020) (Online). 25 Apr 2020, Paper 648, pages 1–13. doi:10.1145/3313831.3376777. <https://jhoffswell.github.io/website/resources/papers/2020-ResponsiveVisualization-CHI.pdf> (cited on page 20).
- Horak, Tom, Wolfgang Aigner, Matthew Brehmer, Alark Joshi, and Christian Tominski [2021]. *Responsive Visualization Design for Mobile Devices*. In: *Mobile Data Visualization*. Edited by Bongshin Lee, Raimund Dachsel, Petra Isenberg, and Eun Kyoung Choe. CRC Press, 23 Dec 2021. Chapter 2, pages 33–65. ISBN 0367534711. doi:10.1201/9781003090823-2. [https://imld.de/cnt/uploads/Horak2021\\_MobileDataVisBook\\_Chap02\\_Responsive.pdf](https://imld.de/cnt/uploads/Horak2021_MobileDataVisBook_Chap02_Responsive.pdf) (cited on page 18).
- Infogram [2016]. *Key Figures in the History of Data Visualization*. 15 Jun 2016. <https://medium.com/@Infogram/key-figures-in-the-history-of-data-visualization-30486681844c> (cited on page 1).

- InfoVis:Wiki [2006]. *Zoom*. 05 Oct 2006. <https://infovis-wiki.net/wiki/Zoom> (cited on pages 27, 69, 99).
- InfoVis:Wiki [2014]. *Semantic Zoom*. 10 Jul 2014. [https://infovis-wiki.net/wiki/Semantic\\_Zoom](https://infovis-wiki.net/wiki/Semantic_Zoom) (cited on page 28).
- Inselberg, Alfred [2009]. *Parallel Coordinates: Visual Multidimensional Geometry and Its Applications*. Springer, 08 Oct 2009. 554 pages. ISBN 0387215077 (cited on pages 18, 84, 125).
- Irish, Paul and Tali Garsiel [2011]. *How Browsers Work*. 05 Aug 2011. <https://web.dev/articles/howbrowserswork> (cited on page 4).
- Jackson, Joab [2012]. *Microsoft Augments JavaScript for Large-Scale Development*. IDG News Service, 01 Oct 2012. <https://infoworld.com/article/2275762/microsoft-augments-javascript-for-large-scale-development-2.html> (cited on page 6).
- Juviler, Jamie [2021]. *Horizontal Scrolling in Web Design: How to Do It Well*. 14 Jun 2021. <https://blog.hubspot.com/website/horizontal-scrolling> (cited on page 14).
- Kim, Hyeok, Dominik Moritz, and Jessica Hullman [2021]. *Design Patterns and Trade-Offs in Responsive Visualization for Communication*. Computer Graphics Forum 40.3 (29 Jun 2021), pages 459–470. ISSN 1467-8659. doi:10.1111/cgf.14321. <https://arxiv.org/abs/2104.07724> (cited on pages 18–20, 103).
- Kirk, Andy [2019]. *Data Visualisation: A Handbook for Data Driven Design*. 2<sup>nd</sup> Edition. Sage Publications, 08 Jul 2019. 328 pages. ISBN 1526468921 (cited on pages 60–61, 72, 76, 78–80, 83, 99, 106, 112, 118).
- Korduba, Yaryna, Stefan Schintler, and Andreas Steinkellner [2022]. *Responsive Data Visualization*. Survey Paper. Information Visualisation SS 2022. Graz University of Technology, Austria, 31 May 2022. 33 pages. <https://courses.isds.tugraz.at/ivis/surveys/ss2022/ivis-ss2022-g2-survey-responsive.pdf> (cited on page 19).
- Kunz, Gion [2017]. *Chartist*. 08 Dec 2017. <https://gionkunz.github.io/chartist-js/> (cited on page 31).
- Manik [2020]. *How JavaScript Was Created and Why the History Behind It Is Important*. 05 Sep 2020. <https://hackernoon.com/how-javascript-was-created-and-why-the-history-behind-it-is-important-fwh3tco> (cited on page 4).
- Marcotte, Ethan [2010]. *Responsive Web Design*. 25 May 2010. <https://alistapart.com/article/responsive-web-design/> (cited on pages 12, 14).
- Marcotte, Ethan [2011]. *Responsive Web Design*. A Book Apart, 07 Jun 2011. 143 pages. ISBN 098444257X. <http://abookapart.com/products/responsive-web-design> (cited on page 12).
- Marcotte, Ethan [2014]. *Responsive Web Design*. 2<sup>nd</sup> Edition. A Book Apart, 02 Dec 2014. 153 pages. ISBN 1937557189. <http://abookapart.com/products/responsive-web-design> (cited on page 1).
- MDN [2023a]. *CSS Flexible Box Layout*. MDN Web Docs, 24 May 2023. [https://developer.mozilla.org/en-US/docs/Web/CSS/CSS\\_flexible\\_box\\_layout](https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_flexible_box_layout) (cited on page 14).
- MDN [2023b]. *CSS Grid Layout*. MDN Web Docs, 15 Jun 2023. [https://developer.mozilla.org/en-US/docs/Web/CSS/CSS\\_grid\\_layout](https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_grid_layout) (cited on page 14).
- MDN [2023c]. *CSS Values and Units*. MDN Web Docs, 06 Sep 2023. [https://developer.mozilla.org/en-US/docs/Learn/CSS/Building\\_blocks/Values\\_and\\_units](https://developer.mozilla.org/en-US/docs/Learn/CSS/Building_blocks/Values_and_units) (cited on page 14).
- MDN [2023d]. *Document Object Model (DOM)*. MDN Web Docs, 17 Dec 2023. [https://developer.mozilla.org/en-US/docs/Web/API/Document\\_Object\\_Model](https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model) (cited on page 3).

- MDN [2023e]. *SVG Positions*. MDN Web Docs, 07 Mar 2023. <https://developer.mozilla.org/en-US/docs/Web/SVG/Tutorial/Positions> (cited on page 53).
- MDN [2023f]. *Web APIs*. MDN Web Docs, 20 Feb 2023. <https://developer.mozilla.org/en-US/docs/Web/API> (cited on page 5).
- MDN [2024a]. *@scope*. MDN Web Docs, 26 Jul 2024. <https://developer.mozilla.org/en-US/docs/Web/CSS/@scope> (cited on page 90).
- MDN [2024b]. *Dominant Baseline*. MDN Web Docs, 19 Aug 2024. <https://developer.mozilla.org/en-US/docs/Web/SVG/Attribute/dominant-baseline> (cited on page 56).
- MDN [2024c]. *Introducing the CSS Cascade*. MDN Web Docs, 26 Jul 2024. <https://developer.mozilla.org/en-US/docs/Web/CSS/Cascade> (cited on page 4).
- MDN [2024d]. *JavaScript Modules*. MDN Web Docs, 30 Jul 2024. <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Modules> (cited on pages 5, 46).
- MDN [2024e]. *Kebab Case*. MDN Web Docs, 01 Aug 2024. [https://developer.mozilla.org/en-US/docs/Glossary/Kebab\\_case](https://developer.mozilla.org/en-US/docs/Glossary/Kebab_case) (cited on page 45).
- MDN [2024f]. *Namespaces Crash Course*. MDN Web Docs, 25 Jul 2024. [https://developer.mozilla.org/en-US/docs/Web/SVG/Namespaces\\_Crash\\_Course](https://developer.mozilla.org/en-US/docs/Web/SVG/Namespaces_Crash_Course) (cited on page 7).
- MDN [2024g]. *Text Anchor*. MDN Web Docs, 02 Aug 2024. <https://developer.mozilla.org/en-US/docs/Web/SVG/Attribute/text-anchor> (cited on page 56).
- MDN [2024h]. *The Box Model*. MDN Web Docs, 25 Jul 2024. [https://developer.mozilla.org/en-US/docs/Learn/CSS/Building\\_blocks/The\\_box\\_model](https://developer.mozilla.org/en-US/docs/Learn/CSS/Building_blocks/The_box_model) (cited on page 53).
- MDN [2024i]. *The HTML DOM API*. MDN Web Docs, 25 Jul 2024. [https://developer.mozilla.org/en-US/docs/Web/API/HTML\\_DOM\\_API](https://developer.mozilla.org/en-US/docs/Web/API/HTML_DOM_API) (cited on page 5).
- MDN [2024j]. *The SVG API*. MDN Web Docs, 25 Jul 2024. [https://developer.mozilla.org/en-US/docs/Web/API/SVG\\_API](https://developer.mozilla.org/en-US/docs/Web/API/SVG_API) (cited on page 5).
- MDN [2024k]. *What is CSS?* MDN Web Docs, 25 Jul 2024. [https://developer.mozilla.org/en-US/docs/Learn/CSS/First\\_steps/What\\_is\\_CSS](https://developer.mozilla.org/en-US/docs/Learn/CSS/First_steps/What_is_CSS) (cited on page 4).
- MDX [2024]. *Markdown for the Component Era*. 30 Jul 2024. <https://mdxjs.com/> (cited on page 12).
- Melkonyan, Samvel [2023]. *Object-Oriented Programming (OOP) vs Functional Programming (FP)*. Flux Technologies, 05 Sep 2023. <https://fluxtech.me/blog/object-oriented-programming-vs-functional-programming/> (cited on page 44).
- Meta [2022]. *JSX*. 04 Aug 2022. <https://facebook.github.io/jsx> (cited on page 12).
- Meta [2024]. *React*. 30 Jul 2024. <https://react.dev/> (cited on page 11).
- Microsoft [2024a]. *Mixins*. 11 Jul 2024. <https://typescriptlang.org/docs/handbook/mixins.html> (cited on page 71).
- Microsoft [2024b]. *TypeScript: JavaScript with Syntax for Types*. 11 Sep 2024. <https://typescriptlang.org/> (cited on page 1).
- NCEI [2024]. *The Global Anomalies and Index Data*. National Centers for Environmental Information, 21 Feb 2024. <https://ncei.noaa.gov/access/monitoring/global-temperature-anomalies/anomalies> (cited on page 30).
- Netlify [2024]. *Netlify*. 04 May 2024. <https://netlify.com/> (cited on page 34).

- NPM [2023]. *registry*. 05 Jan 2023. <https://docs.npmjs.com/cli/v10/using-npm/registry> (cited on pages 1, 10, 37).
- NPM [2024a]. *npm*. 30 May 2024. <https://docs.npmjs.com/cli/v10/commands/npm> (cited on pages 10, 37).
- NPM [2024b]. *package.json*. 27 Jun 2024. <https://docs.npmjs.com/cli/v10/configuring-npm/package-json> (cited on pages 37, 39).
- Oberrauner, Peter [2022a]. *RespVis v1 Repository*. 12 May 2022. <https://github.com/AlmostBearded/respvis-v1> (cited on pages 2, 33).
- Oberrauner, Peter [2022b]. *RespVis: A Browser-Based, D3 Extension Library for Creating Responsive SVG Charts*. Master's Thesis. Graz University of Technology, Austria, 12 May 2022. 131 pages. <https://ftp.isds.tugraz.at/pub/theses/poberrauner-2022-msc.pdf> (cited on pages 2, 33, 37, 44, 90).
- OpenJS [2024a]. *An Introduction to the npm Package Manager*. OpenJS Foundation, 11 Dec 2024. <https://nodejs.org/en/learn/getting-started/an-introduction-to-the-npm-package-manager> (cited on page 10).
- OpenJS [2024b]. *Node.js*. OpenJS Foundation, 11 Dec 2024. <https://nodejs.org/> (cited on pages 5, 10).
- Osmani, Addy [2021]. *Image Optimization*. Smashing, Apr 2021. ISBN 3945749948. <https://smashingmagazine.com/printed-books/image-optimization/> (cited on page 7).
- Ostrowski, Rafał [2023]. *A Simple Guide to JavaScript Concurrency in Node.js*. 04 Jul 2023. <https://tsh.io/blog/simple-guide-concurrency-node-js> (cited on page 10).
- PennState [2023]. *Font Size on the Web*. Pennsylvania State University, 13 Nov 2023. <https://accessibility.psu.edu/fontsizehtml/> (cited on page 21).
- Pickle, Brian [2023]. *Raster Graphic*. 03 Jan 2023. [https://techterms.com/definition/raster\\_graphic](https://techterms.com/definition/raster_graphic) (cited on page 6).
- Plotly [2023]. *Plotly.js*. 30 Aug 2023. <https://plotly.com/javascript/> (cited on page 27).
- pnpm [2024]. *pnpm*. 20 Aug 2024. <https://pnpm.io> (cited on page 11).
- Rabinowitz, Nick [2014]. *Responsive Data Visualization*. 25 Sep 2014. <https://nrabinowitz.github.io/rdv/?scatterplot> (cited on pages 29–30).
- Rendle, Robin [2019]. *Six Tips for Better Web Typography*. 27 Feb 2019. <https://css-tricks.com/six-tips-for-better-web-typography/> (cited on page 13).
- RequireJS [2024]. *RequireJS: A JavaScript Module Loader*. RequireJS, 01 Aug 2024. <https://requirejs.org> (cited on page 5).
- Ribecca, Severino [2024]. *Parallel Coordinates Plot*. The Data Visualisation Catalogue, 22 Jun 2024. [https://datavizcatalogue.com/methods/parallel\\_coordinates.html](https://datavizcatalogue.com/methods/parallel_coordinates.html) (cited on pages 84, 125).
- Rollup [2024]. *Rollup*. 22 Aug 2024. <https://rollupjs.org> (cited on page 11).
- Sarah, Matilda [2023]. *A Comprehensive Guide to Cluster Analysis: Applications, Best Practices and Resources*. 21 Nov 2023. <https://displayr.com/understanding-cluster-analysis-a-comprehensive-guide/> (cited on page 30).
- Sarkar, Manojit and Marc H. Brown [1992]. *Graphical Fisheye Views of Graphs*. Proc. ACM Conference on Human Factors in Computing Systems (CHI 1992) (Monterey, California, USA). 03 May 1992, pages 83–91. doi:10.1145/142750.142763. [https://www.cs.montana.edu/courses/spring2005/430/pg/ft\\_gateway.cfm.pdf](https://www.cs.montana.edu/courses/spring2005/430/pg/ft_gateway.cfm.pdf) (cited on page 27).

- Satori [2022]. *Data Generalization: The Specifics of Generalizing Data*. 2022. <https://satoricyber.com/data-masking/data-generalization> (cited on page 29).
- SciChart [2024]. *Alternatives to D3.js*. 30 Jan 2024. <https://scichart.com/blog/alternatives-to-d3-js/> (cited on page 35).
- Sethi, Basanta Kumar [2024]. *Top Emerging Data Visualization Trends*. 27 Aug 2024. <https://kellton.com/kellton-tech-blog/top-emerging-data-visualization-trends> (cited on page 89).
- Shadeed, Ahmad [2023]. *The Guide to Responsive Design in 2023 and beyond*. 01 Feb 2023. <https://ishadeed.com/article/responsive-design/> (cited on page 14).
- Soueidan, Sara [2017]. *Auto-Sizing Columns in CSS Grid: 'auto-fill' vs 'auto-fit'*. 29 Dec 2017. <https://css-tricks.com/auto-sizing-columns-css-grid-auto-fill-vs-auto-fit/> (cited on page 14).
- Statista [2023a]. *Most Used Programming Languages among Developers Worldwide as of 2023*. 2023. <https://statista.com/statistics/793628/worldwide-developer-survey-most-used-languages> (cited on page 5).
- Statista [2023b]. *Percentage of Mobile Device Website Traffic Worldwide*. 19 Aug 2023. <https://statista.com/statistics/277125/share-of-website-traffic-coming-from-mobile-devices> (cited on page 1).
- Storybook [2024a]. *Storybook: Build UIs without the Grunt Work*. 20 Jul 2024. <https://storybook.js.org/> (cited on pages 11, 43).
- Storybook [2024b]. *Storybook: Introduction to Addons*. 30 Jul 2024. <https://storybook.js.org/docs/7/addons> (cited on page 11).
- Theekshana, Poorna [2024]. *Understanding CommonJS vs. ES Modules in JavaScript*. 28 Feb 2024. <https://syncfusion.com/blogs/post/js-commonjs-vs-es-modules> (cited on page 5).
- UMD [2024]. *Universal Module Definition (UMD)*. 27 Nov 2024. <https://github.com/umdjs/umd> (cited on page 5).
- Vanderkam, Dan [2024]. *Effective TypeScript*. 2<sup>nd</sup> Edition. O'Reilly, 04 Jun 2024. 401 pages. ISBN 1098155068 (cited on page 6).
- Vujovic, Drazen [2024]. *HTML History: Milestones in the Web Markup Language*. 17 Jun 2024. <https://contentsnare.com/html-history> (cited on page 3).
- W3C [1996]. *Cascading Style Sheets, Level 1*. W3C Recommendation. World Wide Web Consortium, 17 Dec 1996. <https://w3.org/TR/2008/REC-CSS1-20080411/> (cited on page 4).
- W3C [1998]. *Cascading Style Sheets, Level 2 (CSS2)*. W3C Recommendation. World Wide Web Consortium, 12 May 1998. <https://w3.org/TR/2008/REC-CSS2-20080411/> (cited on page 4).
- W3C [2001]. *Scalable Vector Graphics (SVG) 1.0 Specification*. World Wide Web Consortium, 04 Sep 2001. <https://w3.org/TR/2001/REC-SVG-20010904> (cited on page 8).
- W3C [2003]. *Scalable Vector Graphics (SVG) 1.1 Specification*. World Wide Web Consortium, 14 Jan 2003. <https://w3.org/TR/2003/REC-SVG11-20030114> (cited on page 8).
- W3C [2010]. *The Secret Origin of SVG*. World Wide Web Consortium, 18 Nov 2010. [https://w3.org/Graphics/SVG/WG/wiki/Secret\\_Origin\\_of\\_SVG](https://w3.org/Graphics/SVG/WG/wiki/Secret_Origin_of_SVG) (cited on page 7).
- W3C [2011a]. *Cascading Style Sheets, Level 2 Revision 1 (CSS 2.1)*. W3C Recommendation. World Wide Web Consortium, 07 Jun 2011. <https://w3.org/TR/CSS21/> (cited on page 4).
- W3C [2011b]. *Scalable Vector Graphics (SVG) 1.1 (Second Edition)*. World Wide Web Consortium, 16 Aug 2011. <https://w3.org/TR/SVG11> (cited on page 8).

- W3C [2014]. *HTML5: A Vocabulary and Associated APIs for HTML and XHTML*. W3C Recommendation. World Wide Web Consortium, 28 Oct 2014. <https://w3.org/TR/2014/REC-html5-20141028/> (cited on page 3).
- W3C [2017]. *HTML 5.2*. W3C Recommendation. World Wide Web Consortium, 14 Dec 2017. <https://w3.org/TR/2017/REC-html52-20171214/> (cited on page 3).
- W3C [2018]. *Scalable Vector Graphics (SVG) 2*. World Wide Web Consortium, 04 Oct 2018. <https://w3.org/TR/SVG2> (cited on pages 8, 89).
- W3C [2024a]. *Cascading Style Sheets*. World Wide Web Consortium, 23 Oct 2024. <https://w3.org/Style/CSS/> (cited on page 4).
- W3C [2024b]. *W3C: Making the Web Work*. World Wide Web Consortium, 30 Jun 2024. <https://w3.org/> (cited on pages 3–4).
- W3Schools [2023]. *Responsive Web Design - The Viewport*. 19 Aug 2023. [https://w3schools.com/css/css\\_rwd\\_viewport.asp](https://w3schools.com/css/css_rwd_viewport.asp) (cited on page 12).
- Ware, Colin [2021]. *Visual Thinking for Information Design*. 2<sup>nd</sup> Edition. Morgan Kaufmann, 14 Jul 2021. 224 pages. ISBN 0128235675 (cited on page 17).
- Wattenberger, Amelia [2019]. *Fullstack D3 and Data Visualization*. Fullstack, 29 Jul 2019. 608 pages. ISBN 0991344650 (cited on page 8).
- WCAG [2023]. *How to Meet WCAG (Quick Reference)*. Web Content Accessibility Guidelines, 13 Nov 2023. <https://w3.org/WAI/WCAG22/quickref/> (cited on page 21).
- Weinreb, Brea [2019]. *What Are Raster Graphics? Definition, Terms, and File Extensions*. 13 Feb 2019. <https://learn.g2.com/raster-graphics> (cited on page 6).
- WHATWG [2024a]. *HTML Living Standard*. Web Hypertext Application Technology Working Group, 23 Oct 2024. <https://html.spec.whatwg.org/> (cited on page 3).
- WHATWG [2024b]. *Web Hypertext Application Technology Working Group*. 23 Oct 2024. <https://whatwg.org/> (cited on pages 3–4).
- Woltmann, Sven [2024]. *Monorepo – Pros and Cons*. 29 Nov 2024. <https://happycoders.eu/software-craftsmanship/monorepo-pros-and-cons/> (cited on page 38).
- Wroblewski, Luke [2011]. *Mobile First*. A Book Apart, Oct 2011. 130 pages. ISBN 1937557022. <https://abookapart.com/products/mobile-first> (cited on page 13).
- Yarn [2024]. *Yarn*. 20 Aug 2024. <https://yarnpkg.com/> (cited on page 11).
- Zanini, Antonello [2023]. *Data Aggregation – Definition, Use Cases, and Challenges*. 20 Nov 2023. <https://brightdata.com/blog/web-data/data-aggregation> (cited on page 30).