Libra: An Interaction Model for Data Visualization

ANONYMOUS AUTHOR(S)

SUBMISSION ID: 5779*

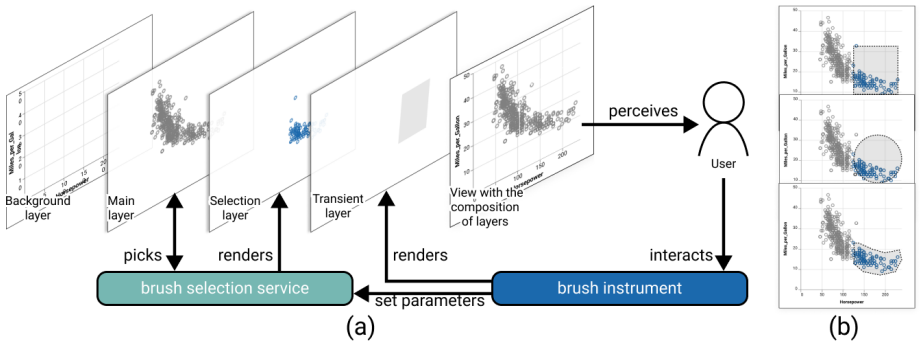

Fig. 1. An example of selecting items from a scatterplot with a selection instrument according to our Libra model. (a) Libra decomposes an interactive visualization into stacked *layers* drawn into a view. The layers are used both for drawing and managing the interaction. *Instruments* transform low-level events from the view into actions. Here, the *brushing instrument* selects items using a selection rectangle drawn in the Transient layer. The selected items are drawn in the Selection layer. The Main layer only shows the original scatterplot and allows the instrument to pick the items inside the selection rectangle. (b) With Libra, the selection instrument can be easily specialized as a rectangular (top), circular (middle), or lasso (bottom) selection instrument.

We contribute to software engineering aspects of data visualization by introducing an interaction model called *Libra* meant to facilitate the description, design, reuse, extension, and composition of interaction techniques applied to visualization. Rather than tightly coupling visual representations and interactions as the standard visualization reference model does, Libra maintains a clear separation between the visual encoding and the interaction by introducing graphical objects external to the visualization but used by the interaction (e.g., selected items and transient lasso) that are managed in multiple layers according to their semantics. Interaction management is done through *instruments* that receive events from the view and orchestrates the layers' graphical objects to trigger actions eventually. Libra wraps the traditional visualization pipeline so it remains compatible with existing visualization software architectures while providing richer control. We demonstrate the effectiveness of Libra with a D3-based implementation, which replicates, extends, and specializes multiple examples of interactive visualizations.

CCS Concepts: • **Human-centered computing → Visualization toolkits**; • **Software and its engineering → Software architectures**.

Additional Key Words and Phrases: Information visualization, interaction, separation

ACM Reference Format:

Anonymous Author(s). 2023. Libra: An Interaction Model for Data Visualization. In . ACM, New York, NY, USA, 23 pages. https://doi.org/XXXXXXX.XXXXXXX

Manuscript submitted to ACM

1 INTRODUCTION

Data visualization systems have evolved tremendously in the last decade, thanks in particular to the Grammar of Graphics (GoG) [40] and several systems inspired by it, such as ggplot2, Tableau, AntV, and Vega [1, 28, 34, 39]. They allow a large number of static visualizations to be specified concisely and expressively, reused, combined, and extended. Interaction is another fundamental part of visualization. Well-designed interaction techniques empower users to effectively explore visualized data while providing an engaging experience. There have been growing calls for enriching interactivity in visualization systems [10, 35]. Unfortunately, interaction has not evolved as well as visual representations. Interactions related to visualizations remain difficult to program and mostly impossible to extend, reuse, and compose, despite efforts to find suitable formalisms, software architecture, and libraries to do so [28, 29, 36].

While popular visualization libraries provide a few primitives for constructing interactive visualizations, most existing interaction techniques remain black boxes with limited extensibility. For example, the popular D3 library [7] provides a set of predefined common interaction techniques (e.g., pan&zoom) but relies on event-handling callbacks to implement custom interactions, leading to "callback hell" [13], difficulties for reusing interaction techniques between applications, even for extending an existing technique within an application. Although Vega-Lite [28] offers a few interaction techniques, such as pan&zoom and selection with "brushing", it uses internal black box mechanisms for implementing its standard interactions.

Vega-Lite relies internally on Vega [29, 30], which provides high-level abstractions for managing the interaction: *streams* and *signals* managed by a Reactive Functional Programming (FRP) system [29]. Still, even with this specialized mechanism, interaction techniques are provided as black boxes using hidden mechanisms, in particular for managing transient graphical objects, e.g. for interactive selection and "brushing". While FRP is a useful formalism for specifying a state machine associated with flowing data, it only handles a limited part of the interaction.

Our Libra interaction model is designed to describe explicitly all the main stages and mechanisms required to implement an interaction technique applied to visualization so that they can be reused, combined, and extended. Figure 1 presents a typical Libra configuration. A dataset is rendered in the *main layer* using a visual representation, here as a scatterplot; it can be decomposed into multiple layers. The *background layer*, under the *main layer*, displays the axes used by the visualization when appropriate. It is separate from the main layer because the two have different visual semantics, and they do not react to input events in the same way. The *selection layer*, stacked on top of the main layer, shows the *selection*—the set of selected items. A *transient layer*, on top of the selection layer, displays a selection rectangle or lasso, depending on the *brushing instrument*, used when interactively selecting the visualized items.

Libra gives an explicit status to the graphical objects that are used for the sole purpose of interaction management, such as the transient rectangle used for interactive selection, the "brush" rectangle, the lasso for lasso selection, and many more. Yet, they are never mentioned by the existing visualization architectural models and libraries, while most interaction techniques rely on them for feedback and state management. Libra relies on a set of important concepts, including *Instrument*, *Interactor*, *Layer*, and *Command*. An *instrument* acts as a mediator between the user input and a visual representation in a way similar to the tools and instruments used in the real world to interact with physical objects. In Libra, an instrument—such as the brush selection and lasso selection instruments handles the low-level events and translates them into high-level actions. An *Interactor* [25] is a state machine to help transform a low-level event stream into actions; Libra does not constrain its implementation to use a specific formalism. *Layers* are used both for feedback (showing the user the interactive state of the interactive system, e.g. selected objects or a transient selection shape) and state management (deciding how to interpret an event inside an Interactor depending on its location relative

to the interactive objects such as the selection or brush). *Commands* are used for undo/redo management. Although undo/redo is not new—it is even considered a requirement for direct-manipulation interfaces for implementing *reversible* actions [32]—it is never provided by the popular visualization libraries (except the Prefuse toolkit [19]) and is extremely hard to retrofit in a working application not designed to support it in the first place. More importantly, relying on undo/redo simplifies the implementation of *feed-forward*—showing in advance what would happen if an action was triggered—that greatly helps novices discover interactions [8]. In Libra, all the actions are performed through a set of **commands**, reifying operations performed on the data models [17, 18] and supporting undo/redo.

To summarize, this article introduces an interaction model—a set of software engineering rules, components, relations among them, and design patterns—called *Libra* to facilitate the description, design, reuse, composition, and extension of rich interaction techniques for data visualization. It maintains a clear separation of concerns between visual representation and interaction management, building on two interaction models: the instrumental [2] and the multi-layer [15] models to control the visualization reference model [9, 18]. We consider it as an extension of the Model-View-Controller (MVC) design pattern [24] applied to visualization, promoting interactions to become first-class citizens of visualization systems.

With Libra a large set of interaction techniques can be designed and retargeted within and across visual designs and input modalities while visualizations remain extensible. In summary, our contributions are:

- the Libra instrument-based layered interaction model that supports reusability, extensibility, and composability of interactions in visualizations;
- a prototype implementation of Libra in JavaScript[1] relying on D3 [7] to showcase its properties; and
- an assessment of the effectiveness of Libra using a variety of examples and the cognitive dimensions of notations.

2 RELATED WORK

2.1 Design Patterns

Model-View-Controller (MVC) is a well-known design pattern [24] fostering reuse and extensibility of a system. It separates GUI-based application components into three parts: model, view, and controller. Among them, the model manages data and application logic, the view presents the data to users, and the controller synchronizes the view when the model changes and translates the user's interactions on the view to changes in the model. To adapt this model to visualization, Heer and Agrawala [18] introduce the visualization reference model pattern. It is composed of a dataset (a specialization of the model in MVC), a view for rendering the visual model, a *visualization* for translating the dataset into a graphical form displayed on the view, and a controller for handling the user's input. The controller communicates with all the components at the user's initiative but remains abstract. The role of the controller is more detailed in the information visualization reference model of Card et al. [9], where the controller is under the user's control and can access and change the view, the visual form, the filtering, the mapping parameters, and the data. Still, this reference model remains abstract and does not explain how the controller operates. It also seems to limit the role of the controller to changing the parameters of the pipeline itself without introducing other components specific to interaction.

Libra further extends MVC for a better separation of concerns in interactive visualization applications. Libra warps the visualization reference model [9] for visually mapping data to multiple layers. By showing visual feedback resulting from interactions on specific layers (e.g., transient layer or selection layer), the main visual representation is separated from the interactions. For interaction management, we introduce the instrument as the controller, which maps low-level

[1] https://libra-js.github.io/

input events arriving at a graphical view into high-level actions for creating visual effects on layers and producing commands. In other words, there are two controllers in Libra: one for handling visual encoding and the other for managing interactions.

2.2 Interaction Implementation

Historically, interactive applications were initially programmed using event-based programming, where events were managed with callback functions, leading to "a spaghetti of callbacks" [26]. Still, several visualization libraries implement interactions through the callback mechanism. For example, Protovis [6] and D3 [7] allow for specifying custom behaviors via low-level imperative event-handling callbacks. They often require interaction programmers to maintain a state machine manually and to coordinate interleaved calls, which is complex to do well, reuse, and extend, and therefore hinders modularity, extensibility, and reuse.

To tame the complexity of event-based programming, the simplest method is encapsulating it into a black-box object so application programmers can use them without seeing its complexity. Toolkits such as Prefuse [19], Improvise [37], and VisDock [11] offer a set of predefined interaction techniques, such as selection, navigation, and annotation, for developers to integrate into their own applications. This approach has the benefit of simplicity and efficiency in reusing interaction techniques, but it is rigid and monolithic. Moreover, creating a new interaction technique with these toolkits still falls back on callback-based programming.

A popular abstraction used to simplify event-based programming relies on a state machine to specify the behavior of an interactive application when receiving events. One classical example is the *Interactor* introduced by Myers [25], a state machine that receives a sequence of events and triggers actions (higher-level callbacks) on each valid state transition. Myers explains that one interactor can be used as a base for implementing most of the standard pointer-based interactions. Libra relies on interactors to trigger higher-level actions from an event stream.

More recently, Vega [29, 30] provides higher-level mechanisms to specify the interaction. It relies on *functional reactive programming* (FRP) to offer composable interaction primitives by abstracting input events as first-class data streams that serve as the input to visualization specifications for defining the feedback of interactions. Vega-Lite [28] further introduces a high-level grammar for interaction specification by using selections as a primitive. However, FRP remains insufficient in characterizing all the aspects of interaction. For example, Vega does not have *Command* objects for encapsulating global state changes and thus cannot maintain interaction histories for supporting undo/redo actions. Second, FRP does not describe the feedback happening during an interaction; it has to be managed with specific mechanisms that are not exposed in Vega. Interactions like lasso selection (see Figure 1(b)) and Dust & Magnet (see Figure 11) require graphical feedback that is outside the scope of FRP alone. Finally, its execution happens in a black-box dataflow graph [29] not exposed by Vega, making it harder for interaction programmers to reuse and compose interaction primitives. Although the latest version of Vega allows for extracting signals, all intermediate computation results cannot be obtained for communicating with other components.

The above methods rely on a mechanism similar to a state machine, but this mechanism alone is not enough to describe the whole interaction. The instrumental interaction model [2] is a conceptual model aimed at describing the whole interaction. It is inspired by how humans use instruments to manipulate objects of interest in the physical world: an instrument is a mediator between users and objects of interest. Recently, Jansen and Dragicevic [22] adapted this model to visualization settings by unifying it with the information visualization pipeline [9] for describing, comparing, and criticizing beyond-desktop visualization systems. However, it is still a conceptual model.

Libra adapts instrumental interaction to visualization. An instrument encapsulates the states associated with low-level events in interactors and manages the feedback, feed-forward, and commands used during interactions. Instruments can be composed and reused within and across visualization designs. In doing so, it promotes a clear separation of concerns between the visual representation and the interaction, and facilitates extensibility

2.3 Layer Representation

The Grammar of Graphics (GoG) [40] allow describing visualizations in a constructive and flexible way; it has become the leading inspiration for modern visualization systems. Wickham [38] extends the GoG with a layered representation to create visualizations from multiple layers of data where each layer contains one visual representation. A visualization can be split into multiple layers, and the model of one layer can be taken as the input data of another layer, resulting in a *cascade of models*. More recent systems implementing the GoG, such as Vega-Lite [28], also use layers to superimpose multiple semantically-related representations. Yet, they do not address interaction.

The multi-layer model proposed by Fekete [15] uses layers to manage both graphical output and event management using picking primitives (i.e., locating which objects intersect a geometric shape). The InfoVis toolkit (IVTK) [14] relies on this model; it organizes an interactive visualization technique into multiple layers to adhere to the principle that each layer manages graphical objects with consistent graphical and interaction semantics. For example, IVTK implements node-link diagrams using two layers: one for nodes and one for links. However, IVTK does not provide a general interaction model and requires developers to write low-level event-handling callbacks.

Libra provides an abstract but complete description of all the stages and mechanisms required to describe and fully implement interactions in visualization. It combines the multi-layer model that can be seen as an extension of the layered GoG for interaction with the instrumental interaction model.

3 LIBRA

In this section, we first present our design goals for Libra and then describe how it manages the interaction process in a transparent way. Libra offers benefits for different roles of users; we distinguish them between the visualization toolkit/library programmer, interaction programmer, application programmer, and application user. One programmer can take several/all the programmers' roles, but the user differs from the programmers.

3.1 Design Goals

To provide comprehensive support for facilitating the description, reuse, extension, and composition of visualization interactions, we settled on the following three design goals for Libra:

DG1: Providing a complete and transparent architectural model for interaction management. Being a general-purpose interaction design, the architecture should cover all aspects of interaction management; our model exposes all its mechanisms transparently to facilitate extensibility. In terms of expressive power, it should support Shneiderman's principles of direct manipulation [32], cover three major aspects of interaction for visualization [22], and allow building interaction-rich visualizations beyond stereotypes, including undo/redo.

DG2: Separating interaction from visual representations while providing high compatibility. Several existing visualization systems can be used to create rich visual representations without interactions (e.g., the GoG [40] and ggplot2 [39]). Libra supports reusing these existing systems with simple adaptations to draw in layers and separate the main layer from the background. In doing so, Libra can maintain high compatibility with existing visualization systems by "wrapping" interactions around them.

DG3: Maximizing composition and reuse for efficient interaction specifications. Most existing visualization systems define interactions through event-driven programming, which stymies the reusability by binding objects and input methods to particular interactions. Libra strives to find the right balance between the ability to create completely new powerful interactions and reuse/compose existing components. We achieve this by clearly decomposing interaction design into a few reusable components that can be composed and extended.

3.2 Our Model

Libra relies on the information visualization reference model for constructing interactive visualizations and wraps around the components needed for interaction. It extends the MVC [24] design pattern by managing multiple models and views and two kinds of "controllers": **Visual Encoding** for transforming data models into visual forms in layers, and the instruments that translate input events from the graphical view to actions on the data models. Figure 2(a) shows the architecture of Libra, where data and visual forms are managed by the Visual Encoding and are subsequently rendered to display by the **Graphical Transformer**. Besides the visual encoding and graphical transformer, possibly built on an existing toolkit, Libra consists of four major components:

- A **Data model** is a data structure, such as a data table, that is visualized on one or multiple layers;
- A **Layer** manages a group of visual elements with consistent graphical and interaction semantics;
- An **Instrument** is a mediator between the application user and the layers; it dispatches the low-level events and translates them into higher-level actions via **Interactors** that produce feed-forward, feedback, and **Commands**. A command execution can, in turn, change one or several data models; and
- A **Service** is a functional abstraction used by the instruments to manage the data models, including data queries and analytical operations. The instruments will access and change the data models through interaction services that can help manage undo/redo and encapsulate different libraries under a common application programming interface (API).

In MVC terms, the graphical transformer and layer serve as the "View" that users perceive, the service acts as the "Model" for managing data, and the instrument is a "Controller" for translating the low-level input events to high-level actions and producing visual effects and commands. The visual encoding is also a "Controller" for transforming data

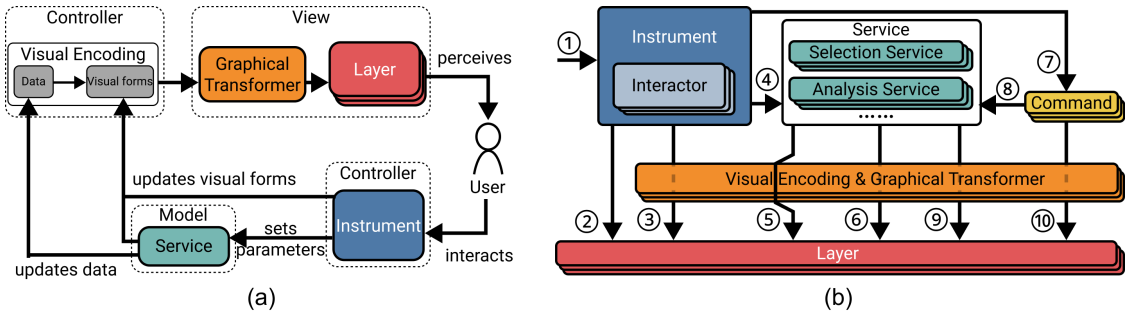

Fig. 2. The architecture of Libra (a) and the communication diagram between components (b). (a) The Libra interactive visualization architecture in terms of the MVC pattern, where the top row is the pipeline for creating static visualizations wrapped by layers for interaction, and the bottom row corresponds to the interaction management in visualizations. (b) A complete interaction process for handling a user's input event, where the service in step 5 directly picks visual marks from the layer, but the one in steps 3, 6, 9, or 10 uses the visual encoding and graphical transformer to render visual marks on the associated layer.

into visual forms and drawing them into its layer. Note that some MVC variants like Model-View-Presenter [27] also have a visual encoding to transform the model's data into a view. However, they might also translate from the view to the model, whereas our visual encoding does not react to user input since the interaction management is done by instruments.

Given a static visualization created by any visualization toolkit, a toolkit programmer needs to first decompose it into multiple layers according to different interaction behaviors. Then, the interaction can be efficiently specified by associating instruments with different layers. Since the instrument is composed of several individual components (interactor, feed-forward, commands, and feedback), interaction programmers are able to compose new instruments by assembling these components. Each instrument can be retargeted across input modalities by re-binding different input events into the interactor. In doing so, DG2 and DG3 are both satisfied.

3.3 Communication between Components

Figure 2(b) shows the communication between the Libra components when handling an input event. The process starts from the low-level input events, and then the instrument transforms them into commands and feedback in ten steps:

(1) The instrument receives a low-level event;
(2) The instrument finds the layer that picks the event and handles the event with the corresponding interactor;
(3) the instrument creates or manipulates the visual objects (feed-forward by default) through the associated with one layer which, by default, is the transient layer;
(4) The instrument invokes one or many interaction services to perform query, analyses, or layout tasks;
(5) The service picks visual elements from the corresponding layer;
(6) The instrument presents the intermediate results returned by the interaction services through the visual encoding associated with its layer;
(7) The instrument creates a command, adds it to the history manager for undo/redo, and then goes to step 8 or 10;
(8) The command persists the changes;
(9) The instrument removes the previous feed-forward (if any), and the command updates a few layers by creating new feedback based on the result of the interaction service if needed; and
(10) The instrument removes the feed-forward (if any), and the command directly produces new feedback that updates a few layers if needed.

These ten steps cover all three aspects of interaction for visualization [22]: the effect of the interaction, the goal behind producing this effect, and the means for achieving this effect in terms of feed-forward, command, feedback, and instruments, respectively. In addition, it provides a history mechanism to support reversible operations as required by the principles of direct manipulation [32]. Thus, we believe that Libra meets DG1. Not all interactions involve all the 10 steps. Some interaction techniques do not involve e.g., interaction services or data persistence, and thus, they might not have steps 4 to 10 (see Figure 5).

In comparison with FRP, when a new event fires on the graphical view, dependent signals are evaluated and propagated to update the parts of the visualization with the changed data. However, FRP does not distinguish between transient and persistent changes since all computations are done in the data domain. Namely, it does not support steps 5 and 7–10, and cannot support undo/redo operations. Meanwhile, the intermediate computation results in the other steps are unavailable to the interaction programmer.

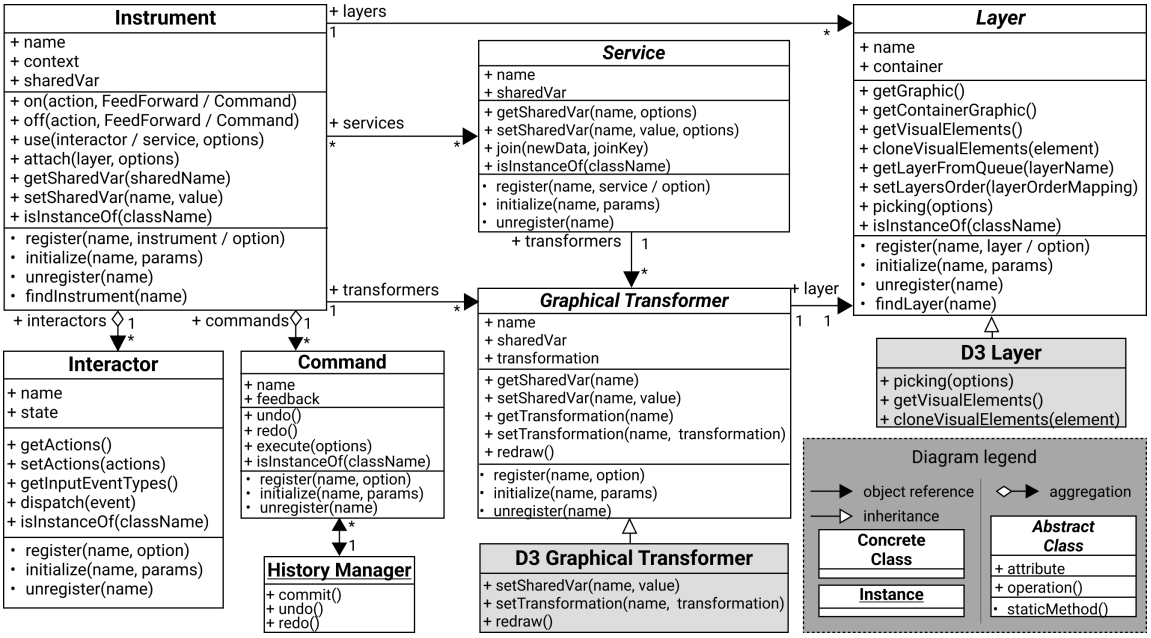

Fig. 3. The UML diagram of Libra with the diagram legend shown in the bottom right and the concrete classes in D3 with the gray background. Here we show the interfaces used in communication between classes and list some typical variables and methods in each class.

4 PROTOTYPING LIBRA IN D3

There are many ways to implement our Libra model, regardless of the language and visualization toolkit/library. We implemented a prototype in JavaScript, relying on D3 as a proof of concept and to implement the interactions described in the next section according to Libra. Figure 3 shows the UML diagram of classes and relations between them, where the instrument orchestrates everything and has access to one or more objects of all the other classes. We can see that each layer object can be accessed by multiple instrument objects with the support of multiple interaction services. Multiple instruments can be selected by the application user through a GUI, and each instrument can use and act on all the layers. Each layer is associated with a visual encoding that transforms data models into visual forms and renders them to their associated layer after applying a graphical transformer. The graphical transformer corresponds to the *view transform* in graphical applications, and to the *coordinate system* implemented by ggplot2 [38]. Since the specifications of the visual encoding and graphical transformer are often combined together in most existing visualization systems (e.g.., D3 and ggplot2), we only show the graphical transformer here. In the following, we mainly describe how we implement the other components and how to map a static D3 visualization to Libra.

4.1 Layers

Layers are passive objects with methods for drawing graphical elements and for picking them to e.g., selecting them or cloning them to the selection layer. The visual elements in each layer are created by the associated visual encoding from their model. Since D3 (as well as most toolkits) provide picking methods, we reuse them in our implementation. In general, they can be reused by applying the adapter pattern [17] to the abstract class interface.

Libra relies on the following layers for visualization applications:

- Background layer: showing the background visual elements like axes, tick marks, grids, titles, and captions;
- Main layer: showing the main visual representation;
- Selection layer: showing the selected elements, managed by the selection service; and
- Transient layer: showing transient graphical elements produced by the instruments, typically during continuous interactions.

These layers are created when the view is created, and other layers can be created as needed by the visualization application and by specific instruments. The background and main layers can also be split into multiple layers if different visual or interaction semantics are desired. Except for the background layer, the visual marks in the other layers can be changed (see section 5.2). Libra provides methods for creating new layers and finding layers by name. In multi-view visualizations, each view has its own layers. The layers are ordered and composited on top of each other in a view.

For creating the visual objects introduced by interactions, our prototype provides built-in visual encoding objects with redraw functions. For the selection layer, a service can "clone" marks from the main layer to show them in the selection layer with a highlighted appearance; this mechanism is meant to avoid interfering with the main layer's visual forms. The transient layer can create graphical objects, such as a brushing rectangle or a lasso, typically to represent the selection instrument. Interaction programmers can create their own objects in existing layers or specific ones.

4.2 Instruments

Instruments receive low-level events from input devices on a view and dispatch them to the layers and interaction services for generating appropriate visual effects and commands. As shown in Figure 3, each instrument consists of one or multiple **interactors** that are state automata for handling low-level events and transforming them into two types of higher-level actions: **feed-forward**, and **command** with their **feedback**. The **history manager** records the commands to manage undo/redo operations. In addition, instruments can share variables with interaction services and visual encodings for e.g., creating transient visual effects.

Interactors. Like Garnet [25], our interactor is based on a state machine with three states: "Start", "Running", and "Outside". The outside state refers to the input device going out of the active region. For each low-level input event, the interactor extracts event information (e.g., a mouse position) and possibly a high-level action. When an instrument uses an interactor, it binds its generic actions to instrument-specific actions. For example, when the brush instrument receives a mouse-down event with no item under the mouse position, it starts an interactor to manage a rectangle selection; its start-action will create a rectangle and save the selection, its Running-action will grow the rectangle and change the selection accordingly, and the stop-action will create a selection command with the collected selection and revert the selection to the saved state. Maintaining the selection while the rectangle is being dragged is an example of feed-forward action. It needs to be undone at the end of the interaction because (1) the interaction can be canceled (e.g., by hitting the Escape key), and (2) because the Command object needs to keep track of the selection before the interaction to be able to undo it later. After the selection is done, the command will be executed, the selection will be set "for real" as the effect of the command, producing the final feedback of the command.

Feed-forward. Feed-forward consists in showing the user what would happen if an interaction were done at this point in time. It is a glimpse of the future to inform the user before the action is done. Showing the future can be difficult, and interactive applications have an obligation to do their best to show it, but they cannot always do it accurately.

In Libra, feed-forward is implemented by a set of transient objects, orchestrated by an instrument, that is irrelevant or invisible to the underlying data models. It indicates the running status of the actions before the end of an interaction.

Feed-forward is shown in the transient layer when the interaction starts. Once the interaction is finished, all feed-forward objects are removed. Such feed-forward can be implemented using the undo facility by first applying the changes during a continuous interaction and then undoing them in the end. This implementation assumes that all the actions performed during a continuous interaction can be undone, which is not always the case. The visual effect of the feed-forward (e.g., the color of the brushing rectangle) can be customized by setting shared variables in the instrument and using them to update the corresponding visual encoding for redrawing.

Commands and Feedback. The Command object produced by an instrument changes the state of one or several data models that, in turn, produce visual feedback on the layers. For example, when a command changes the selection model, the selection layer is changed by using its visual encoding. In Figure 1(a), the command updates the selected status of the data items and the visual feedback consists in highlighting the selected items in blue. Finally, the command is stored at the end of the history manager. For atomic actions such as clicking on an item to select it, the command is directly stored in the history manager before being executed. For continuous commands like rectangle selection, each new command replaces the latest at the end of the history manager, as explained in [21]. In our prototype implementation, we use the Trrack library [12] for history management. It provides simple mechanisms to support history management and visualize the history tree.

4.3 Services

During the interaction process, there are some fundamental operations (e.g., selection) or common computations (e.g., layout) performed by multiple instruments. To improve the modularity of applications and help support undo/redo, we treat them as separated components: **services**. As shown in Figure 3, services communicate with visual encodings through shared variables and notify the visual encodings to update their corresponding layers when their execution is complete.

The services can be divided into two classes: *generic* and *specific*. The generic ones are compatible with a large number of visualizations and instruments, like the selection service, while the specific services are visualization or interaction-specific, e.g., analytics services such as clustering for visual analytics applications. If the service updates some data attributes like the clustering result, it can update the original data, e.g., when assigning class labels to data items after performing a clustering. Similar to the design of layers, these services are designed according to the adapter pattern [17], which enables the created services to be compatible with the related methods in existing toolkits. In the following, we describe three commonly used services in detail.

Selection Service. Since several interactions in visualization rely on a selection [28], Libra contains a selection service. This service maintains a list of items that are selected by the user. Assuming each item has an identifier, it associates a Boolean value to each item identifier. Due to the separation of interaction and visualization, the selection service provides a method to perform queries (similar to the SQL WHERE clause) and for updating the selection when performing composite dynamic queries. A concrete implementation of the selection service may rely on an in-memory service typical to D3 small data applications, a specialized API (e.g., [33]), or a full-fledged database service.

The selection layer uses this service to create graphical elements for all the selected items. Its visual encoding iterates over all the selected item identifiers, access their graphical elements from the main layer, and copies them on the selection layer after changing some of their visual appearance (usually their color) to look highlighted.

Several visualization applications use two selection services, one for persistent selection and one for transient selection when hovering over items for "brushing and linking" [3]. In Libra, adding hovering support boils down to adding a "hovering" layer and a second selection service managing the visual effects when hovering.

Layout Service. Most visualization toolkits provide a few layout algorithms allowing direct manipulation, such as force-directed graph layouts. Layout services allow decoupling the implementation of the actual layout and the direct manipulation to control it. A layout service computes a new layout with the interaction parameters, such as the new position of a set of items. Once a new layout is obtained, it updates the visual encoding through the shared variables. For example, a network visualization application can provide an extended selection instrument allowing to move network nodes in addition to selecting them. When the user clicks over a node, the select&drag instrument starts a click&drag interactor to move all the selected nodes and triggers the layout service to recompute the overall layout interactively at each move until the interaction ends.

Analysis Service. Visual analytics applications use data analysis algorithms such as clustering, regression, and classification. Our model can provide such an analysis service for running statistics or machine learning algorithms on the data of interest and managing the results as one of our models, as shown in Figure 12.

The underlying data model managed by these services can be visualized in a specific layer. For example, k-means clustering creates centroid nodes that can be visualized on top of the main visualization, as shown in Figure 12. A regression service can compute a regression curve that can be visualized in its own layer too.

4.4 Mapping D3 Visualizations to Libra

To map a standard D3 visualization into Libra, a few steps are needed. First, the visualization is rendered into a graphical layer that we call the *main layer*. The visualization pipeline remains almost standard, transforming a data model (e.g., a data table) into a set of marks with visual encodings. The only change is that the background part (axes, tick marks, grids) will be drawn in a separate layer. Concretely, a layer boils down to an SVG group; a view is also an SVG group that contains an ordered list of these layers/groups.

Then, Libra wraps more visual elements on top and below the static visualization using other layers, as shown in Figure 1. Libra simply adds new marks on top of the selected items in the selection layer, which are copies or simplifications (e.g., bounding rectangles) of the selected items and the corresponding data model is managed by a *selection service*. On top of the selection layer, Libra adds a *transient layer*, useful to e.g., interactively grow a rectangle when the user wants to select an area or to draw a polygon for lasso selection.

Next, each layer can be associated with one or multiple instruments, which receive the user's input events, and translate them into higher-level actions using interactor objects. For example, a selection instrument, when receiving a mouse-down event, will decide which interactor to handle this event according to the contents of the layers, from top to bottom. If the selection layer is empty and the event has occurred inside an item's mark in the main layer, then the item becomes selected and a new mark appears in the selection layer. If the event has not occurred inside any mark, the selection instrument will start a rectangle selection interaction using a click-and-drag interactor object. If the event has occurred inside a selected mark, the instrument can decide to toggle the selection, start a selection rectangle, or ignore the event. Finally, the instrument generates visible effects on the layers—such as growing a selection rectangle—and, in the end, produces a command object that will internally use the interaction services to change a model—such as selecting items. The instrument sets the `pointer-events="all"` SVG/CSS attribute on the view's group to bypass the standard SVG event dispatching mechanism and catch all the pointer-events before they are dispatched.

Figure 4 illustrates the interaction management procedure with the sequence diagram for brushing scatterplots (see Figure 1), which describes the sequential order of operations that happened. When the user starts to move the mouse (operation 1), the instrument finds the main layer to accept this event (operation 2 and operation 3) and translates the mouse-move event to the brush action via the brush interactor (operation 4). Then, the instrument invokes the interactor

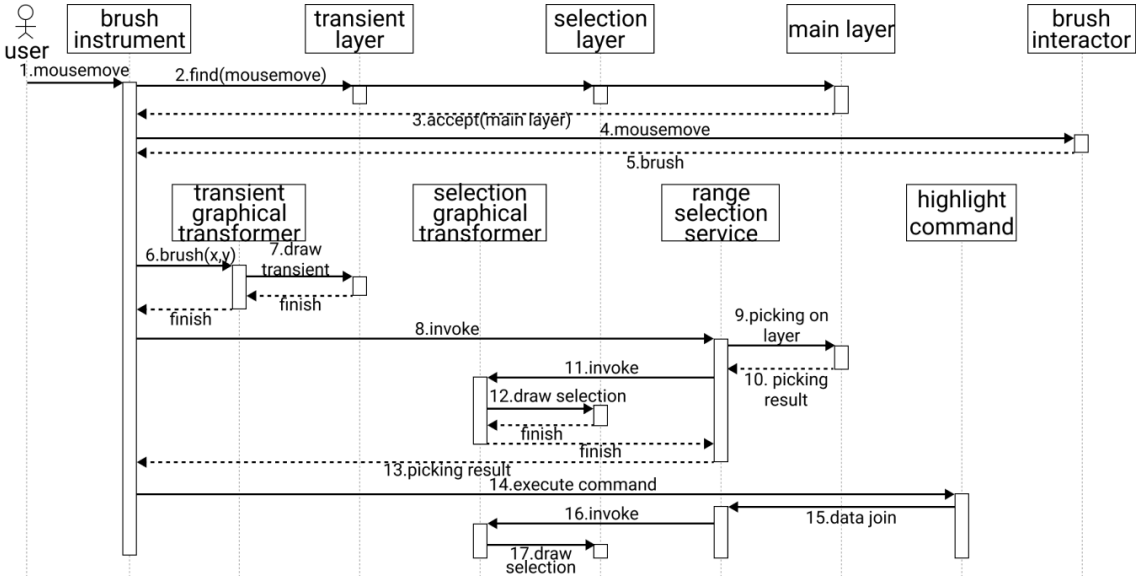

Fig. 4. The sequence diagram with the number indicates the sequence order of operations, illustrating how a brushing instrument selects and highlights the points in a scatterplot shown in Fig. 1(a).

Taxonomy	*Fig.5*	*Fig.6*	*Fig.7*	*Fig.8*	*Fig.9*	*Fig.10*	*Fig.11*	*Fig.12*	*Fig.13*	*Fig.14(a)*	*Fig.14(b)*
Select		*	*	*					*		
Explore	*		*						*	*	
Reconfigure					*	*	*	*		*	*
Encode		*		*			*	*			
Abstract/Elaborate					*				*		
Filter				*							
Connect				*				*	*		*
Undo & Redo								*			

Table 1. The taxonomy of all created examples in terms of the interaction intent.

to emit the brush action (operation 5) and shares the information with the transient visual encoding (operation 6) for drawing a rectangle displayed in the transient layer (operation 7). Next, the instrument invokes the selection service (operation 8) to pick elements from the main layer (operation 9) and get results (operation 10). Next, it invokes its visual encoding (operation 11), draw them into the selection layer (operation 12) and returns them back to the instrument (operation 13). Once the brushing region is determined, the instrument executes the command (operation 14) to persist the data changes (operation 15) and (re)draw them into the selection layer as feedback (operation 16 and operation 17). Meanwhile, the history manager records the changed data points into a Command object and the rectangle in the transient layer is removed.

5 EVALUATION

Libra aligns with Alan Kay's quote: "Simple things should be simple, complex things should be possible". To demonstrate the expressiveness of Libra, we use its D3-based prototype to create examples of visualizations and interactions from the simplest to the most complex.

In terms of the taxonomy of interaction intent [42], most created examples involve multiple intents (see Table 1). For example, the intents of interactive k-means shown in Figure 12 can be considered as the combination of reconfigure, encode, connect, and other intents, as the position of cluster centroids can be moved, each point can be recolored, and each view is connected to each other. Moreover, it provides an additional general intent of undo and redo operations.

In our D3-Libra prototypes, instruments are implemented using sub-classing and composition, allowing the creation of abstract instruments from abstract base classes. Here, we introduce a new taxonomy of instruments: non-intrusive and intrusive instruments, depending on whether the data-encoded visual marks in the input visualization are changed during the interaction. Accordingly, we categorize these examples into three classes: non-intrusive, intrusive, and semi-intrusive instruments, the later combining intrusive and non-intrusive parts.

5.1 Non-Intrusive Instruments

An instrument is non-intrusive when the visual effects of its interactions are achieved without the main visualization being aware of it. In doing so, a clear separation of concerns between the visual representation in the main layer and the other layers (e.g., the transient layer or selection layer) is achieved for the instrument. Here, we describe four non-intrusive instruments: annotating, brushing, excentric Labeling, and cross-filtering.

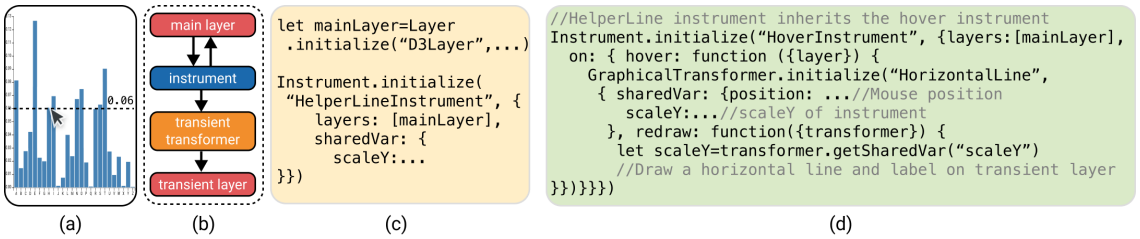

Fig. 5. Comparing multiple bars in a bar chart with a horizontal helper line in (a) is achieved by four components (b); (c) the Libra specification from the interaction programmer, and (d) the built-in specification for the helper line instrument with an overridden visual encoding.

Annotating with a Helper Line. Figure 5 shows an example of an annotating helper line for accurate comparison between multiple bars in a bar chart. When the mouse hovers over the bar chart view, a dotted horizontal line with the label showing the data value on the y-axis is created (Figure 5(a)).

Since this annotating instrument does not require any interaction service and command, we treat the line as the feed-forward in the transient layer, which is totally independent of the bar chart. The communication between the four components used for implementing this instrument is shown in Figure 5(b). To implement this instrument, the interaction programmer only needs to define a main layer with the whole bar chart, and then associate a built-in helper line instrument with the main layer (Figure 5(c)). This instrument is inherited from the hover abstract instrument; it overrides the corresponding visual encoding (Figure 5(d)), which defines a new redraw function for drawing a horizontal line with the label value obtained from the shared scale information. If the label information is not required, the line can be directly drawn by using the pointer position returned by the interactor without using the shared scale information. This instrument can be enabled by default, or only when a modifier is hit (e.g., the Shift key), and can be added on top of any visual representation, but it only makes sense for visualizations that use a meaningful y-axis such as bar charts, line charts, histograms, and scatterplots.

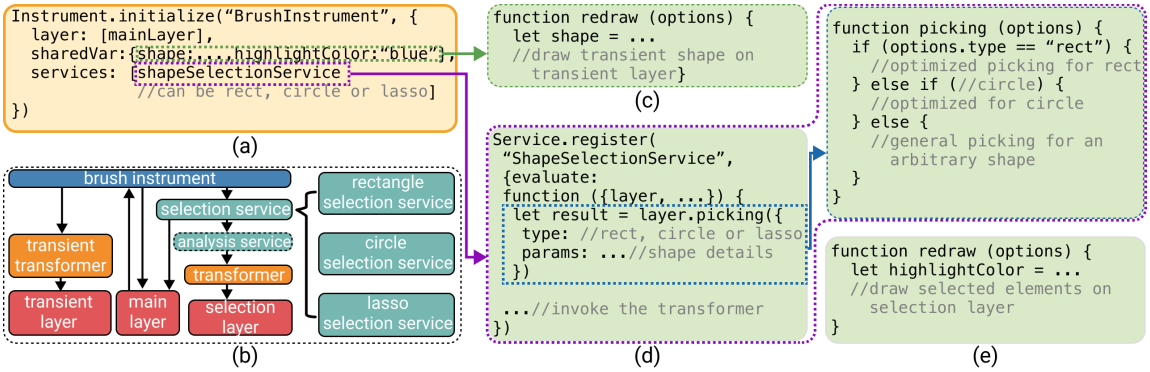

Fig. 6. Brushing a scatterplot. (a) Libra specification from the interaction programmer; (b) the communication between Libra objects (including an optional analysis service) used for achieving this instrument; (c,d,e) the built-in code for generating the side-effects in three steps: creating the feed-forward (d), running the selection service for picking the selected elements (d), and creating the feedback (e).

Brushing a Scatterplot. Selection is a fundamental operation in interactive visualizations; it allows users to mark items of interest. It typically takes the form of point, rectangle, and lasso selection, with more sophisticated extensions for e.g., line charts. Figure 1 shows a brushing example that performs selection directly in the visualization. This instrument can be concisely specified by defining a main layer composed of all the points and then associating it with an abstract BrushInstrument (see Figure 6(a)). After an instrument is initialized, it automatically registers the selection service to the main layer and generates the indispensable objects and the relationship between them (see Figure 6(b)). When the brushing action is triggered, the instrument generates the visual effects in three major steps. First, it creates the selection shape (rectangle, circle, or lasso) on the transient layer as a feed-forward effect with the built-in code shown in Figure 6(c). Then, its selection service invokes the picking function in the main layer to select the data items within the given shape through the selection service. The code in Figure 6(d) shows different picking functions for different selection shapes. Last, it highlights the selected items by creating "ghost" items in the selection layer (see Figure 6(e)).

If the user wants to employ other shapes, such as an ellipse, to select items, the interaction programmer needs to redefine the redraw function in the transient visual encodings (see Figure 6(c)) and the picking function used in the main layer (see Figure 6(d)). These brush instruments can be reused on any visualization thanks to the generic picking function of the main layer.

In addition, the results of the selection service can be taken as the input data (model) of another service (see the dotted box in Figure 6(b)). In other words, such a cascade of models facilitates a sequence of data analysis tasks. For example, an interactive regression lens [31] can be easily implemented by connecting the selection service to a line regression service and displaying the regression line in the selection layer or a specific regression layer.

Excentric Labeling. The results from the selection service can be taken as the input of the other services. For example, the excentric labeling technique [4, 16] was originally designed for interactively labeling of visual items in the vicinity of the mouse cursor in scatterplots. The bottom in Figure 7(a) shows the composition of such an excentric labeling instrument for achieving this interaction.

Based on a hover instrument, this instrument is composed of a shape selection service, a layout service, and an analysis service (the top in Figure 7(a)), where each service has a visual encoding for redrawing the corresponding layer.

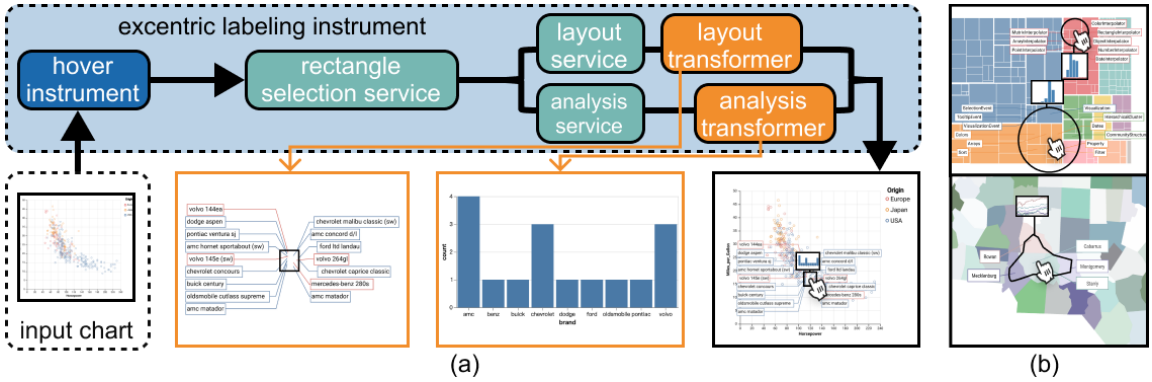

Fig. 7. Implementing an excentric labeling instrument for various visualizations. (a) (top) The components used for composing an excentric labeling instrument to support this interaction for scatterplots, and (bottom) the input chart, the results from the layout service and analysis service and the final result; (b) reusing this instrument for treemaps (top) and maps (bottom).

At runtime, the hover action first triggers the rectangle selection service to query the data items within the selection extents on the main layer and then invokes the visual encoding to draw them on the selection service layer. Meanwhile, the selection service shares the selected items with the other services for computing the excentric layout and the statistics for the "Extended Excentric Labels" [4] through the layout service and the analysis service (Figure 7(a)). Once the computation is done, the corresponding visual encoding updates the layer.

Since data selection, label placement, and statistical analysis are maintained by separate services, the excentric labeling technique can be extended in various ways. For example, the selection instrument can use different shapes such as a rectangle, circle, and lasso. Different statistical charts can be shown, such as bar charts and line charts. Since all these services and layers are separated from the visual representation in the main layer, this excentric labeling technique can be reused for any kind of visualization when it makes sense: a treemap (see the top in Figure 7(b)), or a map (see the bottom in Figure 7(c)) that can also exhibit dense areas.

Cross-Filtering. Rather than managing the data model of one view, the selection service can also be used to manage the selection results in multiple linked views. Figure 8(a) shows an example of cross-filtering for exploring multidimensional data [33], where a scatterplot view and four histogram views are linked. Figure 8(b) shows the Libra specification that

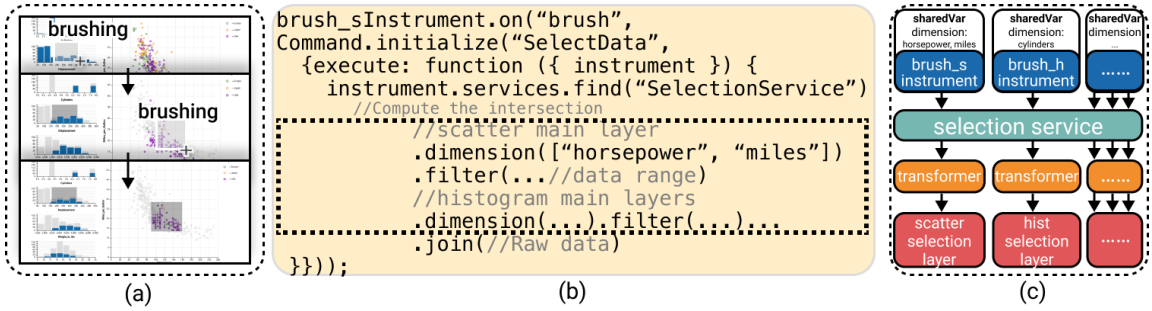

Fig. 8. Implementing a cross-filtering instrument for the coordinated scatterplot and four histograms. (a) Three snapshots of the interaction; (b) attaching the scatterplot layer to the brushing instrument and a command for computing the intersection of selections, and (c) the involved objects and communication between them.

uses the same API as a popular example application [33]. Taking each view as an individual main layer, we use the same selection service to manage the selection results in different views to achieve cross-filtering; see the communication between the components in Figure 8(c). When the user brushes on one view, all the views are updated by combining all the selection results through the intersection operators. By sharing the selection service, each view will automatically be updated with the latest selection result.

5.2 Intrusive Instruments

An intrusive instrument directly modifies the visual elements of the input visualization, which might be split into one or multiple main layers in our model. Namely, it has at least one visual encoding that directly updates a main layer. Yet, the interaction can remain independent from the visual encoding. Here, we describe five intrusive instrument techniques: panning &zooming, interactive lens, dust & magnet, interactive k-means and DimpVis [23], where the last two focus on the exploration of multidimensional data.

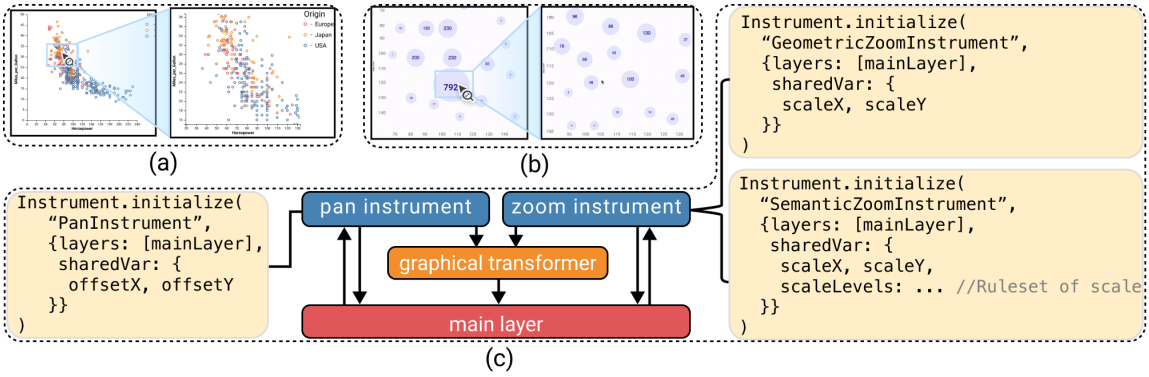

Fig. 9. Two forms of panning & zooming a scatterplot: geometric zooming (a) and semantic zooming (b). (c) The Libra specifications for for defining the panning instrument (left), the geometric zooming instrument (top in the right) and semantic zooming instrument (bottom in the right), and the objects used for managing these instruments (middle).

Panning & Zooming. Figure 9 shows two pan & zoom instruments for a scatterplot with two forms of zooming: geometric zooming (Figure 9(a)) and semantic zooming (Figure 9(b)). The Libra specification for the pan instrument is shown on the left of Figure 9(c); it is reused for two different visualizations in Figure 9(a,b). The right in Figure 9(c) shows the Libra specification for geometric and semantic zooming instruments, which shares the scale information with the visual encoding of the main layer. All the major objects used in both examples and the relationships between them are shown in Figure 9(c), where the pan & zoom instruments are attached to the main layer and drive the visual encoding to redraw the main layer with the latest scale and offset information. We can see that there is no selection service involved.

Compared to geometric zooming, the specification of semantic zooming (see the bottom in the right of Figure 9(c)) involves a mapping between the scale information and the semantic level. Since different datasets have different semantic zoom levels, we let the interaction programmer define the mapping rules and share them with the visual encoding for updating the main layer. Note that we assume the input data is already organized in multiple scales.

EdgeLens. Figure 10(a,b) shows an implementation of the EdgeLens technique [41]: a node-link diagram distorted by a fisheye around the pointer position. Moving the nodes or edges can help mitigate visual clutter. The EdgeLens

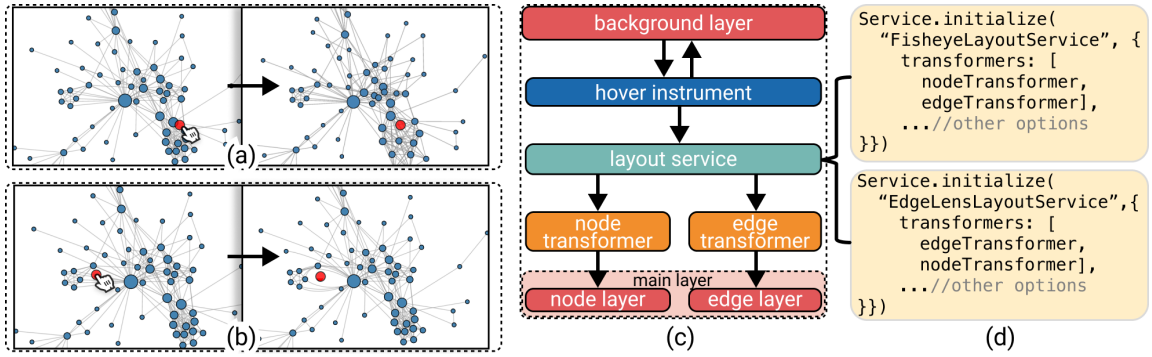

Fig. 10. Implementing the fisheye lens and the edge lens for a node-link diagram with Libra. (a,b) The examples of a fisheye lens and edge lens; (c) the Libra components used for lensing and the relationship between components; (d) the Libra specification for the layout services.

instrument can be defined by composing a hover instrument, a layout service, the node and edge visual encoding, and the node and edge layers together. To implement an EdgeLens instrument, the interaction programmer first splits the input node-link diagram into the node layer and edge layer and attaches a hover instrument to the background layer. A layout service manages the layout of the nodes and edges. Finally, this service is connected with the node and edge visual encoding for re-drawing them. The relationship between these objects are shown in Figure 10(c) and the Libra specification for the layout services used in Figure 10(a,b) is shown in Figure 10(d). By customizing the layout service, different lens effects can be easily achieved in Libra.

Dust & Magnet. Figure 11 shows the "Dust & Magnet" instrument [43] composed of a magnet layer showing the attributes as colored rectangles that can be dragged, and a dust layer showing all the data items as points that can be hovered. As shown in Figure 11(a), clicking a point on the background layer creates a new magnet and recomputes

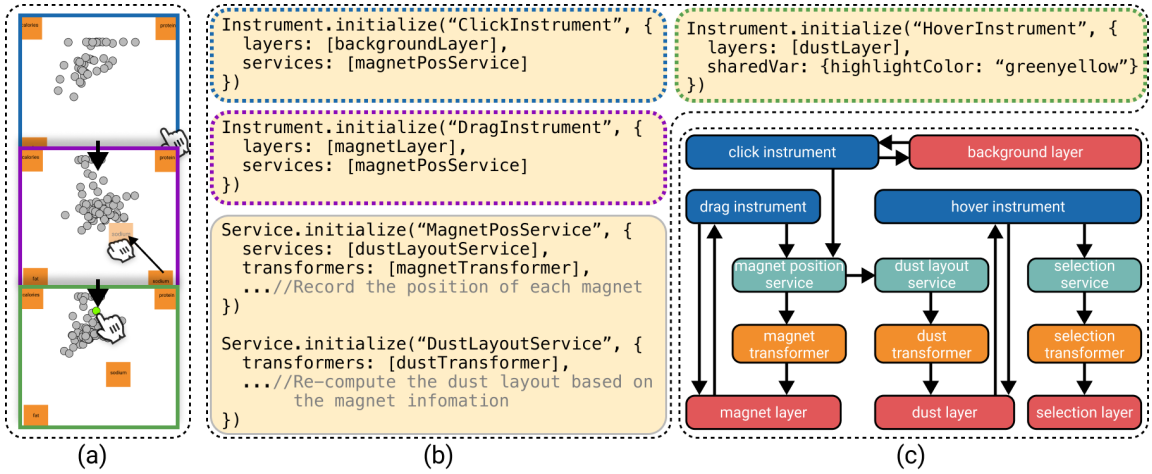

Fig. 11. Using the dust & magnet instrument for exploring multidimensional data. (a) The snapshots of three actions: creating a magnet, moving a magnet, and hovering a dust point; (b) the Libra specification for defining the instrument and services; (c) all the involved objects and the relationship between them.

the layout of all the points on the dust layer; dragging a magnet also triggers a recomputation of the point positions; and hovering a point in the main layer highlights it in green. In other words, this instrument is composed of three instruments (a click instrument, a drag instrument and a hover instrument) and two connected services for managing magnet and dust positions.

To implement the instrument, the interaction programmer first attaches a click instrument, a drag instrument and a hover instrument to the background, magnet, and dust layers, respectively. Then, she associates the click and drag instruments with the magnet position service and connects the magnet position service to the dust layout service. The Libra specification in Figure 11(b) corresponds to these three snapshots shown in Figure 11(a). Figure 11(c) shows the major objects used in the interaction, where a magnet position service is associated with a magnet visual encoding for redrawing the magnet layer, and the other two services are associated with a dust visual encoding for redrawing the dust layer. Creating and dragging a magnet first trigger a data service for adding or changing a magnet (one data attribute) and share the latest magnet position to the layout service for re-arranging points. Hovering a dust point only triggers the selection service on the dust layer.

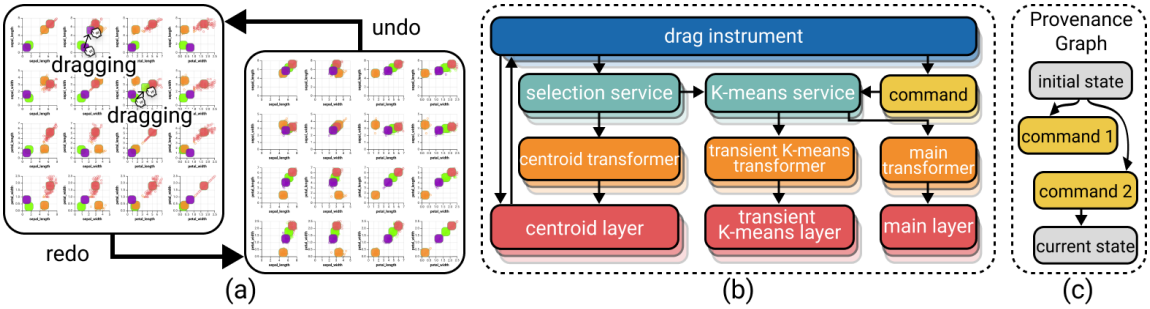

Fig. 12. Interactive *k*-means clustering of multidimensional data shown on a scatterplot matrix. (a) After dragging the blue and green centers (left), the new clustering result is generated (right); (b) The communications between all objects; and (c) the provenance graph for recording commands.

Interactive K-Means. Figure 12(a) shows an example of an interactive k-means clustering instrument applied to a multidimensional dataset visualized as a scatterplot matrix. Here, an application user is allowed to move a centroid if she believes that it is misplaced. This triggers new iterations of the k-means algorithm starting from the specified configuration, leading to a possible better solution. With a few interactions, the application user can check if the clustering is stable or fix it. For example, the green and purple centroids are moved by the user on the left of Figure 12(a) and then a new clustering result is quickly obtained, as shown in the right of Figure 12(a).

To implement the interactive k-means instrument, the interaction programmer needs to define the main layer and two additional layers for each view: a centroid layer and a transient k-means layer. Then, she associates the centroid layer with the drag instrument, which uses the selection service to update centroids and shares the results with the k-means service for generating new clustering results. During the k-means iteration process, the intermediate clustering results are shown in the k-means layer. Figure 12(b) shows the communication between these objects, where the drag action triggers the centroid selection and then the k-means service for updating the clustering result. To persist the clustering result, the command is executed to update the centroid data while re-drawing the main layer after the clustering is computed. The provenance graph shown in Figure 12(c) records the interaction process where the user undoes command 1 and executes a new command based on command 2.

We can see that Libra can easily support complex interactions through the composition of instruments, layers and services. Since the k-means service is independent of the main visualization, this instrument can be extended with many other machine learning algorithms for supporting interactive visual analysis.

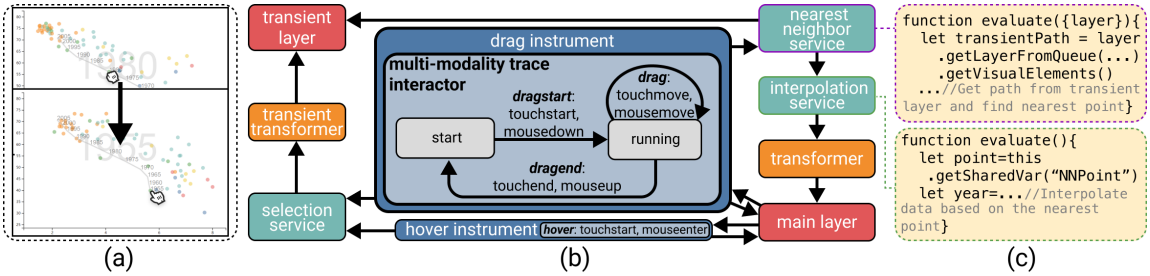

Fig. 13. Implementing the DimpVis interaction technique. (a) Two snapshots of this interaction; (b) the communication between all objects; (c)Libra specifications for the nearest neighbor service and the interpolation service.

Dimpvis for Temporal Navigation. DimpVis [23] is a direct manipulation technique for exploring time-series data shown in a scatterplot at one time-slice. Hovering or touching any data points in the scatterplot reveals a timeline showing the evolution of the selected item through the years. Dragging the selected item through the timeline enables temporal navigation, where the scatterplot is updated to the data at the newly selected year. To improve the robustness of interaction, the dragged position is automatically placed to the nearest position in the timeline when it deviates from the timeline. Figure 13(a) shows two snapshots during the interaction.

To implement this instrument, the interaction programmer first splits the scatterplot showing the dataset into two layers: the main layer consists of data points and the year information, and the background layer consists of the axis and legend. Then, she constructs the hover instrument and the drag instrument by binding the touch event and three touch events (touchstart, touchmove, and touchend) to the corresponding actions and attaches them to the main layer. To display the timeline, she connects the hover instrument to the selection service and re-define the drawing function in the transient visual encoding as the feed-forward. Next, she connects the drag instrument to the nearest neighbor service to find the nearest point of the drag point in the timeline shown in the transient layer. Finally, she shares this nearest point to the interpolation service by connecting it with the nearest neighbor service and associating it with the main layer for updating the visualization. Figure 13(b) depicts the relationship between these objects, among which the Libra specification for the nearest neighbor service and the interpolation service are shown in Figure 13(c).

5.3 Semi-Intrusive Instruments

Semi-intrusive instruments often have more than one service for selecting and manipulating data items. Figure 11 is an example where the hover instrument is non-intrusive, but the click and drag instruments are intrusive. Here, we show two more examples: an index chart and an overview+detail visualization.

An Index Chart. For a line chart with multiple time-series, an index chart allows users to hover over a point to obtain the index point indicated by a vertical line and then re-normalize time-series to show percentage change based on the index point. The left in Figure 14(a) illustrates this process with two snapshots. By using Libra, such interactions can be created by extending the example of annotating with a helper line. The interaction programmer first defines a normalization service and connects it to the helper line instrument for getting the current index. Then, she associates

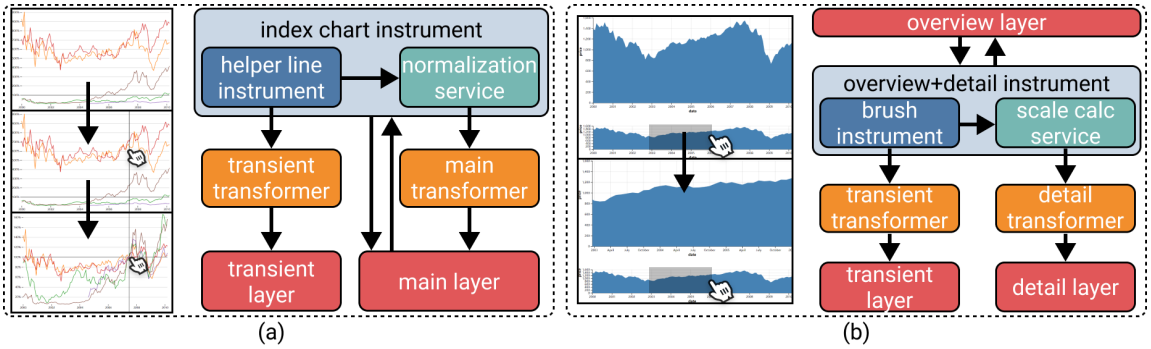

Fig. 14. Implementing two semi-intrusive instruments: an index chart (a) and an overview+detail visualization (b). In (a,b) the left is the snapshots of the visualization and the right is the communication between all objects.

this service with the main layer for re-drawing the time-series. The right in Figure 14(a) shows how these Libra objects communicate with each other.

An Overview+Detail Visualization. The examples in Figure 9 have demonstrated how the input static visualization can be extended to enable zooming in and out. Here, we show that they can be extended to offer support for an overview+detail instrument, where the range selected by a brush selection in one view is shown in another view at a higher resolution (see the snapshots on the left of Figure 14(b)). To implement this instrument, the interaction programmer first reuses the brush instrument used in Figure 6 for the overview display but restricts the brushing in the horizontal dimension. Then she connects this instrument to a scale calculator service for performing scale inversion for the selected range and extracting the detailed data. Finally, she connects this service to the main layer for updating the detail view. The right in Figure 14(b) shows the relationship between these objects.

6 DISCUSSION AND FUTURE WORK

The examples in section 5 systematically demonstrate Libra's capabilities in describing, reusing, composing, and extending interactions. Here, we conduct a usability evaluation of our model based on its expressiveness and on the cognitive dimensions of notation framework [5].

6.1 Expressiveness of Libra vs. existing systems.

Our prototype can be taken as a plugin for D3. Hence, both have the same expressiveness in terms of visual representation, but Libra facilitates the description, reuse, composition, and extension of interactions because of the separation of concerns between the visual representation and the interaction. Even without taking into account the reuse and composition, currently, only a part of our provided examples can be implemented with Vega and Vega-lite for two reasons. First, when modeling input events as streaming signals, it is hard to support interactions involving variable-length signals like lasso selection. Second, Vega provides a limited set of data transforms, which are encapsulated as nodes in the dataflow graph. Although users are allowed to customize transforms, they are required to understand how the nodes are dependent on each other in the internal dataflow graph. As a result, it is hard to implement e.g., the interactive k-means shown in Figure 12. In contrast, Libra services separate well the multiple components involved in complex interactions to facilitate implementations relying on rich analytical operations. Vega-lite shares similar issues with Vega and thus, Libra is more expressive regarding interaction.

In the future, we will extend Libra to manage multiple types of devices by supporting new event types at the instrument level and designing new types of interactors, as shown in the example of DimpVis (see Figure 13). We also plan to extend Libra for synchronous collaboration, providing feedback on other users' cursors and viewports in an extra layer in a way similar to CocoNutTrix [20].

6.2 Cognitive Dimensions of Notation

The cognitive dimensions of notation is a heuristic evaluation framework with 14 dimensions where a relevant subset has previously been used to evaluate Protovis [6] and Vega [30]. Here, we evaluate our model with a relevant subset and draw comparisons to D3 and Vega where appropriate.

Abstraction (types and availability of abstraction mechanism). Libra does introduce new abstractions by creating a new type of object: the instrument, as a first-class citizen in visualization systems. It also introduces abstractions tied to the instrument, and commands that are already well-known in HCI. This approach can facilitate rapid iteration of interaction design by reusing, composing and extending existing instruments. As our examples demonstrate, each component (interactor, feed-forward, feedback, and commands) of an instrument can be customized as desired. By comparison, iterating with imperative event handling callbacks requires interaction programmers to edit callback functions for implementing an instrument without a reference model, resulting in hard mental operations and error-proneness.

Closeness of mapping (closeness of representation to domain) and *Viscosity* (resistance to change). In Libra, the instrument concept is closely linked to interaction: instruments are inspired by tools in our daily life, acting as a mediator between the application user and the object being manipulated. Furthermore, Libra relies on existing toolkits and reuses their specification of visual encodings, leveraging existing knowledge. In contrast, Vega requires interaction programmers to learn the streaming and signal mechanism that couples the specification of visual encodings and visual feedback, yet, it lacks guidance on how to specify each aspect of the interaction in the whole interaction process.

Premature Commitment (constraints on the order of doing things). To enable instruments, Libra requires users to construct layers by putting visual objects with the same interaction semantics together and then associate them with related instruments. For most application programmers using Libra, an application will only require assembling visual representations and instruments, contrary to existing systems that either require implementing the desired interactions from scratch or providing a fixed set of monolithic interaction techniques. For the interaction programmer, Libra provides much more guidance than existing toolkits: first work on the visual representation alone and then assemble the desired instrument by first composing existing instruments and then specializing the main instrument and possibly others. We think premature commitment will happen less often and have less impact with Libra than using existing toolkits, in particular for advanced interactions. In contrast, Vega requires users to create streams and signals before they can use any low-level events to trigger interactions.

Hidden Dependencies (important links between entities are not visible) and Visibility (ability to view components easily). Libra offers a complete and transparent interaction model, where every component can be customized. Although there are some dependencies between different types of objects (e.g., layers and instruments), they are all known to interaction programmers. As our code examples illustrate, with our model, all the aspects of the interaction can be extracted, reused, and customized. In contrast, the abstraction of stream and signals introduce heavy hidden dependencies in Vega and the dependencies between nodes in the black-box dataflow graph are completely hidden, which hinders interaction programmers from customizing interactions, especially commands and feed-forward. With D3, interaction programmers must manage all the interactions through event callbacks.

Consistency (similar semantics are expressed in similar syntactic forms). We believe our interaction model can be implemented on top of any existing visualization toolkit, and thus, our current D3 prototype remains consistent with the D3 visual encodings. Being able to build instruments and interactions by sub-classing and composing existing instruments ensures a higher degree of consistency among the instruments than re-implementing every new interaction technique in isolation. It might as well lead to adopting a specific family of interaction techniques to remain consistent with existing instruments. For example, use drag&drop often, or often rely on external widgets. Even if that happens, we believe consistency within a specific style of interaction is better than inconsistent interactions. With Libra, it is always possible to design a completely new hierarchy of instruments implementing consistently another interaction style (e.g., voice-based or gesture-based).

Just like new versions of D3 provide new visual channels, support for new file formats, and new options for existing visual representations, using Libra would allow new interactions to be non-intrusively or semi-intrusively integrated into existing visualizations. Although there is a learning curve for programmers to understand our model, we believe the extra complexity is justified by the extended capabilities offered: interactions can become first-class citizens of visualization systems.

7 CONCLUSION

We introduce the Libra interaction model to enable the description, design, reuse, composition, and extension of rich interaction techniques for data visualization. We demonstrate the expressiveness and flexibility of Libra in describing several interactions for visualizations. For simple interactions, our model is concise by associating built-in instruments with any visual representation. Additionally, it is expressive enough to cover most of the existing advanced interactions. Thanks to a clear separation of concerns, the visual representation can be decoupled from the interaction management and both can evolve independently, at least for the large category of non-intrusive instruments. As demonstrated by the examples, our D3 prototype only requires programmers to split a given static D3 visualization into layers to benefit from the model. All the examples available in the accompanying website libra-js.github.io, demonstrate Libra's expressiveness. With Libra, interactions become first-class citizens of visualization systems.

REFERENCES

[1] AntV Visualization. https://antv.vision/en, 2022. Accessed: 2022-07-20.

[2] M. Beaudouin-Lafon. Instrumental interaction: an interaction model for designing post-WIMP user interfaces. In *Proc. SIGCHI Conf. Human Factors Comp. Sys.*, pp. 446–453, 2000. doi: 10.1145/332040.332473

[3] R. A. Becker and W. S. Cleveland. Brushing Scatterplots. *Technometrics*, 29(2):127–142, 1987. doi: 10.1080/00401706.1987.10488204

[4] E. Bertini, M. Rigamonti, and D. Lalanne. Extended Excentric Labeling. *Computer Graphics Forum*, 28(3):927–934, 2009. doi: 10.1111/j.1467-8659.2009.01456.x

[5] A. F. Blackwell, C. Britton, A. Cox, T. R. Green, C. Gurr, G. Kadoda, M. S. Kutar, M. Loomes, C. L. Nehaniv, M. Petre, et al. Cognitive dimensions of notations: Design tools for cognitive technology. In *International conference on cognitive technology*, pp. 325–341. Springer, 2001.

[6] M. Bostock and J. Heer. Protovis: A graphical toolkit for visualization. *IEEE Trans. Vis. Comput. Graphics*, 15(6):1121–1128, 2009. doi: 10.1109/tvcg.2009.174

[7] M. Bostock, V. Ogievetsky, and J. Heer. D^3 Data-Driven Documents. *IEEE Trans. Vis. Comput. Graphics*, 17(12):2301–2309, Dec. 2011. doi: 10.1109/TVCG.2011.185

[8] J. Boy, L. Eveillard, F. Détienne, and J. Fekete. Suggested interactivity: Seeking perceived affordances for information visualization. *IEEE Trans. Vis. Comput. Graphics*, 22(1):639–648, 2016. doi: 10.1109/TVCG.2015.2467201

[9] S. K. Card, J. D. Mackinlay, and B. Shneiderman. *Readings in Information Visualization: Using Vision to Think*. Morgan Kaufmann, 1999.

[10] H. Childs, B. Geveci, W. Schroeder, J. Meredith, K. Moreland, C. Sewell, T. Kuhlen, and E. W. Bethel. Research challenges for visualization software. *Computer*, 46(5):34–42, 2013. doi: 10.1109/mc.2013.179

[11] J. Choi, D. G. Park, Y. L. Wong, E. Fisher, and N. Elmqvist. Visdock: A toolkit for cross-cutting interactions in visualization. *IEEE Trans. Vis. Comput. Graphics*, 21(9):1087–1100, 2015. doi: 10.1109/tvcg.2015.2414454

[12] Z. Cutler, K. Gadhave, and A. Lex. Trrack: A Library for Provenance-Tracking in Web-Based Visualizations. In *31st IEEE Visualization Conference, IEEE VIS 2020 - Short Papers, Virtual Event, USA, October 25-30, 2020*, pp. 116–120. IEEE, 2020. doi: 10.1109/VIS47514.2020.00030

[13] J. Edwards. Coherent reaction. In *Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications*, pp. 925–932, 2009. doi: 10.1145/1639950.1640058

[14] J.-D. Fekete. The InfoVis toolkit. In *IEEE Symposium on Information Visualization*, pp. 167–174, 2004. doi: 10.1109/INFVIS.2004.64

[15] J.-D. Fekete and M. Beaudouin-Lafon. Using the multi-layer model for building interactive graphical applications. In *Proceedings of the 9th Annual ACM Symposium on User Interface Software and Technology*, pp. 109–118, 1996. doi: 10.1145/237091.237108

[16] J.-D. Fekete and C. Plaisant. Excentric Labeling: Dynamic Neighborhood Labeling for Data Visualization. In *Proc. SIGCHI Conf. Human Factors Comp. Sys.*, pp. 512–519, 1999. doi: 10.1145/302979.303148

[17] E. Gamma, R. Helm, R. E. Johnson, and J. Vlissides. *Design Patterns: Elements of reusable object-oriented software*. Addison-Wesley Reading, Massachusetts, 1995.

[18] J. Heer and M. Agrawala. Software Design Patterns for Information Visualization. *IEEE Trans. Vis. Comput. Graphics*, 12(5):853–860, 2006. doi: 10.1109/tvcg.2006.178

[19] J. Heer, S. K. Card, and J. A. Landay. Prefuse: a toolkit for interactive information visualization. In *Proc. SIGCHI Conf. Human Factors Comp. Sys.*, pp. 421–430, 2005. doi: 10.1145/1054972.1055031

[20] P. Isenberg, A. Bezerianos, N. Henry, S. Carpendale, and J. Fekete. Coconuttrix: Collaborative retrofitting for information visualization. *IEEE Comput. Graph. Appl. Mag*, 29(5):44–57, 2009. doi: 10.1109/MCG.2009.78

[21] T. Jankun-Kelly, K.-l. Ma, and M. Gertz. A Model and Framework for Visualization Exploration. *IEEE Trans. Vis. Comput. Graphics*, 13(2):357–369, 2007. doi: 10.1109/TVCG.2007.28

[22] Y. Jansen and P. Dragicevic. An interaction model for visualizations beyond the desktop. *IEEE Trans. Vis. Comput. Graphics*, 19(12):2396–2405, 2013. doi: 10.1109/tvcg.2013.134

[23] B. Kondo and C. Collins. Dimpvis: Exploring time-varying information visualizations by direct manipulation. *IEEE Trans. Vis. Comput. Graphics*, 20(12):2003–2012, 2014. doi: 10.1109/TVCG.2014.2346250

[24] G. E. Krasner, S. T. Pope, et al. A description of the model-view-controller user interface paradigm in the smalltalk-80 system. *Journal of Object Oriented Programming*, 1(3):26–49, 1988.

[25] B. A. Myers. A new model for handling input. *ACM Transactions on Information Systems (TOIS)*, 8(3):289–320, 1990. doi: 10.1145/98188.98204

[26] B. A. Myers. Separating application code from toolkits: eliminating the spaghetti of call-backs. In J. R. Rhyne, ed., *Proceedings of the 4th Annual ACM Symposium on User Interface Software and Technology*, pp. 211–220, 1991. doi: 10.1145/120782.120805

[27] M. Qureshi and F. Sabir. A comparison of model view controller and model view presenter. *arXiv preprint arXiv:1408.5786*, 2014.

[28] A. Satyanarayan, D. Moritz, K. Wongsuphasawat, and J. Heer. Vega-lite: A grammar of interactive graphics. *IEEE Trans. Vis. Comput. Graphics*, 23(1):341–350, 2016. doi: 10.1109/tvcg.2016.2599030

[29] A. Satyanarayan, R. Russell, J. Hoffswell, and J. Heer. Reactive Vega: A Streaming Dataflow Architecture for Declarative Interactive Visualization. *IEEE Trans. Vis. Comput. Graphics*, 22(1):659–668, Jan. 2016. doi: 10.1109/TVCG.2015.2467091

[30] A. Satyanarayan, K. Wongsuphasawat, and J. Heer. Declarative interaction design for data visualization. In *Proceedings of the 27th Annual ACM Symposium on User Interface Software and Technology*, pp. 669–678, 2014. doi: 10.1145/2642918.2647360

[31] L. Shao, A. Mahajan, T. Schreck, and D. J. Lehmann. Interactive regression lens for exploring scatter plots. *Computer Graphics Forum*, 36(3):157–166, 2017. doi: 10.1111/cgf.13176

[32] B. Shneiderman. Direct manipulation: A step beyond programming languages. In *Proceedings of the Joint Conference on Easier and More Productive Use of Computer Systems*, p. 143, 1981. doi: 10.1109/mc.1983.1654471

[33] I. Square. Crossfilter: Fast multidimensional filtering for coordinated views, 2013.

[34] Tableau software. https://www.tableau.com/, 2003. Accessed: 2019-07-20.

[35] J. Thomas and J. Kielman. Challenges for visual analytics. *Information Visualization*, 8(4):309–314, 2009. doi: 10.1057/ivs.2009.26

[36] C. Tominski. *Interaction for Visualization*. Morgan & Claypool, 2015. doi: 10.2200/s00651ed1v01y201506vis003

[37] C. Weaver. Building highly-coordinated visualizations in improvise. In *IEEE Symposium on Information Visualization*, pp. 159–166. IEEE, 2004. doi: 10.1109/INFVIS.2004.12

[38] H. Wickham. A Layered Grammar of Graphics. *Journal of Computational and Graphical Statistics*, 19(1):3–28, 2010. doi: 10.1198/jcgs.2009.07098

[39] H. Wickham. *ggplot2: Elegant Graphics for Data Analysis*. Springer Publishing Company, Incorporated, 2nd ed., 2016.

[40] L. Wilkinson. The grammar of graphics. In *Handbook of computational statistics*, pp. 375–414. Springer, 2012.

[41] N. Wong, S. Carpendale, and S. Greenberg. EdgeLens: An Interactive Method for Managing Edge Congestion in Graphs. In *IEEE Symposium on Information Visualization*, pp. 51–58, 2003. doi: 10.1109/INFVIS.2003.1249008

[42] J. S. Yi, Y. ah Kang, J. Stasko, and J. A. Jacko. Toward a deeper understanding of the role of interaction in information visualization. *IEEE Trans. Vis. Comput. Graphics*, 13(6):1224–1231, 2007. doi: 10.1109/tvcg.2007.70515

[43] J. S. Yi, R. Melton, J. T. Stasko, and J. A. Jacko. Dust & Magnet: multivariate information visualization using a magnet metaphor. *Information Visualization*, 4(3):239–256, 2005. doi: 10.1057/palgrave.ivs.9500099