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

Peter Oberrauner

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

Peter Oberrauner B.Sc.

Master's Thesis

to achieve the university degree of

Diplom-Ingenieur

Master's Degree Programme: Software Engineering and Management

submitted to

Graz University of Technology

Supervisor

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

Graz, 03 Dec 2021

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

Peter Oberrauner B.Sc.

Masterarbeit

für den akademischen Grad

Diplom-Ingenieur

Masterstudium: Software Engineering and Management

an der

Technischen Universität Graz

Begutachter

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

Graz, 03 Dec 2021

Diese Arbeit ist in englischer Sprache verfasst.

Statutory Declaration

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

Eidesstattliche Erklärung

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

Date/Datum

Signature/Unterschrift

Abstract

[TODO: Write Abstract]

keywords:

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

Kurzfassung

[TODO: Translate abstract into german]

Contents

Contents iii

List of Figures v

List of Tables vii

List of Listings ix

Acknowledgements xi

Credits xiii

1 Introduction 1

2 Web Technologies 3

2.1 HyperText Markup Language (HTML) . . . 3

2.2 Cascading Style Sheets (CSS) . . . 3

2.2.1 Box Layout . . . 4

2.2.2 Flexible Box (Flexbox) Layout . . . 5

2.2.3 Grid Layout . . . 6

2.3 JavaScript (JS) . . . 7

2.4 TypeScript (TS) . . . 9

2.5 Web Graphics . . . 10

2.5.1 Raster Images . . . 10

2.5.2 Scalable Vector Graphics (SVG) . . . 10

2.5.3 Canvas (2D) . . . 12

2.5.4 Canvas (WebGL) . . . 12

2.6 Layout Engines . . . 15

2.6.1 Browser Engines . . . 15

2.6.2 Yoga . . . 15

2.6.3 FaberJS . . . 15

2.7 Responsive Web Design . . . 16

3 Information Visualization **17**
3.1 History of Information Visualization . . . 19
3.2 Information Visualization Libraries for the Web . . . 24
3.2.1 Data-Driven Documents (D3) . . . 24
3.2.2 Vega and Vega-Lite . . . 26
3.2.3 Template-Based Visualization Libraries . . . 27

4 Responsive Information Visualization **35**
4.1 Responsive Visualization Patterns . . . 35
4.2 Responsive Visualization Examples . . . 37

5 Software Architecture **41**
5.1 Primitives . . . 41
5.1.1 Text . . . 41
5.1.2 Rectangle . . . 41
5.1.3 Circle . . . 41
5.2 Series . . . 41
5.2.1 Bar Series . . . 42
5.2.2 Grouped Bar Series . . . 42
5.2.3 Stacked Bar Series . . . 42
5.2.4 Point Series . . . 42
5.2.5 Line Series . . . 42
5.3 Charts . . . 42
5.3.1 Bar Chart . . . 42
5.3.2 Grouped Bar Chart . . . 42
5.3.3 Stacked Bar Chart . . . 42
5.3.4 Point Chart . . . 42
5.3.5 Line Chart . . . 42
5.4 Chart Windows . . . 42
5.4.1 Bar Chart Window . . . 42
5.4.2 Grouped Bar Chart Window . . . 42
5.4.3 Stacked Bar Chart Window . . . 42
5.4.4 Point Chart Window . . . 42
5.4.5 Line Chart Window . . . 42
5.5 Components . . . 42
5.5.1 Lifecycle . . . 42
5.6 Layouter . . . 42

6 Layouter **43**
6.1 CSS Layouting . . . 43

7 Another Chapter; Maybe about Components? 45

8 Selected Details of the Implementation 47

8.1 D3 Select Function Data Modification 47

8.2 Save as SVG 47

9 Outlook and Future Work 49

9.1 Outlook 49

9.2 Ideas for Future Work 49

9.2.1 Relative Positioning of Series Items 49

9.2.2 Container Queries 49

10 Concluding Remarks 51

A User Guide 53

B Developer Guide 55

Bibliography 57

List of Figures

2.1 CSS Box Model 5
2.2 Flexbox Justify Content Property 6
2.3 Grid Layout Property Comparision 7
2.4 Desktop Browser Market Share 8
2.5 Raster Image Scaling 11
2.6 SVG Scaling 11
2.7 Canvas With Responsive Circles 14

3.1 Anscombe's Quartet 18
3.2 Chart of Planetary Movements from the Tenth Century 19
3.3 Chart of Changes in Sunspots from 1626 20
3.4 Chart of Longitudinal Distance Determinations Between Toledo and Rome From 1644 . . 20
3.5 Line Chart by William Playfair From 1786 21
3.6 Bar Chart by William Playfair from 1786 21
3.7 Area Chart by William Playfair from 1786 22
3.8 Dot Map Plotting Cholera Deaths in London From 1855 22
3.9 Polar-Area Chart by Florence Nightingale From 1859 23
3.10 Screenshot of High-D. 23

4.1 Responsive Bar Chart Example 38
4.2 Responsive Line Chart Example 39
4.3 Responsive Scatterplot Example 40
4.4 Responsive Parallel Coordinates Chart Example 40

List of Tables

2.1 CSS Selector Syntax 4
2.2 TypeScript Type System Design Properties 10

3.1 Categories of Interaction Based on User Intent. 18

4.1 Targets of Responsive Visualization Patterns 37
4.2 Actions of Responsive Visualization Patterns 37

List of Listings

2.1 SVG Document Containing a Circle 11
2.2 Canvas With Responsive Circles 13

3.1 D3 Method Chaining 25
3.2 D3 General Update Pattern 26
3.3 Static Bar Chart in Vega 28
3.4 Bar Chart with Tooltip in Vega 29
3.5 Bar Chart with Tooltip in Vega-Lite 30
3.6 Bar Chart in Highcharts 32
3.7 Bar Chart in D3FC. 33
3.8 Responsive Rules in Highcharts 33

Acknowledgements

[TODO: Write acknoledgements]

Peter Oberrauner
Graz, Austria, 03 Dec 2021

Credits

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

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

[TODO: Add further credits?]

Chapter 1

Introduction

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

[TODO: Outline the various chapters]

Chapter 2

Web Technologies

2.1 HyperText Markup Language (HTML)

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

The primary purpose of HTML is to define the content and structure of web pages. This is achieved with the help of HTML elements, which are composed in a hierarchical tree structure and define modular pieces of content that can be interpreted by web browsers. A strong pillar of HTML's design is extensibility. There are multiple mechanisms in place to ensure applicability to a vast range of use cases. These mechanisms include:

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

2.2 Cascading Style Sheets (CSS)

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

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

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

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

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

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

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

2.2.1 Box Layout

All elements in an HTML document are laid out as boxes, which is defined as the CSS box model. The box model states that every element is wrapped in a rectangular box and every box is described by its content and the optional surrounding margin, border and padding areas. Margins are used to specify the invisible spacing between boxes, whereas the border is meant as a visible containment around the content of a box and the padding describes the invisible spacing between the content and the border. A visual representation of these properties can be seen in Figure 2.1.

In early versions of CSS, before the introduction of the Flexible Box (Flexbox) layout module [W3C 2009], the box model was the only way to lay out elements. Style sheet authors had to meticulously

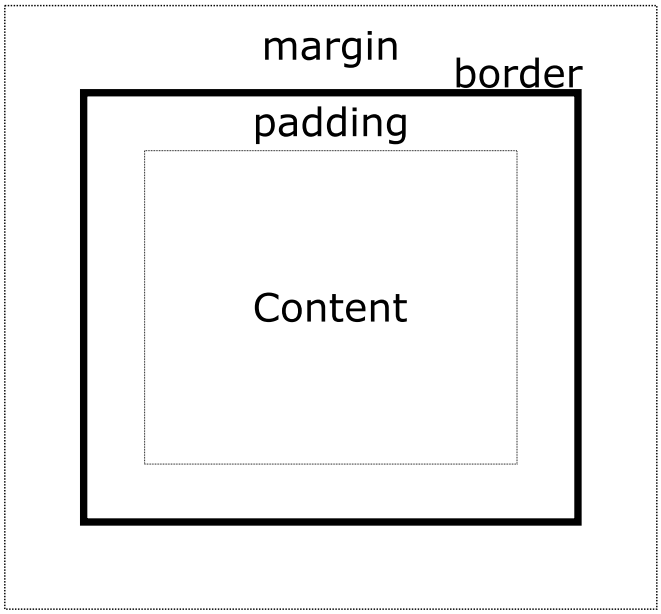


Figure 2.1: The CSS box model is used to define the properties of boxes that wrap around HTML elements. [Image drawn by the author of this thesis.]

define margins of elements and their relative (or absolute) positions in the document tree. The responsive capabilities of this kind of layouting are very limited because different configurations for varying screen sizes have to be specified manually using media queries. More complex features, like the filling of available space, had to be implemented via scripting.

2.2.2 Flexible Box (Flexbox) Layout

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

Flexbox is a mechanism for one-dimensionally laying out elements in either rows or columns. This one-dimensionality is what separates it from grid-based layouting, which is inherently two-dimensional. Flexbox layouting can be enabled for child elements via setting the `display:flex` property on a container element. The direction of the layout can then be specified using the `flex-direction` property which can be set to either `row` or `column`.

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

Another important feature of Flexbox layouting is the controllable spacing of items which can be specified separately for both the main and the cross axis of the layout. Spacing along the main axis can be configured with the `justify-content` property which can take a number of different values that are demonstrated in Figure 2.2. The property that controls alignment of items on the cross axis is either the `align-items` property on container elements or the `align-self` property on the items themselves.

This section only graced the surface of what is possible with the Flexbox layout module. Those properties that were discussed, were only discussed in a very broad sense and there are many useful

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

properties like `flex-grow`, `flex-shrink` or `flex-wrap` that weren't included in this overview. For a more detailed look at this topic it is recommended to review W3C 2018.

2.2.3 Grid Layout

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

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

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

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

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

There are also properties that allow to control the layout of items within grid cells and the layout of grid cells themselves. Similar to how it is described in Section 2.2.2, this can be configured with the

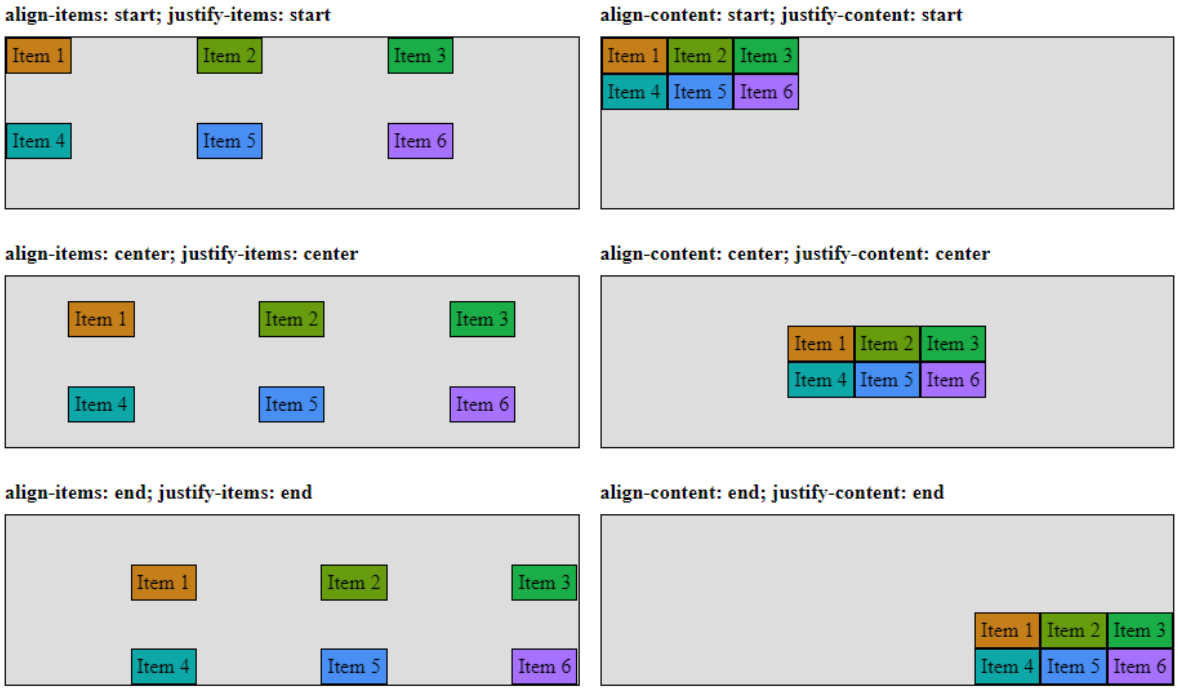


Figure 2.3: The `*-items` properties are used to lay out items within their grid cells, whereas the `*-content` properties are used to lay out the grid cells themselves. [Image created by the author of this thesis.]

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

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

2.3 JavaScript (JS)

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

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

JavaScript has initially been created by Netscape in 1995 [Netscape 1995]. Before that, websites

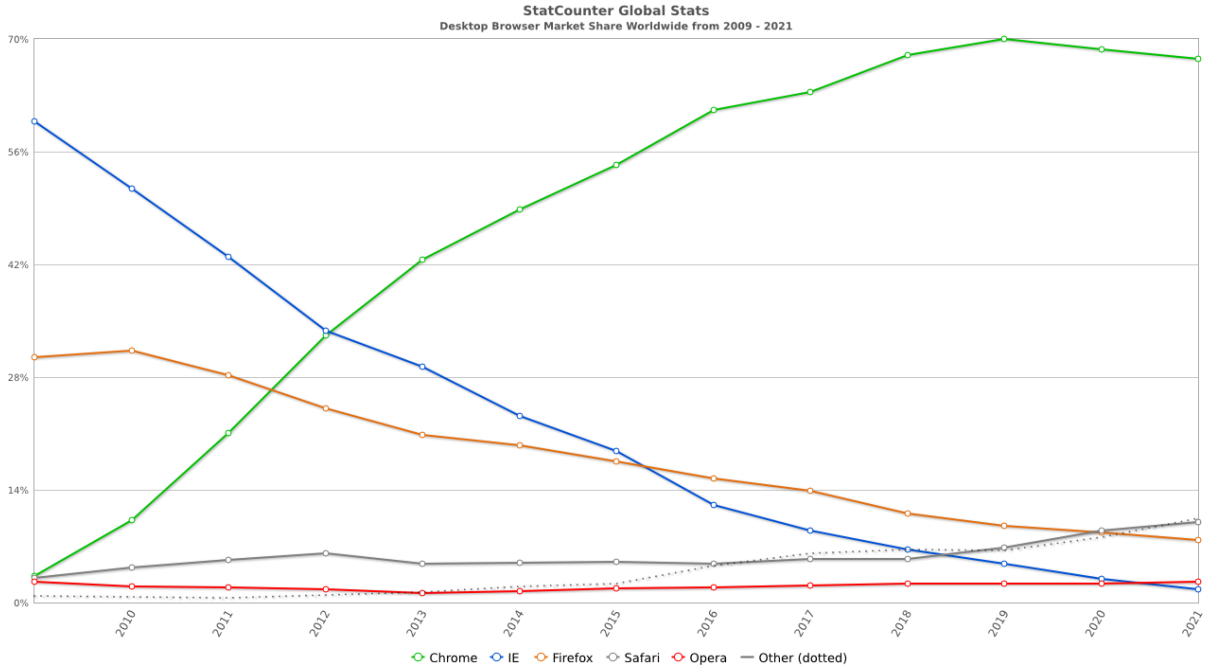


Figure 2.4: Since their release, Firefox and Chrome have contested the monopoly of the Internet Explorer and continuously gained more market share. Recently, Chrome seems to be gaining an increasingly strong position within the market. [Image taken from StatCounter 2021] [TODO: Embed chart as SVG (svg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

were only able to display static content, which drastically limited the usefulness of the web. Seemingly, Microsoft saw JavaScript as a potentially revolutionary development because they reverse-engineered the implementation of Netscape and published their own version of the language for Internet Explorer in 1996 [Microsoft 1996]. Both implementations were noticeably different from one another and the uncontested monopoly of the Internet Explorer [Routley 2020] held back standardization efforts undertaken by Netscape [Ecma International 1997]. When Firefox was released in 2004 [Mozilla 2004] and Chrome in 2008 [Google 2008], they quickly gained a considerable share of the market [StatCounter 2021] (see Figure 2.4). Driven by this new market segmentation, all major browser vendors collaborated on the standardization of JavaScript as ECMAScript 5 in 2009 [Ecma International 2009]. Since then, JavaScript has been continuously developed with its latest version has been released as ECMAScript 6 in 2015 [Ecma International 2015].

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

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

A second important Web API in the context of this work is the `ResizeObserver` API. Its purpose is the

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

2.4 TypeScript (TS)

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

Initially, TypeScript has been released by Microsoft in 2012 [IDG News Service 2012] to extend JavaScript with features that were already present in more mature languages, and whose absence in JavaScript caused difficulties when working on larger codebases. At the time of the initial development, ECMAScript 6 hasn't been published yet and TypeScript was used to already be able to use features that would later be implemented by ECMAScript 6. These features included a module system to be able to split source code into reusable chunks and a class system to aid object-oriented development. The TypeScript code using these features could then be translated via a compiler into standard-conform ECMAScript 5 code, which could be interpreted by JavaScript engines at the time. At the time of writing, ECMAScript 6 is already widely supported by all modern browsers and therefore the main benefit of TypeScript over JavaScript primarily resides in the addition of a static type system.

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

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

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

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

2.5 Web Graphics

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

2.5.1 Raster Images

Raster images represent graphics as a rectangular, two-dimensional grid of pixels with a fixed size. This fixed grid size results in limited scalability and, whenever an image is displayed in a different size, visual scaling artifacts will be noticeable as can be seen in Figure 2.5. Raster images are either created by image capturing devices or special editing software and saved as binary files in varying formats. The most widely used format for raster images is Portable Network Graphics (PNG), which is standardized by the W3C [**PNG**] and optimized for usage on the web. It features a lossless compression and streaming capabilities, which enable progressive rendering of images while they are loaded.

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

2.5.2 Scalable Vector Graphics (SVG)

Scalable Vector Graphics (SVG) is an XML-based format for vector graphics, which describe images as sets of shapes that can be scaled without loss of quality. It was initially published by the W3C in 1999 [**SVG1.0**] and the most recent version SVG 1.1 was released in 2011 [**SVG1.1**]. Because SVG is based on XML, SVG files can be created with any text editor and a simple example of an SVG document can be seen in Listing 2.1 with its visualization being shown in Figure 2.6. In addition to being able to write SVG documents manually, there is also specialized software available that helps with the composition of more complex images.

The encoding in XML leads to SVG being the best format to represent graphics in terms of accessibility. Graphics are directly saved in a hierarchical and textual form, which describes their shapes and how they

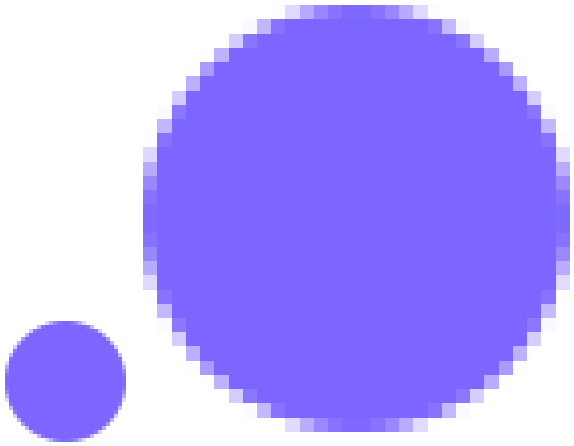


Figure 2.5: This figure demonstrates the artifacts that occur when scaling raster images to a size that is different from the size they were encoded in. (a) shows a raster image of a circle in its original size. (b) shows the same image scaled up considerable. [Image created by the author of this thesis.]

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

Listing 2.1: A simple SVG document containing a circle element. The visual representation of this document in different sizes is shown in Figure 2.6

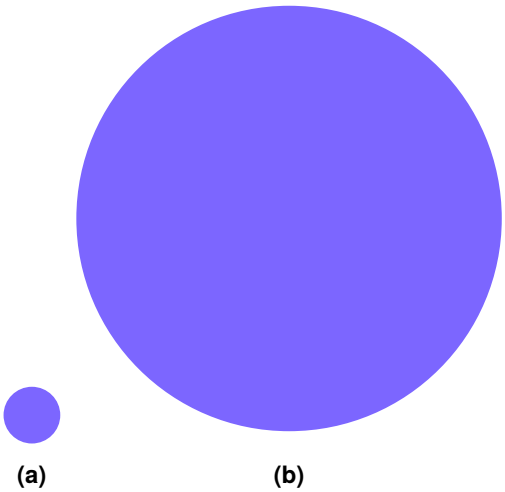


Figure 2.6: SVG documents can be scaled freely without any artifacts occurring. This figure displays the visual representation of the SVG document in Listing 2.1 scaled to different sizes. [Image created by the author of this thesis.]

are composed. In addition to the shapes being inherently accessible, the various elements of an SVG document can be annotated with further information that aids comprehension when consumed in a non-visual way.

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

The most widely supported way of styling SVG elements is via attributes, which is supported by every software dealing with SVG files. However, the specification aims for maximum compatibility with HTML, and therefore it is also possible to use CSS to style and animate SVG elements when they are rendered in a browser. Using CSS to separate presentation from content has many benefits that were already described in Section 2.2. Unfortunately it is not possible to style every SVG attribute with CSS because only the so-called presentation attributes like `fill` and `stroke-width` are available in CSS. These attributes are taxonomically listed in the SVG specification [**SVG1.1**] and have been extended by additional attributes like `x`, `y`, `width` and `height` in upcoming releases [**SVG2**].

2.5.3 Canvas (2D)

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

Rendering is programmed via a low-level API, which is provided by the rendering context of a particular canvas. The type of render context depends on the context mode of a canvas with the two most significant ones being `2d` and `webgl`. A two-dimensional render context is created for canvases that have the `2d` context mode set on them. It enables platform-independent rendering via a software renderer, whose API is standardized directly in the canvas specification. An example of an HTML document containing two differently sized canvases into which responsive circles are drawn using a two-dimensional rendering context can be seen in Listing 2.2 with the corresponding visualization displayed in Figure 2.7.

2.5.4 Canvas (WebGL)

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

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

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

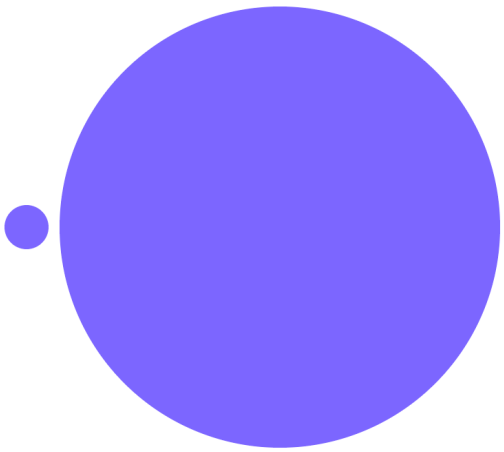


Figure 2.7: Responsive rendering of graphics inside canvas elements has to be implemented manually by calculating everything as ratios of the canvas' dimensions. This figure represents the visual representation of the canvas example in Listing 2.2. [Image created by the author of this thesis.]

2.6 Layout Engines

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

2.6.1 Browser Engines

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

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

The layout module of a browser engine can only be invoked directly by browsers to position HTML elements in actively rendered documents. To use it for calculating layouts of non-HTML constructs, they must be replicated in active documents so that they can be parsed, laid out and rendered by the browser engine. These replicated constructs do not have to be visible, and they can be removed from the document after the layout has been acquired, which means they do not need to be noticeable at all. A strong limitation of using browser engines to calculate layouts is that it requires a browser runtime to work and, even though there are solutions like Electron available that enable development of native applications using web technologies, this forces applications into this very specific stack of technologies.

2.6.2 Yoga

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

2.6.3 FaberJS

FaberJS [**FaberJS**] is a layout engine that is very similar to the Yoga layout engine in the fact that it enables the computation of layouts for constructs other than HTML documents, using a layout grammar that has originally been created for CSS. In contrast to Yoga, which is used to create one-dimensional layouts using the Flexbox layout grammar, FaberJS implements a two-dimensional layout algorithm that is built on the grammar of the Grid layout module (see Section 2.2.3). In the context of two-dimensional information visualization, this inherently two-dimensional approach to layouting makes more sense than

trying to achieve two-dimensionality using a one-dimensional one. FaberJS is an open-source JavaScript project that has been developed since 2019 by Idera, Inc. Even though the layouts it computes are constrained with the grid layout grammar, it only supports a subset of all the functionalities that are defined in the original CSS module. Some examples of missing functionalities include missing support for margins, gaps and the `*-content` and `grid-auto-*` properties. Working around the limitations caused by these missing features is not trivial, and it seems unlikely that support for them will be added by the FaberJS maintainers in the near future because, at the time of writing, the project has not been updated in nearly two years.

2.7 Responsive Web Design

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

Chapter 3

Information Visualization

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

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

Regarding terminology, this work adheres to the separation of the field of visualization into three main subfields as described by **IVISCourseNotes**.

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

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

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

In addition to these three subfields, **IVISCourseNotes** defines the often used term Data Visualization as the intersection of Geographic Visualization and Information Visualization. Therefore, it deals with visualizing spatial as well as abstract data.

Visualizations presented in an interactive medium do not merely consist of visual representations. In order to analyze more complex datasets, it is equally important to provide means for interacting with these representations because without interaction, a visualization becomes a static image, which has only very limited usefulness when dealing with large and multidimensional data. Even though the majority of the attention in the field of Information Visualization has been put on the presentational aspect of visualizations, a lot of research has also been conducted on their interactive aspects. Many

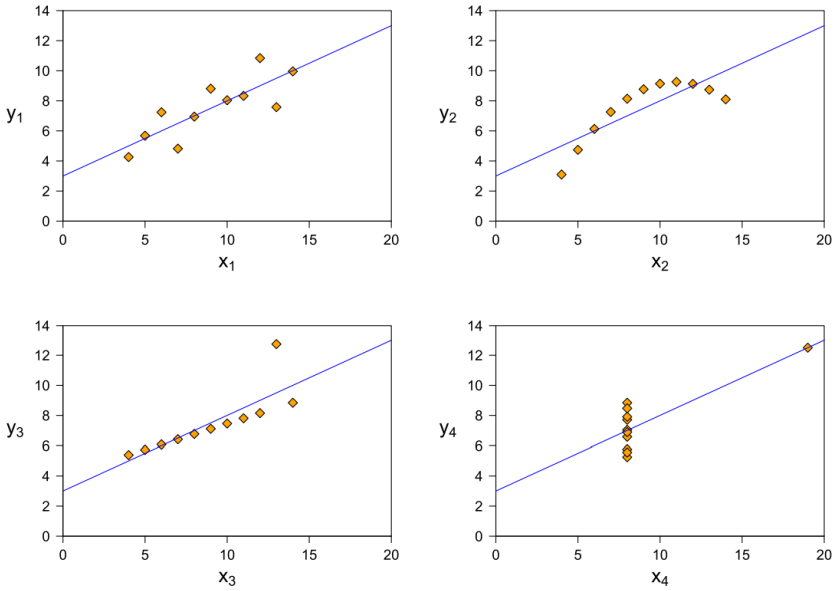


Figure 3.1: Anscombe's Quartet consists of four distinct sets of data that share identical summary statistics. The difference between these datasets is only obvious by carefully examining the textual data or by plotting it. [Image extracted from **IVISCourseNotes** with kind permission by Keith Andrews.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

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

Table 3.1: This table shows the different categories of interacting with visualizations based on user intent. [Table adapted from **RoleOfInteractionInInformationVisualization**]

taxonomies have been formulated with the goal of defining the design space of interactions to support analytic reasoning, but they vary greatly depending on the concepts they are focusing on. Some taxonomies have been defined on the concept of low-level interaction techniques [**TheEyesHaveIt**; **GrammarOfGraphics**], providing a very system-centric view on interaction, while other taxonomies focus on user tasks [**LowLevelComponentsOfAnalyticActivity**] without them being tightly bound to interacting with visualizations. **RoleOfInteractionInInformationVisualization** aims to provide a view that is in between the purely system-centric and purely user-centric extremes by defining a taxonomy based on user intent. The categories of this taxonomy are listed in Table 3.1, and they form a good framework with which interactivity in the context of Information Visualization can be discussed.

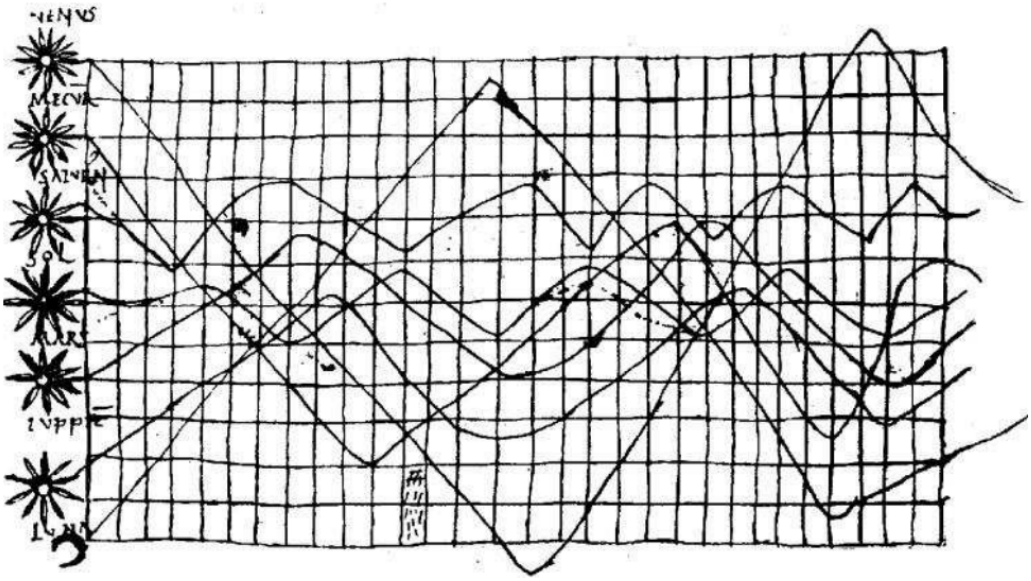


Figure 3.2: A chart created by an unknown astronomer in the tenth century depicting the movements of the seven most prominent planets. [Image extracted from **BriefHistoryOfDataVis**. Original appearance in **CommentariiInSomniumScipionis**.]

3.1 History of Information Visualization

The history of Information Visualization goes back a long time with one of the earliest examples dating back to the 10th century, when an unknown astronomer created a chart about the movement of the most prominent planets [**CommentariiInSomniumScipionis**], shown in Figure 3.2. Other noteworthy early visualizations include the first occurrence of the principle **VisualDisplayOfQuantitativeInformation** later coined "small multiples" in a 1626 chart demonstrating sunspot changes (Figure 3.3) by **RosaUrsina** and a 1644 chart displaying longitudinal distance determinations between Toledo and Rome (Figure 3.4) by Michael Florent van Langren.

William Playfair (1759 - 1823) is by many considered to be one of the forefathers of modern visualizations because his published works contain the first occurrences of many graphical forms that are still widely used today. In one of his earlier works [**CommercialAndPoliticalAtlas**] he introduced the concepts of line (Figure 3.5), bar (Figure 3.6) and area charts (Figure 3.7) to communicate economic factors of England during the eighteenth century. In a related later work [**StatisticalBreviary**] he uses the first ever published pie and circle charts to show and compare the resources of states and kingdoms in Europe. The charts he created are very similar to modern ones because they already contain all the major elements found in today's visualizations, such as labeled axes, grids, titles and color-based categorization.

Albeit not directly an information visualization but rather a geographic one, the dot map created by **ModeOfCommunicationOfCholera** in 1855 to trace cholera outbreaks in London (Figure 3.8) is undoubtedly one of the most famous and influential visualizations in history. It was used to identify a cluster of cholera-related deaths near a contaminated water pump on Broad Street, leading to the recognition of cholera as a waterborne disease.

It would go amiss not to mention Florence Nightingale (1820 - 1910) when talking about the history of information visualization. She was a British statistician, social reformer, founder of modern nursing and might be the first person who used visualizations to persuade others of a need for change. During her service as a superintendent of nurses in the Crimean War, she realized that a large share of deaths in hospitals resulted from preventable diseases that originated in poor sanitary conditions. One of her contributions to the field of information visualization was the creation of a new type of diagram, called a rose or polar-area chart. These charts were used to communicate the data she collected on the mortality

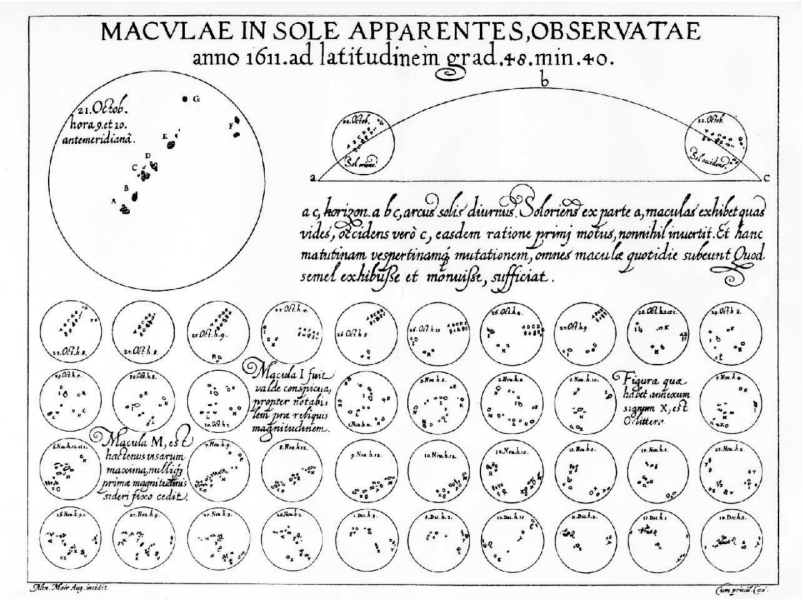


Figure 3.3: This chart shows the observed changes in sunspots based on recordings of two months of data from 1611. It is the first occurrence of the principle later called "small multiples" by **VisualDisplayOfQuantitativeInformation**. [Image extracted from **BriefHistoryOfDataVis**. Original appearance in **RosaUrsina**.]

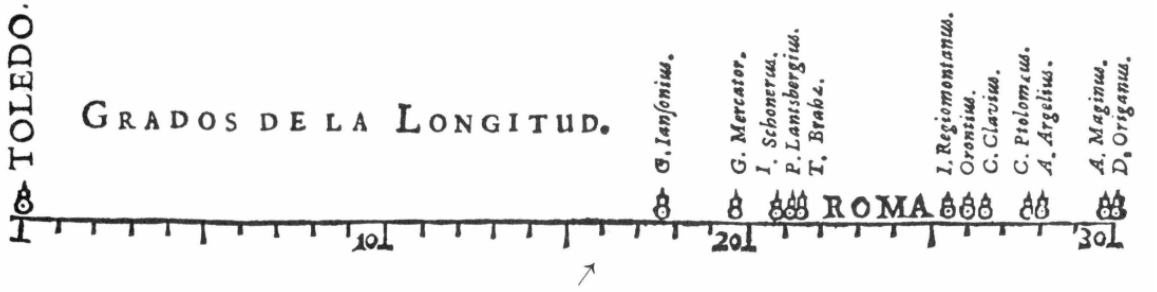


Figure 3.4: This chart compares the twelve known estimates in longitudinal distance between Rome and Toledo by various astronomers. The correct distance is marked by the arrow beneath. It is considered to be the first visual representation of statistical data. [Image extracted from **BriefHistoryOfDataVis**.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

of soldiers during the war and to grab the attention of politicians and the public. One of those charts can be seen in Figure 3.9. [TODO: Cite Nightingale paper]

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

It is out of the scope of this work to provide a detailed account over the long and eventful history of information visualization and this section only provides a brief and very selective view on the topic. Much more comprehensive works exist that go significantly deeper into details of the various actors and the intricate influences they had on each other. Recommendable sources for further reading on the history of visualization include **BriefHistoryOfDataVis**, **HistoryOfDataVisAndGraphicCommunication** and **HistoryOfInformationGraphics**.

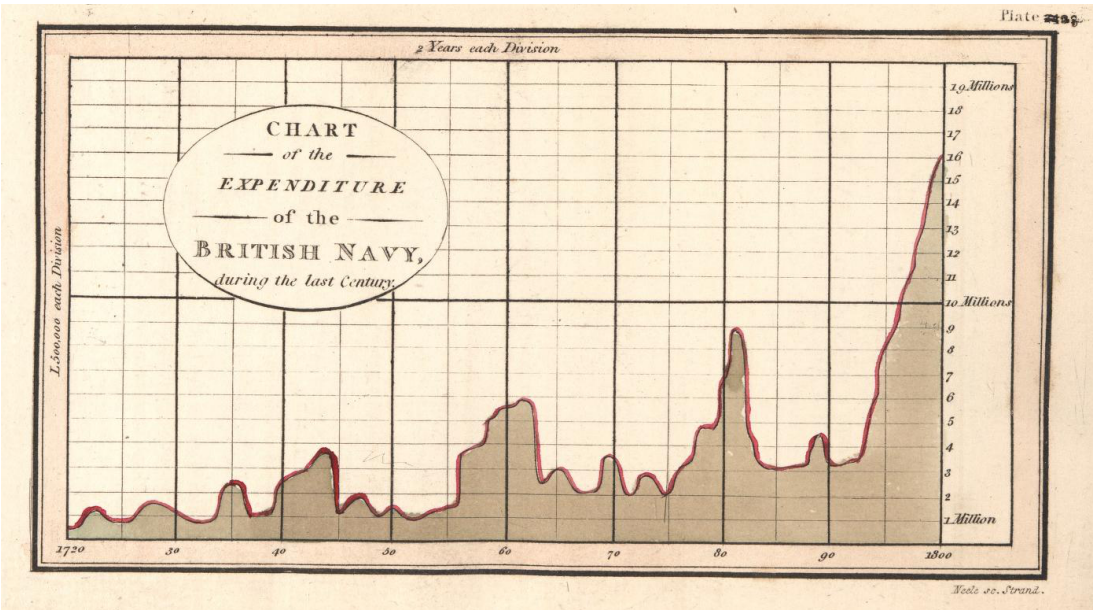


Figure 3.5: This chart shows the expenditure of the British navy during the eighteenth century as a line chart. It was published in 1786 and is considered to be one of the first occurrences of a line chart that contains all components found in modern visualizations. [Image extracted from Schoenberg Center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

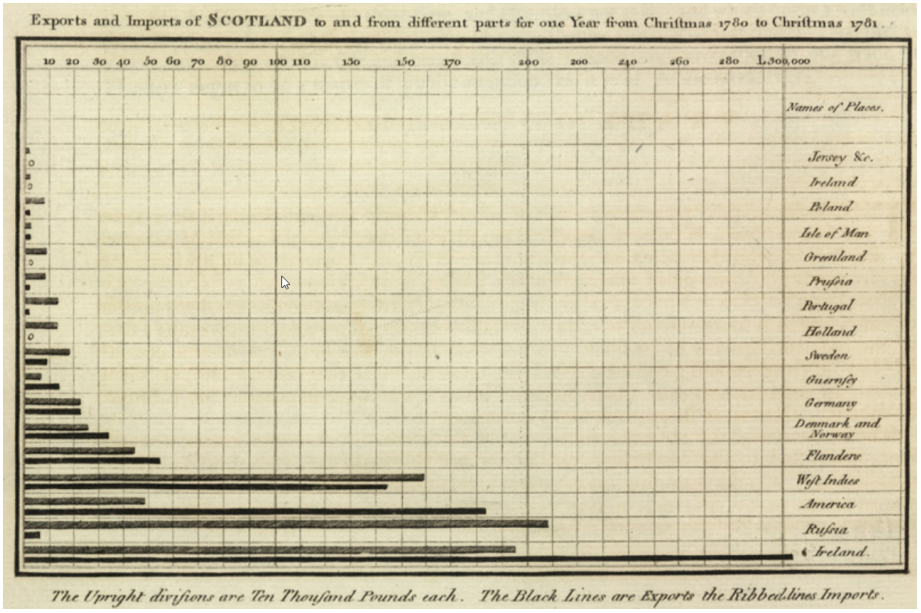


Figure 3.6: This chart shows England's exports and imports from and to Scotland in 1781 visualized as a bar chart. It was published in 1786 and is considered to be one of the first occurrences of a bar chart that contains all components found in modern visualizations. [Image extracted from Schoenberg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

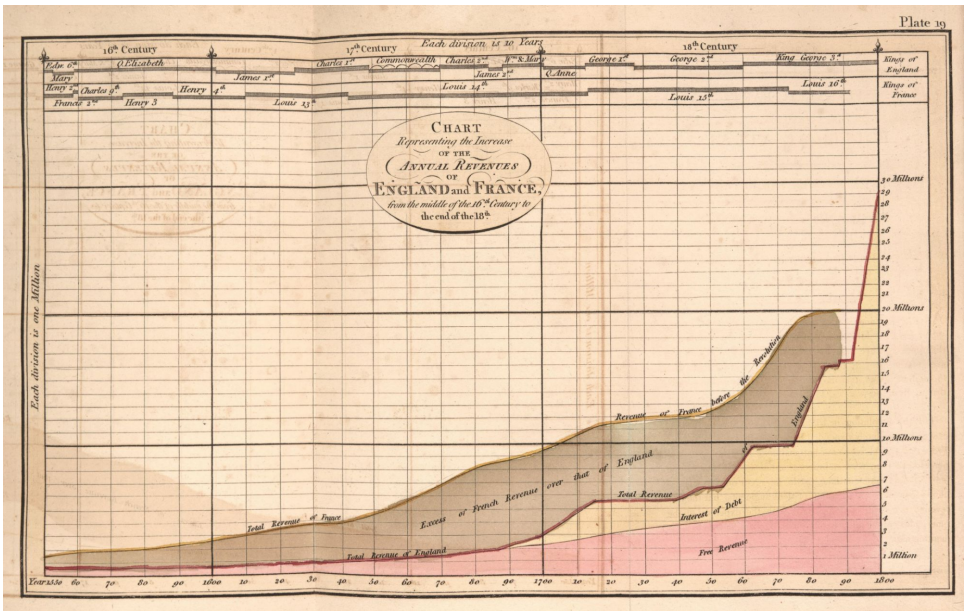


Figure 3.7: This chart shows the annual revenues of England and France between 1550 and 1800 visualized as an area chart. It was published in 1786 and is considered to be one of the first occurrences of an area chart that contains all components found in modern visualizations. [Image extracted from Schoenberg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

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

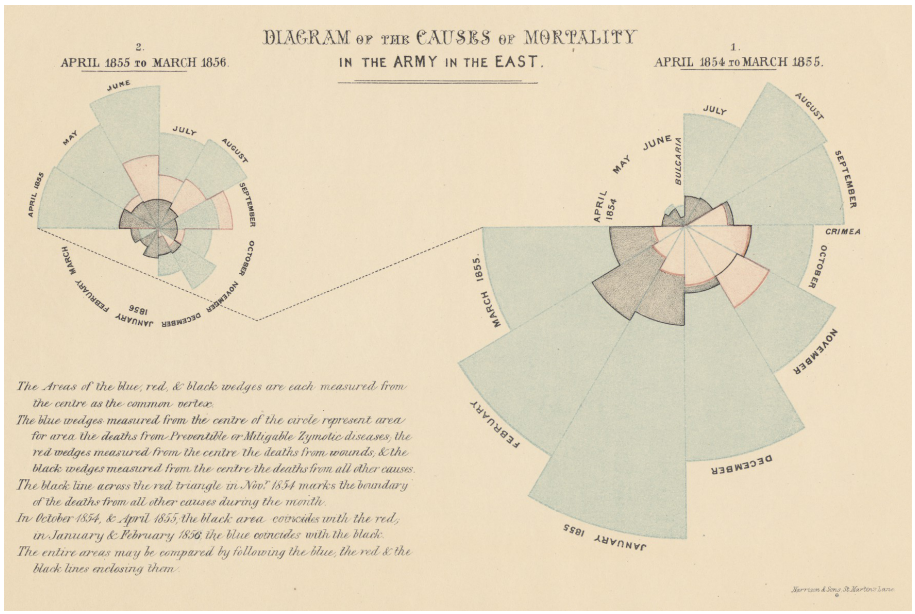


Figure 3.9: This is one of the polar-area charts that Florence Nightingale created in 1859 to convince others of a need for more sanitary conditions in hospitals. It visualizes the causes of mortality for soldiers during the Crimean War and demonstrates that a large percentage of patients died from preventable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

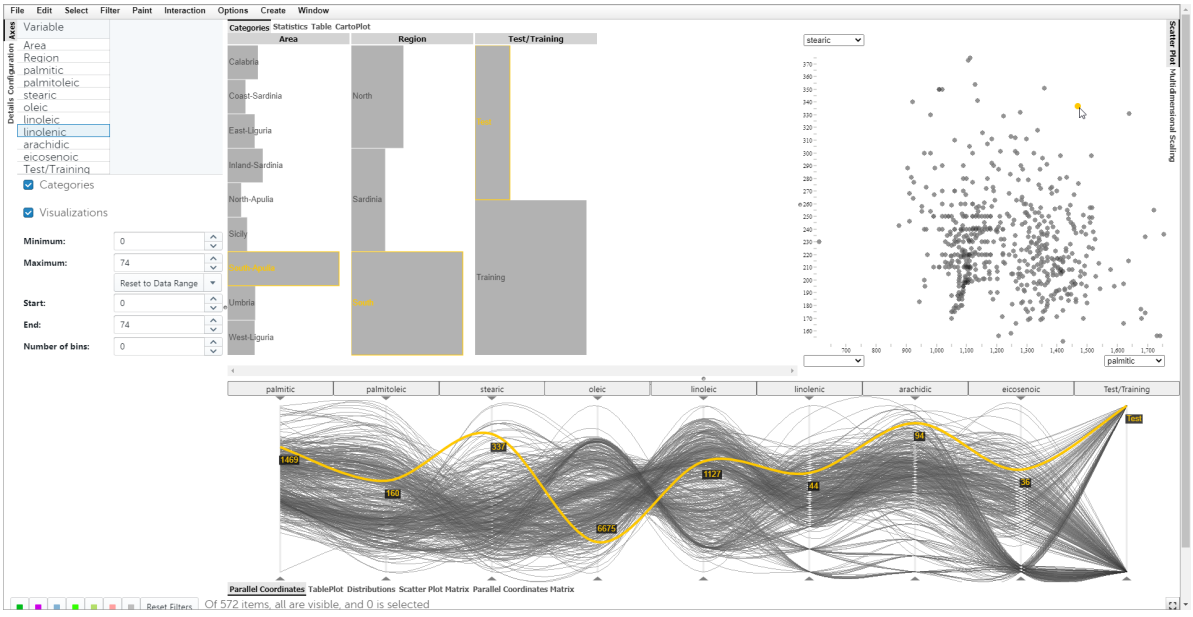


Figure 3.10: High-D from the company Macrofocus is a visual analytics tool that is specialized in analyzing multidimensional data. [Screenshot of **HighD** taken by the author of this work.] [TODO: How to cite this? Copyright?]

3.2 Information Visualization Libraries for the Web

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

3.2.1 Data-Driven Documents (D3)

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

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

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

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

Selections have already been mentioned in the previous paragraphs. They are the atomic building

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

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

blocks of D3 that are used to access almost any functionality. Selections are created using the `d3.select` or `d3.selectAll` methods. These methods are built on the DOM Selectors API, namely the `querySelector` and `querySelectorAll` methods, which allow the selection of elements via CSS selectors (see Section 2.2), and which consequentially allow the `d3.select` and `d3.selectAll` methods to either create a Selection containing a single element matching the provided selector or to create one that contains multiple elements matching it. A Selection acts as a wrapper container around these selected elements, and it provides methods to perform frequently performed DOM operations on them. Among others, the element operations provided by a Selection include the setting and getting of: Attributes using the `Selection.attr` method, styles using the `Selection.style` method, properties using the `Selection.property` method, text or HTML content using the `Selection.text` or `Selection.html` methods, and event listeners using the `Selection.on` method. Selections also provide wrapper methods to add additional elements to the document using the `Selection.append` or `Selection.insert` methods as well as to remove them using the `Selection.remove` method. Accessing the DOM using these methods is less tedious because the API that is natively provided by the DOM is very verbose and also because the method chaining API provided by D3 does not require intermediate variables where none are needed.

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

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

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

Listing 3.2: A demonstration of D3's general update pattern and how it can be used to specify different transformations for entering, updating and exiting elements.

of setting the target value directly. Using D3 Transitions is completely optional and users can also choose to use other animation technologies, like CSS transitions and animations, instead.

At the core, D3 is simply a low-level library to perform data-driven document transformations. Even though this generic core technology is applicable to a wide range of use cases, D3 has been created with a focus on creating visualizations. Many additional modules exist that simplify performing the higher-level tasks necessary for the rendering of visualizations and despite these functionalities being split on multiple modules, they all follow the same patterns inherent to D3 like method chaining. Most modules are implemented as configurable functions that are configured using chainable functions and which can be invoked after configuration to perform a specific action. Listing all available modules here would be out of the scope of this work, but some noteworthy and characteristic ones include: `d3-shape` to create visual primitives like lines and areas, `d3-scale` to encode abstract data dimensions, `d3-axis` to render scales as human-readable axes, and many more such as `d3-array`, `d3-layout` and `d3-zoom`.

3.2.2 Vega and Vega-Lite

Vega [**Vega**] is a library that consists of a grammar to describe interactive graphics and a parser that translates specifications written in this grammar into static images or web-based views built on SVG documents or the Canvas Web API. An interactive visualization in Vega is fully described by a specification written in Vega's grammar, which is a domain specific language designed for the declarative denotation of interactive graphics. The syntax of this grammar is based on JavaScript Object Notation (JSON), an easy-to-read format for data serialization that is among the most frequently used textual serialization formats. It builds on previous research in the field of declarative visualization design [**GrammarOfGraphics**] and, in addition to describing visual appearances, it also contains powerful capabilities to declaratively describe interactions [**ReactiveVega**].

The visual aspects of a visualization are described in a grammar similar to the one defined by **GrammarOfGraphics**. At its top level, a Vega specification contains properties to configure sizing and padding of the container of a visualization. Every specification will also contain a data section that either defines data or specifies where to load it from. The Vega grammar also supports various forms of data transformation that can successively be applied to a dataset to perform various transformations like filtering, deriving additional fields or deriving additional datasets. In a majority of cases, these datasets will contain abstract information that requires being mapped to visual properties. This mapping

is configured and performed using scales. Vega already contains a variety of scales to help with mapping abstract values to visual properties, and they can broadly be categorized into quantitative scales that map quantitative inputs to quantitative outputs, discrete scales that map discrete inputs to discrete outputs, and discretizing scales that map quantitative inputs to discrete outputs. For spatially encoded dimensions, these scales can be visualized as axes, whereas non-spatial encodings such as encodings as colors, sizes or shapes can be visualized as legends. At the core of every visualization lies the encoding of data as visual primitives, which is achieved in Vega via marks. Marks use scales to encode data fields as properties of their shapes. Based on D3's general update pattern, the encoding of marks can be separately controlled for newly created (entering) marks, existing and not exiting (updating) marks, and to-be-removed (exiting) marks. In addition to these basic visualization components, the Vega grammar contains further capabilities to describe interactions (via signals, triggers and event streams), cartographic projections, sequential or layered views (via mark groups), layouts and color schemes. To demonstrate how the various aspects of a Vega specification are defined, an example that represents a basic bar chart can be seen in Listing 3.3.

In template-based visualization libraries, interactions are typically defined by configuring premade interaction templates, which is easy but limiting, or by manually modifying the visualization in various callbacks, which is flexible but tedious and not serializable. The ability to describe custom interactions using a serializable, data-driven grammar is what sets Vega apart from other declarative visualization libraries because it offers a comparable flexibility to callback-driven interactions while still remaining fully serializable and declarative. The grammar to define interactions is based on the syntax of event-driven functional reactive programming [**EventDrivenFRP**], a high-level grammar that resembles mathematical equations to describe reactive systems. In Vega, the primitives to express interactions are called signals. Signals can be seen as dynamic variables that can change their values based on input events or other signals. These signals and the way their values change are declaratively defined, and they can be used as dynamic variables throughout most places in a Vega specification to change various characteristics of a visualization dynamically. An example of how the previously demonstrated bar chart specification can be extended with signals to show a tooltip when hovering bars can be seen in Listing 3.4.

Visualizations created with Vega closely follow their specifications and minimal assumptions are being made in the compilation process. This results in very verbose specifications because all configurations for all parts of the visualization need to be explicitly defined in them. It also means that specification authors have full control over the resulting graphics, which makes Vega a good base on which to build further libraries and tools. Many tools [**Voyager**; **Lyra**; **CompassQL**] have been built on top of Vega but the one worth mentioning most is Vega-Lite [**VegaLite**]. Vega-Lite is described as a "high-level grammar of interactive graphics", which summarizes its difference to Vega fairly well. Compared to Vega, Vega-Lite is a higher-level grammar that allows authors to write specifications for common visualizations in a much more concise form. Specifications written in Vega-Lite are compiled into Vega specifications and the compiler automatically derives default configurations for axes, legends and scales from a set of carefully designed rules. This makes Vega-Lite more convenient to use for quick authoring of visualizations because many details that need to be explicitly stated in a Vega specification can be omitted. Vega-Lite also offers the possibility to override derived defaults and, because Vega-Lite specifications are simply being compiled into Vega ones, it is a sensible choice to use Vega-Lite as a primary tool to describe visualizations and switch to Vega for more exotic cases that are not easily achievable in Vega-Lite. To properly compare the difference between Vega and Vega-Lite specifications, a Vega-Lite version of the Vega bar chart specification seen in Listing 3.3 can be seen in Listing 3.5.

3.2.3 Template-Based Visualization Libraries

Template-based visualization libraries work by providing templates for possible types of visualizations and allowing users to customize them. These types of visualization libraries are easier to use than D3 or Vega because they offer a concise form of configuration that does not require users to have detailed

```
{
  "$schema": "https://vega.github.io/schema/vega/v5.json",
  "width": 600,
  "height": 200,
  "padding": 5,
  "data": [
    {
      "name": "table",
      "values": [
        { "category": "A", "amount": 28 },
        { "category": "B", "amount": 55 },
        { "category": "C", "amount": 43 },
        { "category": "D", "amount": 91 },
        { "category": "E", "amount": 81 },
        { "category": "F", "amount": 53 },
        { "category": "G", "amount": 19 },
        { "category": "H", "amount": 87 }
      ]
    }
  ],
  "scales": [
    {
      "name": "xscale",
      "type": "band",
      "domain": { "data": "table", "field": "category" },
      "range": "width",
      "padding": 0.05,
      "round": true
    },
    {
      "name": "yscale",
      "domain": { "data": "table", "field": "amount" },
      "nice": true,
      "range": "height"
    }
  ],
  "axes": [
    { "orient": "bottom", "scale": "xscale" },
    { "orient": "left", "scale": "yscale" }
  ],
  "marks": [
    {
      "type": "rect",
      "from": { "data": "table" },
      "encode": {
        "enter": {
          "x": { "scale": "xscale", "field": "category" },
          "width": { "scale": "xscale", "band": 1 },
          "y": { "scale": "yscale", "field": "amount" },
          "y2": { "scale": "yscale", "value": 0 }
        },
        "update": {
          "fill": { "value": "steelblue" }
        }
      }
    }
  ]
}
```

Listing 3.3: The Vega specification of a static bar chart. Demonstrates the principle of data, scales, axes and marks. [TODO: Adapt source code to make it your own or cite source]

```
{
  "...": "...",
  "signals": [
    {
      "name": "tooltip",
      "value": {},
      "on": [
        { "events": "rect:mouseover", "update": "datum" },
        { "events": "rect:mouseout", "update": "{}" }
      ]
    }
  ],
  "marks": [
    { "...": "..." },
    {
      "type": "text",
      "encode": {
        "enter": {
          "align": { "value": "center" },
          "baseline": { "value": "bottom" },
          "fill": { "value": "#333" }
        },
        "update": {
          "x": { "scale": "xscale", "signal": "tooltip.category", "band": 0.5 },
          "y": { "scale": "yscale", "signal": "tooltip.amount", "offset": -2 },
          "text": { "signal": "tooltip.amount" },
          "fillOpacity": [{ "test": "datum === tooltip", "value": 0 }, { "value": 1
              }]
        }
      }
    }
  ]
}
```

Listing 3.4: The necessary changes to the bar chart specification in Listing 3.3 to add show a tooltip when hovering over bars. It demonstrates the basic functionality of signals in Vega. When the mouse hovers over a rect mark, the tooltip signal will receive the value of the rect's bound data record and when the mouse leaves the rect mark, the variable will be reset to an empty object. The tooltip signal is then used in the newly added text mark to define the position, text and visibility of it whenever an update occurs. [TODO: Adapt source code to make it your own or cite source]

```
{
  "$schema": "https://vega.github.io/schema/vega-lite/v5.json",
  "width": 600,
  "height": 200,
  "padding": 5,
  "data": {
    "values": [
      { "category": "A", "amount": 28 },
      { "category": "B", "amount": 55 },
      { "category": "C", "amount": 43 },
      { "category": "D", "amount": 91 },
      { "category": "E", "amount": 81 },
      { "category": "F", "amount": 53 },
      { "category": "G", "amount": 19 },
      { "category": "H", "amount": 87 }
    ]
  },
  "mark": "bar",
  "encoding": {
    "x": { "field": "category", "type": "ordinal" },
    "y": { "field": "amount", "type": "quantitative" },
    "tooltip": [{ "field": "amount" }]
  }
}
```

Listing 3.5: This is a Vega-Lite specification of the Vega bar chart specification seen in Listing 3.3 combined with Listing 3.4. [TODO: Adapt source code to make it your own or cite source]

knowledge over the underlying rendering technology or about complex, non-standardized domain specific languages. Even though these types of libraries are usually very flexible and suited to create a huge range of visualizations, at one point users may run into limitations that can not be worked around without writing custom source code that requires a deep understanding of the underlying library.

For this thesis, a total of 20 template-based JavaScript visualization libraries have been collected and compared by different factors such as their rendering technology, usage popularity, open-source popularity, license, and recent development activity. In terms of rendering technology, most libraries render visualizations as either SVG documents or canvas elements, though some implement a hybrid renderer that can be configured to render as either one of them. Usage popularity has been measured by the cumulative package downloads from the npm package manager over the last twelve months. This has been deemed one of the most relevant metrics for the comparison because it reflects actual user behavior and gives an indication for how widespread a library is used in practice. The 20 libraries that were found in the initial collection phase were filtered by their usage popularity and recent development activity to remove those that were not sufficiently used or maintained anymore. This filtering step yielded the following ten libraries: (1) ChartJS [**ChartJS**], (2) Highcharts [**Highcharts**], (3) ECharts [**ECharts**], (4) ApexCharts [**ApexCharts**], (5) PlotlyJS [**PlotlyJS**], (6) C3JS [**C3JS**], (7) Chartist [**Chartist**], (8) amCharts [**amCharts**], (9) billboardJS [**billboardJS**], and (10) D3FC [**D3FC**]. These libraries have been selected for deeper evaluation because they are heavily used and, for the most part, still actively maintained. The focus of this more detailed analysis is on summarizing these libraries based on high-level features and responsive capabilities.

Out of those ten libraries, eight are completely free to use without restrictions, amCharts has a free

license for users that are comfortable with an attribution logo on their visualizations, and Highcharts offers a free license option for non-profit, educational and personal applications. Nine libraries implement a renderer that is based on SVG, two of which (ECharts, D3FC) offer alternative rendering to canvas elements for high-performance scenarios, and only ChartJS purely targets canvas-based rendering. Eight libraries are very actively maintained with most of them showing development activity in the last month. Only C3.js and Chartist seem to not be actively maintained anymore, but they nonetheless have been included in the deeper evaluation because of their historic and thematic relevance and because they are still widely used.

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

All evaluated libraries, except D3FC, expose declarative interfaces in the form of nested configuration objects that are used to specify the characteristics of individual visualizations. Apart from Chartist, all those libraries feature generic high-level functions that create charts from declarative configuration objects that allow the specification of different forms of visualization for different data dimensions. This type of interface is demonstrated by the Highcharts code seen in Listing 3.6. These generic chart creation functions seem to be correlated with the ability of dynamically changing the type of visualization. In Chartist, for instance, which provides separate chart creation functions for each type of chart, it is not possible to alter the type of chart after it has been created. Another limitation that may originate in this interface partitioning via chart types is that mixed charts that combine multiple forms of visualization in one composite visualization can not be expressed. The only library in the deeper evaluation that does not provide a high-level declarative configuration API is D3FC. The design philosophy of D3FC is based on the idea of "unboxing" D3 because even though many visualization libraries are implemented on top of D3, it is usually hidden behind public APIs that are easier to work with but which don't provide the full flexibility that D3 does. D3FC exposes a component-based interface that closely follows design patterns frequently encountered when working with D3. These components form higher-level building blocks on which advanced visualizations can be built. They are also highly configurable and in those cases where the options for configuration are not sufficient, a decorator pattern is used to allow users to hook into the underlying D3 functionality and inject custom code into the various stages of the general update pattern at the core of D3. Code that demonstrates the usage of D3FC can be seen in Listing 3.7. amCharts is the only evaluated library that exposes a hybrid API with the possibility of configuring visualizations using both declarative configuration objects and manually composing higher-level visualizations from lower-level components such as axes and series. The component-based interface is still rather declarative, with most options being configurable by modifying certain properties on the components, but modifying only the properties that require changing is much less taxing in terms of performance than having to process a full configuration object and figure out the necessary changes from it. In addition to these performance benefits, the components provide additional functions to perform operations that would not be available using a purely declarative API.

When comparing the evaluated libraries by their responsive configurability, most libraries offer similar

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

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

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

```
const data = [
  { category: 'A', value: 107 },
  { category: 'B', value: 31 },
  { category: 'C', value: 635 },
  { category: 'D', value: 203 },
  { category: 'E', value: 50 },
];

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

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

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

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

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

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

Chapter 4

Responsive Information Visualization

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

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

4.1 Responsive Visualization Patterns

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

RespVisSurvey have conducted a survey under close supervision by Keith Andrews [**RespVis**] and they identified nine common patterns that reoccurred in several solutions. Slightly reworded, these patterns are: (1) rotate axis labels, (2) remove axis ticks, (3) modify strings, (4) transpose chart, (5) reposition

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

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

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

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

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

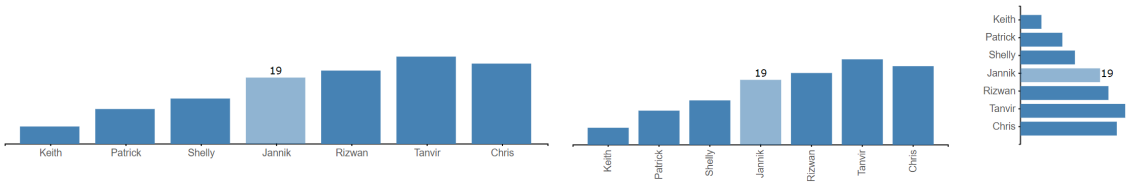
Category	Description
Recompose	Actions that affect the existence of targets. Includes remove, add, replace and aggregate actions.
Rescale	Actions that affect the size of targets. Includes reduce width, simplify labels and elaborate labels actions.
Transpose	Actions that affect the orientation of targets.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

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

4.2 Responsive Visualization Examples

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

The first types of charts that shall be discussed are a bar charts. They are among the most often encountered types of charts, accounting for 135 (= 36%) of the 378 responsive charts studied by Kim et al. 2021. Bar charts are usually used to visualize two-dimensional data, with one dimension being categorical and the other one being quantitative. Two additional variants of bar charts exist to visualize categorical datasets with multiple subdimensions: grouped bar charts [**GroupedBar**], to compare subdimensions with each other, and stacked bar charts [**StackedBar**], to compare part-to-whole relationships of the subdimensions. Even though responsive design of visualizations is slowly becoming more common,



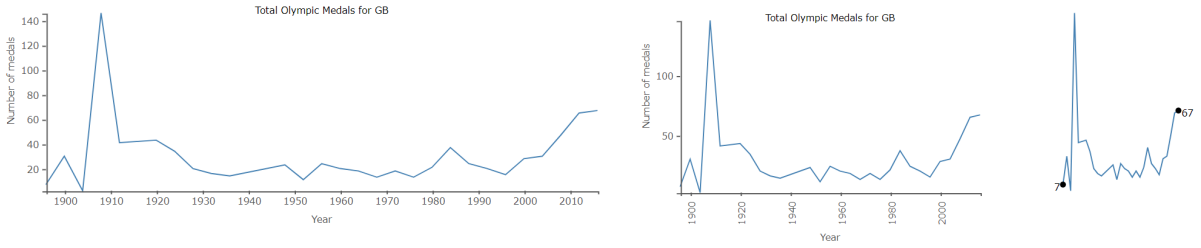
(a) 70rem **(b)** 50rem **(c)** 30rem

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

most charts found in today's web articles are still being created as static images [**HBar**; **VBar**; **HVBar**; **MapBarLine**]. A good example of a responsive bar chart has been created by **RespVis** and can be seen in Figure 4.1. Bar charts are freely scalable by adjusting the width of individual bars [**RespHBar**; **RespHBarHLine**; **RespHBars**], so they all can fit into their allocated space. When reducing the width of any type of chart past a certain point, the tick labels of the horizontal axis may start overlapping each other. This is why the reducing width pattern usually occurs together with the recompose axis ticks and simplify/elaborate axis labels patterns [**RespHBars**; **RespHBarHLine**; **RespVBar**]. Another effective pattern for solving overlapping of tick labels is to rotate them by up to 90 degrees to make them take up less horizontal space [**RespVis**]. If there is too much data to fit into the available width, the chart can be transposed and grown to as much height as necessary [**RespVis**]. Doing this is advisable over simply extending the width of the chart past the viewport because vertical scrolling is easier to achieve than horizontal scrolling. When reducing the size of charts that contain annotations, similar patterns than those targeting tick labels can be applied to avoid the annotations from overlapping. Annotations can simply be removed [**RespHStackedBar**; **RespHLineHStackedBar**], or they can be simplified and relocated [**RespVBar**].

The second most frequent types of responsive charts according to the responsive visualization gallery created by Kim et al. 2021 are line charts, which amount to 98 (= 26%) out of the 378 responsive visualizations in the gallery. Line charts are used to show trends in two-dimensional datasets by plotting them as points that are then connected by lines. They can be extended to compare trends in multiple dimensions with each other by drawing additional points and lines for every additional dimension that shall be compared. Many line charts on the web are still published in non-responsive forms [**HLine**; **HLine2**], though some web authors already took the extra effort to make their charts responsive. The minimum that can be done to make a line chart responsive is to reduce their width [**RespRadialScatterHLine**] by shrinking the horizontal distance between neighboring points. This usually occurs together with the recomposition and simplification of horizontal ticks. If the chart contains annotations, it may also be necessary to recompose, relocate, and simplify them as well [**RespHLines**; **RespHLine**; **RespHBarHLine**; **RespHLineHStackedBar**]. A good demonstration of which responsive patterns can be applied to make a line chart responsive is shown in the responsive line chart created by **RespVis** that can be seen in Figure 4.2. In addition to the recomposition of ticks, tick labels are rotated to reduce their required horizontal space. For exceptionally limited space, it can make sense to remove the axes of a line chart entirely and turn it into a sparkline. However, it should be noted that by doing this, the consumer of the visualization looses a lot of information about the type and scale of the chart's dimensions. This technique should therefore only be applied in cases where no other pattern is applicable or if the trend in the data is the most important message to convey. It is rare to encounter transposed versions of line charts, though the transposing could benefit heavily annotated line charts [**VLine**]. Applying a transpose pattern would allow the chart to take up as much vertical space as necessary to neatly fit all the annotations without requiring the consumer to scroll horizontally.

Scatterplots are also among the rather frequently encountered types of responsive charts, amounting

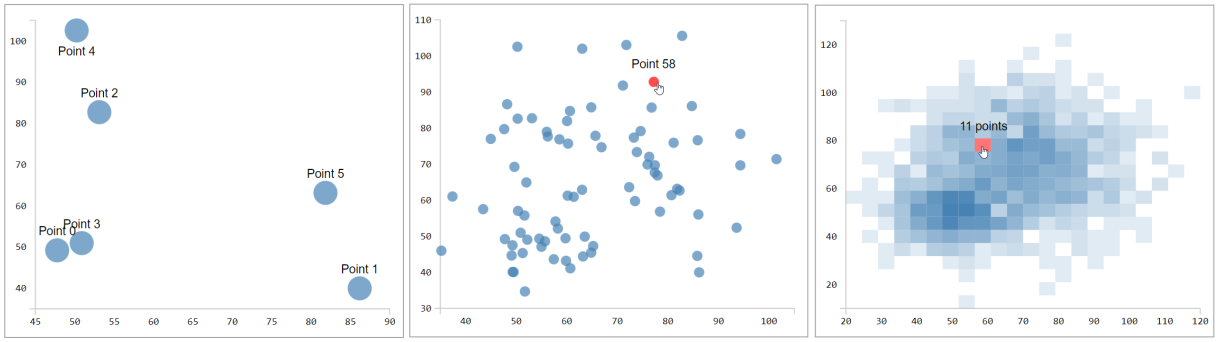


(a) 65rem **(b)** 40rem **(c)** 20rem

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

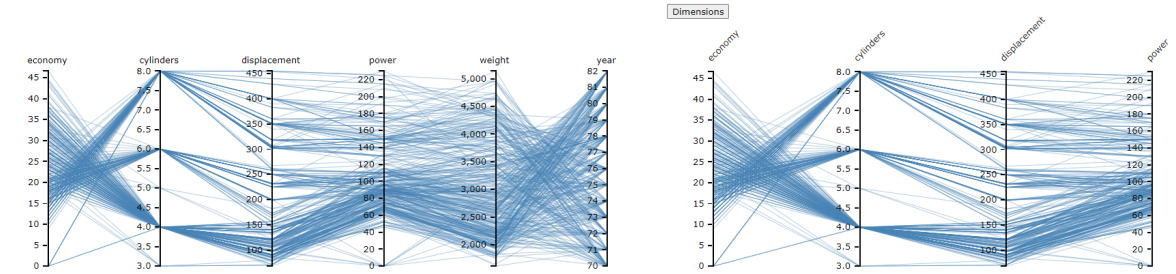
to 26 (= 7%) of the 378 responsive charts contained in the gallery by Kim et al. 2021. A scatterplot is a visualization that represents two-dimensional data as points in a Cartesian coordinate system. There are plenty of examples of scatterplots that are merely being published as static images [**Scatter**; **Scatter2**], even though responsive versions can also already be observed occasionally. The first step to making scatterplots responsive is to reduce their width to fit them into the space available for them. As for other types of charts, care must be taken to avoid overlapping of labels and annotations by applying recomposition, relocation and simplification patterns [**RespScatter**; **RespScatter2**]. To counteract the increased cramming of points when reducing the size of their container, various interaction features are usually implemented in scatterplots that help consumers in making sense of the represented data. The most useful interaction features in these charts are elaborative zooming interactions and the explorative panning interactions. In addition to zooming and panning, **RespVis** employs additional methods that reduce overlapping of individual points, such as fisheye distortion, Cartesian distortion or temporary displacements of points. An interesting technique for responsive visualizations that bases responsive configurations on a visualization's density rather than on its size has been introduced by **NickRabinowitzRDV**. The benefit of this approach is that charts will also adapt to changing amounts of data and reconfigure their appearance accordingly. The patterns applied in the responsive scatterplot by **NickRabinowitzRDV** that can be seen in Figure 4.3 are the recomposition of annotations to only show them for selected data records, and the switching of the encoding from a scatterplot to a heatmap for high densities. A good number of other techniques, such as for example the recomposition of data records, are also applicable to improve the responsiveness of scatterplots, but no examples for such patterns could be found. If the data that shall be encoded is inherently cyclic, a radial scatterplot, which is a polar coordinate system variant of a scatterplot, can be applied to better visualize this cyclic nature of the data [**RespRadialScatterHLine**].

Even though parallel coordinates charts are rarely encountered in non-technical contexts, they are very popular when it comes to visualizing multidimensional data in visual analytics systems [**HighD**]. In these kinds of charts, multiple dimensions are rendered as parallel axes that are connected via paths. Each path represents an individual data record and its values in the corresponding dimensions. The axes of a parallel coordinates chart are typically laid out horizontally, meaning that the chart can be scaled down by reducing the distance between the individual axes. It might also be beneficial to apply previously mentioned axis-related patterns, such as rotating labels and recomposing ticks. Another technique that can be applied to make parallel coordinates charts more responsive is the successive hiding of dimensions based on their priorities. When automatically hiding dimensions, it is necessary to apply compensation patterns that give users additional controls that allow configuration of the displayed dimensions to override the system's hiding behavior. If reducing the chart's complexity is not a desirable approach, it can again be recommended to transpose the chart and expand it to whatever height necessary rather than cram too much information into the limited width available. An example of a responsive parallel coordinate chart incorporating some of these patterns can be seen in Figure 4.4.

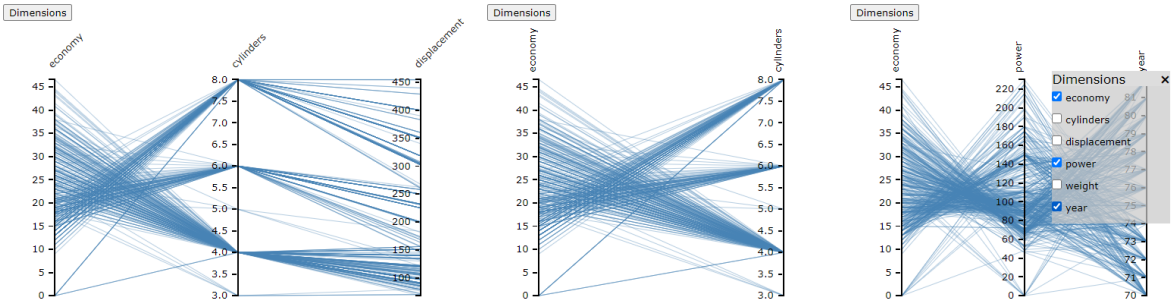


(a) 0.00005 points per pixel **(b)** 0.0007 points per pixel **(c)** 0.017 points per pixel

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



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



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

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

Chapter 5

Software Architecture

[TODO: Checkout these resources: bocoup.com/blog/reusability-with-d3, bocoup.com/blog/introducing-d3-chart]

[TODO: Project setup? Make an own chapter for how the project is set up?]

[TODO: Add software architecture diagram]

[TODO: Describe relationship to D3]

[TODO: Describe storing data on elements]

[TODO: Describe using DOM events for callbacks]

[TODO: Describe components]

5.1 Primitives

5.1.1 Text

5.1.2 Rectangle

5.1.3 Circle

5.2 Series

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

5.2.1 Bar Series

5.2.2 Grouped Bar Series

5.2.3 Stacked Bar Series

5.2.4 Point Series

5.2.5 Line Series

5.3 Charts

5.3.1 Bar Chart

5.3.2 Grouped Bar Chart

5.3.3 Stacked Bar Chart

5.3.4 Point Chart

5.3.5 Line Chart

5.4 Chart Windows

5.4.1 Bar Chart Window

5.4.2 Grouped Bar Chart Window

5.4.3 Stacked Bar Chart Window

5.4.4 Point Chart Window

5.4.5 Line Chart Window

5.5 Components

5.5.1 Lifecycle

events

updating on data change

updating on bounds change

5.6 Layouter

Chapter 6

Layouter

6.1 CSS Layouting

Chapter 7

Another Chapter; Maybe about Components?

Chapter 8

Selected Details of the Implementation

8.1 D3 Select Function Data Modification

8.2 Save as SVG

Chapter 9

Outlook and Future Work

9.1 Outlook

9.2 Ideas for Future Work

9.2.1 Relative Positioning of Series Items

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

9.2.2 Container Queries

Chapter 10

Concluding Remarks

Appendix A

User Guide

Appendix B

Developer Guide

Bibliography

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

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

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

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

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

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

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

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

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

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

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

Kim, Hyeok, Dominik Moritz, and Jessica Hullman [2021]. *Design Patterns and Trade-Offs in Responsive Visualization for Communication*. Computer Graphics Forum (2021). ISSN 1467-8659. doi:10.1111/cgf.14321 (cited on pages 35–39).

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

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

Meyer, Eric A. [2016]. *Grid Layout in CSS: Interface Layout for the Web*. 1st Edition. O'Reilly Media, 18 Apr 2016. ISBN 9781491930212 (cited on page 6).

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

Microsoft [2020]. *Microsoft 365 apps say farewell to Internet Explorer 11 and Windows 10 sunsets Microsoft Edge Legacy*. 17 Aug 2020. https://techcommunity.microsoft.com/t5/microsoft-365-blog/microsoft-365-apps-say-farewell-to-internet-explorer-11-and/ba-p/1591666 (cited on page 5).

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

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

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

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

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

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

W3C [1998]. *Document Object Model Specification*. 16 Apr 1998. https://www.w3.org/TR/1998/WD-DOM-19980416/ (cited on page 8).

W3C [2009]. *CSS Flexible Box Layout Module*. 23 Jul 2009. https://www.w3.org/TR/2009/WD-css3-flexbox-20090723 (cited on pages 4–5).

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

W3C [2011b]. *Grid Layout*. 07 Apr 2011. https://www.w3.org/TR/2011/WD-css3-grid-layout-20110407/ (cited on page 6).

W3C [2018]. *CSS Flexible Box Layout Module Level 1*. 19 Nov 2018. https://www.w3.org/TR/css-flexbox-1 (cited on pages 5–6).

W3C [2020a]. *CSS Grid Layout Module Level 1*. 18 Dec 2020. https://www.w3.org/TR/css-grid-1/ (cited on page 6).

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

W3C [2021]. *DOM Living Standard*. 02 Aug 2021. https://dom.spec.whatwg.org/ (cited on page 8).

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

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