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 19.08.2022
Bearbeitungszeit: 07.07.2022 - 01.09.2022

Brandenburg, den 19.08.2022

Ich, Luis Rothenhäusler, Student im Studiengang Informatik 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

Englische Arbeiten brauchen eine Zusammenfassung auf Deutsch. Mal abgesehen davon, dass wenn die Zusammenfassung interessant ist man ohne English eh nicht weiter kommt...

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 infographics. Some of the data visualization tools have a graphical-user-interface, like Excel[2], others are code based, like R[3] or the Matplotlib[4] 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. 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. 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. Yet knowing when to use which tool can be greatly beneficial for all parties involved.

To evaluate D3 and its possibilities there are three main questions that will be answered in this thesis. What is the potential of D3 in data visualization? What are the advantages and disadvantages of using D3? When is it reasonable to use D3? To be able to evaluate these questions a showcase of a several different diagrams is created and evaluated throughout this thesis. A live version of the showcase can be found at "`https://styxoo.github.io/`".

2. Basics

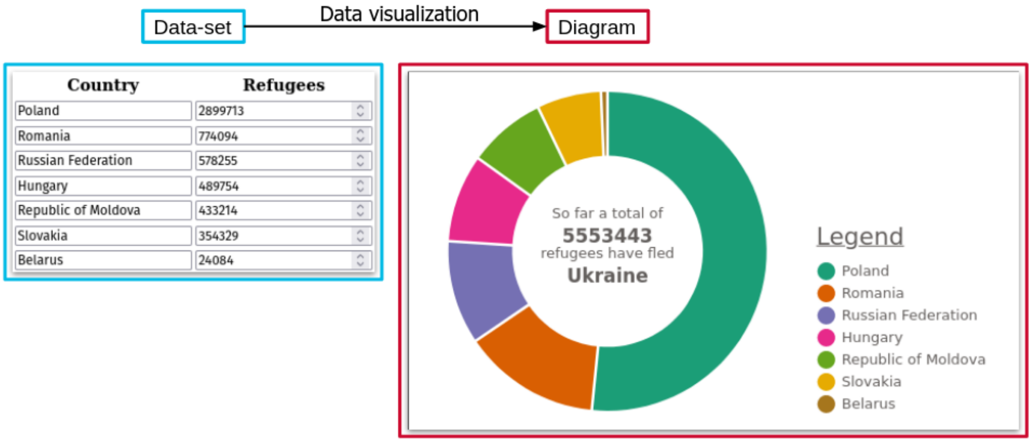
In this chapter, all concepts, technologies and required backgrounds for understanding this thesis are explained. First data and data types are described. Second diagrams and how they are structured are described. Last D3 as a tool to create diagrams is described.

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[5]. 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. 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[6].

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 them in the often limited space of articles, dashboards or other informative purposes. Therefore data visualization (Figure 2.1) is used to turn these datasets, collections of data-points, into diagrams.

Data is commonly preprocessed before turning it into diagrams. Depending on the dataset and the desired result, this can mean different things. One might want to remove excessive information from the dataset, 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[7]. It is important to note, that this preprocessing can happen with specific intentions in 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 have nothing to do with the correctness of the chosen data.



Country	Refugees
Poland	2899713
Romania	774094
Russian Federation	578255
Hungary	489754
Republic of Moldova	433214
Slovakia	354329
Belarus	24084

Figure 2.1: Where data visualization comes into play.

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. 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, they do not allow for arithmetic operations. There are two types of categorical data. Nominal and ordinal data.

Nominal data is mostly descriptive in nature. They are independent and have no inherited order. Examples are 'Country of origin', 'Color of paint', '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.

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. Like the 'Number of Refugees' or 'Tickets sold'. Discrete data is countable.

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

We constantly come across 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 the 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 doughnut-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[8]. 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. Channels describe the way specific marks encode data. The three possible marks are points, lines and areas. Each mark should use at least one channel to encode data. Otherwise it does not convey any information. 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. For example in fig. 2.2 we can see lines being used as marks for each of the seven entries. It might seem like areas are used, but the thickness of the line only serves visual understanding. The lines also 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

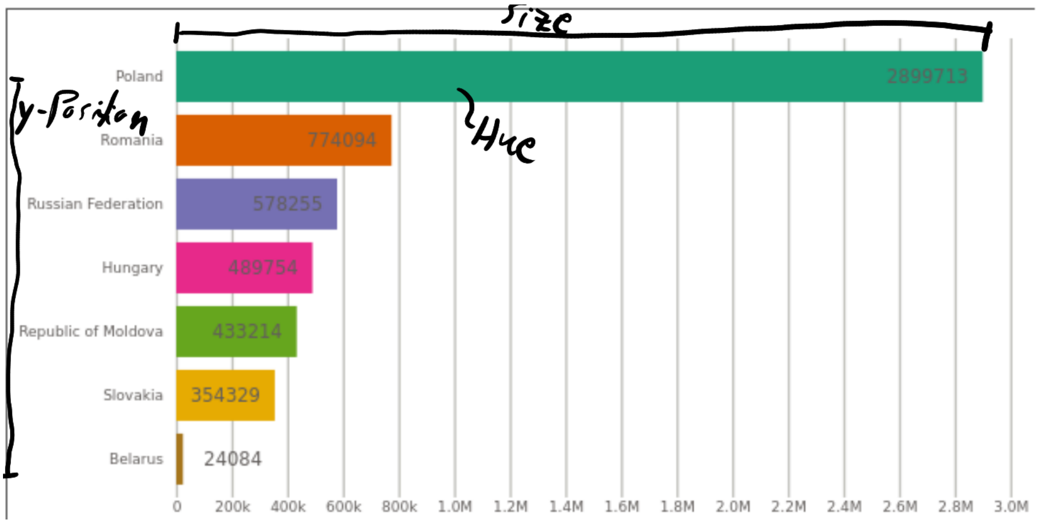


Figure 2.2: This bar-chart uses lines as marks. Each bar is a single line mark. The thickness has no relevance other than making the line visible. The three channels each mark encodes are marked. The y-position and the hue are used to encode the country. The size of the line, aka the length, corresponds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

many refugees have crossed into the country. In fig. 2.3 we see areas used as marks. Just like in the previous example the hue encodes the country and the size encodes the refugee count.

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[9]. 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.

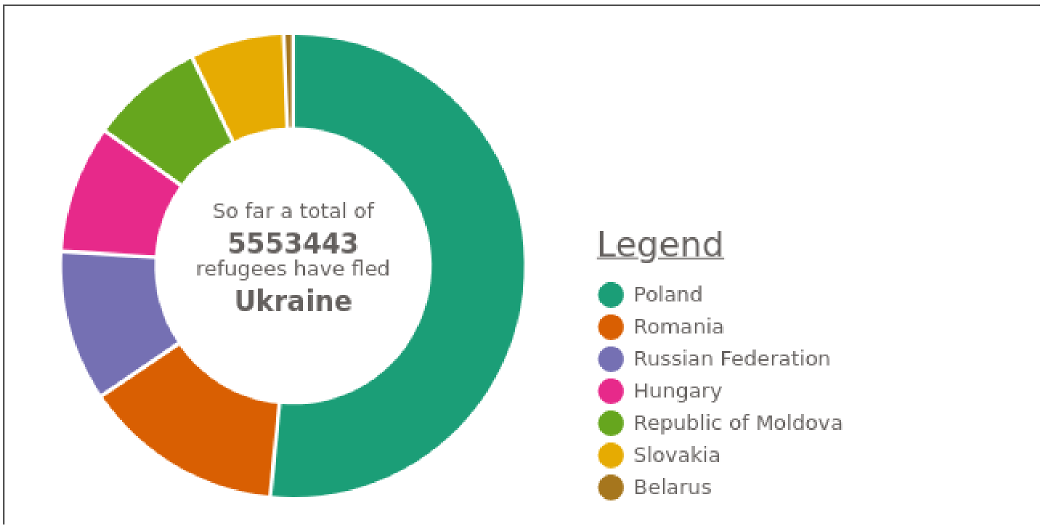


Figure 2.3: This is a donut-chart (TODO: which needs a frame..? Also draw in marks and channels)

2.3 D3.js

"D3.js is a JavaScript library for manipulating documents based on data. D3 helps you bring data to life using HTML, SVG, and CSS."[10]. 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 500kb in size. 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."[10], 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 some helper functions like scales, to decrease the amount of mathematical equations needed to convert from the data extends 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 of these 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 provided with a data join. Finally the behaviors for 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 modularity are explained.

2.3.1 Selections

All operations in D3 run on an arbitrary collection of nodes. These collections of nodes are called selections. 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[11] 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. Thee are also other functions which return selections. Calling a data join creates three new selections. Selections can also be extended or shrunken by adding or removing nodes, 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 fo its selection process.

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

2.3.2 Data Joins

Data joins are a key feature of D3. They link up 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 dataset, an array of objects where each object represents a single data-point, as parameter. This will bind the data-points to the nodes in the selection. This is achieved by using an identifier function. The default identifier function returns the index of the data-point in the dataset. When we want to create diagrams which can respond to data changes over time, this is not a reliable identification. When data-points

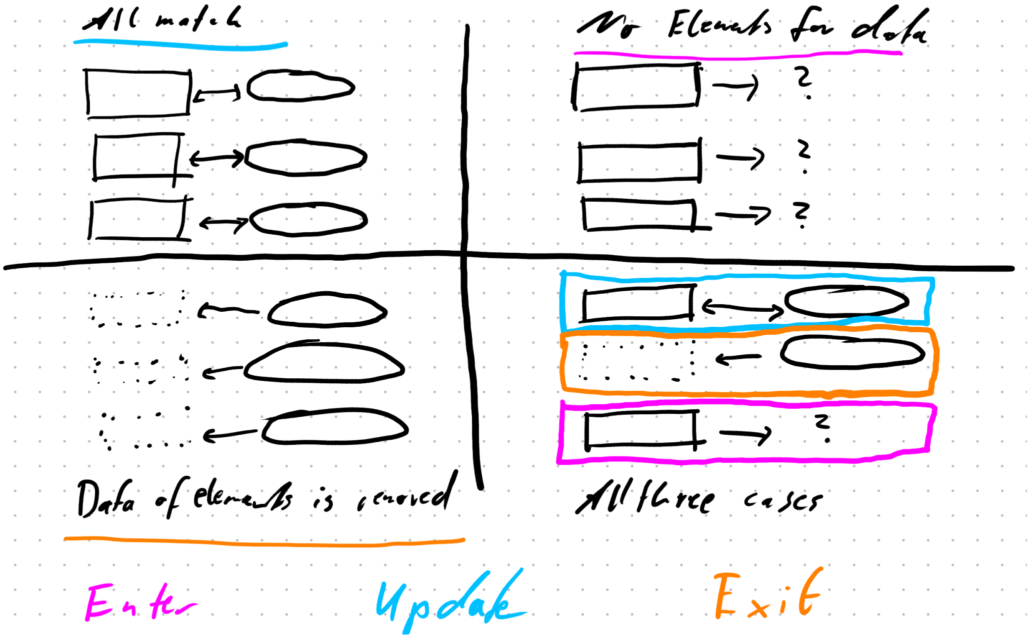


Figure 2.4: A representation of possible data joins. In the top left, the data join was able to match all data points to an element of the provided selection. In the top right, there are data points but no elements in the provided selection. In the bottom left, the provided selection already was filled with elements, but their corresponding data points have been removed. The bottom right shows that all three previous cases can exist in a single data-join.

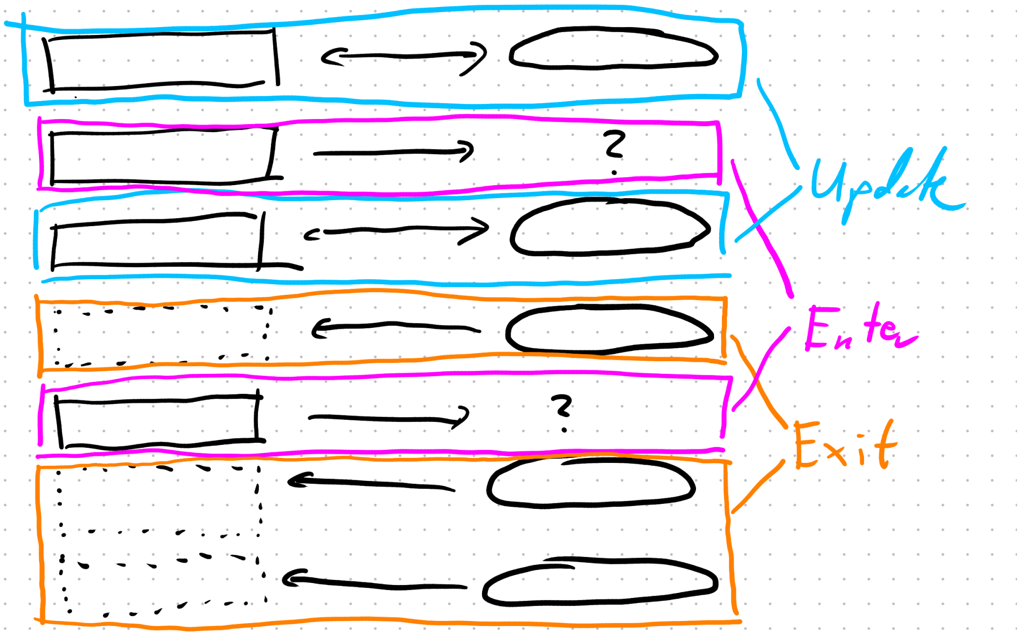


Figure 2.5: A possible data-join. It contains all three of the cases of the general update pattern. The cases marked blue make up the update selection, the pink cases the enter selection and the orange ones make up the exit selection. All these three cases of the general update pattern can have different behavior specified.

are removed or added in arbitrary locations, the index will not match the elements it previously did. Therefore we can specific a custom identifier function. This can be passed as the 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. When updating a data-join one should also recreate the underlying selection.

As seen in fig 2.4, 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 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 comes into play. The general update pattern differentiates between three different cases. For each of these cases a sub-selection is created. For each of these three selections the behavior can be defined. The first selection is the enter selection. It corresponds to the pink elements in fig 2.5. All data-points which have been matched up with a placeholder node while creating the data-join are in here. In the behavior for the enter selection, usually a corresponding element is created as the first step.

All the elements which are already linked to a data-point 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 for which the corresponding data-point has been removed. They are marked in orange in fig 2.5. The behavior of the exit selection is by default defined to remove the respective elements.

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 does not need to be appropriately matched by the data-join. 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. The exit behavior can be defined if a more visually pleasing removal of elements is desired, like fading out before deleting.

2.3.4 Scales

Scales are a way to convert between two data-spaces. Some scales can even convert between two data-types. Scales can be found in many places. For example converting percentages of correct answers in a test, continuous data, to the appropriate grade, ordinal data. Or the scale factor of maps and model-kits.

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 data-space 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 types of scales also allow to be used in reverse.

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[12], to draw the sankey graph.

3. Implementation

In the following sections the process of creating the showcase and the diagrams are described. There are several parts to this. At first the data-sets, which should be represented, are chosen. In most real world usages, this is already given. Afterwards the possible diagrams are considered and chosen. Their implementation and usages of D3 are described. Finally the showcase bringing all the diagrams together is described.

3.1 Datasets

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, no extra efforts was done to cover both these types.

All data used for the creation of the diagrams in this thesis originates from UNHCR Ukraine refugee situation page[13]. The dataset about total cumulative border crossings from Ukraine per day[14] is in JSON format. The data about the border crossings into countries featured in the refugee response plan, as well as into other neighboring countries[13] were extracted directly as CSVs. While all data reference 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"[15]. Therefore this thesis will henceforth use the refugee terminology. The refugees per country cover a time-span between february 24th 2022 up until august 16th 2022[13]. The refugees per day cover the time from february 24th 2022 until july 17th 2022[14].

To keep the implementations of the diagrams as simple as possible, some data preprocessing was done. Therefore two data service JavaScript files have been created. The first JavaScript file, the countryDataService.js reads both csv files containing information about the refugees fleeing into all 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 service, the dailyDataService.js, reads the JSON file containing information about the total refugees per day. As this JSON file contains a lot of filler data, which is not needed, the data service 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. Both data services pass the data to the applicable diagrams. The data services are also responsible for filling the data tables in the showcase and pass along any data changes done here to the applicable diagrams.

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 section is about the selection and implementation of each diagram. 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 is to use one of the diagrams as a template. Therefore all diagrams follow the same pattern. As each diagram is independent, they all consist of three parts. A HTML, a CSS and a JavaScript file. The HTML loads the D3 library in the header. 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 also follows a general pattern. At first there is a initialization section which is run once as the website is loaded. It is followed by a render function which is responsible for drawing and updating the diagram.

3.2.1 Initialization

Generally all things which are data independent are done during the initialization. It starts with setting some core variables. A reference to the svg tag which will be used as the container is made. It is followed by a margin definition, where the margin of our diagram content in relation to the container size is defined. The resulting values for `ourHeight` and `ourWidth`, which we will use as space to draw the diagram, are saved.

Following there are a few group elements which are added to the svg tag. These group elements provide a general hierarchy for different aspects of the diagram. For example the bar chart has separate groups for the axes and the content, while the circle diagram has groups for the background legend and the content. Having a proper structure in place makes working with selections easier as well as helping with human readability of the svg's content. This general hierarchy is only created to a level which is independent of the provided data.

Listing 3.1: JavaScript code to create the hierarchy as used in the bar-chart

```
const diagramGroup = svg.append('g')
    .attr('transform', `translate(${margin.left},${
       margin.top})`);

const xAxisParentGroup = diagramGroup.append('g')
    .attr('id', 'xAxis')

const yAxisParentGroup = diagramGroup.append('g')
    .attr('id', 'yAxis')

const contentParentGroup = diagramGroup.append('g')
    .attr('id', 'content')
```

Listing 3.2: Resulting HTML structure

```
<svg>
    <g transform=translate(118,20)>
        <g id="xAxis"></g>
        <g id="yAxis"></g>
        <g id="content"></g>
    </g>
</svg>
```

If there data independent scales, they are defined here. A common example here is a color scale for discrete values. It is used in all diagrams showing the refugees per country. It is not important to already know the specific input values, to be able to create a list of colors which is used by the scale. When queried, it will then return a new color from the list, for each new query value. It is important to note that when the color list runs out of new colors, it restarts at the beginning of the list.

Listing 3.3: Definition of data independent color scale

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

Finally 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. They are already created here, so they can be filled with appropriate data later.

3.2.2 Render

Following the initialization section is the render function. The render function is called once in the beginning and every time the provided data changes. The render function covers all data dependent tasks. If there are helper functions or constants required by the render function they are defined first. Following this all the data dependent scales are defined.

Scales

The data dependent scales in this thesis are mostly used to acquire the coordinate position and sizing of elements in the diagrams. The bar-chart for example defines two scales. A linear scale to convert from a domain of the refugees [0-MaxNumberOfRefugeesInACountry] to a domain of the available space [0-ourWidth]. Converting from any given country to a y position is done using a scale band. As both these scales depend on the provided data, 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, other diagrams, like the circle-diagram, make use of several data joins. Usually this is in accordance to how many independent parts the diagram consists of. The circle diagram uses one data join for the size legend in the background, one to update the circle showing the current data and one to update the text showing the current number of total refugees.

A data join is created when binding data to a selection. This is achieved by first calling the `.data(DATA)` function of a selection. The data function creates pairs of elements and data entries. By default, these are matched through their index in the selection and data arrays. This can lead to unexpected behavior when entries are removed or inserted at the not last position.

Therefore the default identifier function can be overwritten by passing a custom identifier function as the second optional parameter to the data function. A custom identifier function should return a value and is called for each element in the data array. For the refugees per country data-set in this thesis, the identifier is usually `d => {return d.country}`.

When we initially create the data join, or when data-points are added, we do not have sufficient 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, we add the `.join()` after the data() call. There are two ways to use the join function. We can either pass a string which will result in adding a matching tag to the DOM. 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 daa changes. In this thesis we want all diagrams to implement the full extend of the general update pattern, to be able to react to changing data and use the full possibilities of D3.

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 function has one input parameter, corresponding to the respective sub-selection. In the enter function usually a element is added to the DOM. In the exit selection we remove elements again. The update function is optional, but always used in this thesis, as this is the place to update existing elements to accommodate for data changes and therefore possibly removed or newly added elements as well. All three functions run on all the elements of the appropriate sub-selection.

The enter function should add the applicable placeholder element to the svg as actual content. 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. Following this the applicable styles, attributes and sub-elements are added. This can be achieved with the `.attr("attributeName", "value")`. Whilst styles can be added with the `.style("property", "value")` function, the same can be achieved by predefining styles in the css and adding applicable classes to the element. It is important to add enough attributes, that the provided selector which was used to create the selection for the data join whose behavior we are defining, can also match the newly created element when called again for an update. When positioning a new or existing element the scales are used to find the applicable coordinate space.

The update function is necessary when we want to react to data changes. It is usually similar to the enter function, in that is 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.

All three functions can make use of animations and transitions to improve their feel.

Animations

Animations can improve the feel, appeal and readability of diagrams. Especially when reacting 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 new diagram popping up out of thin air. The animations allow the viewer to keep track of the existing entries and visually follow any changes. It is also possible to see the changes of existing values by following, 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. The duration and delay are both in milliseconds. Animating numerical, color or string values is very easy with transition. It is only required to call the attribute or style with the target value and the transition will take care of the rest. This makes it very fast and easy to animate for example positioning or sizing.

```
    enter.call(enter => enter.transition(t)
    .attrTween('d', (d, index, nodes) => {
            const i = d3.interpolate(0, d.startAngle);
            const j = d3.interpolate(0, d.endAngle);

            nodes[index].previousStartAngle =
               d.startAngle;
            nodes[index].previousEndAngle = d.endAngle;

            return time => {
                d.startAngle = i(time);
                d.endAngle = j(time);
                return arc(d);
            }}))
```

Instead of using the default behaviors for numbers, string 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 tweens are only specifically defined to animate svg path tags.

3.2.3 Bar Chart

How does it work? Which d3 features does it use? how do they work?

3.2.4 Pie chart

How does it work? Which d3 features does it use? how do they work?

3.2.5 Tree map

How does it work? Which d3 features does it use? how do they work?

3.2.6 Sankey

How does it work? Which d3 features does it use? how do they work?

3.2.7 Area graph

How does it work? Which d3 features does it use? how do they work?

3.2.8 Circle graph

How does it work? Which d3 features does it use? how do they work?

3.3 Showcase

To bring all the diagrams together, a showcase has been created. It is split into two main parts. Firstly all the diagrams for refugees per country are covered. Secondly the diagrams for refugees over time are shown. As all diagrams in one section represent the same data-set, it allows for an easy visual comparison, as well as easier comparison of the code. When each diagram would show different data, it would be harder to distinguish between implementation differences which are due to the different representation and differences which are caused by accommodating different data-sets. Additionally each section has a table where the input data can be seen and modified.

3.3.1 Integration of each diagram

Each diagram is implemented to work on its own and without the showcase. Each diagram is also designed to use all the space available in its container. When loading one of the diagrams HTMLs directly, it will therefore fill the whole browser window. The showcase loads each of the diagrams into a separate IFrame tag with a consistent aspect ratio.

3.3.2 Data Updates

Each section of the showcase has a table which allows for data manipulation. Rows of data can be modified, added or removed here. The data changes here are not persistent and therefore do not get written in the original data csv files. When data is changed, the diagrams are provided with the updated data and adapt accordingly. 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 changing 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

As there are three main question this thesis aims to answer, each question will be discussed separately. Beginning with the question of what D3 can do. Looking at the created showcase, D3 can obviously be used to create many different types of diagrams. Looking at the examples found online makes this even more apparent. From simple bar and pie charts, over visualizing hierarchical data using tree-maps or sankey-graphs all the way to map based diagrams using various projections and physics enabled pseudo 3D representations. 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 could also be used for other aspects. One could draw charts 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 does D3 excel at and where does it struggle?

When working in a web environment, it is always easy to start using D3 as well. Its independence from any framework and implementation only in JavaScript, makes 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 hosting a, at least, local web-server before being able to use D3. 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. Especially when it is supposed to react to data changes. Learning D3 is also a double edged sword. Whilst there are a lot of examples and tutorials, they are often to 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 unfamiliar. Once one understands how D3 works, it can be quite fast to create basic diagrams. From here it is quite easy to make the diagrams react to data changes, once one properly understands the general update pattern and selections. Yet trying to animate diagrams can be tricky. It depends on the elements which are used and which attributes need to be animated. As all elements have to manually specified, one has full control over the appearance and behavior of the diagrams. Yet having to specify all aspects so precisely is also quite time consuming.

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, the resulting images can easily be imported here. When working with data which changes over time, but not in a web project, it is probably not worth introducing the additional overhead. If one is already working on a web project and with data changing over time, D3 should be seriously considered. Due to the easy integration with any framework and possibility to visualize anything it can adapt to all projects needs. As D3 allows for the full control of appearance and behavior of the diagrams, it can be adapted to any existing style guides. When only simple diagrams are needed and they only fetch data once as the site is loaded, the initial learning curve of D3 is not as steep. If the visualizations should be more complex, update with live data changes and follow specific specifications, D3 is a very well suited tool. In both cases it is worth using D3. Simple diagrams allow for an easy beginning to get familiar with the tool. Once familiar, D3 is capable to work with more complex visualizations.

5. Conclusion

How well did it work? Was it worth the effort? What could be improved?

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.” [Online]. Available: https://www.microsoft.com/en-us/microsoft-365/excel

[3] “The r project for statistical computing.” [Online]. Available: https://www.r-project.org/

[4] “Matplotlib - visualization with python.” [Online]. Available: https://matplotlib.org/

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

[6] “Total data volume worldwide 2010-2025,” May 2022. [Online]. Available: https://www.statista.com/statistics/871513/worldwide-data-created/

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

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

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

[10] M. Bostock, “Data-driven documents,” accessed:31.03.2022. [Online]. Available: https://d3js.org/

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

[12] “D3/d3-sankey: Visualize flow between nodes in a directed acyclic network.” Sep 2019. [Online]. Available: https://github.com/d3/d3-sankey

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

[14] ——, “Refugees per day,” accessed:23.07.2022. [Online]. Available: https://data.unhcr.org/population/get/timeseries?widget_id=336969&sv_id=54&population
01-01

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

A. Appendix

A.1 Showcase

A.1.1 HTML

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Showcase</title>
    <link rel="stylesheet" href="index.css">

    <!-- Import D3.js libraries -->
    <script src="https://d3js.org/d3.v7.min.js"></script>
</head>
<body>
    <div class="main-grid">
        <div class="main-headline">
            This is the Showcase
        </div>
        <div class="topic">
            <div class="topic-headline">Refugees per
               Country</div>
            <div class="content">
                <div class="diagrams">
                    <iframe class="diagram" src="./
                       total_per_country/bar-chart/
                       diagram.html" frameborder="01"
                       height="100%" width="100%"></iframe
                       >
                    <iframe class="diagram" src="./
                       total_per_country/pie-chart/
                       diagram.html" frameborder="01"
                       height="100%" width="100%"></iframe
                       >
                    <!--<iframe class="diagram" src="./
                       stacked-bar-chart/diagram.html"
                       frameborder="01" height="360"
                       width="720"></iframe>-->
                    <iframe class="diagram" src="./
                       total_per_country/tree-map/
                       diagram.html" frameborder="01"
                       height="100%" width="100%"></iframe
                       >
                    <iframe class="diagram" src="./
                       total_per_country/sankey/
                       diagram.html" frameborder="01"
                       height="100%" width="100%"></iframe
                       >
                </div>
            </div>
            <div class="info-space">
                <div class="info">
                    <div class="legend-container">Legend
                        <div>
                            <iframe class="legend" src="./
                               total_per_country/legend/
                               diagram.html" frameborder="
                               0" height="100%" width="
                               100%"></iframe>
                        </div>
                    </div>
                    <div class="data">Data
                        <div class="showData">
                            <label for="showCountryData">
                               Show Data</label>
                            <input type="checkbox" id="
                               showCountryData" onclick="
                               triggerCountryData(this)">
                        </div>
                        <table id="countryDataTable"
                           hidden>
                            <thead>
                            <tr>
                                <th>Country</th>
                                <th>Refugees</th>
                                <th></th>
                            </tr>
                            </thead>
                            <tbody id="tBody_per_country">
                               </tbody>
                            <tfoot>
                            <tr>
                                <td>
                                    <div class="
                                       autUpdateData">
```

```
                                        <label for="
                                           autoUpdateCountry
                                           ">Auto Update</
                                           label>
                                        <input type="
                                           checkbox" id="
                                           autoUpdateCountry
                                           " onclick="
                                           autoUpdateCountryData
                                           (this)">
                                    </div>
                                </td>
                                <td>
                                    <input class="button"
                                       type="button"
                                       value="Update data"
                                        onclick="
                                       updateCountryData()
                                       ">
                                </td>
                                <td>
                                    <input type="button"
                                       value="Add New Row"
                                        onclick="
                                       addCountryRow()">
                                </td>
                            </tr>
                            </tfoot>
                            <script src="./
                               total_per_country/
                               country-data-service.js">
                               </script>
                        </table>
                    </div>
                </div>
            </div>
        </div>
        <div class="topic">
            <div class="topic-headline">Total Refugees per
                Day</div>
            <div class="content">
                <div class="diagrams">
                    <iframe class="diagram" src="./
                       total_per_day/circle/diagram.html"
                       frameborder="01" width="100%">
                       </iframe>
                    <iframe class="diagram" src="./
                       total_per_day/area/diagram.html"
                       frameborder="01" width="100%">
```

```
                        </iframe>
<!--                    <iframe class="diagram" src="./total_per_country/bar-chart/diagram.html" frameborder="01" width="100%"></iframe>-->
<!--                    <iframe class="diagram" src="./total_per_country/pie-chart/diagram.html" frameborder="01" width="100%"></iframe>-->
<!--                    &lt;!&ndash;<iframe class="diagram" src="./stacked-bar-chart/diagram.html" frameborder="01" height="360" width="720"></iframe>&ndash;&gt;-->
<!--                    <iframe class="diagram" src="./total_per_country/tree-map/diagram.html" frameborder="01" width="100%"></iframe>-->
<!--                    <iframe class="diagram" src="./total_per_country/sankey/diagram.html" frameborder="01" width="100%"></iframe>-->
                </div>
            </div>
            <div class="info-space">
                <div class="info">
                    <div class="slider-container">
                        Current Time Selector
                        <br>
                        <table width="100%">
                            <tbody>
                                <tr>
                                    <td colspan="3">
                                        <input type="range" min="0" max="100" value="0" class="slider" id="timeSlider">
                                    </td>
                                </tr>
                            </tbody>
                            <tfoot>
                                <tr>
                                    <td>Start Date</td>
                                    <td align="center">Current Date</td>
                                    <td align="right">End Date</td>
                                </tr>
                                <tr>
                                    <td id="startDate">DD-MM-YYYY</td>
                                    <td id="currentDate" align="center">
```

```
                   DD-MM-YYYY</td>
                <td id="endDate"
                   align="right">
                   DD-MM-YYYY</td>
            </tr>
        </tfoot>
    </table>
</div>
<div class="data">Data
    <div class="showData">
        <label for="showDailyData">
           Show Data</label>
        <input type="checkbox" id="
           showDailyData" onclick="
           triggerDailyData(this)">
    </div>
    <table id="dailyDataTable" hidden>
        <thead>
        <tr>
            <th>Date</th>
            <th>Refugees</th>
        </tr>
        <tr>
            <td colspan="2">
                <input class="button"
                   type="button"
                   value="Add day
                   before" onclick="
                   addDayBefore()">
            </td>
        </tr>
        <tr>
            <td colspan="2">
                <input class="button"
                   type="button"
                   value="Remove first
                    day" onclick="
                   removeFirstDay()">
            </td>
        </tr>
        </thead>
        <tbody id="tBody-daily"></
           tbody>
        <tfoot>
        <tr>
            <td colspan="2">
                <input class="button"
                   type="button"
                   value="Remove last
```

```
                                    day" onclick="removeLastDay()">
                                </td>
                            </tr>
                            <tr>
                                <td colspan="2">
                                    <input class="button" type="button" value="Add day after" onclick="addDayAfter()">
                                </td>
                            </tr>
                            <tr>
                                <td>
                                    <div class="autUpdateData">
                                        <label for="autoUpdateDaily">Auto Update</label>
                                        <input type="checkbox" id="autoUpdateDaily" onclick="autoUpdateDailyData(this)">
                                    </div>
                                </td>
                                <td>
                                    <input class="button" type="button" value="Update data" onclick="updateDailyData()">
                                </td>
                            </tr>
                            </tfoot>
                            <script src="./total_per_day/daily-data-service.js"></script>
                        </table>
                    </div>
                </div>
            </div>
        </div>
        <script src="index.js"></script>
    </div>
</body>
```

```
</html>
```

A.2 Bar Chart

A.2.1 JavaScript

```
const svg = d3.select('#mainFrame')
    .attr('height', innerHeight)
    .attr('width', innerWidth);

const margin = {
    top: 20,
    right: 20,
    bottom: 20,
    left: 118
}
const ourWidth = innerWidth - margin.left - margin.right
const ourHeight = innerHeight - margin.top - margin.bottom

const diagramGroup = svg.append('g')
    .attr('transform', `translate(${margin.left},${
        margin.top})`);

const xAxisParentGroup = diagramGroup.append('g')
    .attr('id', 'xAxis')

const yAxisParentGroup = diagramGroup.append('g')
    .attr('id', 'yAxis')

const contentParentGroup = diagramGroup.append('g')
    .attr('id', 'content')


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


const render = data => {
    console.log('Rendering bar chart')

    const t = svg.transition()
        .duration(1500);

    const xValue = d => d.refugees;
    const yValue = d => d.country;

    /**
     * Here we set up all the required scales. One for the
```

```
      x-axis, one for the y-axis and one for the color-coding
     */
    const xScale = d3.scaleLinear()
        .domain([0, d3.max(data, xValue)])  // Original range of values
        .range([0, ourWidth])             // Range to map to
        .nice();

    const yScale = d3.scaleBand()
        .domain(data.map(yValue))
        .range([0, ourHeight])
        .padding(0.2);


    /**
     * Here we set up the y and x axes.
     */
    yAxisParentGroup.call(d3.axisLeft(yScale))          // Call axisLeft as function on the g element
        .selectAll('.domain, .tick line')   // and select all lines in tick class tags
            .remove();                      // and remove them

    const xAxisTickFormat = number =>    // For an input number
        d3.format('.2s')(number)        // format the number to have three significant digits
            .replace('0.0', '0');         // and replace the SI-unit G with B

    const xAxis = d3.axisBottom(xScale) // Pass a scale function as axis
        .tickFormat(xAxisTickFormat)    // and pass a formatting function to format the string
        .tickSize(-ourHeight);

    xAxisParentGroup.call(xAxis)
        .attr('transform', `translate(0,${ourHeight})`)
        .select('.domain')
            .remove();

    /**
     * This is where the actual content of the diagram is drawn.
     */
    contentParentGroup.selectAll('g .bar').data(data, d =>
{return d.country})
.join(
    enter => {
        const bar = enter.append('g')
            .attr('class', 'bar')

        bar.append('rect')
            .attr('width', 0)
            .attr('height', yScale.bandwidth())
            .attr('y', d => yScale(yValue(d)))
            .attr('fill', d => colors(d))
            .call(enter => enter.transition(t)
                .attr('width', d => xScale(xValue(
                   d))))

        bar.append('text')
            .text(d => xValue(d))
            .attr('class', 'barText')
            .attr('text-anchor', 'end')
            .attr('dy', '0.32em')
            .attr('y', d => yScale(yValue(d)) +
               yScale.bandwidth()/2)
            .attr('x', 0)
            .call(enter => enter.transition(t)
                .attr('x', d => {
                    const scaleValue = xScale(
                       xValue(d));
                    return (scaleValue - 60 > 0) ?
                        scaleValue - 10 : 60;
                }))
    },
    update => {
        update.select('rect')
            .call(update => update.transition(t)
                .attr('width', d => xScale(xValue(
                   d)))
                .attr('height', yScale.bandwidth()
                   )
                .attr('y', d => yScale(yValue(d)))
                   )
        update.select('text')
            .text(d => xValue(d))
            .call(update => update.transition(t)
            .attr('y', d => yScale(yValue(d)) +
               yScale.bandwidth()/2)
            .attr('x', d => {
                const scaleValue = xScale(xValue(d
                   ));
                return (scaleValue - 60 > 0) ?
```

```
                           scaleValue - 10 : 60;
                    }))
            },
            exit => exit.remove()
        )
};

/**
 * This section is only relevant for the implementation of the diagram within the iframe.
 * It tries to subscribe to the parent window for data.
 * If there is no data providing parent, it'll load its own data.
 */
try {
    parent.registerCountryDiagramRenderCallback(render)
    console.log('Could successfully subscribe to parent for data updates')
} catch (e) {
    console.log('Data is not provided externally. Loading data directly')
    const dataPath = '../total_refugees_per_country_condensed.csv';

// .csv creates a promise, when it resolves .then do something else
    d3.csv(dataPath).then(data => {
        data.forEach(d => {                                 // Foreach data-point in data
            d.refugees = +d.refugees;                   // Cast value to float and take times 1000
            d.country = `${d.country}`;                 // Kind of unnecessary, but fixed webstorm complaining
        });
        render(data);
    })
}
```