DOI: 10.1111/cgf.14005
Eurographics Conference on Visualization (EuroVis) 2020
M. Gleicher, T. Landesberger von Antburg, and I. Viola
(Guest Editors)

Volume 39 (2020), Number 3

Canis: A High-Level Language for Data-Driven Chart Animations

T. Ge[1], Y. Zhao[1], B. Lee[2], D. Ren[3], B. Chen[4], Y. Wang[1†]

[1]Shandong University, Qingdao, China
[2]Microsoft Research, Redmond, WA, United States
[3]University of California, Santa Barbara, CA, United States
[4]Peking University, Beijing, China

Abstract

In this paper, we introduce Canis, a high-level domain-specific language that enables declarative specifications of data-driven chart animations. By leveraging data-enriched SVG charts, its grammar of animations can be applied to the charts created by existing chart construction tools. With Canis, designers can select marks from the charts, partition the selected marks into mark units based on data attributes, and apply animation effects to the mark units, with the control of when the effects start. The Canis compiler automatically synthesizes the Lottie animation JSON files [Aira], which can be rendered natively across multiple platforms. To demonstrate Canis' expressiveness, we present a wide range of chart animations. We also evaluate its scalability by showing the effectiveness of our compiler in reducing the output specification size and comparing its performance on different platforms against D3.

CCS Concepts

• ***Human-centered computing*** → ***Visualization toolkits; Information visualization; Visualization systems and tools;***

1. Introduction

Chart animations are an effective means of attracting people's attention and maintaining their engagement. It is easy to find chart animations from leading journalism outlets, such as the New York Times, the Guardian, and the Washington Post as well as from popular data visualization blogs, such as Flowing Data [Yau] and Visualizing Data [Kir]. Previous research shows that animated charts are more exciting and engaging than static ones in the presentation context, although they might be ineffective for some data analysis tasks [RFF*08, APP10, BLIC19].

To help people craft compelling chart animations, a few highly expressive keyframe tools (e.g., After Effects [Adob]) have been developed. However, most of them require people to manually specify the visual properties for each keyframe, which is a tedious and time-consuming procedure. As such, a few template-based tools (e.g., DataClips [AHRL*16], Flourish [Ltd]) have been proposed to lower the manual burden. They allow people to select a template with pre-defined visualizations types (e.g., bar charts, line charts) and animation effects (e.g., creation, cycling). Although such tools improve the animated chart creation process, their expressiveness is limited by templates.

Declarative languages such as Vega [SRHH15] and Vega-lite [SMWH16] enable a concise specification of visualizations, but they do not provide the ability for authoring chart animations. Although D3 supports animation through transition, it requires the designers to contend with *how* the charts should be evolved over time. The recently proposed grammar of animated graphics, gganimate [PR] alleviates this problem by encapsulating the design of animation effects, but it is designed specifically for the statistics software package R with limited animation effects. Furthermore, the animations created by these tools are difficult to natively deploy on different platforms, further limiting their applicability.

In this paper, we present a concise, declarative language, called *Canis*, the first domain-specific language (DSL) to provide comprehensive support for constructing chart animations, which can be rendered natively across multiple platforms. Our goal with Canis is to facilitate the easy creation of meaningful chart animations while striking a balance between effectiveness and expressiveness. In achieving this goal, we make the following three contributions.

First, we contribute a high-level grammar that enables declarative specifications of data-driven chart animations. By leveraging data-enriched Scalable Vector Graphics (SVG) charts, this grammar can be applied to any charts created by existing chart construction tools. With Canis, designers can select marks from one or more data-enriched SVG charts, (hierarchically) partition them into a set

† Yunhai Wang is the corresponding author.

Figure 1: *The example input dSVGs and Canis specifications for animating two different charts with different animation effects: (a) a donut chart with the "wheel" effect and a bar chart with the "wipe bottom" effect. (**Open this PDF in Acrobat Reader to view the animation.**)*

of mark units based on data attributes, and apply pre-defined or customized animation effects to the mark units, with the control of when the animation effects start. For example, Fig. 1(a) shows the Canis specification that selects the SVG path element and applies the "wheel" effect for the animated creation of a donut chart. (See Section 4 for the detailed description of the Canis spec.)

Second, we present a Canis compiler (Section 5) that automatically synthesizes Lottie JSON specifications, a popular animation library [Aira], that can render animations natively on desktop, web browsers, and any mobile devices. After parsing Canis JSON specifications, the Canis compiler performs build-bind-evaluate operations that build mark unit trees, bind specified animation effects to each leaf node of the mark unit trees, and compute the starting time of each animation effect. With such information, the compiler translates the input specifications into the Lottie specifications with a minimum size.

Lastly, we provide two forms of evaluation (Section 6). We illustrate the expressiveness of Canis through a wide range of examples. We also evaluate the scalability of Canis by demonstrating how our compiler reduces the size of output Lottie specifications and comparing its rendering performances against D3 on different platforms: Lottie web (desktop), Lottie native (mobile), Lottie web (mobile), D3 (desktop), and D3 (mobile).

2. Related Work

Canis is related to research in interactive visualization systems for authoring chart animations, visualization and animation toolkits, and animation effectiveness.

2.1. Interactive Visualization Systems

A few interactive systems can be used for authoring chart animations, and they take a keyframe- or template-based approach. For example, Adobe After Effects [Adob], one of the most graphically expressive tools, uses keyframes to set parameters for motion, effects, audio, and many other properties. However, it is usually tedious and time-consuming to fine-tune multiple keyframe visual properties and synchronize different animation effects for different marks. On the other hand, other interactive systems such as Adobe Stock [Adoa], Flourish [Ltd], and DataClips [AHRL*16] provide data-driven templates to enable the easy creation of chart animations. Because they rely on pre-defined templates, it is hard to create novel expressive animations with such systems.

2.2. Visualization and Animation Toolkits

A complete review of the approaches to visualization authoring is beyond the scope of this paper. We refer the reader to Grammel et al.'s survey [GBTS13] and focus our discussion on programming-based approaches for authoring visualizations. Since the pioneering work of Wilkinson on *The Grammar of Graphics* [Wil99] in 1999, a variety of grammars have been developed for specifying visualizations. They can be categorized into low-level and high-level grammars. Low-level grammars such as Protovis [BH09], D3 [BOH11], and Vega [SRHH15] are more expressive and thus have been widely used for creating explanatory and highly customized visualizations. However, they involve a steep learning curve because everything has to be specified at a low level. In contrast, high-level grammars such as ggplot2 [Wic10] and Vega-Lite [SMWH16] are easier to learn but less expressive. Such grammars allow authors to generate visualizations with concise specifications, where the omitted details are filled by smart default values. In a similar spirit, Canis allows authors to create chart animations with concise, high-level specifications.

The popular websites and mobile apps leverage animation to improve user experience as well as to attract people's attention and maintain their engagement, and many UI animation libraries have been developed [HEN]. To build high-quality expressive animations with these libraries, authors need to tune low-level animation properties and synchronize multiple instances. In contrast, some declarative libraries (e.g., Qt Quick [RZ10]) support a high-level specification of animations and transitions. However, all of these libraries are targeted for the animation of general visual elements, rather than data-driven visual marks in visualizations. As such, it is highly tedious and time-consuming to create animated visualizations based on data using these libraries.

Although scarce, there have been research efforts on visual-

ization libraries to facilitate the authoring of animated visualizations. D3 [BOH11] provides a *transition* operator for implementing animated transitions between selected visual elements. Stardust [RLH17] leverages GPU to significantly improve the rendering performance while providing a similar API as D3. While these tools enable authors to craft expressive animations, they need to manually build the mapping between the properties of visual elements and the desired animation effects, which is a challenging and time-consuming task.

The gganimate [PR], an extension of ggplot2 [Wic10], enables authors to create an animation of multiple charts of the same type, where each chart shows a single data subset. It is mainly used for showing updates of data, but might not support several other animation types [HR07, AHRL*16].

Canis aims to balance the expressiveness and conciseness. On the one hand, it is a high-level grammar and uses a portable JSON syntax for specifying animated visualizations like Vega-Lite. On the other hand, Canis allows authors to customize animation effects for creating highly expressive animations. It not only covers all types of animations provided by DataClips [AHRL*16] but also supports any chart created by existing visualization tools. Since all specifications are compiled to Lottie JSON specifications, which enables the animations to be used as easily as static images.

2.3. Animation Effectiveness

The earlier work by Tversky et al. [TMB02] suggests that chart animation is attractive in presentations, but might not be as effective as the static chart in some data analysis tasks. Archambault et al. [APP10] evaluate the effectiveness of animated dynamic graphs in mental map preservation, while Robertson et al. [RFF*08] and Brehmer et al. [BLIC19] evaluated the effectiveness of animated scatterplots for trend visualization. All results show that animation is still preferable in some data analysis cases.

To help people better perceive changes during animated chart transitions, Heer and Robertson [HR07] developed a taxonomy of animation transition types and contributed guidelines for crafting animated transitions between statistical graphics. Afterward, different aspects of animated transition design have been studied. Dragicevic et al. [DBJ*11] investigated the temporal distortion of animated transition and found that slow-in/slow-out outperforms other techniques. Chevalier et al. [CDF14] studied the staggering strategy, which incrementally delays the start times across the moving elements to reduce occlusion, and found that the staggering strategy has negligible or even negative influences on multiple objects tracking performance. The Canis design strives to follow the guidelines and lessons from these research when relevant, e.g., when handling the input of multiple charts with animated transitions.

3. The Canis Design

In this section, we first introduce the design goals of Canis, and then briefly describe the input SVG and output specification of Canis.

3.1. Design Goals

With the goal of providing comprehensive support for constructing chart animations, we settled on the following three design goals (DGs) for our chart animation specification language. Here, we assume that authors have knowledge about the data patterns and the desired story to be presented.

DG1: High-level Specifications. D3's transition operator provides a way to produce expressive chart animations by manipulating the interpolation of visual marks over time. However, it is time-consuming and tedious to create expressive animations by manipulating such low-level details. Instead, Canis decouples specification from implementation and enables concise, high-level specifications of expressive chart animations.

DG2: Meaningful Partitions and Sequencing. Since data is encoded by visual marks, conveying data patterns with animations requires to effectively organize visual marks for ordering animation sequences. Inspired by the *partition* operator of data illustrator [LTW*18], Canis treats all data-encoded marks as one unit and partitions it into a set of elementary mark units in terms of data attributes. Mapping each unit to one time interval facilitates a meaningful ordering of animation sequences.

DG3: Cross-platform Deployment. Being able to natively run animations on different platforms improves their applicability and utility. Thus, to make it easier for developers to create chart animations targetting multiple platforms including Android, iOS, Windows, and Web, we compile Canis specifications into widely-used animation specifications (i.e., Lottie JSON files [Aira]).

3.2. Data-enriched SVG

A chart animation shows changes of visual marks over time for conveying data patterns of interest. By taking SVG charts as the input, Canis can be agnostic to the visualization authoring libraries and tools. However, using SVG information alone is typically not enough to characterize all data patterns with animations, since it is usually not easy to directly extract accurate data values from the marks [HA14]. To effectively generate meaningful animations, Canis uses a data-enriched variant of SVG (dSVG)[†] that embeds the source data into the SVG chart, allowing the Canis renderer to use this information during the generation of the animation.

Specifically, dSVG adds three properties "id," "class," and "datum" for each mark. The first two properties provide the index and SVG element type (e.g., rect, circle, path) of the mark, while the "datum" property contains the associated data. Fig. 1 shows two examples of dSVG files, where these additional properties are highlighted in bold.

[†] All SVG charts created by D3, VEGA, and Charticulator can be easily converted into dSVG files with our dSVG generator, which is available at https://canisjs.github.io/marker.

3.3. Lottie

To achieve DG3, we define the output Canis specifications as Lottie JSON files. Lottie [Aira] is an open-source library developed by Airbnb that parses After Effects animations into a JSON file and renders them in real-time on Android, iOS, Windows, and Web. Hence, it enables engineers to effortlessly incorporate rich animations created by designers into their products without painstaking efforts of re-writing them. Moreover, the Lottie JSON files can be further edited by different tools, such as Lottie Editor [Airb] and Keyshape [Key]. However, Lottie works only with After Effects, which inherently does not support data-driven animations. By exporting Canis specifications as Lottie JSON files, Canis uses Lottie to render animations in different platforms.

4. The Canis Grammar

Canis is a high-level declarative language that allows describing expressive chart animations. To meet DG1, it provides a novel high-level grammar that abstracts low-level details by directly specifying animation effects to selected marks.

A Canis specification describes how to animate given *charts* with a sequence of well-defined animation units, we call *aniunits*, and an optional view composition operator *facet*:

$$Animation := (charts, aniunits, facet). \quad (1)$$

Based on the previous studies [AHRL*15], we assume that each animation unit consists of four components: timing, mark selector, mark grouping operator for partitioning marks to specify keyframes, and animation effects imposed to the selected marks. In this way, the animation unit for input SVG charts can be specified via a quadruple:

$$aniunit := (timing, selector, partitioner, effects), \quad (2)$$

where the detail of each is described below. Note that *timing*, *partitioner*, and *effects* are all defined in terms of data attributes and thus we need to enrich the input SVG charts with data.

4.1. Timing

The *timing* component is to control when an animation starts and how the marks are performed in a staggering or staging manner [CDF14]. It consists of two attributes: *reference* and *delay*. The *reference* value is either "*start with previous*" or "*start after previous*" that determines if this animation unit starts at the same time with the previous unit or after. The *delay* attribute specifies the number of milliseconds before the animation of this unit starts, which is defined as:

$$delay := constant | (field, minDelay), \quad (3)$$

where *minDelay* is the minimum delay for the mark with the smallest value of the corresponding quantitative attribute *field*. The delays of the other marks are obtained by linear interpolation based on the attribute values. The default values of these two attributes are "start with previous" and 0, respectively.

4.2. Mark Selector

Given the data-enriched SVG charts, the marks to be animated are selected by using the W3C Selectors API [EBR14]. For example, with *"selector": ".bar"*, all bars are selected from the input charts. In line with D3 [BOH11], this operator also supports conditional selections and sub-selections. If marks cannot be found, the corresponding animation unit will be ignored. Note that marks refer to all visual primitives shown in charts such as axis grid and tick marks, not only the data-encoded graphical marks.

4.3. Mark Partitioner

The selector can select multiple types of visual marks to be animated and the sequencing of different types of marks can be specified by using the *timing* parameters. If the number of marks of the same type is one, this mark is the elementary *unit* for animation; otherwise, the marks of this type need to be partitioned into a set of elementary *units*. Based on such units, we define keyframes by mapping each unit to one keyframe and change the mark properties of this unit in the corresponding time interval, while the other units remain constant.

To generate meaningful partitions and sequencing (DG2), our mark grouping operator groups marks in terms of data attribute:

$$grouping := (timing, groupBy, sort), \quad (4)$$

where the *timing* parameters adjust the order between mark units and the *groupBy* string denotes a data attribute or the data index *id* for grouping. By default, a categorical or nominal attribute is used for this operation. Taking all marks as a whole, applying this operator results in an internal two-level mark unit tree, where each leaf node corresponds to a mark unit. Note that if the reference of the *timing* parameter is specified as "start with previous," all partitioned units will be shown simultaneously as a whole.

With a nested *partition*, a multi-level mark unit tree can be formed. Fig. 2 shows an example, where the marks are first grouped by the `Surface` attribute, then by the `Odor` attribute, and then finally by the `IsEdible` attribute. In this way, a four-level mark unit tree is formed, and each leaf node consists of one or multiple dots in the same color as shown in Fig. 2(c). Namely, this animation has seven keyframes, where the dots of the fifth and sixth frames have the same values (Fishy and Smooth) for the `Odor` and `Surface` attributes.

If the *groupBy* attribute is not data index, the ***sort*** operator can be used to specify the animation order of the mark units by their unique values:

$$sort := order([value_1, value_2, \cdots]).$$

Otherwise, the sort operator can be defined as:

$$sort := (field, ordering),$$

where the *field* is one data attribute; *ordering* can be descending or ascending. If the *sort* operator is not specified, the groups are ordered in terms of the corresponding data index.

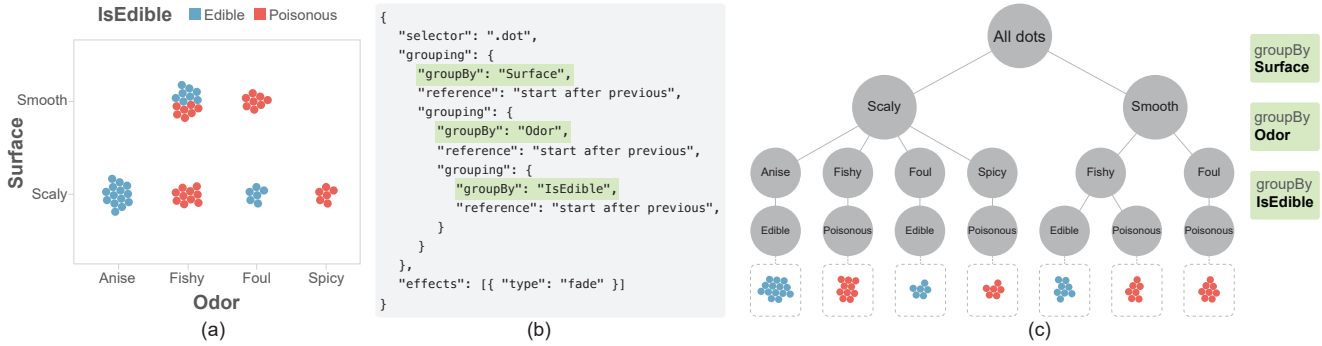

Figure 2: *Specification for animating the faceted dot plot with a nested partitioning: (a) input scatterplot encoding three variables (`Surface`, `Odor` and `IsEdible`); (b) Canis specification where three variables used for the nested partitioning are highlighted; and (c) mark unit tree for partitioning the marks into keyframes.*

4.4. Animation Effects

Each animation unit can have one or multiple animation effects:

$$effect := (timing, type, channel, easing, duration). \quad (5)$$

The *type* specifies an animation effect that is applied to the selected marks: Canis currently provides six types of animation effects: "fade," "wipe," "circle," "wheel," "grow," and "magic move." Different effects correspond to different visual *channels* of the selected marks, for example, "fade" corresponds to the opacity and "grow" to the height. For achieving such effects, we pre-define how these channels change for each effect.

Fig. 1 illustrates how three different effects change the visual properties of different marks. Besides these effects, users can customize the effects by setting *type* as "custom" and directly defining how the value of the visual channel should change from one value to another by using a triple:

$$attribute := (channel, from, to),$$

where *channel* refers to the visual channel like height or opacity, and *from* and *to* are the two transition values.

In addition, we introduce an effect we call "magic move" [Kis15] for morphing between two SVG visualizations to achieve smooth animated transitions. Here, the input charts must consist of multiple SVG charts rather than a single chart and all types of marks are animated with this effect. If users select several types of marks to be animated with specific effects, the other marks will be animated with the effect "magic move," which has the same time duration with the user-specified effects. Note that the multi-staged transition suggested by Heer and Robertson [HR07] can be implemented by adding more intermediate SVG visualizations.

SVG Mask-based Effects. Although most animation effects can be specified by changing mark properties, some effects cannot be easily produced by directly manipulating mark channels. For example, the "wipe bottom" effect shown in Fig. 3(a) need to change y position and height of the rectangle, while the same effect is hard to generate for Fig. 3(b), because the mark wedge defined by the SVG *path* element does not have the height attribute. Although it is possible to achieve such effects by using low-level path attributes, it is complicated and inefficient.

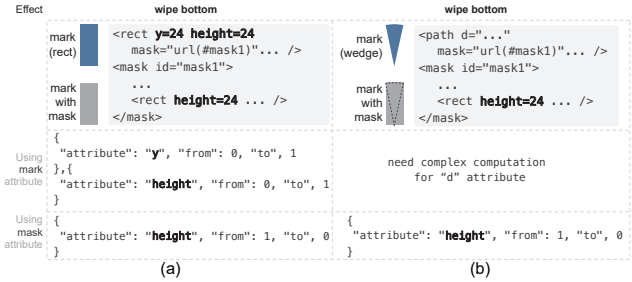

Figure 3: *Implementation the animation effects by using mask attributes (a,b), highlighted in bold. (a) using one mask attribute allows us to achieve the "wipe bottom" effect instead of using the two attributes; and (b) for the marks defined by SVG path element, using the mask attribute allows us to achieve the animation effect easier than by using mark attributes.*

To address this issue, we define SVG masks for such marks and use their attributes to generate desired animation effects. In Fig. 3, the "wipe bottom" effect is implemented by only using the mask's height attribute. Note that such a mask-based implementation is not accessible to users, and the grammar for customizing animation effect is based on the mark attributes rather than mask attributes.

Easing and Duration. Given the specified effect, *easing* indicates the easing function used to change the corresponding mark property over time, while *duration* controls the length of the animation effect. It can be specified as either a constant duration for all marks or constant speed. For the latter one, the duration is defined as Eq. 3, where the minimum duration and an additional quantitative data attribute are specified. The duration of each mark is then computed by linear interpolation in terms of the given data attribute.

4.5. View Compositor

To facilitate side-by-side comparison, our specification also allows for setting up multiple-view animation by introducing the *facet*

operator. It produces a trellis plot by assigning charts to different views with the signature:

$$facet := (orientation, views).$$

Orientation indicates the layout direction of views (i.e., vertical (*row*) or horizontal (*column*)) and *views* is a list of views, where each view specifies a set of chart *id*s as the input. By default, each view has the same number of input charts: the same animation unit is applied to the selected marks of the same type in each view.

5. The Canis Compiler

Taking one or multiple static charts as input, the animated visualization conveys the changes of mark visual properties over time, for all marks. In doing so, there are two challenges in obtaining the corresponding status of the visual properties of each frame. One challenge is posed by potential mistakes in the specification, like some improper data attributes mapped to animation properties. The other is that storing the status of the changed visual properties of all marks of each frame will result in high memory costs, which will affect the rendering performance. Even only storing keyframes can be too large for visualizations with a large number of marks or complex graphical elements.

To overcome these challenges, the compiler generates the output Lottie specification in five distinct phases: *parse* ingests the JSON specification; *build-bind-evaluate* constructs a mark-unit tree for each animation unit, binds the animation effect to leaf units at proper time, and computes the necessary internal representations; and *translate* generates the output Lottie JSON specification for achieving DG3.

5.1. Parse

The *parse* step produces a complete specification by applying rules crafted to produce valid animations [TMB02]. Specifically, if the animation parameter is assigned an invalid value or unspecified, it is replaced by the default value. For example, if a quantitative attribute is used for mark partitioner (see Eq. 4), the attribute is binned into a small set of data ranges and the partitioning is based on these ranges. The above-mentioned effect of "magic move" will be re-defined by the "custom " effect, where the *from* and *to* values are extracted from the corresponding SVG marks. For each animation unit, the compiler will check if there is any animation effect left, otherwise, the unit will be removed.

5.2. Build-Bind-Evaluate

Once a specification is parsed, our compiler analyzes it to collect the necessary information for rendering the chart animations in each view with the steps *build-bind-evaluate*. Rather than storing the mark status at each frame, we build an animation keyframe table with each row describing when a specific animation effect is imposed on one mark and how long this effect lasts. However, storing all information into such a table is redundant in two ways: (1) marks that belong to the same group or even different groups might be assigned the same animation effects and (2) multiple animation effects might involve the same visual channels and thus it is unnecessary to store the visual channel information in each row.

Therefore, we decompose the animation keyframe table into three additional tables: mark unit table, animation effect table, and mark channel table. The animation keyframe table consists of four columns: *unitID*, *effectID*, *start time*, and *duration*, while the mark unit table defines the relationship between unit and marks, with *unitID* and *markID* columns. The animation effect table stores effect properties, such as *effectID*, *channel type*, *from value*, *to value*, and *easing type*, while the mark channel table stores the marks' visual channel values related to the specified animation effects. Fig. 4 shows the construction of these tables with a grouped bar chart.

Note that the duration of each effect is stored within each row in the keyframe table not in the animation effect table, because the duration of the same effect bound to different marks might be different if it is specified via the option of *constant speed*. Moreover, if the *groupBy* attribute is data index, the mark unit table is unnecessary, because each group only contains a single unique mark.

Build. This phase first collects all related marks in each *aniunit* with the *selector* operator and then partitions all the marks into different units in terms of their *groupBy* attribute values. If a nested partitioning is specified, a hierarchical mark unit tree will be formed, where the internal nodes on each level correspond to one *groupBy* attribute and each leaf node corresponds to one mark unit. Fig. 4(b) shows a three-level mark tree built for the grouped bar chart, where each mark unit contains two bars. If a unit contains more than a single mark, we insert the *unitID* and their marks as one row into the mark unit table.

Bind. This phase updates the animation effect table and mark channel tables by parsing the animation effect operator. For example, if the effect "wipe left" is found, a new row recording the width changing from 0 to 100% with a unique *effectID* will be inserted into the animation effect table, and a new column with the widths of all selected marks are added to the mark channel table. Suppose there are l effects in one unit, we insert l rows for each unit into the animation keyframe table by assigning the *unitID*, *effectID*, and effect *duration* to the corresponding cells. Meanwhile, a list of l effects is created and attached to each mark unit. By doing so, all three tables are updated as shown in Fig. 4(c).

Evaluate. This phase computes the starting time of each mark unit listed in the animation keyframe table, based on the reference type and delay time defined in the animation unit. Since each level of the mark unit tree and each effect might have different reference types and delay time, the computation is achieved by traversing the mark unit tree in a depth-first order and processing the effect list of each leaf node. After initializing the starting time of each mark unit to 0, the duration of each mark unit is computed by processing all the effects binding with it and then the starting time is refined with the timing information on each level of the mark unit tree in a bottom-up manner.

The keyframe table in Fig. 4(d) lists the *start time* of each effect bounded to the keyframes. The delay between mark units on the

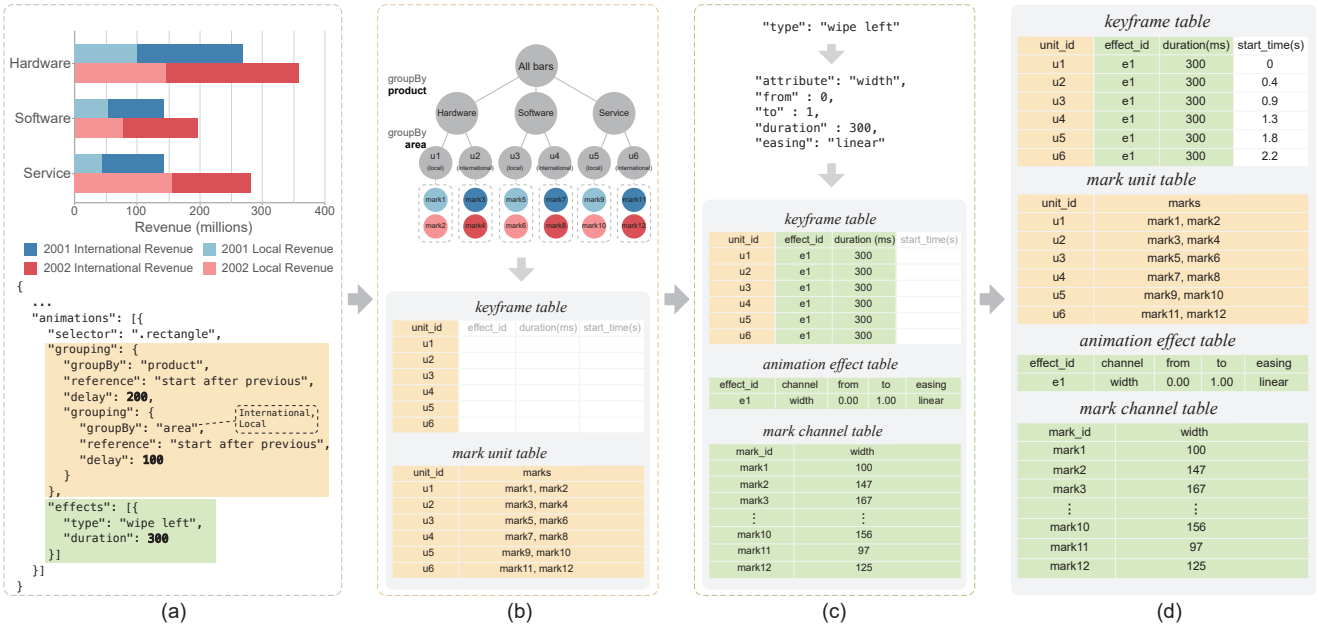

Figure 4: *Illustration of the* build-bind-evaluate *three steps with a grouped bar chart. (a) The input of the grouped bar chart and the* unit *specification; (b) the* build *phase constructs the mark unit tree and updates the mark unit table and animation keyframe table; (c) the* bind *phase associates the animation effect into the mark units and updates the animation effect table, mark channel table and animation keyframe table; and (d) the* evaluation *phase computes the starting time of each effect listed in the keyframe table. Note that the involved information for mask partitioning and animation effects are shown on yellow and green background, respectively.*

second and third levels are 200ms and 100ms, respectively; and the duration of each bar animation is 300 ms. Hence, the whole animation lasts for 2.5s.

As for the specification with multiple views, each view has an individual mark unit tree while different views have the same animation effects. To reduce the number of redundant representations, we re-use the animation effect table for all views, while constructing the other three tables for each view.

5.3. Translate

The Lottie specification describes an animation by defining how the marks of the keyframes evolve over time. This is akin to the motivation of constructing the four internal tables shown in Fig. 4, facilitating the translation from our specifications to the Lottie JSON specifications. Thus, the transition can be done by taking each mark as one object in Lottie and then assigning the corresponding properties in the keyframe table to the object. Since Lottie has its own syntax and terminologies to describe animation, the properties in the keyframe table will be translated to the corresponding Lottie object properties. For example, we need to approximate the specified easing functions with Bessel curves in Lottie.

However, for visualizations of large data, the output Lottie specifications might be redundant, affecting the rendering performance. The main reason is that most marks of the same type are different in several attributes, while storing each of them with the full specification is unnecessary. To produce a Lottie specification with minimum size, we specify the graphical elements as reusable templates by using the Lottie reference mechanism. We introduce two kinds of templates: i) *static template* including fonts, images, and shapes; and ii) *animated template* which includes the mark and its visual channel specifications over time. If the marks can be generated by applying affine transformations or adjusting opacities or timestamps of such (animated) templates, they can also be specified by using the template with the transformations; otherwise, the marks will be stored in the library and referred in the specification. Taking the bar chart animation with the effect “wipe bottom” as an example, the animation of different bars can be specified by one template bar with the changes in scale and position. The effect of this strategy in reducing the file size is evaluated in Section 6.2.

6. Evaluation

In this section, to demonstrate how Canis enables the specification of expressive chart animations we first present a wide range of examples [RLBHR18]. We then evaluate its scalability by measuring the output specification size and comparing the rendering performance on different platforms against D3.

6.1. Examples

The expressivity of chart animations is determined by two orthogonal factors—visualizations and animations—and thus we choose examples that reveal more variations in the design space. On the

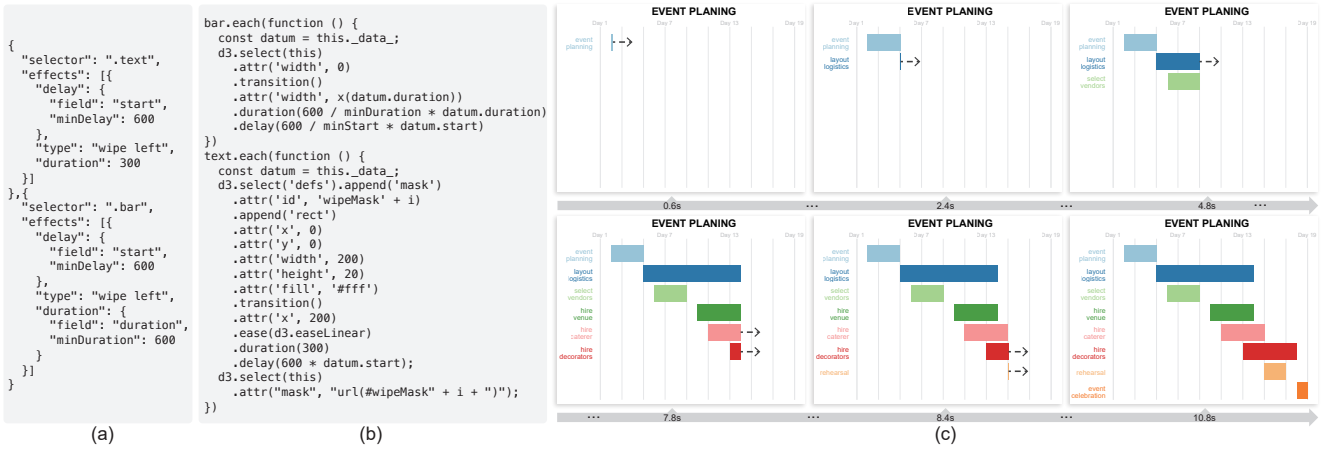

Figure 5: *The animation of creating a Gantt chart designed by using Canis and D3. (a) The partial specification of Canis where two marks "text" and "bar" are selected and specified with same effect but different* timing *parameters; (b) the partial specification of D3, where two functions are provided for animating each type of marks; and (c) six snapshots of the animation, where the arrow indicates the text and bar movement direction. Some chart titles and legends are partially occluded because of the overlap between snapshots.*

one hand, our examples feature a variety of visualizations created by D3, Vega-lite, and Charticulator. On the other hand, we choose examples that cover Amini et al.'s [AHRL*15] taxonomy of animation types, including *creation/deconstruction*, *cycling*, *accumulating*, *transition*, *drill-down and roll-up*, *annotations*, and *multi-views*. Since each chart animation might combine multiple types of animations together, we systematically vary our examples in terms of visualization types and animation types.

Specifically, we compare the conciseness (as a surrogate of speed) of Canis with D3 by using the animations of three bar charts variants and show the animations of multiple charts in the context of storytelling. Complete specifications and animations of all examples can be found in the supplemental material. Note that D3 combines the chart creation and animation generation together and thus it only requires to update data once the chart animation is set up correctly. In contrast, the input of Canis can be the SVG charts generate by any visualization tools.

6.1.1. Bar Charts

In this section, we demonstrate the effectiveness of Canis in authoring three variants of bar charts (grouped bar charts,[‡], Gantt chart, and race bar charts) and compare its specifications again D3 examples for assessing authoring speed.

Gantt Chart. Fig. 5(c) shows snapshots of the *creation* animation of the Gantt chart, where each bar illustrates the event timeline. From the Canis specification in Fig. 5(a), we can see that the marks of text and bar are selected and are associated with the same effect "wipe left." Since the reference parameter is omitted, the animations of these two types of marks start at the same time. The delay

[‡] Due to the limited space, we put the the grouped bar chart example in the supplemental material.

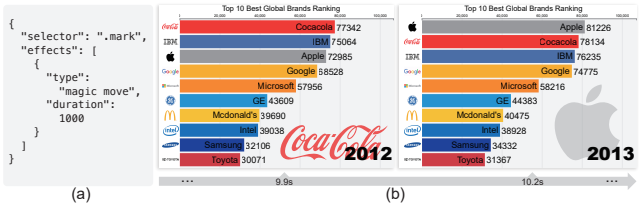

Figure 6: *Authoring the bar chart race animation with the input of multiple data-enriched bar charts by using Canis. (a) The partial specification; and (b) two consecutive snapshots of the animations.*

amounts of both marks are determined by the quantitative data attribute `start`, indicting that two marks encoded by the same data item have the same delay. Likewise, the animation duration of each bar is determined by the data attribute `duration`, which helps users to perceive how an event proceeds.

Fig. 5(b) shows the corresponding D3 specification, where two functions are provided for animating both types of marks. We can see that users not only need to manually build the relationship between the timing parameters and data attributes of the animation, but also have to carefully use SVG mask to animate the text. Here, a rectangular mask is created for each text mark and its x position is changed over time to achieve the effect "wipe left." In contrast, with Canis, user does not deal with implementation details, where users only need to specify the animation effects to the selected marks.

Race Bar Charts. Canis also enables the animated transition between multiple charts. Taking multiple ranked bar charts as the input, users can specify a bar chart race animation with the effect "magic move" (see Fig. 6), corresponding to the *cycling* animation in DataClips. As shown in Fig. 6(a), all types of marks are selected: icons, bars, text, logo and year, which are animated at the same time with "magic move." Since different types of marks need to change

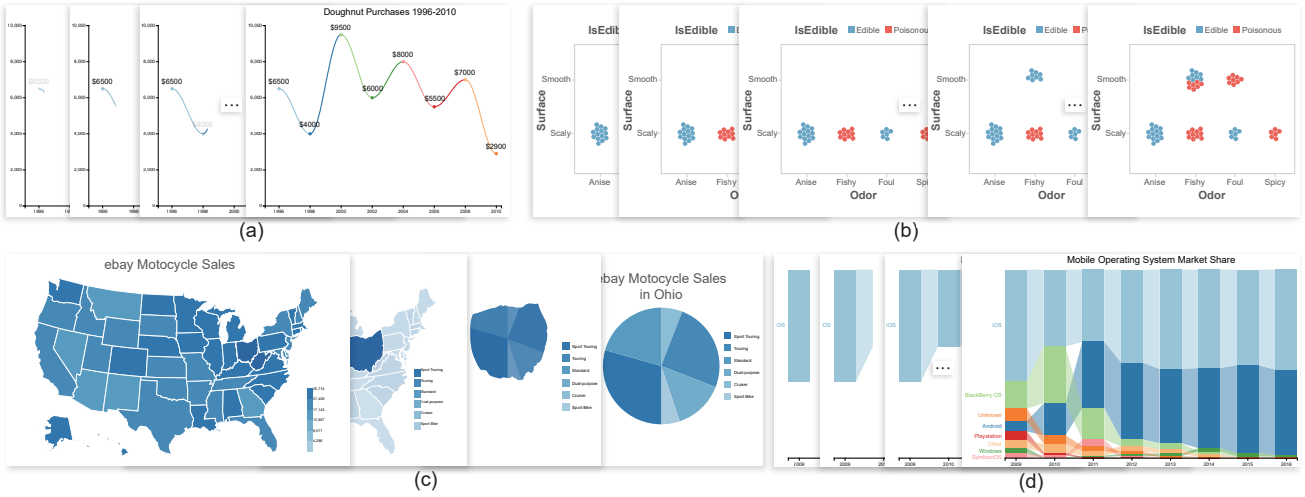

Figure 7: *Sequences of multiple snapshots of chart animations for four chart types—(a) line chart; (b) faceted dot plot; (c) map; and (d) custom chart—showing the expressiveness of Canis, where the snapshots of each animation includes the first and last frames. The complete specification and animations and more examples can be found in the supplemental material.*

different visual properties to achieve this effect, their parsed "custom" animation effects are also different. Fig. 6(b) shows two consecutive snapshots of this animation, where the position and length of the "Apple" bar both are changed from 2012 to 2013, while the SVG icon is completely updated.

6.1.2. Other Chart Types

To further demonstrate the expressivity, we create a variety of animations with a diverse collection of input data-enriched SVG charts. Due to the limited space, we only show the snapshots of four animations in Fig. 7 and provide more examples in the supplemental material.

Line Charts. Annotation is an essential element in conveying key points in visual data-driven storytelling. Fig. 7(a) illustrates an animation of an annotated line chart with four snapshots. Key points and their corresponding annotations appear with the "fade in" effect while the line grows steadily at the same time. The synchronization between the animations of these two types of marks are achieved by setting the duration and delay of the "fade in" effect as 200ms and 400ms, and the duration of the "growing" effect 600ms.

Faceted Dot Plots. A custom chart layout facilitates the expressive visualization authoring. Fig. 2(a) shows a faceted dot plots using circle packing sub-layout. To animate the creation of this scatterplot, we partition all marks with three data attributes (`Odor`, `Surface` and `IsEdible`) and bind each mark unit with the "fade" effect, see the specification in Fig. 2(b). By setting the reference term to "start after previous," the mark units in the first row appear first and then the ones in the second row.Fig. 7(b) illustrates this animation with five snapshots.

Maps. The *drill-down* animation [AHRL*16] is used for transitioning from a subset of marks in the input chart to another visualization, which provides more detailed information. Fig. 7(c) shows an example with four snapshots, where the first frame depicts the eBay sales number of motorcycles at each state and the pie chart in the last frame shows the percentages of motorcycles used for different purposes in Ohio.

Bespoke Charts. Here, we apply Canis to specify animations for a bespoke chart created with Charticulator [RLB18]. In this example, the input chart is the last snapshot shown in Fig. 7(d). It visualizes the share values of the yearly mobile operating system market, where the values of these systems for each year are encoded by a normalized stacked bar chart. Since the bars of each system are connected by bands, the band crossing indicates a rank change. To illustrate how such rank changes over years, we specify an animation that creates the bars and bands of each system year by year and gradually adds the marks of different systems to the chart. Fig. 7(d) shows four snapshots of this animation, where the creation of these systems follows their order in the stacked bar chart of the year 2009.

6.2. Scalability

To evaluate the scalability of Canis in animating the charts of large data, we measure the output Lottie specification sizes and frame rates during rendering different Lottie specifications on multiple platforms. For completeness, we perform two comparisons by : i) computing the compression ratios between the output specification sizes generated with and without template-based translation and ii) measuring the performances (in fps) of rendering our output specifications and D3 counterparts.

Setting. Since Lottie specifications can be run natively on desktop, web and mobile devices, we compare the performances of five versions: Lottie web (desktop), Lottie native (mobile), Lottie web (mobile), D3 (desktop) and D3 (mobile). The desktop test is on a Intel Core i5-8400 CPU, 8GB memory, and NVIDIA GeForce

GTX 1060 card with Windows 10 operating system, while the mobile test was on an iPhone 7 with iOS 13.2.3 system, a 2.34GHz quad-core Apple A10 Fusion processor, 2GB RAM memory and the Safari web browser.

Animations. The tested animations are designed in terms of changing different visual channels, such as length, position, and color and we selected three types of charts—bar charts, line charts, and scatterplots—as the input. For each type of charts, we randomly generated 10 charts with varying number of marks from 100 to 10k, where the height or position of marks in each chart are randomly decided. For the bar charts and line charts, all marks are animated at the same time with the effects "wipe bottom" and "fade in," while the points in scatterplots are moved from one position to another random position with the effect "magic move."

Results. Fig. 8(a) shows two snapshots for each type of specified animations with one input chart. For each chart, we compute the compression ratios of the corresponding specifications resulted by the template-based translation. Fig. 8(b) summarizes the results with the boxplots, where the compression ratio is between 3.9 and 5.3 for all charts. We can see that our compiler largely reduces the output Lottie specification size.

Since the rendering performances of all three animations are similar, we only show the frame rate curves of the line chart animations for five settings in Fig. 8(c). We can see that the frame rates of all settings gradually decreases with the increasing number of marks. However, D3 is not able to render the animation over 2k marks on the mobile web, whereas D3 (mobile) becomes unresponsive when rendering more than 2K marks thus its curve stops early. D3 (desktop) performs similarly with the Lottie web (mobile) as the number of lines is larger than 6K, whereas Lottie web (desktop) is the best. Moreover, the Lottie native renders 1.5 $\sim$ 2 times faster than the Lottie web.

Based on these results along with more tested results on additional platforms provided in the supplemental materials, we conclude that our compiled Lottie specifications are more scalable with smaller sizes, and can be natively rendered with better performance on multiple platforms.

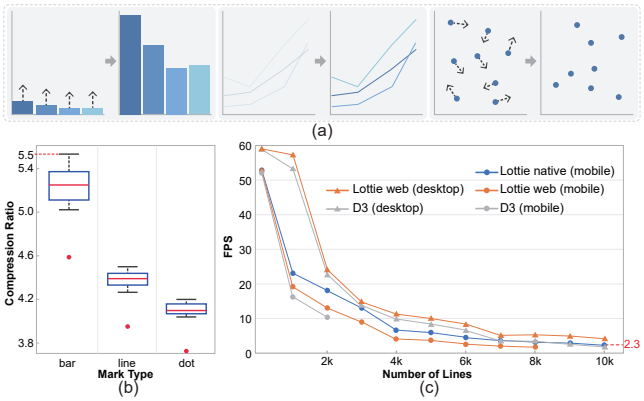

Figure 8: *(a) The two snapshots of three types of animations; (b) the boxplots showing the compression ratios for the specifications of three types of animations; and (c) the frame rate curves of the line charts animations under five rendering settings with the increasing number of lines.*

7. Discussion and Future Work

Canis requires the pre-processing step that binds data into SVG charts. By adding three properties "id" (index), "class" (SVG element type), and "datum" (associated data) for each mark, we can select all visual marks to generate data-driven animations. This means that Canis can be applied to any charts created by existing chart construction tools as long as they can be enriched with data. For the charts created with D3 [BOH11], Vega-lite [SMWH16], and Charticulator [RLB18], because they organize the graphical primitives into a scene graph, we built indices for each leaf node and added an SVG element type information into the id and class attributes while rendering the visualization.

Canis separates the animation creation step from the chart creation step, and animates marks in terms of data attributes in one or more charts. In some cases, this requires more effort for creating chart animations compared to existing libraries. Fig. 6 is such a case, where Canis requires n bar charts (one for each time point), whereas D3 and ggplot can use a transition operator to create animations, requiring only the data to be updated.

Canis is the first step in enabling declarative specifications of chart animations. It would be informative to conduct a user study to learn how easy it is to use and further improve its grammar based on the feedback. In addition, it would be useful to extend Canis with more animation effects, such as the multi-stages transitions [HR07, KCH19], and optimize the Lottie renderer to improve the performances of rendering data-driven chart animations. We also would like to extend Canis to support another types of visualizations such as node-like graphs and word clouds, where procedural animations techniques (e.g., rigid body dynamics) [Par12] might be required to simulate the movement of visual marks.

Even though the Canis specification is simpler than D3 code (as shown in Section 6), it still requires basic programming knowledge. On the other hand, Canis can be used as a building block for interactive tools for authoring chart animations. We plan to design and develop an interactive tool with Canis so that designers without programming skills can create data-driven chart animations without writing any code. Canis uses a partition operator to generate keyframes for sequencing animations. This top-down approach requires designers to clearly know how to organize all low-level marks in terms of data patterns. This means that the designers need to be able to find proper data attributes for partitioning. In contrast, the bottom-up approach used in chart construction [MHN17] starts from the mark level, which is more accessible to non-experts. We would like to explore the possibility of combining both approaches together for improving efficiency.

Acknowledgements

This work is supported in part by the grants of the National Key Research & Development Plan of China (2016YFB1001404) and NSFC (61772315, 61861136012).

References

[Adoa] ADOBE STOCK: Adobe stock. https://stock.adobe.com. [Online; accessed 6-April-2020]. 2

[Adob] ADOBEAFTEREFFECTS CC: Adobe after effects. https://www.adobe.com/products/aftereffects.html. [Online; accessed 6-April-2020]. 1, 2

[AHRL*15] AMINI F., HENRY RICHE N., LEE B., HURTER C., IRANI P.: Understanding data videos: Looking at narrative visualization through the cinematography lens. In *Proc. SIGCHI Conference on Human Factors in Computing Systems* (2015), pp. 1459–1468. 4, 8

[AHRL*16] AMINI F., HENRY RICHE N., LEE B., MONROY-HERNANDEZ A., IRANI P.: Authoring data-driven videos with dataclips. *IEEE Transactions Visualization and Computer Graphics 23*, 1 (2016), 501–510. 1, 2, 3, 9

[Aira] AIRBNB: Lottie docs. https://airbnb.io/lottie. [Online; accessed 6-April-2020]. 1, 2, 3, 4

[Airb] AIRBNB: Lottie editor. https://lottiefiles.com/editor. [Online; accessed 6-April-2020]. 4

[APP10] ARCHAMBAULT D., PURCHASE H., PINAUD B.: Animation, small multiples, and the effect of mental map preservation in dynamic graphs. *IEEE Transactions Visualization and Computer Graphics 17*, 4 (2010), 539–552. 1, 3

[BH09] BOSTOCK M., HEER J.: Protovis: A graphical toolkit for visualization. *IEEE Transactions Visualization and Computer Graphics 15*, 6 (2009), 1121–1128. 2

[BLIC19] BREHMER M., LEE B., ISENBERG P., CHOE E. K.: A comparative evaluation of animation and small multiples for trend visualization on mobile phones. *IEEE Transactions Visualization and Computer Graphics 26*, 1 (2019), 364–374. 1, 3

[BOH11] BOSTOCK M., OGIEVETSKY V., HEER J.: D^3 data-driven documents. *IEEE Transactions Visualization and Computer Graphics 17*, 12 (2011), 2301–2309. 2, 3, 4, 10

[CDF14] CHEVALIER F., DRAGICEVIC P., FRANCONERI S.: The not-so-staggering effect of staggered animated transitions on visual tracking. *IEEE Transactions Visualization and Computer Graphics 20*, 12 (2014), 2241–2250. 3, 4

[DBJ*11] DRAGICEVIC P., BEZERIANOS A., JAVED W., ELMQVIST N., FEKETE J.-D.: Temporal distortion for animated transitions. In *Proc. SIGCHI Conference on Human Factors in Computing Systems* (2011), pp. 2009–2018. 3

[EBR14] EISENBERG J. D., BELLAMY-ROYDS A.: *SVG Essentials: Producing Scalable Vector Graphics with XML*. O'Reilly Media, Inc., 2014. 4

[GBTS13] GRAMMEL L., BENNETT C., TORY M., STOREY M.-A. D.: A survey of visualization construction user interfaces. In *EuroVis (Short Papers)* (2013), Citeseer. 2

[HA14] HARPER J., AGRAWALA M.: Deconstructing and restyling d3 visualizations. In *Proceedings of the 27th annual ACM symposium on User interface software and technology* (2014), pp. 253–262. 3

[HEN] HENRI: 40 javascript ui animation libraries for web & mobile. https://bashooka.com/coding/40-javascript-ui-animation-libraries-for-web-mobile. [Online; accessed 6-April-2020]. 2

[HR07] HEER J., ROBERTSON G.: Animated transitions in statistical data graphics. *IEEE Transactions Visualization and Computer Graphics 13*, 6 (2007), 1240–1247. 3, 5, 10

[KCH19] KIM Y., CORRELL M., HEER J.: Designing animated transitions to convey aggregate operations. *Computer Graphics Forum 38*, 3 (2019), 541–551. 10

[Key] KEYSHAPE: Keyshape features. https://www.keyshapeapp.com. [Online; accessed 6-April-2020]. 4

[Kir] KIRK A.: Home - visualising data. https://www.visualisingdata.com. [Online; accessed 6-April-2020]. 1

[Kis15] KISSELL J.: *Take control of Keynote*. TidBITS Publ., 2015. 5

[Ltd] LTD. K. E.: Flourish. https://flourish.studio. [Online; accessed 6-April-2020]. 1, 2

[LTW*18] LIU Z., THOMPSON J., WILSON A., DONTCHEVA M., DELOREY J., GRIGG S., KERR B., STASKO J.: Data illustrator: Augmenting vector design tools with lazy data binding for expressive visualization authoring. In *Proc. SIGCHI Conference on Human Factors in Computing Systems* (2018), p. 123. 3

[MHN17] MÉNDEZ G. G., HINRICHS U., NACENTA M. A.: Bottom-up vs. top-down: trade-offs in efficiency, understanding, freedom and creativity with infovis tools. In *Proc. SIGCHI Conference on Human Factors in Computing Systems* (2017), ACM, pp. 841–852. 10

[Par12] PARENT R.: *Computer animation: algorithms and techniques*. Newnes, 2012. 10

[PR] PEDERSEN T. L., ROBINSON D.: A grammar of animated graphics | gganimate. https://gganimate.com. [Online; accessed 6-April-2020]. 1, 3

[RFF*08] ROBERTSON G., FERNANDEZ R., FISHER D., LEE B., STASKO J.: Effectiveness of animation in trend visualization. *IEEE Transactions Visualization and Computer Graphics 14*, 6 (2008), 1325–1332. 1, 3

[RLB18] REN D., LEE B., BREHMER M.: Charticulator: Interactive construction of bespoke chart layouts. *IEEE Transactions Visualization and Computer Graphics 25*, 1 (2018), 789–799. 9, 10

[RLBHR18] REN D., LEE B., BREHMER M., HENRY RICHE N.: Reflecting on the evaluation of visualization authoring systems: Position paper. In *2018 IEEE Evaluation and Beyond-Methodological Approaches for Visualization (BELIV)* (2018), IEEE, pp. 86–92. 7

[RLH17] REN D., LEE B., HÖLLERER T.: Stardust: Accessible and transparent gpu support for information visualization rendering. *Computer Graphics Forum 36*, 3 (2017), 179–188. 3

[RZ10] RISCHPATER R., ZUCKER D.: Introducing qt quick. In *Beginning Nokia Apps Development* (2010), Springer, pp. 139–158. 2

[SMWH16] SATYANARAYAN A., MORITZ D., WONGSUPHASAWAT K., HEER J.: Vega-lite: A grammar of interactive graphics. *IEEE Transactions Visualization and Computer Graphics 23*, 1 (2016), 341–350. 1, 2, 10

[SRHH15] SATYANARAYAN A., RUSSELL R., HOFFSWELL J., HEER J.: Reactive vega: A streaming dataflow architecture for declarative interactive visualization. *IEEE Transactions Visualization and Computer Graphics 22*, 1 (2015), 659–668. 1, 2

[TMB02] TVERSKY B., MORRISON J. B., BETRANCOURT M.: Animation: can it facilitate? *International journal of human-computer studies 57*, 4 (2002), 247–262. 3, 6

[Wic10] WICKHAM H.: A layered grammar of graphics. *Journal of Computational and Graphical Statistics 19*, 1 (2010), 3–28. 2, 3

[Wil99] WILKINSON L.: The grammar of graphics. In *Handbook of Computational Statistics*. Springer, 1999. 2

[Yau] YAU N.: Flowingdata. http://www.flowingdata.com. [Online; accessed 6-April-2020]. 1