

C4D3: A View-Level Abstraction and Coordination Library for Building Coordinated Multiple Views with D3

Omkar S. Chekuri^a and Chris Weaver^b

The University of Oklahoma, Norman, Oklahoma, U.S.A.

Keywords: Coordinated and Multiple Views, Visualization Systems and Tools, Visualization Toolkits.

Abstract: D3 is a popular and effective library for the development and deployment of visualizations in web pages. Numerous applications testify to its accessibility and expressiveness for representing and manipulating page content. By eschewing toolkit-specific abstraction, D3 gains representational transparency, but at a cost of modularity. Relatively few D3 applications compose multiple views or support coordinated interaction beyond basic navigation and brushing. Rather than relegate view composition to custom integration code, we overlay D3 with a view-level abstraction that utilizes a general parameter sharing model to offer simple yet flexible composition of coordinated multiple views (CMV) while preserving the expressiveness of individual D3 components. Coupling of event handling to shared parameters recasts modeling of interactive state and simplifies the declarative specification of interactive dependencies between views. We present an example of an extensively coordinated visualization, illustrative code to show CMV construction in C4D3, and demonstrate how view-level abstraction can reduce the code needed to compose complex D3 visualizations.

1 INTRODUCTION


Web-based interactive visualizations are increasingly used for data exploration and analysis. While a single interactive view is often sufficient to present one perspective on a data set, coordinated multiple views (CMV) with interactive capabilities are usually required to support multifaceted perspectives for comprehensive analysis. Yet, it remains challenging to build CMV in popular low-level web-based libraries due to their lack of coordination abstractions (Chen et al., 2022). Higher-level design tools like Tableau and Power BI are conversely limited to predefined sets of relatively simple coordination templates.


Data-Driven Documents (D3) has emerged as a widely adopted library for embedding visualizations within web pages (Bostock et al., 2011). D3 offers an accessible, expressive, declarative language for representing and transforming data in the Document Object Model (DOM) of a web page. Furthermore, it enables manipulation of that data by associating interaction events with transformations, thereby supporting a wide variety of interaction visualization designs.

The tendency in web-based visualization design

is to favor simpler designs suited for communication and explanation. Composite designs, like those with CMV constructions commonly utilized in exploration and analytical applications, are relatively rare. This is not exclusive to visualizations built using D3. Visualizations built using other libraries and toolkits (Fekete, 2004; Bostock and Heer, 2009; Satyanarayan et al., 2014; Satyanarayan et al., 2017) exhibit the same tendency. This scarcity can generally be attributed to limitations in language and feature support, as well as to the inherently more complex nature of designs comprising multiple views.

The scarcity of complex CMV constructions using D3 may arise from its goal to avoid imposing a toolkit-specific abstraction of visual representation. Unlike many visualization libraries, D3 does not explicitly define the concept of a “view”; rather, it leaves the definition of a view largely to developer discretion. This flexibility allows D3 to transform page structure and styling in ways that may not align with commonly recognized view types. However, D3’s constraints on interaction handling and encapsulation of event processing make it challenging to combine different interaction channels that rely on the same events, especially when conflicting events occur (Satyanarayan et al., 2017) (such as how D3 relies on mouse click and drag for both brushing and

^a  <https://orcid.org/0009-0002-7197-7954>

^b  <https://orcid.org/0000-0002-6713-093X>

panning). Composite visualizations typically involve not only multiple views, but also multiple coordinated interactions between those views to support diverse manipulations of data and how it appears. As the number of views and the complexity of interactions increase, so does the challenge of managing dependencies between D3 fragments corresponding to each “view” and its associated portion of the DOM. Vega-lite (Satyanarayan et al., 2017) and the interaction semantics introduced in Reactive Vega (Satyanarayan et al., 2014; Satyanarayan et al., 2016) seek to address these concerns, but do not directly address the issue of composability of CMV in visualization designs. Selection parameters in Vega-Lite convert user interactions into data queries that share state between views. However, these selection interactions are restricted to point and intervals, limiting expressiveness when trying to design custom selection interactions, such as selection of arbitrary data points in views, or when implementing non-selection-based complex data queries that depend on navigation.

C4D3 contributes a view encapsulation and interaction abstraction library implemented on top of D3. It aims to support expressive design and elegant specification of coordinated multiple views by adopting the three main design principles of the Live Properties architecture in the *Improvise* visualization system (Weaver, 2004). First, modular view implementation facilitates the design and reuse of common and custom view types in CMV constructions. Second, clean separation of view implementation and coordination design makes building and modifying complex CMV constructions simpler and easier than with monolithic approaches. Third, lifting the storage and management of interaction state from view-local to visualization-global, while keeping the interpretation of state and changes to it view-centric, facilitates flexible and meaningful sharing of state to implement expressive and understandable coordination designs. We followed these principles to develop C4D3 with the objectives of increasing transparency, extensibility, modularity, composability, consistency, and consonance of web-based CMV visualization design.

In this paper, we present C4D3’s coordination architecture based on Live Properties, a JavaScript implementation of that architecture, the design and implementation of a view wrapper to adapt D3 rendering and interaction handling code for coordination, and several common view types implemented as D3 code within the wrapper. We demonstrate C4D3’s capabilities via a reproduction of a classic CMV visualization with extensive navigation and selection coordinations, illustrate the process of implementing a simple CMV design with C4D3, and assess our progress toward

achieving our objectives for C4D3. We conclude that C4D3 can support an incrementally expanding collection of D3 components as view modules that can be quickly and easily instantiated and interconnected to implement expressive CMV patterns. While implementing particular views for use in a multiple view visualization design in C4D3 does usually require writing at least some custom D3 code for each view, the effort is similar to writing different components in D3 without C4D3, and the modularity of the C4D3 view wrapper amortizes the effort needed by allowing substantial code reuse between similar views, much like how D3 designers often modify existing D3 examples rather than start from scratch.

Supplementary material and the code for the C4D3 examples in this paper can be found online at <https://omkarchekuri.github.io/c4d3/>.

2 RELATED WORK

C4D3 introduces a view abstraction and a general purpose coordination layer adapted from the *Improvise* architecture (Weaver, 2004) for specifying and building coordinated visualizations as data flow graphs. As an implemented library, C4D3 provides coordination capabilities that build on the existing visual representation capabilities of D3, thus applying a separation of concerns between D3’s visualization rendering code and C4D3’s coordination management. This separation allows for swapping out D3 code with other libraries or combining visual representations from multiple libraries to compose a CMV visualization.

Research on coupled interaction have been pivotal in the evolution of information visualization. Constraint-based UIs like *Garnet* (Myers et al., 1990) and *Rendezvous* (Hill et al., 1994), and seminal visualization systems such as *Tioga-2* (Aiken et al., 1996), *Visage* (Roth et al., 1996) *Spotfire* (Ahlberg, 1996), and *XGobi* (Buja et al., 1996), heavily influenced work on CMV systems including *IVEE* (Ahlberg and Wistrand, 1995), *DEVise* (Livny et al., 1997), *Snap-Together Visualization* (North and Shneiderman, 2000), *GeoVISTA Studio* (Takatsuka and Gahegan, 2002), *CViews* (Boukhelifa and Rodgers, 2003), and *Improvise* (Weaver, 2004). This long history underscores the persistent challenge of balancing design accessibility and expressiveness and bridging capabilities of visualization systems (Fekete et al., 2011).

More recently, *Nebula* (Chen et al., 2022) provides an abstraction layer for constructing coordinated visualizations using macros to abstract coordination specifications. It offers efficient CMV design through textual specification of supported coordina-

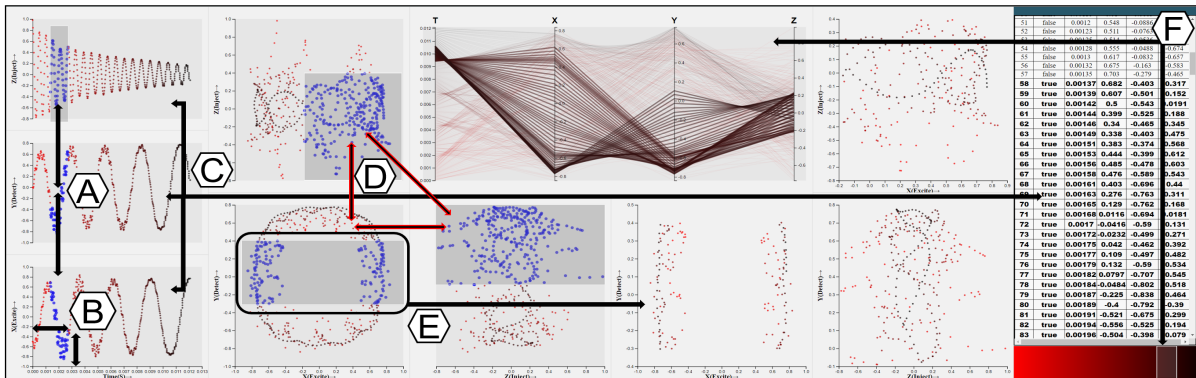


Figure 1: Reproduction of various coordinations in the Improvise cubic ion trap visualization (Weaver, 2004) with 12 C4D3 views: (A) three time series plots synchronize scrolling horizontally; (B) independent vertical scrolling in those plots; (C) shared selection between scatter plots and the table; (D) overview SPLOM showing three sides of the cube; (E) detail SPLOM zoomed to ranges selected in the overview; (F) a selected range in a color slider filters items in a parallel coordinate plot.

tion patterns. The extensibility of coordinations and implementation of complex coordination patterns becomes challenging when users are restricted to the existing natural language patterns backed by higher-level language parsers. Text specification allows one-way binding from the source visualization that accepts the interaction to the destination that shows the result. This makes specification accessible and more natural, but limits design expressiveness. The architecture is also limited to interactions that trigger updates only upon completion. Nebula is similar to C4D3 in how it separates coordination from visual representation, despite their substantial differences in coordination specification and expressiveness.

User interface frameworks like React offer a component-based architecture in which UI elements are treated as self-contained components organized in a hierarchy. React uses a combination of strategies to coordinate view components in the component hierarchy. It uses props, callbacks, and context to coordinate the parent component with the child component. Additionally, libraries like Redux offer a pattern for organizing global state and ensuring predictable state management for JavaScript applications built on React. Although these libraries together support state management for building views (as UI components) that can be coordinated using shared state, they do not provide an abstraction layer for specifying coordination between views. They rely on developers to implement custom logic to manage complex view interactions and state changes across coordinated views.

CViews (Boukhelifa and Rodgers, 2003) offers a coordination model in which objects in a coordination space are directly linked to views through translation functions. Use-Coordination (Keller et al., 2024) provides a reactive library based on this coordination model, implemented in the *ReactJS* library. One

shortcoming of the coordination abstraction layer in this model is that views are directly dependent on coordination objects; the translation functions tightly couple coordination objects to view implementations. In contrast, Live Properties (Weaver, 2004) provides a level of indirection such that coordination objects are not directly coupled to views but instead are linked to properties of the views. Coordination objects can provide information of any type for views to share and translate for individual use. This indirection expands coordination to allow sharing not only of interaction characteristics, but also of behavior and appearance, thereby providing an abstraction layer that is flexible enough to express a wide variety of view couplings far beyond those usually encountered.

C4D3 builds on D3’s capabilities to make coordination capabilities accessible to the wide community of D3 users with implementation cost at or below that of D3 on its own. It provides flexible coordination between views by abstracting the coordination from the visual representation and interaction with it, while still maintaining the flexibility and expressiveness to author open-ended coordination patterns. It also optimizes view updates in response to interaction by only updating the affected parts of the DOM, thereby allowing intermediate updates during protracted interactions and with it a sense of interactive immediacy (depending on the latency of the updates themselves).

3 EXAMPLE

Figure 1 shows a reproduction of the Improvise cubic ion trap visualization comprising 12 coordinated views created with C4D3. The C4D3 library provides a variety of view modules, each tailored to exhibit distinct behaviors and appearances using D3 in-

ternally for visual representation and interaction handling. The example employs two types of scatter plot module: one to provide an overview, and the other to present a detailed view. It also employs a parallel coordinate plot module, a table view module, and a gradient plot module. A significant portion of a view module consists of D3 code for constructing and updating the view, while other code sections pertain to methods for updating the view's internal state stored in the view wrapper. To create new types of views as view modules in C4D3, one makes a copy of the wrapper template and customizes it to include the desired D3 code in the functions in those sections.

In C4D3, the code for CMVs is written in a declarative manner by supplying input data and view-specific parameters to the view modules and binding coordination objects between them. Arbitrary subsets of views can share appearances and behaviors by binding different objects that store and manage the desired appearance and behavior states of the visualization as a whole. For instance, it is possible to select a subset of views with similar or dissimilar appearances, such as scatter plots and a table view, to highlight the same data point when hovered over with a mouse in any of these views. In another scenario, the same or different subsets of views can be selected and coordinated by having selected items in one view highlighted in other views. A more complex example, shown in Figure 1, coordinates a range selection in the gradient view with an item highlighting filter applied in the parallel coordinate plot. Views can also be instantiated from the same view module to be coordinated through common design elements. For example, scatter plots that share the same data can yoke their individual axes to the visual extent of the data as each one displays it. Mixing and matching coordination strategies allows for versatile and open-ended ways to coordinate views, offering designers ample flexibility in achieving their desired outcomes.

4 VIEW ABSTRACTION AND COORDINATION IN C4D3

Visualization designers create and use many different kinds of interactive data views. Although the introduction of fundamentally new view types is less and less frequent, the visualization community continues to study known views, their variations, how to implement them, and how to compose them into larger structures including coordinated multiple views. There is particular interest in helping users create CMV designs themselves. Balancing expressiveness and accessibility of specification is a key chal-

lenge in this effort.

With C4D3, we approach this challenge by adopting a level of view abstraction based on the *parameter set model* of visual exploration (Jankun-Kelly et al., 2007). This model treats visualization state as a collection of parameters, some or all of which can be modified through interaction. The Live Properties architecture in the *Improvise* visualization system (Weaver, 2004) builds on the parameter set model by adding a model for defining views in terms of properties (slots), binding those properties to variables (parameters), and broadcasting update notifications between views whenever any of them changes a variable interactively. Instead of coordinating views through low-level interactions (too mechanical) or high-level semantics (too abstract), Live Properties coordinates views through a middle-level of concrete, meaningful state objects. Navigation interactions translate into ranges, angles, and distances. Selection interactions are captured as an array of brushing hits indexed to data items. Inside view implementations, translating common visualization interactions into such state objects is straightforward, as is parameterizing the visual encoding of data as a function of the state objects. Coordination in Live Properties is *accessible* because the state object data types, and how they relate in a view, are easy to understand. It is *expressive* because views can be interactively coupled with each other in a myriad of ways well suited to both standard and custom coordination design.

C4D3 combines a re-implementation of the Live Property architecture in JavaScript with a wrapper for encapsulating the JavaScript code of D3 visualizations as a Live Property view type. To implement a new view type, a visualization developer makes a copy of the wrapper. They then insert code to create the properties of a view when it is instantiated. Next, they insert the D3 code fragments that perform event handling and visual encoding appropriate to that view type. Finally, they modify the fragments to translate interaction events into property changes and calculate visual encodings from property values. Designers can adapt existing C4D3 views or create view variations by adding relevant Live Properties to support coordination in the widely available rich set of D3 examples.

The view wrapper also provides the functionality needed to propagate interaction events and property updates to keep the view updated. Figure 2 shows the internal components of C4D3 views and how they are connected through variables for coordination. Each view has event listeners attached to the DOM corresponding to various mouse, keyboard, and touch events. Each property of a view registers with relevant listeners to receive notification of events. When it re-

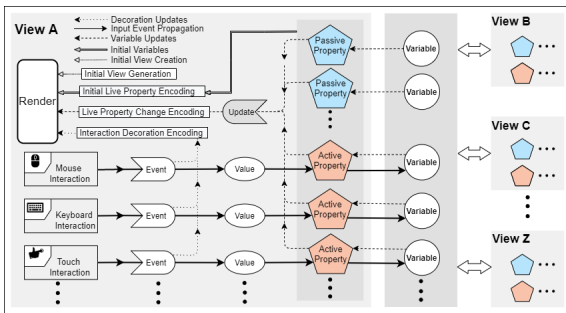


Figure 2: The internal structure of C4D3 views and their external coordination through variables. Interaction events propagate via properties to change variable values. Changed variables broadcast update notifications through all connected properties to trigger view updates.

ceives an event notification, it modifies the value of its bound variable (if it has one). The variable broadcasts the value change to all bound properties, including the originating view itself. Each property that receives a variable change notification triggers a draw event to update its parent view. When a view updates itself, it accesses current property values as needed. The originating view does not necessarily trigger a draw event directly in response to the interaction, but this can be done to draw a visual side-effect of interaction, such as to show the brushing shape itself. This approach allows clean decoupling of event handling code from property update code inside views. This approach also allows variables to be initialized (and even updated by other means) in a way that keeps all views up to date at all times, including upon first display. A property can also be tagged as active or passive to indicate whether interaction in its parent view can affect it or not.

This approach to view abstraction and coordination in C4D3 has several advantages. First, it makes the implementation of individual views entirely independent of each other. Views communicate only indirectly through variables, and have no direct connection to or even knowledge of each other. Second, it eliminates the need to write custom code inside views to implement *the coordinations themselves*. Third, as a consequence of the previous two, it substantially increases the scalability of coordination constructions in terms of the number of views, the number of shared state objects, and the overall amount of interactive interdependence that can be implemented with little code. The specification of the structure of coordination between views is effectively externalized from the views themselves. If one can implement a view type in C4D3, then any number of instances of that view type can be created and coordinated. One can create new interactive designs in D3 and wrap each in the C4D3 wrapper in a way that effectively sepa-

rates the concerns of visual representation design and coordination specification.

C4D3 provides a variety of data types to define interactive state values, such as plot ranges and item selections. Views create, interpret, and utilize property values based on their type. For instance, if a view uses a bit array property to brush and highlight selected items, the visualization designer can create a variable of that type to bind to that view's property, then bind it to other properties of any views to coordinate them through that typed value. Types helpfully communicate each view as a property-based API to designers, and constrain how property values can be expected to effect visual representation and interaction handling in different views. Views can use their property values for representation and/or interaction as they need to, including to parameterize an internal data transformation pipeline. As a result, C4D3 supports flexible coordination well beyond basic navigation and selection, through the addition of view variants that define and utilize their properties in sophisticated ways.

5 IMPLEMENTING CMV IN C4D3

C4D3 is implemented as a JavaScript library that consists of six distinct software layers to build CMVs: (1) the D3 library handles DOM specification and interaction handling of views; (2) modules built on D3 set up a view, render and update it, and update its properties upon a change in state of the view; (3) view wrappers interface view modules to the coordination library; (4) a coordination library manages coordinations between the views; (5) an API layer provides abstraction for interacting with individual view controls to support creation of views and specification of coordinations; and (6) CMV applications build on the API layer. Figure 3 summarizes the general structure of the code one needs to write to implement an application using these layers. First, implement view modules (Figure 3a–d). Next, implement the corresponding view wrappers (Figure 3e–f). Finally, set up and initialize the application itself (Figure 3g–j).

Figure 4 shows a simple example that visualizes a taxonomy of food with three views: a treemap, a radial tree, and a bar chart. Each view has an indication property that determines which item to highlight when that item is hovered over by the mouse in any of the views. The properties are bound to a common variable to establish the coordination. In the rest of this section, we present the complete code to specify the example and explain its construction.

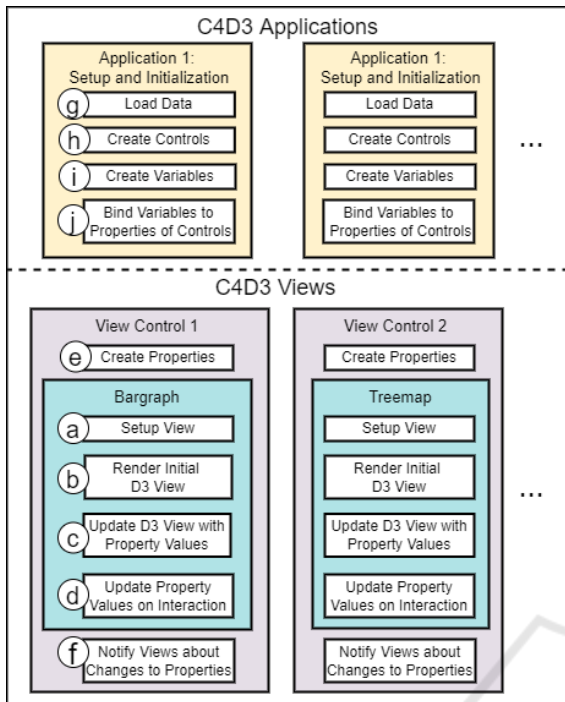


Figure 3: Code organization in C4D3: (a)–(d) view modules; (e)–(f) view wrappers; (g)–(j) CMV applications. See the main text for descriptions of the code for labels (a)–(j).

5.1 Coding View Modules

A CMV application can be built using existing view modules, including several common examples provided by the C4D3 library, and by writing any needed additional view modules and their corresponding view wrappers. View modules are responsible for setting up the initial view representation (Figure 3a–b), updating the views (Figure 3c), and translating interactions in views into property changes (Figure 3d). The majority of code that needs to be written when creating a new view module is the D3 code (Figure 3b–c).

Figure 6 shows the module code for the bar chart view in Figure 4. The code in Figure 6a creates an empty boolean array, initialized with false values, and sized to match the input data, for storing the values of

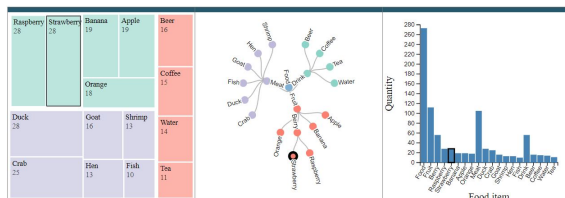


Figure 4: A CMV visualization with three views: a treemap, a radial tree, and a bar chart. The views are coordinated to highlight the item hovered over in any of the views.

an *indication* property to track the mouse hover. Figure 6d shows the code to create the initial bar chart view using the vanilla D3 code for a bar chart shown in Figure 6e (hidden for brevity). The code shown in Figure 6f updates the value of the property when mouse interactions (onMouseOver and onMouseOut) occur, changing the property value by altering the bit values of appropriate DOM elements to true or false. The property change also notifies the view wrapper and triggers any needed updates to the view, as shown in Figure 6c. View modules must separate the code for each of their property types into distinct update methods. The module code for the treemap and radial tree views is structured the same way and was implemented by copying the bar chart module code then modifying the property update code as in Figure 6c and the D3-specific code as in Figure 6e.

```
// Control class for the BarGraph view module
export class ControlBargraph implements Control {
  public CTAG_Indication: string = "Indication";
  public TYPE_Indication: Prototype = new Prototype(
    Array.prototype, this.CTAG_Indication, [false]);
  public proxy: ControlProxy;
  public bargraph: BargraphClass;
  public live_property_Indication: LiveProperty;
  public ControlName: string;

  // Constructor
  constructor(
    name: string,
    data: Array<any>,
    xPos: number,
    yPos: number,
    horizontal: Array<String>,
    vertical: Array<String>,
    inputWidth: number,
    inputHeight: number
  ) {
    this.proxy = new ControlProxy(this);
    this.live_property_Indication = this.proxy.add(
      this.CTAG_Indication, this.TYPE_Indication, true);
    this.ControlName = name;
    this.bargraph = new BargraphClass(
      this, data, name, xPos, yPos,
      horizontal, vertical, inputWidth, inputHeight);
  }

  // Public Methods (Properties)
  public getProxy(): ControlProxy {
    return this.proxy;
  }

  // Method to update the view on live property change
  public propertyChanged(e: LivePropertyEvent): void {
    var tag: string = e.getLiveProperty().getTag();
    if (tag == this.CTAG_Indication) {
      this.bargraph.setIndication(
        e.getLiveProperty().getVariable()
          .getValue() as Array<any>
      );
    }
  }
}
```

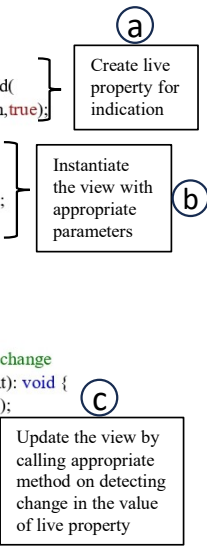


Figure 5: Wrapper (control) code for the example bar chart, showing: (a) initialization of properties, (b) initialization of the view, and (c) notifying the view of property changes.

```

import * as d3 from "d3";
import { Draggable } from "../utils/Draggable.js";
import { ControlBargraph } from "../control/ControlBargraph.js";
import { Utils } from "../utils/utils.js";

export class BargraphClass {
  ControlName: string;
  Data: Array<any>;
  IndicationDictionary: Object;

  constructor(
    control: ControlBargraph,
    data: Array<any>,
    name: string,
    xPos: number,
    yPos: number,
    horizontalAxis: Array<String>,
    verticalAxis: Array<String>,
    inputWidth: number,
    inputHeight: number
  ) {
    this.ControlName = name;
    this.IndicationDictionary = Utils
      .createBooleanDictionaryFromArray( data[0], "Indication" );

    this.render( control, data[0], xPos, yPos,
      horizontalAxis, verticalAxis, inputWidth, inputHeight );
  }

  //Make the view draggable across the layout
  makeDraggable() {
    var D1 = new Draggable();
    var temp = this.ControlName;

    try {
      document
        .getElementById(this.ControlName + "header")!
        .addEventListener( "mousedown",
          function () {
            D1.dragElement( document.getElementById(temp));
          },
          false
        );
    } catch (error) {
      throw Error(
        "The Element by id " + this.ControlName + " do not exist"
      );
    }
  }

  // Setters for updating the view
  setIndication(IndData: Array<any>) {
    if (this.Data === undefined || IndData[0] === undefined) {
    } else {
      d3.select(`#${this.ControlName}`)
        .selectAll("rect")
        .style("stroke-width", 2)
        .style("stroke", function (d, index) {
          return IndData[0][d["id"]].Indication === true ? "black" : null;
        });
    }
  }
}

*** Continued on next column ***

*** Continued from previous column ***

//render the view
private render(
  parentControl: ControlBargraph,
  data: JSON,
  xPos: string | number,
  yPos: string | number,
  xAttr: String[] | (string | number)[],
  yAttr: String[] | (string | number)[],
  plotWidth: number,
  plotHeight: number
): void {
  var div1: HTMLDivElement = document.createElement("div");
  div1.id = this.ControlName;
  div1.className = "bargraph";
  div1.style.left = xPos + "px";
  div1.style.top = yPos + "px";

  //create a div header for the scatterplot
  var div2 = document.createElement("div");
  div2.id = div1.id + "header";
  div2.className = "bargraphheader";

  //make the div header a child to the div element
  div1.appendChild(div2);
  var IndicationDictionary = this.IndicationDictionary;

  function renderBargraph(data: any | ArrayLike<unknown>) {
    // The core D3.js code to implement the bargraph view
    // would be about 70 lines ...

    // Add event listeners to the bars
    bars.on("mouseover", onMouseOver).on("mouseout", onMouseOut);

    // Event handler for mouseover
    function onMouseOver(d: any, event: Event) {
      Utils.setBooleansToFalse(IndicationDictionary, "Indication");
      IndicationDictionary[event.Id].Indication = true;

      parentControl.getProxy().getLiveProperty("Indication")
        .setValue([IndicationDictionary]);
    }

    // Event handler for mouseout
    function onMouseOut(d: any) {
      Utils.setBooleansToFalse(IndicationDictionary, "Indication");
      parentControl.getProxy().getLiveProperty("Indication")
        .setValue([IndicationDictionary]);
    }

    renderBargraph(data);
    document.body.appendChild(div1);
    this.makeDraggable();
  }
}

```

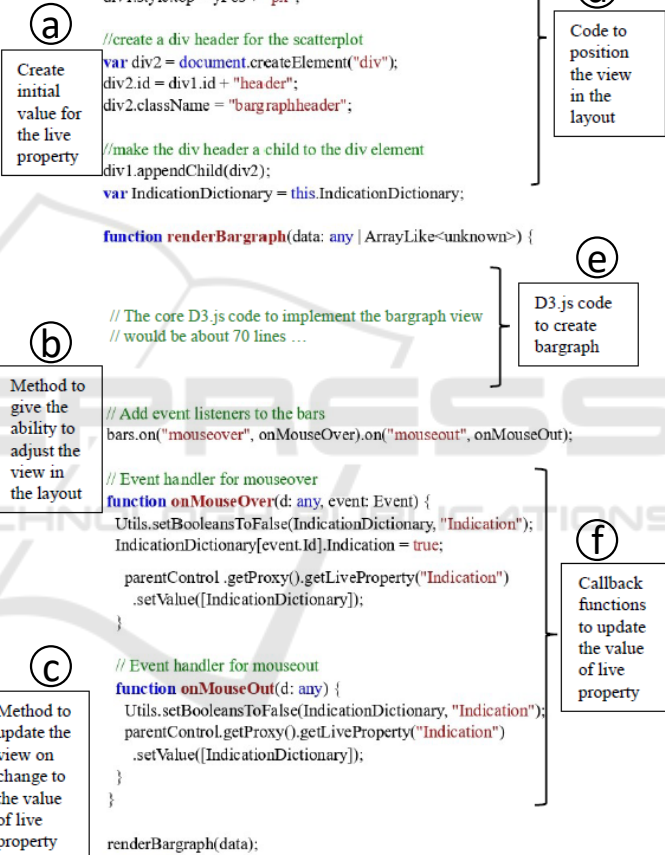


Figure 6: Module code for the example bar chart, showing: (a) initialization of properties, (b) a utility method to enable interactive positioning of the view, (c) a method to update the view upon property changes, (d) code to position the view in the layout, (e) D3 code to render the view, and (f) callback functions to update the property upon interaction in the view.

5.2 Coding View Wrappers

Each view module has a corresponding view wrapper (also referred to in the API as a view control). Creating a wrapper for a new view module focuses on de-

termining which properties a view needs to have for coordination with other views. The wrapper defines and maintains the properties that encapsulate the appearance and behavior of the view (Figure 3e) and acts as an interface between the view module and

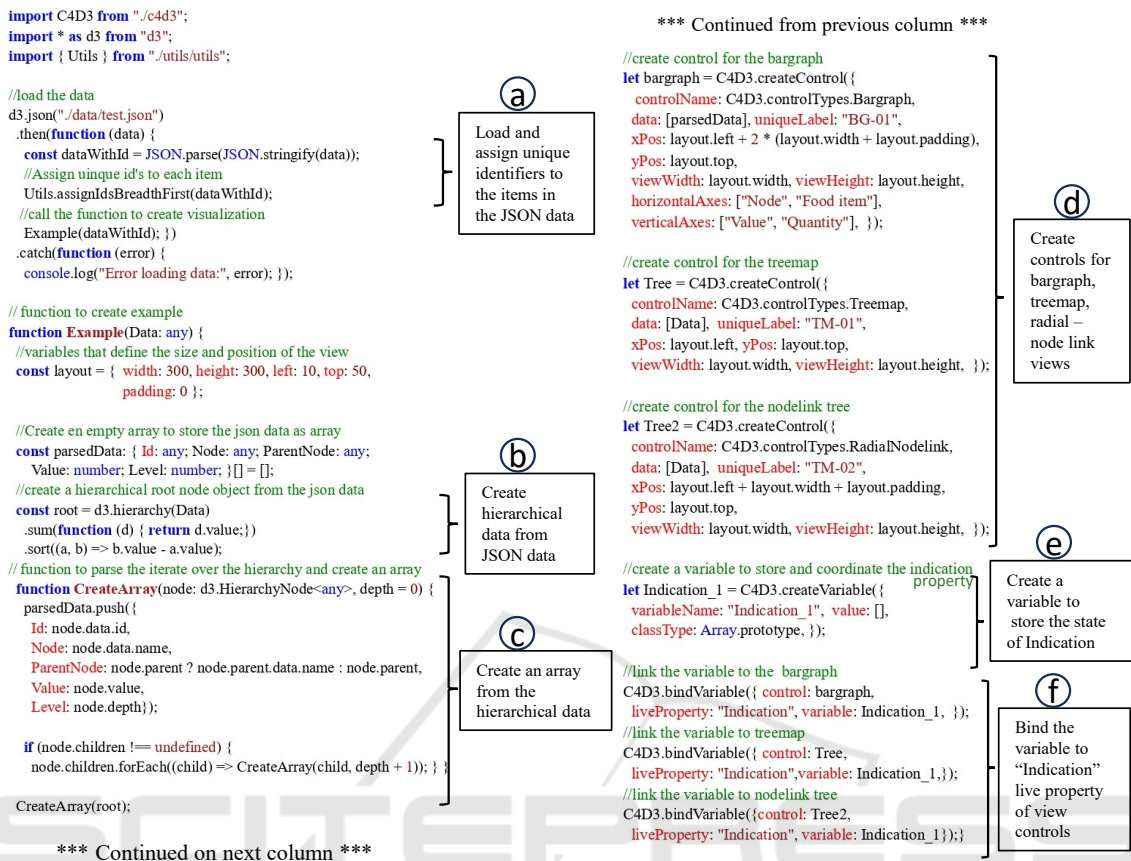


Figure 7: Application code for the example, showing: (a)–(c) loading and preparing data, (d) instantiating views, (e) instantiating variables for the hover coordination, (f) binding the variable to the properties of views to establish the coordination.

the coordination layer by updating the view when a change is detected in any of its properties, either by the result of interaction with the view itself or by the result of coordination with other views (Figure 3f).

Figure 5 shows the wrapper code for the example bar chart view in Figure 4. The code in Figures 5a–c creates the *indication* property (called “live.property.indication”), instantiates the view for that property, and notifies the view of incoming changes to that property from outside the view, respectively. In Figure 5a, the property is tagged as an active property (“true” on the third line) that can be changed by the view itself, as opposed to a passive property (which would be “false” on that line) to indicate that the property can change due to coordination with other views but is not affected by interaction in its own view. The wrapper code for the treemap and radial tree views is structured the same way and was implemented by copying the bar chart wrapper code then modifying the code in Figure 5b to instantiate each respective view with its needed parameters.

5.3 Coding CMV Applications

Creating a CMV application involves initializing the constituent views and specifying coordination using the C4D3 API layer, broken down into the four steps shown in Figure 3g–j. First, load the data and create parameters that define the size and position of the views, such as width, height, and position. Second, instantiate the view controls with this data and the parameters. (Alternatively, the layout of the CMV can be managed using a CSS library like Bootstrap to support a responsive grid system for arranging individual views, further simplifying the code for individual view setup.) Third, create variables to store the state of the views as property values. Fourth, bind the variables to the appropriate properties of the view controls to specify coordination.

For the example shown in Figure 4, we first load and prepare the data (Figure 7a–c), then create layout parameters and instantiate the view wrappers for the bar chart, treemap, and radial tree views (Figure 7d). Next, we create a variable called *Indication* to store

the coordinated interaction state of hovering shared by the views (Figure 7e). We use a simple array as the property type to represent the shared hover interaction state, with *True* and *False* in the array to represent whether a given data item is being hovered over. Finally, we coordinate the views by binding the *Indication* variable to the *Indication* property of all three view wrappers (Figure 7f).

Evolving a CMV application to include additional views and/or coordinations is a straightforward matter of instantiating more view wrappers (adding blocks to Figure 7d), creating more variables (adding blocks to Figure 7e), and/or binding variables to more view properties (adding blocks to Figure 7e). A coordination design can be rapidly and incrementally revised by mixing and matching variables to views through their properties. Incremental creation of new view types or modification of existing ones can be performed by copying and editing view module and wrapper code before returning to the application code to incorporate those view types and coordinate them.

6 COORDINATION PATTERNS

With multiple views in a visualization, users can see and interact with data from multiple perspectives simultaneously. Coordination allows users to combine the interactions of different views to compose more complex queries than would be possible in any single view. To support common data exploration strategies (Shneiderman, 1994), many visualizations employ basic navigation and selection coordinations (North and Shneiderman, 1997), such as brushing (Becker and Cleveland, 1987). Techniques like compound brushing (Chen, 2003) and cross-filtering (Weaver, 2010; Square, 2012) employ coordination constructions that interplay with data processing and querying in more complex ways.

While our current view abstraction focuses on supporting basic forms of coordination, it is sufficient to realize many common coordination patterns (Weaver, 2007b), and visualization designers can vary and synthesize them to suit specific application needs. Transparency of D3 code in views also leaves the door open for designers to implement direct view dependencies if they so desire. Here we describe a few of the general coordination patterns that were readily designed into the above C4D3 examples of ion motion (Figure 1) and food types (Figure 4).

- *Synchronized scrolling* in Figure 1A coordinates the horizontal range of time visible in three time series plots. Each plot has a pair of live properties to represent its visible horizontal and verti-

cal ranges, each corresponding to a translated axis value in D3. Changes to either range are applied to the corresponding axis and the points in the plot are repositioned accordingly.

- A *scatterplot matrix* (SPLOM) shown in Figure 1D is a more complex construction of range bindings. Three data dimensions are laid out in a traditional stair-step arrangement of plots. X, Y, and Z range variables are bound to the range properties of the plots so as to create the expected synchronization of scrolling across all three views.
- Figure 1F shows a custom example of a *perceptual slider* coordination between a color slider and a parallel coordinate plot. The two views share a range variable for a selected range of color in a color gradient. The two views also share a color gradient for sake of visual consistency. In this case, we implemented both as general-purpose views but tweaked their respective D3 code to use the same gradient by coincidence.
- *Shared selection* in Figure 1 supports brushing between the three time series plots and the table view. The three plots in the SPLOM are similarly coordinated, but via a second selection variable.
- *Shared indication* in Figure 4 similarly coordinates hover interactions across several of the views, in this case of a food item. As in shared selection, data items hovered over in any of the coordinated views is highlighted in all views.

7 DISCUSSION

We developed a specialized coordination library that integrates well with the popular data and visual representation capabilities of D3, enabling flexible combination of views and coordination to build simple or complex CMV visualizations. In the following section, we discuss the design choices we made for C4D3 and assess achievement of development objectives.

7.1 Modular View Support

Building CMVs using C4D3 can take advantage of substantial code reuse. A large fraction of the code in view wrappers is shared between views with a small portion needed to add new properties for coordination. Implementation of new views nevertheless requires structuring D3 code in a different, modular way to support coordination, as discussed in Section 5.1.

The complexity of code in a view depends mainly on how expressive we want the view to be to present

itself as a collection of properties available for coordination with other views. New C4D3 designers should start with CMV examples that support simple coordination patterns, such as shared selection and shared navigation, and incrementally develop support for more complex coordination patterns.

Despite its name and concentration on D3, the implementation of C4D3 does not inherently restrict designers from substituting an alternative charting library for D3. However, the alternatives would likely vary widely in how they can be used to manipulate the DOM to express interaction state and visual appearance as a function of coordinated property values.

CMV view initialization in C4D3 is abstracted from the low-level details of view creation, giving designers the ability to build complex CMVs in a declarative manner without getting embroiled by the low-level implementation details of views. The separation of concerns and abstraction offered by properties and shared interaction state appears to provide a strong correspondence between what users want for visual data querying and how designers can specify suitable view inter-dependencies in CMV constructions.

A major challenge in developing support for complex coordination on top of visualization libraries like D3 is their limited range of interactions available to modify DOM elements (brush, drag, zoom, pan, lasso) and the ways they can conflict. This limitation makes it much harder to implement multiple interactions in D3 to support multiple properties in a C4D3 view, either by supporting a coordination pattern by spreading out needed interactions across extra views, or forcing developers to write custom code to process low-level mouse and keyboard events to sufficiently distinguish interaction forms and map them into particular facets of coordinated interaction state.

7.2 Modular View Update Mechanism

In general, D3 views are constructed such that mouse events trigger callbacks that update the DOM, making code look clean and simple. This is true for simple and direct interactions. However, when changes are complex and indirectly triggered by coordinated views, updating in this manner makes the code monolithic and adds complexity to the already lengthy code for some D3 views. To manage this complexity, C4D3 decouples the code responsible for re-rendering views into separate callback functions, simplifying the D3 code and making it more modular. This delegates the responsibility of mouse events to only update properties, allowing each view to interpret changes to properties and modify themselves accordingly.

This approach also allows views to support coordi-

inations that span multiple properties in an easily understandable manner. For example, if a view receives shared selection state in its properties from multiple views via multiple variables, it can adjust its appearance based on the combined values of any or all variables. Achieving this level of coordination without C4D3 would be highly complex in D3 alone.

C4D3 can also support sharing data as a coordination parameter, allowing it to be loaded once instead of multiple times across views. This approach not only improves data handling, but also enables the development of views that support data annotations through interactions and share the same modified data across views, ensuring consistency of annotated data across coordinated views.

7.3 Assessment of Objectives

Our first objective is to preserve **transparency** of D3 code in view implementations. Parameterizing views via minimal sets of properties needed to share coordination state provides accessibility. Giving each view responsibility over how to use property values to determine appearance and behavior provides expressiveness. A visualization designer can choose a visual encoding appropriate to the data and application, and implement interaction as needed to accomplish many common visual querying tasks. Item selection in a view can involve clicks, rubber bands, lassos, or other selection gestures. Indication by drawing or hovering is similarly flexible. In both cases, the view simply performs the usual hit tests on data to populate a bit array to share via a selection variable.

Specifying views without connecting them directly offers both **extensibility** and **modularity** in view development. D3 practitioners can take advantage of the view abstraction to create new view types, view variants, and coordinate them incrementally as needed to realize their designs. Design variations can be explored by quickly changing the set of available variables and their bindings to views being considered, often with little or no modification to the D3 code itself. Doing so depends on how well the aspects of appearance and behavior needed for variation are exposed as properties in available view types. Overall, C4D3 provides a sharp separation of concerns for view design. Starting from available view types, new view types can readily exploit redundancy in C4D3 code, and often in D3 code as well. The view types in the examples all share code that is essentially the same except for D3 fragments that handle visual encoding specifics. Event handling code, such as for panning, can be reused in the same way as in the world of D3 examples. View variants can offer

custom combinations of interactions for use in special design cases. Curating collections of view types is likely to be an integral activity of C4D3 designers, much like in the D3 community.

View designers specify how property values affect a view's appearance and behavior, and how interactions in a view affect property values. The choice of properties to design into a view type is flexible, yet often calls for only a small set of obvious and sensible value types. Properties tend to be compatible (and often identical) across views, even quite different ones. Consequently, the **composability** of views is high.

When an interaction occurs in a view, it could update itself by responding directly to the interaction, indirectly to a property notification, or both. In C4D3, views translate most interactions into one or more property changes. Updates happen indirectly via propagation of variable changes, including in the view in which an interaction originated. By updating themselves independently of where interactive modifications originate, views effectively track the current coordination state of the visualization, and thus maintain visual **consistency** within themselves and between each other. The asynchronous propagation of events in C4D3 makes visual consistency between views *eventual*, but on current systems updates appear immediate for data sizes and view counts like in the examples. It also avoids cycles in view updates.

The main objective of C4D3 is **consonance** in CMV design; that is, to support harmonious mixing and matching of multiple coordinations between diverse views. The two examples represent different C4D3 design cases. The first recreates an existing CMV visualization with many, but basic views. The second creates a new CMV visualization with fewer, but mixed, views. Together, they show off a variety of navigation and selection coordination patterns. To build a C4D3 visualization, the designer creates variables to model interactive state, creates views to access and modify that state, then binds variables to views to coordinate them. These steps are simple, flexible, and easy to write as C4D3 code. The number of variables and views, and more importantly the number of bindings, is typically quite small—tens of each—even in complex CMV constructions like the ions example. Even so, there is still much practical work to be done, particularly to expand beyond the small set of essential view and property value types currently included in the library. It also remains unclear how to adapt certain D3 event handling, notably for drag interactions, to the properties model.

Overall, C4D3 lets designers focus on expressing the state, behavior, and appearance of views as shareable parameters at an abstraction level that provides

a closeness of mapping between desired view interdependencies and the code needed to express them, leaving most of the complexity of handling coordination proper to the library itself.

8 CONCLUSION

We present a working prototype of a view-level abstraction for coordinating multiple views. We adapted the Live Properties architecture as a thin coordination layer for use with the D3 web-based visualization library, and applied it to create examples that demonstrate its effectiveness to create complex CMV visualizations. Thinking about views as collections of abstract shareable properties, rather than directly shareable values, may encourage general thinking of coordination on a more abstract level. This middle-level abstraction appears to allow designers to think about coordination at a higher level of abstraction like *Nebula* (Chen et al., 2022), while keeping the flexibility benefits of a lower level of mechanism (Keller et al., 2024). Moving forward, we will explore migration of the view abstraction to serve as a CMV abstraction layer for other visualization libraries such as *Vega-Lite* (Satyanarayan et al., 2017) and *Plotly.js*. More broadly, we aim to support access to the full visualization data processing pipeline, starting with dynamic queries (Ahlberg et al., 1992) and extending C4D3 capabilities to include a functional reactive language for visualization pipelines like in *Improvise* (Weaver, 2004). Another possibility is to extend C4D3 into a client-server architecture for desktop, mobile, and web clients using the coordination architecture as the basis for remote interaction including collaborative interaction via coordination (Weaver, 2007a).

ACKNOWLEDGEMENTS

We thank Remco Chang and Wenbo Tao for their insights and feedback. This work was supported by National Science Foundation Award #1351055.

REFERENCES

- Ahlberg, C. (1996). Spotfire: An information exploration environment. *ACM SIGMOD Record*, 25(4):25–29.
- Ahlberg, C., Williamson, C., and Shneiderman, B. (1992). Dynamic queries for information exploration: An implementation and evaluation. In *Proceedings of the SIGCHI Conference on Human Factors in Comput-*

- ing Systems, CHI '92, pages 619–626, New York, NY, USA. Association for Computing Machinery.
- Ahlberg, C. and Wisstrand, E. (1995). IVEE: An information visualization & exploration environment. In *Proceedings of the IEEE Symposium on Information Visualization (InfoVis)*, pages 66–73, 142–143, Atlanta, GA. IEEE.
- Aiken, A., Chen, J., Stonebraker, M., and Woodruff, A. (1996). Tioga-2: A direct manipulation database visualization environment. In *Proceedings of the Twelfth International Conference on Data Engineering*, pages 208–217, New Orleans, LA, USA. IEEE Comput. Soc. Press.
- Becker, R. A. and Cleveland, W. S. (1987). Brushing Scatterplots. *Technometrics*, 29(2):127–142.
- Bostock, M. and Heer, J. (2009). Protovis: A Graphical Toolkit for Visualization. *IEEE Transactions on Visualization and Computer Graphics*, 15(6):1121–1128.
- Bostock, M., Ogievetsky, V., and Heer, J. (2011). D³ Data-Driven Documents. *IEEE Transactions on Visualization and Computer Graphics*, 17(12):2301–2309.
- Boukhelifa, N. and Rodgers, P. J. (2003). A Model and Software System for Coordinated and Multiple Views in Exploratory Visualization. *Information Visualization*, 2(4):258–269.
- Buja, A., Cook, D., and Swayne, D. F. (1996). Interactive High-Dimensional Data Visualization. *Journal of Computational and Graphical Statistics*, 5(1):78–99.
- Chen, H. (2003). Compound brushing. In *Proceedings of the IEEE Symposium on Information Visualization (InfoVis)*, pages 181–188, Seattle, WA. IEEE Computer Society.
- Chen, R., Shu, X., Chen, J., Weng, D., Tang, J., Fu, S., and Wu, Y. (2022). Nebula: A Coordinating Grammar of Graphics. *IEEE Transactions on Visualization and Computer Graphics*, 28(12):4127–4140.
- Fekete, J.-D. (2004). The InfoVis Toolkit. In *IEEE Symposium on Information Visualization*, pages 167–174, Austin, TX, USA. IEEE.
- Fekete, J.-D., Hémy, P.-L., Baudel, T., and Wood, J. (2011). Obvious: A meta-toolkit to encapsulate information visualization toolkits — One toolkit to bind them all. In *2011 IEEE Conference on Visual Analytics Science and Technology (VAST)*, pages 91–100.
- Hill, R. D., Brinck, T., Rohall, S. L., Patterson, J. F., and Wilner, W. (1994). The *Rendezvous* architecture and language for constructing multiuser applications. *ACM Transactions on Computer-Human Interaction*, 1(2):81–125.
- Jankun-Kelly, T., Ma, K.-I., and Gertz, M. (2007). A Model and Framework for Visualization Exploration. *IEEE Transactions on Visualization and Computer Graphics*, 13(2):357–369.
- Keller, M. S., Manz, T., and Gehlenborg, N. (2024). Use-Coordination: Model, Grammar, and Library for Implementation of Coordinated Multiple Views.
- Livny, M., Ramakrishnan, R., Beyer, K., Chen, G., Donjerkovic, D., Lawande, S., Myllymaki, J., and Wenger, K. (1997). DEVise: Integrated querying and visual exploration of large datasets. In *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data - SIGMOD '97*, pages 301–312, Tucson, Arizona, United States. ACM Press.
- Myers, B., Giuse, D., Dannenberg, R., Zanden, B., Kosbie, D., Pervin, E., Mickish, A., and Marchal, P. (1990). Garnet: Comprehensive support for graphical, highly interactive user interfaces. *Computer*, 23(11):71–85.
- North, C. and Shneiderman, B. (1997). A taxonomy of multiple window coordinations. Technical Report CS-TR-3854, University of Maryland Department of Computer Science.
- North, C. and Shneiderman, B. (2000). Snap-together visualization: A user interface for coordinating visualizations via relational schemata. In *Proceedings of the Working Conference on Advanced Visual Interfaces, AVI '00*, pages 128–135, New York, NY, USA. Association for Computing Machinery.
- Roth, S., Lucas, P., Senn, J., Gomberg, C., Burks, M., Strofolino, P., Kolojechick, A., and Dunmire, C. (1996). Visage: A user interface environment for exploring information. In *Proceedings IEEE Symposium on Information Visualization '96*, pages 3–12, San Francisco, CA, USA. IEEE Comput. Soc. Press.
- Satyanarayanan, A., Moritz, D., Wongsuphasawat, K., and Heer, J. (2017). Vega-Lite: A Grammar of Interactive Graphics. *IEEE Transactions on Visualization and Computer Graphics*, 23(1):341–350.
- Satyanarayanan, A., Russell, R., Hoffswell, J., and Heer, J. (2016). Reactive Vega: A Streaming Dataflow Architecture for Declarative Interactive Visualization. *IEEE Transactions on Visualization and Computer Graphics*, 22(1):659–668.
- Satyanarayanan, A., Wongsuphasawat, K., and Heer, J. (2014). Declarative interaction design for data visualization. In *Proceedings of the 27th Annual ACM Symposium on User Interface Software and Technology*, pages 669–678, Honolulu Hawaii USA. ACM.
- Shneiderman, B. (1994). Dynamic queries for visual information seeking. *IEEE Software*, 11(6):70–77.
- Square (2012). Crossfilter: Fast multidimensional filtering for coordinated views.
- Takatsuka, M. and Gahegan, M. (2002). GeoVISTA Studio: A codeless visual programming environment for geoscientific data analysis and visualization. *Computers & Geosciences*, 28(10):1131–1144.
- Weaver, C. (2004). Building Highly-Coordinated Visualizations in Improve. In *IEEE Symposium on Information Visualization*, pages 159–166, Austin, TX, USA. IEEE.
- Weaver, C. (2007a). Is Coordination a Means to Collaboration? In *Fifth International Conference on Coordinated and Multiple Views in Exploratory Visualization (CMV 2007)*, pages 80–84, Zurich, Switzerland. IEEE.
- Weaver, C. (2007b). Patterns of coordination in Improve visualizations. In *Visualization and Data Analysis 2007*, volume 6495, pages 188–199. SPIE.
- Weaver, C. (2010). Cross-Filtered Views for Multidimensional Visual Analysis. *IEEE Transactions on Visualization and Computer Graphics*, 16(2):192–204.