Otto von Guericke University Magdeburg

Faculty of Computer Science

Institute of Simulation and Graphics

Bachelor Thesis

Designing a Library to Create Animated Sequences Using D3.js

Author:

Vincent Göring

March 20, 2023

1. Reviewer

Prof. Dr. Bernhard Preim

Institute of Simulation and Graphics
Otto von Guericke University Magdeburg

2. Reviewer

Benedikt Mayer

Institute of Simulation and Graphics
Otto von Guericke University Magdeburg

Göring, Vincent:
Designing a Library to Create Animated Sequences Using D3.js
Bachelor Thesis, Otto von Guericke University Magdeburg, 2023.

Abstract

In online storytelling, the incorporation of data-driven visualizations is widespread. To explain the results of the analysis of complex datasets or to highlight different aspects, a variety of visualizations is employed. Rather than displaying a series of graphics, many stories seamlessly transition from one graphic to another graphic, creating an animated sequence. This thesis presents the implementation of a JavaScript library that allows the efficient creation of such animated sequences and is compatible with the JavaScript library D3.js. The resulting library, GSAP-ASEQ, is based on the animation library GSAP and showcased by implementing a lead scenario. In addition, the necessity of such a library is justified and requirements are determined. The developed library is validated according to these requirements. To determine the actual added value of the library and to identify potential improvements or issues, empirical studies are necessary. This work is aimed at persons with basic programming skills, experience in web development, and an interest in developing digital visualizations.

Contents

1

Introduction

1.1 Motivation

Data visualizations nowadays play an important role in digital storytelling. Many online journalists and news agencies, for example, *The Guardian*, incorporate interactive graphics in their articles to support their narratives and present complex issues [1].

Segel and Heer [1] identify three common structures that are used in so-called *data stories*. One of them, the *Interactive Slideshow*, is characterized by a linear sequence of views through which the user is guided. An interactive slideshow is mainly author-driven but may allow interaction within the single visualizations. According to Segel and Heer [1], interactive slideshows work effectively with complex datasets as data dimensions and modifications of the displayed visualization are presented step by step. To help the user stay in context and to increase engagement, the use of animated transitions between the views can be vastly beneficial [2]. Thus, *animated sequences* play an important role in narrative visualizations. The article *Homicide database: Mapping unsolved murders in major U.S. cities* [3] published by *The Washington Post* provides an example of the use case of animated sequences. In this case, displayed graphics are supplemented by short texts and scrolling is used to navigate within the interactive slideshow.

A popular possibility to create data visualizations for web pages is using the JavaScript library *D3.js (D3)* [4]. However, it is not obvious how to efficiently implement an animated sequence that uses D3 for the creation of the single views. Especially implementing features such as skipping or reversing transitions is a rather complex task. It is not clear how to structure the code and avoid a large coding overhead.

1.2 Aim of this Thesis

This thesis aims to present a library that facilitates the creation of animated sequences and supports the creation of views based on D3. The desired functionality of the library will be specified by determining requirements. By showing that existing approaches are not able to meet the determined requirements, I highlight the importance of this work.

As this work presents a JavaScript library, basic programming knowledge and fundamental HTML and CSS skills are required.

1.3 Structure

For a better overview, the structure of this thesis is described in more detail:

Chapter 2 (Background and Fundamentals) deals with the main concepts of D3 and the role of animated transitions in visualizations. In addition, I specify the use of animation in the context of this work and summarize the results of my research on existing approaches for creating animated sequences.

Chapter 3 (Design Concept). In this chapter, requirements for the library to be developed are established. Moreover, the concept of the lead scenario is presented. The lead scenario indicates certain requirements and serves as an introduction to the use case of animated sequences.

Chapter 4 (Approaches Based on D3 Transitions) discusses possible strategies for implementing animated sequences using only D3. First, an existing pattern and a self-developed library are explained and validated. Second, the limitations of D3 transitions and resulting issues are depicted. In addition, implementation-specific considerations regarding the library to be developed are made.

Chapter 5 (GSAP-ASEQ: A Library Based on GSAP). In this chapter, the developed library (*GSAP-ASEQ*) for creating animated sequences is presented. GSAP-ASEQ is based on the animation library GSAP. Therefore, general information about GSAP and its features, tailored to the required context, is pointed out. Next, the implementation and usage of GSAP-ASEQ are explained. The developed library will be validated on the basis of the requirements, defined in Chapter 3. To showcase the developed library, the concept of the lead scenario, developed in Chapter 3, is realized and presented.

Chapter 6 (Conclusion) discusses the process and the result of this work. In addition, an outlook on possible future work regarding the developed library is given.

2

Background and Fundamentals

This chapter discusses the advantages of incorporating animation in visualizations. After specifying the kind of animation I focus on in this work, an introduction to D3 and its basic usage is given. This is necessary for fully understanding Chapter 4 and Chapter 5. In addition, I highlight the relevance and benefits of using D3 for creating data visualizations. At last, I present the results of my research on existing approaches that deal with the implementation of animated sequences based on D3.

2.1 Animation in Visualizations

Studies have shown that users often prefer animated visualizations over non-animated visualizations [2, 5] and have given strong evidence that animations can be of great benefit. For example, Bederson and Boltman [6] showed that using animation when navigating around spatial data supports the user in building mental maps of the spatial information. Heer and Robertson [2] found that animated transitions can improve the perception of changes when switching between statistical data graphics.

A well-known advantage of animation is that it can be employed to support object constancy when transitioning between different states or representations. In their paper about *Cone Trees*, a technique to visualize hierarchical structures in 3D, Robertson et al. [7] describe that "object constancy enables the user to track substructure relationships without thinking about it" (p. 190). Thus, the cognitive load can be reduced. The user's comprehension of changes may be enhanced by maintaining object constancy and supporting object tracking through animation. This benefit is demonstrated in the following example: First, a geographical map is displayed to the user. Then, an enlarged section of a specific region is shown.

Without any animation, the user would probably have difficulties locating the region in the firstly shown map and would need to search the map for landmarks, such as rivers or mountains. With the help of an animated transition, e.g., zooming in from the first view to the second view, the user would probably be able to quickly locate the rough location of the region on the first map. The animated transition helps the user to stay in context and improves the understanding of the relationships between different views.

Robertson et al. [2] name four possible advantages of using animation in their paper *Animated Transitions in Statistical Data Graphics*, including the support of object constancy. One advantage is that the animation of motion is effective to direct the viewer's attention to the animated object. Robertson et al. argue that in comparison to other visual properties, such as color [8], motion is easier to perceive in the peripheral vision. Another advantage mentioned by the authors is that animation can help construct narratives by conveying cause-and-effect relations. Moreover, animations may capture the viewer's emotions and heighten the level of interest and delight.

Robertson et al. [2] also mention drawbacks of animation, such as distraction, time-error tradeoffs, and the possible misuse of object constancy. Fast animations may result in errors if the viewer cannot follow the animated changes. To counteract this, longer animations may be used. However, this leads to a longer waiting period which may bore the viewer. Object constancy can be misused by transforming elements into unassociated elements. Consequently, false relationships are conveyed to the viewer.

To conclude, animations can be of great benefit in visualizations, provided they are used appropriately.

2.2 Animation in the Context of this Thesis

Animation can be used for enhancing multiple types of changes, such as visualizing trends or showing functionality [9]. In this thesis, I mainly focus on animations in the context of animated transitions between views. That means transitioning from an initial state to an end state. However, animated transitions could also be used to visualize a change of data, a change over time or other different kinds of changes, which exceed a simple transition between two views.

In the context of this work, I focus on the animation of *attributes*[1] and *CSS (Cascading Style Sheets) properties*[2] of *HTML (HyperText Markup Language) elements*[3] and *SVG (Scalable Vector Graphics) elements*[4]. Thus, when speaking of the *state* of an element, the entirety of attributes and CSS properties and their corresponding values are meant. Consequently, the term *element* and *visual element* refer to an SVG or HTML element.

2.3 D3.js: Data-Driven Documents

D3 is an open-source JavaScript library created by Michael Bostock, Vadim Ogievetsky, and Jeffrey Heer. It allows efficiently creating and transforming elements in the *Document Object Model (DOM)*, e.g., HTML and SVG elements, depending on data [4]. The DOM represents the content and structure of a webpage as objects and provides a standard interface for accessing and manipulating its elements, for example by using a scripting language, such as JavaScript [10]. According to D3's website [11], the library is very performant and supports large datasets and corresponding transformations of the DOM, which may be triggered by user interactions and data changes.

2.3.1 Features

In 2011, Bostock et al. [4] introduced D3 and presented its advantages over other visualization tools: Most visualization tools do not allow direct manipulation of the DOM but provide scenegraph abstractions. This may lead to inefficient debugging and a limited variety of possible visualizations. In addition, the authors argue that scenegraph abstractions can be a barrier for new users as they do not consider or support the user's possible prior knowledge in web development. That means known standards or tools are not applicable and, thus, the learning expense may be relatively high. Consequently, in order to solve these issues, which mainly lead to inefficiency and constraints, D3 pursues three goals: compatibility (1), debugging (2), and performance (3).

(1) As D3 is a JavaScript library that directly manipulates the DOM, it can coexist with most other tools that may improve efficiency in other tasks leading to a high compatibility.

[1] `https://developer.mozilla.org/en-US/docs/Web/HTML/Attributes` and `https://developer.mozilla.org/en-US/docs/Web/SVG/Attribute` (visited on 01/22/2023)
[2] `https://developer.mozilla.org/en-US/docs/Web/CSS/Reference` (visited on 01/10/2023)
[3] `https://developer.mozilla.org/en-US/docs/Web/HTML/Element` (visited on 01/22/2023)
[4] `https://developer.mozilla.org/en-US/docs/Web/SVG/Element` (visited on 01/22/2023)

(2) For debugging in D3, all JavaScript debuggers, such as built-in debuggers of browsers or Node.js[5], can be used. As D3 has no scenegraph abstraction, it is possible to inspect the scenegraph by directly inspecting the DOM.

(3) Bostock et al. also state that the performance is increased (compared to other visualization tools) based on the following argument: As D3 does not use a scenegraph abstraction, no mapping of the tool-specific representation to the DOM is necessary. Consequently, changes, such as transformations, are directly applied. The performance of D3 is also benchmarked and compared to other tools - however, the present relevancy must be questioned as the paper [4] is from 2011 and D3 and other visualization tools may have evolved, and new tools and libraries have been developed.

2.3.2 How to Use D3

The following paragraphs are mainly based on the paper *D3 Data-Driven Documents* [4] and the D3's website [11]. Most of the functionality of D3 is accessed by the globally imported `d3` object.

Selections.

To select DOM elements D3 provides two functions: `select` (returns the first matching element or an empty selection) and `selectAll` (returns all matching elements or an empty selection). Both functions need to be provided with a selector, which is specified by the W3C Selectors API[6]. This may, for example, be a class, ID, tag, a combination of multiple selectors, or a node reference. The returned selection can be mutated in various ways, such as changing attributes or *style properties* (CSS properties) or creating a subselection. This is done by *method chaining* and applying operators, such as `select` or `style`, whose values can be provided either as a constant or function. In object-oriented programming, method chaining allows directly calling multiple methods one after another without storing intermediate results. This is possible if the previous function returns the object on which the subsequent function is called.

Example:

```
d3.selectAll(".blue")
    .attr("class", "red")
    .select("p")
        .style("font-size", "30px");
```

The above code selects all elements with a class attribute of the value of "blue" and overwrites their classes with the new class "red". Then, within the previ-

[5] `https://nodejs.org/en` (visited on 01/23/2023)
[6] `https://www.w3.org/TR/selectors-api/` (visited on 01/23/2023)

ous selection, we select the paragraph first appearing in the DOM by using the corresponding tag "p" and alter its style by changing the font size.

Binding Data to Nodes.

The `data` operator binds data to visual elements. The data must be provided in the form of an array of arbitrary values. The bound data can then be accessed when applying other operators and used to define corresponding values.

Example:

```
d3.selectAll("p")
    .data(["A", "B", "C", "D"])
    .text(function(d, i){
        return d + i;
    });
```

In this example, we first select all paragraph elements, bind the data array to them, and change the text contents of the paragraphs according to the bound data. The parameters `d` and `i`, represent the bound data value and the index of the bound data in the data array. Assuming that the DOM exactly contains four paragraphs, the text of these paragraphs would now be "A0", "B1", "C2", and "D3". However, this example is not representative for most visualizations as we assume that the number of selected elements equals the length of the data array. Thus, after applying the `data` operator, placeholder elements for data that could not be bound to an element and elements to which no data could be bound can be accessed via the subselections `enter` and `exit`. This enables creating elements, for example by using the operator `append`, depending on data and handling leftover elements.

Animated Transitions.

D3 supports the use of animated transitions. Transitions can be applied by calling the `transition` function on a selection. This function returns a transition instance on which the end state of the transition can be defined. Consequently, a D3 transition animates from an initial state to an end state. Transitions can be further customized, for instance, by setting a delay, duration, or easing function. In addition, by using the `attrTween` function, custom interpolators can be defined. An interpolator determines the value of an animated property for every frame of the transition. Multiple transitions can be triggered one after another by method chaining. A chained transition starts directly after the previous transition is finished.

Example:

```
d3.selectAll("rect")
    .transition()
```

```
            .attr("fill", "red")
            .attr("height", "50px")
            .transition()
                .attr("width", "0px");
```

In this code snippet, all rectangles (referenced via "rect") are transitioned to a fill color of red and a height of 50 pixels. After that, the width is transitioned to zero.

2.4 Existing Approaches

My research of existing approaches for creating animated sequences using D3 has led to few results. I found many visual stories that incorporate animated sequences but almost no publicly available generalized approaches to create animated sequences based on visualizations implemented with D3. A possible programming pattern using scrolling as a navigation technique is provided by Jim Vallindgham [12]. However, in Chapter 4, I will show that this approach does not meet the later determined requirements.

3

Design Concept

In this chapter, I first present the concept for a lead scenario in order to provide an example of the use case of animated sequences. This lead scenario is later used to showcase the developed library. In Section 3.2, requirements for the library to be developed are established and justified.

3.1 Lead Scenario

The lead scenario to be developed is based on a task sheet[1] of the visualization lecture at the Otto von Guericke University Magdeburg.

3.1.1 Subject and Goal of the Lead Scenario

The lead scenario focuses on a heatmap that shows the development of the global air temperature anomalies from 1850 to 2022 (see Fig. 3.1). The reference value for the monthly anomalies is the average air temperature of the corresponding month between 1951 and 1980. The data used was extracted from Berkely Earth[2]. An animated sequence shall be used to explain how the heatmap is constructed. The heatmap resembles Ed Hawkins' *Warming Stripes*[3] graphics. Warming stripes visualize the change of temperature over time and are employed in popular science. The animated sequence shall counteract possible misinterpretations of the heatmap and give an understanding of Warming Stripes in general.

[1] `https://observablehq.com/d/b0c077a75a8dc71f` (visited on 01/25/2023)
[2] `https://berkeleyearth.org/data/` (visited on 01/25/2023)
[3] `https://showyourstripes.info/s/globe` (visited on 01/25/2023)

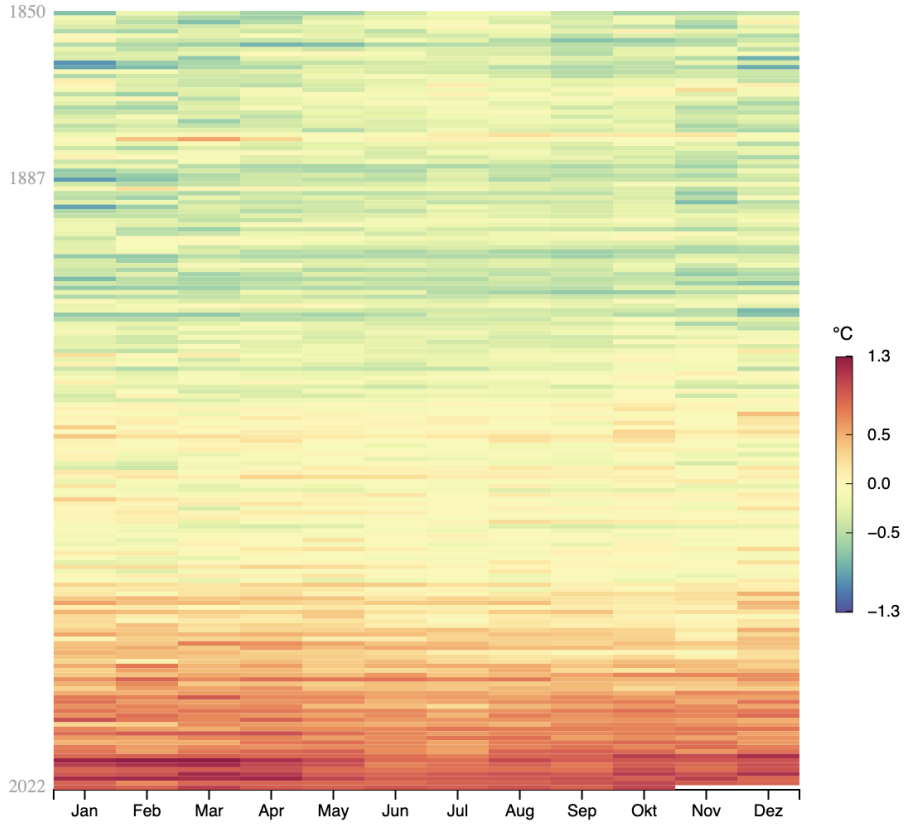

Figure 3.1: This heatmap visualizes the global monthly temperature anomalies per month on the x-axis and per year on the y-axis.

3.1.2 Concept

Each view of the animated sequence shall be supplemented by a text on the left-hand side of the visualization. The texts shall provide additional information or explanations regarding the displayed view. Moreover, a *stepper*, as described by McKenna et al. [13] shall be used for navigating between the views. That means, discrete control over animations, a linear story with the possibility of linear skipping, and navigation through buttons. In the further thesis, including the following section, I mainly focus on the described type of navigation.

In order to build up the heatmap, we first focus on a specific year and construct the corresponding horizontal stripe in the heatmap. This stripe is later inserted into the right position in the heatmap. The shown graphics in the corresponding figures are the resulting views of the implementation of the lead scenario with JavaScript and D3. When navigating from a view to a neighboring view, animated transitions shall be used.

View 1. First, the reference value of the anomalies, the average global temperature per month, calculated in the period from 1851 to 1980, is displayed in a simple step chart. The temperature is provided on the y-axis. The months are displayed on the x-axis.

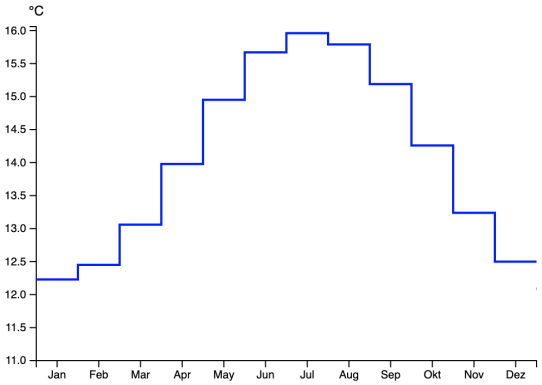

Figure 3.2: View 1

View 2. The monthly temperatures of 1887 are drawn into the chart by adding another step-line. I chose the year 1887 as the global anomaly during the winter (reaching its maximum in January) of this year was extremely high. Extreme weather conditions, such as early snow and cold temperatures, especially in continental North America, led to major losses in the cattle industry [14]. There, the winter of 1886 / 1887 is also known as *The Big Die-Up* [14].

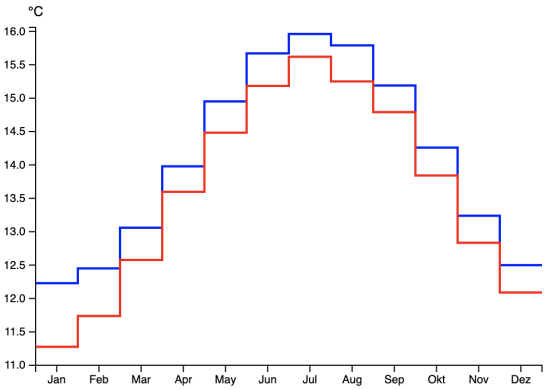

Figure 3.3: View 2

View 3. To visualize the anomalies of 1887, the enclosed areas between the two displayed step lines are filled with color. Thus, this view demonstrates how anomalies are calculated.

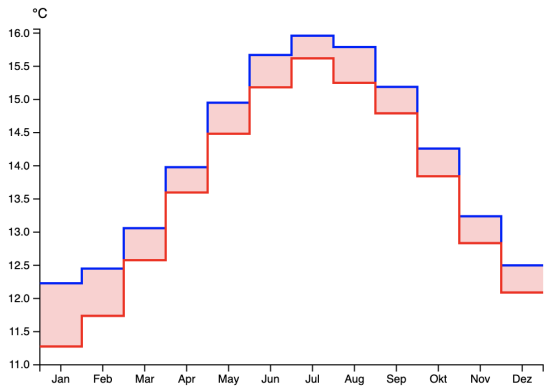

Figure 3.4: View 3

View 4. Next, the previous graph is transitioned into a bar chart that allows the direct reading of the anomaly values. Thus, the y-axis now represents a scale for the anomaly values. To improve the viewer's understanding of the transformation, the step-line of the average global temperature is flattened to a straight line at a y-axis value of zero. To create a bar chart, the colored differences of the previous view are transformed into rectangles.

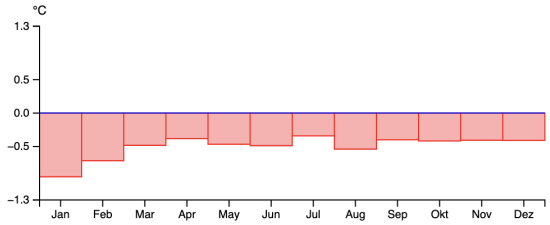

Figure 3.5: View 4

View 5. The heatmap to be constructed displays anomaly values by using a color scale. Thus, in this view, the bars are color-coded relative to the lowest and highest anomaly that ever appeared between 1850 and 2022. The corresponding color scale should be shown.

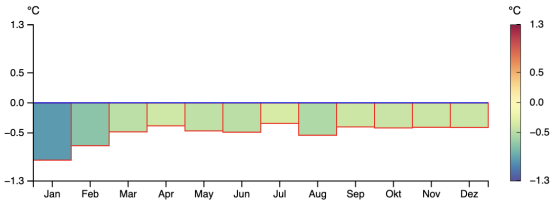

Figure 3.6: View 5

View 6. The sixth view transitions all bars to the same height, and removes the y-axis. Now, the shown graphic represents a single stripe of the upcoming heatmap.

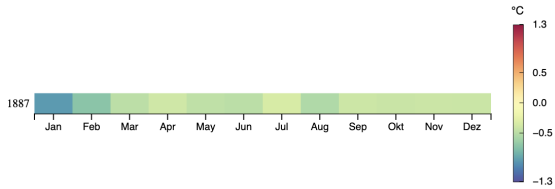

Figure 3.7: View 6

View 7. The last view (see Fig. 3.1, p. 10), shows the complete heatmap. The stripe of the year 1887 is inserted into the corresponding position. To avoid overloading the screens with labels and to reduce the cognitive load, only the labels of the years 1850, 1887 and 2022 shall be permanently visible. Other labels shall appear when hovering over the corresponding position in the heatmap. The heatmap enables the viewer to put the year 1887 in context with other years or generally identify trends or outliers.

3.2 Determination of Requirements

The lead scenario represents an example that would benefit from a library which simplifies the implementation of animated sequences. In this section, I list and describe the general requirements for such a library. The requirements were determined in cooperation with an employee of the Institute of Simulation and Graphics at the Otto von Guericke University Magdeburg, who deals with interactive D3 visualizations on a daily basis. The requirements are divided into two categories: visualization-related requirements and programming-related requirements.

3.2.1 Visualization-Related Requirements

Bi-Directonal Navigation (RBiDiNav)

Undo and redo operations are commonly used in a diversity of applications, including data stories, to enable the user to reset the system to the previous or following state [15]. These operations allow the user to go back and forth in the state history. It could be possible that the user wants to go back to the first view or revisit certain views. For example, regarding the lead scenario concept in Section 3.1.2, the user may want to revisit step two after having seen the whole sequence of views, because they want to have a more detailed look at it. Thus, reverting animation steps is a basic interaction technique that should be supported by the library to be developed.

Reversing Animated Transitions (RReverseTrans)

This requirement builds upon the requirement RBiDiNav. When reverting to a previous state, the corresponding transitions should be played in reverse in order to support object constancy the same way as it is done when advancing to the next view.

Handling Active Transitions when Navigating (RHandleTrans)

If the user navigates through the sequence of views without skipping, fast-forwarding animated transitions reduces the waiting periods and may benefit the user's efficiency and satisfaction. In addition, the benefits of animated transitions are maintained. If, in the currently displayed view, transitions are still playing (*active*) and the user advances to the next view, transitions shall be fast-forwarded. If the transitions are played in the forward direction and the user navigates to the previous view, the playing transitions shall be reversed immediately, starting from their current state. The described behavior of active transitions during a navigation action shall apply in both directions of the sequence.

Skipping Views (RSkipViews)

Skipping views allows the user to directly navigate to an arbitrary view. Thus, it makes the navigation in a long sequence more efficient. The user may not be interested in intermediate steps, e.g., when they already have seen them or are only interested in a particular view. Consequently, navigating to a certain view by incrementally navigating to the next or previous view, as described in RBiDiNav, is time-consuming. Thus, the usability may be increased by providing the possibility of skipping multiple views.

When skipping views, all transitions shall be executed without any animation. This equals changing the duration of all transitions to zero. I decided that this is more effective and beneficial than playing all intermediate transitions as this would likely lead to a long waiting period and not necessarily provide any benefits. To counteract this, the intermediate transitions could be sped up. A high speed-up leads to multiple transitions played in a very short time period. Thus, it is probable that the viewer cannot follow and fully comprehend the animated transitions. Consequently, the use of animated transitions while skipping has no benefit but may irritate the viewer.

Staggering Animated Transitions (RStaggerTrans)

Heer and Robertson [2] showed evidence that staggering animated transitions can be helpful to avoid occlusions and support object tracking. Furthermore, staggering animations can help to reduce the perceived complexity of a transition. In the last step of the lead scenario concept, described in Section 3.1.2, a staggered transition is beneficial: Ideally, the heatmap stripe of the year 1887 should be inserted into the heatmap after the heatmap is fully displayed. Consequently, the user is able to track the position of the anomaly values of 1887 in the heatmap more easily. Without staggering, the transition might appear complex and the user may lose track of the values of 1887. The full staggered transition is shown in Figure 3.8.

Completing Animated Transitions (RCompleteTrans)

Before advancing from one view to the next view, all previous transitions should first be completed. Then, the following transitions may be started. The same shall apply in the reverse direction. In some cases, it could be useful that playing transitions are simply interrupted by a new transition, but, in order to avoid unexpected states, I decided to avoid this behavior.

Expressiveness (RExpress)

The library to be developed should be applicable to a preferably large variety of animated sequences. However, other requirements and the focus on creating views

with D3 need to be taken into account. Consequently, the expressiveness is mainly characterized by the customizability of animated transitions.

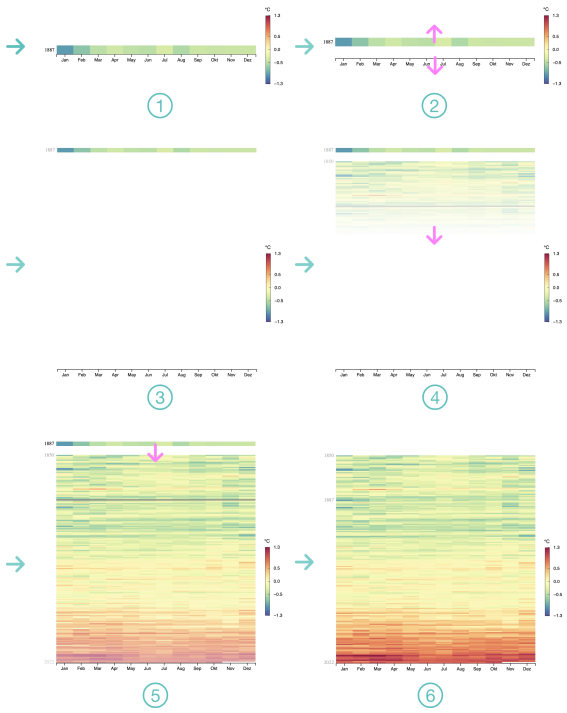

Figure 3.8: The screenshots visualize the different steps of the staggered transition of view 7 of the lead scenario. The motion direction of objects is indicated by pink arrows.

3.2.2 Programming-Related Requirements

One-Directional Programming (ROneDiProg)

This is the main requirement for the library to be developed. When intuitively programming an animated sequence using only D3's functionality, the developer would need to code reversed transitions manually. To fulfill the requirement RReverseTrans, every transition would need to be programmed once in the forward and once in the backward direction. This leads to a high number of lines of code that could be avoided. With an increasing amount of transitions and attributes that are affected by transitions, more code and effort are needed. Especially when using staggered transitions by setting different delays, the programming of the reverse of the staggered transition can get tedious.
Moreover, the previous attribute values of an element, which are needed for the reversed transition, are not always obvious. Consequently, the developer may need to spend time searching the previous attribute values in the code or inspecting the element in the previous view via a HTML inspector. If the previous attribute values are changed, all transitions that revert to these values need to be manually updated. Therefore, the effort to apply changes to transitions is relatively high.

Concluding, the developer should ideally only need to program in the forward direction and the effort for reverting or skipping animation steps should be minimal or none.

Learning Effort and Complexity (RLeEffort)

The learning effort, and thus, the complexity of the library to be developed should be as low as possible. It is important that in this case, the term complexity refers to the complexity of using the library and not the complexity of the library itself. It has to be taken into account that this requirement may influence the requirement RExpress. The higher the expressiveness the more probable it is that the learning effort and complexity grow to cover all functionalities.

Lines of Code (RCodeLines)

This requirement partly supplements the previous two requirements RLeEffort and ROneDiProg. The amount of code that is needed to perform elementary tasks should be reduced to a minimum. Moreover, many function calls or functions with many parameters should be avoided.

4

Approaches Based on D3 Transitions

In this chapter, I describe two approaches that shall facilitate the implementation of animated sequences. Both approaches are solely based on *D3 transitions*[1] and do not use any additional libraries. Additional libraries are likely to increase the learning effort (RLeEffort). Moreover, maintaining a library with dependencies may lead to issues when dependencies are updated, not well maintained, or the library itself shall be updated. I first examine a programming pattern presented by Vallandingham for creating an animated sequence. The discussion of this pattern results in two concept points that are incorporated into the library to be developed (D3-ASEQ and GSAP-ASEQ). Next, I present and validate a self-developed library called D3-ASEQ for facilitating the creation of an animated sequence. Vallandingham's programming pattern and D3-ASEQ are both unable to match the determined requirements from Section 3.2. Thus, in Section 4.4, I describe why it is difficult to reach the established requirements by using D3 to animate elements.

4.1 Vallandingham's Approach

As already mentioned in Section 2.4 (Existing Approaches), Vallandingham [12] presents a pattern for programming scrollable data visualizations with discrete control over transitions. This approach could possibly be transferred to a new approach that uses a stepper to navigate between the views. First, I explain the relevant components of the pattern in more detail. After that, I show why this approach does not meet the established requirements.

[1] `https://github.com/d3/d3-transition` (visited on 02/01/2023)

4.1.1 How it Works

This entire subsection is based on Vallandingham's article *So You Want to Build A Scroller* [12]. Vallindgham advises providing a function that is run prior to showing the first view and generates all visual components of all views. This *setup function* directly hides all elements by setting their opacity to zero. If the animated sequence consists of a large number of different visual elements, a setup function can be implemented for each view. Thus, the structure and readability of the code are improved.

Each view requires a function that renders the matching view to the screen. These *draw functions* are then stored in an array from which they can be invoked by providing the right index. When scrolling over a specific position, the corresponding draw function is called. Each draw function consists of two steps:

1. Hide all visual elements of the two neighboring views of the corresponding view that shall not be visible. This is done by setting the opacity to zero.
2. Reveal all visual elements that shall be displayed in the corresponding view and that are not used in the neighboring views or transform already visible visual elements.

When hiding or revealing elements, attributes or style properties of elements can also be modified. All modifications of elements shall be applied in the context of D3 transitions. For immediate changes, the transition duration shall be set to zero. This leads to the following behavior: Running transitions in the displayed view may be interrupted by new transitions when navigating to the next view. That means an active transition is immediately stopped and a new transition is played. This particularly happens when the user scrolls quickly. A new transition that modifies the same element as an active transition stops and deletes the active transition regardless of the animated attributes or style properties. For transitions that should not be interrupted, *named transitions* (provided by D3) are used. As long as two or more transitions don't alter the same attribute or style property of an element, transitions with distinct names can be played concurrently without interfering with one another.

When quickly scrolling from the first to the last view, Vallandingham's implementation of the scrolling mechanism ensures that all draw functions of intermediate views are called in the right sequence. Consequently, all intermediate transitions are started but not necessarily completed as they may be interrupted by subsequent transitions.

4.1.2 Validation

The pattern presented by Vallandingham may be of great benefit for structuring the source code of an animated sequence. However, it does not provide any reusable functionality that drastically simplifies the implementation of an animated

sequence, e.g., reversing a transition. Thus, the main goal was not reached. An overview of the outcome of the validation is shown in Table 4.1. In the following, the fulfillment of the requirements is examined in more detail.

Requirements			
Visualization-Related		Programming-Related	
RBiDiNav	✓	ROneDiProg	✗
RReverseTrans	✓	RCodeLines	✗
RHandleTrans	✗	RLeEffort	✓
RSkipViews	✓		
RStaggerTrans	✓		
RCompleteTrans	✗		
RExpress	✓ (same as D3)		

Table 4.1: Result of the validation of Vallandingham's programming pattern according to the determined requirements from Section 3.2.

Visualization-Related Requirements

RHandleTrans and RCompleteTrans. These requirements are not supported. Active transitions are either interrupted or played concurrently with the following transitions when the viewer advances to the next view. Thus, transitions are neither sped up nor completed before the subsequent transitions are played.

RBiDiNav, RReverseTrans, RSkipViews, and RStaggerTrans. These requirements are fulfilled.

RExpress. Regarding the expressiveness, the customizability of transitions is determined by the options provided by D3. As RBiDiNav is not fulfilled, it must be considered that programming the reverse of complex transitions is a tedious and possibly challenging task.

Programming-Related Requirements

ROneDiProg, RCodeLines, and RLeEffort. The main issue is that Vallandingham's approach does not support one-directional programming, which is one of the most important requirements. In every draw function, the developer has to deal with the visual elements of the subsequent view and their behavior when navigating back to the view described by the draw function. That means every transition has to be programmed twice: once in the forward direction and once in the reversed direction. Consequently, regarding the implementation of animated transitions, this approach does not reduce the number of lines of code. However, the learning effort and complexity are low, as this pattern mainly represents an example of how to structure the D3 code in functions and does not provide any additional functions or complex code.

4.2 Implementation-Specific Considerations

Regarding the implementation of a library, crucial considerations were made. These considerations are explained in the following two concept points.

Code Structure

Structuring the code into functions for each view and storing them together in an array, as described by Vallandingham, increases the readability of the code and ensures easy access of the draw functions. Therefore, this organization of code is assessed as good practice and beneficial. I consider creating all visual elements in one or more setup functions as a disadvantage: Creating the visual elements directly inside the draw functions and not before they are needed leads to more coherent code and may be more intuitive. Thus, it is possible to directly apply a transition to an element after its creation. Otherwise, the elements are first created in a setup function and then need to be reselected with D3 or accessed via a variable to apply a transition. Consequently, removing the setup functions may reduce the number of lines of code and improve the structure without any drawbacks.

Values, objects, or functions that need to be accessed in multiple draw functions, e.g., scales or general layout values, shall be stored in variables outside the draw functions. Consequently, they can directly be used in any draw function that lies in the same scope. The initialization of such variables may be implemented in a separate function that is invoked prior to any other function. Thus, the general code structure of an animated sequence should look similar to this:

```
let varX, varY, varZ; // to be initialized
const drawFunctions = [drawView1, drawView2, ...]

function initialize(){ ... }
function drawView1(){ ... }
function drawView2(){ ... }
...
```

If an animated sequence is very complex or consists of many views, the code structure may be improved by defining each function in a separate file. Consequently, variables that are needed in several draw functions may also be defined in a separate file and imported into the corresponding functions.

Creating and Deleting Visual Elements

In the pattern described by Vallandingham, visual elements are hidden or shown by changing their opacity value to zero or one. Consequently, hidden elements still

trigger *pointer events*[2], such as clicking or hovering over an element. In addition, the elements still take up space in the layout. To ensure that hidden elements do not affect the rendered document in any way, I decided to hide elements by setting their CSS property `display` to `none`. Setting the value of `display` back to any other value than `none` makes the element visible again. It was also considered to delete visual elements. When returning to a view from another view, deleted elements would need to be recreated. This requires that the relevant data of visual elements must be saved beforehand. If an element is deleted, it is completely removed from the DOM. This increases the readability of the DOM and may shorten the loading times of the website. However, effects on the loading time will probably only occur with a huge amount of elements. I concluded that hiding elements using their `display` property instead of deleting them has no crucial drawbacks and is easier to implement.

4.3 D3-ASEQ: A Library based on D3 Transitions

I developed the D3-ASEQ library without using any third-party libraries and relied solely on D3's functionality. The source code and a simple example are uploaded on GitHub as parts of the project (and library) named d3-aseq[3]. Further development of this library was halted as I found that without using a third-party animation library the requirements are difficult to reach. Therefore, no documentation is provided. However, the main aspects of the implementation are explained to make the subsequent section (A Deeper Dive into D3 Transitions) more understandable and to present an exemplary library.

4.3.1 Implementation Details

All classes and functions are exported by the file *helperFunctions.js* (see Fig. 4.1). The main functionality is implemented in a class called `TransitionsManager`. This class is mainly used for reversing animated transitions. It contains four *private instance fields* [4]. The `step` field stores the index of the view that is displayed to the viewer. The index of a view is defined by its position in the animated sequence. When switching to another view, the `step` value must be incremented or decremented using corresponding methods provided by the `TransitionsManager`.

The field `numberOfActiveTransitions` specifies how many transitions are playing at the time it is called. By reading the value of this variable, the developer can

[2] `https://developer.mozilla.org/en-US/docs/Web/API/Pointer_events` (visited on 02/02/2023)
[3] `https://github.com/vinccenttt/d3-aseq` (visited on 02/14/2023)
[4] `https://www.geeksforgeeks.org/javascript-class-level-fields/` (visited on 02/28/2023)

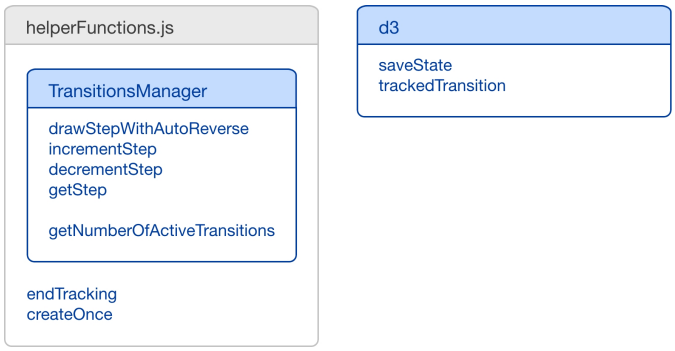

Figure 4.1: Diagram of the exported functions of D3-ASEQ.

ensure that the user is only able to navigate to another view when all playing transitions are completed (see RCompleteTrans). This is done by only allowing a navigation interaction if the number of active transitions equals zero.

D3-ASEQ only focuses on animated transitions of attributes and excludes the animation of style properties. In order to reverse transitions, the previous state of an element must be known. Therefore, every time the state of a visual element changes, the new state shall be saved by the `TransitionsManager`. The `TransitionsManager` stores the states of all visual elements by using two fields. The first field is an array containing a list of the changed elements for each view. The index of a list represents the index of the corresponding view inside the animated sequence. Elements are stored and identified by their IDs. The second field is a *Map Object*[5] that holds the IDs of the visual elements as a key and the corresponding attribute list (state) as a value. The following pseudo-code demonstrates the structure of the two fields used for storing and accessing the different states of elements.

```
let field1 = [[id0, id1, ...], [id3, id0, ...], ...];

let field2 = new Map([
    [id0, [[attr1, valueX], [attr2, valueY], ...],
    [id1, [[attr4, valueZ], ...],
    ...],
)
```

The order of the IDs in the elements of `field1` and the order of the attribute-value pairs in `field2` are arbitrary. In addition, array lengths may differ.

As suggested by Vallandingham [12], it is beneficial to define each view of the animated sequence in a single function. Thus, in order to use D3-ASEQ, each view needs to be implemented in a corresponding draw function. To use any

[5] `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Map` (visited on 02/04/2023)

methods provided by the `TransitionsManager` Class, first, an instance of it needs to be created. The draw functions are not directly called but given as a parameter to the `drawStepWithAutoReverse` method together with a boolean value that determines whether the view is accessed from a previous or following view (regarding the order of the views in the sequence). If the boolean value is true, `drawStepWithAutoReverse` only calls the passed draw function. Otherwise, it calls a private method that plays the animated transitions of the passed draw function in reverse. The latter is done by reading and afterward deleting the previously stored states from the stored fields. Every time the user advances from one view to the next view, the new states of changed elements are stored. Every time the user navigates from one view to the prior view in the sequence, the corresponding states are deleted. Consequently, the fields that are used to store the states of elements are altered each time the user navigates to another view.

To ensure that the reversing and calling of draw functions work correctly, every state change of a visual element needs to be saved. Therefore, two functions (`saveState` and `trackedTransition`) are provided which both take an instance of the `TransitionsManager` class as an input parameter. These two functions can be accessed via any D3 selection (for example: `d3.select("rect").attr("opacity", 0).saveState(manager))`. This is done by extending `d3.selection.prototype`. The function `saveState` needs to be called after the creation of a new element. It stores the state of the element and returns the D3 selection it was called on. The function named `trackedTransition` is used for applying transitions. It returns a D3 transition and increments the number of active transitions. In addition, when using `trackedTransition`, the function `endTracking` needs to be called at the end of the corresponding transition. This function decrements the saved number of active transitions and stores the attribute list for the corresponding element. The correct usage of `trackedTransition` is demonstrated in the code snippet below.

```
d3.select("pseudo-class")
    .trackedTransition(manager)
    .attr("opacity", 0)
    .on("end", function () {
        endTracking(manager, this);
    });
```

Another important function is the `createOnce` method. It takes an ID and a function as input parameters. If an element with the given ID already exists, the provided function is not executed. If no matching element is found, the specified function is called. Thus, the `createOnce` method can be used to ensure that an element is only created once when calling a draw function multiple times, e.g., when navigating back and forth. D3-ASEQ does not support the deletion of elements (as advised in Section 4.2). The usage of the `createOnce` is demonstrated in the example below.

```
createOnce("pseudo-id", () => {
    d3.append("rect")
        .attr("id", "pseudo-id");
});
```

4.3.2 Validation

D3-ASEQ allows one-directional programming but does not meet the majority of the defined requirements (see Table 4.2). It has major shortcomings in terms of complexity and expressiveness. In the following, the fulfillment of the requirements is analyzed in depth.

Requirements			
Visualization-Related		Programming-Related	
RBiDiNav	✓	ROneDiProg	✓
RReverseTrans	✓	RCodeLines	✗
RHandleTrans	✗	RLeEffort	✗ (tendency)
RSkipViews	✗		
RStaggerTrans	✗		
RCompleteTrans	✓		
RExpress	✗		

Table 4.2: Result of the validation of D3-ASEQ according to the determined requirements from Section 3.2.

Visualization-Related Requirements

RSkipViews, RHandleTrans, RStaggerTrans. These requirements are not supported.

RBiDiNav, RReverseTrans, and RCompleteTrans. D3-ASEQ fulfills the named requirements. By only allowing to navigate to another view if the number of active transitions is zero, RCompleteTrans can be fulfilled.

RExpress. Regarding the expressiveness, transitions can only be applied to attributes and not style properties. Moreover, transition properties, such as delay or duration, are completely ignored when playing reverse transitions. Thus, transitions cannot be staggered. In addition, D3-ASEQ does not support hiding visual elements over multiple views and then revealing them in a later view. In conclusion, the customizability of transitions is very limited.

Programming-Related Requirements

ROneDiProg. D3-ASEQ supports one-directional programming as the reversing of transitions is done automatically when using the library correctly.

RLeEffort. As demonstrated in the previous code snippets, functions are called in multiple ways: directly, via the `TransitionsManager` class, via a D3 selection, or inside another function. This increases the complexity and might result in a heightened error rate. In addition, navigating between the views requires multiple function calls. Before calling a draw function, the number of active transitions needs to be checked and the `step` field may need to be incremented or decremented. To conclude, the learning effort and complexity are relatively high.

RCodeLines. Every time a transition is created, `endTracking` needs to be called after the completion. Every time a new element is created, the `createOnce` function and the `save` method need to be called. This leads to many function calls with corresponding parameters which may make the implementation process tedious. As a result, the number of lines of code increases.

D3-ASEQ only works if all elements have a unique ID, as it is used to store the different states of an element. This could be a hindrance to some use cases.

There are additional issues regarding the efficiency. D3-ASEQ always stores all attributes of an element and not only the ones that were changed by the transition. This leads to inefficiency but could be improved by further development. As mentioned earlier, elements' stored states are deleted when navigating from a view to the previous view. When navigating back to the latter view, the same states are stored again. Consequently, the same data is stored and deleted multiple times when navigating back and forth. This is inefficient and possibly avoidable.

4.4 A Deeper Dive into D3 Transitions

D3-ASEQ does not fulfill the majority of the determined requirements. Even with further development, it is questionable whether the requirements can be met. This is due to the limits of D3 transitions. D3 does not provide any functions to reverse transitions. Therefore, in order to support one-directional programming (ROneDiProg), a library needs to track and store the attribute values and style properties of each visual element. In addition, custom transition settings need to be considered. This complicates the implementation of a library that is easy to use. The following text is based on D3's documentation of transitions [16] and my personal programming experience in D3.

Staggered Transitions. D3 supports staggering transitions. One possibility of staggering transitions is to use delays. Consequently, if one delay or transition duration is changed, all the delays of subsequent transitions need to be adjusted as they are provided as constant values. Storing delays and durations in variables may solve this issue. However, this may lead to a high number of variables and consequently worsen the readability of the code. In addition, the implementation process becomes time-consuming and tedious. Another way of implementing transitions in D3 is chaining transitions by making use of event listeners. The

end-event is triggered at the end of a transition. Thus, a subsequent transition can be initiated after the completion of a previous transition. However, this leads to nested code which worsens the code structure and influences the readability and traceability. This could be counteracted by wrapping each transition into a function. Thus, the amount of nested code would be reduced. This is demonstrated in the following example:

```
transition1().on("end", () =>
    transition2().on("end", () =>
        transition3().on("end", () =>
            transition4();
        )
    )
);
```

Transitions one to four represent methods that select one or more elements, apply a transition on them and return the transition. The traceability of the code is questionable as it is nested in multiple layers and includes a high number of brackets. Moreover, it should be evaluated if structuring the transitions into separate functions has drawbacks. In conclusion, the implementation of staggered transitions using D3 is possible but may worsen the traceability of the code and increase the needed effort when applying changes to a staggered transition.

As already mentioned, automatically reversing transitions in D3 is complex to implement. Reversing staggered transitions adds additional complexity to this task. Each transition needs to be reversed individually. When reversing a staggered transition, the delays need to be adjusted. As mentioned, staggered transitions can be implemented in D3 in various ways, such as using their delays or adding event listeners. Thus, automatically reversing staggered transitions is a challenging task.

Modifying Active Transitions. To fulfill the requirement RHandleTrans accessing and modifying playing transitions is necessary. As already mentioned in chapter 2, D3 transitions are played immediately after their creation. Changing their properties afterwards is not possible. For example, when trying to change the duration or an attribute value of an active transition, the transition is not affected and an error is thrown. If transitions are interrupted or completed, they are deleted. Thus, D3 does not support storing, reusing, and altering transition instances. In order to modify an active transition, it needs to be interrupted by a new transition with the desired changes. Regarding the speeding up of a transition after a specific event is triggered, the transition would need to be interrupted and a new transition needs to be played. To correspond to the old transition, all properties of the old transition, except properties regarding timing, need to be passed to the new transition. However, there is no obvious way to retrieve all the changed attributes and style properties of a transition instance. To summarize, modify-

ing active transitions would result in destroying active transitions and creating new transitions. Therefore, many values need to be passed from one transition to another transition.

Although D3 supports the use of animated transitions, many features that are necessary to fulfill the established requirements, such as reversing, pausing, or modifying transitions, are not well supported, require high effort and can only be implemented by using complicated workarounds. Thus, the use of an animation library was considered.

5

GSAP-ASEQ: A Library Based on GSAP

In this chapter, I present the final result of the development process of creating a library that facilitates the implementation of animated sequences. The library is called *GSAP-ASEQ*[1]. The source code and a detailed documentation are published on GitHub.

5.1 GSAP

There are numerous JavaScript libraries that focus on animation, such as *Anime.js*[2], *GSAP*[3], *mo.js*[4], or *Popmotion*[5]. I found that the GreenSock Animation Platform (GSAP) provides the necessary functionality for reaching the requirements. In the context of this work, no comparison of different animation libraries and their benefits and drawbacks was made.

According to their website [17], GSAP allows the animation of any numeric property of animatable objects, including HTML and SVG elements. It is entirely written in JavaScript and does not depend on any third-party libraries. On their website, GSAP is advertised to be robust, highly compatible, fast, and lightweight. In addition, there is active support and a forum to help developers solve difficulties. *GreenSock*, the company behind GSAP, refers to globally recognized companies,

[1] `https://github.com/vinccenttt/gsap-aseq` (visited on 02/14/2023)
[2] `https://github.com/juliangarnier/anime` (visited on 02/07/2023)
[3] `https://greensock.com/gsap/` (visited on 02/07/2023)
[4] `https://mojs.github.io` (visited on 02/07/2023)
[5] `https://popmotion.io` (visited on 02/07/2023)

such as YouTube, Samsung, or Amazon that use GreenSock products regularly [17].

5.1.1 Advantages of Using GSAP

GSAP provides a large variety of options for implementing and organizing animated transitions. From this point forward, all references to the functionality (including implementation details) provided by GSAP are based on GSAP's documentation [18].

GSAP provides three different methods to create a *tween*. A tween executes an animation. It determines the value of the animated properties at each time step in accordance with its settings, such as duration or delay. Thus, tweens are used to create transitions. The following code snippets juxtapose implementing a transition on HTML elements using D3 (left) and using GSAP (right).

```
d3.selectAll(".dummy-
   class")
     .transition()
     .duration(2000)
     .style("color",
            "#fff");
```

```
gsap.to(".dummy-class",
    {duration: 2,
     color: "#fff"});
```

This example demonstrates the similarities between GSAP and D3 regarding the implementation of transitions. (The duration value is different due to different units.) GSAP's and D3's functionality are both accessed via a globally imported object (`gsap` and `d3`). The `gsap.to` function creates and returns a tween. By default, as also in D3, the animation is played directly. Similar to D3, `gsap.to` needs to be given a selector to identify the target elements. All other animation properties, such as duration or target values of certain style properties or attributes, are passed as an object to the `gsap.to` function.

In contrast to D3 (see Section 4.4), GSAP supports storing, pausing, and modifying transitions. For example, the returned tween of the `gsap.to` function in the previous code snippet could be stored in a variable. GSAP provides many methods that can be called by a tween, e.g., reversing, pausing, or speeding up the animation. This functionality is particularly important to achieve the requirements RReverseTrans, RHandleTrans, and ROneDiProg.

Moreover, GSAP provides a *timeline* that facilitates dealing with complex sequences of animations. This is crucial to support the requirement described by RStaggerTrans and RExpress. A timeline consists of multiple animations that are organized in time. The functionality is similar to the functionality of a tween. Thus, a timeline can simply be reversed by calling the corresponding function. The code snippet below demonstrates how a timeline can be implemented.

```
let timeline = gsap.timeline();
timeline.to("#id1", {opacity: 0, duration: 3});
timeline.to("#id2", {x: 0, delay: 0.5});
timeline.to("#id3", {y: 0, duration: 1});

timeline.reverse(); // reverses the entire timeline
```

The `timeline.to` method creates a tween and schedules it to start at the end of the previous transition. In contrast to D3 (see Section 4.4), the durations and delays of transitions can be altered without needing to adjust subsequent transitions. In addition, GSAP provides another method that gives multiple options to position a tween inside a timeline, e.g., inserting a transition one second after the start of the previous transition. Compared to D3, GSAP vastly reduces the effort needed to implement staggered transitions.

5.1.2 Possible Drawbacks of Using GSAP

GSAP's *"No Charge" License* [19] allows incorporating GSAP into commercial and non-commercial software, e.g., websites or computer applications, free of charge. However, there are two cases where annual fees must be paid:

1. If the developer wants to have access to additional plugins.

2. If GSAP is used as part of a product that is sold to multiple customers. This also applies, if end users are charged a fee to use the product, e.g., subscription-based websites. In this case, a commercial license must be purchased.

I found that the core functionality of GSAP is sufficient to implement a library that fulfills the defined requirements. Thus, no additional plugins are needed. However, the costs that occur when a commercial license is needed may be a barrier for some individuals or smaller businesses.

Including GSAP as a dependency in the library to be developed increases the learning effort (RLeEffort) and may cause difficulties when maintaining the library, as already mentioned in Chapter 4. However, the similarities between D3 and GSAP indicate a relatively low learning effort for persons who are familiar with D3.

In conclusion, GSAP provides substantial functionality that simplifies the implementation of a library that satisfies all determined requirements without any serious drawbacks.

5.2 Implementation Details

Like D3 and GSAP, the functionality of GSAP-ASEQ (the developed library) is accessed via a globally imported object. This object is named `aseq`. It consists of the `TransitionsManager` class and a function (`createTransition`) (see Fig. 5.1). The `TransitionsManager` provides methods to navigate between views and store transitions. When importing GSAP-ASEQ, an additional function (called `gsapTo`) is provided. This function can be accessed via any D3 selection. The implemented library considers the concept points described in Section 4.2.

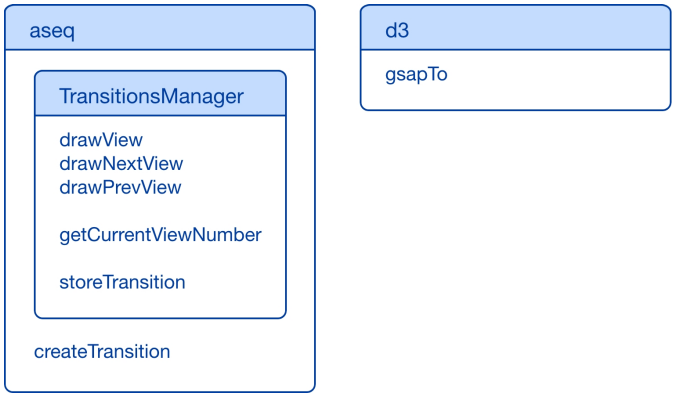

Figure 5.1: Diagram of the exported functions of GSAP-ASEQ.

GSAP-ASEQ is based on creating animated transitions with GSAP. Transitions created otherwise are ignored. The `TransitionsManager` stores a list of all transitions for each step and triggers them when navigating between views. Transitions may be automatically reversed, sped up, or otherwise adjusted as needed.

Every transition that is created in a draw function shall be stored immediately to the `TransitionsManager` instance. Every time a view is rendered the first time, the associated draw function is executed. After that, only the stored transitions are played or modified. Thus, all code that is not executed by a transition is only executed once. This leads to the advantage that, in contrast to D3-ASEQ (see Section 4.3), inside a draw function, no method is required to determine if an element needs to be created or already exists.

Navigating

The `TransitionsManager` class provides three functions to navigate between views.

1. `drawNextView`. If the next view is visited the first time, this function calls the associated draw function. If the view was already visited before, the stored transitions are played. If the transitions of the target view are played in reverse at the time of the function call, the play direction of the transitions is changed.

2. `drawPrevView`. This function works similarly to `drawNextView` but plays the stored transitions of the current view in reverse.

Both, `drawNextView` and `drawPrevView` are implemented to meet the requirements RBiDiNav, RReverseTrans, RHandleTrans, and RCompleteTrans. To support RCompleteTrans, both functions first call a private method to complete currently playing transitions of neighboring views. Active transitions are found by iterating through the stored array of transitions of the corresponding view and examining the progress value of every transition. The found transitions are then sped up to finish in a certain amount of time. This time span can be customized by specifying the `maxViewDelay` when creating an instance of the `TransitionsManager`.

3. `drawView`. This function needs to be given the index of the view to be drawn in the sequence. If the provided view is a neighbor of the displayed view, `drawNextView` or `drawPrevView` is called. Otherwise, `maxViewDelay` is temporarily set to zero and `drawNextView` or `drawPrevView` are called for each view in between the displayed view and the target view. Thus, no animated transitions are perceived as all transitions are directly applied with a duration of zero. This method realizes the requirement RSkipViews.

Transitioning

There are various ways to implement a transition within a draw function. One option is to store a transition by calling the `saveTransition` method of the `TransitionsManager` class. This function accepts any kind of GSAP tween or a timeline and stores it.

The `createTransition` method, provided by the `aseq` object, offers additional functionality to a tween. This method internally creates a tween by using the `gsap.to` method. Therefore, it receives a selector (`target`) and transition properties (`gsapVars`) as input values. In addition, another parameter called `customVars` can be passed. The `customVars` object can hold three different key-value pairs. The keys `autoHideOnReverseComplete` and `autoHideOnComplete` enable the described concept of *Creating and Deleting Elements* in Section 4.2 by automatically changing the style property `display`. Additionally, a function that is called when the corresponding transition is reverted can be specified inside of the `customVars` object.

Another, more D3-like way to implement a transition is using the provided `gsapTo` method that can be called from any D3 selection. Like `aseq.createTransition()` it receives a `gsapVars` and a `customVars` object as input values. As this function also stores the created transition, an instance of the `TransitionsManager` must be passed. Thus, this function animates all elements that are contained in the selection. Moreover, like in D3 transitions, it is possible to animate properties dependent on bound data. This is done by passing `gsapVars` or `customVars` as

functions that return an object depending on the bound data (see the code snippet below).

```
d3.selectAll("rect")
    .gsapTo(manager, (d, i) =>
        { attr: { height: d }, duration: 2 },
        { autoHideOnReverseComplete: true }
    );
```

The firstly passed function to `gsapTo` defines the `gsapVars`. The second object defines the `customVars`. (The `attr` key determines that the `height` is an attribute and not a style property.)

When reversing the transitions of a view, GSAP-ASEQ reverses and plays all corresponding transitions at once. Thus, delays are not taken into consideration. To correctly reverse staggered transitions, a timeline must be used.

Additional Methods

The `TransitionsManager` provides a method to display the transition progress of each view. When creating a `TransitionsManager` instance, an optional *update function* can be defined. This function receives two input values and is called for each frame. First, the progress value of the current transition to a view (a value between zero and one). Second, a boolean value that determines if the transitions are played in reverse. Consequently, this function can be used to visualize the progress of a transition from one view to another view.

5.3 Implementation of the Lead Scenario

The lead scenario was implemented using the library GSAP-ASEQ. The result is published on a website[6]. The corresponding source code[7] is available on GitHub. However, it is not recommended to use the source code for learning purposes (especially for new users), as the lead scenario is a very complex example. An excerpt of the website is shown in Figure 5.2. There are two buttons to navigate to the next or previous view. In addition, the user can navigate to an arbitrary view in the sequence by clicking on one of the numbered buttons. The transition progress of each view is visualized in the form of a gray progress bar.

[6] `https://vinccenttt.github.io/anomaly-heatmap-aseq/` (visited on 02/14/2023)
[7] `https://github.com/vinccenttt/anomaly-heatmap-aseq` (visited on 02/14/2023)

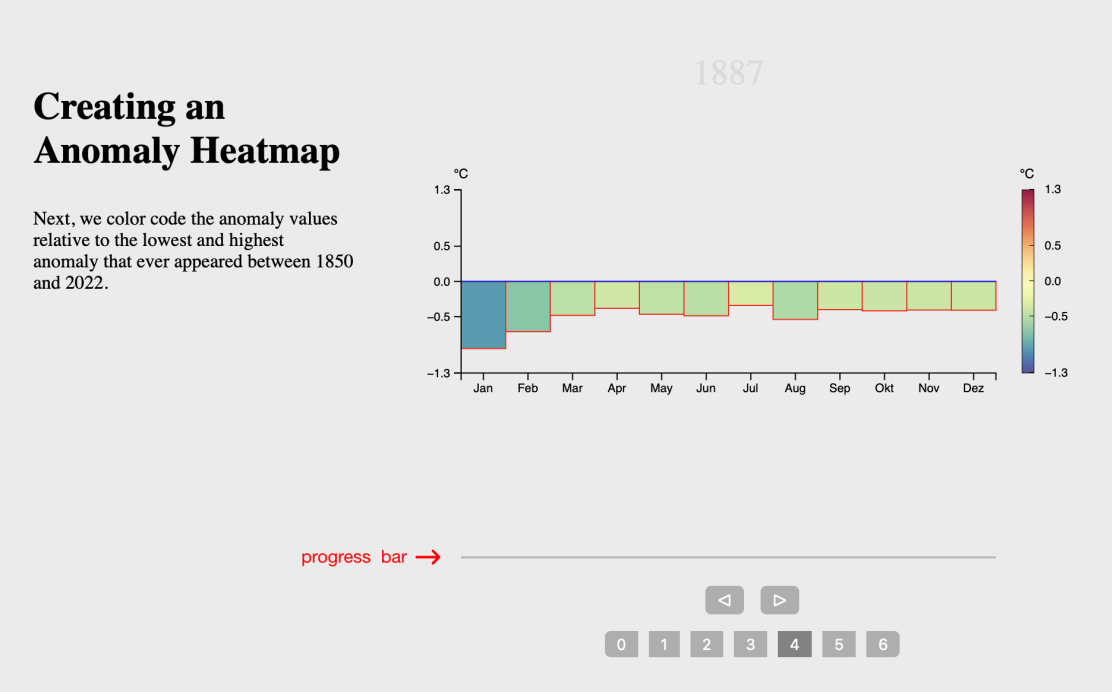

Figure 5.2: Screenshot of the implemented lead scenario. The progress bar is annotated.

The lead scenario demonstrates all visualization-related requirements (see Section 3.2.1). For example, it is possible to skip views and staggered transitions are used.

5.4 Validation

GSAP-ASEQ meets all of the established requirements from Section 3.2 (see Table 5.1). In the following, the fulfillment of the requirements is analyzed in more detail.

Visualization-Related Requirements

RBiDiNav, RReverseTrans, RHandleTrans, RSkipViews, RCompleteTrans, and RStaggerTrans. All these requirements are met by the implementation of GSAP-ASEQ. As already mentioned, this can be observed in the realization of the lead scenario.

RExpress. The developed library exclusively supports transitions that are created with GSAP. As shown in Section 5.1.1, the creation of animated transitions in D3 and GSAP is similar. Like D3, GSAP supports creating an animated transition on a selected element by providing target values for style properties and attributes. When creating a tween with GSAP, many animation properties can be defined. Functions that are invoked on different animation events may be passed. It is

Requirements			
Visualization-Related		Programming-Related	
RBiDiNav	✓	ROneDiProg	✓
RReverseTrans	✓	RCodeLines	✓
RHandleTrans	✓	RLeEffort	✓ (tendency)
RSkipViews	✓		
RStaggerTrans	✓		
RCompleteTrans	✓		
RExpress	✓ (same as GSAP)		

Table 5.1: Result of the validation of GSAP-ASEQ according to the determined requirements from Section 3.2.

also possible to define how many times an animation shall be repeated. Based on the documentation of D3 transitions and GSAP, I suppose that the majority of transitions that can be generated with GSAP can also be created with D3 and vice versa. However, GSAP substantially simplifies the implementation of more complex transitions by providing corresponding functions. Predefined D3 transitions, such as transitioning axes, may be programmed manually when using GSAP. It is also possible to determine style properties or attributes dependent on the progress of a transition (similar to `attrTween` in D3, see Section 2.3.2). This is described in depth in the documentation of GSAP-ASEQ. To conclude, GSAP-ASEQ supports highly customized transitions.

Programming-Related Requirements

ROneDiProg. GSAP-ASEQ supports one-directional programming. The developer only needs to program in the forward direction. The reversing of transitions and skipping of views are handled by GSAP-ASEQ. However, it is possible to provide code that shall be executed at the start or end of a reversed transition, if needed.

RLeEffort. I suppose that the learning effort and complexity of using GSAP-ASEQ are relatively low. The three provided functions to navigate between views do not need any parameters (except the index of the view when calling `drawView`) and no previous checking of any conditions. The possibilities of creating transitions are all very similar and based on the provided methods by GSAP. D3 developers probably get used to GSAP quickly, as both libraries use a declarative approach and the creation of transitions is similar. The required learning effort increases with the complexity of the transitions.

RCodeLines. The lines of code should not increase notably when using GSAP-ASEQ. However, an additional function call to save the transition may be necessary when creating a transition. When initiating a transition with GSAP, all the

properties, e.g., attributes or style properties, are passed as one single object. In D3, the target value of each animated attribute needs to be set using the `attr` operator. This often leads to many calls of the `attr` operator. Thus, compared to D3, GSAP reduces the writing expense.

In conclusion, GSAP-ASEQ fulfills all programming-related requirements.

6

Conclusion

This thesis presented a library that substantially facilitates the implementation of animated sequences, which consist of views that are created using D3. In Chapter 2, I justified the current need and benefits of such a library. After determining the requirements, I presented and assessed various implementation strategies of an appropriate library. The final result, the library GSAP-ASEQ, was validated and showcased in an example. In addition, a publicly available documentation of GSAP-ASEQ was created. GSAP-ASEQ fulfills the majority of the requirements and therefore makes the implementation progress of an animated sequence more efficient. The library will assist future data story coders in implementing individual narratives with seamless transitions and high usability.

6.1 Discussion

The requirements were determined together with an employee of the Otto von Guericke University Magdeburg, who deals with interactive D3 visualizations on a daily basis. As the library to be developed should focus on animated sequences but no specific use case, the main aim was that it should be applicable to a preferably large range of animated sequences. Although I provided arguments for the reasonableness of the requirements, an empirical analysis to identify the requirements would have led to more representative results and objectiveness. This is especially important as the library is not developed for a specific scenario.

Regarding the validation, the fulfillment of the requirements RExpress, RLeEffort, RCodeLines was analyzed argumentatively and not supported by measurements or empirical evaluation. However, the assessment of the expressiveness and the lines of code is challenging if there are no other equivalent libraries to compare to.

The learning effort could have been evaluated by giving tasks to participants and using an appropriate evaluation method. Due to a shortage of time and lacking connections to persons who are experienced in D3, this was not done.

6.2 Future Work

As already mentioned in the previous section, an empirical analysis to determine requirements and an empirical evaluation could be done. The outcome would demonstrate whether the defined requirements and the result of the validation line up with the empirical outcomes. Thus, the mentioned arguments may be affirmed. If the empirical analysis or evaluation leads to new or contradicting findings, the developed library may need to be updated to meet these findings.

The published documentation provides a good overview and introduction to how to use GSAP-ASEQ. However, it would be beneficial to present a simple example that uses the library, e.g., a well-commented source code. In addition, it would be helpful to provide examples of the different kinds of transitions and special cases that might appear during the implementation of an animated sequence. Both aspects may lower the learning curve, improve the library's attractivity for new users and give a better insight into the use cases in which the library is helpful.

GSAP-ASEQ can be imported by referencing it in a <script> tag inside the DOM. Furthermore, providing GSAP-ASEQ as an *npm*[1] package would be useful. Npm is a popular and widely used package manager [20] for the runtime environment Node.js. Npm consists of the *npm CLI* (Command Line Interface) and the *npm Registry* [20]. The npm Registry is a collection of JavaScript packages, including D3.js[2]. According to npm's website [20], the npm Registry is the world's largest software registry.

In the context of this thesis, I focused on using buttons for navigating between views. Thus, GSAP-ASEQ is not applicable to animated sequences that use scrolling as a navigation technique. *Scrollytelling* is a popular technique to create narrative visualizations. Therefore, extending GSAP-ASEQ to support navigation by scrolling would offer additional value.

[1] `https://www.npmjs.com` (visited on 03/10/2023)
[2] `https://www.npmjs.com/package/d3` (visited on 03/10/2023)

Bibliography

[1] Edward Segel and Jeffrey Heer. "Narrative visualization: Telling stories with data". In: *IEEE transactions on visualization and computer graphics* 16.6 (2010), pp. 1139–1148.

[2] Jeffrey Heer and George Robertson. "Animated Transitions in Statistical Data Graphics". In: *IEEE Transactions on Visualization and Computer Graphics* 13.6 (2007), pp. 1240–1247. DOI: 10.1109/TVCG.2007.70539.

[3] Steven Rich et al. *Homicide database: Mapping unsolved murders in major U.S. cities*. URL: https://www.washingtonpost.com/graphics/2018/investigations/unsolved-homicide-database/?tid=graphics-story&city=detroit (visited on 02/17/2023).

[4] Michael Bostock, Vadim Ogievetsky, and Jeffrey Heer. "D^3 data-driven documents". In: *IEEE transactions on visualization and computer graphics* 17.12 (2011), pp. 2301–2309.

[5] Benjamin Bach, Emmanuel Pietriga, and Jean-Daniel Fekete. "GraphDiaries: Animated Transitions andTemporal Navigation for Dynamic Networks". In: *IEEE Transactions on Visualization and Computer Graphics* 20.5 (2014), pp. 740–754. DOI: 10.1109/TVCG.2013.254.

[6] B.B. Bederson and A. Boltman. "Does animation help users build mental maps of spatial information?" In: *Proceedings 1999 IEEE Symposium on Information Visualization (InfoVis'99)*. 1999, pp. 28–35. DOI: 10.1109/INFVIS.1999.801854.

[7] George G Robertson, Jock D Mackinlay, and Stuart K Card. "Cone trees: animated 3D visualizations of hierarchical information". In: *Proceedings of the SIGCHI conference on Human factors in computing systems*. 1991, p. 190.

[8] Thorsten Hansen, Lars Pracejus, and Karl R. Gegenfurtner. "Color perception in the intermediate periphery of the visual field". In: *Journal of Vision* 9.4 (Apr. 2009), pp. 26–26. ISSN: 1534-7362. DOI: 10.1167/9.4.26. eprint:

https://arvojournals.org/arvo/content_public/journal/jov/933534/jov-9-4-26.pdf. URL: https://doi.org/10.1167/9.4.26.

[9] George Robertson et al. "Effectiveness of animation in trend visualization". In: *IEEE transactions on visualization and computer graphics* 14.6 (2008), pp. 1325–1332.

[10] Lauren Wood et al. "Document object model (dom) level 1 specification". In: *W3C recommendation* 1 (1998).

[11] Mike Bostock. *D3.js - Data-Driven Documents*. URL: https://d3js.org/#introduction (visited on 01/16/2023).

[12] Jim Vallandingham. *So You Want to Build A Scroller*. URL: https://vallandingham.me/scroller.html (visited on 01/25/2023).

[13] S. McKenna et al. "Visual Narrative Flow: Exploring Factors Shaping Data Visualization Story Reading Experiences". In: *Computer Graphics Forum* 36.3 (2017), pp. 377–387. DOI: https://doi.org/10.1111/cgf.13195. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/cgf.13195. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/cgf.13195.

[14] H. Allen Anderson. *Big Die-Up*. URL: https://www.tshaonline.org/handbook/entries/big-die-up (visited on 01/25/2023).

[15] Ji Soo Yi et al. "Toward a Deeper Understanding of the Role of Interaction in Information Visualization". In: *IEEE Transactions on Visualization and Computer Graphics* 13.6 (2007), pp. 1224–1231. DOI: 10.1109/TVCG.2007.70515.

[16] Mike Bostock. *GitHub - d3/d3-transition: Animated transitions for D3 selections.* URL: https://github.com/d3/d3-transition (visited on 01/16/2023).

[17] GreenSock. *GSAP - GreenSock*. URL: https://greensock.com/gsap/ (visited on 02/07/2023).

[18] GreenSock. *GSAP Docs*. URL: https://greensock.com/docs/ (visited on 02/07/2023).

[19] GreenSock. *Standard License - GreenSock*. URL: https://greensock.com/standard-license/ (visited on 02/07/2023).

[20] Inc. npm. *npm*. URL: https://www.npmjs.com (visited on 02/14/2023).

Statement of Authorship

I herewith assure that I wrote the present thesis independently, that the thesis has not been partially or fully submitted as graded academic work and that I have used no other means than the ones indicated. I have indicated all parts of the work in which sources are used according to their wording or to their meaning.

I am aware of the fact that violations of copyright can lead to injunctive relief and claims for damages of the author as well as a penalty by the law enforcement agency.

Magdeburg, March 20, 2023

Vincent Göring