d3.js and its potential in data visualization

Creating a diagram showcase using ukrainian refugee data

Luis Rothenhäusler

20202459

Technische Hochschule Brandenburg
University of Applied Sciences
Fachbereich Informatik und Medien

Bachelorarbeit

Fachbereich Informatik
und Medien
Technische Hochschule Brandenburg

Betreuer: Prof. Julia Schnitzer
2. Betreuer: Prof. Alexander Peterhänsel

Brandenburg, den 28.08.2022
Bearbeitungszeit: 07.07.2022 - 01.09.2022

Brandenburg, den 28.08.2022

Ich, Luis Rothenhäusler, Student im Studiengang Applied Computer Science der Technischen Hochschule Brandenburg, versichere an Eides statt, dass die vorliegende Abschlussarbeit selbstständig verfasst und nicht mit anderen als den angegebenen Hilfsmitteln erstellt wurde. Sie wurde in dieser oder ähnlicher Form noch keiner Prüfungskommission vorgelegt.

Luis Rothenhäusler

Abstract - German

Die moderne Welt produziert täglich riesige Mengen an Daten. Diese zu verstehen und auszuwerten kann in allen Bereichen Grundlage für Entscheidungen und Entwicklung sein. Da es oft schwierig ist große Datensätze zu überblicken und Schlussfolgerungen daraus zu ziehen, wird Datenvisualisierung genutzt Datensätze begreifbar und überschaubar zu machen. Die Ergebnisse von Datenvisualisierung lassen sich überall im Alltag finden. Zum Beispiel als Diagramme in Zeitungen und Nachrichten, sowie die Darstellung von Arbeitsbläufen und Zusanmenhängen unterschiedlicher Faktoren.

Es gibt eine vielzahl an Werkzeugen die zur Datenvisualisierung genutzt werden können. Während manche dieser Werkzeuge, wie Excel und SPSS, eine Grafische Oberfläche bieten, nutzen andere, wie R und Matplotlib, einen code basierten Ansatz. Die Entscheidung welches der Werkzeuge für die eigenen Anforderungen am besten geeignet ist, ist jedoch nicht immer einfach. Besonders, da die Möglichkeiten, so wie die Vor- und Nachteile der eizelnen Werkzeuge für jemand ohne vorhandene Erfahrung nicht einfach zu überblicken sind. Daher gibt diese Arbeit einen Einblick, sowie eine Einschätzung eines dieser code basierten Werkzeuge, der d3js JavaScript library. Es wird nicht nur das Potential, sondern auch die Vor- und Nachteile, sowie eine Einschätzung wann der Einsatz dieses Werkzeugen sinnvoll ist, evaluiert.

Da d3js in reinem JavaScript implementiert ist, lässt es sich Problemlos mit anderen Frameworks kombinieren. Der Fokus von d3js liegt auf der schnellen und einfachen Manipulation von Elementen im Document Object Model. d3js ist jedoch kein Werkzeug mit dem in wenigen Zeilen Quellcode ganze Diagramme erstellt werden können. Vielmehr müssen sowohl die einzelnen Elemente eines Diagrammes, wie auch deren Position und Aussehen händisch definiert werden. Das ermöglicht ein hächstgrad an Kontrolle über das Aussehen und Verhalten der Diagramme. Es ist jedoch auch Zeitintesiv und erlaubt für die Implementation hüchst unübersichtilicher Diagramme. Gepaart mit der hohen initialen Lernkurve, kann die Nutzung von d3js nur in speziellen Fällen, welche die volle Kontrolle über Aussehen und Verhalten benötigen, empfohlen werden.

Contents

1. Introduction

The postmodern world produces huge amounts of data every second. Analyzing this data can lead to better-informed decision-making in every sector. Yet the wast amounts of gathered data is often hard to comprehend with the human mind. Data visualization is about finding ways to represent this data in visually appealing and easily comprehendible ways [1]. Doing this quickly, ideally instant, and being always up to date can be crucial. While it is possible to create data visualizations manually, it is common to use computer tools to help in their creation.

There are many tools available to help with the creation of diagrams for data visualization. Some of these tools have a graphical-user-interface, like Excel [2] or SPSS[3], while others are code based, like R [4] or the Matplotlib [5] library for Python. As the requirements for a data visualization project can vary, it is often not easy to decide which tool best suits ones needs. Especially when one is unfamiliar with the potential of the tools available and one has no prior experience. Yet knowing when to use which tool can be greatly beneficial for all parties involved. Therefore this thesis will be a deep dive into the broad possibilities of one of these code based tools, the 'd3.js'(D3) library for JavaScript. Knowing its potential, as well as the strengths and weaknesses of the tool can be greatly beneficial when deciding if it is the right tool for the job. Whilst there is a lot of information and examples on how to use D3, the available information makes use of a variety of code styles and different versions of D3. This makes it hard to properly evaluate the possibilities of D3 as a data visualization tool, without using it.

To evaluate D3 and its possibilities there are three main questions that will be investigated in this thesis. What is the potential of D3 in data visualization? What are the advantages and disadvantages of using D3? When should D3 be used? To be able to evaluate these questions and get a better understanding of D3, a showcase of several different diagrams is created and evaluated throughout this thesis. This showcase is created using and visualizing refugee data of the currently ongoing Ukraine conflict. A live version of the showcase can be found at "`https://styxoo.github.io/`".

All the theoretical background as well as the core concepts of D3, which are necessary to understand and follow along with this thesis, are described in chapter 2. Afterwards, in chapter 3, the implementation of each of the diagrams, as well as the showcase are described in detail. In chapter 4, the resulting diagrams are discussed and the initial questions about D3s potential and limitations are evaluated. Finally, a conclusion is drawn in chapter 5.

2. Basics

In this chapter, all concepts, technologies and required backgrounds for understanding this thesis, as well as the showcase implementation, are explained. First, data and data types are described. Second, diagrams and how they are structured are described. Last, D3 as the tool to create diagrams is described and its core concepts explained.

2.1 Data

Since ancient times, humans have recorded data. Recording the ins and outs of available resources and other administrative record-keeping were one of the driving factors behind the conceptualization of writing [6]. With the introduction of computers the amounts of gathered data have grown drastically. Nowadays vast amounts of data are gathered across all aspects of life. According to Statista, the total amount of data created, consumed and stored by 2020 was already at 64.2 zettabytes and is projected to reach about 180 zettabytes by 2025 [7].

The vast amounts of data gathered in databases are often hard to comprehend and evaluate with the human mind. They are also unwieldy to present in the often limited space of articles, dashboards or other informative purposes. Therefore data visualization (Figure 2.1) is used to turn these data sets, collections of data points, into diagrams. These diagrams can easily be shown in more limited spaces, as well as allow for a quick general understanding and overview of the provided data.

Data is commonly preprocessed before turning it into diagrams. Depending on the data set and the desired result, this can mean different things. One might want to remove excessive information from the data set, which is not necessary for the representation. On the other hand, additional data can be added by evaluating the existing data points. These could for example be the median of values or grouping of certain value ranges [8]. It is important to note, that this preprocessing can happen with specific intentions in

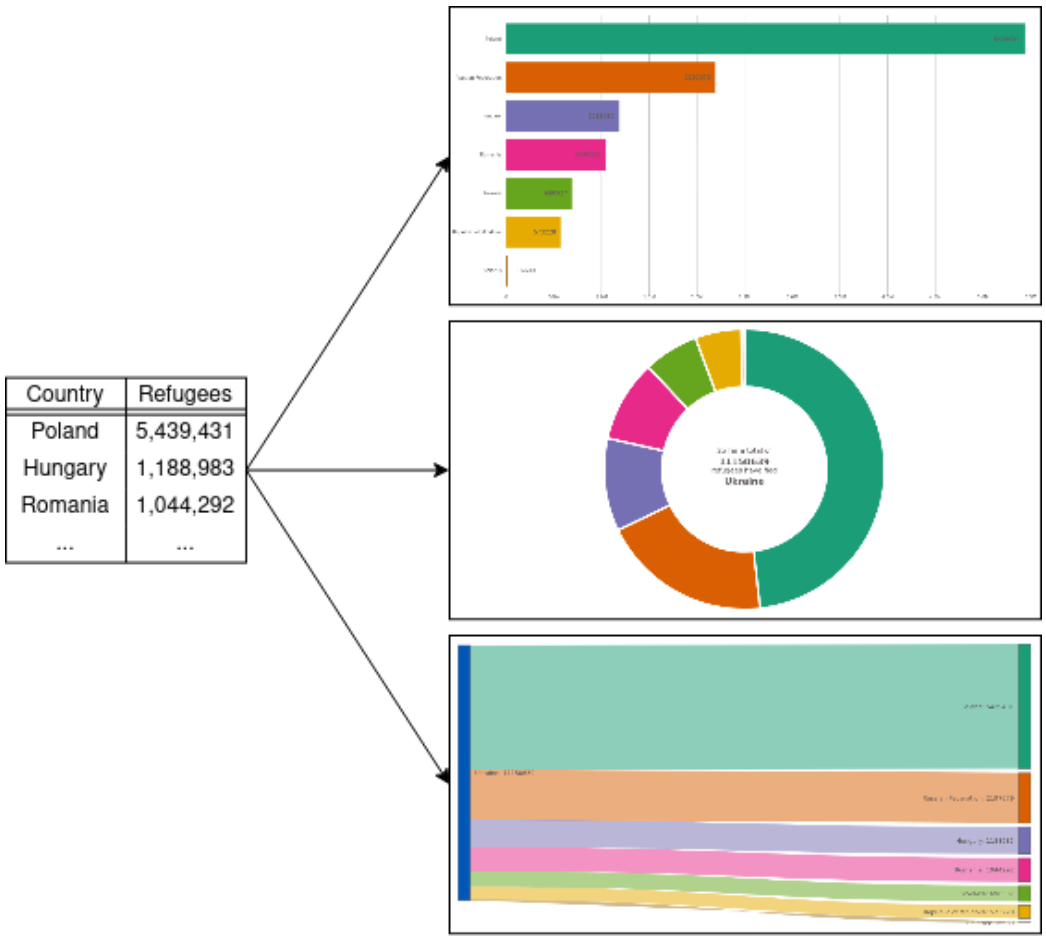

Figure 2.1: Data visualization describes the process of turning raw data into visual representations. There can be a multitude or possible representations for a data set.

mind. While it is only supposed to make the representations easier and more concrete, it can be abused to make data align with the desired results or to create a certain emphasis. This thesis is not too concerned with this, as the possibilities of D3 are independent of the validity and completeness of the presented data.

Even though data comes from a huge variety of sources and can express a plethora of things, there are only four different types of data [9]. They are split into two categories: categorical and numerical data. Each category has two subtypes. In the following each of the types of data will be explained.

2.1.1 Categorical

Categorical or qualitative data is information collected in groups. It is often of descriptive nature. Whilst the values can be represented in numbers, like grades, they do not allow for arithmetic operations. Yet as it is possible to count the data points. Therefore it is possible to find the mode. The mode is the most frequently occurring data.

The two sub-types of categorical data are nominal and ordinal data. Each of them is described below.

Nominal data is mostly descriptive in nature. Values have no relation to each other and have no inherited order. Examples are the 'Country of origin', 'Color of paint' or 'Brand of car'.

Ordinal data is also descriptive, yet the data does have a internal order. For example different dates each describe a day, but one day also comes after another. Grades also have an internal order, as one grade is better then another. Whilst ordinal data has an ordering, the order is not necessarily equidistant. Due to its internal order, it is also possible to find the median. The median is the value where half the values in the data set are higher and the other half of the values are lower.

2.1.2 Numeric

Numeric or quantitative data is all data expressed in numbers, where numbers do not represent categories. It allows for arithmetical operations and can be split into discrete and continuous data.

Discrete data can only take certain defined values. This usually means whole numbers to represent things that can not be split up further. Discrete

data is usually countable. Examples are the 'Number of Refugees' or 'Tickets sold'.

Continuous data can be measured. It can have any real number as value. Therefore fractions are possible as well. For example when measuring the temperature, or the length or weight of an object.

2.2 Diagrams

One constantly comes across diagrams as the results of data visualization in everyday life. They can be commonly found across all kinds of reports, information campaigns or as part of user-interfaces in machinery or control systems. Yet the selection of which diagram should be used to visualize which data set is not trivial. Mostly there are several possible diagram choices for any given data. Furthermore there are a plethora of diagrams already in use and anyone can create totally new diagrams to suit their needs. Yet the vast majority of use cases can be accomplished by one of the more commonly known diagram types, like bar and column charts, pie and donut charts, line and area charts, scatter plots and heat maps. Due to their popularity, tools like Excel provide support for these diagrams out-of-the-box [10]. More specialized diagrams might use combinations or variations of the aforementioned diagram types.

Whilst there are countless types of diagrams, all diagrams use a combination of marks and channels to present data. Marks are used for entries in the diagram. The three possible marks are points, lines and areas. Channels describe the way specific marks encode data. While there is no definitive list of channels, the most commonly used channels are position, size, color and texture. The position in 2D can be split into the x and y positions. The color can be split into hue and luminescence. Each mark should use at least one channel to encode data. Otherwise it does not convey any information. For example in fig. 2.2 lines are being used as marks for each of the seven entries. It might seem like areas are used. Yet the thickness of the line, and therefore the bar, only serves visual understanding and holds no relevant information. The lines use three channels to encode data. The y-position is used to represent the categorical data of which country. The hue of the bar encodes the same data. This is a bit redundant, as the country is already encoded. Yet the hue makes it easy to follow along when data is changing and bars are shifting positions. The size, in this case length, of the bar encodes the discrete data of how many refugees have crossed into the country. In fig. 2.3

areas are used as marks. Just like in the previous example the hue encodes the country and the size encodes the refugee count.

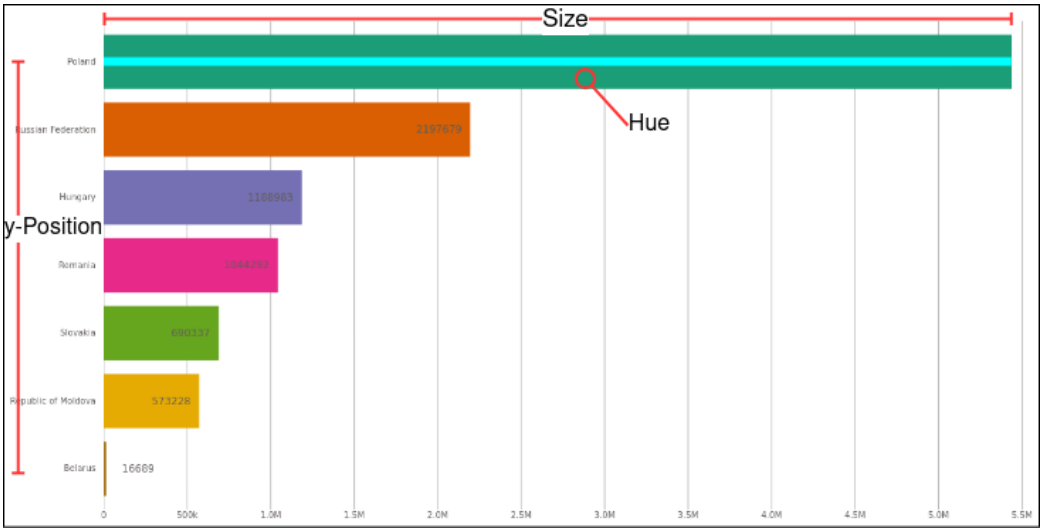

Figure 2.2: This bar chart uses lines as marks. Each bar is a single line mark. The first line mark is highlighted in cyan. The thickness has no relevance other than making the line visible. Each mark uses three channels to encode data. They are highlighted in red. The y-position and the hue are used to encode the country. The size of the line, aka the length, corresponds to the number of refugees.

All marks can be used with all channels. But not all data types should be represented by all channels. For example nominal data should not be encoded using the size channel. The different sizes would lead to a perceived order, which does not exist in nominal data. As the channels all differ in their appearance they are also not equally good in adequately representing the data types. Therefore it is important to consider which channels are chosen to represent the given data types. According to a study by Jock Mackinlay from 1986, the position channels can always be considered the strongest channels, no matter which marks are combined with them [11]. Therefore the selection of marks and channels should be considered carefully. If chosen poorly it can lead to undermine the purpose of the diagram of easily presenting data to a viewer.

Another factor which plays a role here, is the data-ink ratio described by Edward Tufte [12]. It describes the ratio of ink necessary for representing data over the total ink necessary for the diagram. The idea is to draw only what is necessary for showing the data, as this is the main purpose of a diagram. Whilst a lot of diagrams are digital nowadays and therefore do not

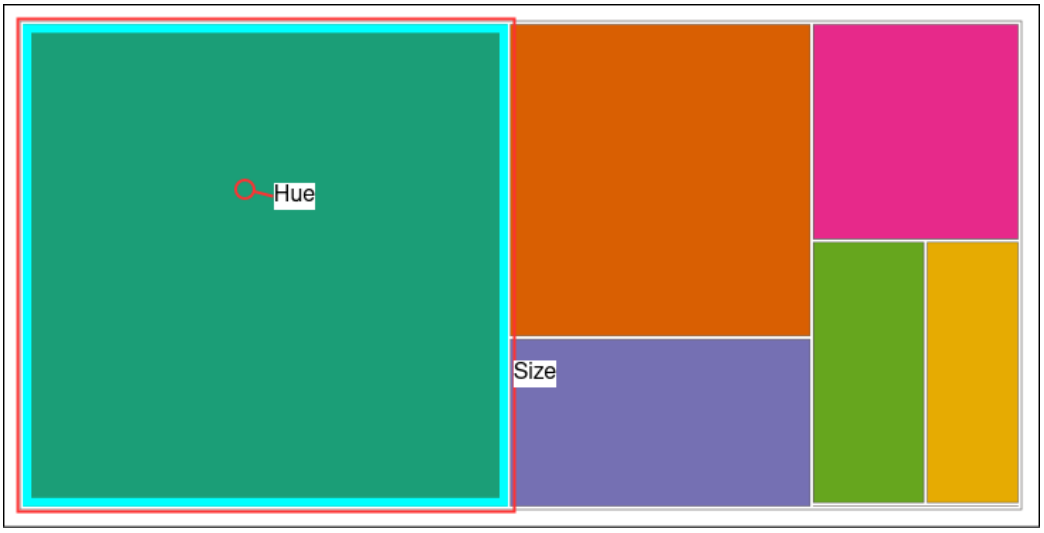

Figure 2.3: This tree map uses areas as marks. A representative mark has been highlighted in cyan. The areas use the size and hue channels to encode data. The size corresponds to the number of refugees, while the hue corresponds to the country.

require ink, diagrams should still try to get as close as possible to a data-ink ratio of one. The lower the data-ink ratio drops, the harder it gets for a viewer to see and comprehend the relevant data.

As some viewers might not be able to perceive the whole range of colors, choosing a color scale should also be carefully considered. Besides using colors which retain a high contrast even with color blindness, they should also be perceptually uniform. This means having similar hues for values close together, and more distinctively different hues for values further apart, but also having a consistent rate of change in the hue. This is especially important when trying to encode quantitative data using the hue channel.

While data can already be skewed during collection and preprocessing, diagrams can also skew perception. Tufte introduced the lie factor for evaluating how accurately data is shown [13]. It is defined as the effect size in the diagram over the effect size in the data. Most sources of skewed representations of data can be prevented by using zero baselines, equidistant axes, accurate scaling when using areas and value adjustments for monetary values to contradict inflation influences.

2.3 D3.js

While there are many ways to turn data into diagrams, this thesis makes use of D3 to achieve this. Therefore this chapter introduces D3 by elaborating what it is and how it works.

"D3.js is a JavaScript library for manipulating documents based on data. D3 helps you bring data to life using HTML, SVG, and CSS."[14]. The name D3 is short for data-driven documents. The D3 library was originally created by Mike Bostock and is published under the BSD-3-Clause open-source license. It is about 350kB in size. As it is fully implemented in JavaScript, it does not require a specific framework and can therefore be easily integrated into all kinds of web-based projects. Whilst D3 is not limited to using SVG, the visualization created using D3 mostly rely on SVG elements for their implementation.

D3 is not a high-level API for creating out-of-the-box visualizations. Instead, "[it] allows you to bind arbitrary data to a Document Object Model, and then apply data-driven transformations to the document."[14], therefore making Document-Object-Model(DOM) manipulation easier and less tedious. The DOM represents the structure of an HTML in memory and offers scripts the possibility of accessing and modifying the represented HTML. D3 also provides helper functions like scales, to decrease the amount of mathematical equations needed to convert from the data range to the necessary coordinates in the desired visualization.

There are three main concepts that make up the core of D3. Selections, data joins and the general update pattern. All three concepts are working closely together. Whilst selections can be used without data joins and the general update pattern, these two aspects both rely on selections. Data joins can also be used without explicitly using the general update pattern. Usually all three of these concepts are used consecutively. First, a selection is created. This selection is then used to create a data join. Finally, the behaviors of the general update pattern are defined for this data join. In the following all three of the core concepts of D3, as well as scales and D3's plugins are explained.

2.3.1 Selections

A selection contains references to one or more DOM elements. These references are organized in groups. There are two functions in D3 to create a new selection: `d3.select("selector")` and `d3.selectAll("selector")`. Both functions require a selector for identifying the appropriate elements.

The selectors are defined in the "W3C Selectors API" [15] and function like CSS selectors. Whilst `select` only selects a single element, the first element matching the selector, `selectAll` selects all elements which match the selector. It is important to note, that `select` also propagates the existing information of this node, whilst `selectAll` does not. Selections can also be extended or shrunken by adding or removing groups, or by combining multiple selections. `select` and `selectAll` can also be called on on elements of an already existing selections. The selector will then assume the existing element as root for its selection process.

It is possible to directly access DOM elements through the selections. The respective DOM elements are linked in the groups which make up the selection. But usually this is not required, as there are predefined functions for easily modifying properties for all elements referenced in a selection. This includes the modification of attributes and styles of DOM-elements, as well as event handling.

A selection is required before a data join can be made. How this is achieved is described in the following section.

2.3.2 Data Joins

Data joins are the second key feature of D3. They bind a specific data point to a specific DOM element. To create a data join, one has to first create a selection of elements. These are the elements one wants to match to specific data points. The data join is then created by calling the `.data(dataSet)` function on the selection. It takes a data set, an array of objects where each object represents a single data point, as parameter. This will bind the data points to the applicable elements in the selection. This is achieved by using an identifier function. This identifier function is called for each data point in the provided data set. The default identifier function returns the index of the current data point in the data set. When one wants to create diagrams which can respond to data changes over time, this is not a reliable identification. When data points are removed or added in arbitrary locations, the index will not match the elements it previously did. Therefore a custom identifier function can be specified, as seen in figure 2.4. This can be passed as the optional second parameter of the data function, will be called for each data point and has to return some value which will be used as the ID. This ID is saved on the element the data point was matched to. Every time the data set is updated, the data join and its underlying selection need to be called again.

As seen in fig 2.5, it can be that the number of data points does not match up with the number of elements to represent them. When there is no

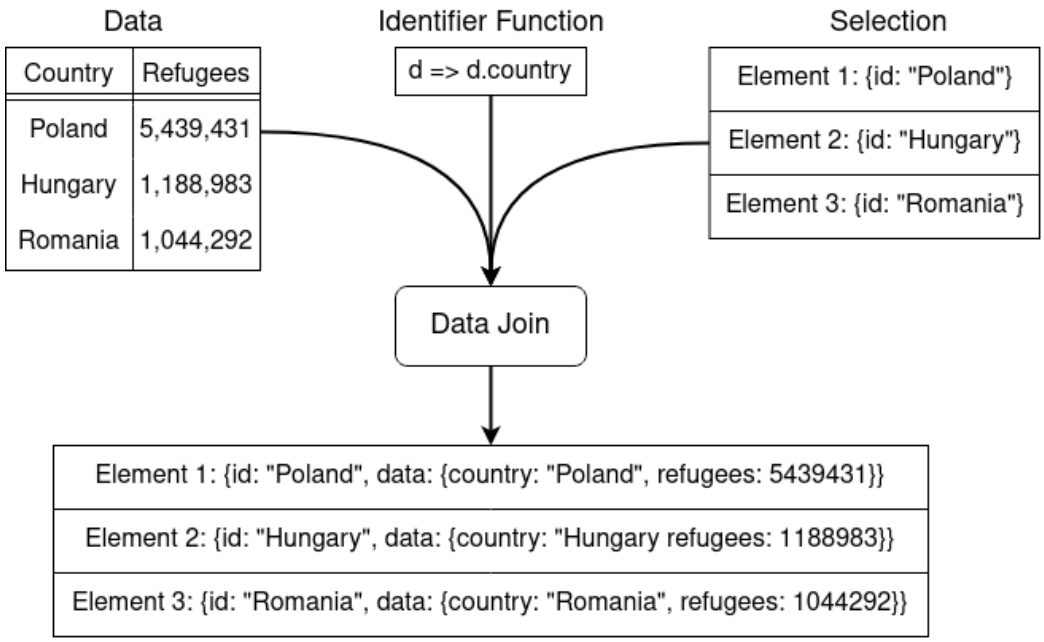

Figure 2.4: A representation of how data joins are created. A selection, consisting of DOM-elements, as well as the data need to be present first. The data join then combines the two. The identifier function is needed if the diagram is supposed to be able to update and can be specified when creating the data join. As a result the data is matched to the applicable DOM-elements using the identifier function.

element matched to a certain data point, D3 will create an empty placeholder node for this data point. What happens to the placeholders is defined in the general update pattern.

2.3.3 General Update Pattern

The general update pattern is another core concept of D3. Every time a data join is created or updated, it can be made use of. The general update pattern differentiates between three different cases: enter, update and exit. For each of these cases a sub-selection is created by the data join. For each of these three sub-selections the behavior can be defined. As sub-selections are just selections themselves, they will be referred to as such henceforth. The first selection is the enter selection. It corresponds to the magenta elements in fig 2.5. All the placeholders created by the data join for data points without a matching element are in here. In the behavior for the enter selection,

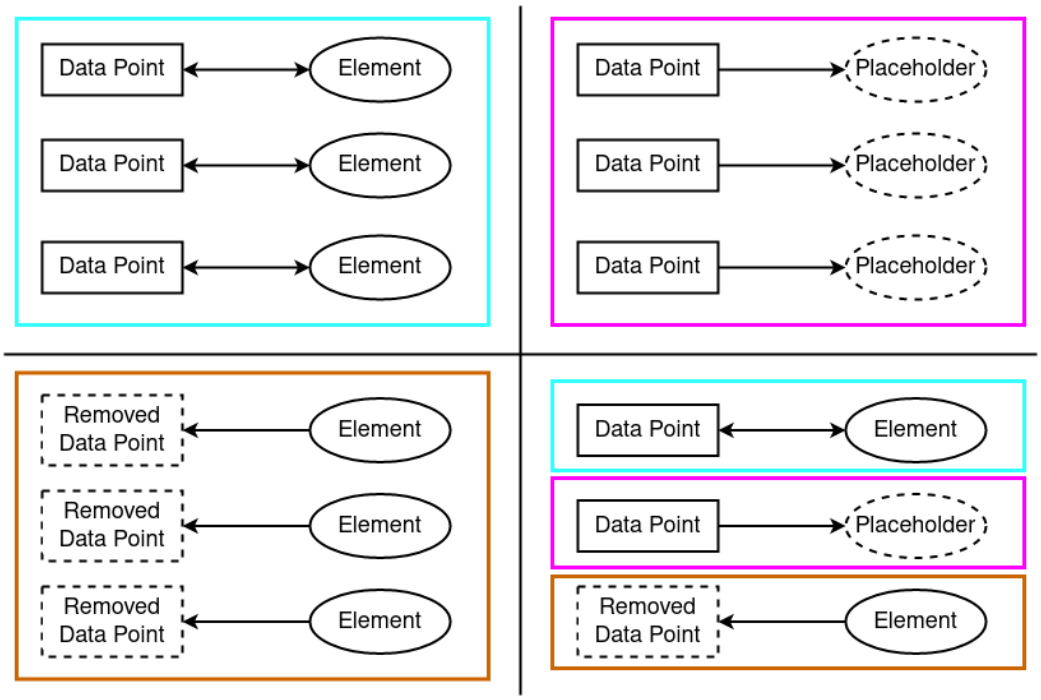

Figure 2.5: A representation of the possible cases when creating data joins. In the top left, the data join was able to match all data points to an element of the provided selection. Those cases are marked in cyan. In the top right, there are data points but no elements in the provided selection. Therefore they are matched to placeholder elements. These cases are marked in magenta. In the bottom left, the provided selection is filled with elements, but their previously matched data points have been removed. These cases are marked in orange. The bottom right shows that all three previous cases can exist in a single data join. For each of the three cases, the general update pattern can have different behavior specified.

usually a corresponding element is created as the first step. Often these elements serve as the marks representing data points in the diagram. Adding elements includes providing enough attributes to the elements, for them to be appropriately matched the next time the data join is called. When creating an element which is a mark in a diagram, providing appropriate attributes and styles of an element which corresponds to using the desired channels for data encoding.

All the elements which are already linked to a data point and therefore identified correctly using the identifier function, make up the update selection. They are marked in blue in fig 2.5. Specifying the behavior of the update selection allows the diagram to react to changing data by moving existing elements or changing their appearance to accommodate for other new or removed elements.

The last selection, the exit selection, is made up of all the elements previously matched to a data point, for which the corresponding data point has been removed from the data set. They are marked in orange in fig 2.5. The behavior of the exit selection is by default defined to remove the respective elements. The exit behavior can be defined if a more visually pleasing removal of elements is desired, like fading out before deleting.

When the goal is to create only static diagrams, which are only initially created from data, it is enough to define the behavior for the enter selection, as all data points will be matched up with a placeholder when first creating the data join. Here the identifier function is also not important, as the created element will not need to change over time and therefore do not need to be appropriately matched by the data join. But if diagrams should be able to react to data changes and update their appearance, like in this thesis, it is important to define the update behavior as well as a proper identifier function, so elements are always matched with the same data points. It is also important to provide elements which are created in the enter behavior with enough information, that the next time the data joins underlying selection is done, the newly added elements are matched as well.

2.3.4 Scales

Scales are a way to convert between two data ranges, like the scale factor of maps and model-kits. There are three types of scales: scales with continuous domain and range, like the example above, scales with categorical or discrete domain and range and scales with a continuous domain and a categorical or discrete range. The reason why categorical or discrete are always mentioned together, is due to the implementation of scales. As the domain and range in discrete or categorical cases can always be represented in an array filled

with a set number of entries, there is no difference in the array consisting of strings or integers.

An example for scales with continuous domain and categorical range is converting percentages of correct answers in a test, continuous data, to the appropriate grade, categorical, more specifically ordinal data. An example for a scale with a categorical domain and range is sorting mail. Depending on the destination town of a letter, it will be sorted into a specific box. It is noteworthy that scales with continuous domain and range can be used in reverse as well.

As most diagrams created with D3 are created as SVG, the scales provided by D3 are, in this thesis, mostly used to convert from the range of available data to the coordinate space in which elements should be drawn. All scales require a domain and a range. The domain describe the input values, the range where they should map to. Some scales can dynamically create their domain. Each new value used for querying the scale is added to the domain if it is not included yet.

2.3.5 Plugins

D3 provides the most used, general functionalities in the core library. Yet there are many plugins which can be added to add functionalities for more specific use-cases. Plugins needs to be loaded additionally to the core library. This thesis makes use of the Sankey plugin [16], to draw the Sankey graph.

As D3 is an open-source project, the plugins available are not all created by the creator of D3, Mike Bostock. Instead a majority is created by the community using D3.

3. Implementation

In the following sections the process of creating the diagrams and the showcase are described. There are several parts to this process. At first the data sets to be represented are selected. In most real world usages, this is already given when having to create data visualizations. Afterwards, section 3.2.1lists not only the implemented diagrams, but also why they have been selected. In section 3.2.2 and following, their implementation and usage of D3 is described. Finally, in section 3.3, the showcase, as well as how data updates are realized, are described.

The implementation uses the currently latest version of D3, version 7.6.1. The JavaScript code uses exclusively arrow functions which have been introduced in ECMAScript6, the specification which JavaScript is build upon[17]. The implementation has been tested using the following operating systems, browsers and versions:

1. Windows 10:
   (a) Opera - version xxxxx
2. Pop!_OS 22.04 LTS:
   (a) Firefox - version 103.0 (64-bit)

Besides the usual code comments, all the code files created for this thesis are commented to highlight and show the working and functionality of D3 and how it is used. As not every section of code is described in this chapter, the comments provide additional help to readers trying to understand the implementations of the diagrams. Furthermore it helps developers interested in adapting the diagrams implementations for their own use.

3.1 Data sets

As different data-types allow for different representations and require varying parts of D3, the data used in this thesis has been specifically chosen to contain

both types of categorical as well as numeric data. As there are no differences in the implementation of discrete and continuous data, only discrete data is used.

All data used for the creation of the diagrams in this thesis originates from the UNHCR Ukraine refugee situation page [18]. "UNHCR stands for United Nations High Commissioner for Refugees, also known as the UN Refugee Agency. It was created in 1950 to help millions of Europeans who had fled or lost their homes during the Second World War. Today, UNHCR protects and assists millions of displaced and stateless people around the world."[19] This thesis makes use of three data sets. The first data set is about the total cumulative border crossings from Ukraine per day [20] and is in JSON format. The other two data set are about the border crossings into countries featured in the refugee response plan and about border crossings into other neighboring countries [18]. They were extracted directly as CSVs. While all data references border crossings from Ukraine and not refugees directly, the UNHCR states that "[they do] not count border crossings of individuals from bordering countries leaving Ukraine to return home (i.e. Romanians returning to Romania), nonetheless among those forced to flee Ukraine are also Ukrainian nationals with dual citizenship"[21]. Therefore this thesis will henceforth refer to the individuals that crossed the borders as refugees. The refugees per country cover a time span between february 24th 2022 up until august 16th 2022[18]. The refugees per day cover the time from february 24th 2022 until july 17th 2022[20].

As the situation in the Ukraine is still ongoing, it is hard to acquire accurate refugee data. This is also mentioned on the UNHCR situation page, were it is stated that "Statistics are compiled mainly from data provided by authorities. While every effort has been made to ensure that all statistical information is verified, figures represent an estimate. Triangulation of information and sources is performed on a continuous basis. Therefore, amendments to figures may occur, including retroactively."[18].

To keep the implementations of the diagrams as simple as possible data preprocessing is done as the data is loaded from their respective files, to make it align with the internally used data structure. Therefore two data loader JavaScript files have been created. The first JavaScript file, the countryDataLoader.js reads both csv files containing information about the refugees crossing into neighboring countries. Both files are then combined to one data array containing an object, with properties for country and refugees, for each data entry. The second data loader, the dailyDataLoader.js, reads the JSON file containing information about the total refugees per day. This JSON file contains a lot of filler data repeating http request headers, timestamps in both unix and YYY-MM-DD format and a brief description of the Ukraine

situation as well as the data. Therefore the data loader strips all unnecessarily information away and produces a single array. This array contains an object, with properties for date and refugees, for each data entry. A section of both of the resulting data arrays can be seen in table 3.1. Both data loaders are accessed either by the respective data services which pass the data to the applicable diagrams, or by the diagrams directly. The data services, countryDataService.js and dailyDataService.js, are used by the showcase to fill the data tables in the showcase and pass along any manual data changes done in the showcase to the applicable diagrams.

Country	Refugees
Poland	5439431
Russian Federation	2197679
...	...
Belarus	16689

Date	Refugees
Feb 24, 2022	79209
Feb 25, 2022	179525
...	...
Jul 19, 2022	9567033

Table 3.1: A preview of the two data arrays after preprocessing. The left table contains the refugees per country. The right table shows the cumulative refugees per day. Each entry in the table corresponds to one object in the appropriate data array. Each object has two properties, according to the headers of the tables. The dates in the right table are JavaScript date objects. They are shortened here for readability.

Together both resulting data sets contain most of the data types. The number of refugees, which can be found in both data sets, is a discrete attribute. The countries in one data set are a nominal attribute. The date in the other data set is a ordinal attribute instead.

3.2 Diagrams

The following sub-sections are about selecting and implementing each diagram. As there are countless ways to show the selected data sets, a selection of diagrams was made first.

3.2.1 Diagram selection

Two major aspects are taken into the account for selecting the diagrams. Primarily, they should all require different functionalities of the D3 library. This ensures that this thesis actually tests the broad possibilities of D3. To

achieve this, four diagrams have been chosen to show the refugees per country data set. A bar chart, a donut chart, a tree map and a Sankey diagram. For the data set showing the total amount of refugees over time, a circle graph and an area graph have been chosen. Secondly, they should be realistically usable. This means that they should be usable by for example news agencies or on the UNHCR website, as there is no point in creating unnecessarily complex and unusable visualizations.

The bar chart was chosen for implementation due to its simplicity and minimal amount of D3 functions needed. As it mostly relies on simple `rect` and `text` SVG tags, it provides a good starting point for learning D3. Additionally, it is a frequently occurring diagram. Especially considering that column charts only differ in the orientation, but are functionally the same. The bar chart was chosen over a column chart, as the horizontal orientation of the bar chart allows the viewer to read the country name and the number of refugees in one line. The main reason for choosing the donut-chart was its use of the specific D3 functions for creating pie and donut-charts. The donut was chosen for it's compact form while still allowing some additional information to be shown in the center. It is also used to present custom attribute tweens for animations. The usage of D3s tree map functions was the main reason for choosing the tree map. Additionally it offers a good example for including CSS styling tricks, a tooltip and event-handling using D3. Both the tree map and Sankey diagram also provide an insight into working with hierarchical data structures. The Sankey diagram was additionally chosen to present the usage of D3 plugins. The circle graph was chosen to show more D3 scales, as well as the possibilities of using scales to create custom legends. Lastly, the area graph uses more basic D3 functions for rendering lines and areas. These have not been used before, but can prove quite powerful when creating diagrams showing trends over time.

Whilst all diagrams are presented in one showcase, each diagrams is implemented to work standalone. This makes the comparison between diagrams, as well as evaluating the effort needed to create them easier. It also allows for easier adaptation if one were to use one of the diagrams as a template. Therefore all diagrams are implemented independently, using three parts: a HTML, a CSS and a JavaScript file. The HTML loads the D3 library and the applicable data loader in the header. This data loader is only used when the diagram is accessed directly. This can be done by clicking the diagram in the showcase. The body of the HTML consists of a SVG tag where the diagram will be drawn, and a script tag which loads the JavaScript file. The CSS defines the general styling of the diagram which is not dependent on the input data. The main part of the implementation is done in the Javascript section.

The JavaScript file follows a general pattern. At first there is a initialization section (3.2.3) which is run once as the diagram is loaded. It is followed by a render function (3.2.4) which is responsible for drawing and updating the diagram. As there is no point in drawing a diagram without data, the process of how the diagram is provided the data to show is described next.

3.2.2 Data acquisition

Each diagram has two possible ways to acquire data. This is due to the fact that they are implemented to be shown in the showcase, but also work standalone. Each diagram will first check if there is a data service available. If a data service is available, the diagram will register itself with the data service for data and data updates (Figure 3.1). Otherwise the diagram will access the data loader directly (Figure 3.2). This will only provide data once, as the diagram is loaded.

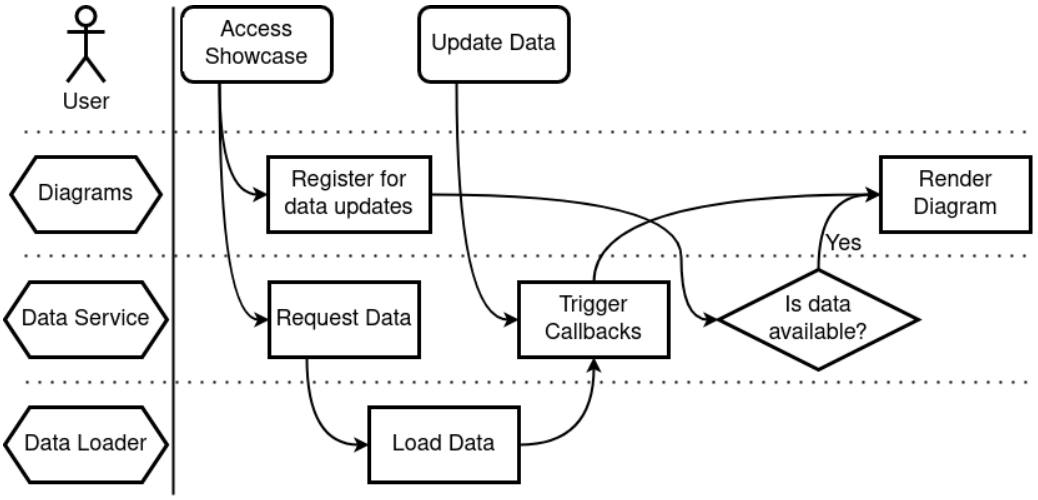

Figure 3.1: This flowchart describes the behavior when loading the showcase and when data is updated. As registering with the data service and loading the data happens simultaneously, the data service returns the latest available data to the diagram when it registers for data updates. This prevents issues which might occur due to timing.

3.2.3 Initialization

All things which are independent of the data are done during the initialization of a diagram. It starts with setting some core variables. A reference to the SVG tag, which will be used as the container for the diagram, is made. It

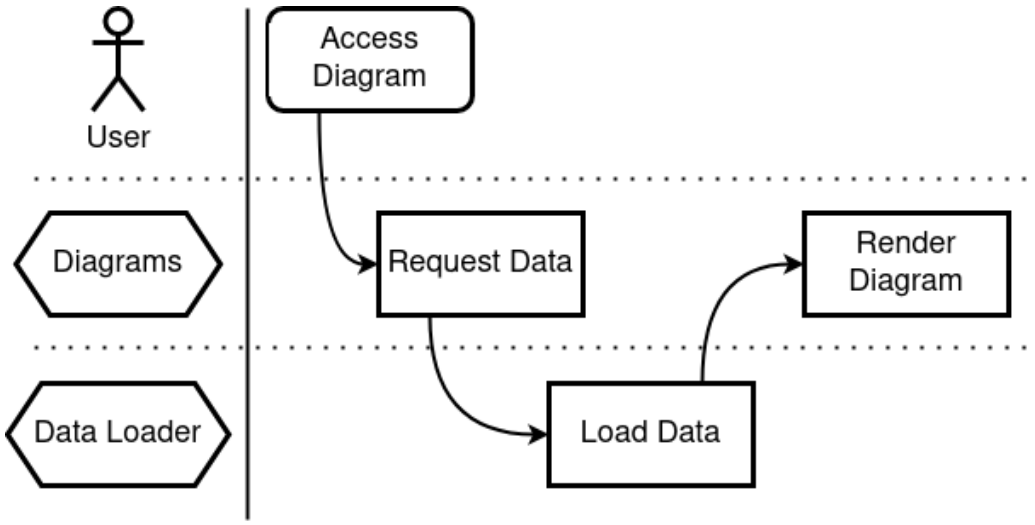

Figure 3.2: This flowchart describes the behavior when loading a diagram directly. As the diagram is unable to register with a data service, it will request the data directly from the data loader.

is followed by a margin definition for all four sides, where the margin of our diagram content in relation to the container size is defined. The resulting values for `contentHeight` and `contentWidth`, which are used as space to draw the diagram, are saved.

Afterwards, there are a few SVG group elements added to the SVG tag. Adding elements is achieved by calling `.append('elementName')` on an existing element. In the first case, this element is the previously stored reference to the SVG. As the `append` command returns the newly created element, it can be directly stored in a variable for later reference, or provided with attributes via method chaining. The group elements added here provide a general hierarchy for different aspects of the diagram. Listing 3.1 demonstrates how this works for the bar chart. Having a proper structure in place makes working with selections easier, helps with human readability of the SVGs content and makes debugging using the browsers inspector faster. As SVG elements are drawn on top of each other depending on their hierarchical order, this can also be used to mimic layers as they would be used in drawing applications. This general hierarchy is only created to a level which is independent of the provided data and differs depending on the type of diagram. For example the bar chart has separate groups for the axes and the content, see listing 3.2, while the circle diagram has groups for the background legend and the content.

```
1  const diagramGroup = SVG.append('g')
2      .attr('transform', `translate(${margin.left},${
           margin.top})`);
3
4  const xAxisParentGroup = diagramGroup.append('g')
5      .attr('id', 'xAxis')
6
7  const yAxisParentGroup = diagramGroup.append('g')
8      .attr('id', 'yAxis')
9
10 const contentParentGroup = diagramGroup.append('g')
11     .attr('id', 'content')
```

Listing 3.1: JavaScript code to create the hierarchy as used in the bar chart. The first line adds a new group element to the main SVG container using the `append` command. The newly added group element is saved in a constant for later references. Furthermore in line two an attribute is added to the new group element using the `attr` command and method chaining. It moves the group element from the left and top to allign with the margin definition. In each of the lines four, seven and ten, another group element is added. They are added to the previously created group element. They are all stored in constants for later reference and are provided with ID's for easier identification and debugging.

```
1  <SVG>
2      <g transform=translate(118,20)>
3          <g id="xAxis"></g>
4          <g id="yAxis"></g>
5          <g id="content"></g>
6      </g>
7  </SVG>
```

Listing 3.2: The HTML structure which results from the JavaScript code in listing 3.1. The resulting tree structure clearly separates the different aspects of the diagram. Using a hierarchical approach makes not only later selections easier, but also increases human readability and simplifies debugging.

If there are data independent scales, they are defined next. A common example here is a color scale for nominal values. It is used in all diagrams showing the refugees per country. When queried, it will return a new color from the provided range for each new query value. This scale does not require a predefined domain. Instead it is dynamically defined and extended with each new value querying the scale. It is important to note that when the

predefined color list runs out of new colors, it starts reusing the same color list from the beginning.

```
1 const colors = d3.scaleOrdinal(d3.schemeDark2);
```

Listing 3.3: Definition of the data independent color scale. `d3.schemeDarkv2` is a predefined list of color values which is used as the range of the scale.

If there are any static elements, they are also defined in the initialization. For example the tooltip used by the tree map or the center text fields in the donut chart. Even though the data which these fields are supposed to show is not yet known, these fields are persistent and can therefore already be created here.

Finally any general helper functions are also defined here. Both diagrams about the cumulative refugees per day make use of a date conversion function, which produces a nicely formatted date string from a JavaScript date object. The resulting strings are in the form of mmm-DD, for example Feb-07 or Jun-15.

3.2.4 Render

Following the initialization section is the render function. The render function is responsible for drawing and updating the diagram. The render function is called once in the beginning and every time the data set changes. The render function covers all data-dependent tasks, including the implementation of the data joins and general update pattern. As the implementation of the render function greatly varies between the diagrams, all common features are described first. Afterwards the specifics for each diagram are described. If there are helper functions or constants required by the render function they are defined first.

As all the diagrams make use of transitions for animations, a transition is defined with a duration of 1500 milliseconds. This transition is later called for each element which should be animated. Defining the transition here allows for all later calls to not only take the same amount of time without having to change the duration in more than one position, but also to reuse the same transition instance.

Scales

The data-dependent scales in this thesis are mostly used to calculate the coordinate position and sizing of elements in the diagrams. The bar chart for example defines two scales, a linear scale to find appropriate x-coordinates

and a band scale for the y-coordinates of each bar. As these scales domains depend on the range of the provided data set, it is necessary that they are redefined with every render call.

Data Joins

After the scales are defined, the data joins are created. While some diagrams, like the bar chart, only use a single data join (see A.1 line XX), other diagrams, like the circle graph, make use of several data joins (see A.5 lines XX and YY). Usually this is in accordance to how many independent parts the diagram consists of. The circle graph uses one data join for the size legend in the background and to update the circle showing the current data as well as the text showing the current number of total refugees.

As all diagrams in this thesis can react to data changes, they have a custom identifier function. For the refugees per country data set in this thesis, the identifier function is `d => {return d.country}`.

When a data join is initially created, or when data points are added, there is not a sufficient amount of elements in the selection to pair them with data entries. D3 will therefore create empty placeholders for these elements. To make these placeholders become a part of the DOM, the `.join()` function is added after the `data()` call. There are two ways to use the join function. One can either pass a string which will result in adding a matching tag to the DOM, or define functions for the general update pattern. When using the simple string method, the attributes and style for each new element can then be defined by method chaining. This approach is reasonable for diagrams that do not need to react to data changes. As all diagrams in this thesis implement the full extent of the general update pattern to be able to react to changing data and use the full possibilities of D3, the second approach is used.

General Update Pattern

When the `join` function is called, instead of passing a single string as parameter, three functions can be passed as parameters. These three functions correspond to the three cases of the general update pattern and describe their respective behavior. Each of the three functions has one input parameter, corresponding to the respective sub-selection. In the enter function an element is added to the DOM. In the update function existing elements are updated to accommodate for data changes and therefore possibly removed or newly added elements as well. In the exit function elements are removed. As this is the default behavior, the exit function definition can be omitted if

no extra behavior is desired. All three functions run on all the elements of the appropriate sub-selection.

The enter function adds the applicable element to the SVG. Therefore the first part of the enter function is usually an `.append(string)` call. The string describes the tag which will be added to the DOM. Afterwards the applicable styles, attributes and sub-elements are added. This can be achieved with the `.attr("attributeName", "value")`. While styles can be added with the `.style("property", "value")` function, the same can be achieved more cleanly by predefining styles in the CSS and adding applicable classes to the element. The selector used to define the current data join, should be able to match the newly created element as well. Therefore enough attributes need to be provided. This is important when the selection is recreated for updating the diagram. When positioning and sizing a new or existing element the scales are used to compute the applicable coordinate space.

The update function is necessary when the diagram should be able to react to data changes. It is usually similar to the enter function, in that it adjusts the positioning and sizing of the elements according to the possibly changes scales.

The exit function is defined by default to simply remove the applicable elements. The diagrams is this thesis do not specify a more complex exit behavior.

All three functions can make use of animations and transitions to improve their feel. The diagrams created for this thesis only animate the enter and update behavior. Why animations are useful and how they are implemented is described in the following section.

Animations

Animations can improve the feel, appeal and readability of diagrams. When a diagram is updated due to data changes, it is easier to understand and see the changes when for example bars in a bar chart shift to their new positions, instead of a seemingly entirely different diagram popping up. The animations allow the viewer to keep track of the existing entries and visually follow any changes. For example the growth or shrinking of the length of a bar in a bar chart. Animations can also be used when initially drawing the diagram, to guide viewer attention.

Animating elements in D3 is achieved by using transitions. Transitions are called from a selection and run on all the elements of the selection. A transition requires a duration and can also be provided with a delay and an easing function to improve the feel. The duration and delay are both in milliseconds. Animating numerical or color values is very easy with transitions.

It is only required to call the attribute or style with the target value and the transition function will calculate and show interpolated frames. This makes it very fast and easy to animate for example positioning or sizing of elements.

Instead of using the default behaviors for numbers and colors or when trying to animate other values like SVG paths, a tween function can be defined using `attrTween` or `styleTween`. Both tweens need to return a function which will be invoked for each frame of the animation, with a time value between 0 and 1, depending on the frame. The returned function must itself return a value, which is applied to the desired style or attribute every frame. In this thesis custom tweens are only specifically defined to animate SVG path tags in the donut chart (see listing 3.5).

Bar Chart

The bar chart, as well as the following implementations of the donut chart, the tree map and the Sankey graph all make use of the dataset containing the number of refugees per country they fled to.

The bar charts first defines two scales, the `xScale` as well as the `yScale` (see A.1 lines 42ff). The `xScale` is a linear scale to convert from a domain of the refugees [0, HighestNumberOfRefugeesInAnyCountry] to a range of the available space [0, contentWidth]. This allows to find the appropriate x-coordinate for any number of refugees. It is used to draw each bar to its appropriate length. The `yScale` converts from any given country to a y-position is done using a scale band. Therefore the domain is defined by providing an array of all possible countries and the range is [0, contentHeight]. This allows the proper height positioning of each bar using the resulting y-coordinate.

The bar chart also makes use of axes. The y-axis shows the countries, while the x-axis shows the amount of refugees. D3 has predefined functions to create axes from scales. An axis in D3 consists of many ticks. By default each tick has a label and a small line indicating its position. Furthermore there is a start and end line to indicate the whole domain. The bar chart removes all the domain lines for styling reasons. The tick lines for the y-axis are also removed, as they are unnecessary here. The tick lines of the x-axis are extended to cover the whole diagrams height. This is done to have a proper reference to read out the bars length.

```
1  const xAxisTickFormat = number =>
2      d3.format('.2s')(number)
3          .replace('0.0', '0');
4
5  const xAxis = d3.axisBottom(xScale)
6      .tickFormat(xAxisTickFormat)
7      .tickSize(-contentHeight);
8
9  xAxisParentGroup.call(xAxis)
10     .attr('transform', `translate(0,${contentHeight
           })`)
11     .select('.domain')
12         .remove();
```

Listing 3.4: The x axis implementation of the bar chart. The first constant defines the tick format. The provided '.2s' means that for each number there should be two significant digits. As the zero value is therefore represented as "0.0" to match the previous rule of two significant digits, it is simply replaced by "0". The second constant defines the function creating the axis. The previous tick format is passed here. Furthermore the `tickSize` is set to the height of the diagram. This way the initially small tick lines now cover the whole height of the diagram and allow for easier and more accurate readouts. Finally the `xAxisParentGroup` element, which is part of ther permantent hierarchical structure of the bar chart, calls the `xAxis` function. This adds the Axis to the diagram. As a last step the domain lines are selected and removed for styling reasons. (Section from A.1 lines 60ff)

Each bar in the bar chart is composed of a SVG `rect` for the bar itself and a `text` field as the label with the precise amount of refugees. The `rect` and `text` are both children of one group element with the `bar` class as attribute. Due to this structure, the bar chart only requires a single data join. The required selector matches all group elements with the `bar` class. In the enter sub-selection of the general update pattern, first a new group element with the `bar` class is added. Afterwards a `rect` as well as a `text` are added to the newly created group element. The `rect` is filled with the appropriate color by calling the color scale. Both elements are positioned and sized accordingly using the previously defined scales. The x-position of the text as well as the width of the rect are initialized as 0. These two values are animated using a transition, to reach their actual values. This way new bars always build themselves up from the left side. The text field usually tries to stick to the inside right side of the rect. In case where there is not enough space available

to the left, as the bar is shorter than the number to show, the text is placed to the right of the rect.

Elements in the update sub-selection are resized in the width and text value if the number of refugees for this country or the range of the `xScale` changed. They might also be repositioned and resized in height, as new countries are added, or old ones removed from the data set. All resizing is done using the transition for smooth animation of the changes. Elements in the exit selection are simply removed.

Donut chart

As the donut chart shows the total amount of refugees in the center, this value is computed first. This is achieved using the `d3.sum(data, d => d.refugees)` function (see A.2 line 82). It creates the sum of all entries in the data, using the `refugees` field for each entry. After calculating this value, the appropriate text field is updated to contain the new number.

As each section of the donut-chart is made up of a path element, D3 provides two functions to generate pie and donut charts. The `d3.pie()` function calculates the appropriate start and end angle of each data-point (see A.2 lines 90ff). A padding angle has also been specified for some spacing between the sections of the diagram. The `d3.pie()` function returns a new object which holds a reference to our original data, additionally to the new sections information (see A.2 lines 96ff).

The `d3.arc()` function is set up with an inner and an outer radius. Having an inner radius of zero generates a pie chart, whereas an inner radius greater than zero, like in this instance, creates a donut chart. The `d3.arc()` function which is set up here is later used in the general update pattern to generate SVG path objects from the pie pieces containing the start and end angles of each section.

As each section of the donut chart is made up of a `path` element nested inside a group element, the selector for the data join matches all group elements with the `arc` class. The data join is then created using the previously created pie object. In the enter selection of the general update pattern, the parent group element is created first and provided with the `arc` class. Afterwards a path element is added to this group element. This path element is colored according to the color scale. Drawing the actual arc piece is done in an attribute tween. This animates the donut chart to smoothly fill itself in the beginning. The implementation of this initial animation is almost same as the one seen in 3.5 for updating the arc sections. The only difference lies in the two `interpolate` functions using 0 as initial value instead of the respective `previousStartAngle` and `previousEndAngle`, as these two values

are not assigned yet.

Lastly the path elements are registered to two callbacks, `mouseover` and `mouseout` using D3s `.on()` function (see A.2 lines 123 and 129). Both these events are used to show and update the appropriate text in the center of the donut chart. The effect which creates an outline around the currently hovered over path element is not linked to these events, but instead is achieved by CSS styling.

As the update behavior of the donut-chart consists only of updating the paths, it is fully described in listing 3.5. The exit behavior removes appropriate sections.

```
1  // The core of the donut animation is defined here.
2  const animate = (nodes, index, d, i, j) => {
3      nodes[index].previousStartAngle = d.startAngle;
4      nodes[index].previousEndAngle = d.endAngle;
5
6      return time => {
7          d.startAngle = i(time);
8          d.endAngle = j(time);
9          return arc(d);
10     };
11 };
12
13 [...].call(update => update.transition(t)
14     // The update animation for the donut pieces is
           specified here.
15     .attrTween('d', (section, index, nodes) => {
16         const interpolateStartAngle = d3.interpolate
               (nodes[index].previousStartAngle,
               section.startAngle);
17         const interpolateEndAngle = d3.interpolate(
               nodes[index].previousEndAngle,
               section.endAngle);
18
19         return animate(nodes, index, section,
               interpolateStartAngle,
               interpolateEndAngle);
20     }));
```

Listing 3.5: The implementation of the arc update animations. As the core of the animation donut sections is used by the enter and the update behavior, it is defined first. The new values for start and end angle are stored on the node itself. This needs to be done to be able to reference these values again for the next update, as the previous angles will not be accessible through the pie object after regenerating it for an update. Finally the function which is called for each frame of the animation is defined and returned. This function first interpolates the start and end angle values using the passed interpolation functions and the time value. This time value is in the range of zero to one, depending on how far along the animation is. These newly interpolated angles define the start and end angle of the pie piece, which is then turned into a path element by calling the `arc` function for this pie piece. As the interpolate functions `interpolateStartAngle` and `interpolateEndAngle` differ for the enter and update behavior, they are defined in the respective sections. They are passed into the core `animate` function. (Section from A.2 lines 136ff)

Tree map

As the tree map is intended to work with hierarchical data, it requires all data points to have a link to their parent data point. There is only one data point without a link to a parent, which serves as the root element. As the refugees per country data set is not in hierarchical structure, this is simulated first. Therefore a dummy object is added to the data array. Using the `d3.stratify()` command turns the data set into a tree object by connecting each data point to a parent, in this case the dummy object (see A.3 line 70). The dummy element has no parent and serves as the root of the tree object. After removing the dummy element from the data again, the sum of refugees in the tree object is calculated. Using the `d3.treemap` command and providing it with information about the available space and padding between elements provides the trees leafs with their relevant size and position information (see A.3 line 85).

The general update pattern adds a `rect` for each leaf of the tree object before styling it appropriately and animating its size and position. The `mouseover`, `mousemove` and `mouseout` events are registered for showing, updating and hiding the tooltips content and position, as the mouse moves over a `rect` in the tree map. The update selection smoothly moves and resizes the rects when applicable, while the exit selection removes them.

Sankey diagram

The Sankey diagram consists of nodes and links. Therefore these two arrays are constructed first. Each node represents one country. The links describe the flow of values, in this case refugees, between nodes. This is achieved by providing a source node, a target node and the value. To be able to show a flow from the Ukraine to the other countries, the Ukraine is added as a node. All countries from our data are added as nodes, as well as a link to the Ukraine node is created. The resulting arrays are provided to the `d3.sankey()` function (see A.4 line 48). This function is not part of the default D3 namespace. Instead it is added by additionally importing the Sankey package in the Sankey diagrams HTML page header. The `d3.sankey()` function adds additional information to the nodes and links allowing for appropriate placement of the according elements. It also adds all the links' values to create a source value for the Ukraine node.

The Sankey diagram makes use of two data joins. The first one links the nodes to appropriate group elements. During the enter behavior, each group elements is filled with a `rect` and a `text` element. The `rect` represents a node in the Sankey diagram and is sized according to the number of refugees

and styled according to the country using the color scale. To make the color scale consistent with the other diagrams, the Ukraine node is provided with a fixed color. This prevents the Ukraine node from querying the color scale and creating inconsistency with the other diagrams which do not contain a representation for the Ukraine. The `text` label contains the name and amount of refugees per country. It is placed next to the appropriate `rect`s. This can be on the left or rights side, depending on the x-position of the rects. If the rect is in the left half of the diagram the label is right, if the rect is in the right half the label is left. Both rect and text make use of transitions to smoothly build up the diagram. The text also makes use of a small delay to the transition by using the `.delay(100)` function (see A.4 line 93).

The second data join takes care of the links. They are matched to path elements. Each path is styled to match with the country it leads to. The required SVG path is created using the `d3.sankeyLinkHorizontal()` command (see A.4 line 111). As the resulting path is only a single line, the `stroke-width` attribute corresponds to the size of the link. The links are also animated using the transition. When initially appearing, the links are provided with a delay before fading into existence after the nodes have settled in their positions. The update behavior makes sure that the links are smoothly transitioned to their necessarily position and the width adjusted as needed.

Circle graph

The circle graph, as well as the area graph described in the following section both make use of the data set about the cumulative refugees per day.

The circle graph makes use of two data dependent scales. A time scale is created using `d3.scaleQuantize()` (see A.5 line 55). It is used for converting the time slider value to an actual date. As the time slider value provides a value between zero and one, this domain is mapped to the range of available dates in the data set. The second scale is used to get the correct radius of the circle. Because the area of the circle corresponds to the number of refugees, it is important to not scale the radius linearly. This would lead to circle areas which do not represent the correct number of refugees. Scaling the radius linearly would introduce a lie-factor higher than one. In fact, due to the relation between a linear change in radius and a change which keeps the area consistent, the lie factor increases, the higher the amount of refugees is. To avoid this, a `d3.scaleSqrt()` is used (see A.5 line 60). The domain is set to [0, HighestNumberOfTotalRefugees] and the range to [0, contentHeight/2]. This scale is also used for drawing the background legend. This is achieved by getting the ticks of the scale and saving the appropriate

values in an array. This array is rendered in circles using the first data-join. This data-join draws and updates a circle and a text label for each tick.

As the actual content of the diagram does not draw one mark per data-point, the refugee number for only one data entry is shown at a time. The time scale is used to determine which is the currently selected date from the time value which is provided by the slider in the range of zero to one. The resulting date is used to get the corresponding data for that date from the data set. This allows the diagram to reuse only use one circle and one text label in its representation. Therefore the second data join also looks a bit different. Instead of linking actual data, an array with an arbitrary single entry is linked to the selection. Without this single element, D3 would not draw the circle, as it assumes that there is no data to show. The identifier function also always returns the same value, no matter what data was linked. This makes sure that the same circle element is matched by the selector on each render call, as to not draw a new circle every time the diagram is supposed to update. The circle and text are both created and styled using the data which was found for the current date.

Area graph

The area graph makes use of the same time scale as the circle graph. Additionally it uses a linear scale for the y-axis and a scale band for the x-axis. The y-axis represents the number of refugees, while the x-axis shows the days. Both axes are added to the diagram in the same way as is done in the bar chart. As showing all days on the x-axis would be too dense, the values are filtered and only 15 equidistant days are shown. This is achieved by using JavaScript `filter` function for arrays and specifying a custom filter which is to be evaluated for each entry in the domain of the x-axis scale band (see A.6 line XX).

The main content of the area graph consists of two parts. One part shows the area, while the other one is the date line showing the current date. The area and the line following along the top of the area, the top-line, are two separate path elements. Their definition can be seen in listing 3.6. Because the area and line both cover the whole range of data at once, the data join, which creates and updates them, uses the same single element dummy data as the circle graph. When creating or updating the line and area, the whole data set is passed to the respective functions.

```
1  const line = d3.line()
2      .x(d => xScaleWithOffset(d.date))
3      .y(d => yScale(d.refugees));
4
5  const area = d3.area()
6      .x(d => xScaleWithOffset(d.date))
7      .y1(contentHeight)
8      .y0(d => yScale(d.refugees));
```

Listing 3.6: The first constant defines the function for creating the top-line of the area graph. This is achieved by specifying functions for x- and y-values. D3 uses them to calculate the position of each point on the line. Using the arrow functions here applies this for each entry of the data set which is passed as a parameter when calling this function. When the function is called and a data set is provided, by calling `line(data)`, the line is constructed by calculating the according x- and y-positions for every data point. Defining and creating the area works in similar fashion. Yet there are two y-positions for each x-position. `y0` is the bottom of the shape, while `y1` is the top. Switching the definition of `y0` and `y1` results in a reversed direction of the SVG path describing the outline of the shape. The `d3.area()` function allows for a huge variety of shapes. Yet it is not possible to create concave ends on the right or left side of the shape, assuming left and rigth are oriented horizontally. Of course the created shape can be rotated and oriented using all possible SVG tags and attributes. (Section from A.6 lines 111ff)

The date line is made up of three elements. A circle element rides on the top line of the area, a vertical line to indicate the current day on the x-axis and a text label to show the days refugee value. All three parts are simply created and updated in the second data-join. This data-join makes use of the same kind of dummy data as seen in the previous data-join and the circle graph.

3.3 Showcase

To bring all the diagrams together, a showcase has been created. In addition to containing the diagrams, it also allows for manually modifying the data used to create the diagrams. The layout of the showcase as well as how data can be modified are described in the following sections.

3.3.1 Layout

The showcase is split into two main sections. The first section is about the refugees per neighbor country they crossed into. The second section is about the cumulative refugees over time. As all diagrams in one section represent the same data set it allows for an easy visual comparison. While the left side of each of the two section holds all the diagrams, the right side contains information regarding the diagrams, as well as a possibility to view and change data. The information area always tries to stick to the top of the screen, as long as space is available.

The information area of the first section shows a legend first. This legend is implemented using D3 and reacts to data changes like the diagrams do. The information area of the second section contains a time slider. This is used to select the current date which the circle and area graph take into account.

The showcase loads each of the diagrams into a separate `iframe` tag with a consistent aspect ratio. As the diagrams are set up to use all available space in the provided container, the showcase makes sure to size the `iframe` containers appropriately. When a diagram is loaded in the showcase, it also registers with the applicable data service. This allows the data service to update the diagram as the data is changed, as is described in the following section.

3.3.2 Data Updates

Each section of the showcase has a table which allows for data manipulation. These tables are hidden when the showcase is loaded. They can be shown by ticking the appropriate 'Show Data' checkbox. Rows of data can be modified, added or removed here. The data changes here are not persistent and are therefore not saved in the original data files. While the correctness of the initial data is not guaranteed, arbitrarily changing the data entries obviously falsifies the shown data. Changes in the data are registered by the respective data service when a data update is triggered. Data updates are triggered automatically as soon as any data is changed or rows are added or removed while the 'Auto Update' checkbox is ticked. This is the default case. If one wants to change more data points before a update is triggered, the auto update option can be deselected and the manual 'Update Data' button used instead. When a update is triggered, the data service then creates a new data set from the data in the table and passes it along to the applicable diagrams when calling their render function to update.

For this thesis it is important to be able to modify the data, as one of

the core features of D3 tested in this thesis is reacting to changes in data. This manual style of modifying data is probably not so common in real world applications. Yet it is easy to replace these manual data changes to regular API calls or other automatically updating data sources. As the source of the data changes does not matter for the functionality of D3, the manual approach chosen here is sufficient in demonstrating the possibilities of D3.

4. Discussion

The discussion is split into two parts. The first part is about the diagrams and comparing them to each other. The second part tries to answer the initial questions about the potential of D3.

4.1 Diagrams

The first four diagrams show the data about the refugees by country crossed into. While the underlying data is the same, the resulting diagrams are quite different. The donut chart and tree map make it immediately obvious, that almost half of all refugees went to Poland. While this is hard to guess from the bar chart without a ruler or calculator, the vertical spacing of the nodes in the Sankey diagram also does not make this obvious. Only the donut chart and the Sankey diagram allow the viewer to immediately see the total amount of refugees. While the bar chart and tree map also contain this information, they would require the viewer to calculate it themselves, defeating the purpose of data visualization. The Sankey diagram and bar chart directly show all the numbers of refugees per country. This information can be found in the donut chart and tree map as well, but only by using the mouse to inspect the diagrams more closely. This can be a struggle when trying to get more precise data about the refugees which went to Belarus, as their amount is so small, that these two diagrams hardly show Belarus at all. While the shown legend is universally true, both the Sankey diagram and the bar chart also directly show the names of the countries, making comprehension of the data more easy. The Sankey diagram is also the only diagram which conveys the flow from the Ukraine to the other countries. The other diagrams only achieve this through context, not in their actual graphic.

When data is updated, all four diagrams smoothly transition to their new states. This makes the changes quite easy to follow and pleasant to look at. The tree map can sometimes be a bit confusing, as the different rectangles can shift across each other.

The two diagrams showing the cumulative refugees per day, are quite different from each other. While the circle graph actually only ever shows the value for a single day, the area graph shows the whole extend of the data with an extra indicator for the current day. Due to the scaling of the radius in the circle diagram it can also be quite hard to read the number of refugees from the legend, especially for smaller values. The area graph does a much better job of not only consistently showing not the number of refugees on the selected day, but also the development of the refugee count over time. Therefore it struggles with accurately showing the current date. This is easier in the circle chart, as it is simply displayed at the bottom.

When data is updated, the circle graph can be completely unaffected. Only when the data for the currently selected day is changed, or the size legend adjusts itself changes can be seen. Because of the way that the functions used to create the line and area of the area graph work, the updates of the area graph are only easy to follow when values are adjusted. When days are added and removed, it is hard to follow changes.

4.2 D3

The tree initial questions presented which this thesis tries to answer were: What is the potential of D3 in data visualization? What are the advantages and disadvantages of using D3? When should D3 be used? All three of these questions will be evaluated and discussed separately in the following discussion, beginning with the question of what D3s potential is.

Looking at the created showcase, D3 can obviously be used to create many different types of diagrams. Looking at the examples found online, D3 has their own showcase of projects, makes this even more apparent. From simple bar and pie charts, over visualizing hierarchical data using tree maps or Sankey diagrams, like the ones created for this thesis, all the way to map based diagrams using various projections[22], physics enabled bubble graphs[23] and pseudo 3D animations[24]. Of course D3 can also be used to created animations which do not necessarily serve data visualization purposes, like the tadpoles example[25]. Due to D3's low level approach, fast speed, and the general update pattern, D3 can be used to create all visualizations one can imagine and have them react to data changes in real time. As D3 is built around simple DOM manipulation, it can also be used for other use cases. One can create HTML tables and populate them with data. One can make small animations to add visually appealing aspects to a website. Or maybe one can adapt the scales for their own needs of converting data. But what are the advantages an disadvantages of D3?

When working in a web environment, it is always easy to start using D3 as well. Its independence from any framework and its pure JavaScript implementation, make it possible to include D3 in any web-based project. On the other hand, if one is not already working on a web-based project but still wants to use D3, one has to deal with all the additional overhead of working with a web technology stack. Whilst importing the D3 library is really easy, the initial learning curve is everything but easy. Without first internalizing the core concepts of D3 and SVG, it is impossible to make any kind of visualization. While this is true for any framework or technology, the concept of the data joins and the general update pattern seem especially abstract. Yet it is crucial to understand it when a diagram is supposed to react to data changes. While learning the basics of D3 is quite a big hurdle, it can be broken down by first creating only static diagrams. This can be achieved without a deeper understanding of the general update pattern. While D3s low-level approach is cumbersome to comprehend initially, it actually allows D3 to be very flexible. In addition, while there are a lot of examples and tutorials, they are often too complex to understand as a beginner, or use varying versions of D3 or JavaScript styles. Besides some functionalities of D3 being obsolete in newer versions, the different styles of JavaScript can be additionally confusing when the developer is not familiar with the evolution of JavaScript. Once one understands how D3 works, it can be quite fast to create basic diagrams. Once one properly understands the general update pattern and selections, it is also not too difficult to react to data changes. Yet trying to animate diagrams can become tricky. It depends on the elements which are used and which attributes need to be animated. As all elements have to be manually specified, one has full control over the appearance and behavior of the diagrams. This allows D3 to adapt to any existing style guides, yet is also very time consuming. It also allows creating diagrams with a high lie-factor or a bad data-ink ratio. This risk can be mitigated using a more high-level library. High-level libraries also allow for the automatic creation of legends, instead of having to create them manually as was done in this thesis.

Another concern might be the performance of D3 when dealing with many marks in a diagram. Therefore the limits of smooth animation of the bar-chart and the tadpoles example[25] were briefly tested. Of course these results vary from device to device. In this case, it was possible to smoothly animate up to approximately 1200-1300 tadpoles. The bar-chart is harder to evaluate. The singular bars approach, and partially pass, the limits of a singular pixel in thickness, and therefore their visibility on the screen, with this number of entries in the diagram. While this thesis does not provide a proper performance test and comparison with other visualization libraries

performances, it can be safely assumed that performance will not be an issue for most diagrams.

So when should one use D3? This depends on the task at hand. If the goal is to create some diagrams as a one time job, D3 is unnecessarily complex. Tools like Excel can easily excel here. Even when working with a web project, D3 is probably too complex. Most developers will find all necessarily functionality using libraries like Chart.js[26] or the Plotly[27] JavaScript library, which is actually built on top of D3 to offer a more high level approach to data visualization. Both these open-source libraries allow creating some of the most common diagrams without having to learn all the quirks of D3. Yet if someone is to create a custom visualization or wants more fine control over all aspects of a diagram, D3 can handle it. Nevertheless, due to the high initial learning curve getting into D3 can only be recommended when the creating custom visualizations, which high level libraries can not provide.

5. Conclusion

While D3 is an immensely powerful tool, learning and understanding the core concepts of D3 took a surprisingly long time. While there are many examples, the inconsistencies in D3 versions as well as JavaScript versions were quite confusing. Having never worked with D3 and only having very limited experience with JavaScript, it took a fairly long time to get used to both. Even now, if I was tasked to create some simple diagrams, I would probably use a more high level library. Unless the required diagrams require the full potential of D3. A few parts have been especially cumbersome. While animating updates with transitions is usually easy, creating a smooth animation for the donut chart took longer than expected. Having to work with the custom attribute tweens and storing information on the DOM element itself, made this even more confusing. Yet this was in part due to my lacking proficiency in working with JavaScript and the differences between function declarations using arrow functions and the function keyword. I was also unable to animate the area-graph in a way where the date-line follows the line along the top of the area. Furthermore it would be nice if the date line would be draggable using the mouse. This is something which can most certainly be done. But not by me in the time span of creating this thesis. Creating the right selections and sub-selections when implementing the general-update pattern also is not always easy. While drawing a diagram initially is usually an easily achievable feat, making sure that update behavior reuses existing elements is sometimes tricky. The widespread use of D3 at least helped finding information on common issues, which was very helpful for bug fixing.

While the implementation doesn't differ for using discrete and continuous data, it would still have been nice to show this in an actual example. Another interesting aspect for which I did not have the time to implement, is working with maps and projections. A map could also have easily shown the refugee streams.

Due to the currentness of the data, the UNHCR was updating their situation page as well. On of the effects of this was that the terminology they

used changed from initially mentioning refugees, to later solely mentioning border-crossings. This can actually still be seen when looking at the raw data JSON file, which contains the cumulative number of refugees per day. It contains a description clearly mentioning the data being about refugees. As the diagrams and showcase were implemented before writing the text for this thesis, the renaming was glossed over, as all the created work would have to be redone. This is also why the code files always mention refugees.

Finally, while it was interesting to work with D3 and get to know its immense potential, it is devastating to see that, yet again, millions of people have been displaced by unnecessary violence and aggression.

Bibliography

[1] M. Sadiku, A. E. Shadare, S. M. Musa, C. M. Akujuobi, and R. Perry, "Data visualization," *International Journal of Engineering Research And Advanced Technology (IJERAT)*, vol. 2, no. 12, pp. 11–16, 2016.

[2] "Microsoft excel spreadsheet software: Microsoft 365," accessed:20.08.2022. [Online]. Available: https://www.microsoft.com/en-us/microsoft-365/excel

[3] "Spss software." [Online]. Available: https://www.ibm.com/analytics/spss-statistics-software

[4] "The r project for statistical computing," accessed:20.08.2022. [Online]. Available: https://www.r-project.org/

[5] "Matplotlib - visualization with python," accessed:20.08.2022. [Online]. Available: https://matplotlib.org/

[6] W. M. Senner, *The origins of writing*. U of Nebraska Press, 1991.

[7] "Total data volume worldwide 2010-2025," May 2022, accessed:22.08.2022. [Online]. Available: https://www.statista.com/statistics/871513/worldwide-data-created/

[8] S. García, J. Luengo, and F. Herrera, *Data preprocessing in data mining*. Springer, 2015, vol. 72.

[9] N. Henze, *Grundbegriffe der deskriptiven Statistik*, 13th ed. Springer Berlin, 2021, p. 21.

[10] "Available chart types in office," accessed:22.08.2022. [Online]. Available: https://support.microsoft.com/en-us/office/available-chart-types-in-office-a6187218-807e-4103-9e0a-27cdb19afb90

[11] J. Mackinlay, "Automating the design of graphical presentations of relational information," *Acm Transactions On Graphics (Tog)*, vol. 5, no. 2, pp. 110–141, 1986.

[12] E. Tufte and O. Katter, "The visual display of quantitative information," *Professional Communication, IEEE Transactions on*, vol. PC-27, 06 1984.

[13] E. Tufte, "The visual display of quantitative information."

[14] M. Bostock, "Data-driven documents," accessed:31.03.2022. [Online]. Available: https://d3js.org/

[15] A. v. Kesteren and L. Hunt, accessed:31.03.2022. [Online]. Available: https://www.w3.org/TR/selectors-api/

[16] "D3/d3-sankey: Visualize flow between nodes in a directed acyclic network." Sep 2019, accessed:22.08.2022. [Online]. Available: https://github.com/d3/d3-sankey

[17] "Ecmascript 6: New features: Overview and comparison," accessed:27.08.2022. [Online]. Available: http://es6-features.org/#ExpressionBodies

[18] UNHCR, "Operational data portal," accessed:18.08.2022. [Online]. Available: https://data2.unhcr.org/en/situations/ukraine

[19] ——, "Frequently asked questions." [Online]. Available: https://www.unhcr.org/frequently-asked-questions.html#whatdoesUNHCRstandfor

[20] ——, "Refugees per day," accessed:23.07.2022. [Online]. Available: https://data.unhcr.org/population/get/timeseries?widget_id=336969&sv_id=54&population_group=5460&frequency=day&fromDate=1900-01-01

[21] ——, "Explanatory note," accessed:18.08.2022. [Online]. Available: https://data.unhcr.org/en/documents/details/91338

[22] J. Davies, accessed:24.08.2022. [Online]. Available: https://www.jasondavies.com/maps/transition/

[23] S. Carter, "Four ways to slice obama's 2013 budget proposal," Feb 2012, accessed:24.08.2022. [Online]. Available: https://archive.nytimes.com/www.nytimes.com/interactive/2012/02/13/us/politics/2013-budget-proposal-graphic.html

[24] J. Davies, "Sphere spirals," accessed:24.08.2022. [Online]. Available: https://www.jasondavies.com/maps/sphere-spirals/

[25] M. Bostock, "Tadpoles," Sep 2020, accessed:24.08.2022. [Online]. Available: https://observablehq.com/@mbostock/tadpoles

[26] "Chart.js," accessed:27.08.2022. [Online]. Available: https://www.chartjs.org/

[27] "Plotly javascript open source graphing library," accessed:27.08.2022. [Online]. Available: https://plotly.com/javascript/

A. Appendix

A.1 Bar Chart - JavaScript

```
1  /**
2   * This script was created as part of a bachelor thesis.
3   * The results can be found here: https://github.com/
        StyxOo/styxoo.github.io
4   * Author: Luis Rothenhäusler
5   * Last edit: 25th August 2022
6   *
7   * This file contains the JavaScript implementation of the
        bar-chart.
8   */
9
10 /**
11  * In this first section, some data independent constants
        are defined.
12  */
13 // This creates a reference to the SVG container on the
       HTML page. This will contain the whole diagram.
14 const svg = d3.select('#mainFrame')
15     .attr('height', innerHeight)
16     .attr('width', innerWidth);
17
18 // The margin definition for the diagram. The content is
       padded from the sides using the margins.
19 const margin = {
20     top: 20,
21     right: 20,
22     bottom: 20,
23     left: 118
24 };
25
26 // contentWidth and contentHeight store the available
       coordinate space for the content of the diagram.
27 const contentWidth = innerWidth - margin.left -
       margin.right;
```

```
28  const contentHeight = innerHeight - margin.top -
        margin.bottom;
29
30  /**
31   * This section defines the hierarchy of the diagram.
32   * This makes later selections and debugging in the
         browser inspector easier.
33   */
34  const diagramGroup = svg.append('g')
35      .attr('transform', `translate(${margin.left},${
            margin.top})`);
36
37  const xAxisParentGroup = diagramGroup.append('g')
38      .attr('id', 'xAxis');
39
40  const yAxisParentGroup = diagramGroup.append('g')
41      .attr('id', 'yAxis');
42
43  const contentParentGroup = diagramGroup.append('g')
44      .attr('id', 'content');
45
46  /**
47   * This section defines the color scale used to color
         elements according to their country.
48   * It can be defined here, as it is independent of the
         data
49   */
50  const colors = d3.scaleOrdinal(d3.schemeDark2);
51
52  /**
53   * The render function is defined here.
54   * It is called to initially draw the diagram, as well
         every time the data changes and the diagram should
         update.
55   */
56  const render = data => {
57      console.log('Rendering bar chart');
58
59      /**
60       * The following defines the transition which is used
             for all animations.
61       */
62      const t = svg.transition()
63          .duration(1500);
64
65      /**
66       * Here all the required scales, which are dependent
             on the data, are defined.
67       */
```

```
68      // The xScale is used to convert from the number of
            refugees to the applicable x coordinate.
69      // It is also used while creating the x-axis legend.
70      const xScale = d3.scaleLinear()
71          .domain([0, d3.max(data, d => d.refugees)])
72          .range([0, contentWidth])
73          .nice();
74
75      // The yScale is used to convert country to the
            applicable y coordinate.
76      // It is also used while creating the y-axis legend.
77      const yScale = d3.scaleBand()
78          .domain(data.map(d => d.country))
79          .range([0, contentHeight])
80          .padding(0.2);
81
82      /**
83       * This section is responsible for creating the x and
             y axes of the bar-chart.
84       */
85      // This creates the y-axis from the scale and adds it
            to the diagram. It also removes the domain and tick
             lines.
86      yAxisParentGroup.call(d3.axisLeft(yScale))
87          .selectAll('.domain, .tick line')
88              .remove();
89
90      // This defines the function responsible for
            formatting the x-axis ticks.
91      const xAxisTickFormat = number =>
92          d3.format('.2s')(number)
93              .replace('0.0', '0');
94
95      // This creates the x-axis taking the formatting into
            account. Also tick lines will be drawn over the
            whole diagram.
96      const xAxis = d3.axisBottom(xScale)
97          .tickFormat(xAxisTickFormat)
98          .tickSize(-contentHeight);
99
100     // This adds the x-Axis to the diagram, positions it
            accordingly and removes the domain lines.
101     xAxisParentGroup.call(xAxis)
102         .attr('transform', `translate(0,${contentHeight})
                `)
103         .select('.domain')
104             .remove();
105
106     /**
```

```
107         * This is where the actual content of the diagram is
                drawn.
108         * Therefore, a data-join is created and the behavior
                of the general update pattern is specified.
109         */
110        contentParentGroup.selectAll('g .bar').data(data, d =>
               {return d.country})
111            .join(
112                // This describes the behavior of the enter
                       sub-selection of the general update
                       pattern.
113                enter => {
114                    // A group element is added for a new bar
115                    const bar = enter.append('g')
116                        .attr('class', 'bar')
117
118                    // The rectangle is added to the bar. It
                           is styled, positioned and animated.
119                    bar.append('rect')
120                        .attr('width', 0)
121                        .attr('height', yScale.bandwidth())
122                        .attr('y', d => yScale(d.country))
123                        .attr('fill', d => colors(d))
124                        .call(enter => enter.transition(t)
125                            .attr('width', d => xScale(
                                   d.refugees)));
126
127                    // The text is added to the bar. It is
                           provided the refugee value, as well as
                           positioned and animated.
128                    bar.append('text')
129                        .text(d => d.refugees)
130                        .attr('class', 'barText')
131                        .attr('text-anchor', 'end')
132                        .attr('dy', '0.32em')
133                        .attr('y', d => yScale(d.country) +
                               yScale.bandwidth()/2)
134                        .attr('x', 0)
135                        .call(enter => enter.transition(t)
136                            .attr('x', d => {
137                                // If the rectangle is too
                                       small, the text is placed
                                       to the right of it
138                                const scaleValue = xScale(
                                       d.refugees);
139                                return (scaleValue - 60 > 0) ?
                                        scaleValue - 10 : 60;
140                            }));
141                },
```

```
142             // This describes the behavior of the update
                   sub-selection of the general update
                   pattern.
143             update => {
144                 // The rectangle is selected and updated
                       in position and size.
145                 update.select('rect')
146                     .call(update => update.transition(t)
147                         .attr('width', d => xScale(
                               d.refugees))
148                         .attr('height', yScale.bandwidth()
                               )
149                         .attr('y', d => yScale(d.country))
                               );
150
151                 // The text is selected and updated in
                       value and position
152                 update.select('text')
153                     .text(d => d.refugees)
154                     .call(update => update.transition(t)
155                         .attr('y', d => yScale(d.country)
                               + yScale.bandwidth()/2)
156                         .attr('x', d => {
157                             const scaleValue = xScale(
                                   d.refugees);
158                             return (scaleValue - 60 > 0) ?
                                   scaleValue - 10 : 60;
159                         }));
160             },
161             // This describes the behavior of the update
                   sub-selection of the general update
                   pattern.
162             // Applicable elements are simply removed.
                   This is also the default behavior and doesn
                   't need specification.
163             exit => exit.remove()
164         );
165 };
166
167 /**
168  * This section tries to subscribe to the
        country-data-service for data updates.
169  * The diagram will not work without the
        country-data-service.
170  */
171 try {
172     parent.registerCountryDiagramRenderCallback(render);
173     console.log('Could successfully subscribe to the
          country-data-service for data updates.');
```

```
174 } catch (e) {
175     console.log('Could not subscribe to the
            country-data-service for data updates. ' +
176         'Data is loaded directly.');
177
178     loadCountryData(render)
179 }
```

A.2 Donut Chart - JavaScript

```
1  /**
2   * This script was created as part of a bachelor thesis.
3   * The results can be found here: https://github.com/
        StyxOo/styxoo.github.io
4   * Author: Luis Rothenhäusler
5   * Last edit: 25th August 2022
6   *
7   * This file contains the JavaScript implementation of the
        donut-chart.
8   */
9
10 /**
11  * In this first section, some data independent constants
        are defined.
12  */
13
14 // This creates a reference to the SVG container on the
       HTML page. This will contain the whole diagram.
15 const svg = d3.select('#mainFrame')
16     .attr('height', innerHeight)
17     .attr('width', innerWidth);
18
19 // The margin definition for the diagram. The content is
       padded from the sides using the margins.
20 const margin = {
21     top: 20,
22     right: 20,
23     bottom: 20,
24     left: 20
25 };
26
27 // contentWidth and contentHeight store the available
       coordinate space for the content of the diagram.
28 const contentWidth = innerWidth - margin.left -
       margin.right;
29 const contentHeight = innerHeight - margin.top -
       margin.bottom;
30
```

```
31 // The radius of the donut is set to use as much space as
       available.
32 const radius = d3.min([contentHeight/2, contentWidth/2]);
33
34 /**
35  * This section defines the hierarchy of the diagram.
36  * This makes later selections and debugging in the
       browser inspector easier.
37  */
38 const diagramGroup = svg.append('g')
39     .attr('transform', `translate(${margin.left},${
          margin.top})`);
40
41 const contentParentGroup = diagramGroup.append('g')
42     .attr('id', 'content')
43     .attr('transform', `translate(${contentWidth/2},${
          contentHeight/2})`);
44
45 const diagramParentGroup = contentParentGroup.append('g')
46     .attr('id', 'diagram');
47
48 /**
49  * This section adds all the necessary text fields for
       showing the total refugees,
50  * as well as the refugees for the currently hovered over
       country.
51  */
52 // The following adds the text to display when no section
       is hovered over.
53 const totalTextGroup = contentParentGroup.append('text')
54     .attr('id', 'totalTextGroup')
55     .attr('display', true);
56
57 totalTextGroup.append('tspan')
58     .text('So far a total of')
59     .attr('dy', '-2.3em')
60     .attr('x', 0);
61
62 const totalTextSpan = totalTextGroup.append('tspan')
63     .text('TotalNumberHere')
64     .attr('class', 'important')
65     .attr('dy', '1.3em')
66     .attr('x', 0);
67
68 totalTextGroup.append('tspan')
69     .text('refugees have fled')
70     .attr('dy', '1.1em')
71     .attr('x', 0);
72
```

```
73  totalTextGroup.append('tspan')
74      .text('Ukraine')
75      .attr('class', 'important')
76      .attr('dy', '1.3em')
77      .attr('x', 0);
78
79  // The following adds the text to display if a section is
        hovered over.
80  const currentTextGroup = contentParentGroup.append('text')
81      .attr('id', 'currentTextGroup')
82      .attr('display', 'none');
83
84  const currentNumberTextSpan = currentTextGroup.append('
        tspan')
85      .text('CurrentNumberHere')
86      .attr('class', 'important')
87      .attr('dy', '-0.5em')
88      .attr('x', 0);
89
90  currentTextGroup.append('tspan')
91      .text('refugees have fled to')
92      .attr('dy', '1.1em')
93      .attr('x', 0);
94
95  const currentCountryTextSpan = currentTextGroup.append('
        tspan')
96      .text('DestinationCountryHere')
97      .attr('class', 'important')
98      .attr('dy', '1.3em')
99      .attr('x', 0);
100
101 /**
102  * This section defines the color scale used to color
        elements according to their country.
103  * It can be defined here, as it is independent of the
        data
104  */
105 const colors = d3.scaleOrdinal(d3.schemeDark2);
106
107 /**
108  * The render function is defined here.
109  * It is called to initially draw the diagram, as well
        every time the data changes and the diagram should
        update.
110  */
111 const render = data => {
112     console.log('Rendering pie chart');
113
114     // The total refugees are calculated and the according
```

```
        center text is updated.
115     const totalRefugees = d3.sum(data, d => d.refugees);
116     totalTextSpan.text(`${totalRefugees}`);
117
118     /**
119      * This section defines all helper functions and
            constants necessary for creating the diagram.
120      */
121     // The pie function generate start and end angles for
            each data-point.
122     const pie = d3.pie()
123         .value(d => d.refugees)
124         .padAngle(0.015)(data);
125
126     // The arc functions is used to convert pie sections
            into SVG paths.
127     const arc = d3.arc()
128         .innerRadius(radius * .6)
129         .outerRadius(radius);
130
131     // The core of the donut animation is defined here.
132     const animate = (nodes, index, d, i, j) => {
133         nodes[index].previousStartAngle = d.startAngle;
134         nodes[index].previousEndAngle = d.endAngle;
135
136         return time => {
137             d.startAngle = i(time);
138             d.endAngle = j(time);
139             return arc(d);
140         };
141     };
142
143     /**
144      * The following defines the transition which is used
            for all animations.
145      */
146     const t = svg.transition()
147         .duration(1500);
148
149     /**
150      * This is where the actual content of the diagram is
            drawn.
151      * Therefore, a data-join is created and the behavior
            of the general update pattern is specified.
152      */
153     diagramParentGroup.selectAll('g .arc').data(pie, d =>
            {return d.data.country})
154         .join(
155             // This describes the behavior of the enter
```

```
                sub-selection of the general update
                pattern.
156             enter => {
157                 // A group and a child path element are
                        added and styled.
158                 enter.append('g')
159                     .attr('class', 'arc')
160                     .append('path')
161                     .attr('fill', d => colors(d.data))
162                     .call(enter => enter.transition(t)
163                         // The initial animation for the
                                donut pieces is specified here.
164                         .attrTween('d', (section, index,
                                nodes) => {
165                             const interpolateStartAngle =
                                    d3.interpolate(0,
                                    section.startAngle);
166                             const interpolateEndAngle =
                                    d3.interpolate(0,
                                    section.endAngle);
167
168                             return animate(nodes, index,
                                    section,
                                    interpolateStartAngle,
                                    interpolateEndAngle);
169                         }))
170                     // The behaviour on the mouseover
                            event is specified to update the
                            center text accordingly.
171                     .on('mouseover', (e, d) => {
172                         currentNumberTextSpan.text(
                                d.data.refugees)
173                         currentCountryTextSpan.text(
                                d.data.country)
174                         currentTextGroup.attr('display', '
                                true')
175                         totalTextGroup.attr('display', '
                                none')
176                     })
177                     // The behaviour on the mouseover
                            event is specified to update the
                            center text accordingly.
178                     .on('mouseout', () => {
179                         currentTextGroup.attr('display', '
                                none')
180                         totalTextGroup.attr('display', '
                                true')
181                     });
182             },
```

```
183             // This describes the behavior of the update
                   sub-selection of the general update
                   pattern.
184             update => {
185                 update.select('path')
186                     .call(update => update.transition(t)
187                         // The update animation for the
                               donut pieces is specified here.
188                         .attrTween('d', (section, index,
                               nodes) => {
189                             const interpolateStartAngle =
                                   d3.interpolate(nodes[index
                                   ].previousStartAngle,
                                   section.startAngle);
190                             const interpolateEndAngle =
                                   d3.interpolate(nodes[index
                                   ].previousEndAngle,
                                   section.endAngle);
191
192                             return animate(nodes, index,
                                   section,
                                   interpolateStartAngle,
                                   interpolateEndAngle);
193                         }));
194             }
195         );
196 };
197
198 /**
199  * This section tries to subscribe to the
        country-data-service for data updates.
200  * The diagram will not work without the
        country-data-service.
201  */
202 try {
203     parent.registerCountryDiagramRenderCallback(render);
204     console.log('Could successfully subscribe to the
          country-data-service for data updates.');
205 } catch (e) {
206     console.log('Could not subscribe to the
          country-data-service for data updates. ' +
207         'Data is loaded directly.');
208
209     loadCountryData(render)
210 }
```

A.3 Tree Map - JavaScript

```
1  /**
2   * This script was created as part of a bachelor thesis.
3   * The results can be found here: https://github.com/
        StyxOo/styxoo.github.io
4   * Author: Luis Rothenhäusler
5   * Last edit: 25th August 2022
6   *
7   * This file contains the JavaScript implementation of the
        tree-map.
8   */
9
10 /**
11  * In this first section, some data independent constants
        are defined.
12  */
13 // This creates a reference to the SVG container on the
       HTML page. This will contain the whole diagram.
14 const svg = d3.select('#mainFrame')
15     .attr('height', innerHeight)
16     .attr('width', innerWidth);
17
18 // The margin definition for the diagram. The content is
       padded from the sides using the margins.
19 const margin = {
20     top: 20,
21     right: 20,
22     bottom: 20,
23     left: 20
24 };
25
26 // contentWidth and contentHeight store the available
       coordinate space for the content of the diagram.
27 const contentWidth = innerWidth - margin.left -
       margin.right;
28 const contentHeight = innerHeight - margin.top -
       margin.bottom;
29
30 /**
31  * This section defines the hierarchy of the diagram.
32  * This makes later selections and debugging in the
        browser inspector easier.
33  */
34 const diagramGroup = svg.append('g')
35     .attr('transform', `translate(${margin.left},${
          margin.top})`);
36
37 const contentParentGroup = diagramGroup.append('g')
38     .attr('id', 'content');
39
```

```
40 const treemapParentGroup = contentParentGroup.append('g')
41     .attr('id', 'treeMapParent');
42
43 /**
44  * This draws a background rectangle for the tree-map.
45  */
46 contentParentGroup.append('rect')
47     .attr('id', 'contentBackground')
48     .attr('x', 0)
49     .attr('y', 0)
50     .attr('width', contentWidth)
51     .attr('height', contentHeight)
52     .attr('fill', 'none');
53
54 /**
55  * This section creates and hides the tooltip,
56  * which is used to display information about the
       currently hovered over country.
57  */
58 // A secondary small SVG created and added to the body.
59 const tooltip = d3.select("body")
60     .append('svg')
61     .attr('height', 50)
62     .attr('width', 400)
63     .attr('id', 'tooltip')
64     .style('position', 'absolute')
65     .style('z-index', 10)
66     .classed('hidden', true);
67
68 // The tooltips background is styled here.
69 const background = tooltip.append('rect')
70     .attr('height', 50)
71     .attr('width', 100)
72     .attr('rx', 10)
73     .attr('ry', 10);
74
75 // The tooltips text field is created here.
76 const tooltipText = tooltip.append('text')
77     .attr('y', 20)
78     .attr('x', 5)
79     .text('Some text');
80
81 /**
82  * This section defines the color scale used to color
       elements according to their country.
83  * It can be defined here, as it is independent of the
       data
84  */
85 const colors = d3.scaleOrdinal(d3.schemeDark2);
```

```
86
87  /**
88   * The render function is defined here.
89   * It is called to initially draw the diagram, as well
       every time the data changes and the diagram should
       update.
90   */
91  const render = data => {
92      console.log('Rendering tree map');
93
94      /**
95       * This section is responsible for the required
           preprocessing of the data,
96       * as the tree-map is intended to work with
           hierarchical data.
97       */
98      // A dummy parent object is created here. It's
         necessary for simulating hierarchical data.
99      const parent = {
100         "country": "Dummy Parent",
101         "refugees": 0
102     };
103
104     // Adds the dummy parent to the data.
105     data.push(parent);
106
107     // This turns the data provided into a hierarchical
        data structure.
108     const root = d3.stratify()
109         .id(d => {return d.country})
110         .parentId((d) => {
111             if (d.country === 'Dummy Parent')
112                 return undefined
113             else
114                 return 'Dummy Parent'
115         })(data);
116
117     // The dummy parent is removed from the data again, as
            it is no longer needed.
118     data.pop();
119
120     // The total amount of refugees is calculated here.
121     root.sum(d => {return d.refugees});
122
123     // The data is converted into leaves used to draw the
        tree-map.
124     d3.treemap()
125         .size([contentWidth, contentHeight])
126         .padding(4)(root);
```

```
127
128         /**
129          * The following defines the transition which is used
                for all animations.
130          */
131         const t = svg.transition()
132             .duration(1500);
133
134         /**
135          * This is where the actual content of the diagram is
                drawn.
136          * Therefore, a data-join is created and the behavior
                of the general update pattern is specified.
137          */
138         treemapParentGroup.selectAll('rect').data(root.leaves
               (), d => {return d.data.country})
139             .join(
140                 // This describes the behavior of the enter
                       sub-selection of the general update
                       pattern.
141                 enter => {
142                     // A rect is added for a leaf of the
                           tree-map. It is positioned, styled and
                           animated.
143                     enter.append('rect')
144                         .attr('x', 0)
145                         .attr('y', `${contentHeight}`)
146                         .attr('width', 0)
147                         .attr('height', 0)
148                         .attr('fill', d => colors(d.data))
149                         .call(enter => enter.transition(t)
150                             .attr('x', d => { return d.x0; })
151                             .attr('y', d => { return d.y0; })
152                             .attr('width', d => { return d.x1
                                   - d.x0; })
153                             .attr('height', d => { return d.y1
                                    - d.y0; }))
154                         // The mouseover event is specified to
                               show the tooltip and update its
                               text accordingly.
155                         .on('mouseover', (e, d) => {
156                             tooltip.classed('hidden', false)
157                             tooltipText.text(`${d.data.country
                                   }\nRefugees : ${d.data.refugees
                                   }`)
158                             const textWidth = tooltipText.node
                                   ().getBBox().width
159                             background.attr('width', textWidth
                                    + 10)
```

```
160                         })
161                         // The mousemove event is specified to
                                update the tooltips position
                               accordingly.
162                         .on('mousemove', e => {
163                             const position = d3.pointer(e)
164                             tooltip.style("top", (position
                                    [1]+0)+"px");
165                             if (position[0] > contentWidth/2)
                                    {
166                                 const rect = tooltip.select('
                                        rect')
167                                 const width = rect.attr('width
                                        ')
168                                 tooltip.style("left", (
                                        position[0] - width + 10) +
                                         "px");
169                             } else {
170                                 tooltip.style("left", (
                                        position[0] + 35) + "px");
171                             }
172                         })
173                         // The mouseout event is specified to
                                hide the tooltip.
174                         .on('mouseout', () => {
175                             tooltip.classed('hidden', true)
176                         })
177
178                 },
179                 // This describes the behavior of the update
                        sub-selection of the general update
                        pattern.
180                 update => {
181                     // The applicable rects are animated to
                            resized and repositioned.
182                     update.call(update => update.transition(t)
183                             .attr('x', d => { return d.x0; })
184                             .attr('y', d => { return d.y0; })
185                             .attr('width', d => { return d.x1
                                    - d.x0; })
186                             .attr('height', d => { return d.y1
                                    - d.y0; }))
187                 }
188             );
189 };
190
191 /**
192  * This section tries to subscribe to the
        country-data-service for data updates.
```

```
193  * The diagram will not work without the
         country-data-service.
194  */
195 try {
196     parent.registerCountryDiagramRenderCallback(render);
197     console.log('Could successfully subscribe to the
            country-data-service for data updates.');
198 } catch (e) {
199     console.log('Could not subscribe to the
            country-data-service for data updates. ' +
200         'Data is loaded directly.');
201
202     loadCountryData(render)
203 }
```

A.4 Sankey Graph - JavaScript

```
1  /**
2   * This script was created as part of a bachelor thesis.
3   * The results can be found here: https://github.com/
        StyxOo/styxoo.github.io
4   * Author: Luis Rothenhäusler
5   * Last edit: 25th August 2022
6   *
7   * This file contains the JavaScript implementation of the
        sankey-diagram.
8   */
9
10 /**
11  * In this first section, some data independent constants
        are defined.
12  */
13 // This creates a reference to the SVG container on the
       HTML page. This will contain the whole diagram.
14 const svg = d3.select('#mainFrame')
15     .attr('height', innerHeight)
16     .attr('width', innerWidth);
17
18 // The margin definition for the diagram. The content is
       padded from the sides using the margins.
19 const margin = {
20     top: 20,
21     right: 20,
22     bottom: 20,
23     left: 20
24 };
25
26 // contentWidth and contentHeight store the available
```

```
      coordinate space for the content of the diagram.
27  const contentWidth = innerWidth - margin.left -
      margin.right;
28  const contentHeight = innerHeight - margin.top -
      margin.bottom;
29
30  /**
31   * This section defines the hierarchy of the diagram.
32   * This makes later selections and debugging in the
       browser inspector easier.
33   */
34  const diagramGroup = svg.append('g')
35      .attr('transform', `translate(${margin.left},${
          margin.top})`);
36
37  const contentParentGroup = diagramGroup.append('g')
38      .attr('id', 'content');
39
40  const linksParentGroup = contentParentGroup.append('g')
41      .attr('id', 'links');
42
43  const countriesParentGroup = contentParentGroup.append('g'
      )
44      .attr('id', 'countries');
45
46
47  /**
48   * This section defines the color scale used to color
       elements according to their country.
49   * It can be defined here, as it is independent of the
       data
50   */
51  const colors = d3.scaleOrdinal(d3.schemeDark2);
52
53  /**
54   * The render function is defined here.
55   * It is called to initially draw the diagram, as well
       every time the data changes and the diagram should
       update.
56   */
57  const render = data => {
58      console.log('Rendering sankey');
59
60      /**
61       * This section is responsible for the required
           preprocessing of the data,
62       * as the sankey-graph is intended to work with
           hierarchical data.
63       */
```

```
64      // The nodes and links necessary for creating a sankey
            diagram are created from the provided data.
65      const nodes = [{name: 'Ukraine'}];
66      const links = [];
67      for (const d of data) {
68          nodes.push({name: d.country});
69          links.push({source: 'Ukraine', target: d.country,
                value: d.refugees});
70      }
71
72      // The sankey function adds information to the data
            allowing the nodes and links to be drawn.
73      d3.sankey()
74          .nodeId(d => d.name)
75          .nodeAlign(d3.sankeyJustify)
76          .size([contentWidth, contentHeight])({nodes, links
                });
77
78      /**
79       * The following defines the transition which is used
             for all animations.
80       */
81      const t = svg.transition()
82          .duration(1500);
83
84      /**
85       * This is where the actual content of the diagram is
             drawn.
86       * Therefore, two data-joins are created and their
             behavior of the general update pattern is
             specified.
87       */
88      // The first data join is used to draw the nodes of
            the sankey graph.
89      countriesParentGroup.selectAll('g .country').data(
            nodes, d => {return d.name})
90          .join(
91              // This describes the behavior of the enter
                    sub-selection of the general update
                    pattern.
92              enter => {
93                  // A new group element is added for each
                        country
94                  const country = enter.append('g')
95                      .attr('class', 'country');
96
97                  // The rect representing the country is
                        created, positioned, sized, styled and
                        animated.
```

```
98                  country.append('rect')
99                      .attr('x', 0)
100                     .attr('y', d => d.y0)
101                     .attr('width', 0)
102                     .attr('height', d => d.y1 - d.y0)
103                     .attr('fill', d => {
104                         // To be consistent with the other
                               diagrams, the Ukraine does not
                               query the color scale.
105                         if (d.name === 'Ukraine') {
106                             return '#0057B8';
107                         } else {
108                             return colors(d);
109                         }
110                     })
111                     .call(enter => enter.transition(t)
112                         .attr('x', d => d.x0)
113                         .attr('width', d => d.x1 - d.x0));
114
115                 // The label text is added, provided with
                       the appropriate text, positioned,
                       styled and animated.
116                 country.append('text')
117                     .text(d => `${d.name}: ${d.value}`)
118                     .attr('x', 0)
119                     .attr('y', d => (d.y0 + d.y1)/2 + 5)
120                     .attr('text-anchor', d => d.x0 <
                           contentWidth/2? 'start' : 'end')
121                     .attr('opacity', '0%')
122                     .call(enter => enter.transition(t)
123                         .attr('x', d => d.x0 <
                               contentWidth/2? d.x1+10 :
                               d.x0-10))
124                     .call(enter => enter.transition(t).
                           delay(100)
125                         .attr('opacity', '100%'));
126             },
127             // This describes the behavior of the update
                   sub-selection of the general update
                   pattern.
128             update => {
129                 // The countries' rectangle is animated to
                       resize and reposition
130                 update.select('rect').call(update =>
                       update.transition(t)
131                     .attr('height', d => d.y1 - d.y0)
132                     .attr('y', d => d.y0));
133
134                 // The countries' text label is animated
```

```
                    to reposition and update its value.
135                 update.select('text').call(update =>
                        update.transition(t)
136                         .attr('y', d => (d.y0 + d.y1)/2 + 5))
137                         .text(d => `${d.name}: ${d.value}`);
138             }
139         )
140
141     // The second data-join is used to draw the links
            between the nodes.
142     linksParentGroup.selectAll('path').data(links, d => {
            return [d.source.name, d.target.name]})
143         .join(
144             // This describes the behavior of the enter
                sub-selection of the general update
                pattern.
145             enter => {
146                 // A path is added for each link. It is
                    also styled and animated.
147                 enter.append('path')
148                     // D3 constructs the appropriate SVG
                        path from the information available
                         in the link.
149                     .attr('d', d3.sankeyLinkHorizontal())
150                     .attr('stroke', d => colors(d.target))
151                     // The stroke-width represents the
                        width of the link and depends on
                        the data.
152                     .attr('stroke-width', ({width}) =>
                            Math.max(1, width))
153                     .attr('fill', d => colors(d.target))
154                     .attr('opacity', 0)
155                     .call(enter => enter.transition(t).
                            delay(500)
156                         .attr('opacity', '50%'));
157             },
158             // This describes the behavior of the update
                sub-selection of the general update
                pattern.
159             update => {
160                 // The SVG paths are recalculated and the
                    width adjusted.
161                 update.call(update => update.transition(t)
162                     .attr('d', d3.sankeyLinkHorizontal())
163                     .attr('stroke-width', ({width}) =>
                            Math.max(1, width)));
164             }
165         );
166 };
```

```
167
168 /**
169  * This section tries to subscribe to the
        country-data-service for data updates.
170  * The diagram will not work without the
        country-data-service.
171  */
172 try {
173     parent.registerCountryDiagramRenderCallback(render);
174     console.log('Could successfully subscribe to the
          country-data-service for data updates.');
175 } catch (e) {
176     console.log('Could not subscribe to the
          country-data-service for data updates. ' +
177         'Data is loaded directly.');
178
179     loadCountryData(render)
180 }
```

A.5 Circle Graph - JavaScript

```
1  /**
2   * This script was created as part of a bachelor thesis.
3   * The results can be found here: https://github.com/
        StyxOo/styxoo.github.io
4   * Author: Luis Rothenhäusler
5   * Last edit: 25th August 2022
6   *
7   * This file contains the JavaScript implementation of the
        sankey-diagram.
8   */
9
10 /**
11  * In this first section, some data independent constants
        are defined.
12  */
13 // This creates a reference to the SVG container on the
      HTML page. This will contain the whole diagram.
14 const svg = d3.select('#mainFrame')
15     .attr('height', innerHeight)
16     .attr('width', innerWidth);
17
18 // The margin definition for the diagram. The content is
      padded from the sides using the margins.
19 const margin = {
20     top: 20,
21     right: 20,
22     bottom: 30,
```

```
23      left: 20
24  };
25
26  // contentWidth and contentHeight store the available
        coordinate space for the content of the diagram.
27  const contentWidth = innerWidth - margin.left -
        margin.right;
28  const contentHeight = innerHeight - margin.top -
        margin.bottom;
29
30  // The factor by which the legend values should be divided
        for easier readability
31  const legendScaleFactor = 100000;
32
33  /**
34   * This section defines the hierarchy of the diagram.
35   * This makes later selections and debugging in the
        browser inspector easier.
36   */
37  const diagramGroup = svg.append('g')
38      .attr('transform', `translate(${margin.left},${
            margin.top})`);
39
40  const legendParentGroup = diagramGroup.append('g')
41      .attr('id', 'legend');
42
43  const contentParentGroup = diagramGroup.append('g')
44      .attr('id', 'content');
45
46  // This text is added to inform about the scaling of the
        legend.
47  legendParentGroup.append('text')
48      .text('* scale in 100,000 refugees')
49      .attr('class', 'description')
50      .attr('x', contentWidth)
51      .attr('y', contentHeight);
52
53  /**
54   * This section defines a helper functions necessary for
        creating the diagram.
55   */
56  // This function converts a JavaScript date object into a
        string of the style Feb-07 or Jun-15.
57  const dateToDisplay = date => {
58      const day = date.getDate();
59      const month = date.toLocaleString('default', { month:
            'short' });
60
61      let dayString = day;
```

```
62      if (day < 10) {
63          dayString = '0' + dayString;
64      }
65      return [month, dayString].join('-');
66  }
67
68  /**
69   * The render function is defined here.
70   * It is called to initially draw the diagram, as well
         every time the data changes and the diagram should
         update.
71   */
72  const render = (data, time01 = 0) => {
73      console.log('Rendering circle chart');
74
75      /**
76       * The following defines the transition which is used
             for all animations.
77       */
78      const t = svg.transition()
79          .duration(1500);
80
81      /**
82       * Here all the required scales, which are dependent
             on the data, are defined.
83       */
84      // The time scale is used to convert between the time
           value of [0, 1], to the actual date.
85      const timeScale = d3.scaleQuantize()
86          .domain([0, 1])  // Original range of values
87          .range(data.map(d => d.date));
88
89      // The radius scale provides the appropriate radius
           for a given number of refugees.
90      const radiusScale = d3.scaleSqrt()
91          .domain([0, d3.max(data, d => d.refugees)])
92          .range([0, contentHeight / 2]);
93
94
95      /**
96       * This section is responsible for drawing the
             background size legend of the diagram.
97       * This is achieved using a data-join and specifying
             the general-update-behavior.
98       */
99      // The ticks are extracted from the time scale.
100     const ticks = radiusScale.ticks(10).filter(d => d !==
          0);
101     let tickData = [];
```

```
102     for (let i = 0; i < ticks.length; i++) {
103         tickData.push({ id: i, value: ticks[i] });
104     }
105     // A data-join is responsible for drawing the ticks.
106     legendParentGroup.selectAll('g').data(tickData, d => {
            return d.id })
107         .join(
108             // This describes the behavior of the enter
                sub-selection of the general update
                pattern.
109             enter => {
110                 // A group is added and styled for each
                    tick.
111                 const tick = enter.append('g')
112                     .attr('opacity', '0%')
113                     .call(enter => enter.transition(t)
114                         .attr('opacity', '75%'));
115
116                 // Each tick is provided a circle, which
                    is positioned and sized appropriately
117                 tick.append('circle')
118                     .attr('cx', contentWidth / 2)
119                     .attr('cy', d => contentHeight -
                        radiusScale(d.value))
120                     .attr('r', d => radiusScale(d.value))
121                     .attr('class', 'legend');
122
123                 // Each tick also provided with a text
                    label to show the quantity.
124                 tick.append('text')
125                     .text((d, i) => {
126                         let value = d.value /
                            legendScaleFactor;
127                         if (i === tickData.length - 1) {
128                             value += '*';
129                         }
130                         return value;
131                     })
132                     .attr('dy', '-0.1em')
133                     .attr('x', contentWidth / 2)
134                     .attr('y', d => contentHeight - 2 *
                        radiusScale(d.value));
135             },
136             // This describes the behavior of the update
                sub-selection of the general update
                pattern.
137             update => {
138                 // The circle for each tick is animated to
                        change position and size accordingly.
```

```
139                 update.select('circle').call(update =>
                        update.transition(t)
140                     .attr('cy', d => contentHeight -
                            radiusScale(d.value))
141                     .attr('r', d => radiusScale(d.value)))
                            ;
142
143                 // The text label value is updated and its
                        position change animated.
144                 update.select('text')
145                     .text((d, i) => {
146                         let value = d.value /
                                legendScaleFactor;
147                         if (i === tickData.length - 1) {
148                             value += '*';
149                         }
150                         return value;
151                     })
152                     .call(update => update.transition(t)
153                         .attr('y', d => contentHeight - 2
                                * radiusScale(d.value)));
154             }
155             ,
156             // This describes the behavior of the exit
                    sub-selection of the general update
                    pattern.
157             exit => {
158                 // Each element is faded out using
                        animations, before being removed.
159                 exit.call(exit => exit.transition(t)
160                     .attr('opacity', '0%'))
161                     .remove();
162             }
163         )
164
165
166     /**
167      * This is where the actual content of the diagram is
            drawn.
168      * Therefore, a data-join is created and the behavior
            of the general update pattern is specified.
169      */
170     // The current date is found using the time scale.
171     const unixTime = timeScale(time01)
172     const datum = data.find(d => d.date === unixTime)
173     contentParentGroup.selectAll('g .content').data([0],
        () => [0])
174         .join(
175             // This describes the behavior of the enter
```

```
                sub-selection of the general update
                pattern.
176             // This behavior is used only for the first
                time the diagram is drawn.
177             enter => {
178                 // A group element is added
179                 const content = enter.append('g')
180                     .attr('class', 'content')
181
182                 // A circle is added, positioned, sized,
                    styled and animated accordingly.
183                 content.append('circle')
184                     .attr('cx', contentWidth / 2)
185                     .attr('cy', contentHeight)
186                     .attr('r', 0)
187                     .attr('fill', 'red')
188                     .attr('opacity', '50%')
189                     .call(enter => enter.transition(t)
190                         .attr('cy', () => contentHeight -
                            radiusScale(datum.refugees))
191                         .attr('r', () => radiusScale(
                            datum.refugees)))
192
193                 // A text is added to the bottom and
                    provided with the correct value of
                    refugees.
194                 content.append('text')
195                     .attr('x', contentWidth / 2)
196                     .attr('y', contentHeight + 17)
197                     .text(datum.refugees + ' refugees by '
                        + dateToDisplay(datum.date));
198
199             },
200             // This describes the behavior of the update
                sub-selection of the general update
                pattern.
201             update => {
202                 // The circle is animated to update in
                    position and size.
203                 update.select('circle')
204                     .call(update => update.transition(t)
205                         .attr('cy', () => contentHeight -
                            radiusScale(datum.refugees))
206                         .attr('r', () => radiusScale(
                            datum.refugees)));
207
208                 // The text value is updated
209                 update.select('text')
210                     .text(datum.refugees + ' refugees by '
```

```
                                + dateToDisplay(datum.date));
211             }
212         )
213 };
214
215 /**
216  * This section tries to subscribe to the
        daily-data-service for data updates.
217  * The diagram will not work without the
        daily-data-service.
218  */
219 try {
220     parent.registerDailyDiagramRenderCallback(render);
221     console.log('Could successfully subscribe to the
          daily-data-service for data updates.');
222 } catch (e) {
223     console.log('Could not subscribe to the
          daily-data-service for data updates. ' +
224         'Data is loaded directly.');
225
226     loadDailyData(render)
227 }
```

A.6 Area Graph - JavaScript

```
1  /**
2   * This script was created as part of a bachelor thesis.
3   * The results can be found here: https://github.com/
        StyxOo/styxoo.github.io
4   * Author: Luis Rothenhäusler
5   * Last edit: 25th August 2022
6   *
7   * This file contains the JavaScript implementation of the
        sankey-diagram.
8   */
9
10 /**
11  * In this first section, some data independent constants
        are defined.
12  */
13 // This creates a reference to the SVG container on the
       HTML page. This will contain the whole diagram.
14 const svg = d3.select('#mainFrame')
15     .attr('height', innerHeight)
16     .attr('width', innerWidth);
17
18 // The margin definition for the diagram. The content is
       padded from the sides using the margins.
```

```
19 const margin = {
20     top: 20,
21     right: 20,
22     bottom: 30,
23     left: 70
24 }
25
26 // contentWidth and contentHeight store the available
      coordinate space for the content of the diagram.
27 const contentWidth = innerWidth - margin.left -
      margin.right
28 const contentHeight = innerHeight - margin.top -
      margin.bottom
29
30 /**
31  * This section defines the hierarchy of the diagram.
32  * This makes later selections and debugging in the
       browser inspector easier.
33  */
34 const diagramGroup = svg.append('g')
35     .attr('transform', `translate(${margin.left},${
          margin.top})`);
36
37 const xAxisParentGroup = diagramGroup.append('g')
38     .attr('id', 'xAxis')
39
40 const yAxisParentGroup = diagramGroup.append('g')
41     .attr('id', 'yAxis')
42
43 const contentParentGroup = diagramGroup.append('g')
44     .attr('id', 'content')
45
46 contentParentGroup.append('g')
47     .attr('id', 'dateLine')
48
49 /**
50  * This section defines a helper functions necessary for
       creating the diagram.
51  */
52 // This function converts a JavaScript date object into a
      string of the style Feb-07 or Jun-15.
53 const dateToDisplay = date => {
54     const day = date.getDate();
55     const month = date.toLocaleString('default', { month:
          'short' });
56
57     let dayString = day;
58     if (day < 10) {
59         dayString = '0' + dayString;
```

```
60      }
61      return [month, dayString].join('-');
62  }
63
64  /**
65   * The render function is defined here.
66   * It is called to initially draw the diagram, as well
         every time the data changes and the diagram should
         update.
67   */
68  const render = (data, time01 = 0) => {
69      console.log('Rendering circle chart')
70
71      /**
72       * The following defines the transition which is used
             for all animations.
73       */
74      const t = svg.transition()
75          .duration(1500);
76
77      /**
78       * Here all the required scales, which are dependent
             on the data, are defined.
79       */
80      // The time scale is used to convert between the time
         value of [0, 1], to the actual date.
81      const timeScale = d3.scaleQuantize()
82          .domain([0, 1])
83          .range(data.map(d => d.date))
84
85      // The y scale is used to calculate a y coordinate
         from a given refugee number.
86      const yScale = d3.scaleLinear()
87          .domain([0, d3.max(data, d => d.refugees)])
88          .range([contentHeight, 0])
89          .nice();
90
91      // The x scale is used to calculate a x coordinate
         from a given date.
92      const xScale = d3.scaleBand()
93          .domain(data.map(d => d.date))
94          .range([0, contentWidth])
95          .padding(0.2);
96
97      // this is used to offset the calculated x positions,
         so they align to the center of a date-line.
98      // As the scale is a scaleBand, they would otherwise
         be offset slightly to the left.
99      const xScaleWithOffset = d => {
```

```
100             return xScale(d) + xScale.bandwidth() / 2
101         }
102
103         /**
104          * This section is responsible for creating the x and
                 y axes of the area-graph.
105          */
106         // The y-axis is created from the scale. Additionally,
                 the tick size is specified to cover the whole
                background.
107         const yAxis = d3.axisLeft(yScale)
108             .tickSize(-contentWidth)
109
110         // The y-axis is added to the diagram, but the domain
                lines are removed.
111         yAxisParentGroup.call(yAxis)
112             .selectAll('.domain')
113             .remove();
114
115         // This defines the function responsible for
                formatting the x-axis ticks.
116         const xAxisTickFormat = date =>
117             dateToDisplay(date)
118
119         // This specifies the modulo value to be used, so that
                 the resulting axis has 15 ticks.
120         const tickModulo = Math.floor(data.length / 15)
121
122         // This creates the x-axis. The values are filtered so
                 only 15 values appear.
123         const xAxis = d3.axisBottom(xScale)
124             .tickFormat(xAxisTickFormat)
125             .tickSize(-contentHeight)
126             .tickValues(xScale.domain().filter((d, i) => {
                    return !(i % tickModulo) }))
127
128         // The x-axis is added to the diagram, positioned to
                the bottom and has its domain line removed.
129         xAxisParentGroup.call(xAxis)
130             .attr('transform', `translate(0,${contentHeight})
                    `)
131             .select('.domain')
132             .remove();
133
134         // All x-axis labels are moved a small bit further
                towards the bottom.
135         xAxisParentGroup.selectAll('text').attr('transform', `
                translate(0,${10})`)
136
```

```
137
138     /**
139      * This section defines more helper functions
             necessary for creating the diagram.
140      * As these require the scales, they are defined here.
141      */
142
143     // This function creates a SVG line for a dataset,
           where each points x and y values are calculated as
           defined.
144     const line = d3.line()
145         .x(d => xScaleWithOffset(d.date))
146         .y(d => yScale(d.refugees));
147
148     // This function creates a SVG line enclosing an area
           for a dataset.
149     // Each x, as well as the higher and lower y positions
            values are calculated as defined.
150     const area = d3.area()
151         .x(d => xScaleWithOffset(d.date))
152         .y0(contentHeight)
153         .y1(d => yScale(d.refugees));
154
155     /**
156      * This is where the actual content of the diagram is
             drawn. This consists of an area and a line atop.
157      * Therefore, a data-join is created and the behavior
             of the general update pattern is specified.
158      * The enter behavior is only executed once, as the
             diagram is loaded.
159      */
160     contentParentGroup.selectAll('g .areaGroup').data([0],
         () => [0])
161         .join(
162             // This describes the behavior of the enter
                   sub-selection of the general update
                   pattern.
163             enter => {
164                 // A group is added for hierarchical
                       purposes.
165                 const areaParent = enter.append('g')
166                     .attr('class', 'areaGroup')
167
168                 // The area is drawn in the diagram.
169                 areaParent.append('path')
170                     .attr('class', 'area')
171                     .attr('d', area(data))
172
173                 // The top-line is drawn above the area in
```

```
                    the diagram.
174                 areaParent.append('path')
175                     .attr('class', 'topLine')
176                     .attr('d', line(data))
177
178             },
179             // This describes the behavior of the update
                    sub-selection of the general update
                    pattern.
180             update => {
181                 // The area is recreated and transitions
                        to the new path.
182                 update.select('.area')
183                     .call(update => update.transition(t)
184                         .attr('d', area(data)))
185
186                 // The top-line is recreated and
                        transitions to the new path.
187                 update.select('.topLine')
188                     .call(update => update.transition(t)
189                         .attr('d', line(data)))
190             });
191
192     /**
193      * This section is responsible for the date-line. It
            is drawn and updated using a data-join.
194      * The enter behavior is only executed once, as the
            diagram is initially drawn.
195      */
196     // Using the time scale, the current date is found.
197     const unixTime = timeScale(time01)
198     const datum = data.find(d => d.date === unixTime)
199
200     contentParentGroup.selectAll('g .dateLine').data([0],
            () => [0])
201         .join(
202             // This describes the behavior of the enter
                    sub-selection of the general update
                    pattern.
203             enter => {
204                 // A group element is added for the
                        date-line.
205                 const dateLine = enter.append('g')
206                     .attr('class', 'dateLine')
207                     .attr('transform', `translate(${
                            xScale.bandwidth() / 2},0)`)
208
209                 // The circle which intersects the
                        date-line and top-line is added,
```

```
                positioned and sized.
210             dateLine.append('circle')
211                 .attr('class', 'dateLineDot')
212                 .attr('cx', xScale(datum.date))
213                 .attr('cy', yScale(datum.refugees))
214                 .attr('r', 6)
215
216             // The line is added to the date-line.
217             dateLine.append('line')
218                 .attr('class', 'dateLineLine')
219                 .attr('x1', xScale(datum.date))
220                 .attr('x2', xScale(datum.date))
221                 .attr('y1', yScale(datum.refugees))
222                 .attr('y2', contentHeight)
223
224             // The text showing the current refugee
                    number is added above the date-line.
225             dateLine.append('text')
226                 .attr('class', 'dateLineText')
227                 .text(datum.refugees)
228                 .attr('x', xScale(datum.date))
229                 .attr('y', yScale(datum.refugees) -
                        10)
230         },
231         // This describes the behavior of the update
                sub-selection of the general update
                pattern.
232         update => {
233             // The circle is transitioned to its new
                    position.
234             update.select('circle').call(update =>
                    update.transition(t)
235                 .attr('cx', xScale(datum.date))
236                 .attr('cy', yScale(datum.refugees)))
237
238             // The line is shifted and adjusted in
                    length.
239             update.select('line').call(update =>
                    update.transition(t)
240                 .attr('x1', xScale(datum.date))
241                 .attr('x2', xScale(datum.date))
242                 .attr('y1', yScale(datum.refugees)))
243
244             // The text value is updated and
                    repositioned.
245             update.select('text').call(update =>
                    update.transition(t)
246                 .text(datum.refugees)
247                 .attr('x', xScale(datum.date))
```

```
248                         .attr('y', yScale(datum.refugees) -
                               20))
249                 }
250             )
251 };
252
253 /**
254  * This section tries to subscribe to the
        daily-data-service for data updates.
255  * The diagram will not work without the
        daily-data-service.
256  */
257 try {
258     parent.registerDailyDiagramRenderCallback(render);
259     console.log('Could successfully subscribe to the
          daily-data-service for data updates.');
260 } catch (e) {
261     console.log('Could not subscribe to the
          daily-data-service for data updates. ' +
262         'Data is loaded directly.');
263
264     loadDailyData(render)
265 }
```