Libra: An Interaction Model for Data Visualization

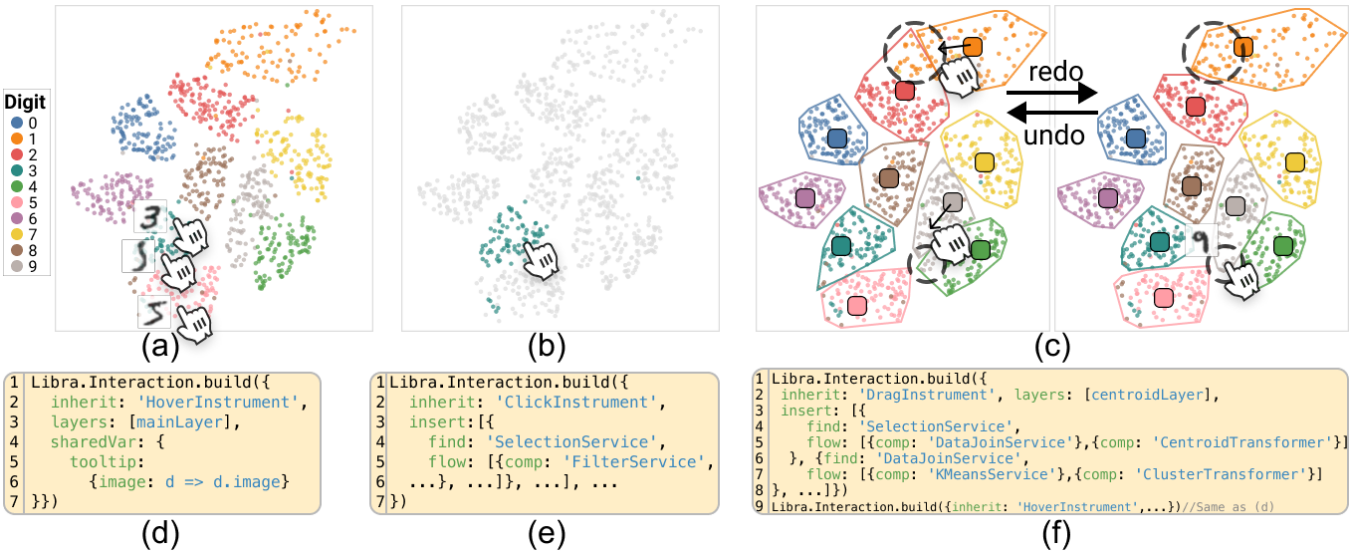
Yue Zhao
Shandong University
Qingdao, China
jack.zhao9802@gmail.com

Yunhai Wang*
Renmin University of China
Beijing, China
wang.yh@ruc.edu.cn

Xu Luo
Renmin University of China
Beijing, China
luoxu9days@gmail.com

Yanyan Wang
Ant Group
Hangzhou, China
shiwu.wyy@antgroup.com

Jean-Daniel Fekete
Inria & Université Paris-Saclay
Orsay, France
Jean-Daniel.Fekete@inria.fr




```
Libra.Interaction.build({
  inherit: 'HoverInstrument',
  layers: [mainLayer],
  sharedVar: {
    tooltip:
      {image: d => d.image}
}})
```

```
Libra.Interaction.build({
  inherit: 'ClickInstrument',
  insert:[{
    find: 'SelectionService',
    flow: [{comp: 'FilterService',
  ...}, ...]}, ...], ...
})
```

```
Libra.Interaction.build({
 inherit: 'DragInstrument', layers: [centroidLayer],
 insert: [{
    find: 'SelectionService',
    flow: [{comp: 'DataJoinService'},{comp: 'CentroidTransformer'}]
  }, {find: 'DataJoinService',
    flow: [{comp: 'KMeansService'},{comp: 'ClusterTransformer'}]
}, ...])
Libra.Interaction.build({inherit: 'HoverInstrument',...})//Same as (d)
```

Figure 1: Libra facilitates efficient interaction modeling by enabling the reuse, extension, and combination of built-in interactions. Its prototype, `Libra.js`, supports seamless exploration of the t-SNE projection of the MNIST dataset with various interactions: (a) hovering a point to show the corresponding image, (b) clicking a data point to highlight the whole class, and (c) dragging cluster centroids to interactively refine k-means clustering while seamlessly integrating with point hovering from (a). (d,e,f) The corresponding `Libra.js` code snippets for the interactions in (a,b,c), respectively.

ABSTRACT

While existing visualization libraries enable the reuse, extension, and combination of static visualizations, achieving the same for interactions remains nearly impossible. Therefore, we contribute an interaction model and its implementation to achieve this goal. Our model enables the creation of interactions that support direct manipulation, enforce software modularity by clearly separating visualizations from interactions, and ensure compatibility with existing visualization systems. Interaction management is achieved through an *instrument* that receives events from the view, dispatches these events to graphical layers containing objects, and then triggers actions. We present a JavaScript prototype implementation of our model called `Libra.js`, enabling the specification of interactions for visualizations created by different libraries. We demonstrate the effectiveness of Libra by describing and generating a wide range of existing interaction techniques. We evaluate `Libra.js` through diverse examples, a metric-based notation comparison, and a performance benchmark analysis.

*corresponding author

CHI '25, April 26–May 1, 2025, Yokohama, Japan

ACM ISBN 979-8-4007-1394-1/25/04…$15.00
https://doi.org/10.1145/3706598.3713769

CCS CONCEPTS

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

KEYWORDS

Information visualization, interaction, software modularity, direct manipulation, undo/redo

ACM Reference Format:
Yue Zhao, Yunhai Wang, Xu Luo, Yanyan Wang, and Jean-Daniel Fekete. 2025. Libra: An Interaction Model for Data Visualization. In *Proceedings of CHI '25: CHI Conference on Human Factors in Computing Systems Proceedings (CHI '25).* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3706598.3713769

1 INTRODUCTION

Interaction is a fundamental aspect of visualization. Well-designed interactions empower users to explore visualized data effectively while providing an engaging experience [14, 56]. As a motivating example, Figure 1 illustrates the interactions with a scatterplot visualizing the t-SNE projection [58] of the MNIST dataset [70] containing images of handwritten digits, each digit being labeled with its class, i.e., the digit it represents, from 0–9. Assume the analyst asks an application developer to provide custom interactions to evaluate the projection and clustering quality. Our goal is to reuse, extend, and combine existing interactions to create specialized ones tailored to specific visualizations. For this example, an application programmer will *reuse* existing interactions (hover, click), *extend* them, and *combine* a "drag" interaction on top. Hovering over a data point displays its corresponding image (Figure 1a), and clicking highlights *all points of the same class* (Figure 1b). Overlaying the projection visualization, the cluster centroids are visualized as large rounded square points providing an interaction *combined* with data point hovering interaction in Figure 1a. Dragging a cluster centroid moves it to escape a possibly non-optimal position and triggers the recomputation of the k-means clustering, updating the visualization (Figure 1c). These interactions can be structured by reusing, extending, and combining basic interactions such as hover, click, and drag. Hovering and clicking reuse basic interactions while extending them with customizations, such as tooltip adjustments or specialized selection actions. The k-means example further combines dragging cluster centroids with hovering over projection points, enabling different behaviors based on point semantics. Additionally, interactions designed for specific visualizations, like t-SNE projections, can be easily adapted to other contexts, such as node-link diagrams.

Although existing libraries follow the same visualization reference model for the rendering pipeline [11], their interaction techniques vary widely, with no clear consensus on the underlying concepts, mechanisms, or their interplay. The widely-used D3 library [9] offers only three predefined reusable interaction techniques: brush, drag, and zoom. While D3 allows for implementing custom interactions through event-handling callbacks, it requires application programmers to manually manage interaction states [16]. Returning to our example, developers have to manually handle the states of the tooltip in Figure 1a, and disambiguate the click event between a projected point and a class centroid in Figure 1b. In Figure 1c, even with the predefined D3 "drag" mechanism, developers still need to combine point-clicking and dragging interactions to implement the interactive k-means clustering. Furthermore, D3 provides no guidance or support on how to reuse or extend user-improved interactions.

Vega [51, 52] (and Vega-Lite [50] which relies on Vega) models input events as streaming data and relies on *functional reactive programming* (FRP [62]) to apply various data transformation operators and visual encoding primitives to these streams. This approach supports event stream creation and composition but tightly couples visual feedback with visual representation during interactions. Therefore, it is difficult to reuse and adapt Vega's interaction specification to other visualizations. In addition, it lacks built-in undo/redo support, which complicates the implementation of interactions like interactive k-means clustering shown in Figure 1c. *While new visual representations are quickly incorporated into visualization libraries, new interactions often remain confined to research papers and prototypes, rarely influencing the broader visualization landscape.* We speculate that this is due to the lack of well-defined software models for interaction design compared to rendering.

We introduce an interaction model that aims to meet three types of requirements: *direct manipulation* [53], *software modularity*, and *compatibility* with existing implementations of libraries. *Direct manipulation* interfaces require a set of properties (e.g., physical actions, rapid, reversible operations, and layered or spiral approach to learning) that are considered essential in HCI. However, these properties are rarely fully supported in existing visualization libraries, likely due to the complexity and cost of implementation without proper software support. *Software modularity* refers to the logical partitioning of library design that allows complex software to be manageable for the purpose of implementation, extension, and maintenance. Interactions are currently not manageable in a modular way, taxing their development. *Compatibility* implies that our model requires relatively small changes to existing libraries to be retrofitted. Our model and sample implementations are designed to help increase the quality of interactions supported by visualization libraries with limited added complexity. We advocate for a model, not one implementation, because we want to help improve existing implementations. We also want to clarify how interaction works in visualization at a conceptual level.

One of our goals is that interactions become *first-class citizens* [1, p. 102] in visualization libraries, i.e., that they may be named by variables, passed as arguments to procedures, returned as results of procedures, and be included in data structures. Furthermore, we introduce a complete, modular, component-based model for interaction design in visualization that developers can follow in their implementations. By translating the concept of interaction into a comprehensive model, we elevate interactions to a higher level of abstraction. Our approach enables interactions to be descriptive, evaluative, and generative [4].

We present implementations of our model through a JavaScript-based working prototype called `Libra.js`[1]; it enables the declarative specification of interaction techniques for static visualizations created by three libraries: D3, Vega, and Observable Plot [46] (Plot in the remaining). We show that a diverse range of interaction techniques can be seamlessly integrated within and across visual designs and input modalities. As illustrated in Figure 1(d-f), our model allows developers to add non-trivial interactions to a new visualization application with just a few lines of code. Interactions can be referred to by name and added to visualizations, with default parametrization and the flexibility to customize them to fit the specific application without requiring a complete re-implementation.

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

Relying on our model, the interaction defined in Figure 1(d-f), can be implemented by reusing, extending, and combining existing interactions, ensuring consistent interaction specifications across visualizations (see interaction with scatterplot matrices and node-link diagrams in supplemental material). In Figure 1f, the built-in history management mechanism allows developers to effortlessly implement undo/redo operations for interactive k-means clustering, and provide it for future reuse. Moreover, as more interactions become available in our prototype `Libra.js`, either through library updates or third-party contributions, they can be integrated into existing applications with minimal effort.

We perform a two-fold evaluation. First, we demonstrate the effectiveness of Libra by describing and generating many existing interaction techniques with it. After reviewing the papers introducing novel interaction techniques in the VisPubData collection [34], we map their interactions to Libra's components and, for several of them, articulate the technical scope of their contribution. Second, we evaluate `Libra.js` both qualitatively and quantitatively across three aspects. (1) we construct a diverse set of interaction techniques that integrate seamlessly within and across visual designs and input modalities. (2) we employ a metrics-based approach [39] to compare the usability of notations for interaction specifications between `Libra.js` and existing libraries. (3) we conduct benchmark evaluations of interactive visualization libraries, showing that `Libra.js` matches Vega's performance and outperforms D3. The results demonstrate that our model promotes interactions as reusable components in visualization systems without any performance penalty.

In summary, our contributions are:

(1) the description of our model of interaction that integrates and organizes multiple components from the literature to support the creation, reusability, extensibility, and composability of interactions;
(2) a prototype implementation in JavaScript: `Libra.js`, introducing declarative APIs to *create/reuse/extend/combine* interactions for visualizations built with different visualization libraries; and
(3) a two-fold evaluation: (1) assessing Libra by demonstrating its ability to describe and generate a wide range of existing interaction techniques for visualization, and (2) evaluating `Libra.js` qualitatively and quantitatively through diverse examples, a metric-based notation comparison [39], and a performance benchmark analysis.

2 BACKGROUND

Before describing previous work, we introduce four types of interaction stakeholders who will benefit from our interaction model:

EU the *visualization end user (the analyst)*, who will have richer interaction techniques available;
AD the *visualization application developer*, who will have a larger library of interaction techniques that can be easily reused and combined in a way similar to developers reusing existing visualization techniques;
ID the *visualization interaction technique developer*, who will be able to design a large set of components for interaction techniques, extensible and combinable, in a way similar to visualization developers who can design new visualization techniques from scratch and share them (**ID** can sometimes also be **AD**);
LD In addition, we also refer to the *interaction library developer*, who will implement our interaction model in a new or existing visualization library.

Libra draws on previous work in HCI, interactive visualization toolkits, and interaction software models.

2.1 Direct Manipulation in VIS

Shneiderman introduced in 1983 the "direct manipulation" principles [53]: (1) Continuous representation of the object of interest; (2) Physical actions or labeled button presses instead of complex syntax; (3) Rapid, incremental, reversible operations; (4) Layered or spiral approach to learning. These principles remain the golden standard of HCI, but visualization systems only meet principle (1), whereas 2–4 are rarely met.

Interaction Specification. For principle (2), an input visualization can be interacted with by clicking on a graphical item to select it, dragging an item to move it, or adjusting a slider to filter items. However, the visualization pipeline [11] does not explain how graphical marks, axes, and coordinate systems interact with these actions. For example, parallel coordinate plots often allow dragging a range selection on a data axis. A few scatterplot visualizations also allow that type of range selection by dragging on the axes. Yet, this kind of interaction performed on the axis is not standard and not mentioned by the visualization pipeline or the Grammar of Graphics (GoG). As a result, the support for interactions (where and how) varies significantly across libraries, potentially confusing users.

Interactions were initially implemented using event-based programming with callback functions, leading to a "spaghetti of callbacks" [43]. Despite this, several visualization toolkits like Protovis [8] and D3 [9], still rely on event-handling callbacks for custom behaviors. These often require **ID** to maintain a state machine and coordinate interleaved calls, making it difficult to reuse, extend, and combine. To manage the complexity, the simplest approach is to encapsulate it within a black-box object, allowing **AD** to utilize it without exposing its intricacies. Toolkits like Prefuse [31], Improvise [64], VisDock [15], and DIVI [54] provide such predefined interaction techniques, such as selection, navigation, and annotation, for developers to integrate into their applications. While this approach simplifies and streamlines reuse for **AD**, it is rigid and monolithic from the perspective of **ID**.

Alternatively, Vega [51, 52] offers higher-level mechanisms for specifying interactions. It uses FRP [62] to provide composable interaction primitives and implicitly manage state machines. By abstracting input events as data streams, Vega allows events to be treated as explicit inputs to visualization specifications, enabling rich interactions for **EU**. Vega-Lite [50] builds on this by introducing a high-level grammar for interaction specifications, using selections as primitives, and simplifying interaction creation for **AD**. However, FRP does not explicitly describe the feedback during interactions, relying instead on internal, non-exposed mechanisms. While extensions such as signal recording [32] are possible, it is

difficult for ID to create, reuse, extend, and combine expressive interactions. In contrast, Libra provides a complete and transparent framework for describing and implementing all stages of interactions in visualizations.

Undo/Redo. According to principle (3), supporting Undo/Redo is standard in most professional user interface toolkits, relying on the Command design pattern described in [61]. Instead of directly performing the actions in the user interface code, a Command object is created, which supports the methods "execute" and "undo". A history manager is given this command object; it first executes it and stores it in a list so the user can undo it later.

In visualization, a few libraries implement the Command design pattern and an interactive history manager. VisTrails [10] supports it for provenance management, and Trrack [18] supports history management in JavaScript visualization systems. However, undo/redo is not supported by the visualization libraries directly (more on section 5).

Affordances, Feedback, and Feedforward. Most popular visualization libraries follow the GoG. Yet, interactive visualizations need to manage graphical objects that are not part of the GoG to implement principle (4): *affordances* to help users understand where interactions can be performed, *feedback* to show how objects are changed during the interactions, and *feedforward* to show what would happen at the end of an interaction early on.

Besides the graphical items in the input visualization, interactions often generate additional graphical items (e.g.,, the image shown by hovering in Figure 1a), which can create potential interference in understanding their role. According to Don Norman [45], the best way to avoid these interferences is to separate the marks into different affordances, overlaid in *layers*, synchronized to share the same view and coordinate systems but disjoint. In doing so, the selection can then be updated without changing the graphical structure managed by the visualization, and the visualization is not affected by the interactive selection changes.

This separation of concern is important for modularity, not only for the selection but also for other transient objects used in interactions [22, 23]. For example, an interaction implementing lasso selection by dragging the pointer requires drawing the selection lasso. This object is a feedback of the ongoing interaction; it does not belong to the visualization or the selection layer. For the sake of modularity, graphical objects with specific semantics should then be drawn in their layer to avoid interference and resolve ambiguity during picking [23]. Improvise [64] also uses layers to separate the visualization from the graphical objects used for the interactions. Visualization systems such as ggplot2 and Vega support layering for visualization purposes only, e.g.,, adding a regression curve on top of a scatterplot. Our model also relies on layering to achieve modularity, extensibility, and composition.

Many standard interactive tools provide feedforward, but very few visualization systems provide it. Implementing feedforward is difficult without library support because it requires changing the graphical appearance of the visualization without changing it for real. Although the HCI and visualization literature praise the benefits of feedforward, no library provides specific support for it. With Libra, we describe possible mechanisms to support it in generic ways by relying on undo/redo to show the result early and undo it if not validated. Layers can also show the animation of actions, such as layout changes, without performing them on the main visualization.

2.2 Interaction Models

Interaction models were a popular topic in HCI research to provide abstractions, mechanisms, and guidance for the implementation of graphical user interfaces (GUIs). There has been a long history starting with Smalltalk's MVC model [49] that separates software components into three parts: the Model, View, and Controller. After evolving through many steps [29], a few MVC variants like the Model-View-Presenter model [48] have been adopted by the industry. Based on these models, component-based architectures [41] that promote the separation of concerns between components and reusability have become popular for building user interface components. For example, React [27] provides a set of components that can be composed to build complex user interfaces, allowing customization of most aspects of each component. Unfortunately, existing components are tied to their specialized interactions and vice versa, and, to our knowledge, none of the existing component architecture aims at separating them.

The visualization reference model is a high-level architecture model [11] that has been adopted and improved by the visualization community [30, 37]. It describes well the rendering pipeline used in information visualization, from data to view, but it is less precise in explaining the interaction part, not mentioning any particular component dedicated to event handling. Heer and Agrawala [30] have listed several design patterns used in the Prefuse system [31]. However, their patterns address only a few aspects of interaction. Most importantly, a set of patterns only provides partial solutions to a system's architecture. Yi et al. provide a taxonomy of interaction techniques [71] including seven categories (select, explore, reconfigure, encode, abstract/elaborate, filter, and connect), which is abstract.

The instrumental interaction model [3] is a general 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 [37] adapted this model to visualization settings by unifying it with the visualization reference model [11] for describing, comparing, and criticizing beyond-desktop visualization systems. However, it is still a conceptual model. It does not identify the essential components for the implementation of interaction techniques and thus cannot be used for generating new interactions. In contrast, our model describes interactions concretely.

3 LIBRA INTERACTION MODEL

In this section, we first present the design goals for our interaction model, then describe the essential components for characterizing interactive visualizations, and finally highlight the differences between our model and the ones supported by existing libraries.

3.1 Design Goals

To fully support the design, reuse, extension, and combination of visualization interactions, we established the following three design goals:

DG1: Complete and transparent model to fully implement direct manipulation. The model should allow complying with Shneiderman's principles of direct manipulation [53] as explained in section 2.1. To apply to any data visualization, it should cover all aspects of interaction management for 2D interfaces. To facilitate extensibility, our model exposes all its mechanisms transparently. In terms of expressive power, it should allow for building interaction-rich visualizations beyond stereotypes, including all the exemplar interaction techniques such as Excentric Labels [6, 25], Dust & Magnets [72], and DimpVis [38].

DG2: Software modularity to maximize reuse, extension, and combination of interactions. Our model provides the ability to create completely new interactions and reuse, extend, and combine existing ones, allowing **AD** to create interaction-rich applications and **ID** to enrich the interactions available to **AD** and eventually to **EU**.

DG3: Compatibility with the existing libraries while separating interaction from the visual representations. Several existing visualization systems can create rich but static visual representations (e.g., by implementing the GoG [67]). Our model allows **LD** to reuse existing systems with simple adaptations, mainly by separating the background and foreground with layers. In doing so, our model can maintain high compatibility with existing visualization systems by "wrapping" interactions around them.

Here, "reusing" refers to the ability to directly utilize already designed interactions in a visualization. "Extending" implies that some components of an interaction can be added or removed. "Combining" means either composing multiple interaction techniques sequentially or running them in parallel, e.g., a hover interaction can be combined with a drag interaction as shown in the example of Figure 8.

3.2 Our Model

Figure 2a illustrates the architecture of our model, where interaction *instruments* manage interactions originating from the visualization views. Each instrument, associated with a view and its *layers*, acts as a mediator between the user and the visual presentation of the data. It transforms user input into parameter changes across the data transformation, visual mapping, or view rendering stages of each layer. These changes produce *interaction effects*, which might involve visualized data items and provide corresponding feedback in the relevant layers.

Building on this architecture, interactions with a visualization view can be formally defined as:

Interaction:= (name, layers, instruments, effects),

where *name* uniquely identifies the interaction, and *layers*, *instruments*, and *effects* represent the target, means, and results of the interaction, respectively. This structured model allows most **AD** to combine instruments and, when necessary, tailor them to specific needs by leveraging the defined components.

3.3 Layers

Layers are components providing methods for drawing visual elements, picking them, and cloning them to another layer. They are essential for separating the interaction management, involving transient objects, from the main visualization, satisfying DG3. Each layer can manage a data model that creates and updates the layer's visual elements using a *graphical transformer*. Libra relies on the following four standard layers for visualization applications:

- **Background layer**: under the main layer, displays the visual elements (e.g., axes and tick lines). By default, it does not react to picking. Its data model is composed of the visualization axes, rendered in the background with tick lines and labels.
- **Main layer**: shows the main visual representation and allows picking visualized data items. Its data model is the static visualization data model rendered as the visual items associated with the visualized data items.
- **Selection layer**: stacked on top of the main layer, visualizes the selected data items without any awareness from the main layer's visualization and allows picking them. Its data model is a set of items from the main layer's data model that are selected, and rendered using the same geometry as the item in the main layer using specific visual channels to highlight them.
- **Transient layer**: on top of the selection layer, shows and dynamically updates transient interaction objects, such as a selection rectangle used to select the visualized items interactively. By default, it does not react to picking. Its data model is simply a list of visual objects to render directly.

Layers are stacked and displayed in the order described above, although an instrument can change it if needed. For more complex interactions, an instrument can create a new layer; see the label layer and histogram layer used by the Excentric Labeling instrument in Figure 3c.

3.4 Instruments

As shown in Figure 2b, instruments translate the received low-level events into higher-level actions through *interactors* that are executed by *services*. In this way, each instrument with a unique *name* can be specified as follows:

Instrument:= (name, interactors, services).

Since one instrument manages multiple layers, it can evaluate any of them to determine how to interpret an event. For example, when the selection instrument receives a *mousedown* event, it first checks the selection layer. If the selection layer is empty and the main layer contains an item under the mouse, the item is selected and passed to the *selection service* (see below) to update the data model and selection layer. If no item is found under the mouse, the instrument can create a rectangle in the transient layer and initiate a rectangle selection (brushing) by starting an interactor (see below).

Based on a set of existing instruments, **AD** can extend them by replacing or inserting new interactors and services. Figure 2b shows how different components communicate within an instrument. After receiving an input event, the instrument decides how to handle it according to the event position, the contents of the layers, and if an *interactor* has already been created to handle further events. The instrument can start an interactor to map low-level events to high-level actions. Next, the interactor's actions are interpreted by e.g., invoking the selection service to select visual items of interest

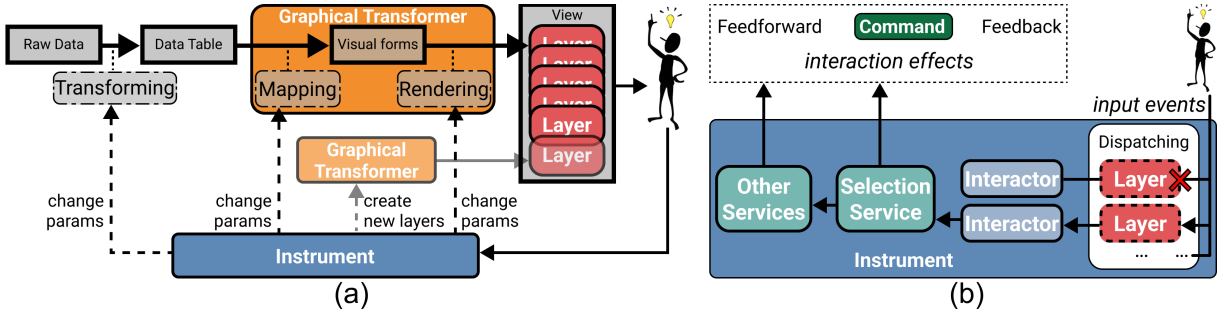


Figure 2: Our interaction model (a) and the communication between components within an instrument (b). (a) The top row represents the visualization reference model, where our model wraps the visual mapping and view rendering into a *graphical transformer*. The bottom row corresponds to the instrument for manipulating the three stages. The objects resulting from the interactions, transient or persistent, are shown on the new layers, their associated visual effects updating the view. (b) For the input events, the instrument first interprets them in the context of the layers and uses an *interactor* to translate them into high-level actions. These actions subsequently use the *selection service* or other services, and produce interaction effects—*feedforward* and *feedback*—that are shown in their corresponding layers, and commands that enable undo/redo.

for further manipulation, potentially using other services. Finally, the instrument generates *interaction effects*, including *feedforward* (optionally) and *feedback* on the layers—such as moving a cluster centroid in Figure 1c—and, in the end, produces a *command* object that records the involved services and the changed data items–such as the changed centroid position. In doing so, the instrument separates the event-driven part done by the *interactor* from the interpretation of the sequence done by the *service*. We now describe the interactors, services, and effects in detail.

Interactors. In Libra, we call *interactor* any kind of state machine that transforms the sequence of events received by the instrument into higher-level actions. When an instrument uses an interactor, it binds its actions to instrument-specific actions. For example, when the drag instrument receives a *mousedown* event with an item under the mouse position, it creates a trace interaction. The start action saves the current selection. The running action moves the item along the mouse trace. The stop action, triggered by the *mouseup* event, creates a selection command with the collected selection and restores the saved state. The command returned to the instrument is eventually added to the *history manager*, a global component responsible for managing commands, and then executed.

New interactors can emulate the default one, e.g., for handling a tablet instead of the mouse, providing actions are compatible with the default interactor, hence with all the instruments relying on it. If incompatible, it can provide a different set of actions, requiring the extension of instruments by **AD** or **ID**, or the creation of a new instrument by **ID**.

Services. During interactions, fundamental operations (e.g., selection) or common computations (e.g., analysis or layout) are often used across multiple instruments. To enhance modularity and consistently support undo/redo, these are encapsulated as **services**, which manage interaction-related functionality, undo/redo, and states rather than serving as mere function calls or data storage. In addition to the default ones, services can be flexibly extended based on interaction needs, such as an analysis service for tasks like interactive k-means clustering.

Each service provides core functionalities, including state management through shared variables, computational operations, and inter-component communication. Services are often linked to graphical transformers to display the data items they manage (e.g., the selection service in Figure 3c) and can share processed items with other services for further manipulation. Services also coordinate with other components, notifying graphical transformers (if present) to update their layers upon completing their tasks. In the following, we describe the selection, layout, and analysis services.

Selection Service. For automatically managing the selection over visual and data spaces, each instrument in Libra uses a default selection service, which 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 and update the selection when performing composite dynamic queries. In our model, selection can be performed in data space (similar to the SQL WHERE clause) or visual space (by picking), according to the instrument's semantics. The graphical transformer associated with the selection service iterates over all the selected items, accesses the related graphical elements from the main layer, and either copies them on the selection layer or creates proxies, changing some of their visual appearance (e.g., color) to look highlighted.

Once the target objects are selected, other services can use the selection to manipulate them or to introduce new related elements (e.g., annotations). The cascade of creations can update the transient layer, selection layer, or others. In other words, our model supports a cascade of models to handle sequences of interactive data analysis tasks.

For a given target object, additional visual elements around (and on a layer above) the selected object can be created and processed (see Figure 8). If the target object of the interaction is one or multiple items (e.g., hovering), a visual proxy of these items is usually added to the selection or transient layer to appear highlighted in the same view. Moreover, some interactions (e.g., a rectangular brush) require identifying the value range that encloses the selected items

for highlighting the ones inside, translating queries from visual to data space.

Layout Service. Most visualization toolkits provide a few layout algorithms allowing for direct manipulation, such as force-directed graph layouts. Layout services decouple the implementation of the actual layout from 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, the service shares the obtained positions of all items with the corresponding graphical transformer for updating visual encoding.

Analysis Service. Visual analytics applications often employ data analysis algorithms including clustering, regression, and classification. Our model supports an analysis service to run statistical or machine learning algorithms on the data of interest and manage the results as a dedicated service. The data model managed by these services can be visualized in a specific layer. For example, the clustering service in Figure 1 shows the centroids of clusters computed on multidimensional items, and the histogram service in Figure 3 shows the distribution of a quantitative attribute for the selected items, overlaid on the main layer.

Interaction Effects. Interaction effects consist of **feedforward** and **command** with their **feedback**. During and after performing an action, feedforward and feedback need to be shown similarly to the user. Yet, the *feedforward* is usually shown in a transient layer, while the feedback can be shown in the main or another persistent layer. If any data item or visual element in the main layer is changed, the main layer will be refreshed; if there is a selection layer, the changed selected elements are then updated.

When an action is performed by an instrument (e.g.,, selecting items), all the services invoked along with the data they modify are recorded in a *Command* object. This object is managed by a *history manager*, a global component that manages operations independently of any specific service for undo/redo. This design ensures consistent tracking of interactions and allows for undo when needed. In Figure 1a, the Command updates the selected status of data items, and the visual feedback highlights the selected items in blue. The Command is first executed and then stored in the history manager. For continuous commands like dragging a selection rectangle, each new command replaces the latest in the history manager.

Communication between Components. Our model requires coordinating the communication between multiple components. For example, an instrument can call a service API, and the service can forward the changes to other services or update a layer through a graphical transformer. The instruments associated with the layers among different views also need to communicate. Libraries use various mechanisms for coordination, such as shared variables in ggplot2 or live properties in Improvise. Our model does not prescribe a particular mechanism but requires one for communication and coordination. In doing so, a dataflow graph among components is constructed. As a user provides input events to one view, an update propagates through the dataflow graph and triggers updates to the related layers.

Since an instrument is composed of well-specified components, **ID** can compose new instruments by assembling these components, and **AD** can reuse these instruments to provide rich interactions to **EU**. Each instrument can be adapted across input modalities (e.g., mouse and touch) by re-binding different input events into the interaction; it can also be extended by involving different interaction services. In doing so, both DG1 and DG2 are satisfied.

3.5 Comparing Libra to Existing Models

Table 1 shows the differences between our model and the support of popular visualization libraries along five aspects related to interaction management.

Layer. While layers have been used by ggplot2 [65] and Vega/Vega-Lite to superimpose multiple semantically related representations in a view, they are not used to specify interaction. So far, only Improvise and IVTK match our model's use of layers to manage both visualization and interaction.

Interactor. Instead of implementing an interactor as a state machine, Improvise, IVTK, Prefuse, and D3 use the callback model that requires users to manually maintain all states, except D3 providing a few generic interactions, such as *pan* and *zoom*, that can trigger user-defined actions. Using FRP streams and signals, Vega and Vega-Lite provide the mechanism to support a state machine. All the implementations bind the state machine to actions, leading to a strong coupling between visual representation and interaction feedback. In contrast, our model introduces two stages in the management of events: interactor to transform low-level events into high-level actions and binding of high-level actions to commands. The first stage allows reusing interactors for different interactions similar at the state-machine level [42] and the second is the basis for reversible actions [53] (undo/redo).

Selection as Service. Since D3 does not use layers, it only supports the selection of the nodes in a document object model (DOM) tree, that usually contains all the visual marks. For visual marks in one specific layer, Improvise, IVTK, and Prefuse match our model which supports selecting them in both data and visual space. In contrast, the selection model provided by Vega/Vega-Lite is done exclusively in data space and is difficult to adapt to work in the visual space, e.g., for selecting a complex shape (as in Figure 8).

Command. None of the libraries provide them with persistent data changes. While Improvise maintains a history of specific high-level (variable change, query, render) and low-level (keyboard, mouse, painting) interaction events in a running visualization, it does not implement undoing these actions or any others. IVTK and Prefuse have command-like objects but do not support undo/redo. To maintain the separation between interactors and data services, our model

Table 1: How existing systems' interaction models match our model, regarding five components: interactor, layer, selection, command, and feedforward; * means partial support, "Vis" and "Int" indicate support for visualization and interaction, respectively.

	Improvise	IVTK	Prefuse	D3	Vega/Vega-Lite
Interactor	×	×	×	×	✓
Layer	Vis+Int	Vis+Int	×	×	Vis
Selection as Service	✓	✓	✓	×	✓*
Command	✓	✓	✓	×	×
Feedforward	×	×	×	×	×

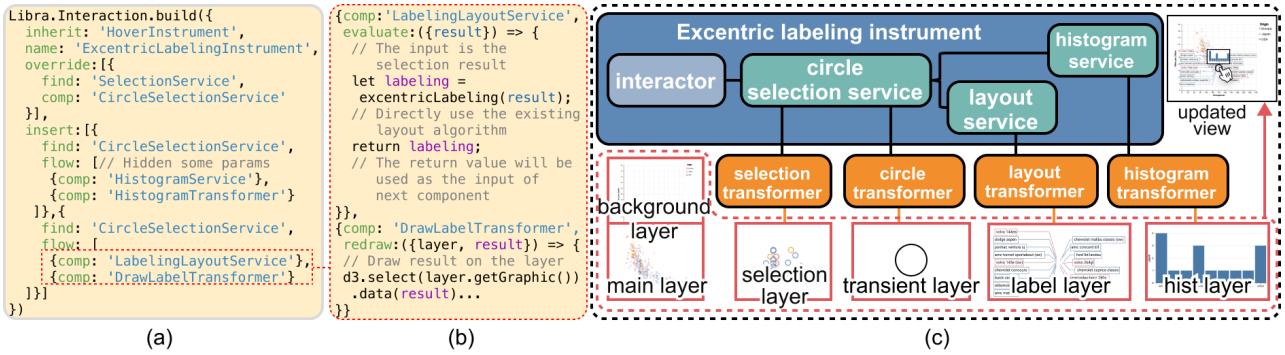


Figure 3: Illustrating the design of a new instrument with `Libra.js`. (a) The specifications for creating an excentric labeling instrument [6, 25]; (b) the specifications for creating a new layout service and the corresponding transformer; (c) (top left) The components used for composing an excentric labeling instrument to support this interaction for scatterplots; (bottom left) the main layer and the other three layers generated by interactions; and (right) the final result.

prescribes the use of commands supporting undo/redo, only recording high-level actions performed on the data services.

Feedforward. As far as we know, none of the existing libraries provide support for feedforward. Our model prescribes feedforward when possible, which is often made simpler to implement using the history management mechanism.

In summary, none of the libraries support undo/redo and feedforward, two essential mechanisms for direct manipulation characterized by our model. Improvise and IVTK show the lowest discrepancy to our model, but they do not provide a state machine, making it hard for ID to extend and combine interactions. All the systems except Vega/Vega-Lite provide a transparent model for interaction management. Although Vega/Vega-Lite provide a state machine, they do not allow for communicating the extracted signals with external components, and their execution model is hidden in a dataflow graph, making it harder for ID to reuse, extend, and combine interaction primitives.

4 IMPLEMENTATION OF OUR MODEL

There are many ways to implement our model, regardless of the library used. As a proof of concept, we implemented a prototype in JavaScript called `Libra.js` and show that it can specify interaction techniques to static SVG visualizations created by any JavaScript libraries. It only requires ID to implement the few methods of our *Layer API*.

Furthermore, we have added a declarative mechanism in our model that shows how interactions can be reused, extended, and combined with a simple syntax. Interactions can be designed in four ways: creating instruments, reusing instruments, reusing and extending/specializing instruments, and combining instruments. Although it can take specific ID skills to create a new instrument (understanding the full interaction model), AD skills are usually sufficient to reuse, extend, and specialize instruments. We present the *interaction* API designed to provide a concise and structured specification of interaction instruments. This type of specification consists of several properties such as *inherit, layers, insert, override*, and *remove* (see Figure 3a and Figure 4a). With the *interaction* API, AD and ID can easily specify interactions.

Figure 3 shows the specification for composing and extending the built-in hover instrument to explore a scatterplot. The built-in histogram service and the corresponding transformers are composed to show the statistics for the selected points, while a newly created labeling layout service and the corresponding transformer are used to display the labels. In the following, we mainly describe how we implement the three major components.

4.1 Layers

By default, all visual elements are rendered as SVG elements, and we extend the picking methods provided by the web browser to support the query of arbitrary graphical shapes. We implement the *background*, *main*, *selection*, and *transient* layers with SVG groups.

For creating the visual elements used during interactions, `Libra.js` provides built-in graphical transformers with redraw functions, which can be overridden. 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 representation. The transient layer can create graphical objects, such as a brushing rectangle or a lasso, typically as a visual representation for the selection instrument. Hence, users can create their objects in existing layers or create specific ones. Figure 3b shows the specification of creating a transformer to display all labels.

4.2 Interactors

A state machine can be implemented in many ways and our default interactor is based on a Garnet-like [42] state machine with three states: "Start", "Running", and "Outside." State transitions are triggered by low-level events, defined as

Interactor:= (name, state, transitions),

where the transitions determine how the interactor responds to different input events and changes states accordingly. 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 uses this information to trigger a higher-level action (see Figure 4b). When an instrument uses an interactor, it binds its generic actions to instrument-specific actions.

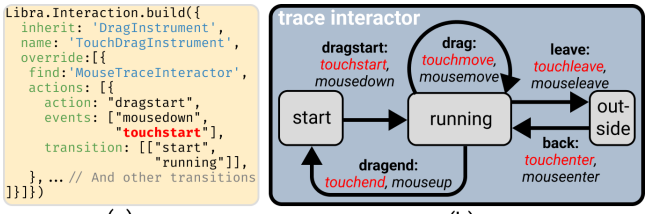


Figure 4: Extending the trace interactor used by the drag instrument in Figure 1c to support touch operations. (a) The `Libra.js` specification that adds the touch operations to the interactor; (b) a state machine with the transition between states triggered by different low-level events.

For example, when the drag instrument receives a *mousedown* event with an item under the mouse position, it creates a trace interaction. Its start action will find and save the current selection. Its running action will move the item following the mouse trace. Its stop action triggered by the *mouseup* event will create a selection command with the collected selection and revert the selection to the saved state. The command returned to the instrument is eventually added to the history manager and executed. The `Libra.js` specification in Figure 4a illustrates the extension of the trace interactor to support touch operations. Other interactors can be created by **ID** if needed, e.g., for speech-based or multimodal interaction; see the examples in the supplemental material.

New interactors can emulate the default one, e.g., for handling a tablet instead of the mouse, providing actions compatible with the default interactor hence with all the instruments relying on it, or incompatible, providing a different set of actions, requiring the extension of instruments by **AD** or **ID**, or the creation of a new instrument by **ID**.

4.3 Data Services

In `Libra.js`, services can communicate with other services and graphic transformers through shared variables. They notify the graphical transformers (if there are any) to update their layers when their work is completed. A service with a unique identifier is defined by:

Service:= (name, type, params, operationOrComputation).

where the *type* categorizes the service (e.g.,, selection, layout, analysis), *params* are configuration parameters for customizing the service's behavior, and *operationOrComputation* specifies the data processing or manipulation performed by the service.

`Libra.js` provides several built-in services for **AD** to reuse and extend, which 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, and the specific services are visualization or interaction-specific, e.g., layout services for arranging visual elements generated by interaction or e.g., histogram services triggered by interaction for inspecting the distribution of the data items, see examples in Figure 3.

Selection Service. In `Libra.js`, the graphical transformer associated with the selection service iterates over all the selected items, accesses the related 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. **Layout Service**. `Libra.js` provides the two standard data services: selection and layout. For our implementation of excentric labeling, as shown in Figure 3b, a label layout service is created and inserted in the interaction composition for manipulating the items selected, passed by the selection service. Here, `Libra.js` does not use a selection layer but a new *label layer* for showing the arranged labels because this layer has a different interaction semantics than the selection.

More advanced examples of the selection service and more services like analysis service are discussed in Section 2–4 of the supplemental material.

4.4 Commands, Feedback and Feedforward

To manage commands and enable undo/redo functionality, `Libra.js` leverages the Trrack history manager [18] and adheres to the Command design pattern. Commands are implemented as classes with three essential methods:

- execute(): executes the action and stores the current state of affected data models;
- undo(): reverts the action using the previously stored state; and
- redo(): re-applies the action

To support continuous interactions, **ID** can use a boolean flag to determine whether the command is continuous [36]. Once a command is executed, affected services will notify their corresponding transformers to update their respective layers. These transformers utilize the latest state of data models as the data source to provide visual feedback.

Feedforward. In `Libra.js`, feedforward is implemented by using a graphical transformer to render a set of transient objects on the transient layer, orchestrated by an instrument; it is invisible to the underlying data models. It indicates the running status of the actions to the user when the interaction starts. Once the interaction is finished, all feedforward objects are removed. See the example of the selection rectangle used for brushing in Figure 1a.

When feedforward is expensive, the response time and user experience can be affected by the computational cost of e.g., an ML algorithm or the number of elements involved. We suggest employing approximate or progressive methods [24] to achieve real-time feedforward. For example, this can be done by using a small number of iterations in ML or using a set of judiciously chosen data samples. There are several efficient approximate methods for query processing [12] and clustering [2]. It is crucial for **ID** to thoughtfully consider the intrinsic data characteristics, the algorithm involved, and the visualization to ensure that feedforward does not become detrimental to interaction.

4.5 Reuse, Extend and Combine of Instruments

To treat interactions as first-class citizens, `Libra.js` introduces high-level mechanisms to facilitate reuse, extension, and combination of interactions. **AD** can specify an interaction by customizing multiple aspects of an existing instrument using four operators: *insert*, *flow*, *override*, and *remove* (see Figure 3 and Figure 4).

Insert/Flow: Building on an inherited instrument, the *insert* operator first locates an existing service and then uses the *flow* operator to chain all the newly added services with a transformer. A newly added service receives the output of an existing service as its input and either passes its result to a service or, for the final stage, to a transformer. To facilitate the analysis of data items from multiple perspectives, users can define multiple flows with the results of these flows presented in distinct layers. A more formal definition of this operator is

insert(parentComponent, [flow1, flow2, ...]), where
flow := [newComponent1, newComponent2, ...].

The *insert* operator accepts a *parentComponent* and an array of *flows* as arguments. Each flow is an array of new components (e.g.,, services or transformers) that are sequentially chained together. For example, the specification "insert:[{find:'CircleSelectionService', flow:…}, …]" in Figure 3 adds the layout and histogram services as two separate flows.

Override: The *override* operator allows replacing an existing component with a new one with the syntax "override(existingComponent, newComponent)," while maintaining all connections in the interaction chain. As shown in Figure 4, the specification "override:[{find:…, transitions:…}]" shows that low-level input events can be reconfigured to support additional input modalities. Likewise, the selection service in Figure 3 is replaced by a circle selection service.

Remove: The *remove* operator deletes a specified component and, optionally, all its dependent components, maintaining the validity of the interaction chain. This behavior is controlled through the syntax "remove:[find:…, cascade:true/false]."

By chaining all components including interactors, selection service, other services, and graphical transformers through shared variables [60], `Libra.js` constructs a dataflow graph that manages the communication between the components. If `Libra.js` does not provide some desired services like the label layout service in Figure 3, **ID** can define them as new data services for **AD** to use them in existing interaction components.

4.6 Adapting to Different Libraries

`Libra.js` is designed to augment rather than replace existing visualization libraries by managing the interaction components of the visualizations created by the original library.

To specify interaction for SVG visualizations created by different libraries, **LD** is only required to provide an abstract base class of layers, inheriting and overriding the methods for managing and querying the visual marks specified by the original library. Specifically, they need to re-implement the initialization method for creating and managing the collection of visual marks. As for Vega and Plot, she only needs to specify the name of visual marks that belong to the same layer, while an additional "g" element is required in D3 to manage all related visual elements.

Since the visual queries can be performed in either data or visual space, she needs to re-implement the corresponding API. She is required to reimplement data-query methods since data is stored in different SVG DOM attributes in different libraries, while we provide a generic method for querying SVG visual elements. We refer to the D3 and Vega bindings as `Libra.js`-D3 and `Libra.js`-Vega, respectively. With the abstraction of our model, `Libra.js` shares the same interaction semantics for different libraries while providing a consistent interaction model. Specifically, the interaction specifications of `Libra.js`-D3 and `Libra.js`-Vega define different layers to be rendered by distinct engines. The original libraries render static visual elements, while those created during interactions are rendered by `Libra.js`-D3 or `Libra.js`-Vega. To ensure visual consistency, different layers share global information (e.g., scale, color mapping, etc.).

5 EVALUATION OF LIBRA

As an interaction model [3], Libra should have descriptive, generative, and evaluative powers. The former two powers allow it to capture a wide range of existing interactions for visualizations and assist designers in creating new ones, while the evaluative power provides metrics for comparing alternative interactions [3]. Interaction for visualization is defined as "the interplay between a person and a data interface involving a data-related intent, at least one action from the person, and an interface reaction perceived as such" [21]. This definition can be seamlessly mapped to Libra's components, such as layers, instruments (interactor and services), and interaction effects (feedforward, feedback, and commands). Specifically, action and data-related intent align with the interactor and service within the instrument, while reaction corresponds to interaction effects, with feedforward and feedback potentially revealed in certain layers. Given the broad design space of our model, suitable metrics for evaluating interaction techniques in visualization remain an open question. In the following, we only demonstrate the descriptive and generative powers of Libra by using it to analyze existing interaction techniques and don't discuss its **evaluative power**.

Descriptive power. To demonstrate the descriptive power of our model, we conducted a comprehensive analysis of 3753 papers collected in VisPubData [34]. We initially shortlisted 99 papers based on their contributions to interaction techniques. To identify relevant papers, we searched for terms related to interaction in the abstracts, such as "interact" and "interactive," and extended this list of terms to include additional terms like "interface," "insight," and "analyze." We then labeled these papers based on four key criteria: additional layers in interaction creation, extending generic interactors for low-level event handling, undo/redo commands, and services beyond selection for data transformation. Due to space constraints, we present a mapping table (Table 2) that illustrates only three interaction techniques as examples. A detailed version of the reference table and the labeled papers is available in the supplementary materials. The result shows that 94/99 surveyed papers require services for data transformation, highlighting the crucial role this component plays in interaction techniques. Additionally, 24 papers incorporate custom-made undo/redo commands, 18 necessitate extensions to generic interactors for low-level event handling, and 66 require additional layers for interaction creation. These numbers indicate the significance of layers and services in interaction techniques while also suggesting that generic interactors are adequate for most interactive visualizations. Notably, the relatively low usage of undo/redo commands in existing systems indicates that most

Table 2: Mapping Interaction Techniques to Libra Components

Component		Filtering and Dynamic Queries	Interactive Lenses	Dust & Magnet
Layers		Query interface as individual layer	Additional layer for lens effect	Magnet layer, dust layer, background layer
Interactor		Mouse trace interactor	Mouse position interactor	Mouse trace interactor for magnets, mouse position interactor for dust
Service	Selection Serv.	Selects data subset	Selects pixels or data subset	Selects data item
	Dedicated Serv.	N/A	Lens layout service	Magnet position service, dust layout service
Feedback		Reflected on main layer	Overlay of lens effect on base visualization	Updates on respective layers

systems suffer from the lack of a history management mechanism. To provide a clear understanding of the descriptive and generative power, we take three interaction techniques as examples.

Filtering and Dynamic Queries allow the display of interesting data subsets through a query interface (e.g., linked views or widgets). In Libra, the query interface is treated as an individual layer. An interactor translates low-level events into high-level actions, which then invoke the selection service to update the main layer. Instead of classifying this interaction at the data level [69] or view level [13], Libra seamlessly integrates it into the visualization pipeline, where the intent of data selection is managed by the service, and the reaction is reflected through feedback on the main layer.

Interactive Lenses create localized and temporary changes within a visualization, adjusting the visual representation in selected areas [57]. Once the lens is removed, the visualization returns to its original state. Aligning with the conceptual model of interactive lens [57], Libra defines this interaction with two services: a selection service to select pixels or data subset and a dedicated service for adjusting the visualization underneath the lens. By showing the lens effect on an additional layer, the final visualization is updated by overlaying it with the base visualization. By using specific layout services, the fisheye lens [63] and edge lens [68] for the node-link diagram can be described by Libra.

Dust & Magnet [72] is a multivariate data exploration technique composed of a magnet layer showing the attributes as colored rectangles that can be dragged, and a dust layer showing all data items as points that can be hovered. Clicking a pixel on the background layer creates a new magnet and recomputes the layout of all points that are updated 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 a different color.

After associating the magnet layer with the drag instrument, Libra defines two services: the magnet position service and the dust layout service. The magnet position service computes the positions of the magnets based on the user's drag actions, while the dust layout service calculates the new positions of the data points based on their attraction to the magnets. These new positions are shared with the respective layers' graphical transformers to update the visualization. Similarly, Libra associates a hover instrument to the dust layer to highlight its points of interest. In doing so, the drag action triggers the magnet position service and the dust layout service to compute the new positions, while the hover action does not conflict with the dragging.

By describing these systems in terms of our model's components, we demonstrate how existing interactive visualization systems can be analyzed and understood within this framework, highlighting its descriptive power and flexibility across a range of visualization tools and interaction paradigms.

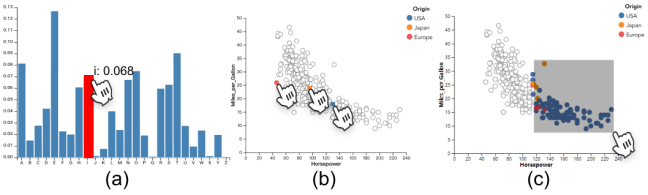
Generative power. As for its generative power, since all the interactive visualizations can be decomposed into our components, we can reuse, combine, and extend existing and new techniques easily and then refine each component independently; this ability gives a generative power to Libra, in contrast with traditional monolithic approaches. Hence, an interactive visualization can be improved by adding more existing interaction techniques as well as refining existing ones, with little or no change from the visualization side. For example, interactive lenses can be easily combined with Dust&Magnet to explore the data items in the dust layer. In doing so, experimenting with novel interaction techniques becomes easier and can be done starting with existing components, initially without changing them and later by refining them if necessary. It provides a constructive path to newer interactions, minimizing the effort to implement them and fostering experimentation.

6 EVALUATION OF `LIBRA.JS`

To demonstrate the effectiveness of `Libra.js`, we first show how it facilitates the implementation of interactions with use cases, and then quantitatively compare `Libra.js`-D3 and `Libra.js`-Vega with D3 and Vega in two aspects: i) the usability analysis of visualization notations with three metrics [39] and ii) the runtime performance.

6.1 Expressiveness

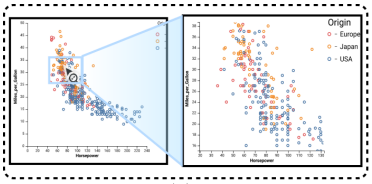
As described in section 4.6, `Libra.js` can be adapted to different visualization libraries; we implemented `Libra.js`-D3, `Libra.js`-Vega, and an experimental binding for Plot. In this section, we demonstrate the expressiveness of our prototype `Libra.js` by using




```
// Create a main layer on the existing SVG and use D3 to handle its group element
const svg = d3.select("svg");
const mainLayer = Libra.Layer.initialize('D3Layer',{
  name: 'mainLayer', container: svg.node()
})
const g = d3.select(mainLayer.getGraphic());
// Move existing visual specification codes into the main layer
g.selectAll("rect").data(data).join("rect")
 .attr("x", d=>d.x).attr("y", d=>d.y).attr("height", d=>d.frequency);
```

(d)

Figure 5: Examples of reusing built-in instruments: (a) hovering, (b) multiple clicking, and (c) brushing; and (d) a code snippet for gluing existing visualizations with `Libra.js`'s instrument.




```
Libra.Interaction.build({
 inherit: 'PanInstrument',
 layers: [mainLayer],
 sharedVar: {translateX,
             translateY}
})
Libra.Interaction.build({
 inherit: 'ZoomInstrument',
 layers: [mainLayer],
 sharedVar: {scaleX, scaleY}
})
```

(b)

```
const zoom = d3.zoom().on("zoom", zoomed);

svg.call(zoom);

function zoomed({transform}) {
  const zx = transform.rescaleX(x);
  const zy = transform.rescaleY(y);
  gx.call(xAxis, zx);
  gy.call(yAxis, zy);
  gGrid.call(grid, zx, zy);
}
```

(c)

```
{"data": ...,
 "params": [{
   "name": "grid",
   "select": "interval",
   "bind": "scales"
  }],
  "mark": "circle",
  "encoding": {
   "x":{"field":"Horsepow."},
   "y":{"field":"MpG"},
   "stroke":{"field":"Cyl."}
}}
```

(d)

Figure 6: Comparing the implementations of `Libra.js`, D3 and Vega-Lite for the panning & zooming. (a) The snapshots for zooming a scatterplot. (b) The `Libra.js` specification consists of the separated pan and zoom instruments, and (c,d) the D3 implementation and Vega-Lite specifications coupling pan and zoom operations.

`Libra.js`-D3 to create a variety of interactions for visualizations from the simplest to the most complex through reusing, combining, and extending a set of built-in instruments. All the examples, including visualizations created by Vega and Plot, are available on the accompanying website libra-js.github.io, and the correspondence between the examples and Yi et al's taxonomy of interaction techniques [71] is shown in the supplemental material.

Reusing Instruments. We provide six built-in instruments: *hover*, *click*, *brush*, *drag*, *pan*, *geometric zoom* and *semantic zoom* instruments; each of them has the corresponding interactor and selection service. Figure 5(a-c) shows three examples that can be created by reusing these instruments. To reuse these instruments, **AD** first wraps the existing visual specification code into the main layer, which only requires 5 lines of code (see lines 2-6 in Figure 5d). Taking the hover instrument as an example, **AD** attaches the built-in *hover* instrument to the main layer and sets the value of a specific


```
Libra.Interaction.build({
  inherit: 'HoverInstrument',
  layers: [mainLayer],
  // Insert the normalization service after
     the selection service
  insert: [{
    find: 'SelectionService',
    flow: [
      {comp:'NormalizeService'},
      {comp:'MainTransformer'}
    ]
  }],
  // Override the default transformer
  override: [{
    find: 'SelectionTransformer',
    comp: 'LineTransformer'
  }]
})
```

(b)

Figure 7: Example of composing instruments for implementing (a) the index chart and (b) corresponding code snippets for composing these new instruments with `Libra.js`.

shared variable to change the appearance of the selected items (e.g., "fill: red"). Note that the built-in transformer reads the datum from the selected element and displays the data information around the selected mark. Similarly, **AD** can specify the pan & zoom interactions [5] for a scatterplot in Figure 6a by binding the corresponding instruments to the main layer and setting the shared translation and scale variables, see `Libra.js` specification in Figure 6b. The D3 and Vega-Lite specifications shown in Figure 6c and 6d rely on the D3-zoom library [20] and the interval selection, which both encapsulate the pan and zoom operations together. Compared to `Libra.js`, they are limited in extensibility. For example, automatically enabling panning when the size of a canvas exceeds the viewport size, as in Kyrix [55], is not straightforward in D3. Achieving this functionality requires conditionally enabling event listeners and managing state coordination, which can be both complex and error-prone.

Likewise, Figure 1d shows a straightforward reuse of built-in instruments to implement a hover interaction that displays each data item's image. In contrast, using D3 or Vega requires **AD** to manually handle event callbacks (e.g., `mouseover` and `mouseout`) or define signals, resulting in more lines of code for a similar functionality.

Extending Instruments. **AD** can extend new instruments by reusing various built-in interactors and services. Figure 7 shows the index chart created in this way. Figure 7b illustrates how **AD** implements an Index Chart [52] shown in Figure 7a for comparing multiple series in a line chart with `Libra.js`. When the cursor hovers over the line chart, a vertical line appears, indicating the data value at that point on the x-axis, and all the rest of the data points are normalized by rescaling with the current value (see Figure 7a). This interaction can be achieved by inheriting the built-in hover instrument, as shown in line 2 of Figure 7b. To normalize the whole data with the selected data value, **AD** inserts a normalization service to chain with the selection service (lines 6-12). Rather than highlighting selected data elements, **AD** overrides the default selection transformer with a line transformer to indicate the cursor position with a line (lines 14-17). Similarly, **AD** extends the click instrument in Figure 1e with a filter service to highlight all the points of the same class.

Combining Instruments. **ID** can create new instruments by combining the built-in instruments with new interactors and services. Figure 1f shows an example that combines a drag instrument with a point hover instrument to enable interactive k-means clustering

on the MNIST dataset. Here, we further demonstrate the comnation of interactions with DimpVis.

DimpVis [38] is a direct manipulation technique for exploring time-series data shown in a scatterplot at one time-point. Hovering or touching any data points in the scatterplot reveals a time trajectory showing the evolution of the selected item through time. Dragging the selected item over the trajectory enables temporal navigation, where the scatterplot is updated to the data at the currently selected extent. The dragged position is snapped to the nearest position in the trajectory, requiring geometric computations in visual space. Figure 8a shows two snapshots of the scatterplot for interactive exploration of the Gapminder data [26], with each tuple consisting of five properties: fertility, life expectancy, country, region, and year in multiples of 5. Figure 8b shows the core `Libra.js` specification, and Figure 8c presents the corresponding architecture diagram, clarifying the input/output relationships among components and explaining how the behavior propagates from one component to another. **ID** puts all the data points of the scatterplot into the main layer and then uses the *interaction* API to create interactions. She first associates the built-in hover instrument (line 2) and the drag instrument (line 11) with the main layer (lines 3 and 12). To display the time trajectory when hovering or dragging a point, she first takes the scale information of the scatterplot as the global shared variables (line 4) for the selection service to find the selected data item. Then, she inserts a new flow consisting of a filter service and a trace transformer (lines 5-9), which are responsible for filtering the data in terms of the country property (lines 7-8), connecting the points of two properties (fertility and life expectancy) of one selected country with a line in temporal sequence (line 9) shown on the automatically generated trace layer rendered by `Libra.js`. For the drag instrument, she inserts an additional flow that connects the selection service to the nearest point service (lines 17-20) for finding the year nearest to the drag point in the trajectory. Here, the trace layer containing the time trajectory is referenced in a variable shared with the nearest point service. Then, she shares the nearest point with the interpolation service (lines 21-22) for calculating new interpolated quantitative properties (fertility, life expectancy, and year) of all data points. These interpolated properties are passed to the main transformer through the shared variables (line 23) to create smooth transitions between years. Note that the interpolated year is rounded to a multiple of 5. Figure 8c visualizes the relationship among these components, where the command of the interpolation service helps maintain the interaction history. To persist the interpolation result, the command is executed to update the interpolated data while re-drawing the layer consisting of all points after the interpolation is computed.

We further compare the implementations of `Libra.js` with D3 and Vega, with the major pieces of code provided by the DimpVis authors [19] and the Vega authors [59] shown in Figure 8d and Figure 8e. Since the Vega-Lite implementation does not fully support all required interactions, such as dragging a point through time, we do not consider it here. However, it is worth noting that `Libra.js` can enhance the Vega-Lite implementation to enable the full range of interactive functionality. Although the D3 implementation nicely structures the involved functions, it still requires **ID** to manually maintain the communication between different functions and the state of transient objects (e.g.,, time trajectory). Vega eases the implementation of callback functions but manages all selections in data space, whereas searching the nearest point in the time trajectory is natural in the visual space. Hence, its specification takes an approximate method based on the previous and next time points; they might yield incorrect nearest points in some cases.

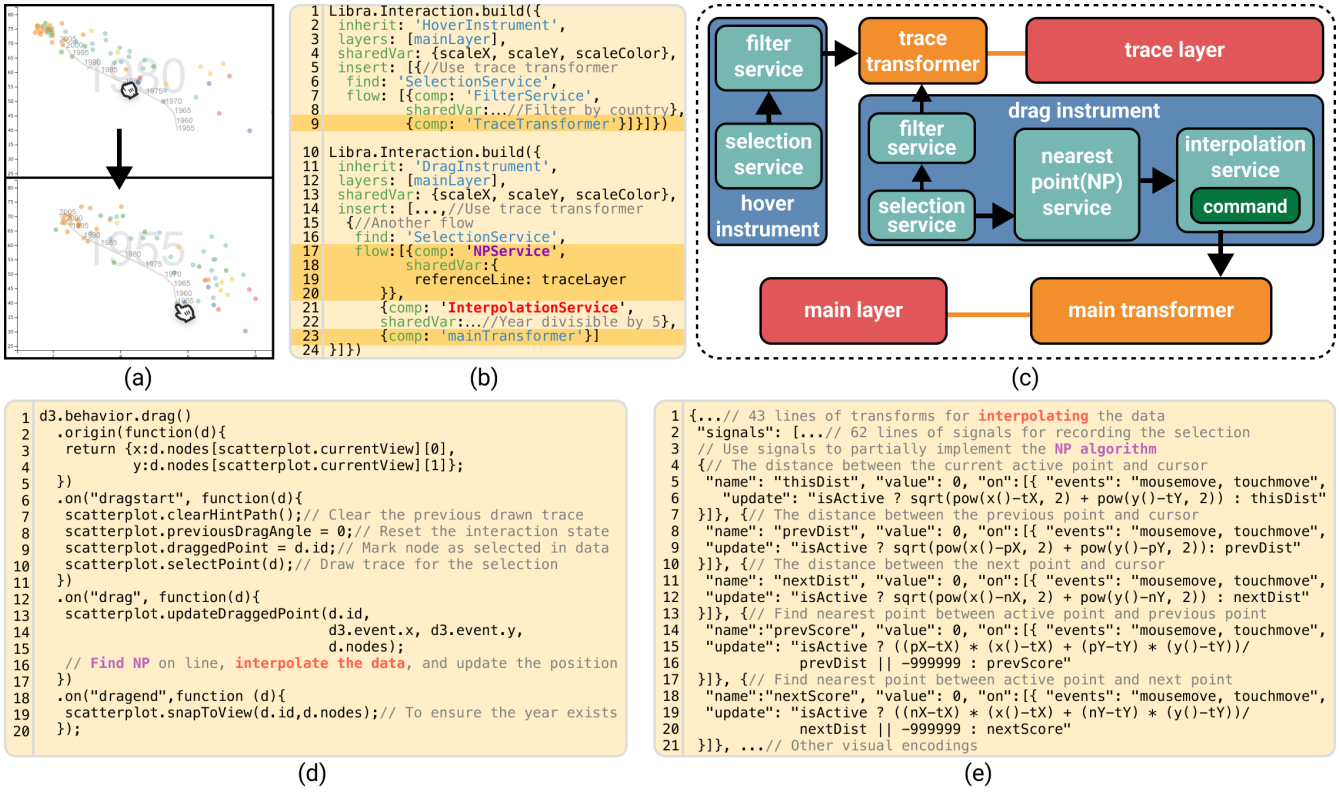
The `Libra.js`-D3 and `Libra.js`-Vega specifications both follow our interaction model Libra; however, they are not directly reusable across languages for two key reasons. First, the libraries use different data structures for mapping data to visual elements. For instance, in D3, data is stored in the "`data`" field, while in Vega, it is stored in the "`data.datum`" field. As a result, users must account for these differences when managing data mappings in the interaction-related layers. Second, if interactions involve changes to the visual elements in the main layer, the associated rendering procedures must be wrapped by `Libra.js`'s graphical transformers, which require certain code to be specified by the original libraries. In a similar fashion regarding the visualization techniques, Observable Plot and Vega rely on the GoG but are also incompatible with each other.

6.2 Metrics-Based Analysis

Rather than conducting a heuristic evaluation with the Cognitive Dimensions of Notations framework [7], as done in several other technical articles such as [51], we quantitatively assess three aspects of notation in a library: viscosity [28] (the difficulty of changing specifications), economy [35] (the number of elements and rules a user must remember when using the notation), and terseness [28] (the ability to express a lot in a small space). We employ three metrics proposed by Kruchten et al. [39], including sprawl (the median distance between all pairs of specifications), vocabulary size, and specification length.

To do so, we first choose the Wikipedia clickstream dataset [66] with good coverage of variable types to create a variety of static visualizations (e.g., bar charts, scatter plots, line charts, treemaps, and maps). Then, we follow the interaction taxonomy [71] to specify a few interaction techniques, including hovering, brushing, panning, and geometric zooming for all charts, semantic zooming and excentric labeling for scatter plots and treemaps, and some advanced ones such as index chart and Dimpvis. After implementing these examples with D3, `Libra.js`-D3, Vega, and `Libra.js`-Vega, we form 26 examples of interactive visualizations and then run the metric evaluation with the web-based tool NotaScope [39]. To alleviate the bias of the coding style, we run bootstrapping experiments for each metric and use Kernel Density Estimation (KDE) to draw the distribution. We did not consider Vega-Lite since it cannot specify interactions like excentric labeling and Dimpvis.

Figure 9a-c shows the results with three scatterplots of pairwise metrics computed from the specifications. We can see that `Libra.js`-D3 and `Libra.js`-Vega have lower sprawl than the original D3 and Vega, respectively, indicating that our interaction model largely reduces the cost of changing an interactive visualization to another one. Yet, `Libra.js`-D3 exhibits the longest specification length and the largest vocabulary size. We speculate that it is because `Libra.js` is not designed to be consistent with D3. Unlike




```
Libra.Interaction.build({
 inherit: 'HoverInstrument',
 layers: [mainLayer],
 sharedVar: {scaleX, scaleY, scaleColor},
 insert: [{//Use trace transformer
  find: 'SelectionService',
  flow: [{comp: 'FilterService',
          sharedVar:...//Filter by country},
         {comp: 'TraceTransformer'}]}]})

Libra.Interaction.build({
 inherit: 'DragInstrument',
 layers: [mainLayer],
 sharedVar: {scaleX, scaleY, scaleColor},
 insert: [...,//Use trace transformer
  {//Another flow
   find: 'SelectionService',
   flow:[{comp: 'NPService',
          sharedVar:{
           referenceLine: traceLayer
       }},
       {comp: 'InterpolationService',
       sharedVar:...//Year divisible by 5},
       {comp: 'mainTransformer'}]
}]})
```

```
d3.behavior.drag()
  .origin(function(d){
   return {x:d.nodes[scatterplot.currentView][0],
           y:d.nodes[scatterplot.currentView][1]};
  })
  .on("dragstart", function(d){
   scatterplot.clearHintPath();// Clear the previous drawn trace
   scatterplot.previousDragAngle = 0;// Reset the interaction state
   scatterplot.draggedPoint = d.id;// Mark node as selected in data
   scatterplot.selectPoint(d);// Draw trace for the selection
  })
  .on("drag", function(d){
   scatterplot.updateDraggedPoint(d.id,
                                  d3.event.x, d3.event.y,
                                  d.nodes);
   // Find NP on line, interpolate the data, and update the position
  })
  .on("dragend",function (d){
   scatterplot.snapToView(d.id,d.nodes);// To ensure the year exists
  });
```

```
{...// 43 lines of transforms for interpolating the data
 "signals": [...// 62 lines of signals for recording the selection
 // Use signals to partially implement the NP algorithm
 {// The distance between the current active point and cursor
  "name": "thisDist", "value": 0, "on":[{ "events": "mousemove, touchmove",
    "update": "isActive ? sqrt(pow(x()-tX, 2) + pow(y()-tY, 2)) : thisDist"
 }]}, {// The distance between the previous point and cursor
  "name": "prevDist", "value": 0, "on":[{ "events": "mousemove, touchmove",
  "update": "isActive ? sqrt(pow(x()-pX, 2) + pow(y()-pY, 2)): prevDist"
 }]}, {// The distance between the next point and cursor
  "name": "nextDist", "value": 0, "on":[{ "events": "mousemove, touchmove",
  "update": "isActive ? sqrt(pow(x()-nX, 2) + pow(y()-nY, 2)) : nextDist"
 }]}, {// Find nearest point between active point and previous point
  "name":"prevScore", "value": 0, "on":[{ "events": "mousemove, touchmove",
  "update": "isActive ? ((pX-tX) * (x()-tX) + (pY-tY) * (y()-tY))/
             prevDist || -999999 : prevScore"
 }]}, {// Find nearest point between active point and next point
  "name":"nextScore", "value": 0, "on":[{ "events": "mousemove, touchmove",
  "update": "isActive ? ((nX-tX) * (x()-tX) + (nY-tY) * (y()-tY))/
             nextDist || -999999 : nextScore"
 }]}, ...// Other visual encodings
```

Figure 8: Implementing the DimpVis interaction technique. (a) Two snapshots of this interaction; (b) Libra specification (custom components in orange); and (c) communication between all components, where the orange ones are defined by ID. (d) D3 code requiring ID to provide callback functions for all three events. (e) Vega specification for finding the point nearest to the point dragged on the time trajectory.

D3, Vega decouples low-level event processing from visual representation, and hence `Libra.js`-Vega replaces the signals and event streams used by Vega to specify interactions and reuse similar lines of gluing code as the one of the original Vega. However, Vega has large variations in specification length and vocabulary size. After checking the specification of each example, we found that Vega uses fewer lines of code for specifying simple interactions but requires long codes for complex interactions.

In contrast, `Libra.js`-Vega uses a consistent number of lines of code for the different interactions. Thus, it forms a compact distribution in three scatterplots with the smallest sprawl and reasonably small specification length and vocabulary size. We conclude that `Libra.js` enables easy changes to specifications, while `Libra.js`-Vega provides concise and learnable specifications for interaction.

To further learn the differences in the interaction specifications of `Libra.js`-D3 and `Libra.js`-Vega, we manually extracted the interaction components from the full specifications of each interactive visualization and then evaluated them using the NotaScope metrics. Here, we present only the scatterplots for the metrics sprawl and specification length, with additional metrics provided in the supplemental material. As shown in Figure 9d, `Libra.js`-D3 exhibits similar specification lengths to `Libra.js`-Vega but has slightly greater sprawl. Upon reviewing the specifications, we identified that the main difference arises from the sections involving the original library specifications (e.g.,, modifying the main layer). Specifically, `Libra.js`-D3 shows greater variation in defining both simple and complex static visualizations, while `Libra.js`-Vega demonstrates less variation due to its declarative abstraction.

6.3 Comparative Performance Benchmarks

We perform the benchmark study comparing the `Libra.js` version of D3 and Vega with the original D3 and Vega measured in terms of interactive frame rate. Following the configuration for evaluating Vega [51], we utilized the same three examples: brushing & linking a scatterplot matrix, a time-series overview+detail visualization, and panning & zooming a scatterplot, where both D3 and Vega implementations are available. For each example, we use the Cypress tool [73] to perform automated testing with datasets ranging from 100 to 100,000 tuples, conducting 50 trials per size. To mitigate the influence of browser-based just-in-time (JIT) optimizations, we launch a fresh Chrome browser instance for each test. All tests were performed on a system equipped with an Intel i3 8400, NVIDIA GeForce GTX 1080 Ti, 32GB RAM, and Windows 11, using Chrome 122.0.6261.69. The versions of D3 and Vega are 7.8.5 and 5.27.0, respectively.

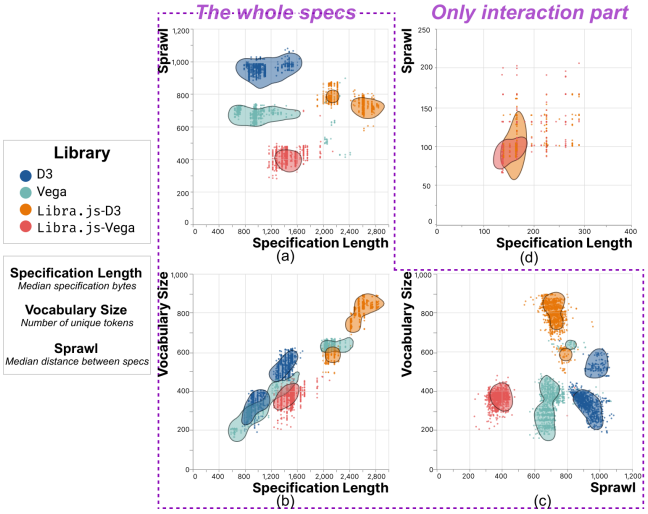


Figure 9: We run 1k bootstrapped variations for each metric based on our gallery, with (a-c) and without the static visualization code (d), and show the results via a scatter plot with a Kernel Density Estimation (KDE) of each pair of metrics: specification length, vocabulary size, and sprawl. Each scatter represents the median value of one variation, while the shaded regions represent the areas containing 75% of the probability mass of each library's KDE distribution.

Figure 10 presents the average frame rates for three interactive visualizations, where `Libra.js`-D3, `Libra.js`-Vega, and Vega consistently outperform D3. This aligns with the findings reported by Satyanarayan et al. [51]. In the examples of brushing & linking a scatterplot matrix and the time-series overview+detail visualization shown in Figure 10a and Figure 10b, `Libra.js`-D3 and `Libra.js`-Vega demonstrate superior interactive performance compared to D3 and Vega. As the number of data items increases, `Libra.js`-D3 and `Libra.js`-Vega are significantly faster than D3 and Vega. We speculate that this advantage might be caused by different redrawing strategies, where `Libra.js` only draws the selected elements on the selection layer. D3 requires redrawing the entire visualization, resulting in the worst performance, while Vega also redraws the subset of all data tuples affected by interactions. In Figure 10c, `Libra.js`-D3 and `Libra.js`-Vega perform similarly to D3 but slightly worse than Vega. We speculate that this difference is their direct access to view transforms, and we will profile them to understand why. Overall, `Libra.js`'s performances without deeper optimizations are competitive with the other libraries. In the future, we aim to explore these optimizations and integrate them into `Libra.js`.

7 CONCLUSION, LIMITATIONS & FUTURE WORK

We introduce an interaction model that supports the creation, reuse, extension, and combination of rich interaction techniques for data visualization. Building on and extending previous work in HCI and visualization, our model, Libra, incorporates key concepts such as layers, instruments, feedback, and feedforward. By managing interactions independently of visual representations, Libra provides a comprehensive abstract framework. This separation of concerns overcomes a key limitation in existing systems, where interactions are often tightly coupled with visualization rendering, limiting their adaptability and reuse.

To demonstrate the expressiveness and flexibility of our model, we present `Libra.js`, a prototype that supports various interactions for visualizations through a declarative syntax. `Libra.js` allows concise association of a diverse set of instruments with visual elements for simple interactions while remaining expressive enough to describe most advanced interactions found in the literature.

7.1 Limitations

Libra is designed for 2D visualizations and does not take into account 3D, VR, or AR, where the layering concept might be insufficient to clearly separate data representation from interaction elements. While we have primarily focused on pointer- and keyboard-based interactions, we have also experimented with Libra using more advanced non-standard and virtual input devices (e.g.,, voice-based interactions). Libra can be extended to support them through appropriate interactors, though some devices might require deeper modifications, particularly for handling multimodal interactions such as "put that there"-style commands [17].

Yet, our article review (section 5) shows that most research focus on 2D visualizations. We are confident that a large portion of the advanced interactive 2D visualizations from the review could be reimplemented using Libra, often in a more compact form, as demonstrated in the metric-based evaluation of `Libra.js` (section 6.2).

Learning cost of `Libra.js`. While our metric-based evaluation focuses on specification length, vocabulary size, and sprawl, we recognize the importance of other critical factors, particularly learnability. Similar to learning D3—with its power and associated complexity—there is also a learning curve for developers to fully grasp our model and `Libra.js`. Rather than developing additional metrics to justify this, we prioritize creating more examples—both to explore advanced visualization interactions and to help users better understand our model. While this may add some complexity, it ultimately enhances the landscape of visualization interactions, reinforcing the core advantage of our model: treating interactions as first-class citizens in visualization systems.

Implementation of Interactors. Our model does not specify how interactors should be implemented or specified. Many formalisms exist in the HCI community [44] with no clear consensus on which one is best; we believe that many of them can be used to implement interactors in Libra and that library designers should choose the one they consider the most suited.

Use of Layers for Selection. At a more concrete level, our model assumes that interaction effects occur on the transient layer or selection layer, which might not be true in some cases. For example, brushing scatterplots often de-emphasize the unselected points by using dimmed colors while keeping the color of the selected points. In our model, layers would be a natural mechanism to stack a visualization with the unfiltered visual elements under a filtered colored version, and the instrument could be either a brush-able

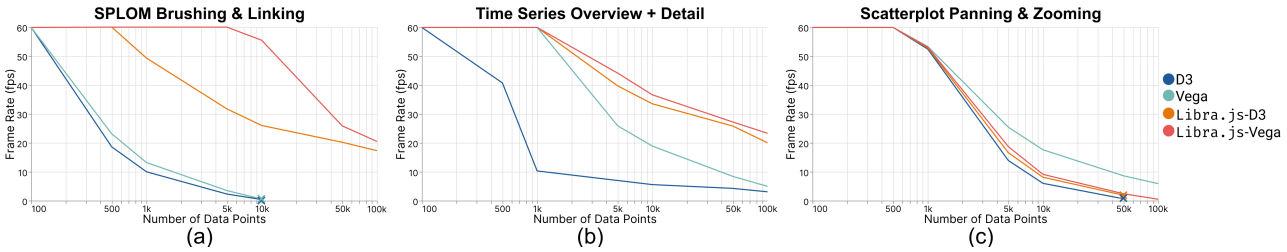


Figure 10: Average frame rates for three interactive visualizations (higher means faster): where (a) brushing and linking on a scatterplot matrix; (b) brushing and linking on an overview+detail visualization; and (c) panning and zooming on a scatterplot.

scatterplot or a range slider [40]. It would require an alternative interaction policy within Libra.

In the future, we plan to experiment with this filtering style that needs to dynamically adjust the drawing order of layers.

7.2 Future Work

`Libra.js` is designed to be highly extensible, allowing it to manage interactions across a wide range of devices. By handling new event types at the interactor level, `Libra.js` can accommodate emerging interaction modalities, such as speech, gaze, or gesture inputs from mobile, wearable, or immersive devices. This adaptability ensures that `Libra.js` evolves with changing user interaction paradigms.

In addition, `Libra.js` can be extended to support synchronous collaboration by incorporating features like real-time feedback on users' cursors and viewports. This functionality can be implemented in an extra layer, similar to the approach used in CocoNutTrix [33], enabling users to collaborate and interact seamlessly within the same environment.

Building on the k-means service we described, we aim to develop additional machine learning-driven services to enhance the expressiveness of our instruments, leveraging the scikit-learn toolkit [47].

ACKNOWLEDGMENTS

This work is supported by the grants of the National Key R&D Program of China under Grant 2022ZD0160805, NSFC (No.62132017 and No.U2436209), the Shandong Provincial Natural Science Foundation (No.ZQ2022JQ32), the Beijing Natural Science Foundation (L247027), the Fundamental Research Funds for the Central Universities, and the Research Funds of Renmin University of China. The authors thank Andrew M. McNutt and Oliver Deussen for their valuable suggestions.

REFERENCES

[1] Harold Abelson and Gerald J. Sussman. 1996. *Structure and Interpretation of Computer Programs* (2nd ed.). MIT Press, Cambridge, MA, USA.

[2] Maria-Florina Balcan, Avrim Blum, and Anupam Gupta. 2009. Approximate clustering without the approximation. In *Proceedings of the twentieth annual ACM-SIAM symposium on Discrete algorithms*. SIAM, 1068–1077.

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

[4] Michel Beaudouin-Lafon. 2004. Designing interaction, not interfaces. In *Proceedings of the working conference on Advanced visual interfaces*. ACM, Gallipoli Italy, 15–22. https://doi.org/10.1145/989863.989865

[5] Benjamin B Bederson. 2001. PhotoMesa: a zoomable image browser using quantum treemaps and bubblemaps. In *Proceedings of the 14th annual ACM symposium on User interface software and technology*. 71–80.

[6] Enrico Bertini, Maurizio Rigamonti, and Denis Lalanne. 2009. Extended Excentric Labeling. *Computer Graphics Forum* 28, 3 (2009), 927–934. https://doi.org/10.1111/j.1467-8659.2009.01456.x arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1111/j.1467-8659.2009.01456.x

[7] Alan F Blackwell, Carol Britton, Anna Cox, Thomas RG Green, Corin Gurr, Gada Kadoda, Maria S Kutar, Martin Loomes, Chrystopher L Nehaniv, Marian Petre, et al. 2001. Cognitive dimensions of notations: Design tools for cognitive technology. In *International conference on cognitive technology*. Springer, 325–341.

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

[9] Michael Bostock, Vadim Ogievetsky, and Jeffrey Heer. 2011. D^3 Data-Driven Documents. *IEEE Trans. Vis. Comput. Graphics* 17, 12 (Dec. 2011), 2301–2309. https://doi.org/10.1109/TVCG.2011.185

[10] Steven P. Callahan, Juliana Freire, Emanuele Santos, Carlos Eduardo Scheidegger, Cláudio T. Silva, and Huy T. Vo. 2006. VisTrails: visualization meets data management. In *ACM SIGMOD Conference*. 745–747.

[11] Stuart K. Card, Jock D. Mackinlay, and Ben Shneiderman. 1999. *Readings in Information Visualization: Using Vision to Think*. Morgan Kaufmann.

[12] Surajit Chaudhuri, Bolin Ding, and Srikanth Kandula. 2017. Approximate query processing: No silver bullet. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 511–519.

[13] Ed Huai-hsin Chi and John T. Riedl. 1998. An operator interaction framework for visualization systems. In *IEEE Symposium on Information Visualization*. 63–70.

[14] Hank Childs, Berk Geveci, Will Schroeder, Jeremy Meredith, Kenneth Moreland, Christopher Sewell, Torsten Kuhlen, and E. Wes Bethel. 2013. Research challenges for visualization software. *Computer* 46, 5 (2013), 34–42. https://doi.org/10.1109/mc.2013.179

[15] Jungu Choi, Deok Gun Park, Yuet Ling Wong, Eli Fisher, and Niklas Elmqvist. 2015. Visdock: A toolkit for cross-cutting interactions in visualization. *IEEE Trans. Vis. Comput. Graphics* 21, 9 (2015), 1087–1100. https://doi.org/10.1109/tvcg.2015.2414454

[16] Gregory H. Cooper and Shriram Krishnamurthi. 2006. Embedding dynamic dataflow in a call-by-value language. In *European symposium on programming*. Springer, 294–308. https://doi.org/10.1007/11693024_20

[17] James L. Crowley. 2018. Put That There: 20 Years of Research on Multimodal Interaction. In *Proceedings of the 2018 on International Conference on Multimodal Interaction, ICMI 2018, Boulder, CO, USA, October 16-20, 2018*, Sidney K. D'Mello, Panayiotis G. Georgiou, Stefan Scherer, Emily Mower Provost, Mohammad Soleymani, and Marcelo Worsley (Eds.). ACM, 4. https://doi.org/10.1145/3242969.3276309

[18] Zach Cutler, Kiran Gadhave, and Alexander Lex. 2020. 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*. IEEE, 116–120. https://doi.org/10.1109/VIS47514.2020.00030

[19] d3dimpvis 2013. DimpVis: Prototyping for direct interaction techniques with information visualizations. https://github.com/vialab/dimpVis. Accessed: 2023-03-14.

[20] d3zoom 2016. D3-zoom. https://github.com/d3/d3-zoom. Accessed: 2023-03-14.

[21] Evanthia Dimara and Charles Perin. 2019. What is interaction for data visualization? *IEEE Trans. Vis. Comput. Graphics* 26, 1 (2019), 119–129. https://doi.org/10.1109/tvcg.2019.2934283

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

[23] Jean-Daniel Fekete and Michel Beaudouin-Lafon. 1996. Using the Multi-Layer Model for Building Interactive Graphical Applications. In *Proceedings of the 9th Annual ACM Symposium on User Interface Software and Technology*. ACM, 109–118. https://doi.org/10.1145/237091.237108

[24] Jean-Daniel Fekete, Danyel Fisher, and Michael Sedlmair. 2024. *Progressive Data Analysis: Roadmap and Research Agenda*. Eurographics. 231 pages. https://doi.org/10.2312/pda.20242707

[25] Jean-Daniel Fekete and Catherine Plaisant. 1999. Excentric Labeling: Dynamic Neighborhood Labeling for Data Visualization. In *Proc. SIGCHI Conf. Human Factors Comp. Sys.* 512–519. https://doi.org/10.1145/302979.303148

[26] Gapminder Foundation. [n. d.]. Gapminder trendalyzer. http://www.gapminder.org. http://www.gapminder.org

[27] Cory Gackenheimer. 2015. *Introduction to React.* Apress.

[28] Thomas RG Green. 1989. Cognitive dimensions of notations. *People and computers V* (1989), 443–460.

[29] Derek Greer. [n. d.]. Interactive Application Architecture Patterns. http://aspiringcraftsman.com/2007/08/25/interactive-application-architecture/. Accessed: 2023-02-23.

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

[31] Jeffrey Heer, Stuart K. Card, and James A. Landay. 2005. Prefuse: a toolkit for interactive information visualization. In *Proc. SIGCHI Conf. Human Factors Comp. Sys.* 421–430. https://doi.org/10.1145/1054972.1055031

[32] Jane Hoffswell, Arvind Satyanarayan, and Jeffrey Heer. 2016. Visual debugging techniques for reactive data visualization. *Computer Graphics Forum* 35, 3 (2016), 271–280.

[33] Petra Isenberg, Anastasia Bezerianos, Nathalie Henry, Sheelagh Carpendale, and Jean-Daniel Fekete. 2009. CoCoNutTrix: Collaborative Retrofitting for Information Visualization. *IEEE Comput. Graph. Appl. Mag* 29, 5 (2009), 44–57. https://doi.org/10.1109/MCG.2009.78

[34] Petra Isenberg, Florian Heimerl, Steffen Koch, Tobias Isenberg, Panpan Xu, Chad Stolper, Michael Sedlmair, Jian Chen, Torsten Möller, and John Stasko. 2017. vispubdata.org: A Metadata Collection about IEEE Visualization (VIS) Publications. *IEEE Transactions on Visualization and Computer Graphics* 23, 9 (Sept. 2017), 2199–2206. https://doi.org/10.1109/TVCG.2016.2615308

[35] Kenneth E Iverson. 1980. Notation as a tool of thought. *Commun. ACM* 23, 8 (1980), 444–465.

[36] T.j. Jankun-Kelly, Kwan-liu Ma, and Michael Gertz. 2007. A Model and Framework for Visualization Exploration. *IEEE Trans. Vis. Comput. Graphics* 13, 2 (2007), 357–369. https://doi.org/10.1109/TVCG.2007.28

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

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

[39] Nicolas Kruchten, Andrew M McNutt, and Michael J McGuffin. 2023. Metrics-Based Evaluation and Comparison of Visualization Notations. *IEEE Transactions on Visualization and Computer Graphics* (2023).

[40] Qing Li, Xiaofeng Bao, Chen Song, Jinfei Zhang, and Chris North. 2003. Dynamic query sliders vs. brushing histograms. In *Extended abstracts of the 2003 Conference on Human Factors in Computing Systems, CHI 2003, Ft. Lauderdale, Florida, USA, April 5-10, 2003*, Gilbert Cockton and Panu Korhonen (Eds.). ACM, 834–835. https://doi.org/10.1145/765891.766020

[41] M Douglas McIlroy, J Buxton, Peter Naur, and Brian Randell. 1968. Mass-produced software components. In *Proceedings of the 1st international conference on software engineering, Garmisch Pattenkirchen, Germany.* 88–98.

[42] Brad A. Myers. 1990. A new model for handling input. *ACM Trans. Inf. Syst.* 8, 3 (1990), 289–320. https://doi.org/10.1145/98188.98204

[43] Brad A. Myers. 1991. Separating application code from toolkits: eliminating the spaghetti of call-backs. In *Proceedings of the 4th Annual ACM Symposium on User Interface Software and Technology*, James R. Rhyne (Ed.). 211–220. https://doi.org/10.1145/120782.120805

[44] Brad A. Myers. 2024. *Pick, Click, Flick!: The Story of Interaction Techniques.* ACM Books, Vol. 57. ACM. https://doi.org/10.1145/3617448

[45] Donald A Norman. 1999. Affordance, conventions, and design. *interactions* 6, 3 (1999), 38–43.

[46] observable-plot [n. d.]. Observable Plot | The JavaScript library for exploratory data visualization. https://observablehq.com/plot/. Accessed: 2024-02-23.

[47] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830. https://doi.org/10.5555/1953048.2078195

[48] Mike Potel. 1996. MVP: Model-View-Presenter – The Taligent Programming Model for C++ and Java. http://www.wildcrest.com/Potel/Portfolio/mvp.pdf. Access: 2023-02-23.

[49] Trygve Mikjel H. Reenskaug. 1979. The original MVC reports.

[50] Arvind Satyanarayan, Dominik Moritz, Kanit Wongsuphasawat, and Jeffrey Heer. 2017. Vega-lite: A grammar of interactive graphics. *IEEE Trans. Vis. Comput. Graphics* 23, 1 (2017), 341–350. https://doi.org/10.1109/tvcg.2016.2599030

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

[52] Arvind Satyanarayan, Kanit Wongsuphasawat, and Jeffrey Heer. 2014. Declarative interaction design for data visualization. In *Proceedings of the 27th Annual ACM Symposium on User Interface Software and Technology*. 669–678. https://doi.org/10.1145/2642918.2647360

[53] Shneiderman. 1983. Direct Manipulation: A Step Beyond Programming Languages. *Computer* 16, 8 (1983), 57–69. https://doi.org/10.1109/MC.1983.1654471

[54] Luke S. Snyder and Jeffrey Heer. 2024. DIVI: Dynamically Interactive Visualization. *IEEE Trans. Vis. Comput. Graph.* 30, 1 (2024), 403–413. https://doi.org/10.1109/TVCG.2023.3327172

[55] Wenbo Tao, Xiaoyu Liu, Yedi Wang, Leilani Battle, Çağatay Demiralp, Remco Chang, and Michael Stonebraker. 2019. Kyrix: Interactive pan/zoom visualizations at scale. *Computer Graphics Forum* 38, 3 (2019), 529–540.

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

[57] Christian Tominski, Stefan Gladisch, Ulrike Kister, Raimund Dachselt, and Heidrun Schumann. 2017. Interactive lenses for visualization: An extended survey. *Computer Graphics Forum* 36, 6 (2017), 173–200. https://doi.org/10.1111/cgf.12871

[58] Laurens van der Maaten and Geoffrey Hinton. 2008. Visualizing Data using t-SNE. *Journal of Machine Learning Research* 9, 86 (2008), 2579–2605. http://jmlr.org/papers/v9/vandermaaten08a.html

[59] vegadimpvis 2018. Global Development Example. https://vega.github.io/vega/examples/global-development/. Accessed: 2023-03-14.

[60] Allan Vermeulen, Gabe Beged-Dov, and Patrick Thompson. 1995. The pipeline design pattern. In *Proceedings of OOPSLA'95 Workshop on Design Patterns for Concurrent, Parallel, and Distributed Object-Oriented Systems*. Citeseer.

[61] John M. Vlissides and Mark A. Linton. 1990. Unidraw: A Framework for Building Domain-Specific Graphical Editors. *ACM Trans. Inf. Syst.* 8, 3 (July 1990), 237–268. https://doi.org/10.1145/98188.98197

[62] Zhanyong Wan, Walid Taha, and Paul Hudak. 2002. Event-driven FRP. In *International Symposium on Practical Aspects of Declarative Languages*. Springer, 155–172.

[63] Yunhai Wang, Yanyan Wang, Haifeng Zhang, Yinqi Sun, Chi-Wing Fu, Michael Sedlmair, Baoquan Chen, and Oliver Deussen. 2018. Structure-aware fisheye views for efficient large graph exploration. *IEEE transactions on visualization and computer graphics* 25, 1 (2018), 566–575.

[64] Chris Weaver. 2004. Building Highly-Coordinated Visualizations in Improvise. In *IEEE Symposium on Information Visualization*. IEEE, 159–166. https://doi.org/10.1109/INFVIS.2004.12

[65] Hadley Wickham. 2016. *ggplot2: Elegant Graphics for Data Analysis* (2nd ed.). Springer Publishing Company, Incorporated.

[66] Wikimedia Foundation. 2017. Wikipedia Clickstream. https://meta.wikimedia.org/wiki/Research:Wikipedia_clickstream. Accessed: 2024-03-15.

[67] Leland Wilkinson. 2012. The grammar of graphics. In *Handbook of computational statistics*. Springer, 375–414.

[68] Nelson Wong, Sheelagh Carpendale, and Saul Greenberg. 2003. EdgeLens: An Interactive Method for Managing Edge Congestion in Graphs. In *IEEE Symposium on Information Visualization*. 51–58. https://doi.org/10.1109/INFVIS.2003.1249008

[69] Michael Wybrow, Niklas Elmqvist, Jean-Daniel Fekete, Tatiana Von Landesberger, Jarke J van Wijk, and Björn Zimmer. 2014. Interaction in the visualization of multivariate networks. In *Multivariate Network Visualization: Dagstuhl Seminar# 13201, Dagstuhl Castle, Germany, May 12-17, 2013, Revised Discussions*. Springer, 97–125.

[70] Chhavi Yadav and Léon Bottou. 2019. Cold Case: The Lost MNIST Digits. *CoRR* abs/1905.10498 (May 2019). arXiv:1905.10498 https://arxiv.org/abs/1905.10498

[71] Ji Soo Yi, Youn ah Kang, John Stasko, and Julie A. Jacko. 2007. Toward a deeper understanding of the role of interaction in information visualization. *IEEE Trans. Vis. Comput. Graphics* 13, 6 (2007), 1224–1231. https://doi.org/10.1109/tvcg.2007.70515

[72] Ji Soo Yi, Rachel Melton, John T. Stasko, and Julie A. Jacko. 2005. Dust & Magnet: multivariate information visualization using a magnet metaphor. *Information Visualization* 4, 3 (2005), 239–256. https://doi.org/10.1057/palgrave.ivs.9500099

[73] Jidong Zhai, Jianfei Hu, Xiongchao Tang, Xiaosong Ma, and Wenguang Chen. 2014. CYPRESS: Combining Static and Dynamic Analysis for Top-Down Communication Trace Compression. In *International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2014, New Orleans, LA, USA, November 16-21, 2014*, Trish Damkroger and Jack J. Dongarra (Eds.). IEEE Computer Society, 143–153. https://doi.org/10.1109/SC.2014.17