# IMPROVISE: A USER INTERFACE FOR INTERACTIVE CONSTRUCTION OF HIGHLY-COORDINATED VISUALIZATIONS

By

**Christopher Eric Weaver**

A DISSERTATION SUBMITTED IN PARTIAL FULFILLMENT OF THE

REQUIREMENTS FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

(COMPUTER SCIENCES)

at the

**UNIVERSITY OF WISCONSIN – MADISON**

2006

For Kim . . . gdmfsoab!

# Abstract

The development of exploratory visualization tools based on coordinated multiple views has become an important area of information visualization research. One goal of this research is to allow users to explore their data by switching freely between building and browsing in a flexible, integrated, interactive graphical environment that requires minimal programming skill to use. However, the range of possibilities for displaying data across multiple views depends on the flexibility of coordination, the expressiveness of visual abstraction, and the ability of users to comprehend the structure of their visualizations as they work. As a result, data exploration has been limited in practice to a small subset of visualizations that are both useful and usable.

This dissertation describes four primary contributions to the field of information visualization. The first contribution is a conceptual model of multiple view coordination, *Live Properties*, in which views tightly couple through shared objects that describe the navigation and selection state of a visualization.

The second contribution is a visual abstraction language, *Coordinated Queries*, loosely based on the relational data model. Views aggregate, filter, sort, and graphically encode data records using declarative expressions that can be defined in terms of interactive parameters.

The third contribution is a conceptual model of *metavisualization*, in which the interactive structure of coordinated visualizations can be explored by visualizing it *in situ*, directly in the running visualizations themselves.

The fourth contribution is *Improvise*, an implemented system in which users build and browse multiview visualizations interactively using Live Properties and Coordinated Queries. By coupling visual abstraction with coordination, users gain precise control over how navigation and selection in a visualization affects the appearance of data in individual views. In the

Improvise user interface, users can create and modify visualizations quickly and incrementally during data exploration, optionally using metavisualization to explore interactive structure as it evolves. As a result, it is practical to build visualizations with many more views and much richer coordination in Improvise than in other exploratory visualization environments.

# Acknowledgements

Thanks to the members of my dissertation committee—Raghu Ramakrishnan, Mike Gleicher, Miron Livny, Chuck Dyer, and Mark Harrower—for their generous time, effort, and many suggestions. Grateful thanks to Miron and Raghu for their support and guidance during my time with the DEVise research group. Thanks also to Lorene Webber, Marie Johnson, Cathy Richard, Virginia Werner, Heather Cotes, John O'Malley and the Computer Systems Lab for making university and department life so much easier over the years. Special thanks to Chris North, Steve Eick, Harry Hochheiser, Marjan Trutschl, and Urska Cvek for feedback, advice, encouragement, and fellowship when I needed it most.

Thanks to members of the DEVise team, including Kevin Beyer, Donko Donjerkovic, Vuk Ercegovac, and Shilpa Lawande. In particular, discussions with Kevin about potential connections between coordinated multiview interfaces and multiple query processing strategies led to many of the early ideas that evolved into Coordinated Queries (chapter 4). The work on standalone metavisualization of DEVise visualizations (chapter 8) is an outgrowth of Shilpa's DEVise Layout Manager. I am especially indebted to Kent Wenger for answers to my innumerable questions about the inner workings of DEVise.

Cheers to the whole gang at my new home in GeoVISTA: Alan MacEachren, Mark Gahegan, Donna Peuquet, Ian Turton, James Macgill, and all the great graduate students. Special thanks to Ritesh Agrawal, Chaomei Chen, Jin Chen, Jianwei Dou, David Fyfe, Deryck Holdsworth, Junyan Luo, Scott Pezanowski, Anthony Robinson, Mike Stryker, and all the students in the Geovisual Analytics seminar for actually putting Improvise to work. The Penn State Department of Geography is truly an extraordinary place, and I am profoundly grateful to be a part of it.

Thanks to the members of my distraction committee—Nate Bockrath, Nick Leavy, and Rajesh Raman—for only slightly more carousing than is generally called for. What's in the flask, Egg? Magic potion?

I would be remiss if I did not note my camaraderie with the grand old boys on the Condor team including Jim Basney, Todd Tannenbaum, and Derek Wright. Up to the Mickey's Challenge, every last one of them. For many parties and the occasional couch, thanks to Brenda Mudry, Kartik Chandran, Kay Shot, Jim Mosher & Christie Truly, and Aaron Williamson. To my new friends in Happy Valley—Anthony Robinson, Brandi Nagle, Matt & Jess Whitehead—it's like being on reality TV without the cameras!

To my old friend Paul Fontana (and founding member of TSMWK), stop reading immediately and get back to work!

To my dear friend Kim Johnson, thank you for everything. Enjoy the book report, sweetie. Where's my trip to New Zealand?

Most of all, I would like to thank my family—David, Carol, Kate, Mike, and Annika—for their unequivocal love and support. All I've ever really needed is a steady northwest wind...

# Contents

# List of Tables

# List of Figures

# Chapter 1

# Introduction

## 1.1 Overview

The development of interactive visualization construction tools is an important and rapidly growing area of research in information visualization. The purpose of these tools is to allow users to explore and analyze complex information visually, starting with raw data from their knowledge domain and a few ideas for processing and presenting it. The goal is to give users as much control over the process of data exploration and analysis as possible. Ideally, users would be able to explore their data by switching freely between building and browsing in a flexible, integrated, graphical environment that is completely interactive and requires minimal programming skill to use.

The high dimensionality of many data sets necessitates visualization designs composed of multiple views that display low dimensional slices of the data. In such visualizations, interaction can be defined in terms of *coordinations* that determine how the appearance and behavior of each view depends on navigation and selection in other views. Coordination is a way to overlap data dimensions graphically and interactively across multiple views, enabling the display of hundreds or thousands of data dimensions despite the fact that views are individually capable of usefully displaying only a handful of dimensions (typically two to seven) each. Common patterns of coordination include:

- *Synchronized Scrolling.* Two views show the same data items, the corresponding items

in different data sets, or the same region of a coordinate space. For instance, two scatter plots can be synchronized to show the same `X` and/or `Y` ranges. When the user pans or zooms in one scatter plot, the other translates or scales simultaneously.

- *Overview+Detail.* One view shows details about the items selected in another view. Overview+Detail can also involve navigation, such as when a rubber band in one scatter plot (the "overview") is coupled with the visible `X` and `Y` ranges of another scatter plot (the "detail view"). When the user drags or stretches the rubber band, the detail scatter plot translates or scales in tandem. Similarly, panning or zooming the detail scatter plot causes the rubber band to move or change size.

- *Brushing.* Items selected in a view are highlighted in another view. Brushing is usually closed over a group of views, so that selection of items in any view causes the corresponding items to be highlighted in all views. Highlighting can take the form of any visual differentiation between selected and unselected items, including (in)visibility.

- *Drill-Down.* Selection of an item in a primary view specifies data to be shown in secondary views, such as by loading or aggregation. For example, the rows in a list might show the names of multiple related data sets (or subsets of one large data set) collected over a sequence of laboratory experiments. Selecting one of the names in the list causes secondary views to display the corresponding data set.

- *Semantic Zoom.* Navigation in a view changes the appearance of items shown in it. For instance, the items visible in a scatter plot can be drawn as rectangles when zoomed out, and as text when zoomed in. Multiple levels of zoom are possible, such as in geographic maps to show progressively higher detail as the user zooms in.

There are a growing number of both general-purpose and domain-specific interactive construction tools for building visualizations with multiple coordinated views. However, the approaches to coordination employed by these tools have two shortcomings.

First, the set of available visualization "building blocks" is small. Existing systems provide a handful of the most common views and support a limited set of distinct coordination types. Each view is compatible with a subset of this already small set of coordination types. Moreover, each instantiated view in a visualization can be involved in only a few coordinations at any given time. The limited number of ways in which any given combination of views can be coordinated precludes the construction of many potentially useful multiview visualizations.

Second, visualization designers cannot create new coordination types or customize existing ones. In particular, they cannot specify interactive dependencies between views in a manner that is simultaneously flexible, precise, high-level, live, and driven by the data. For instance, designers cannot specify:

- which spatial dimensions are coupled (synchronized scrolling),
- which navigations cause what details to be shown (overview+detail),
- how to highlight items (brushing),
- where to load data or how to aggregate data (drill-down), or
- how navigation affects the appearance of items (semantic zoom).

Instead, visualization designers must choose from a limited set of coordinations anticipated and provided by visualization system developers. As a result, they have little control over how user interaction affects the behavior of views and the appearance of data in them.

A key problem facing developers of visualization construction tools is providing flexible, expressive, easy-to-use means for designers to customize coordinations. Existing tools provide

views and coordinations that are easy to learn individually and simple to combine into small visualizations. Unfortunately, constructing visualizations in these tools using pre-packaged building blocks becomes increasingly hard—if not impossible—as the need to explore and analyze increasingly complex information motivates visualization designs that contain more views and more extensive coordination.

## 1.2 Highly-Coordinated Visualization in Improvise

This dissertation addresses the problem of providing a small but flexible set of coordination building blocks that can be used for interactive construction of highly-coordinated visualizations. The general approach is to allow visualization designers to declare and compose views, interactive parameters, and declarative language statements at an intermediate level of abstraction. This is in contrast with previous approaches in which designers either connect views using a small set of predefined coordination types at a high level of abstraction, or write programs or scripts to specify arbitrary interactive dependencies between views at a low level of abstraction. The approach aims for a "sweet spot" that most effectively balances expressiveness, speed, and difficulty in visualization design. In effect, visualization construction is treated as a niche of interactive user interface construction. Specializing the design process for higher level abstractions particular to visualization (and thus familiar to visualization designers) allows designers to focus on aspects of construction that are necessary and sufficient to realize a large subset of user interfaces that are widely recognized as being "information visualizations."

Improvise is an implemented system and end-user application for building and browsing visualizations of structured information, including relational data. Like other visualization systems, Improvise enables users to load data, create views, specify visual abstractions, and establish coordinations interactively. Unlike other systems, Improvise provides a rich visual

abstraction language that can be coupled with a shared-object coordination mechanism, thereby increasing the expressive power of both.

Improvise facilitates flexible data exploration by offering users fine-grained control over the appearance of visualized data. It combines a simple, direct coordination mechanism called *Live Properties* with a more powerful, indirect coordination mechanism called *Coordinated Queries*. The combination is a significant improvement over existing coordination approaches because it enables users to define complex interactive dependencies between views in terms of both appearance and behavior.

Views in Improvise are coordinated using a symmetric update and notification mechanism that connects views and other controls through shared objects. Controls interpret these shared objects as basic graphical characteristics (colors, fonts, fill patterns, etc.), spatial locations and regions (points, ranges, angles, etc.), data, and data querying operations (projections, filters, selections, etc.) Visual abstractions are created by projecting and filtering data items through declarative expressions that can themselves be defined in terms of shared objects. By editing the expressions used by views to project and filter data, users are able to specify what data to draw, how to draw it, and where to draw it in the views of a visualization.

### 1.2.1 Example

The following example demonstrates how Improvise can be used to build and browse a visualization of albums in a music collection by importing data, creating and laying out views, and specifying data appearance interactively. Even in this small example—which takes only a few minutes for an experienced designer to build—there are sophisticated coordinations between multiple different views of six-dimensional data. Moreover, building occurs in four editor dialogs that pop up directly within the evolving visualization user interface. As a result, design

is incremental, scalable (both in number of views and amount of coordination), and occurs in context in a way that enables ongoing exploration in which analysis prompts additional design steps, thereby encouraging a data-driven, organic style of visualization construction.

The user starts by importing tabular data that describes album attributes (in this case, from a text file containing a playlist that was exported from a popular music application). The `Lexicon` window (figure 1) displays imported data sets, grouped by relational schema. The same window is used to dynamically build and edit lexical language statements (query expressions) in Coordinated Queries.

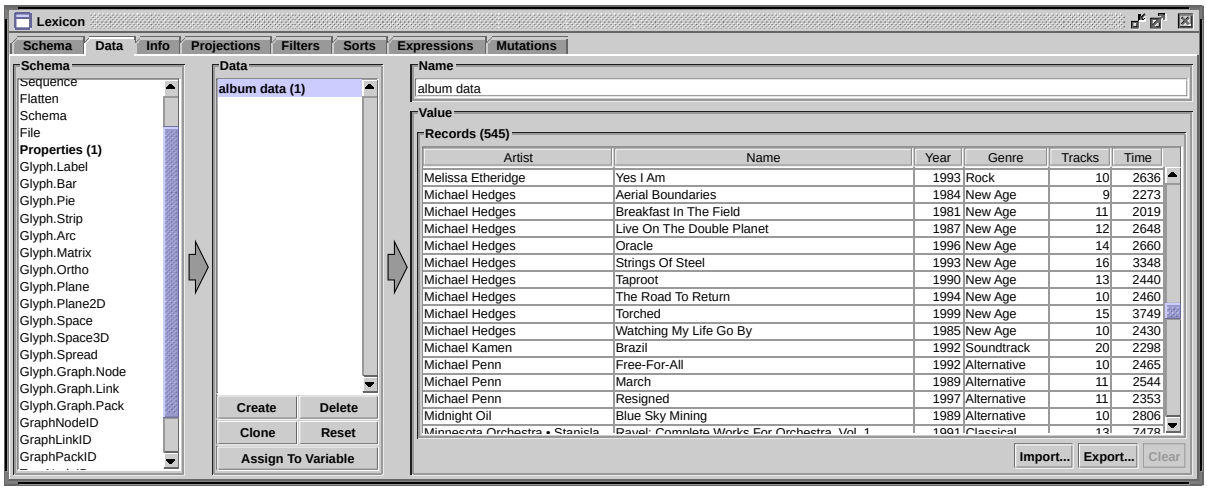



Figure 1: Importing data.

Next, the `Controls` window (figure 2) is used to create a table view, three scatter plots, and four axis controls. The `Frames` window (figure 3) is used to layout the views in two frames: one containing a three-dimensional scatter plot matrix (consisting of the scatter plots and axis controls), the other containing the table view.

To synchronize pan-and-zoom operations in the scatter plot matrix, the user creates range ({`min,max`}) *variables* in the `Variables` window (figure 4), then binds them to the appropriate navigational *properties* of the scatter plots and axis controls using the `Controls` window. At this point, the user can interact with the scatter plot matrix to navigate between

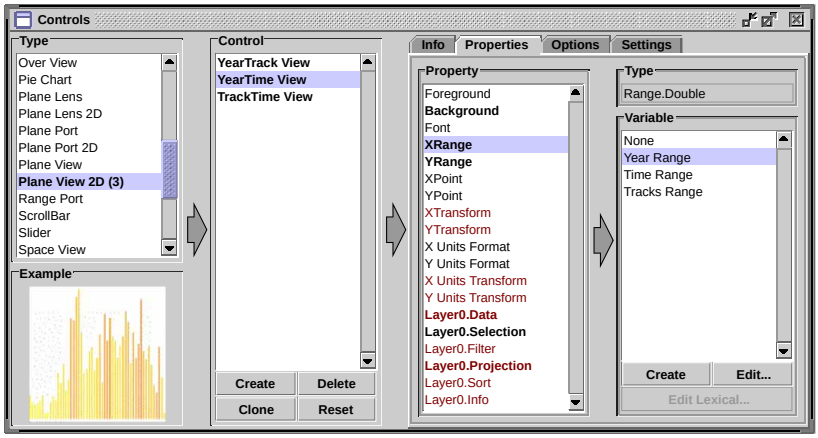



Figure 2: Creating views and other controls.

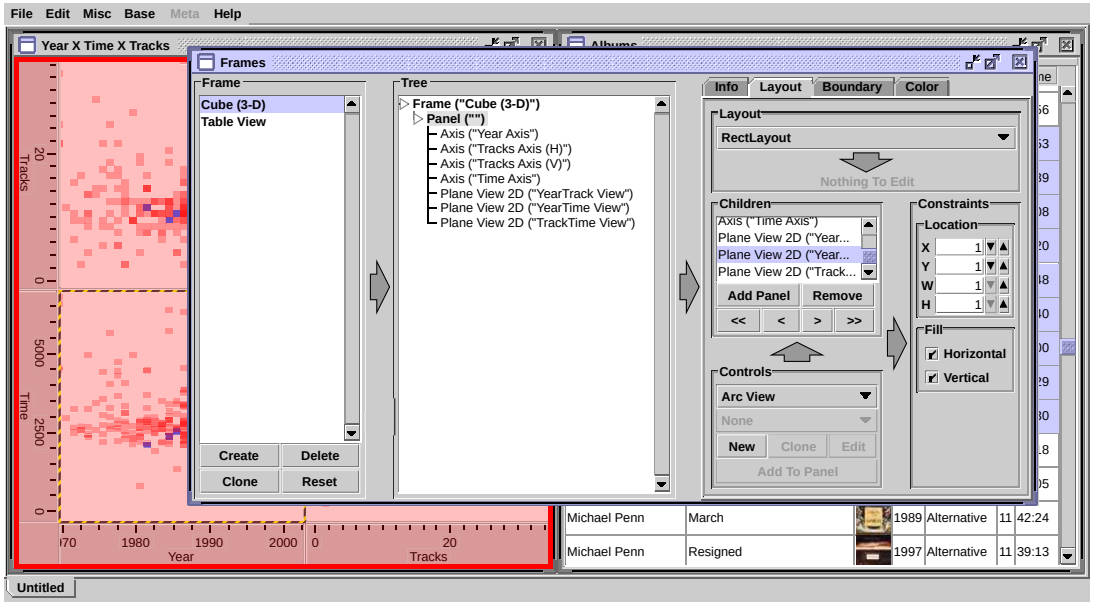



Figure 3: Laying out controls in frames.

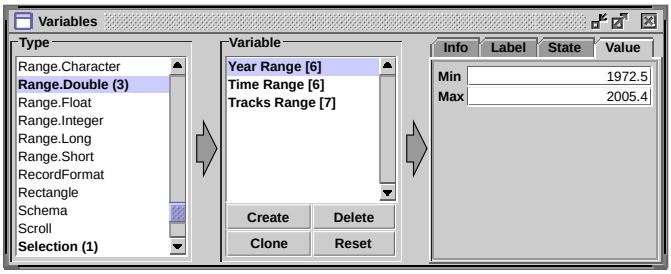



Figure 4: Creating variables and editing their values.

cubic regions in an empty three-dimensional space by changing the values of the three range variables.

The next step is to construct visual encodings that specify exactly how album information is displayed in all four views. Figure 5 shows the projection expression used by the scatter plot of album duration as a function of year (`YearTime View`) to visually encode albums as rectangles. Updates to rectangle color occur whenever the user brushes albums in any view (including `YearTime View` itself), all of which are coordinated through a single shared *selection* variable (`$Selection`) that represents the binary selection state of records in the albums data set. Because the expression is defined in terms of `$Selection`, `YearTime View` draws selected albums as filled blue rectangles.

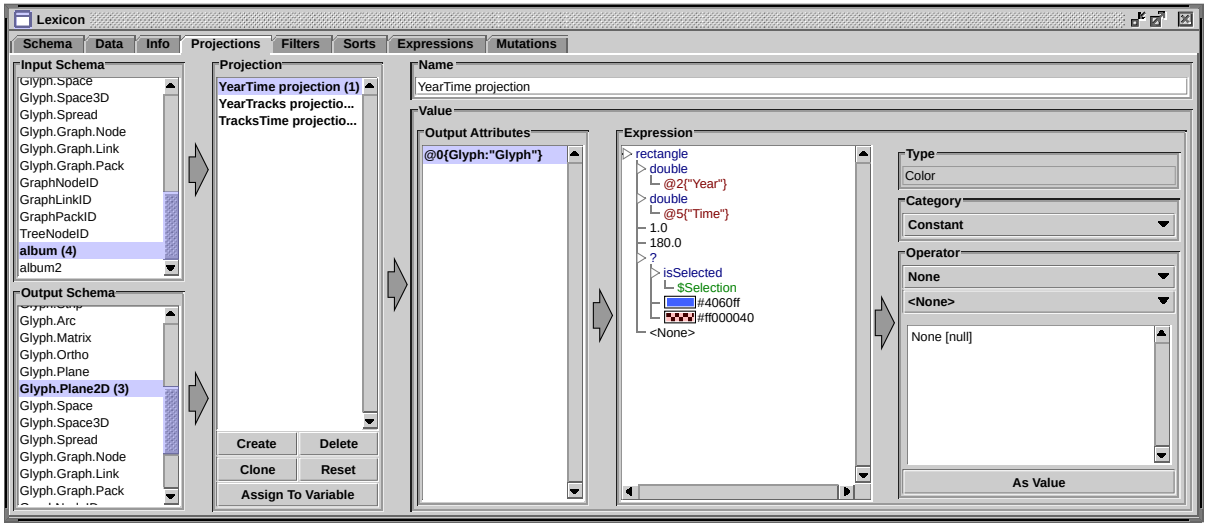



Figure 5: Constructing a projection (visual encoding).

The final step is to construct a filter expression (figure 6) that excludes all albums that fall outside the range of years, durations, and number of tracks visible in the scatter plot matrix. The filter is assigned to a variable which is bound to the table view. As a result, the subset of albums shown in the table view changes as the user pans and zooms in the three scatter plots.

The coordinations that have already been incorporated into this example—multidimensional

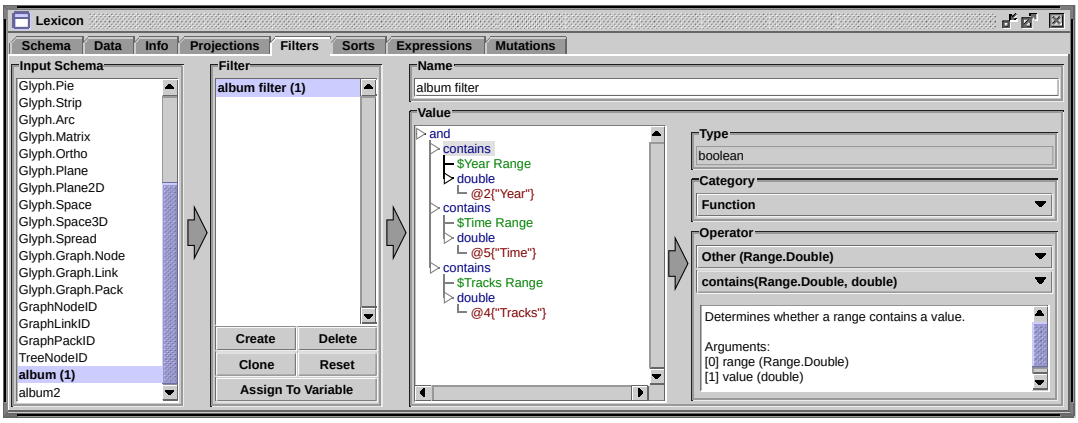



Figure 6: Constructing a filter.

synchronized scrolling, fully connected brushing, and a form of overview+detail in which multiple overviews affect the detail view—are difficult, if not impossible, to recreate in existing visualization tools. Although the visualization at this point (figure 7) is complete enough for browsing albums, the user can continue to build by creating views, editing projections and filters, and coordinating views through variables. A version of the albums visualization with more extensive functionality is presented in appendix A with other example visualizations.

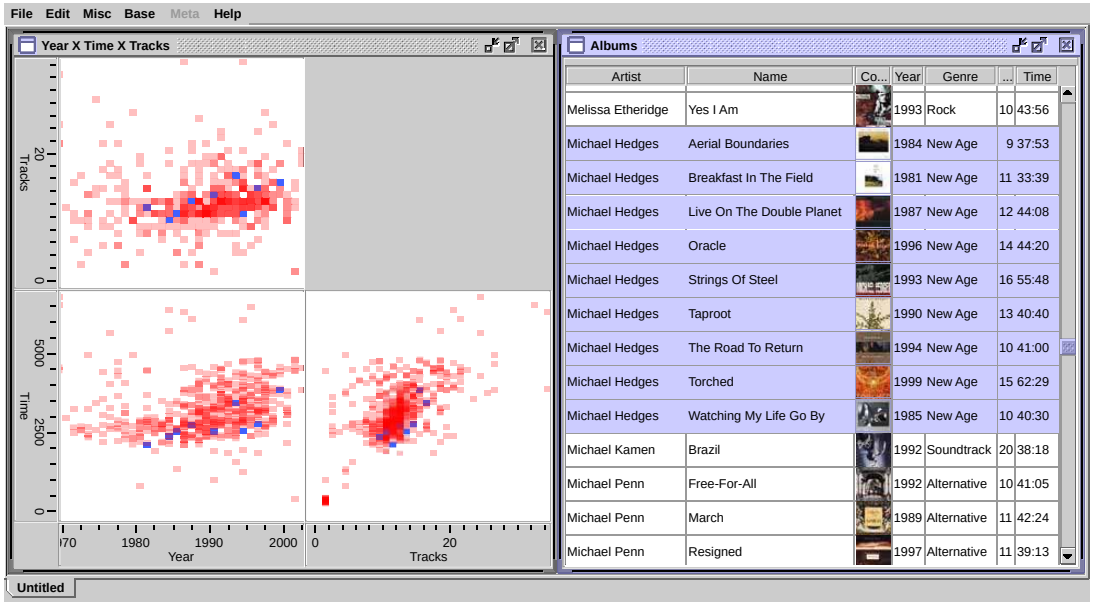



Figure 7: Selecting the albums of a particular artist in the completed visualization.

## 1.3 Thesis Statement and Research Contributions

My thesis is as follows: *By combining coordination, visual abstraction, and data querying in a single coordinated query language, it is possible and practical to build useful visualizations with more views and richer interactive appearance and behavior than in visualization systems in which these functions are independent.*

This dissertation describes four primary contributions to the field of information visualization. The first contribution is a conceptual model of coordination, *Live Properties*, in which views tightly couple through shared objects that describe the data contents and the navigation and selection state of views in a visualization.

The second contribution is a visual abstraction and data querying language, *Coordinated Queries*, loosely based on the relational data model. Views access, aggregate, sort, filter, and visually encode data records by evaluating declarative expressions.

The third contribution is a conceptual model of *metavisualization*, in which the screen layout and coordination structure of multiview visualizations can be explored by visualizing them *in situ*, directly in the running visualizations themselves.

The fourth contribution is *Improvise*, an implemented system in which users build and browse multiview visualizations interactively using Live Properties and Coordinated Queries. By coupling visual abstraction with coordination, users gain precise control over how navigation and selection in a visualization affects the appearance of data in individual views. In the Improvise user interface, users can alter visualizations quickly and incrementally during data exploration, optionally using metavisualization to explore interactive structure as it evolves. As a result, it is possible to build richly coordinated visualizations with many views in Improvise.

## 1.4 Research Questions and Scope

The research presented in this dissertation must answer several questions raised by the thesis.

First, the concept of coordination remains poorly understood, despite many attempts at categorization and formal modeling. Coordinations are generally understood to be interactive dependencies between views. However, the exact semantics of these dependencies vary widely across numerous visualization systems and toolkits. Can coordination be captured in a formal model that is general, flexible, and extensible?

Second, the concept of visual abstraction has been treated as functionally independent from coordination. In existing visualization systems, coordination affects the appearance of data in views, but does so in fixed ways defined by visualization system developers. In what ways can visual abstraction depend on coordination? How can this dependency be expressed in a formal visual abstraction language? What level of programming skill is required to use such a language?

Third, how can users build visualizations interactively? What user interfaces are needed to enable users to create and connect views and coordinations? How do users edit visual abstractions and attach them to views? How much effort is required to build visualizations with many views, extensive coordination, and complex visual abstractions?

Fourth, Improvise focuses on the processing and display of abstract tabular data. As such, it is less concerned with other structured data and image formats targeted by scientific and other spatially oriented visualization applications, as well as with unstructured and semi-structured information targeted by knowledge management and data mining applications. What limits does the relational data model impose upon the range of possible visualization applications? How might coordination and visual encoding be extended to other data models?

There are practical issues that impact the implementation of Improvise, but that are outside

the research scope of this dissertation. Among these are data access, query optimization, visualization persistence, interactive layout of views, and a myriad of minor features that users have come to expect in modern software applications. Although Improvise provides many these features, they are not discussed in detail here.

## 1.5 Organization of the Dissertation

The remaining chapters in the dissertation are organized as follows.

**Chapter 2** provides an overview of visualization in general and coordinated and multiple view approaches in particular, including a survey of important information visualization systems and toolkits.

**Chapter 3** describes the Live Properties coordination model, using a formal graph model of the relationships between visualization building blocks.

**Chapter 4** describes the Coordinated Queries visual abstraction language, using a formal graph model of the relationships between language elements.

**Chapter 5** describes the Improvise user interface and software architecture, using an example visualization of atomic properties in the periodic table of the elements.

**Chapter 6** surveys common patterns of coordination employed in numerous visualization systems, uncommon patterns available in a few systems, and novel patterns previously unseen in visualization, all of which can be reproduced and extended in Improvise using Coordinated Queries.

**Chapter 7** presents the concept of metavisualization, discusses the problem of capturing the

interactive structure of visualizations as visualizable data, describes techniques for integrating metavisualizations directly into visualizations, and explores potential benefits and drawbacks of metavisualization.

**Chapter 8** compares the capabilities of Improvise with those of DEVise in the form of a case study. It enumerates the DEVise coordination primitives, outlines early efforts to explore DEVise visualization structure, gives two examples of Improvise visualizations for exploring the screen layout and coordination structure of DEVise visualizations, and describes how all DEVise visualizations can be reproduced and usefully extended using Coordinated Queries.

**Chapter 9** concludes with an evaluation of the benefits and limitations of Coordinated Queries and its implementation in Improvise, an outline of possible extensions and future work, and a summary of contributions.

# Chapter 2

# Background

## 2.1 Visualization

Visualization is about discovering perceptually and cognitively effective ways to display data graphically. Data comes in many shapes and sizes from a variety of sources. Scientific, geographic, economic, demographic, and other domains of human knowledge produce vast amounts of wildly different kinds of data. The numerous visualizations that have been developed to display all these kinds of data vary widely in style and complexity. Most visualizations can be placed into one of two categories, depending on the kind of data they display and how they display it:

- *Scientific visualization* focuses on the display of predominately spatial data, such as that which describes weather fronts, aerodynamics in wind tunnels, mechanical stress of aircraft, fluid flow, and medical scans of the human body. Because of the large amount of data involved, the user interfaces of scientific visualizations are typically not very interactive. Users specify what they want to see, then wait while the visualization processes and draws the data.

- *Information visualization* focuses on the display of predominately abstract data by mapping attributes into spatial and other perceptual dimensions such as color, shape, and size.

Information visualizations have several qualities that make them effective for data exploration. First, they can display high-dimensional, abstract data in readily understandable ways. Exploring such data is quicker and easier when it is presented in graphical rather than textual form. Second, they can display data from multiple perspectives simultaneously using multiple views. Different views can show different aspects of the data, allowing users to explore the data in alternate ways at a glance. Third, they are usually highly interactive, allowing users to rapidly select which data items to show and how to show them, using the mouse and keyboard to directly manipulate views. Finally, they can coordinate the appearance and behavior of multiple views, allowing users to manipulate visualizations as a whole by interacting with individual views.

Numerous works provide broad insight into visualization in general and information visualization in particular. General discussions of the graphical display of data can be found in the works of Bertin [15, 14] and Cleveland [32, 33]. Wilkinson [154] focuses on statistical and other quantitative aspects of visualizing data. Ware [146] addresses perceptual issues related to the design of user interfaces for information visualization. MacEachren [93] discusses technical and cognitive issues that affect cartographic representation of spatial information in geovisualizations. Shneiderman [130] summarizes visualization techniques, including coordination of multiple views in information visualizations. In his series of books [139, 140, 141], Tufte outlines the principles of visual display and describes methods for using these principles to create explanations visually. Two of the most important principles are: keep visual displays simple by minimizing graphics that carry no information; keep them to the point by minimizing graphics that depict unrelated information. These principles must guide visualization designers who, using interactive construction tools, have extensive control over the appearance of data.

The recent trend toward *visual analytics* [138] is driven by the increasing need to support open-ended management and exploration of large, loosely-connected, and often unstructured information sources as well as the smaller, isolated, structured data sets typical of information visualization applications. Information collection often involves assembling "shoeboxes" of loosely related nuggets and data sets [158]. Visual analysis of information occurs by following chains of evidence, evaluating formal hypotheses [34], testing competing explanations [131], or telling stories [53] using visual metaphors to convey relationships and dynamics. These activities are particularly challenging in intelligence analysis, emergency management, epidemiology, and other critical areas that involve high-dimensional abstract information [125] and large geospatial datastores [51].

## 2.2 Concepts

### 2.2.1 Visual Encoding

A *visual encoding* is a projection of raw data records into graphical attributes [95]. Graphical characteristics that can be usefully employed in visual encodings include location, size, shape, color, rotation, texture, etc. Moreover, many of these characteristics can be decomposed into independent subcharacteristics (size into width and height, color into red-green-blue or hue-saturation-lightness, etc.) The perceptual effectiveness of various graphical characteristics is a widely studied problem (e.g. [108]). Although area, shape, and hue are the most common means to distinguish items visually, effective visual encoding depends strongly on the nature of the attributes of the data being encoded, including whether they are ordinal or nominal, dense or sparse, few or many. Tweedie addresses the various ways of visualizing different kinds of complex-structured data in the form of interactive visualization artifacts [142] and interactive

externalizations [143].

A *view* is a user interface widget that displays data and allows user interaction on that data or the space in which it is shown. The display and interaction characteristics of views constrain how data can be visually encoded inside them. Whereas scatter plots typically allow panning and zooming over a 2-D space in which data items are graphically embedded using location, size, and shape, table views usually allow vertical scrolling and selection of rows drawn with color, text, and icons. Different analysis tasks have motivated the development of numerous views designed to display 1-D, 2-D, 3-D, multidimensional, temporal, tree, and graph data structures [129]. The most common and useful views have been studied and categorized extensively [28]. Although many new and enhanced view implementations are introduced every year, substantial improvement is still needed in the ability of users to rapidly and interactively combine views into whole visualizations that are usable and useful for real visual analysis tasks.

### 2.2.2 Coordination

A *coordination* is an interactive dependency between two or more views. During direct manipulation [127] of a view, such as a mouse drag, coordinated views can either update themselves continuously or wait until the action ceases. Continuous updating between coordinated views is called *tight coupling* [10]. Tight coupling increases the effectiveness of information visualizations for data exploration by increasing the fluidity of interaction and by maintaining consistency between views. For example, a pair of scatter plots can be coordinated using synchronized scrolling so that they always show the same rectangular region of the cartesian coordinate plane.

North and Shneiderman [104] describe a taxonomy of strategies for coordinating multiple

views in terms of navigation and selection. The taxonomy classifies coordinations along two dimensions: (1) the possible combinations of navigation and selection between any pair of views, and (2) whether those views show the same or different data sets. For instance, brushing [9] (also known as brushing-and-linking) is a selection-selection coordination between two views of the same data. Synchronized scrolling is a navigation-navigation coordination between two views that can show the same or different data. Detail-on-demand generally involves a selection-navigation coordination in which selecting items in a view causes another view to navigate to show details about those items.

The difference between selection and navigation is one of objects versus space. Selection interactions are how users identify interesting objects or groups of objects, regardless of the space those objects inhabit. Navigation interactions are how users identify interesting regions of space, regardless of the objects those regions contain. Coordination is a way to tie these two kinds of interactions together in order to use visual space to make sense of the relationships between spatial, temporal, and abstract characteristics of data items. Coordination between multiple views makes it possible to do this even for high-dimensional data.

### 2.2.3 Dynamic Queries

Dynamic Queries [3, 128] are coordinated visualizations that display the results of a database query. Queries are defined in terms of parameters that can be dynamically adjusted using sliders. A central view is tightly coupled with sliders through the query. As a result, the visualization updates whenever interactive modification causes the query to be recalculated. Queries serve to both visually encode and filter items; Shneiderman advocates dynamic filtering by coordinating sliders to views in accordance with an information seeking mantra: “overview first, zoom and filter, then details-on-demand” [129]. Techniques such as fish-eye lenses [49]

that combine overview and details-on-demand in a single view are generally referred to as *focus+context* techniques.

Dynamic Queries extends naturally to visualizations containing multiple views that coordinate with sliders through multiple queries. When views have queries with interactive parameters in common, they coordinate with each other as well. Moreover, parameters can be modified through interaction in the views themselves. Synchronized scrolling, overview+detail, brushing, drill-down, and semantic zoom coordinations can thus replace many or all sliders in a visualization, freeing up display space for richer, multiperspective display of larger data sets. Handling even larger volumes of data in dynamic queries can involve aggregation [54] as well as pixel-based and multi-stage hierarchical techniques [78].

North and Shneiderman [105] have categorized dynamic query visualization systems into four levels which differ by whether users can modify data, views, and coordinations as they work. Level 0 allows modification of none of the three. Each successive level from 1 to 3 allows modification of one additional aspect. Fully capable (level 3) interactive visualization systems generally support modular, extensible view designs and make compound constructions like visualization sliders [41] easy to construct and reuse. Eick [43] summarizes compatibility guidelines for effective visualization interfaces, including: user-centric design; multiple coordinated views; overview, filtering, and details in one or more views; and flexible content rendering and animation.

### 2.2.4 Visual Abstraction

User-centric design in visualization software differs from expert design in integrated development environments and other visual programming software [55], because the number of components tends to be smaller and the connections between them are typically simpler. Specifying

the data, views, and coordinations of visualization interfaces tends to occur at a higher level of abstraction than specifying the data models and interaction handling semantics of general-purpose user interfaces. In particular, visualization construction focuses on specification of *data abstractions* for processing data and *visual abstractions* for rendering data.

Stolte, Tang, and Hanrahan define a visual abstraction as a set of commands written in "a formal specification language for describing table-based visualizations" [133]. The use of such a formal language rather than ad hoc data structures differentiates data processing and rendering in Polaris from other visualization systems. However, visual abstraction in these systems is limited to simple parameterization of algorithms and basic visual encodings in terms of raw data attributes. In Polaris, visual abstractions are specified using a graphical representation of formal language statements; in other systems, attributes are mapped into graphical attributes using form-based interfaces with limited expressiveness. In contrast, data and visual abstractions in Improvise are formal language statements about arbitrary derived attributes across multiple data tables, declared dynamically in a flexible user interface.

## 2.3 Foundation Systems

Like other interactive visualization construction environments, Improvise is a descendant of graphic user interface construction systems that employ visual techniques like programming by demonstration [99]. Demonstrational techniques have been successfully employed in frameworks such as ConMan [60] and Garnet [101]. EAGER [38] augments HyperCard by watching for patterns in end-user interactions, then writing scripts that automate them.

The coordination and visual abstraction models implemented in Improvise build upon strategies for modular separation of user interface from data representation, typified by Model-View-Controller architectures [83]. For instance, Taps [13] separates the data abstraction from

the interface. The data abstraction takes the form of a monolithic data structure mapped directly into the physical form of the interface. Rendezvous [67] connects views in a multi-user interface to one or more abstractions via links that deliver user commands from views to abstractions and change notifications from abstractions to views. Because Rendezvous is intended as a general-purpose user interface construction architecture, abstractions are application-specific black boxes that are generally few in number. In contrast, Improvise was developed to be an interactive information visualization construction environment in which interactive parameters and declarative query language statements are connected to form coordinated query graphs that play a role similar to abstractions in Rendezvous.

Many early user interface development systems provide interfaces to help the user conceptualize interface structure. In ThingLab/Animus [17, 18], a direct manipulation browser shows a graph representation of the user interface. In rendering by example [84], the designer can see application semantics as an overlay on the interface using gestures which induce visual metaphors, such as connecting lines or shared highlighting. Interaction Object Graphs (IOGs) allow the user to specify dependencies within and between widgets using graph-like diagrams [25]. Nodes in the graph are drawn as pictures of widgets in particular interactive states in order to strongly associate screen appearance with logical structure. Animation of the IOG during interaction with the widget strengthens the association. The metavisualization capability of Improvise is a systematic application of these kinds of approaches to visualization interfaces using coordination, visual encoding, and other visualization techniques.

## 2.4 Visualization Tools, Toolkits, and Systems

Hundreds of tools have been developed for visualization of information using multiple coordinated views. The majority of these tools apply a few specific visual techniques to problems

in a particular knowledge domain. As such, most tools are designed and developed to work with specific sources and kinds of data. Some of the earliest visualizations of this sort are the Dynamic HomeFinder [155], LifeLines [118], and the Visible Human [106]. Chi [28] provides a taxonomic comparison of the data, transformations, visual abstractions, and interactions utilized in these and other early tools. These visualizations demonstrate coordination techniques but predate both visualization toolkits that allow developers to program visualization tools quickly, and visualization systems that allow the user to construct visualizations on the fly as they explore arbitrary data sets. In particular, many of these tools could be reproduced in Improvise. As such, Improvise might be an ideal environment for building new user interfaces by creating, coordinating, and laying out views, allowing visualization developers to focus on the visual techniques and data processing algorithms needed to support information from particular knowledge domains. Several other important toolkits and systems are summarized here.

Vz [42] is a library (implemented separately in C++ and Java) of interactive, linkable visualization components that can be combined into monolithic applications and embedded into web pages. Whereas coordination in Vz involves direct linking of views pairwise, coordination in Improvise involves indirect connection of views via paths through a graph of interactive parameters and query operations.

The InfoVis Toolkit [44] is a set of Java visualization components designed around OpenGL and a data model that represents tables, trees, graphs, and metadata in column format for efficient selection, filtering, visual encoding, and coordination. Views include scatter plots, parallel coordinate plots, treemaps, and a variety of node-edge tree and graph displays that can incorporate fisheye lenses and dynamic labeling of items. Visualizations created in the toolkit display textboxes, sliders, and other controls alongside views for dynamic editing of visual encodings. Because it is a toolkit rather than an integrated builder environment, it lacks a complete front end for interactive construction and layout of multiple views like in Improvise.

Dynamic editing of visual encodings by direct manipulation in Infovis Toolkit visualizations is easier and more accessible but much less flexible than by editing query expressions in Improvise builder dialogs. Whereas the Infovis Toolkit can represent, process, and visually encode tree- and graph-structured data, Improvise is currently limited to tabular data sets.

Prefuse [62] is an extensible toolkit for building interactive visualizations from fine-grained building blocks that specify processing algorithms, visual encodings, and coordinations between graph-structured data sets. Prefuse is implemented in Java around Swing and Java2D to serve as a high-level library for development of visualization-specific user interfaces. Views include graphs, hyperbolic trees, treemaps, and scatter plots using a variety of layout and distortion algorithms to place data items. Prefuse and Improvise both visualize information by accessing, filtering, rendering, then displaying data in views. Moreover, in both cases each of these stages can be parameterized in terms of user interactions. However, the details of each of these stages differ. In Prefuse, graph-structured data is filtered through a sequence of composable *action* modules, then is transformed into visualizable node, edge, and aggregate items for rendering in views. Coordination occurs whenever the arbitrary Java code that implements actions and transformations depends on interaction events in views. In Improvise, table-structured data is mapped into other data sets through operations in a declarative query language that can be defined in terms of navigations and selections in views. Views render data sets that have been projected into self-drawing glyph objects. Visualizations in Prefuse have a directed pipeline structure with some feedback from a user interface at the end, as compared with the more general graph structure with extensive bidirectional connections to the user interface in Improvise visualizations. The integrated builder and browser interface in Improvise is a key advantage over Prefuse, one made practical by the home-grown query language and smaller number of visualization component types.

Visage [124, 82] integrates editing and browsing through interactions that project data attributes into graphical attributes. All graphical elements (including top-level frames) are accessible, first-class data objects. Views are coordinated using drill-down, highlight, and copy operations. In contrast with the declarative query language used in Improvise to specify the effect of coordination on visual abstraction across arbitrary types of views, Visage uses a procedural scripting language to specify visual encodings and the semantics of editing-style interactions (such as copies) in predominantly 2-D and 3-D views. Like Improvise, Visage appears to be well-suited for metavisualization. First-class representation of all visualization objects makes it possible to incorporate metavisual aspects directly into Visage visualizations, but may inhibit separability and reusability of metavisualizations by allowing dependencies between visual and metavisual elements.

VisAD [64] enables users to specify visual abstractions by defining mappings from data attributes into numeric display primitives used to draw pixels and voxels, including 2-D/3-D location, color, transparency, and animation. Mappings can be used for glyph-based as well as pixel-based rendering. Sliders and other controls can be attached to display dimensions to perform filtering over subsets or subranges of displayed values. A Java version of VisAD [65] shifts the focus from an end-user interface (in form of a mapping dialog as in other systems) to a toolkit API. VisAD implements a data flow and coordination model in which changes to data propagate to other views through computations written in conventional C/Java code. As a result, there are several key differences between VisAD and Improvise. Whereas Improvise visualizations are built around tabular data representations of any object type, VisAD visualizations are built around complex representations of predominantly numeric types. In Improvise, navigation and selection can effect data access, filtering, and sorting as well as preprocessing transformations and visual encodings, although only of tabular representations. Although animation in visual encodings is possible in Improvise—using a background thread to mimic

stepped navigation along a time dimension by the user—it is harder to incorporate into visualizations and slower in performance than in VisAD. Improvise also trades off performance for analytic immediacy in terms of coordination and visual abstraction by focusing on dynamic editing of an interpreted language rather than on offline editing of a compiled language as in VisAD. Unlike the simple mapping user interface provided by VisAD in its C incarnation, Improvise provides a full builder and browser interface at a high level of design abstraction, trading off some flexibility for accessibility by users with little or no programming experience.

Users visualize high-dimensional data in XGobi [135, 21] by arranging, linking, and focusing scatter plots of points, lines, or glyphs. These three activities operate on three structural aspects of visualization: screen layout, coordination structure, and the interactive state of visual encodings. GGobi [136] is a redesigned version of XGobi that supports multiple plot windows and richer statistical data analysis functionality using the R statistical data processing language [71]. Although Improvise has minimal built-in statistical processing capabilities and currently has no connections to external data processing libraries, it provides a larger (and extensible) library of views that can render data as glyphs and other graphics in much richer ways by allowing visual encodings to be defined in terms of coordinations.

LinkWinds [73] is a multi-application visualization environment for visual analysis of primarily scientific data. Data-linking serves to coordinate multiple interface frames. Non-view controls append clauses to queries for filtering purposes. Tight coupling of coordination is supported by an optional "track" mode in which mouse drags continuously update filtering of views. Views in Improvise are always tightly coupled over all coordinations, including all coordinated queries that specify data sets as a function of interactive parameters. Moreover, Improvise views can coordinate with each other as well as with non-view controls.

Given an input table, IVEE [4] automatically selects appropriate controls (such as range sliders or checkboxes) for each data attribute. Users can create one or more views (scatter

plots, geographic maps, or cluster views) and specify simple mappings of attributes into view parameters. The conjunction of slider selections is used to filter the contents of all views as well as the contents of the sliders themselves. In Improvise, views can be filtered independently using filter expressions that depend on navigation or selection in any combination of sliders or views.

Tioga-2 [157] uses a data flow model to support advanced navigation features such as tunneling (wormhole-like hyperlinks), view cloning, and nested views. DataSplash [110] adds end-user visualization construction in the form of tuple painting and a zoom layer manager for editing how tuples appear at different levels of magnification. VIQING [109] is an extension of DataSplash that allows users to express queries by conjoining views into the visual equivalent of database projections, selections, and joins. By comparison, visual abstraction in Improvise is declarative rather than procedural; users can generate nested views and semantic zoom, but features like tunneling and layer management have to be built into view implementations.

Snap-Together Visualization [107] uses a relational data model that coordinates views using *primary key actions*. When two views are coordinated, invoking an action in one view causes the other view to perform its corresponding action. Actions are extensible and include loading (of a relation), selection (of tuples), and scrolling (over a list of tuples). Visualization schemas [103] display the structure of Snap visualizations using a graph to show relationships between views and actions. Coordinated Queries create similar interactive dependencies between views, but allow fine-grained user customization of dependencies between visual encodings as well as data.

GeoVISTA *Studio* [137] is an integrated visualization development environment for building geovisualizations interactively using a graph-based visual coordination editor. Any component that conforms to the JavaBeans specification can be a view. Development of new views by the community of GeoVISTA *Studio* users has resulting in a large library of views utilized in

numerous visualizations. Improvise and GeoVISTA *Studio* have opposite strengths. Whereas GeoVISTA *Studio* has extensive functionality for representing and displaying geospatial information (based on the GeoTools [89] open source Java GIS toolkit), all but the simplest patterns of coordination can be hard to specify. Conversely, Improvise has limited geospatial data handling capability (e.g. using scatter plots to draw geographic regions as simple polygons), but allows interactive construction of a variety of coordination patterns.

Polaris [134] automatically generates multiscale visualizations and the queries needed to draw them using a formal specification language. Zooming in Polaris is conceptually equivalent to traversing an edge of a *zoom graph* [133] in which each node corresponds with a particular visual representation in a data cube. Nodes are drawn in a graphical notation that describes the visual query at that point in the graph. Because Improvise users build coordinated queries using custom-defined expressions, complex or unusual visual representations are possible, but simple or common ones cannot be manipulated as quickly or as easily as in Polaris.

In the coordination model prototyped in CViews [19], explicit *coordination objects* in a *coordination space* manage visual parameters and access data using a data flow model to define a particular type of coordination, such as brushing. Views connect to coordination objects through translation functions. The equivalent space in Improvise consists of coordinated query graphs that connect views through navigational parameters, selections, data, and expressions. In this space, coordinations exist implicitly as recognizable patterns of interactive dependence between views, rather than as explicit objects.

DEVise [90] uses a relational data model to coordinate multiple views of large data sets [91]. Users can create, destroy, coordinate, and specify the contents of views interactively. Its only view—the scatter plot—and few coordination types—*cursor*, *visual link*, *record link*, and *set link*—are quite powerful. However, reproducing common visualization constructions in DEVise frequently involves convoluted chains of coordinated scatter plots (many of which are

undesirable artifacts that must be intentionally hidden off screen). Metavisualization of DE-Vise visualizations reveals that all four coordination types can be reproduced by treating the `X` and `Y` ranges of scatter plots as shared objects or as interactive parameters in simple query expressions. This discovery motivated the design of Live Properties and Coordinated Queries.

# Chapter 3

# Model of Coordination

## 3.1 Overview

Coordination is a key feature of many information visualizations that have multiple views. Improvise visualizations are based on a model of coordination called *Live Properties*. The goal of Live Properties is to provide a general foundation for flexible specification of interactive dependencies between multiple views. To achieve this goal, Live Properties not only must be compatible with common forms of interaction, kinds of views, and forms of coordination, but also must support:

- design of novel view types,
- customization of how views depend on coordinations,
- discovery of novel coordination types,
- extrapolation of coordination types to distinct, useful variants, and
- composition of coordinations.

Several interrelated observations motivated the development of Live Properties. First, coordination is a function of visualizations as a whole. The appearance and behavior of each view in a visualization is determined by the combined interactive state of all views in that visualization. Coordinations need not be limited to pairwise relationships between views.

Second, coordination is an interaction leveraging problem. That is, coordination is about giving users interactive control over parameters that affect the overall appearance and behavior of a visualization. In visualizations, users manipulate these parameters by interacting with views and other controls. Designers incorporate views into visualizations not only to show data, but also to allow manipulation of parameters that determine where and how data is drawn.

Third, coordination is a visual coherence problem. A visualization is a graphical projection of a high-dimensional abstract data space. A view shows a region of this space, and displays data in that region. Because each view shows particular data in a particular region, it is both data and space that are coordinated in visualizations. Coordination enforces visual coherence between views that share spatial dimensions or data attributes.

## 3.2 Mediation Model of Coordination

Live Properties is designed to overcome limited generality in previous approaches to coordination. These approaches fall into five categories:

- *Raw Inputs.* Mouse and keyboard inputs in each view are translated into corresponding inputs in other views.

- *Events.* Views describe internal changes to interested listeners through callbacks. Mouse and keyboard inputs in each view are translated into semantic events for notification of other views.

- *Constraints.* Dependencies between views are defined by a system of mathematical equations. Solving these equations determines how views appear and behave in response to user input.

- *Filters.* In a network of views, interaction in a view filters the data shown in other views.

- *Links.* Views are coordinated pairwise using a small set of predefined link types. Links translate navigation and selection in one view into placement and highlighting in the other view.

Live Properties is based on the Mediator software pattern [52], in which objects communicate with each other indirectly through intermediaries. The Mediator pattern provides several advantages for coordination models:

- decoupling of views,
- abstraction of how views coordinate,
- simplification of notification protocols, and
- centralization of control over coordination.

These qualities provide several benefits for the Improvise software architecture:

- simplification of view design and implementation,
- separation of design from implementation for views and coordinations,
- reduction of the need to subclass views to handle different user inputs,
- automatic tight coupling of views, and
- absence of cycles in coordination graphs.

They also provide several benefits to visualization designers and users:

- a clean conceptual separation between views and coordination,
- simplification of view instantiation and manipulation,

- centralization of coordination instantiation and manipulation,
- increased coordination flexibility,
- support for coordinations that involve more than two views, and
- multiple avenues for manipulating visualization parameters.

### 3.2.1 Live Properties Overview

Live Properties is a user interface architecture for directly coordinating *controls*—including views, sliders, and other widgets—through shared objects called *variables*. Each control defines one or more *properties*, each of which can bind to at most one variable. Changes to variables are propagated to controls via their properties, as shown in figure 8.

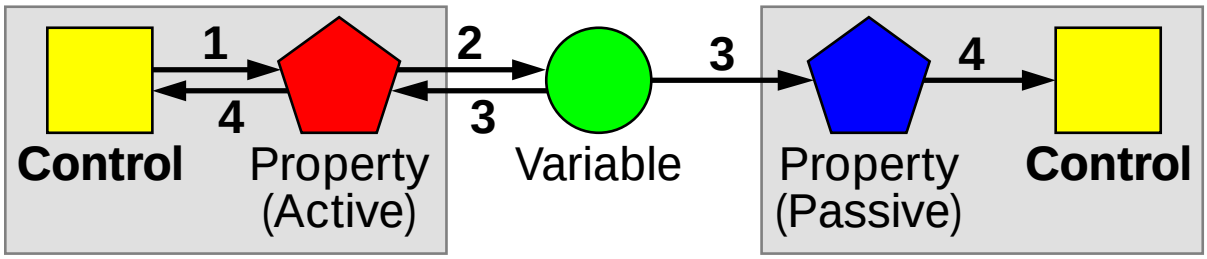



Figure 8: Direct coordination. (1) A control modifies the value of one of its (active) properties in response to interaction. (2) The property assigns the new value to its bound variable. (3) The variable sends a change notification to all properties bound to it. (4) The properties notify their respective parent controls of the change. The controls update themselves appropriately.

Properties serve two purposes. First, they are value slots that controls use to determine their appearance and behavior. For instance, a scatter plot has two range properties that specify which region of the cartesian coordinate plane to show, and a color property that specifies a color used to fill its background. Second, properties act as ports through which controls communicate with each other as a result of interaction.

Live Properties may be thought of as an instance of the Model-View-Controller architecture [83] with many small models. Similarly, the Abstraction-Link-View paradigm (ALV) [66] employs an encapsulated communications mechanism between views and data to link views shared by one or more users.

### 3.2.2 Model Components

Visualizations built with Live Properties consist of three kinds of components:

- *Variables* are named, typed value wrappers. Each variable may be assigned a value, which must be compatible with its type. If a variable has no assigned value, it uses a default value instead. Each variable also has three flags that indicate whether or not the variable is in focus, is being actively edited, or is locked to prevent interactive changes.

- *Properties* are named ports in controls. A property can bind to a variable of the same type. If bound, its value is the value of its variable. If not, it uses a default value instead.

- *Controls* are views, sliders, and other interface elements. Each control is defined by a list of properties. Each property determines some aspect of interaction, including basic appearance (background color, foreground color, label font, etc.), region of data space displayed (`X` and `Y` ranges, etc.), data contents (data source, visual encoding, ordering, etc.), and data item selection. In Live Properties, a *view* is a control that has at least one data-related property.

Controls can change properties dynamically. Some views allow users to mix and match multiple spatial regions or data layers during exploration by adding or removing (sets of) properties. For example, scatter plots display one or more data layers in a single rectangular spatial region; adding a data layer to a scatter plot involves adding data, visual encoding, filtering,

ordering, and selection properties for that layer. Most controls build an unchanging list of properties when they are first instantiated. (By design contract, controls may change properties in response to building-level activities, but should never add or remove properties in response to browsing-level activities, particularly coordination during normal interaction.)

Variables and properties are defined in terms of a *prototype* that consists of an object type, a default value, and a default name. Prototypes enforce legal property-variable bindings through strong typing. They also ensure that unbound properties possess default values for use by their parent controls during updates.

Properties may be either *active* or *passive*. An active property can be modified by its control, such as in response to keyboard or mouse interaction. A passive property can be accessed by its control, usually for the purpose of determining aspects of appearance, but cannot be modified by it. In other words, an active property can access and modify its variable (if bound), but a passive property can only access its variable. This restriction applies to modification of a variable's name, focus flag, and edit flag, as well as to its value.

For example, axis controls—visualization sliders used to select a range of floating-point values—have seven properties (table 1). An axis redraws itself to show its current range whenever a variable bound to one of its properties changes. Mouse drag and keypad interactions in the axis change the value of the variable bound to its `Range` property. If no variable is bound

| *Category* | *Name* | *Type* | *Access* | *Description* |
|---|---|---|---|---|
| Navigation | `Range` | range | active | Visible extent |
| Navigation | `Point` | double | active | Location of mouse |
| Look&Feel | `Background` | color | passive | Background fill color |
| Look&Feel | `Foreground` | color | passive | Label/Unit/Tick color |
| Look&Feel | `Font` | font | passive | Label/Unit font |
| Look&Feel | `Units Format` | string | passive | Numeric form of units, e.g. `0.00E0` |
| Look&Feel | `Label Format` | string | passive | Label, incorporating name of `Range` |

Table 1: Properties of Improvise axis controls.

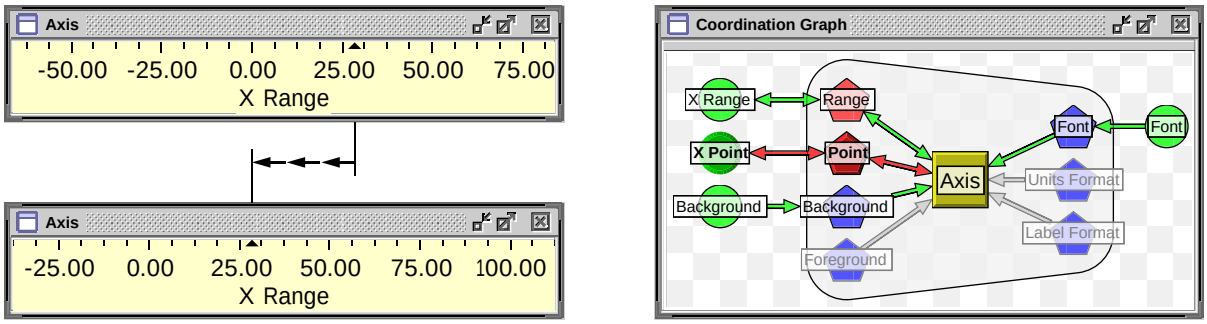



Figure 9: Manipulating a numeric range by dragging the mouse toward the left in an axis control. The drag point is marked by a small triangle that tracks the axis value at the current mouse location. The coordination graph shows how the axis is bound to several variables through its properties.

to its `Range` property, the axis does not respond to drags, making it effectively non-interactive. Mouseover movement similarly affects the `Point` property. Figure 9 shows panning navigation in an axis control.

### 3.2.3 Notification Protocol

Controls access and modify variables through their properties. Whenever the value of a variable is modified, all controls having properties bound to that variable are notified by callback. As a result, two or more controls can be connected to each other through variables bound to their respective properties.

A three-step sequence of events triggers updates in Live Properties:

1. The user interacts with a control.

2. The control assigns new values to its properties.

3. Each property assigns its new value to its bound variable.

A three step notification sequence occurs along the reverse path:

1. When a variable changes, it notifies all properties bound to it.

2. When a property changes, it notifies its parent control.

3. Notified controls update themselves appropriately.

Changes to a variable's name, focus flag, or edit flag also trigger the update mechanism. Notifications carry information about which kind of change occurred, so that individual controls can determine which changes affect them. In an axis control, for example, the label depends on the name of the `Range` property.

Notification also occurs in response to dynamic changes in the structure of a coordination graph during visualization construction. For instance, controls are notified of property-variable binding and unbinding events just as they are for variable change events. In the case of unbinding, a control updates itself using the default value of the property rather than the last known value of the previously bound variable.

The symmetry between update and notification means that the results of interaction in controls echos back to them. This approach makes the design and implementation of controls much simpler than in asymmetric approaches by:

- Defining internal processing in terms of external coordination. Upon notification, controls access their properties to determine what, where, and how to update themselves.

- Isolating rendering from interaction. Upon interaction, controls modify their properties by translating inputs into value changes, but need not update themselves immediately.

### 3.2.4 Implementing Views

Controls are implemented by defining their behavior (response to interaction) and appearance (rendering of space and data) in terms of properties. Properties typically fall into four categories:

- Navigation properties (active) delimit a region of multidimensional space. Users choose and change regions through view-dependent gestures.

- Information properties (passive) point to data sets and instructions for rendering them. By definition, all views have at least one set of information properties; other controls have no information properties. Layers, visual joins, and other forms of multiple table display can be achieved using additional sets of information properties.

- Selection properties (active) indicate user preference for particular data items over others. Views typically have one selection property for each set of information properties, i.e. for each data property. Selection is bijective with (one-to-one and onto) data items.

- Look&Feel properties (passive) determine appearance and behavior characteristics that are independent of navigation, information, and selection. By default, all controls have foreground, background, and font Look&Feel properties.

The process of designing and implementing a new control involves specification of its desired presentation and interaction capabilities in terms of an appropropriate set of properties. For instance, scatter plots in Improvise are designed for continuous updates during dimensionally-uncoupled panning, zooming, and mouse-tracking over multiple layers of data items that can be selected by additive click, rubber band, and lasso gestures. Scatter plots implement these capabilities using properties from all four categories (table 2).

Users can add, remove, and reorder scatter plot layers dynamically. Each layer has an independent set of information and selection properties. Figure 10 shows interaction in a scatter plot with one layer (`#0`). The view accesses its `Data` and `Projection` properties to render data items as shapes. As the user selects with a lasso, the view updates the value of its `Selection` property to include data items corresponding to shapes fully contained within the lasso. Zooming causes the values of the `XRange` and `YRange` properties to scale.

| *Category* | *Name* | *Type* | *Access* | *Description* |
|---|---|---|---|---|
| Navigation | `XRange` | range | active | Visible horizontal extent |
| Navigation | `YRange` | range | active | Visible vertical extent |
| Navigation | `XPoint` | double | active | Horizontal location of mouse |
| Navigation | `YPoint` | double | active | Vertical location of mouse |
| Look&Feel | `Background` | color | passive | Background fill color |
| Look&Feel | `Foreground` | color | passive | Mouse-track line/text color |
| Look&Feel | `Font` | font | passive | Mouse-track text font |
| Information | `Data#0` | data | passive | Source of data items |
| Information | `Projection#0` | projection | passive | Instructions for drawing data |
| Information | `Filter#0` | filter | passive | Instructions for eliding data |
| Information | `Sort#0` | sort | passive | Instructions for z-ordering data |
| Selection | `Selection#0` | selection | active | List of selected data items |

Table 2: Properties of an Improvise scatter plot with one layer.

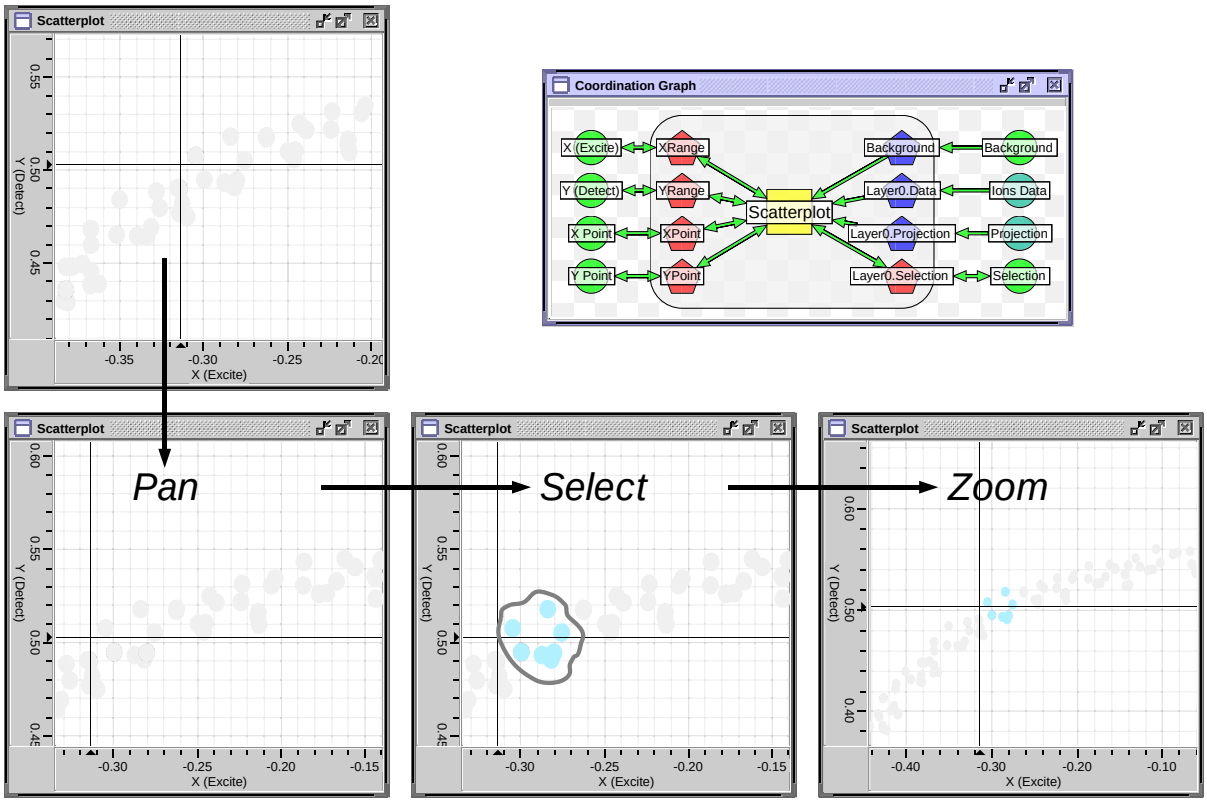



Figure 10: Panning navigation, lasso selection of data items, and zooming navigation in a scatter plot. Crosshairs mark the mouse location during all three gestures. The coordination graph shows bound properties used by the scatter plot to redraw itself during user input. (The `X` and `Y` axes are independent controls that are coordinated with the scatter plot.)

Notification occurs asynchronously in implemented views, by scheduling each variable or property change notification on the same event thread which handles user input and screen redraw operations. This approach increases overall interactivity and reduces the possibility that individual controls, when notified as a result of user interaction, can impede further user interactions by performing expensive operations immediately. (Most views in Improvise, including scatter plots, offload expensive data processing and rendering operations to other threads anyway.) By implementing controls so as to change their properties only in response to local user interaction, potential cycles and deadlocks are avoided.

### 3.2.5 Coordinating Views

Two controls are coordinated whenever they are connected through their properties via at least one variable. For any given shared variable, the coordination is bidirectional if both properties are active. Because more than two controls can share the same variable, coordinations can be multidirectional, i.e. transitively bidirectional. It is not necessary to connect multiple views pairwise to create transitively closed coordinations like in other systems [90, 105].

Controls, properties, and variables can be created and destroyed at runtime. Variables can be bound to and unbound from properties on the fly. This means that a visualization interface, as a collection of controls and variables, can be freely modified and even swapped out entirely during data exploration. As a result, extensive coordination is easier to incorporate into visualizations in Improvise than in other systems and toolkits.

## 3.3 Graph Model of Coordination

Coordinated multiple view visualizations built using Live Properties consist of sets of controls (C), properties (P), and variables (V) connected by slot (S) and binding (B) relationships:

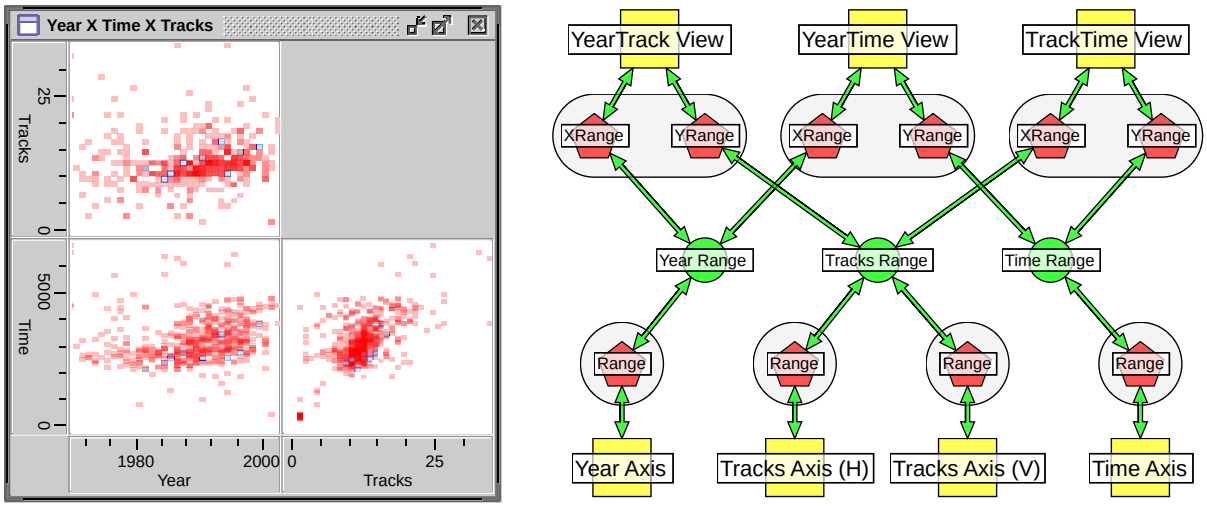



Figure 11: Coordination graph for a 3-D matrix of scatter plots with axis controls. Each of the twelve paths through the graph corresponds with a navigational coordination between views.

$$
\begin{aligned}
CMV &= (C, P, V, S, B),\ where \\
C &= \{c_0, \ldots, c_{N_C}\}, c_i = (type, name, p_i ... p_M), p_i \in P \\
P &= \{p_0, \ldots, p_{N_P}\}, p_i = (type, name, \{v | null\}, default), v \in V \\
V &= \{v_0, \ldots, v_{N_V}\}, v_i = (type, name, focused?, editing?, locked?) \\
S &= \{s_0, \ldots, s_{N_S}\}, s_i = (c_j, p_k) \\
B &= \{b_0, \ldots, b_{N_B}\}, b_i = (p_j, v_k)\ where\ p_j.type = v_k.type
\end{aligned}
$$

In this *coordination graph*, nodes are controls, properties, and variables. Edges are slot and binding relationships. Because controls are not connected directly to each other, coordinations do not correspond to particular objects in the graph. Instead, coordinations correspond to paths of form $c \leftrightarrow p \leftrightarrow v \leftrightarrow p \leftrightarrow c$ through the graph. For example, figure 11 shows the coordination graph for a scatter plot matrix (excerpted from the visualization in figure 7).

The topology of coordination graphs constrains the design and operation of multiview visualizations: variables are at the center of star-shaped subgraphs with controls at the points; controls are at the center of star-shaped subgraphs with variables at the points. Coordination graphs based on Live Properties are:

- *Reflective.* Controls act as if they coordinate with themselves through each of their properties. [$c \leftrightarrow p \leftrightarrow v$]
- *Reflexive.* Controls can also coordinate with themselves through different properties connected to a common variable. [$c \leftrightarrow p_a \leftrightarrow v \leftrightarrow p_b \leftrightarrow c$]
- *Symmetric.* Coordination between controls is bidirectional. Modification of a variable by either control triggers notification of the other control. [$c_1 \leftrightarrow p_a \leftrightarrow v \leftrightarrow p_b \leftrightarrow c_2$]
- *Closed.* Controls that share a variable are coordinated pairwise. Modification of a variable by any control triggers notification of all other controls. [$c_i \leftrightarrow p_a \leftrightarrow v \leftrightarrow p_b \leftrightarrow c_j \; \forall \; c_i, c_j \in C', C' \subset C$]
- *Intransitive.* Although variable sharing is transitive, coordination is not. [$c_1 \leftrightarrow c_2, c_2 \leftrightarrow c_3 \not\Rightarrow c_1 \leftrightarrow c_3$]
- *Acyclic.* Users initiate all property/variable modifications by interaction in views. By design contract, controls never programmatically modify properties/variables in response to change notifications. [$v_1 \rightarrow p_a \rightarrow c \not\rightarrow p_b \not\rightarrow v_2$]
- *Composable.* Graphs can be combined (possibly with overlapping nodes and edges) to build large visualizations from smaller ones. Subgraphs can be isolated to extract small visualizations from larger ones.

Because Live Properties has these mathematical properties, it is possible to redefine the term "link"—as it used to describe coordination in other visualization systems and toolkits—as *a useful combination of coordinations between a pair of views*. In particular, these combinations reproduce the functionality of common types of interactive dependencies between views in other systems. It thus becomes possible to understand common link types as *patterns of coordination* that occur in multiview visualization architectures. Each "coordination" is defined by harmonious action between views in terms of navigation and selection gestures that lead to transitions between states in a high-dimensional abstract data space, much like the Data State Model described by Chi [28].

## 3.4 Limitations

### 3.4.1 Constraints

Live Properties is not a constraint-based interaction model like the ones used in ThingLab [17], Garnet [101], and similar user interface construction environments. The absence of constraints in Live Properties eliminates the need for a constraint-solving engine, the continuous operation of which would degrade interactivity. However, it means that several common, useful restrictions on interaction cannot be reproduced using Live Properties. Instead, it is necessary to customize views or add "workaround" properties to achieve common constraint semantics. For example, a scatter plot can have a fixed aspect ratio (such as for drawing a geographic map) by defining it in terms of center point and zoom factor properties rather than two range properties. Other constraints could be implemented as special object types having limited value domains. For example, bounded ranges consisting of four floating point values $min, lo, hi, max$ could be used to limit the explorable extent of cartesian space views like scatter plots and axes. In

general, constraints would take the form of a reduction in the number of interactive degrees of freedom inherent in the active properties of particular views.

### 3.4.2 Granularity

For visualization system developers using Live Properties, the central design challenge is parameterizing the desired set of interactive states of each control (in terms of its appearance and behavior) as a set of objects suitable for encapsulation in properties. Parameterization of even simple controls can be highly ambiguous. When spatial parameters are involved, there are often many valid combinations of objects/properties that can be used to capture navigational states. For example, the `Range` property in axis controls could be replaced with independent `Minimum` and `Maximum` properties. The `XRange` and `YRange` properties of scatter plots could be replaced similarly with independent `XMin`, `XMax`, `YMin`, and `YMax` properties. They could also be replaced with a single `Bounds` property.

Several factors influenced the choice of properties in the views that have been implemented so far. First, it is necessary for different kinds of views to be defined in terms of properties of the same granularity in order to recreate common coordination patterns. For instance, it is not currently possible to coordinate scatter plots defined in terms of one rectangle property with axis controls defined in terms of two numeric properties.

Second, there is a trade off between granularity of spatial objects and flexibility of coordination. Finer-grain objects allow a wider variety of coordinations at the cost of increasing the complexity of the most common and useful coordinations. For instance, defining scatter plots and axis controls using numeric properties would allow zero-bounded horizontal navigation (by leaving the plot `XMin` and axis `Minimum` properties unbound with default value zero). However, synchronized scrolling—the most common navigational coordination—would necessitate

connecting each extra pair of properties through an additional variable, thereby multiplying the size of the coordination graph.

Third, finer granularity leads to extra, often redundant events in the notification protocol. For example, two scatter plots designed using four numeric properties would spawn four times as many events during diagonal pan and zoom gestures as the same scatter plots designed using one rectangle property. (In the implementation, views coalesce redundant notification events to avoid unnecessary updates. Although notification events tend to arrive together and in order, imperfect foresight sometimes compels views to initiate updates before all redundant events arrive. For instance, it is impossible to distinguish an isolated event describing a horizontal zoom from one of a pair of events describing a diagonal zoom. Interactivity suffers if views wait more than a short period—around 100 milliseconds [128]—in anticipation of future redundant events.)

Range properties provide a good balance of granularity and flexibility in scatter plots, axis controls, parallel coordinate plots, and other views that involve spatial navigation within a finite rectangular region. Chapter 6 describes how all common navigational coordination patterns can be reproduced using range properties. Chapter 8 describes how every DEVise link type can be similarly reproduced.

## 3.5 Extensions

### 3.5.1 Type Composition

Property and variable types are limited to primitives (e.g. int, float, String) and object classes (e.g. Range, Color, Data, Selection) supported by the visualization system implementation. One extension would be to allow visualization designers to define new types by composing

existing types, such as a `Rectangle` type composed of two `Range` types. Operations on variables would be the same for user-defined types as for built-in types.

Views provided by a visualization system implementation are necessarily defined in terms of built-in object types supported by that system. In order to connect properties of views to variables of user-defined types, it would be necessary to allow non-one-to-one binding across compositions. A variable value could bind to a member of a property value, or a property value could bind to a member of a variable value. For instance, horizontal navigational coordination between a scatter plot and an axis control could be constructed as $c : Plot \leftrightarrow p : rectangle \leftrightarrow v : (x : range, y : range), c : Axis \leftrightarrow p : range \leftrightarrow v.x$.

This approach would reduce the granularity problem at the cost of increasing the overall complexity of coordination graphs, by multiplying the number of binding relationships. The coordination graph without composed types would be equivalent to a graph with composed types in which each composed variable/property were split into a variable/property for each member type. Recursive composition of user-defined types would increase complexity further.

As later chapters will show, it is possible to built a rich variety of useful visualizations by designing views around value types of appropriate granularity. It is unnecessary to resort to composable types and the increase in structural complexity that would make building visualizations more difficult.

### 3.5.2 Value Transformation

Interposing transformation functions between properties and variables would make it possible for visualization designers to fine tune the appearance and behavior of views by specifying exactly how each property depends on its bound variable. Whenever a view accessed one of its properties, that property would silently pass the value of its bound variable through the

forward transformation function, producing a derived value of the same type. Whenever a view modified one of its properties, that property would have to assign its bound variable the value generated by passing the modified value through the inverse transformation function.

Most transformation functions are not invertible. In constraints architectures, constraint solvers exist to resolve large sets of bidirectional relationships [17, 101] that may include non-invertible functions. In Live Properties, the problem of invertibility could be addressed by avoiding it in the first place [79]; that is, by disallowing the use of transformation functions by active (read/write) properties. Passive properties (read-only) could use transformation functions without issue.

In Athena MUSE [68], the global parameters shared by views are integer values in bounded ranges ("multidimensional information"). Bidirectional equality constraints link parameters to view attributes through reversible linear functions ("declarative constraints"). Similarly, views in CViews [19] connect to coordination objects through translation functions. Variable values in Live Properties are points or regions in a multidimensional space of interaction, but are not limited to bounded integers. Bindings between variables and properties are like declarative constraints or translation functions limited to the identity function. In fact, Live Properties is a proper subset of constraint-based coordination approaches in general.

## 3.6 Summary

Live Properties provides a simple, elegant basis for coordination of multiple views. Coordination takes place through shared parameters that determine how and where views display data. Live Properties is the foundation for Coordinated Queries, the visual abstraction language used to build highly-coordinated visualizations in Improvise.

# Chapter 4

# Model of Visual Abstraction

## 4.1 Overview

Visual abstraction of data is the central functional purpose of all visualization software, regardless of whether it is implemented in a toolkit, integrated development environment, or custom end-user application. Improvise views process and render data based on a model of visual abstraction called *Coordinated Queries*. The goal of Coordinated Queries is to provide a general foundation for the flexible mapping of raw data attributes into perceptual attributes across multiple coordinated views. To achieve this goal, Coordinated Queries not only must be compatible with common types and sources of data, algorithms for querying data, and methods of rendering data, but also must support:

- extension to new data types and sources,
- specification of how views access data,
- customization of how views aggregate, filter, sort, and render data, and
- composition and reuse of visual abstractions.

In order to support all of these capabilities across multiple coordinated views, Coordinated Queries must also support not only the capabilities of a general model of coordination (listed in section 3.1), but also coupling of both sets of capabilities. In particular, it must support

coordination between views in terms of how user interactions affect data access, aggregation, filtering, sorting, rendering, and composition of these operations.

Several interrelated observations motivated the development of Coordinated Queries. (The strong relationship between these observations and those about coordination, in section 3.1, is not coincidental.) First, like coordination, visual abstraction is a function of visualizations as a whole. The appearance of data in each view in a visualization is determined by the combined interactive state of all views in that visualization, including the single or multiple data sets shown in those views. Visual abstraction need not be independent in individual views or even limited to pairwise data relationships between views.

Second, visual abstraction is simultaneously a perception leveraging problem and an interaction leveraging problem. That is, visual abstraction is about maximizing the analytic capabilities of the human visual and cognitive systems by giving users interactive control over the display of data in a visualization. Visualization designers incorporate views into visualizations in order to let users see and manipulate processing that determines what, where, and how data is drawn.

Third, visual abstraction is a cognitive coherence problem. As distinct but connected graphical projections of a high-dimensional abstract data space, each view in a visualization reveals different relationships between the data it displays and the space used to display it. Coupling visual abstraction with coordination enforces cognitive coherence between views by providing consistent spatial, perceptual, and interactive indications of these relationships.

## 4.2 Relational Model of Visual Abstraction

Coordinated Queries is designed to overcome limited generality in previous approaches to interactive specification of visual abstractions. These approaches fall into four categories:

- *Fixed Encoding.* Accessing, processing, and rendering of data is hard-coded into views that have been designed for a specific visual analytic purpose.
- *Mapping.* Views expose a small set of adjustable parameters for specifying the source and visual encoding of data. Different view types expose different parameters. Users edit these parameters in dialog boxes specific to each kind of view.
- *Data Flow.* Data "flows" downstream from sources to sinks (including views) in a graph of modules that perform querying and rendering operations.
- *Linking.* Views share data using a small set of predefined link types. Links translate interaction in one view into data access and processing operations in another view.

Coordinated Queries is based on the relational data model—itself based on relational algebra [35]—in which a declarative language is used to express query operations on tables of records. The relational data model provides several advantages for interactively coupling data access, data processing, and data rendering between multiple views:

- one-to-one mapping of entire records into view-specific glyphs,
- many-to-many mapping of data attributes into perceptual attributes (e.g. location, color, shape, size),
- filtering, ordering, and grouping of records,
- cross-table indexing on attributes using primary-foreign key relationships, and,
- potential for referencing interactive parameters in query statements.

These qualities provide several benefits for the Improvise software architecture:

- simplification of view design and implementation,

- separation of design from implementation for querying and rendering data,
- unification of data accessing and processing,
- reduction of the need to subclass views to handle different querying and rendering algorithms, and
- automatic tight coupling of visual abstractions between views.

They also provide several benefits to visualization designers and users:

- a clean conceptual separation between views, data, and data operations,
- centralization of data access and pre-processing,
- simplification of steps to populate views with data,
- visual as well as logical composition of multiple queries across multiple tables,
- increased visual abstraction flexibility,
- specification of visual abstractions in terms of interaction, and
- manipulation of visual abstractions interactively.

### 4.2.1 Coordinated Queries Overview

Coordinated Queries is a declarative language for specifying visual abstractions of record-based data representations. Each statement in this language, called an *expression*, is a tree of operators that calculates the value of an output field using the fields of an input record. Expressions make up query operations that views use to process and render data records, including:

- *Projections* map data attributes into view-specific glyphs or other visual attributes.

- *Filters* map data attributes into booleans that indicate (in)visibility of glyphs.
- *Sorts* map data attributes into comparators that determine relative spatial ordering.
- *Infos* produce data sets for rendering by accessing external data sources or by further processing other data sets (possibly produced by other infos).

Expressions are composed of eight different kinds of operators. *Function operators* perform a variety of duties including object construction, type casting, member access, arithmetic, and statistics. *Value operators* evaluate to a user-edited value of a particular object type. *Attribute operators* evaluate to the value of a field of the input record. In addition to these three basic kinds of operators, *aggregate operators* allow the calculation of simple aggregates on tables, *constant operators* provide easy access to frequently used fixed values (e.g. pi), and *conversion operators* perform common numeric conversions between units (e.g. feet to meters). *Index operators* provide indexed data lookup by mapping a primary key value of an attribute to a foreign key value of an attribute in a different data set.

Coordinated Queries extends Live Properties coordination graphs (as defined in section 3.3) to contain query operations whose expressions can be defined in terms of variable-referencing *variable operators*, thereby enabling *indirect coordination* of views. Figure 12 shows how interaction propagates through one level of dependence in an indirect coordination. Whenever an expression is evaluated during data processing and rendering, variable operators take on the current value of their corresponding variable. When a control depends directly on a variable that contains a query operation, it also depends indirectly on any variables referenced (possibly recursively) by the expressions that define that operation. In this way, it is possible to specify complex interactive dependencies between views.

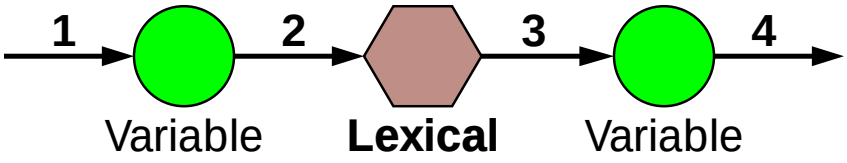



Figure 12: Indirect coordination. (1) An upstream object propagates a value change to a variable. (2) The variable notifies all lexical values that contain expressions which reference the variable. (3) Each expression notifies variables to which it is assigned as a value. (4) The variable sends a change notification to all downstream objects. Upstream and downstream objects can be properties (as in figure 8), or other lexical values.

Unlike a true relational data model, Coordinated Queries neither enforces primary-foreign key relationships nor provides support for join operations. Nonetheless, through variable operators, expressions can depend not only on the navigation and selection parameters of a visualization, but also on its projections, filters, and data sets. This multi-stage dependence enables a rich variety of visual queries including aggregation, grouping, indexing, and hierarchical nesting of views.

### 4.2.2 Model Components

Visualizations built with Coordinated Queries consist of a fourth kind of component, in addition to the three that comprise Live Properties (section 3.2.2):

- *Lexicals* are named wrappers for external data sets (*data*), query operations (*projections*, *filters*, and *sorts*), and derived data sets (*infos*). Lexicals can be assigned as values to variables, just like any other interactive parameter. Views access and process data through properties of lexical types.

In each Improvise visualization, the *lexicon* is a central repository for managing data sets and query operations. The lexicon contains one lexical for each data, info, projection, filter, and sort in the visualization. Lexicals can be created, deleted, and edited interactively. Changing

the value of a variable of a lexical type is a matter of selecting a different lexical from the lexicon rather than tediously respecifying data sets or editing query operations directly. Moreover, lexical wrapping allows each data set and query operation to be reused as the value assigned to multiple variables simultaneously. Using the lexicon, visualization designers can build up large libraries of data references and query operations for use during ongoing data exploration and analysis.

### Data

Coordinated Queries represents information in essentially the same way as other implementations of the relational data model, using attributes, schemas, fields, records, and tables to access and process data and metadata:

- An *attribute* describes the type and name of a single table column. Attribute types can be primitive (e.g. `int`, `double`, `string`) or composite (e.g. `date`, `URL`). Visualization involves additional graphical types such as `color`, `font`, `range`, and `rectangle`.

- A *schema* is a column-by-column metadata description of a table, of form $\{attribute_0, attribute_1 \ldots attribute_{n-1}\}$ in which $n$ is the number of columns in the table. Attributes can be accessed by index or name.

- A *field* is an instance of an attribute that contains a value of the attribute's type. Because fields are mutable, their values can be changed so long as they are of the correct type.

- A *record* is an instance of a schema, of form $\{field_0, field_1 \ldots field_{n-1}\}$ in which $n$ is the number of table columns. Record fields can be accessed and modified by index.

- A *table* is a list of records of a particular schema. Records in a table can be accessed by *scanning* (iterating over) the list. Records in tables also have integer *record identifiers*.

The records in a table can come from a variety of sources, including flat files, traditional databases, web services, or in-memory arrays of internal data structures. In Improvise, each data source implements an interface that allows it to be accessed as a table; that is, to scan it and to determine its size and schema. Lexical data objects encapsulate and name instances of this table interface. By referring indirectly to data sources through lexical data objects, views can access records from any source uniformly and transparently.

### Projections

Projection is the means by which all views display data for users. Projections specify the appearance of records in a data set by calculating view-specific *visual attributes*—such as position, shape, color, etc.—as functions of the attributes of each record. Views render *glyphs* composed of each record's visual attributes.

A projection is a function that maps records of one schema into records of another schema. Each field of the output record is calculated by evaluating an expression that is defined in terms of the fields of the input record:

$$record_{src} = \{field_{src,0}, field_{src,1} \ldots field_{src,n-1}\} \tag{1}$$

$$record_{dst} = \{field_{dst,0}, field_{dst,1} \ldots field_{dst,m-1}\} \tag{2}$$

$$field_{dst,i} = f_i(field_{src,0}, field_{src,1} \ldots field_{src,n-1}) \tag{3}$$

$$projection = [f_0, f_1 \ldots f_{m-1}] \tag{4}$$

In order to enforce strong type matching, projections are defined in terms of input and output schemas. The return type of each expression $f_i$ must be exactly the type of $field_i$ in the output record, as specified by $attribute_i$ of the output schema.

All views specify the required output schema of their glyphs or other visual attributes, in

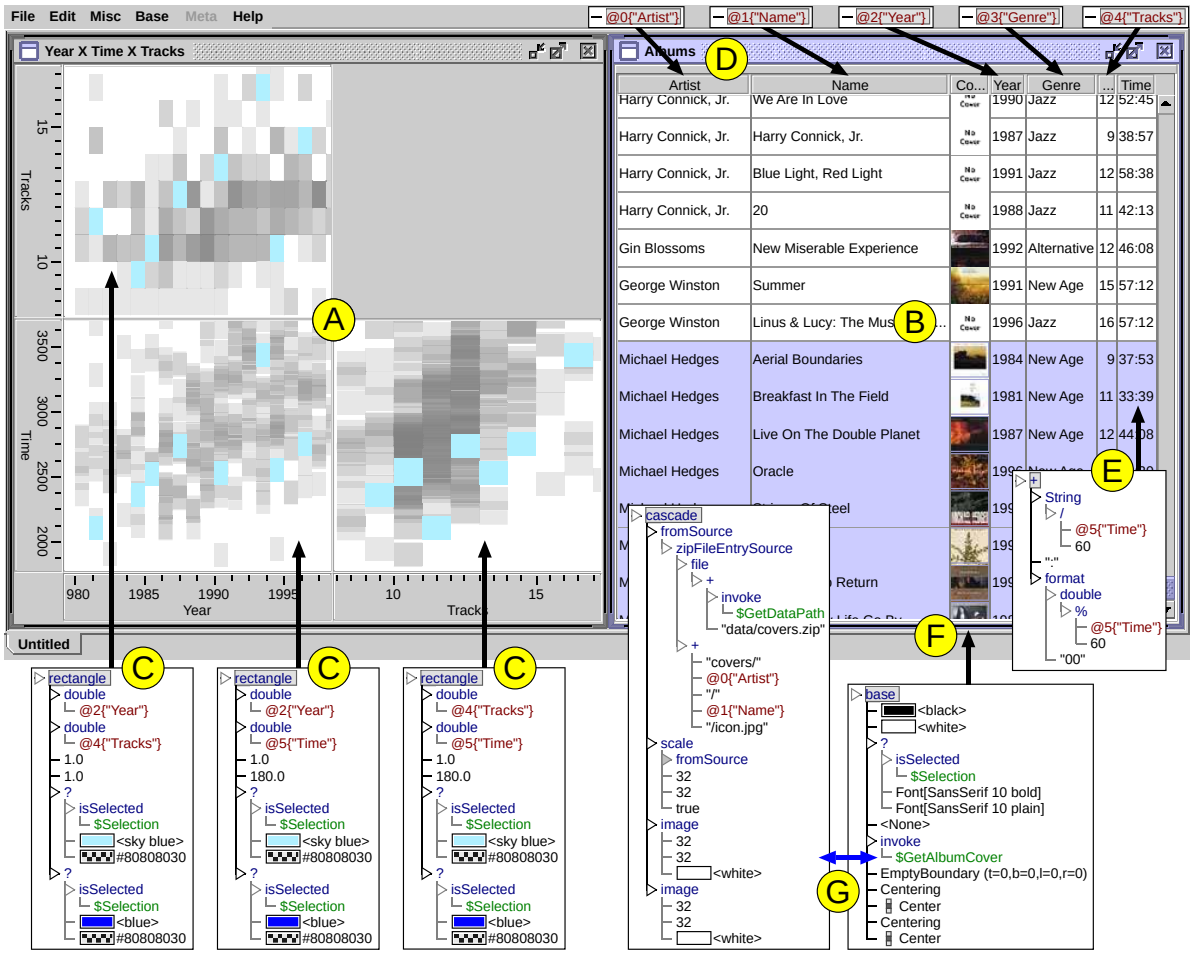



Figure 13: Music albums drawn in a 3-D scatter plot matrix (A) and a table (B) using projection expressions that generate orthogonal rectangular scatter plot glyphs (C) and the text contents of cells in each table column (D). Cells can display calculated strings (E) and/or images (F) accessed from media sources that are identified by a function of data attributes (G).

accordance with the way they render data. For example, projections for scatter plots produce single-field records containing glyph objects that know how to draw themselves on the cartesian coordinate plane. Projections for table views produce multi-field records containing either glyphs or arbitrary objects that can be turned into strings for display in the cells of each row. For example, figure 13 shows several coordinated projections used to render data in a variation of the music album visualization from figure 7.

### Filters

Filtering is the means by which views isolate data subsets for more focused visual exploration. Filters specify the visibility of records in a data set by calculating a `true` or `false` result as a function of the attributes of each record. Views render projected glyphs of only those records that pass the filter (that is, records for which the calculated filter value is `true`).

A filter is a function that maps records of an input schema into records containing a single field of boolean type. The field of the output record is calculated by evaluating an expression defined in terms of the fields of the input record:

$$record_{src} = \{field_{src,0}, field_{src,1} \ldots field_{src,n-1}\} \tag{5}$$

$$record_{dst} = \{field_{dst,0}\} \tag{6}$$

$$field_{dst,0} = f_0(field_{src,0}, field_{src,1} \ldots field_{src,n-1}) \tag{7}$$

$$filter = [f_0] \tag{8}$$

A single filter can be applied to multiple data sets of the same schema in multiple views. For example, figure 14 shows how a filter can be used to focus on the same subset of records, even though they are rendered in two different kinds of views using different projections.

### Sorts

Sorting is the means by which views prioritize records for display. Sorts specify the order of records in a data set by calculating *comparators*—objects that know how to determine their order relative to each other—as a function of the attributes of each record. Ordering of records in determined by pairwise evaluation of their corresponding comparators in the inner loop of the table sorting algorithm. While the computational complexity of the sorting algorithm is `O(n log n)`, the complexity of expression evaluation during pre-sorting is `O(n)`.

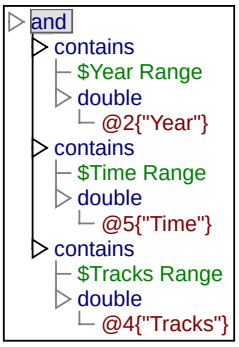



Figure 14: Filter expression used by all four views in figure 13 to filter out music albums outside the visible range of years, times, and track counts in the scatter plot matrix. Applying the filter to the scatter plots themselves allows examination of a true cubic region rather than the union of three infinitely deep rectangular regions.

A sort is a function that maps records of an input schema into records containing a single field having a comparator as its value. The field of the output record is calculated by evaluating an expression defined in terms of the fields of the input record:

$$record_{src} = \{field_{src,0}, field_{src,1} \ldots field_{src,n-1}\} \tag{9}$$

$$record_{dst} = \{field_{dst,0}\} \tag{10}$$

$$field_{dst,0} = f_0(field_{src,0}, field_{src,1} \ldots field_{src,n-1}) \tag{11}$$

$$sort = [f_0] \tag{12}$$

Different kinds of views utilize sorts in different ways. In some views, the screen location of a rendered record is merely an indication of where it is in its table, such as ascending or descending order in list views. In other views, location is a spatial function of a record's attributes, such as cartesian position in scatter plots. Whereas in list views sorting affects vertical position, in scatter plots sorting affects the z-order (drawing precedence) of glyphs.

Like filters, sorts can be applied to multiple data sets of the same schema in multiple views. For example, figure 15 shows how a sort can be used to raise selected records to the top of each

view, even though “top” has a different meaning in each view (frontmost z-order in the scatter plots, lowest rows in the table view).

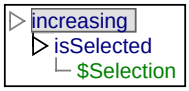



Figure 15: Sort expression used by all four views in figure 13 to give higher visual priority to selected music albums.

### Infos

Infos are the means by which views generate derived data sets for display. Evaluating a single expression produces an entire data set as its result. Unlike other lexicals, infos do not take an input record. Instead, the expression provides a functional specification of where to find new data sets (from a file, database, etc.) or how to process data sets that are already located.

An info is a procedure (that is, a function that takes a record of the empty schema as input) that generates one record containing a single field having a data set as its value. The field of the output record is calculated by evaluating an expression defined in terms of no input attributes:

$$record_{src} = \{\} \tag{13}$$

$$record_{dst} = \{field_{dst,0}\} \tag{14}$$

$$field_{dst,0} = f_0() \tag{15}$$

$$info = [f_0] \tag{16}$$

All views access data through properties that have a lexical info as their type. To render data, views apply projections, filters, and sorts to the data obtained by evaluating info expressions. For example, figure 16 shows how a view renders data from a flat text file that is located by specifying its path and name in the local file system.

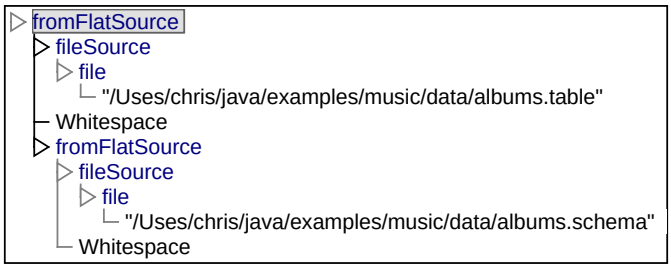



Figure 16: Info expression used to produce a single data set (from a whitespace-formatted text file with a separate metadata file) for rendering in the four views in figure 13.

Views can access multiple data sets using multiple info properties. Moreover, each info property can be associated with distinct projection, filter, and sort properties. Scatter plots use this approach to render multiple data sets in a stack of independent layers, for purposes such as highlighting glyphs, overlaying time series, or building layered geographic maps.

### 4.2.3 Expressions

Expressions that make up lexical types are functions that calculate the value of an output field using the fields of an input record. Like functions in programming languages in general, expressions are defined by a tree of operators. Expressions are strongly typed; they specify and enforce the required attribute types of their input and output fields.

Expression evaluation proceeds depth-first, post-order. Although the evaluator acts as an interpreter, the operators themselves are compiled. For each node in the expression tree, the evaluator evaluates each subexpression, marshals the results, and then invokes the operator on them. The result of the operator invocation is returned to the node's parent expression. At all stages, sets of arguments are wrapped in arrays of fields, and output values are wrapped in single fields. Upon completion, the output field of the root expression contains the glyph, boolean, comparator, or data set that is the result of the projection, filter, sort, or info.

Functional programming languages typically provide (at least) three kinds of operators:

functions, argument references, and literals (constants). Coordinated Queries has equivalents called *function operators*, *attribute operators*, and *value operators*, respectively. In addition to these three basic kinds of operators, *constant operators* provide easy access to frequently used values and *conversion operators* perform unit conversions. *Aggregate operators* allow the calculation of simple aggregates on tables. *Index operators* perform indexed lookups across tables. *Variable operators* are references to variables, including those that are assigned other lexical objects. Figure 17 shows an example of how operators can be composed into expressions that make up several interrelated lexical objects.

### Function Operators

Exploration and analysis activities require function operators for performing calculations on the wide variety of data object types that are encountered in visualization, including:

- routine exploration (e.g. integers, doubles, strings, colors, dates),
- advanced visual analysis techniques (e.g. color gradients, transparency), and
- metavisualization (e.g. components, events, coordinations).

Improvise provides about 1400 function operators for types from all three of these groups. The coverage of this library offers visualization designers the flexibility needed to construct rich visual abstractions of data across multiple coordinated views through projection, filtering, and sorting. New visualization systems built using Coordinated Queries would need to provide appropriate function operators that correspond to any object types particular to their unique data analysis capabilities, in addition to those already provided by the Improvise library.

Function operators perform a variety of duties including object construction, type casting, member access, geometry, arithmetic, and statistics, much like built-in functions in functional

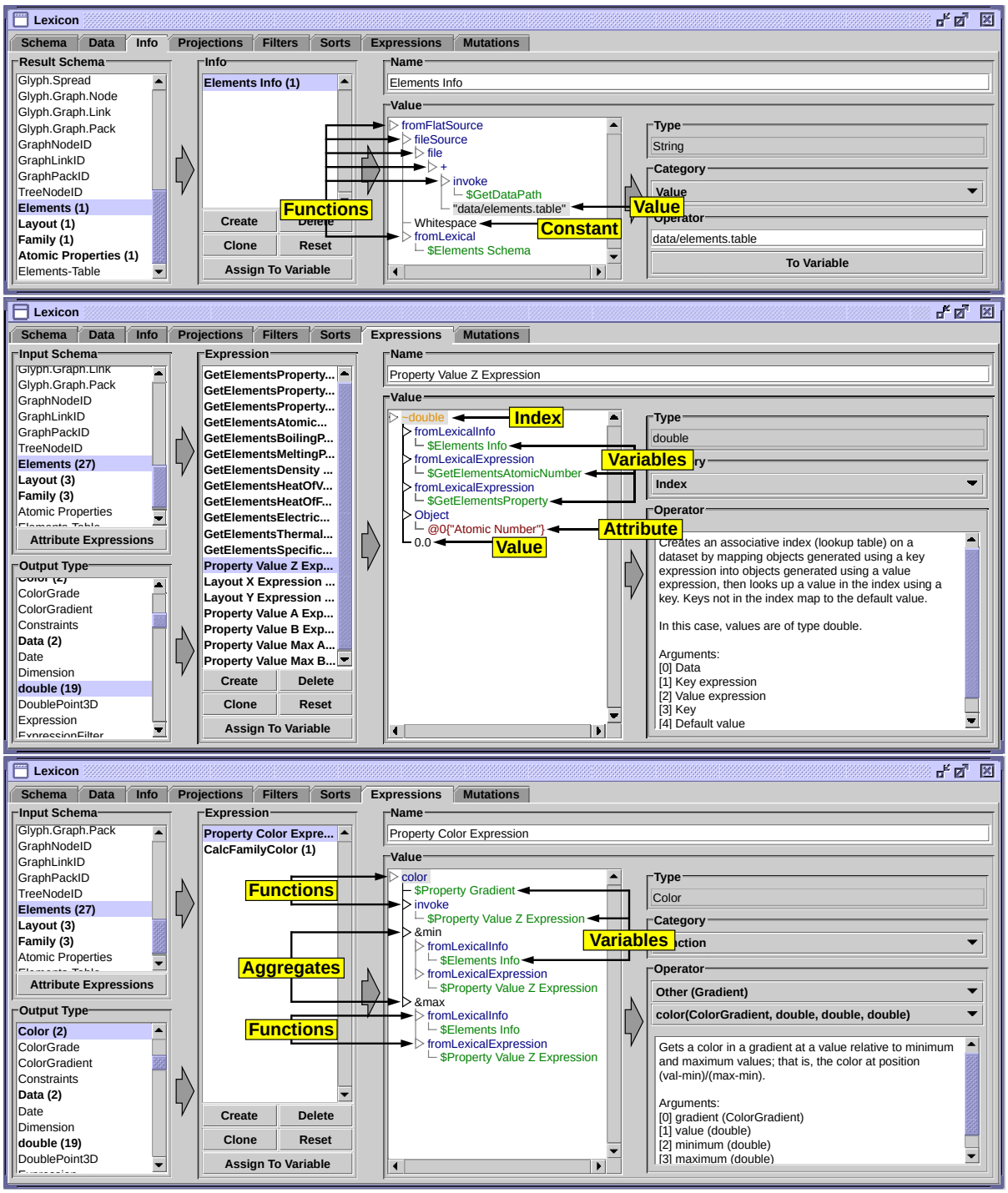



Figure 17: Operator trees for expressions that specify a whitespace-delimited flat file data source (top), an index for looking up the value of a chemical property of an element given its atomic number (middle), and a color calculated by looking up the chemical property value in a gradient relative to minimum and maximum values over all elements (bottom). Multiple views in the visualization in appendix A.4 encode elements using these expressions.

programming languages. Each function operator has an output type, a list of input types, a description, a name, and a *family*. Families are a simple namespace mechanism used to group related function operators. They are used in the expression construction interface as a way to help users select functions categorically. Table 3 shows several typical function operators.

| | |
|---|---|
| *Name* | int |
| *Family* | Casts |
| *Output Type* | `int` |
| *Input Types* | `{string}` |
| *Description* | Parses a string as an integer. |
| *Name* | add |
| *Family* | Mathematics |
| *Output Type* | `double` |
| *Input Types* | `{double ...}` |
| *Description* | Calculates the sum of numbers. |
| *Name* | color |
| *Family* | Other (Gradient) |
| *Output Type* | `color` |
| *Input Types* | `{gradient, double, double, double}` |
| *Description* | Looks up a color in a color gradient relative to minimum/maximum values. |
| *Name* | titled |
| *Family* | Basic |
| *Output Type* | `border` |
| *Input Types* | `{string}` |
| *Description* | Creates a titled border (a labelled box). |

Table 3: Typical function operators of different types.

Some function operators are ellipsis functions. Similar to ellipsis functions in C, they take a user-definable number of arguments when it makes sense to do so. Numeric addition (the second example in table 3) is one such function.

### Attribute Operators

Attribute operators are the means by which expressions refer to fields of an input record. As such, they are similar to argument references in functional programming languages. Because

any given attribute operator evaluates to the value of a particular field of the input record, the result of the evaluation may change from record to record, unlike other operators.

An attribute operator accesses a field by specifying its column number. The field in that position must have the same type as the attribute operator. In particular, attribute operators do not perform automatic type casting, e.g. trying to use an attribute operator of type `int` to calculate an output field of type `double` fails. Instead, casts can be performed using function operators that have been defined for that purpose.

### Value Operators

Value operators are a means to incorporate arbitrary constants into expressions, much like literals in functional programming languages. A value operator evaluates to a value of a particular numeric or object type, such as those shown in table 4.

| *Type* | *Value* |
|---|---|
| `int` | `6` |
| `double` | `5.47` |
| `string` | `"foo"` |
| `color` | `#ffff00` |
| `font` | *serif-bold-12* |
| `range` | `[-5.0, 13.1]` |

Table 4: Typical values of value operators of different types.

A value can be any legal value of the operator's type. Special values, such as the floating point values `NaN` and `Inf` and the object value `NULL`, can be used in expressions to indicate absent data, erroneous data, or other special cases. (A key advantage of expression-based visual abstraction is the ability of visualization designers to assign special case semantics to arbitrary data values of any type.)

### Constant Operators

Constant operators are a convenience to provide easy access to common, important, or hard-to-remember values. Constant operators usually involve scientific or calendrical numbers, but can be of any type. Table 5 shows several typical constant operators.

| | |
|---|---|
| *Name* | c |
| *Family* | Physics |
| *Type* | `double` |
| *Value* | `2.99792458e8` |
| *Description* | Speed of light in a vacuum, in m/s. |
| *Name* | January |
| *Family* | Dates |
| *Type* | `string` |
| *Value* | `"January"` |
| *Description* | The name of the first month of the Gregorian calendar. |
| *Name* | Copper |
| *Family* | Metal |
| *Type* | `color` |
| *Value* | `#c09488` |
| *Description* | A copperish color. |

Table 5: Typical constant operators of different types.

Special values of particular types (such as `NaN`, `Inf`, and `NULL`) are also provided as constant operators. Although similar to value operators, constant operators are predefined and cannot be modified. Like function operators, each constant operator is classified by type and family, has a name, and provides a description of its value with units, if any.

### Conversion Operators

Units are conceptually and methodologically important in data analysis. Analysts need to be aware of the measurement units of displayed data attributes in order to correctly interpret what they see. Coordinated Queries does not support enforcement or automatic conversion of units in its processing of data (although it could be extended to do so). Instead, conversion operators

are a convenience to provide visualization designers with a way to specify common unit conversions explicitly, without the need to construct the functional form of the conversion as an expression. For example, table 6 shows a pair of operators for converting temperatures from Fahrenheit to Centigrade and back. The visualization designer is spared from remembering and constructing forward and inverse conversion expression trees of form $C = (F - 32.0)/1.8$ and $F = 1.8C + 32.0$.

| | |
|---|---|
| *Name* | FtoC |
| *Family* | Temperature |
| *Type* | `double` |
| *Description* | Converts temperatures from Fahrenheit to Centigrade. |
| *Name* | CtoF |
| *Family* | Temperature |
| *Type* | `double` |
| *Description* | Converts temperatures from Centigrade to Fahrenheit. |

Table 6: Typical pair of conversion operators.

A conversion operator takes a single argument of a particular type and returns a value of the same type. Like function operators, conversion operators have a type, a family, a name, and a description of the original and converted units. Conversion operators come in pairs, one for the forward conversion and one for the inverse conversion.

### Aggregate Operators

A major challenge for visualization research is finding usable, useful ways to display large, high-dimensional data sets in finite screen space [78, 45, 69]. Aggregation is a common way of reducing the size and/or dimensionality of data prior to rendering. Current problems in visual analytics involve data sets with billions of records and hundreds of dimensions [77]; the huge size of these data sets necessitates multiple stages of aggregation. Moreover, visualization systems need to support rich visual abstraction of aggregated data in the same way as for

smaller, non-aggregated data sets [54, 85]. Multiscale data cubes [133], focus+context [12], overview+detail [129, 114], fisheye views [49] and constant density semantic zoom [156] are general visual techniques that compliment previsual aggregation algorithms.

Coordinated Queries supports aggregation using aggregate operators. An aggregate operator calculates an aggregate value over all records in a data set by applying an arbitrary function to the attributes of those records. Aggregate operators take two arguments:

- A lexical info that specifies a data set to aggregate.
- A lexical expression that calculates an output value as a function of the attributes of each input record in the data set. Aggregation is performed on this derived attribute value.

Aggregate operators lazily calculate an aggregate, storing the result in a cache keyed on the syntactic form of the expressions that make up the two arguments. Typically, neither argument varies from evaluation to evaluation; caching amortizes the cost of calculating the aggregate over multiple evaluations of the aggregate operator. As a result, the application of projection, filter, and sort expressions to a data set can involve aggregates of any data set (including the one being processed) in a way that is both computationally efficient and syntactically embeddable in the query language. Table 7 outlines several typical aggregate operators.

| | |
|---|---|
| *Name* | min |
| *Type* | `string` |
| *Description* | The lexicographic minimum of a set of strings. |
| *Name* | average |
| *Type* | `double` |
| *Description* | The numeric average of a set of doubles. |
| *Name* | union |
| *Type* | `rectangle` |
| *Description* | The smallest rectangle that contains a set of rectangles. |

Table 7: Typical aggregate operators of different types.

### Index Operators

Analysts are often interested in visualizing data from multiple sources simultaneously [138]. Visual cross-referencing between data sets is a powerful means of exploring high-dimensional information split into multiple data sets collected at different times and places in different ways.

In relational databases, primary-foreign key relationships provide a means to perform indexed lookups within or across tables. Index operators provide similar capability in Coordinated Queries. Using index operators, it is possible to project, filter, and sort the records of one data set by cross-indexing arbitrary functions of attributes of those records into an arbitrary key-value mapping of another data set. Index operators take five arguments:

- A lexical info that specifies a data set to index.
- A lexical expression that calculates keys as a function of attributes of each record.
- A lexical expression that calculates values as a function of attributes of each record.
- A key to map into a value using the index.
- A default value to use when the index finds no value for the specified key.

For example, the elections visualization in appendix A.3 accesses an ancillary two-column data set that associates each political party with a unique color. An index on this data set maps attributes in the first column (party name as a string) into attributes in the second column (RGB color as a hexcode string). Views draw each candidate by using the index to look up the color for the corresponding party, defaulting to gray when the party name is absent from the index.

Index operators lazily build an index—currently implemented as a simple hash table keyed on the syntactic form of expressions making up the first three arguments—that maps calculated key objects into calculated value objects for each record in the specified data set. Typically, the

first three arguments do not vary from evaluation to evaluation. As a result, simple caching of the hash table amortizes the cost of building the index over successive evaluation of projection, filter, or sort expressions to the records of a table (and often even over multiple applications of these query operations to multiple tables), reducing the cost of evaluating the index operator to a simple index lookup (of a key that typically does vary from evaluation to evaluation). Because Coordinated Queries does not enforce primary-foreign key relationships, however, redundant keys are ignored when building an index; lookups always produce the first value calculated for any given key during index construction. Nevertheless, such relaxed index semantics are often sufficient for visualization purposes.

### Variable Operators

Interactive coupling of coordination and visual abstraction is the key innovation of Coordinated Queries. Variable operators make extensive coordination and rich visual abstraction possible in Improvise by allowing expressions to depend on any interactive parameter in a visualization (including those assigned expression-based lexical objects). Variable operators are simply references to variables (section 3.2.2). When its parent expression is applied to an input record, a variable operator evaluates to the current value assigned to its variable; that is, to the most recent navigation or selection parameter value as determined by user input in some control.

By incorporating variable operators, expressions become interactively dependent on controls. Because views render infos using projections, filters, and sorts made up of expressions, variable operators allow visualization designers to specify complex interactive dependencies between views that are difficult or impossible to realize with other coordination approaches. For example, the views in figure 13 coordinate through mutual references to a selection variable in their projection expressions. Selecting items in any view causes those items to appear

highlighted in all views; each record is redrawn as a function of whether it is selected according to the changed variable value. The same views coordinate by sharing a filter expression (figure 14) that refers to three range variables that are bound to navigational input in the three scatter plots. Dragging in any one of the three scatter plots changes the range value of one or two variables, causing all views to redraw themselves with different items filtered out.

As this example shows, views can coordinate indirectly either by sharing the same lexicals, or by sharing different lexicals whose expressions refer to the same variables through variable operators, giving visualization designers great flexibility for interconnecting views.

### 4.2.4 Graph Relationships and Notification Protocol

In Live Properties, controls access and modify variables through their properties; whenever the value of a variable changes, all controls having properties bound to that variable are notified by callback. Coordinated Queries extends the Live Properties notification protocol by allowing expressions-based query objects to: (1) be defined in terms of variables and (2) be themselves assigned to variables as values. As a result, two or more controls can be connected to each other through their respective properties via one or more paths through a *coordinated query graph* (figure 18).

Notification is also triggered in response to dynamic changes in the structure of a coordinated query graph. There are four kinds of relationships (graph edges) that can be established and broken during visualization construction:

- A *slot* is a parent-child relationship between a control and one of its properties. Slots are established and broken automatically whenever a control creates or destroys properties. For example, adding a data layer to a scatter plot creates info, projection, filter, sort, and selection properties and the corresponding slots.

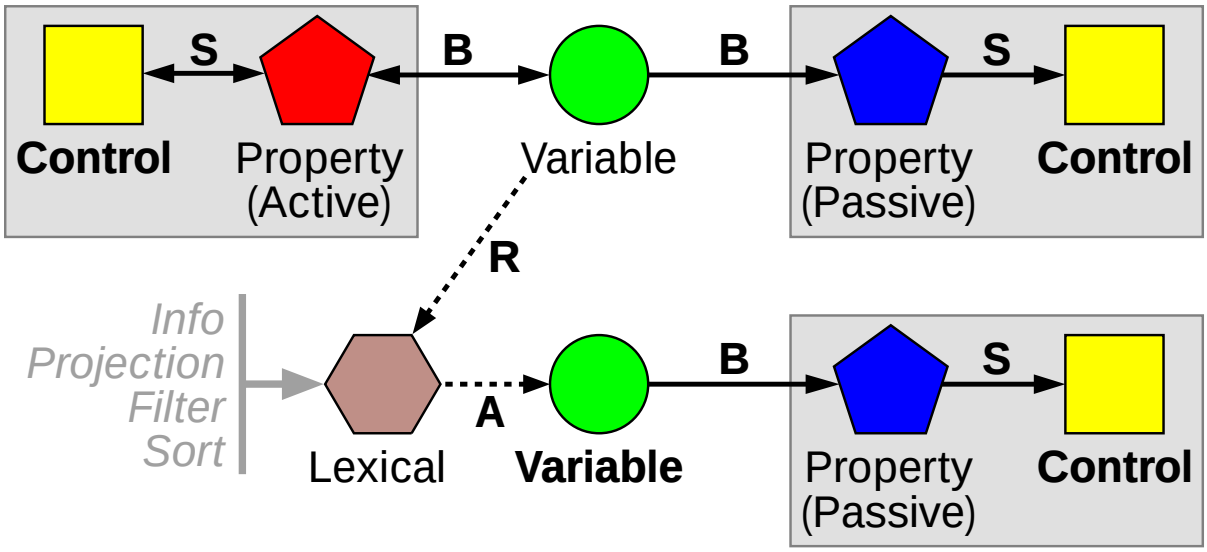



Figure 18: Coordinated query graph structure. Slot (S), binding (B), assignment (A), and reference (R) relationships connect control, property, variable, and lexical objects.

- A *binding* is a bidirectional relationship between a property and a variable. Bindings are established and broken in order to share variable values between one or more controls (via slots).

- An *assignment* is a property-owner relationship between a lexical and a variable. Assignments are established by setting the value of a variable to be a lexical, and broken by setting the value to `null`.

- A *reference* is a subject-object relationship between a lexical and a variable. References are established by incorporating variable operators into the expressions of a lexical, and broken when the operators are removed or replaced.

Deletion of model components (graph nodes) necessarily breaks all relationships in which they are involved, possibly triggering a flood of notifications throughout the graph. For example, deleting the `Year` range variable in the visualization in figure 13 would eliminate panning and zooming over years in the scatter plot matrix (by breaking binding relationships with two

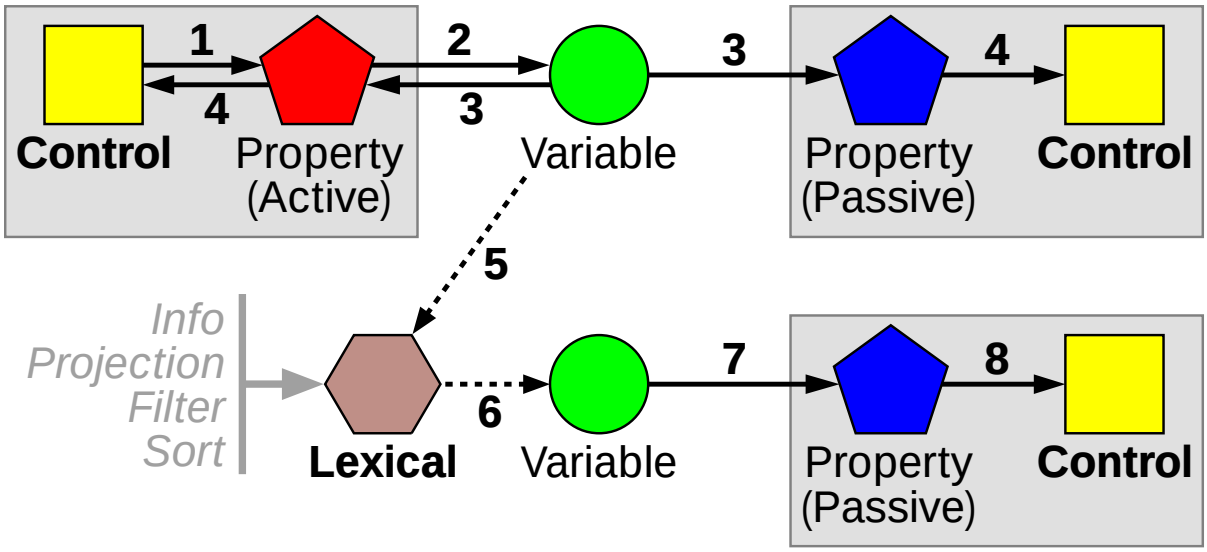



Figure 19: Notifications in coordinated query graphs. In response to interaction, a control modifies the value of one of its active properties (1), which assigns the new value to its bound variable (2). The variable sends a change notification to all properties bound to it (3), each of which notifies its control of the change (4). The variable also notifies all lexicals whose expressions refer to it (5). Each lexical notifies any variables to which it is assigned as a value (6). When a view receives notification through one of its properties that a lexical has changed (7, 8), it updates itself by processing the modified query.

scatter plots and one axis control). It would also change filtering behavior between the scatter plot matrix and the table view (by replacing the `$Year Range` variable operator in the expression in figure 14 with a value operator corresponding to the variable's value).

User interactions in controls trigger updates in Coordinated Queries just as they do in Live Properties (figure 19):

1. The user interacts with a control.

2. The control assigns new values to its properties.

3. Each property assigns its new value to its bound variable.

The four step notification sequence that occurs along the reverse path is more complicated than in Live Properties:

1. When a variable changes, it notifies all lexicals that reference it (through variable operators in expressions) as well as all properties bound to it.

2. When a lexical changes, it notifies all variables to which it is assigned as a value.

3. When a property changes, it notifies its parent control.

4. Notified controls update themselves appropriately.

Views are notified of changes to any variables they depend on in the coordinated query graph, whether directly (through non-lexical properties) or indirectly (through variable operators in the expressions of lexicals). In particular, when a view receives notification through one of its lexical properties that a lexical has changed, it updates itself using the current values of its info, projection, filter, and sort properties as well as the values of its non-lexical properties. Because lexicals can refer to variables that may themselves be assigned lexical values, the first two steps in the notification sequence may repeat, branch, and converge inside the "core" part of the graph that consists of variable and lexical components having assignment and reference relationships. For example, figure 20 shows how notification flows through the graph in response to interactive changes to the `Year` range variable in the `YearTime` scatter plot.

### 4.2.5 Implementing Views

The clean separation of query editing and change notification from input handling and update rendering makes the implementation of views in Coordinated Queries both modular and elegant (figure 21). Different kinds of views support different input gestures and utilize different rendering methods, designed and implemented around properties and lexical objects on a case-by-case basis.

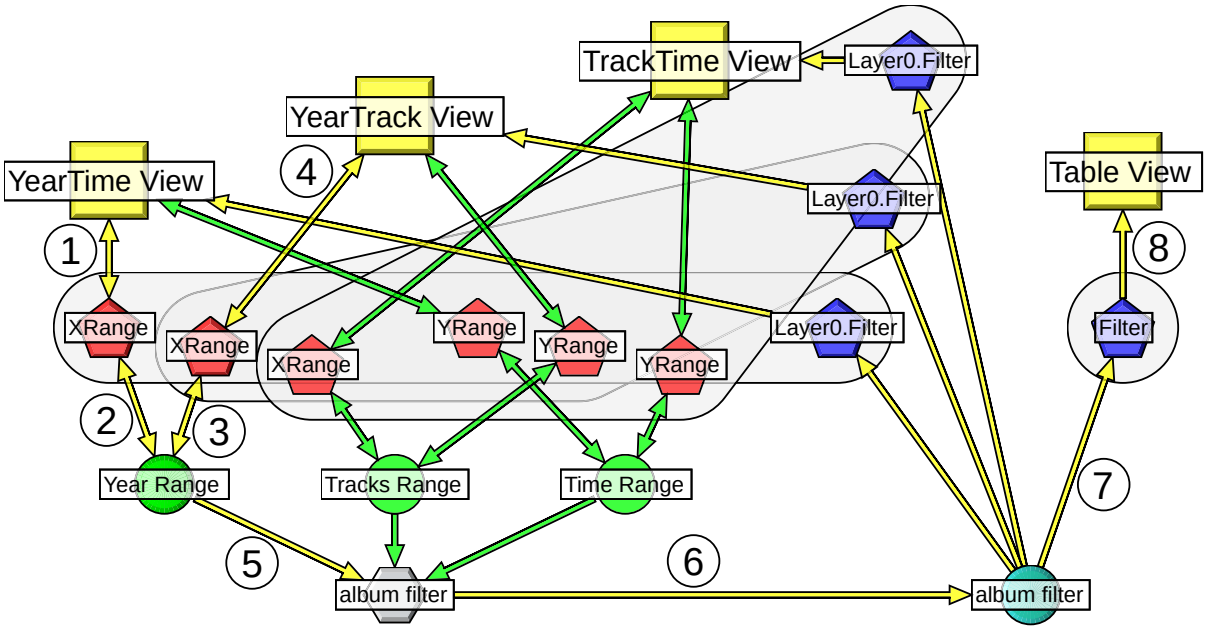



Figure 20: Example of notification. In response to horizontal dragging, a scatter plot modifies the value of its horizontal range property (1), which assigns the new value to its bound variable (2). The variable sends a change notification to all range properties bound to it (3), each of which notifies its parent scatter plot of the change (4). The variable also notifies a filter whose expression refers to it through a variable operator (5). The filter notifies a variable to which it is assigned as a value (6). When the table view receives notification through its filter property that its filter has changed (7, 8), it updates itself by reprocessing its data using the modified filter. All three scatterplots receive notification of the changed filter in the same manner.

For instance, a scatter plot changes the overall interactive state of a visualization in response to local input gestures by converting mouse screen locations and keystrokes into cartesian coordinates, then setting the values of navigation and selection properties correspondingly. The scatter plot displays its current interactive state in three steps. First, it evaluates its current info expression to obtain a data set. Second, it filters, sorts, and projects this data set into glyphs. Third, it draws glyphs by translating view coordinates into screen coordinates, using the current values of its two range properties.

Navigation gestures in scatter plots include:

- *Panning.* Horizontal and vertical mouse drags and numeric keypad presses translate the

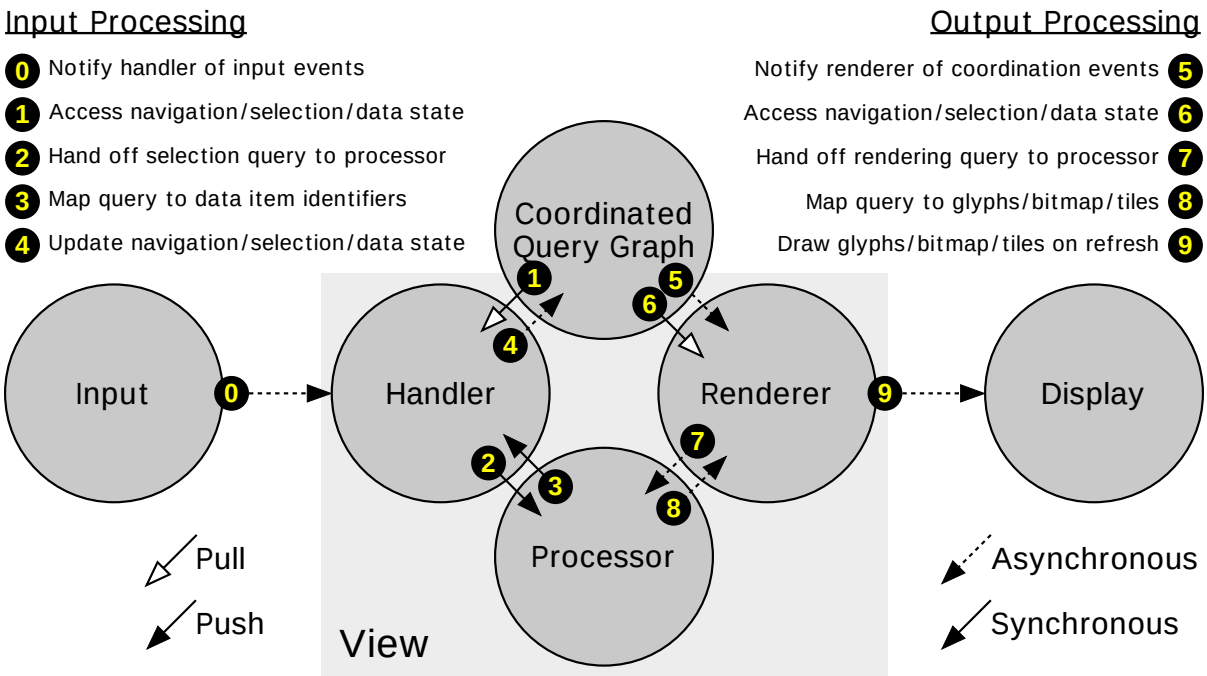



Figure 21: Interaction processing in Coordinated Queries views.

`XRange` and `YRange` properties, respectively.

- *Zooming.* While the `Shift` key is pressed, the same mouse drags and keypad presses scale the two range properties rather than translating them.

- *Home.* The `home` and `keypad 5` keys change the two range properties so that the plot shows the minimal rectangular region containing all data items.

Selection gestures (all of which are additive when the `Shift` key is down) include:

- *Select At.* A mouse click selects all data items whose glyphs contain the mouse point.

- *Rubber band.* Mouse drags with the `Control` key pressed draw a rubber band around a rectangular selection region. Items whose glyphs are fully inside the region are selected.

- *Lasso.* Mouse drags with the `Control` and `Alt` keys pressed draw a lasso around a polygonal selection region. Items whose glyphs are fully inside the region are selected.

- *Select All.* `Control-A` selects all data items.

During protracted input gestures (such as lassoing), changes to variables in the coordinated query graph via range and selection properties repeatedly initiate the notification protocol, causing the scatter plot to update its display continuously.

### 4.2.6 Coordinating Views

The combination of structural components and notification protocol in Coordinated Queries provides a unified event handling mechanism that supports interactive building and browsing operations in visualization interfaces, thereby encouraging more open-ended information exploration and analysis. The visualization building process consists of designing and constructing a coordinated query graph by loading data sets from various sources, defining lexical query operations by editing expressions, and specifying the visual queries displayed in views by connecting data sets and query operations to the views' lexical properties (as described in section 3.2.4).

A coordinated query graph is a specification for how views that make up a visualization interface collectively respond to repeated interaction by displaying a snapshot of a high-dimensional data space in its current interactive state. Visual analysis of each snapshot suggests further exploration in the form of more interactions that lead successively to snapshots of other interactive states. In this iterative process, views translate raw inputs having low-dimensional, physical, display space coordinates into semantic inputs having high-dimensional, abstract, data space coordinates. Navigation inputs change the spatial region of the displayed snapshot; selection inputs mark data objects for visual differentiation in the displayed snapshot. In both cases, input translation involves the current spatial state of the coordinated query graph. In

the case of selection inputs, translation also involves the current data, query, and object selection state of the coordinated query graph. Figure 22 summarizes the relationships between the users, views, and coordinated query graph of a Coordinated Queries visualization.

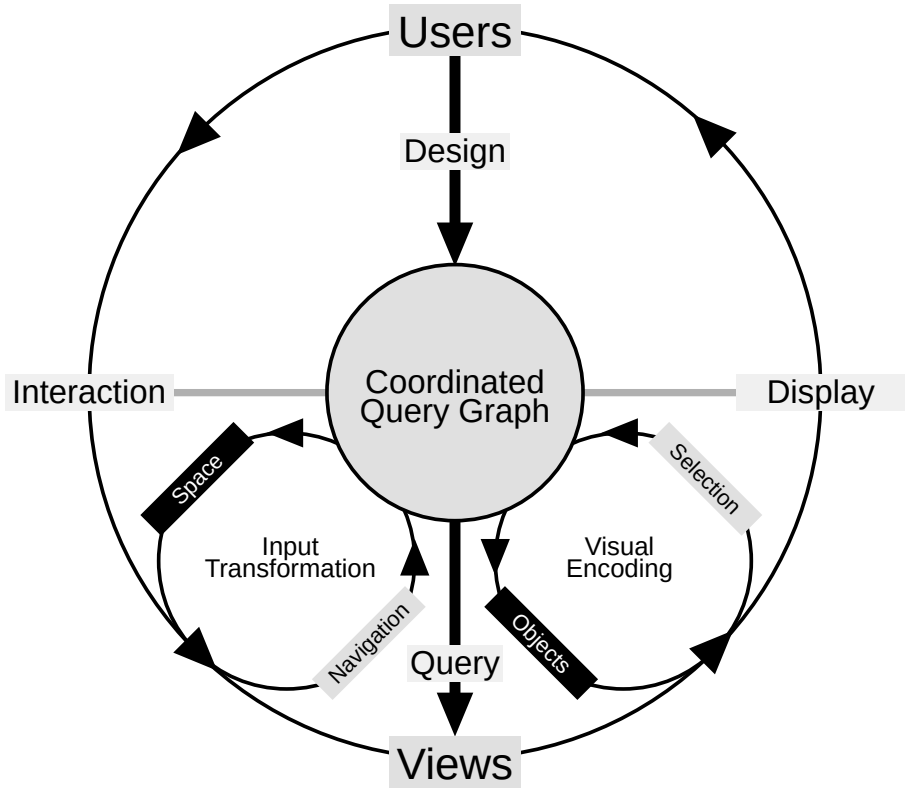



Figure 22: Cycle of user input and view rendering in a Coordinated Queries visualization interface. Views translate interaction in space into navigation coordinations, and interaction with items into selection coordinations.

As the face of all visualizations, views are the focus of attention during both building (window layout, coordination, and query specification) and browsing (navigation and selection) tasks. Users interact with views to change the interactive parameters that determine the coordinated query state of a visualization. For this reason, it is views and not lexicals or expressions that are responsible for initiating and responding to changes to variables. Thus, it is views that coordinate with each other. Chapter 6 classifies and enumerates some of the many useful patterns of coordination possible in multiview visualizations using Coordinated Queries.

## 4.3 Graph Model of Visual Abstraction

Coordinated multiple view visualizations built using Coordinated Queries consist of sets of controls (C), properties (P), variables (V), and lexicals (X) connected by slot (S), binding (B), assignment (A), and reference (R) relationships:

$$
\begin{aligned}
CMV &= (C, P, V, L, S, B, A, R),\ where \\
C &= \{c_0, \ldots, c_{N_C}\}, c_i = (type, name, p_i ... p_M), p_i \in P \\
P &= \{p_0, \ldots, p_{N_P}\}, p_i = (type, name, \{v | null\}, default), v \in V \\
V &= \{v_0, \ldots, v_{N_V}\}, v_i = (type, name, focused?, editing?, locked?) \\
X &= \{x_0, \ldots, x_{N_X}\}, x_i = (type, name, v_0 ... v_M), v_i \in V \\
S &= \{s_0, \ldots, s_{N_S}\}, s_i = (c_j, p_k) \\
B &= \{b_0, \ldots, b_{N_B}\}, b_i = (p_j, v_k)\ where\ p_j.type = v_k.type \\
A &= \{a_0, \ldots, a_{N_A}\}, a_i = (v_j, x_k)\ where\ v_j.type = x_k.type \\
R &= \{r_0, \ldots, r_{N_R}\}, r_i = (x_j, v_k)
\end{aligned}
$$

In this *coordinated query graph*, nodes are controls, properties, variables, and lexicals. Edges are slot, binding, assignment, and reference relationships. As in Live Properties, controls are not connected directly to each other. Instead, coordinations correspond to paths of form $c \leftrightarrow p \leftrightarrow v ... \{\rightarrow x \rightarrow v\}^n ... \leftrightarrow p \leftrightarrow c$ through the graph. For example, figure 23 shows a portion of the coordinated query graph for the visualization in figure 13, in which selection of items in the table view affects the appearance of items in the scatter plot matrix.

The topology of coordinated query graphs is more complicated than the topology of simple coordination graphs (section 3.3). Assignment and reference relationships result in a partially

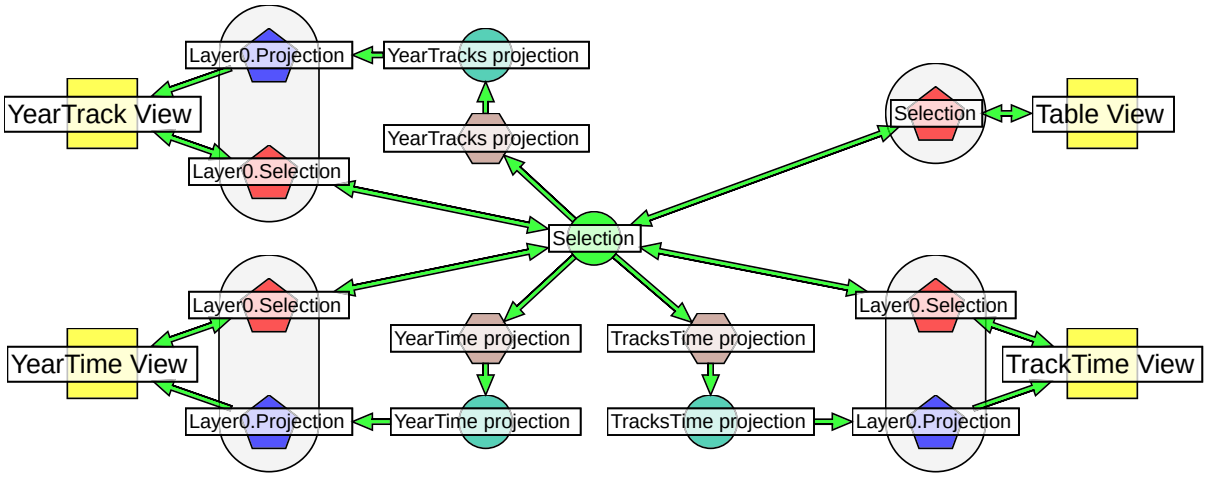



Figure 23: Coordinated query graph for a table coordinated with a 3-D matrix of scatter plots with axis controls. Selecting items in any of the views causes them to be highlighted in all views. Each of the 18 paths through the graph corresponds with a selection coordination between views; 12 of the paths involve projection expressions.

directed topology in which a core subgraph of variables and lexicals is surrounded by a shell of properties and controls. Like in Live Properties, two controls are coordinated whenever their properties are connected by at least one path through the core subgraph. Unlike in Live Properties, paths are directed whenever they contain directed edges. As a result, coordinated query graphs based on Coordinated Queries are:

- *Partially reflective.* Controls act as if they coordinate with themselves through each of their active (but not their passive, including lexical) properties. $[c \leftrightarrow p_{active} \leftrightarrow v]$

- *Highly reflexive.* Controls can also coordinate with themselves through different properties connected to a common variable, including those assigned lexical values. In particular, controls can coordinate with themselves via paths that involve expressions. $[c \leftrightarrow p_a \leftrightarrow v_1 \rightarrow x \rightarrow v_2 \leftrightarrow p_b \leftrightarrow c]$

- *Asymmetric.* Coordination between controls can be unidirectional. Modification of a variable by one control triggers notification of the other control. $[c_1 \leftrightarrow p_a \leftrightarrow v_1 \rightarrow x \rightarrow$

$v_2 \leftrightarrow p_b \leftrightarrow c_2]$

- *Closed (under query).* Controls that share a variable at the end of a path through the core subgraph are coordinated pairwise. Modification of a variable from anywhere in the graph notifies all dependent controls. $[v \leftrightarrow p_a \leftrightarrow c_i, v \leftrightarrow p_b \leftrightarrow c_j \Rightarrow c_i \leftrightarrow c_j \; \forall \; c_i, c_j \in C]$

- *Intransitive.* Coordination through paths in the core subgraph is not cumulative. $[c_1 \leftrightarrow c_2, c_2 \leftrightarrow c_3 \not\Rightarrow c_1 \leftrightarrow c_3]$

- *Cyclic.* Although direct coordinations are acyclic, indirect coordinations can be cyclic. Because expressions can refer to lexicals through variable operators, it is possible to form paths in the core subgraph in which lexicals depend on themselves directly or indirectly. $[v_1 \rightarrow x_a \rightarrow v_2 \rightarrow x_b \rightarrow v_1]$

- *Composable.* Graphs can be combined (possibly with overlapping nodes and edges) to build large visualizations from smaller ones. Subgraphs can be isolated to extract small visualizations from larger ones. In particular, subgraphs without controls or properties can be isolated and combined into multiple query plans for reuse across different visualizations.

Because Coordinated Queries has these mathematical properties, it is possible to refine the previous redefinition of the term "link" (section 3.3) as *a useful combination of coordinated queries amongst a set of views*. As it turns out, these combinations significantly increase the functionality of common types of interactive dependencies between views as compared with other systems. It thus becomes possible to extend vastly the useful and usable patterns of coordination that can be incorporated into multiview visualizations.

## 4.4 Limitations

### 4.4.1 Cycles

A shortcoming of the current implementation of Coordinated Queries is that it does not detect or handle cycles in coordinated query graphs. Straightforward search-based and stack-based algorithms [81] exist to detect cycles in general graphs. Because editing of coordinated query graphs is interactive, it would be necessary to perform cycle detection upon modification of any expression as well as for changes to assignment and reference relationships in the graph. Because the periphery of the graph is acyclic (section 3.3), it would not be necessary to redetect cycles for new slot and binding relationships.

Once detected, handling a cycle could involve implicit "pruning" of expressions so as to no longer refer to the next lexical in the cycle. Providing suitable error indication in this case could involve strong highlighting of all cyclic lexicals (and references to them within expressions) in the builder interface. (The implementation currently falls back on the stack-based cycle "detection"—in the form of stack overflow exceptions—in the underlying Java runtime system.)

In practice, cycles in coordinated query graphs arise very rarely. In particular, common uses of expressions that refer to other expressions—including invocable subexpressions, key and value expressions in aggregate and index operators, and function operators that take lexicals as arguments—only make sense for distinct expressions. In particular, the need for recursion has not yet arisen in the numerous Improvise visualizations created to this point. Nevertheless, the possibility of accidentally building cycles in the coordinated query graph during visualization construction makes detecting and handling them a desirable future addition to the implementation.

### 4.4.2 Unary Operations

Expression evaluation in Coordinated Queries is a unary operation; that is, expressions take a single record as input and return a single field as output. As a result, only unary query operations that iterate over independent records of a single table—such as projections and filters—are straightforward to implement. Nevertheless, through clever design it is also possible to implement binary and higher order operators using expressions.

For instance, the inner loop of the sort operation performs a binary comparison of comparators generated by applying a presort expression to the records of the input table. Similarly, it would be possible to implement an equijoin operation by performing a binary nested loop equality comparison of objects generated by applying two prejoin expressions to the records of two input tables. In cases such as these, query operations can be executed within Coordinated Queries using additional function operators that take appropriate utility expressions as arguments, much like aggregate and index operators do, but that must be implemented outside of Coordinated Queries just like calculations executed by existing operators.

Grouping queries in Improvise also work this way, using a function operator that takes a data set and a key expression as arguments. The operator constructs a two-column table in which the first column contains unique keys generated by evaluating the key expression on each record in the input data set. The second column contains subtables of records that generate the corresponding keys. Because the result of a grouping query is itself a data set, it can be further processed and rendered in a visualization like any other data set. Looking up the group of a particular key (using another function operator) likewise produces a visualizable data set.

## 4.5 Extensions

### 4.5.1 Extensible Lexical Types

Coordinated Queries has been implemented with the goal of providing the minimal set of query operations needed to reproduce most, if not all, of the multiview coordination patterns that have been utilized in information visualization tools to date. Table 8 summarizes the functional characteristics of the original five lexical types. The example visualizations in appendix A are all constructed using coordination patterns (chapter 6) based on these five types.

| *Lexical* | *Form* | *Input* | *Output* | *Purpose* |
|---|---|---|---|---|
| Projection | Many-To-Many | object[] | object[] | Item drawing |
| Filter | Many-To-One | object[] | boolean | Item hiding |
| Sort | Many-To-One | object[] | comparator | Item ordering |
| Expression | Many-To-One | object[] | object | Derived attributes |
| Info | None-To-One | void | data | Data access and preprocessing |

Table 8: Functional characteristics of the existing lexical types.

However, ongoing and future research in visual analytics is likely to uncover additional ways of connecting views in ways that cannot be reproduced with existing Coordinated Query components. Adding the capability to specify arbitrary lexical types in terms of expressions, inputs, and outputs would allow visualization researchers to provide the means to integrate new views, information sources, and visual techniques into a single visual abstraction architecture. For instance, table 9 summarizes the functional characteristics of three currently experimental lexical types that would allow specification of: (1) visual encoding of items independent of view type (Item Look); (2) behavioral encoding of items independent of view type (Item Feel); and (3) variable-to-property value transformations (Mutation).

| *Lexical* | *Form* | *Input* | *Output* | *Purpose* |
|---|---|---|---|---|
| Item Look | Many-To-One | object | glyph | Item appearance |
| Item Feel | Many-To-One | object | action | Item behavior |
| Mutation | One-To-One | object | object | Value transformation |

Table 9: Functional characteristics of experimental lexical types.

### 4.5.2 Expression Binding

A mechanism for transforming values between variables and passive properties (as described in section 3.5.2) would allow visualization designers to define the Navigation, Selection, and Look&Feel properties of views and other controls in terms of data and queries, much like Information properties. Applications of such a mechanism might include:

- Filling the background of a 3-D stereogram with the color corresponding to an attribute of an item selected in another view.
- Setting the text of a label control to the number of items at least partially visible in a scatter plot.
- Using a bold font in the same label when at least some items fall completely outside the scatter plot.
- Labeling an axis with formatted dates rather than a decimal number of days.
- Using the number of records in a data set to determine a uniform pixel height for all rows in a table view.
- Using aggregation on a data attribute to determine the minimum and maximum values of a slider or scrollbar.

One possible way to implement such a mechanism, *expression binding*, would involve binding a property to a variable whose value is an expression rather than an object of the property's

type. The "value" of the property would be calculated by evaluating the expression without an input record. Another way would be to pair a mutation property with each passive property of a view. The "value" of a property would be calculated by evaluating the mutation expression with the value of the property's bound variable as the input field. The latter technique has been implemented and is being considered for permanent inclusion in Coordinated Queries.

## 4.6 Summary

Coordinated Queries is a flexible, yet relatively high-level visualization query language for coordinating data access, data processing, and data rendering across multiple views. Visual abstraction takes place through expressions that specify how to map data attributes into graphical attributes in views. Multiple data sets can be loaded, indexed, grouped, filtered, sorted, and projected as a function of interactive navigation and selection in and between multiple views. The combination of Live Properties and Coordinated Queries in Improvise enables open-ended visual analysis by allowing users to construct and explore highly-coordinated visualizations of multiple simultaneous data sets interactively.

# Chapter 5

# Improvise

## 5.1 Overview

Improvise consists of a graphic user interface on top of a modular library of visualization components, forming a fully implemented, web capable Java application that appears and behaves like other "office productivity" applications based on the multiple document desktop metaphor. Users browse visualizations using mouse and keyboard input gestures to navigate in space and select data items. Although interaction is confined to a single view at any given time, tightly coupled coordination between views produces the illusion of browsing each visualization interface as a coherent whole. Improvise has been used to build a wide variety of visualizations (appendix A), each of which is a self-contained Extensible Markup Language (XML) [20] document that can be saved, opened, copied, and shared as a regular file.

In the Improvise user interface, designers create, layout, parameterize, and coordinate views using implementations of the Live Properties coordination model (chapter 3) and the Coordinated Queries visual abstraction language (chapter 4). Building occurs inside the same top-level window that contains views. Moreover, building is interactive; changes take effect immediately without the need for a separate compilation stage. This live, amodal interface design allows users to switch rapidly between building and browsing. The goal is to enable exploration that is free form and open-ended, particularly during initial inspection of newly encountered data sets. In other words, the goal is to support *improvisational visualization*.

## 5.2 Users

Members of the rapidly growing Improvise community fall into four categories determined by their visualization background, programming skill, and analytic motivation:

- *Researchers* use Improvise as a testbed for rapid prototyping and usability evaluation of new visualization techniques.
- *Developers* design and implement new view, data source, processing algorithm, and function operator modules in order to extend and enhance Improvise functionality. Some developers are even reimplementing the functionality of their own visualization systems and toolkits in Improvise.
- *Designers* plan, construct, debug, test, and deploy new Improvise visualizations for ongoing evaluation and routine operation by end-users.
- *End-users* use existing Improvise visualizations to analyze data from their own knowledge domains and to browse data with which they are less familiar.

These categories are not mutually exclusive. Researchers can develop their own Improvise modules in order to evaluate their techniques. Developers can design and use visualizations in order to evaluate the correctness, flexibility, performance, and usability of their views and algorithms. The tight integration of building and browsing in the Improvise user interface is intended especially to support a fifth category of user:

- *Explorers* combine the roles of designers and end-users, extending and redesigning Improvise visualizations on the fly during open-ended exploration of their data.

## 5.3 Requirements

Improvise is based on strict implementations of its underlying coordination and visual abstraction models, and is built around a library of data access modules, algorithms, and views. In order to support building and browsing operations, the user interface and library implementations must support the activities of the various kinds of visualization users.

For developers, the library must support an application programming interface (API) for adding software modules that implement:

- data access,
- queries and other data transformation algorithms,
- views, and
- operators in the visual abstraction language.

In particular, the API must support the definition of operators—including their input and output data object types—that allow designers to express relationships between data, queries, and views.

For designers, the user interface must support the ability to:

- access data sets (and metadata) from local or remote sources in various formats,
- create and position views on the screen,
- specify how navigation and selection affects views,
- specify queries on data,
- parameterize queries in terms of interaction, and

- attach data sets and queries to views.

In particular, designers must be able to specify the entire appearance and behavior of a visualization from within the user interface, without resorting to programming or other workarounds for limitations in the user interface (which would effectively turn designers into developers). It is also desirable to provide a way to deal with the problem of limited screen real estate for large visualizations.

For end-users, the user interface must:

- run in the user's normal working environment,
- require no programming or design activities, and
- provide a way to disseminate analytical results.

In particular, it is desirable for the user interface to run easily on any platform, allow visualizations to be opened and saved as normal documents for sharing between users, and provide the ability to bookmark or screen capture visualizations in different graphical states.

For improvisational visualization, the user interface must exceed the requirements of designers and end-users by:

- supporting live building of complete browser interfaces, including immediate designing, debugging, and testing of intended functionality,
- facilitating collaboration between end-users and designers to turn analytical questions into structural changes (through remote, nearby, or side-by-side efforts to communicate and effect rapid visualization prototyping and polishing), and
- enabling rapid switching between building and browsing to perform more extensive exploratory visualization by modifying visualization views and queries on the fly.

In particular, it is highly desirable for explorers to be able to:

- see all raw data quickly to make decisions about how to visualize it,
- rapidly create and lay out views,
- rapidly attach data and queries to views,
- rapidly modify queries,
- store, copy, and reuse views,
- copy-and-paste/drag-and-drop visualization components, and
- use macros to build common multiview constructions.

These capabilities are desirable for non-exploring designers as well.

## 5.4 User Interface

Improvise follows the multiple document model in which one or more visualizations run in parallel, each in its own top-level window. The builder interface is built around a set of editor dialogs, local to each visualization window, in which designers access data, build expressions for querying data, and create, coordinate, and layout views. The browser interface consists of the set of views as they are laid out inside each visualization window. All changes made in editing dialogs take place immediately and reversibly, resulting in an incremental, organic style of visual exploration.

### 5.4.1 Data

#### Metadata

In accordance with the relational model, metadata in Improvise consists of schemas that describe columns in tabular data sets by name and object type. Schemas are used for two distinct purposes: to validate the contents of accessed data sets, and to define the input and output characteristics of query expressions. As a strongly-typed declarative language, Coordinated Queries uses schemas to enforce legal evaluation of expressions over tables of records, much the same way that other strongly typed languages use function prototypes. For instance, projections are defined in terms of input and output schemas (section 4.2.2). Separating schema definition from data access allows expressions to be applied to all data sets of the same schema, enabling reuse of expressions for multiple or changing data sets.

Schemas are built under the `Schema` tab in the lexicon editor (figure 24). To build a schema, the visualization designer:

1. Starts by creating new empty schemas or cloning existing ones.
2. Chooses a schema to edit.
3. Gives the schema a name. (All lexical objects have unique names for convenient reference throughout the builder interface.)
4. Builds a list of attributes for the selected schema.
5. Enters the name and type of each attribute.
6. Optionally exports or imports the schema's attributes to or from a file.
7. Assigns the schema to a variable for referencing in expressions.

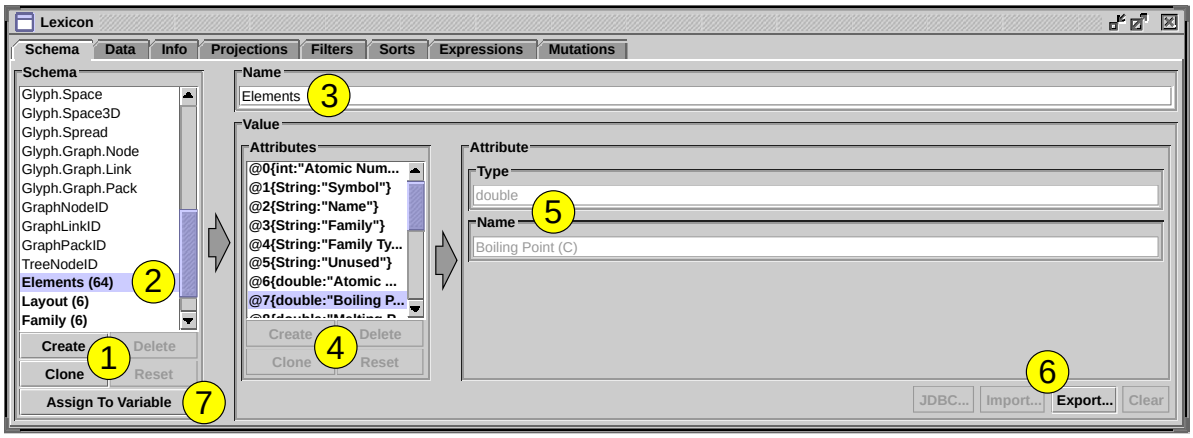



Figure 24: Editing metadata.

Schemas can be imported and exported to flat (space-delimited, tab-delimited, CSV), XML, and DBF formats. Import and export is implemented using a simple inheritance-based API that can be extended for access to metadata in database catalogs and other information sources.

### Access

There are two ways of accessing data in Improvise: literal and functional. Literal access involves identification of a data set from a particular source, such as a local file path, database relation name, or web service URL.

Literal access to data sets occurs under the `Data` tab in the lexicon editor (figure 25). To create a reference to a literal data set, the visualization designer:

1. Selects the schema of the data to be accessed.
2. Creates a new empty data reference or clones an existing one.
3. Picks a data set of the selected schema.
4. Imports data from a file.
5. Examines the rows and columns of the imported data set.

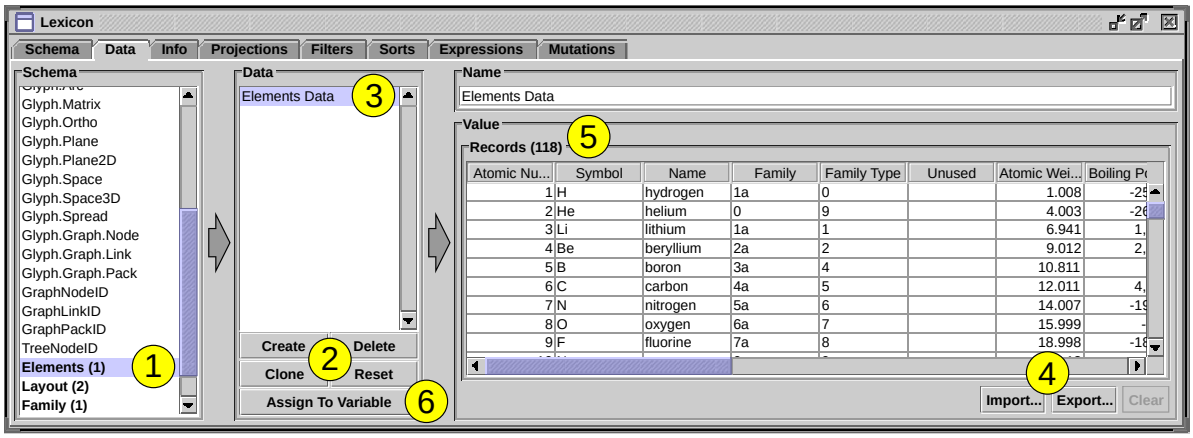



Figure 25: Creating references to literal data sets.

6. Assigns the data to a variable for binding to view properties and referencing in expressions.

Functional access produces data sets by evaluating an info expression. Using variable operators in info expressions makes it possible to load data as a function of interaction and even in terms of the information contained in other data sets. As a result, functional access is much more flexible than literal access.

Info expressions are built under the `Info` tab in the lexicon editor (figure 26). To build a lexical info, the visualization designer:

1. Specifies the schema of records to be produced by the info expression.

2. Creates a new lexical info or clones an existing one.

3. Picks an info of the selected schema.

4. Builds an expression using function operators to generate, access, or transform data. In this case, the data:

5. comes from a flat text file,

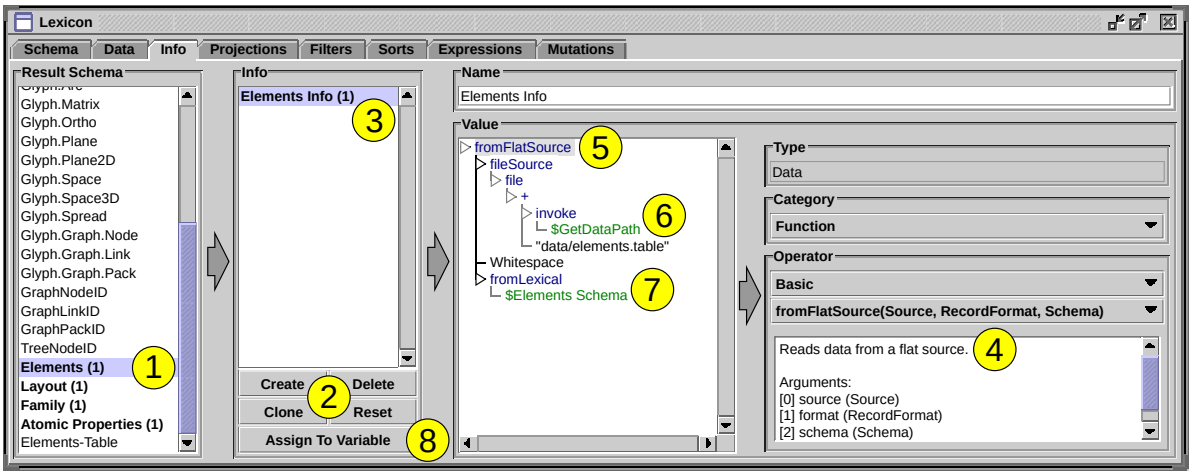



Figure 26: Creating references to functionally specified data sets.

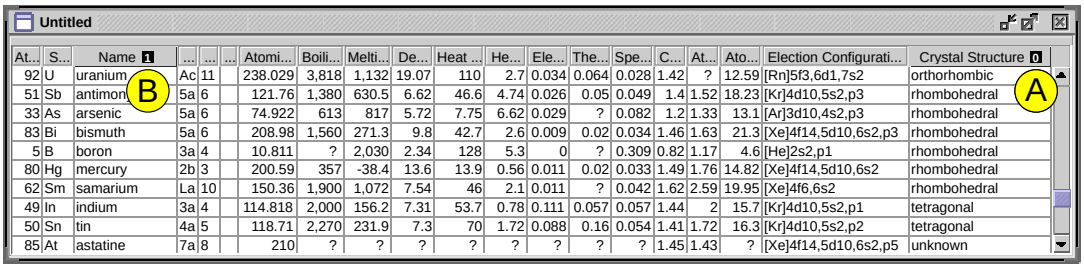
6. at a path location relative to the visualization document,

7. using the schema in a lexical schema variable to validate the records of the file.

8. Assigns the info to a variable for binding to view properties and referencing in expressions that perform queries.

Functional access is particularly useful for two-tiered data access, in which a master data set describes a collection of distinct but related data sets. In the visualization in appendix A.2, for example, a master list summarizes and names files that contain sequential experimental runs on a scientific apparatus. An info expression describes how to access data for a particular experiment as a function of a selection on the master data set, itself displayed in a view through functional access using an info expression that produces the names of matching filenames in a local directory.

Data sets can be accessed from local sources in flat (space-delimited, tab-delimited, CSV), XML, and DBF formats. Data access modules are implemented using a simple inheritance-based API that can be extended to add the ability to import data from other sources. Because modules that conform to the API only need to provide the size and schema of the data set and




| At... | S... | Name | ... | ... | ... | Atomi... | Boili... | Melti... | De... | Heat ... | He... | Ele... | The... | Spe... | C... | At... | Ato... | Election Configurati... | Crystal Structure |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 92 | U | uranium | Ac | 11 | | 238.029 | 3,818 | 1,132 | 19.07 | 110 | 2.7 | 0.034 | 0.064 | 0.028 | 1.42 | ? | 12.59 | [Rn]5f3,6d1,7s2 | orthorhombic |
| 51 | Sb | antimony | 5a | 6 | | 121.76 | 1,380 | 630.5 | 6.62 | 46.6 | 4.74 | 0.026 | 0.05 | 0.049 | 1.4 | 1.52 | 18.23 | [Kr]4d10,5s2,p3 | rhombohedral |
| 33 | As | arsenic | 5a | 6 | | 74.922 | 613 | 817 | 5.72 | 7.75 | 6.62 | 0.029 | ? | 0.082 | 1.2 | 1.33 | 13.1 | [Ar]3d10,4s2,p3 | rhombohedral |
| 83 | Bi | bismuth | 5a | 6 | | 208.98 | 1,560 | 271.3 | 9.8 | 42.7 | 2.6 | 0.009 | 0.02 | 0.034 | 1.46 | 1.63 | 21.3 | [Xe]4f14,5d10,6s2,p3 | rhombohedral |
| 5 | B | boron | 3a | 4 | | 10.811 | ? | 2,030 | 2.34 | 128 | 5.3 | 0 | ? | 0.309 | 0.82 | 1.17 | 4.6 | [He]2s2,p1 | rhombohedral |
| 80 | Hg | mercury | 2b | 3 | | 200.59 | 357 | -38.4 | 13.6 | 13.9 | 0.56 | 0.011 | 0.02 | 0.033 | 1.49 | 1.76 | 14.82 | [Xe]4f14,5d10,6s2 | rhombohedral |
| 62 | Sm | samarium | La | 10 | | 150.36 | 1,900 | 1,072 | 7.54 | 46 | 2.1 | 0.011 | ? | 0.042 | 1.62 | 2.59 | 19.95 | [Xe]4f6,6s2 | rhombohedral |
| 49 | In | indium | 3a | 4 | | 114.818 | 2,000 | 156.2 | 7.31 | 53.7 | 0.78 | 0.111 | 0.057 | 0.057 | 1.44 | 2 | 15.7 | [Kr]4d10,5s2,p1 | tetragonal |
| 50 | Sn | tin | 4a | 5 | | 118.71 | 2,270 | 231.9 | 7.3 | 70 | 1.72 | 0.088 | 0.16 | 0.054 | 1.41 | 1.72 | 16.3 | [Kr]4d10,5s2,p2 | tetragonal |
| 85 | At | astatine | 7a | 8 | | 210 | ? | ? | ? | ? | ? | ? | ? | ? | 1.45 | 1.43 | ? | [Xe]4f14,5d10,6s2,p5 | unknown |

Figure 27: Previewing a data set in a table view. Successive clicking in the table header causes sorting (A) and subsorting (B) of arbitrary columns in increasing and/or decreasing order.

an iterator over its records, it is possible to implement a wide variety of streaming and caching access protocols that are compatible with Coordinated Queries.

#### Rapid Preview

Improvise provides a menu shortcut for rapid preview of any literal or functional data set. The shortcut automatically creates a table view, binds the chosen lexical data or info variable to the view's corresponding property, then lays out the view in a new frame (figure 27). Previews provide a quick glimpse as a first analysis step, and frequently remain in the developing visualization as a detail view—often with lexical projections bound to them to provide richer visual encodings.

### 5.4.2 Views

#### Creation and Parameterization

Figure 28 shows the process of creating and parameterizing a table view that displays properties of the chemical elements (as a part of the *elements* visualization described in appendix A.4). To build views, the visualization designer:

1. Starts by choosing the desired type of view or control.

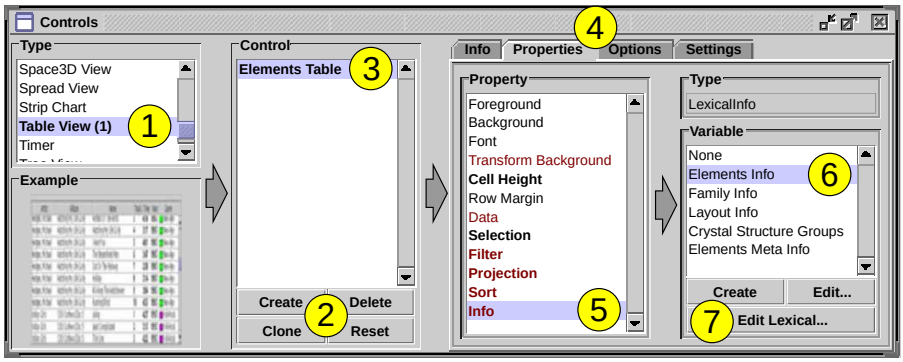



Figure 28: Creating and editing views and other controls.

2. Creates new instances (or clones old ones) of the chosen control type, as desired.

3. Selects a control for editing.

4. Coordinates the control by binding variables to its properties. Non-property characteristics of controls can also be edited, including name, annotation, flags for simple boolean options, and miscellaneous settings.

5. Selects the table's `Info` property for binding.

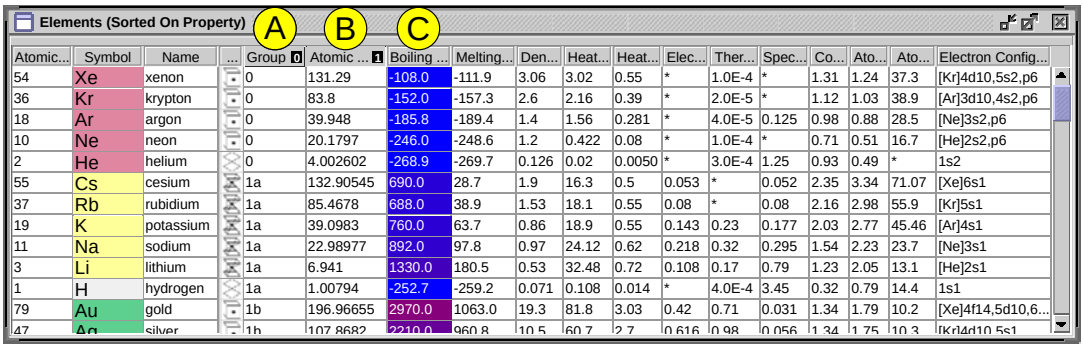
6. Binds a variable having type `LexicalInfo` to the property. Selecting `None` unbinds the property.

7. Optionally creates and edits variables (and lexical objects assigned to them) as needed.

Binding non-`Info` properties transforms the table in figure 27 into the one in figure 29.

### Display and Layout

Each visualization window displays a desktop-like workspace. The desktop is split across one or more *pages* that mimic tabbed folders. Figure 30 shows the process of creating additional pages in a visualization window. To build pages, the visualization designer:




| Atomic... | Symbol | Name | ... | Group 0 | Atomic ... 1 | Boiling ... | Melting... | Den... | Heat... | Heat... | Elec... | Ther... | Spec... | Co... | Ato... | Ato... | Electron Config... |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 54 | Xe | xenon | | 0 | 131.29 | -108.0 | -111.9 | 3.06 | 3.02 | 0.55 | * | 1.0E-4 | * | 1.31 | 1.24 | 37.3 | [Kr]4d10,5s2,p6 |
| 36 | Kr | krypton | | 0 | 83.8 | -152.0 | -157.3 | 2.6 | 2.16 | 0.39 | * | 2.0E-5 | * | 1.12 | 1.03 | 38.9 | [Ar]3d10,4s2,p6 |
| 18 | Ar | argon | | 0 | 39.948 | -185.8 | -189.4 | 1.4 | 1.56 | 0.281 | * | 4.0E-5 | 0.125 | 0.98 | 0.88 | 28.5 | [Ne]3s2,p6 |
| 10 | Ne | neon | | 0 | 20.1797 | -246.0 | -248.6 | 1.2 | 0.422 | 0.08 | * | 1.0E-4 | * | 0.71 | 0.51 | 16.7 | [He]2s2,p6 |
| 2 | He | helium | | 0 | 4.002602 | -268.9 | -269.7 | 0.126 | 0.02 | 0.0050 | * | 3.0E-4 | 1.25 | 0.93 | 0.49 | * | 1s2 |
| 55 | Cs | cesium | | 1a | 132.90545 | 690.0 | 28.7 | 1.9 | 16.3 | 0.5 | 0.053 | * | 0.052 | 2.35 | 3.34 | 71.07 | [Xe]6s1 |
| 37 | Rb | rubidium | | 1a | 85.4678 | 688.0 | 38.9 | 1.53 | 18.1 | 0.55 | 0.08 | * | 0.08 | 2.16 | 2.98 | 55.9 | [Kr]5s1 |
| 19 | K | potassium | | 1a | 39.0983 | 760.0 | 63.7 | 0.86 | 18.9 | 0.55 | 0.143 | 0.23 | 0.177 | 2.03 | 2.77 | 45.46 | [Ar]4s1 |
| 11 | Na | sodium | | 1a | 22.98977 | 892.0 | 97.8 | 0.97 | 24.12 | 0.62 | 0.218 | 0.32 | 0.295 | 1.54 | 2.23 | 23.7 | [Ne]3s1 |
| 3 | Li | lithium | | 1a | 6.941 | 1330.0 | 180.5 | 0.53 | 32.48 | 0.72 | 0.108 | 0.17 | 0.79 | 1.23 | 2.05 | 13.1 | [He]2s1 |
| 1 | H | hydrogen | | 1a | 1.00794 | -252.7 | -259.2 | 0.071 | 0.108 | 0.014 | * | 4.0E-4 | 3.45 | 0.32 | 0.79 | 14.4 | 1s1 |
| 79 | Au | gold | | 1b | 196.96655 | 2970.0 | 1063.0 | 19.3 | 81.8 | 3.03 | 0.42 | 0.71 | 0.031 | 1.34 | 1.79 | 10.2 | [Xe]4f14,5d10,6... |
| 47 | Ag | silver | | 1b | 107.8682 | 2210.0 | 960.8 | 10.5 | 60.7 | 2.7 | 0.616 | 0.98 | 0.056 | 1.34 | 1.75 | 10.3 | [Kr]4d10,5s1 |

Figure 29: Viewing a filtered, sorted, and projected data set in a table view. Rapid column sorting (A) and subsorting (B) override the default ordering on increasing boiling point (C) that is specified by the view's sort property.

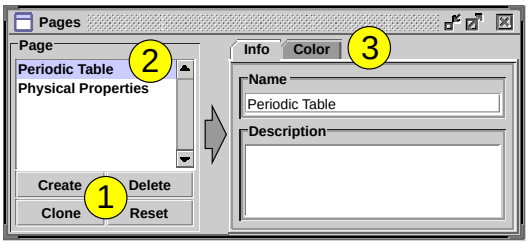



Figure 30: Creating pages on the visualization desktop.

1. Starts by creating or cloning new pages.

2. Selects a page for editing.

3. Edits the page's basic information (name and annotation) and background color.

Multiple pages can be used to increase the overall usable area of a visualization substantially. Views can coordinate with each other even when on different pages. However, most visualization designs should minimize cross-page coordination in order to avoid user confusion that might arise from unexpected side effects of interaction when moving between pages.

Frames on the visualization desktop contain trees of nested panels with views and other controls at the leaves. In any given panel, one of several constraint-based spatial layout schemes positions sibling views and subpanels relative to each other. Constraint-based layout is more flexible than both free form layout (in which an absolute bounding box is specified for each

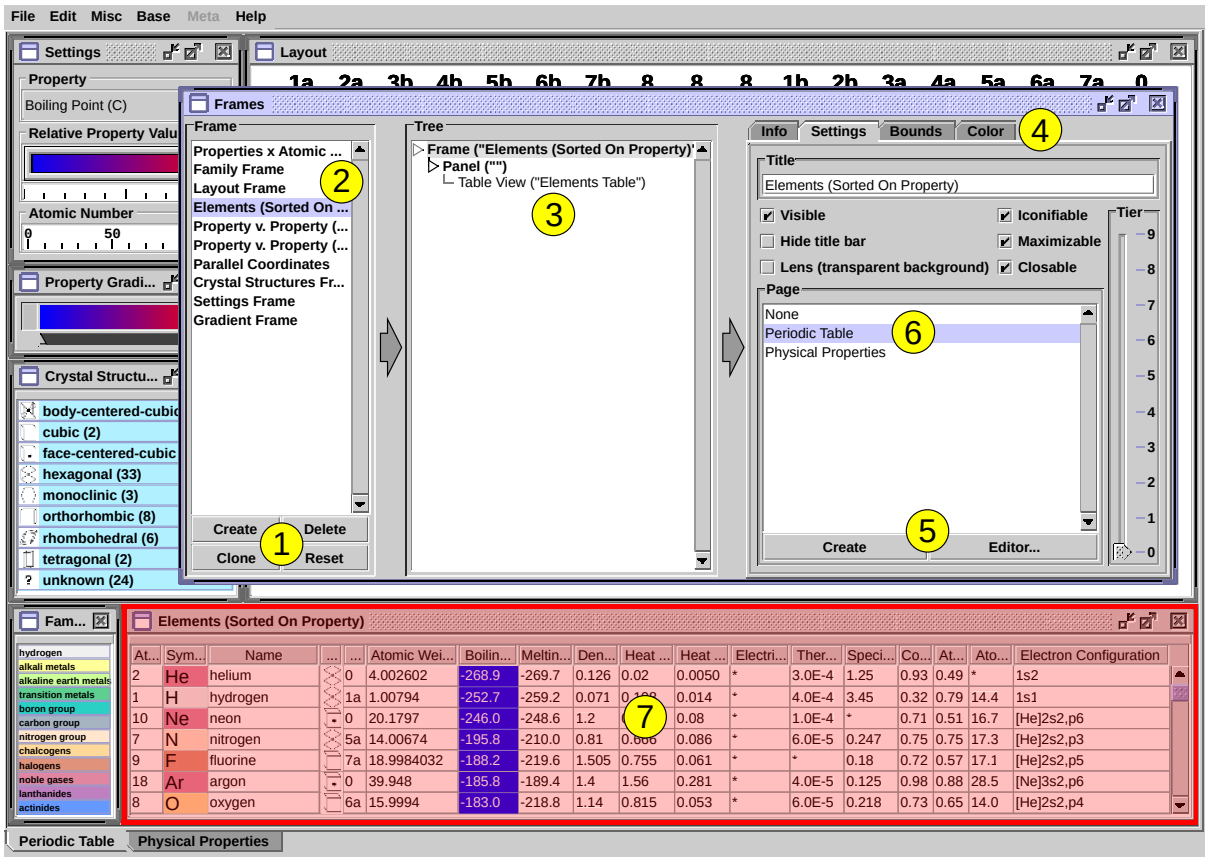



Figure 31: Creating internal frames.

control) and heuristic layout (in which views are automatically tiled on the screen in space-filling fashion) [100, 75].

Figure 31 shows the process of creating and parameterizing internal frames on the visualization desktop. To build frames, the visualization designer:

1. Starts by creating new frames or cloning old ones.

2. Selects a frame for editing.

3. Builds a tree of panels with views at the leaves and the frame at the root. Selecting the root allows editing of the frame's non-layout characteristics.

4. Edits the frame's basic information (name and annotation), bounding box on the visualization desktop, fill color, and decoration settings.

5. Optionally creates and edits new pages as needed.

6. Selects which page displays the frame. Selecting `None` makes the frame effectively invisible.

7. Gets immediate feedback for all changes to the selected frame.

Figure 32 shows the process of building view containment trees inside frames. Each tree consists of a hierarchy of panels with a frame at the root and views at the leaves. The visualization designer:

1. Selects a panel in its root frame's hierarchy.

2. Edits the panel's basic information (name and annotation), decorative border, and fill color.

3. Selects one of several layout schemes for the panel.

4. Adds subpanels to create branches in the containment hierarchy.

5. Adds existing views to the panel, optionally creating new views as desired.

6. Constrains each view or subpanel to a particular position in the selected layout.

7. Gets immediate feedback for all changes to the selected panel.

Cloned frames are complete deep copies, containing fully functional copies of the panel tree and views, and are laid out identically. Because cloned views bind to the same variables, they are automatically and completely coordinated (*isocoordinated*) with the originals. Cloning

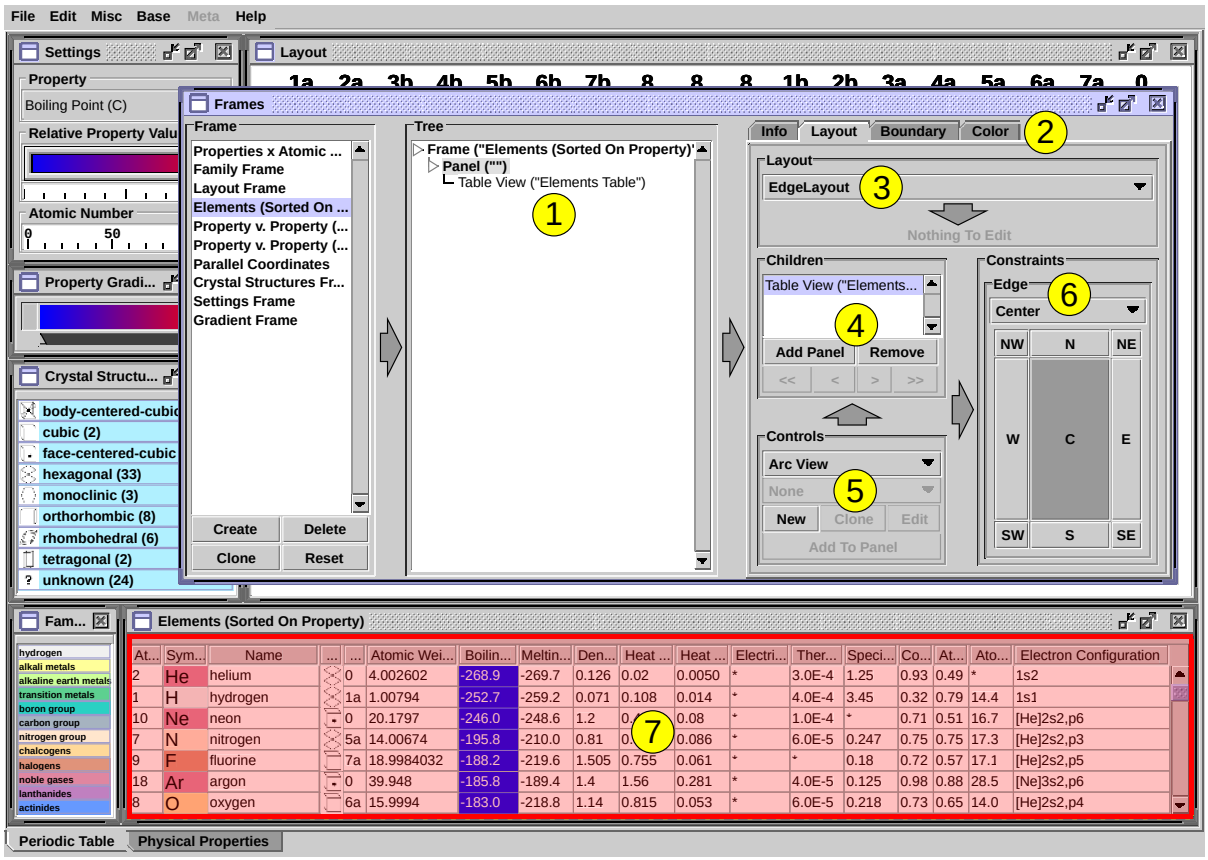



Figure 32: Laying out views in frames by building a tree of nested panels.

frames allows reuse of complex layouts without multiplication of time and effort, such as for placing identical frames on different pages. It is also a convenient starting point for minimizing the effort of building multiple frames that have minor differences.

Frame and panel editing involves a form of brushing in which selected frames and panels are highlighted in the visualization itself (step 7 in figures 31-32), intended to help designers quickly locate layout elements during editing.

### 5.4.3 Queries

Construction of projections, filters, and sorts occurs under the corresponding tabs in the lexicon editor (figure 33). To create a lexical projection, the visualization designer:

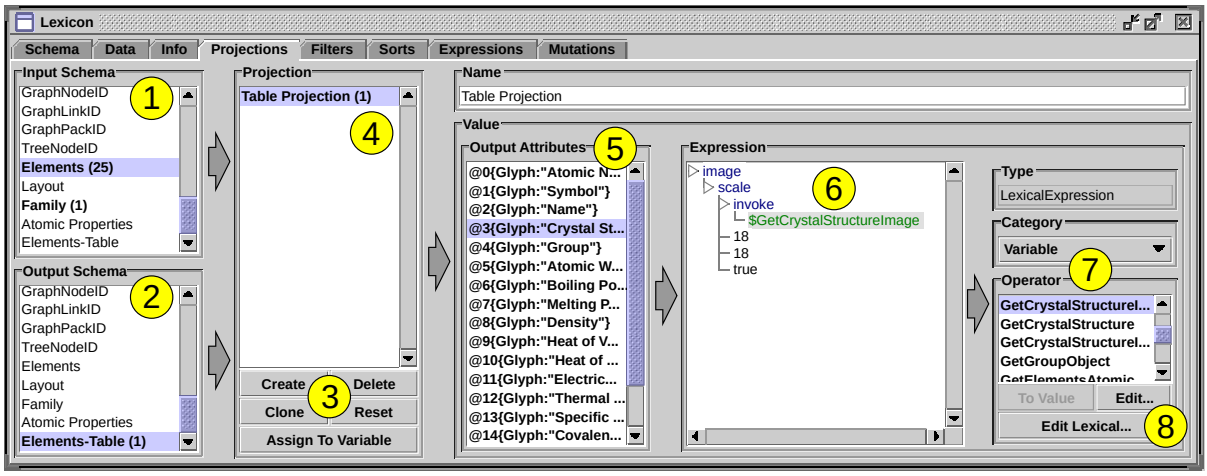



Figure 33: Building lexical query objects.

1. Selects the input schema of the projection to be edited.

2. Selects the output schema of the projection to be edited.

3. Creates a new projection or clones an existing one, as desired.

4. Picks a projection of the selected input and output schemas.

5. Selects one of the projection's output attributes.

6. Edits the expression that generates that output attribute for each record. In this case, the expression invokes a user-defined subexpression,

7. by referencing a variable that has the subexpression as its value.

8. Optionally jumps to the subexpression for editing.

Because filters and sorts have no output schema, the corresponding tabs in the lexicon editor omit selection of output schema (step 2) and output attribute (step 5).

Expressions can also be defined independently of lexical query objects. The resulting user-defined functions serve as reusable subexpressions that can be invoked by multiple projections,

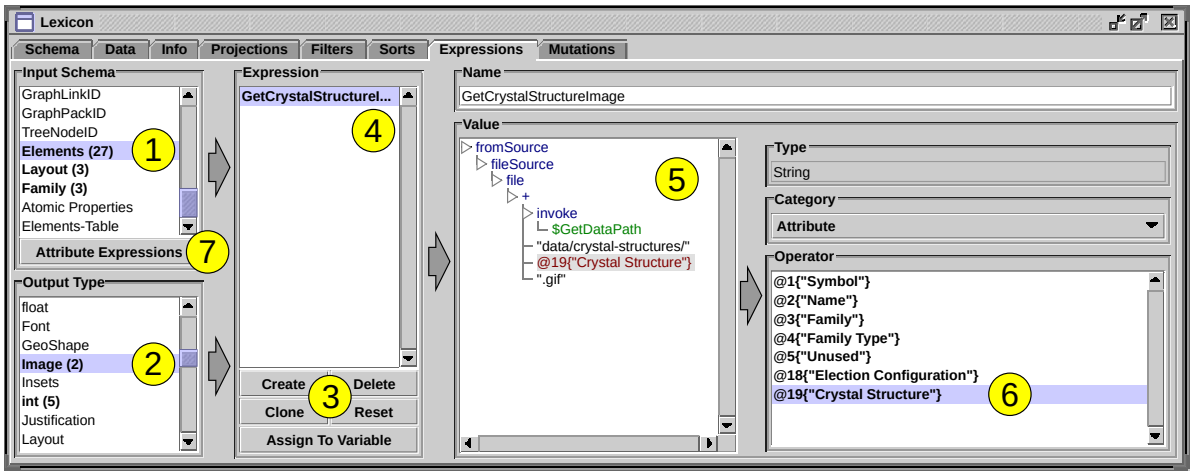



Figure 34: Building user-defined functions (invocable subexpressions).

filters, and sorts. Moreover, they provide a way to calculate arbitrary derived attributes of data records for use in aggregation, grouping, and indexing query operations. Figure 34 shows the process of creating independent expressions. The visualization designer:

1. Selects the input schema (argument types) of the expression to be edited.

2. Selects the output attribute (return type) of the expression to be edited.

3. Creates a new expression or clones an existing one, as desired.

4. Picks an expression of the selected argument and return types.

5. Builds the expression tree,

6. in terms of the input attributes.

7. Optionally uses a shortcut to create simple expressions for accessing input attributes.

Individual expressions are constructed in a tree-based editor. The user builds each expression top-down, by choosing an operator for each position in the tree. Subexpressions are automatically appended whenever the chosen operator takes arguments. Editing this way takes

a little getting used to, but has the advantage of being syntactically constrained. Editing is live; the visualization reflects changes immediately. Cloning entire projections, filters, and sorts allows users to experiment with variations of expressions quickly and reversibly. Large libraries can be built up for reuse or rapid switching during visualization design and exploration.

Figure 35 shows the process of building an expression for visually encoding atomic number as colored text in the table view in figure 29. The visualization designer:

1. Starts by selecting the function operator that generates the desired glyph.
2. Leaves the foreground (text) color as the default, black.
3. Selects the conditional (if-then-else) function operator for the background (fill) color.
4. Sets the conditional test to be whether of not the evaluated record has been selected. Because the default argument of the `isSelected` function operator is the empty selection, at this point the test will always evaluate to false.
5. Makes `isSelected` refer to the selection variable shared by the table and other views. At this point, the test depends on whether or not the element has been selected in any of the views. However, at this point true and false results both produce black.
6. Sets the color of selected elements to the `<sky blue>` constant operator.
7. Sets the color of unselected elements to the `<white>` constant operator.
8. Leaves the text font as the default.
9. Prepares to cast the `atomic number` attribute (an integer) into a generic object, to comply with strong type enforcement of the fourth argument of the `text` glyph.

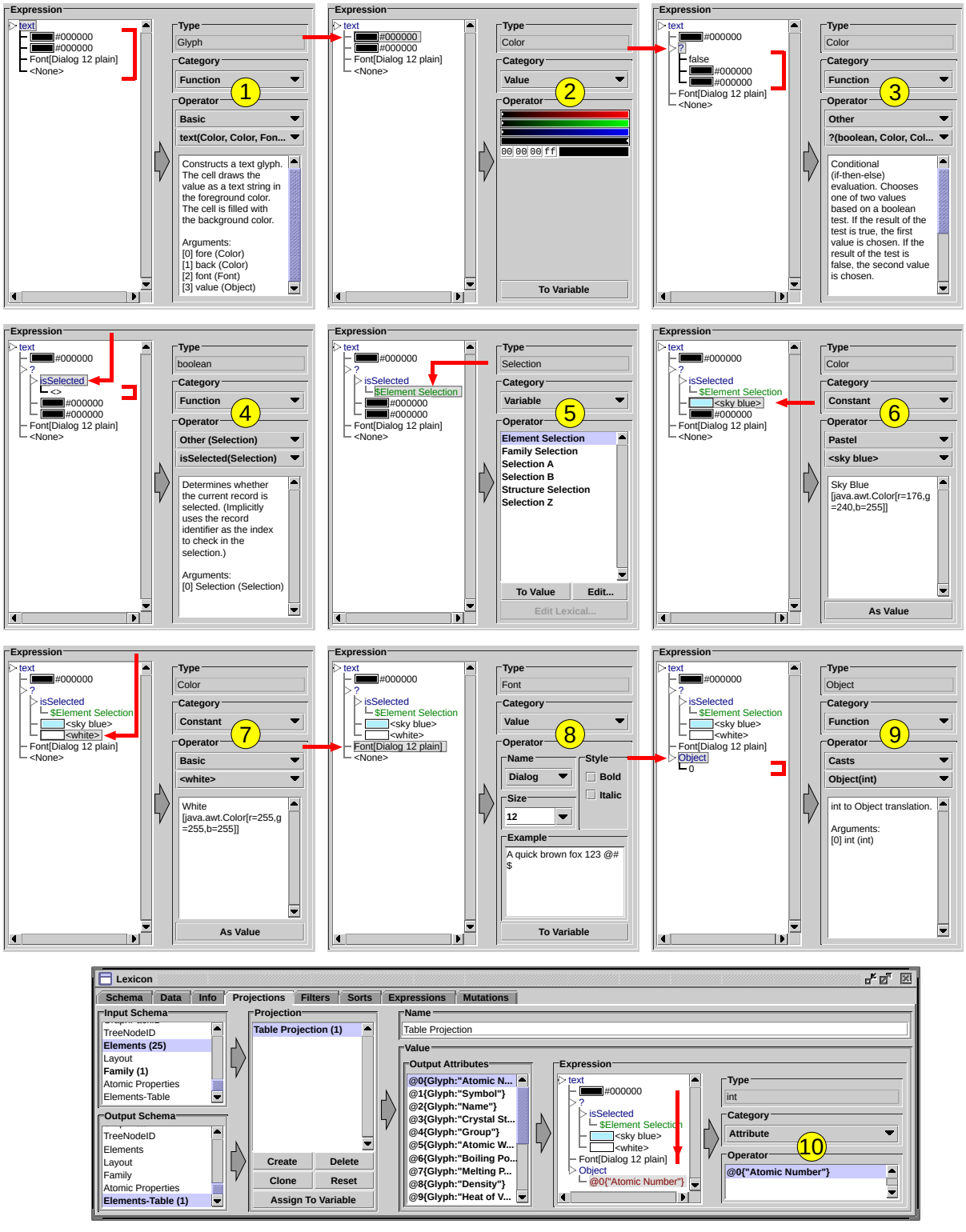



Figure 35: Top-down expression construction.

10. Chooses `atomic number` as the data attribute to display as text in the table view's first column.

The expression editor incorporates several improvements over standard tree interfaces, including syntactic coloring, branch highlighting, and automatic expansion and collapse of subexpression trees. The editor also supports full cut-copy-paste of subexpressions, including drag-and-drop between multiple editors. Pasting takes into account the type of the operator being replaced and the input schema of the expression being edited. Attribute operators in pasted subexpressions are replaced with attribute operators corresponding to the input schema in a best-effort way that attempts to match attribute types and names. Pasting between different visualizations is also supported in a limited manner by replacing variable operators with value operators of equivalent value at the time of pasting.

Sometimes the user wants to assign a variable operator to a subexpression when no variable of the return type of the subexpression exists. In this case, a variable of the right type is created with an appropriate default value.

### 5.4.4 Coordination

Figure 36 shows the process of creating and editing the values of two variables used to coordinate views in the *elements* visualization. To build variables, the visualization designer:

1. Starts by choosing the desired type of the variable.

2. Creates new instances (or clones old ones) of the chosen type, as desired.

3. Selects a variable for editing.

4. Edits the variable's basic information (name and annotation), label, and interactive state (focused, editing, locked) flags.

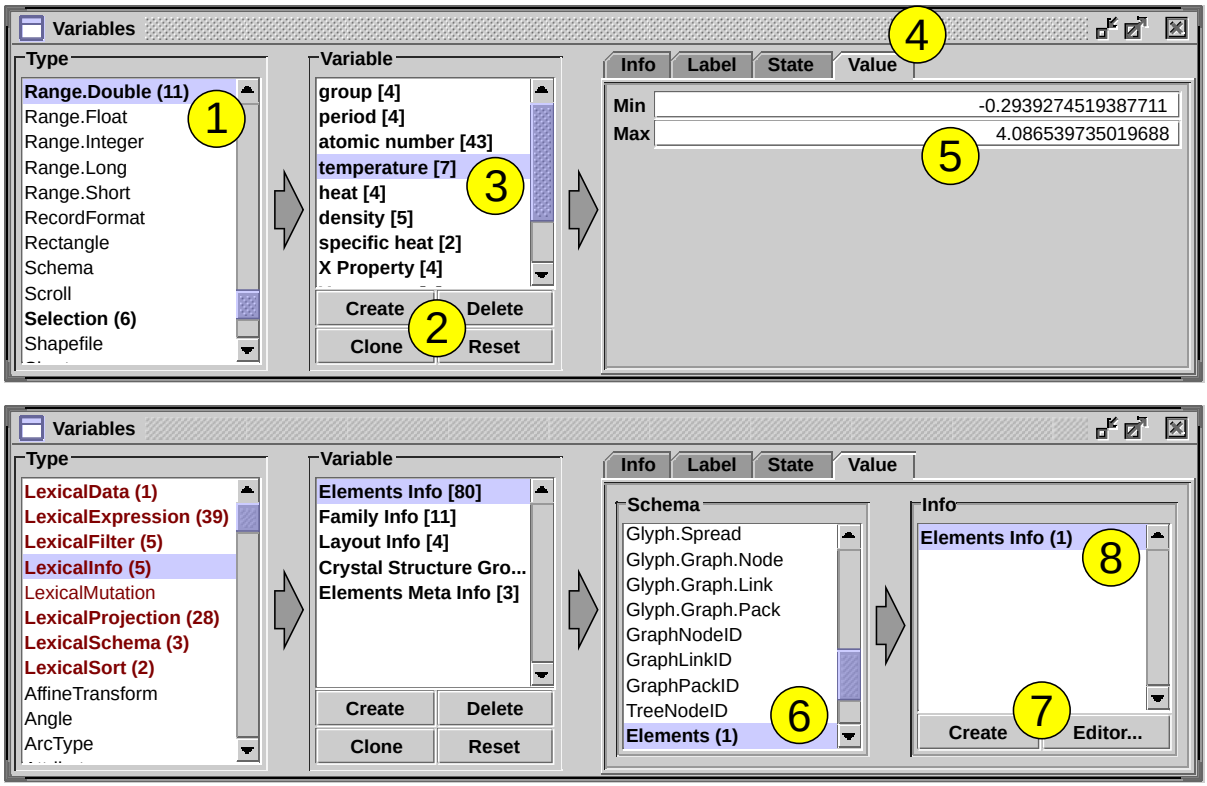



Figure 36: Building variables.

5. Edits the variable's value in a type-specific editor. For most non-lexical objects, the editor enables direct editing of the value itself (in this case, by entering the minimum and maximum of a range).

For variables that have lexical objects as their values, the visualization designer:

6. Chooses input and/or output characteristics for the desired lexical object.

7. Optionally creates and edits lexical objects by jumping to the lexicon editor.

8. Selects which lexical object to assign to the variable.

Once created, variables can be bound to control properties and referred to by variable operators in expressions. Building multiview visualizations with complex interactive behaviors

is thus a matter of constructing coordinated query graphs by specifying how views and queries depend on each other in terms of data sets and shared interactive parameters.

Deleting a variable renders all variable operators that refer to it invalid. To avoid subsequent evaluation errors, expressions automatically replace deleted variable operators with value operators having the value of the variable at the time of deletion.

### 5.4.5 Enhancements

The Improvise user interface has several enhancements designed to facilitate visualization construction and exploration, including:

- *Predefined macros* create, lay out, and navigationally coordinate common multiview constructions, such as scatter plot matrices with axis controls. Future work will address the need for user-defined macros to save and instantiate arbitrary visualization fragments, including coordinated query graph templates.

- *Built-in queries* speed up design and exploration by automatically displaying data in views using appropriate default queries. Examples include rapid preview and multicolumn sorting in table views.

- *Ubiquitous deep copying* allows the designer to duplicate every object with dependencies intact, including entire visualizations. Deep cloning is useful for experimental variation of expressions during exploration. Working on unconnected copies of expressions is also useful when immediate updates are distracting or slow.

- *Scaled screenshots* allow users to capture screenshots of Improvise visualizations in Scalable Vector Graphics (SVG) [46] and Portable Network Graphics (PNG) [39] formats. PNG screenshots can be captured at scaled resolutions for use in publications or on web

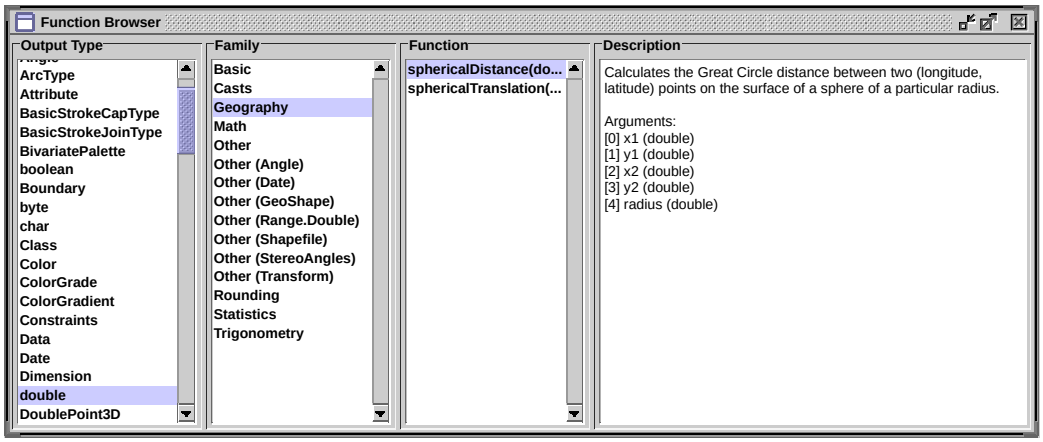



Figure 37: Browsing function operators.

pages. (Most of the figures shown throughout this dissertation, including the Improvise visualization screenshots, were captured either as SVGs or 4x resolution PNGs using this feature.)

- *Function operator browsing* gives designers a compact way to explore documentation on the Improvise library of over 1400 function operators. The user interface (figure 37) provides a multi-column window for browsing these functions by return type and family.

- *Integrated metavisualization* allows designers and users to visualize the coordination structure of Improvise visualizations *in situ* (described in chapter 7).

## 5.5 Software Architecture

The overall software architecture of Improvise is shown in figure 38. The architecture consists of an integrated building and browsing user interface on top of a modular, extensible library of visualization components. The architecture offers flexibility during design and exploration by allowing users to interactively create and connect data sets, queries, and views in terms of interactive parameters. Building occurs inside the same top-level window that contains views.

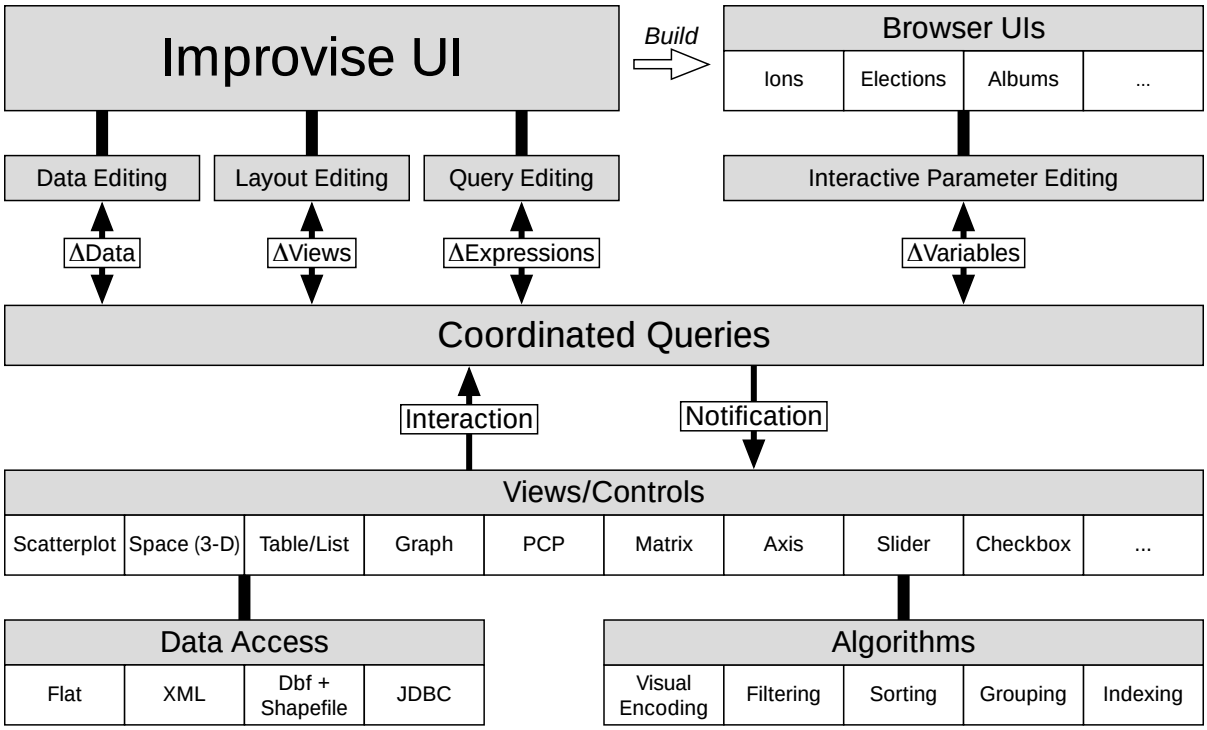



Figure 38: The Improvise software architecture.

Building is fully live, so that all design changes take effect immediately without the need for a separate compilation stage. The live, amodal interface design supports rapid switching between building and browsing.

The architecture is modular in terms of:

- *Views.* Data display techniques include scatter plots, time series, 3-D scene rendering, tables, grids, etc.

- *Glyphs.* Individual data items are visually encoded in views using the position, size, shape, color, and other perceptual parameters of view-specific glyphs that include simple geometric shapes, text labels, arrows, icons, etc.

- *Data Access.* Data sources include local files, compressed archives, web resources, etc. Data formats include delimited text, XML, DBF, Shapefiles [1], etc.

- *Queries.* Data processing operations include projection, filtering, ordering, aggregation, categorization, indexing, etc.

- *Object Types.* The values of variables, properties, and record fields include numbers, strings, booleans, ranges, dates, data sets, lexical query statements, etc.

- *Operators.* Calculations on objects include object construction, type casting, member access, arithmetic, and statistics, as well as glyph construction and query operations on data sets.

In particular, the process of adding a new object type module to the Improvise library is a matter of specifying a default value, a user interface component for editing values, and a suitable variety of function, aggregate, and constant operators for the type.

### 5.5.1 Types of Views and Other Controls

The implementation provides the following view types:

- *Plane views* display scatter plots, time series, histograms, etc. Navigation involves panning and zooming of two ranges. Plane views can also be used to draw maps and cartograms. Although they currently render map layers without geographic projection, they could be extended to do so using geocomputation functionality [89].

- *Space views* display 3-D scatter plots, OpenGL scenes, maps with altitude, etc. Navigation involves translating, scaling, and rotating a camera relative to the origin.

- *Lists and tables* display data attributes in one or more columns. Cells contain formatted text and/or icons. Navigation involves scrolling using a scrollbar, mouse wheel, or arrow keys.

- *Graph views* display node-and-edge graphs. Packs visually cluster related nodes. Navigation involves dragging the entire graph or individual elements.

- *Grid views* display a 2-D grid of cells. Navigation involves panning over the grid.

- *Ortho views* are individual sections of a parallel coordinate plot (PCP) [72]. Multiple ortho views can be laid out horizontally to form whole PCPs. Navigation involves vertical panning and zooming of each section's two ranges.

- *Arc diagrams* connect points along a divider line using half-circular arcs above or below the line. Arc diagrams are useful for visualizing patterns in text strings [147], e-mail threads [80], book reviews [27], and other sequences of temporal events.

- *Pie, bar, and strip views* display the corresponding charts, without navigation.

Some of the new view modules being developed in Improvise include: cartographically-accurate maps (in which the cartesian coordinate plane is geographically projected and navigation involves a fixed aspect ratio), zoomable graphs for displaying ontologies as concept maps [50], and a technique called *reruns* that displays overlapping artificial and natural temporal cycles in a wrapping grid view [151].

The implementation also provides common user interface widgets, including checkboxes, labels, text fields, scrollbars, sliders, and scatter plot axes. Instead of displaying data, these controls provide ways to manipulate interactive parameters of different types during browsing. The type-specific editors used to edit values in the variables builder dialog (section 5.4.4) can also be laid out in frames inside Improvise visualizations, just like regular controls.

### 5.5.2 Implementing Views

Implementation of views in Improvise revolves around management of properties. Over the lifetime of a view, six situations arise that involve dynamic changes to its set of properties:

- Initialization. When views are created (upon opening a visualization or during building activities), they define the name, type, and default value of their properties.
- Modification. Views can create and delete properties at any time. For example, scatter plots can add or remove layers at designer behest by dynamically creating or deleting info, projection, filter, sort, and selection properties.
- Interaction. Views access property values to interpret local input events as navigations and selections, then modify property values correspondingly. For example, a scatter plot interprets horizontal mouse drags as panning gestures that translate the value of its `XRange` property.
- Notification. Views receive notification of events involving their own properties, including creation, deletion, binding, unbinding, and value changes.
- Presentation. Views access property values to update themselves in response to notification as well as to any other internal or external changes.
- Termination. When views are deleted (upon closing a visualization or during building activities), they dispose of all their properties. Disposed properties unbind themselves from variables as needed to avoid dangling dependencies.

Many of the data structures and processing tasks involved in these situations are the same for all controls. Improvise provides a `ControlProxy` class that encapsulates common aspects of property management. Each view contains an instance of `ControlProxy` to perform

property management on its behalf. Centralizing property management radically reduces the amount of code required to implement interaction handling and display updating in views. (For example, the code particular to ortho views (PCPs) involves only 1300 lines plus another 1000 lines for individual glyph classes.) Views implement a simple interface consisting of two methods: `propertyChanged()` is the callback for all notifications; `describe()` returns the view's instance of `ControlProxy`. The latter provides a uniform means for the builder interface to access all aspects of coordination for any given view.

One of the benefits of this design is a "set-and-forget" style of interaction response. Views never update themselves in direct response to their own mouse and keyboard inputs. Instead, views translate raw inputs into navigation and selection parameters which are assigned to properties then forgotten. Changes eventually return to the view through the notification protocol. In a sense, each view unidirectionally coordinates with itself to connect interaction to presentation. As a result, input handling code is cleanly segregated from update code, making view design easier than in monolithic coordination approaches. Implementing new views is often sped up by reusing large code fragments, often unchanged, from existing views. For example, reusing scatter plot code allowed arc diagrams to be implemented in under two hours.

Presentation involves additional data structures and processing algorithms that fall outside the scope of coordination. Views update themselves by asynchronously processing queries pulled from the coordinated query graph through their properties. Notification causes a view to post a redraw event to the main interaction thread event queue. When the redraw occurs, the view issues a new query, displaying whatever past query result it already has. Query completion also posts a redraw event. This "query-on-draw" approach maintains interface responsiveness because multiple views perform a tumbling cascade of quick, cheap tasks rather than slow, expensive ones. Moreover, views that can ignore redraw events (such as those on hidden pages of the visualization) perform negligible work even when being continuously notified of

coordination events, thus improving scalability significantly.

### 5.5.3 Enhancements

The Improvise software architecture has several enhancements designed to improve performance and interactivity during exploration, including:

- *Expression optimization* dynamically rewrites expressions using *scan-constant propagation* to replace subexpressions that always evaluate to the same result over the course of a scan (i.e. those that contain no attribute operators) with a value operator containing the constant result.

- *Function caching* amortizes the cost of performing expensive type-specific operations, such as shapefile generalization (resolution downsampling), image scaling, font measurement, calendar calculations, etc.

- *Data buffering* retains data sets that have been accessed from local disk and remote network sources in memory.

- *Query caching* amortizes the cost of performing queries by caching query results in memory, keyed on the syntactic form of the expressions involved. Caching applies to generation of derived data sets using infos as well as to calculation of non-tabular data structures using expressions (such as by aggregate and index operators).

- *Asynchronous display* pushes data processing and rendering onto throttled secondary threads in order to maintain interactivity in the user interface.

- *Tile caching* amortizes the cost of rendering glyphs in some views (including plane views, grid views, ortho views, and arc diagrams) by breaking the 2-D cartesian plane

into fixed-size image tiles. Tiles are keyed as in query caching, making common case interactions (such as panning) very efficient while maintaining correctness in the face of unusual coordinations (such as semantic zoom).

The last two enhancements are described in more detail in [152].

### 5.5.4 Packaging and Deployment

Improvise runs as a complete, self-contained, end-user, multi-document application. The coordination architecture, module libraries, and user interface are implemented entirely in Java, consisting of over 3600 classes and 300,000 lines of code. Improvise uses several external libraries, including:

- Java Swing (`http://java.sun.com/products/jfc/tsc/index.html`) for user interface components.
- Java Database Connectivity (JDBC, `http://java.sun.com/products/jdbc/`) for access to data in traditional databases.
- Apache Xerces (`http://xerces.apache.org`) for serializing visualization documents as XML files.
- Apache Batik (`http://xmlgraphics.apache.org/batik/`) for capturing SVG screenshots.
- Java Bindings for OpenGL (JOGL, `https://jogl.dev.java.net/`) for 3-D rendering.

Improvise is packaged as two Java jar files: the main application (*improvise.jar*) and its supporting libraries (*oblivion.jar*). These files are deployed in several ways:

- As a cross-platform Webstart/JNLP application that can be executed in a web browser or downloaded for launching from the desktop.
- As a UNIX shell script for execution from a command line.
- As a double-clickable Mac OS X application bundle.

Improvise is free software, distributed under the GNU General Public License (GPL).

## 5.6 Issues and Tradeoffs

### 5.6.1 View Compatibility

Most visualization systems include a small collection of views that are incompatible with the coordination models of other systems. One way to address this problem is to wrap view implementations with a small amount of code to convert coordinations into those of another system. Another way is to provide a single, common, high-level coordination mechanism for translation between any two systems. Although Snap-Together Visualization [105] provides just such a mechanism, it has not been widely adopted.

A number of visualization systems and toolkits (such as GeoVISTA *Studio* [137]) are designed around views built using JavaBeans [61]. This approach provides automatic compatibility across systems for the numerous visual techniques that have been implemented over the years. It also allows visualization systems to take advantage of the large number of existing beans created for non-visualization applications. However, beans communicate directly over asymmetric connections using fine-grained messages that describe value changes at the level of raw inputs. As a result, beans are generally unsuitable for interactive construction of highly-coordinated multiple view visualizations by non-programmers, especially for exploratory visualization.

By comparison, Improvise views communicate indirectly over symmetric connections using coarse-grained messages that describe navigation and selection changes at the level of input gestures, making them suitable for the particular needs of multiple view visualizations in general and highly-coordinated visualizations in particular. Although it is generally straightforward to convert existing views and controls—including standard Java widgets such as checkboxes and sliders—to use Live Properties, the ability to automatically convert them would provide a cheap and easy way to expand the Improvise library. Doing this with existing beans would be a matter of implementing a `BeanProxy` class that converts JavaBean events to and from Live Properties events. The real challenge, however, would be converting the data access, query processing, and visual encoding models used in other visualization systems.

### 5.6.2 Data Access

Improvise implements a memory-resident database for all data sources, trading off scalability and long-term performance for ease of implementation and rendering speed. In particular, the database performs full syntactic caching of all data sources and query results, rapidly filling memory in exchange for short-term maintenance of interactivity. Replacing the current data model with one of the many available database libraries (such as BerkeleyDB for Java [132]) would impose moderate interactive overhead in exchange for scalable, persistent caching of query results. It would also allow transparent importation and management of external data sets with dynamic consistency checking.

### 5.6.3 Query Processing

The order of query operations strongly affects processing speed and memory requirements. For example, views can apply filtering to data both before and after projection. Filtering first reduces the number of records that need to be projected, possibly saving substantial time by skipping unneeded generation of glyphs and other complex objects. Projecting first reduces query processing time in common cases like overview+detail in which the filter changes frequently but the projection stays the same. The speed and memory requirements of query operations thus depend in a hard to predict way on how combinations of expressions depend on patterns of interaction during open-ended exploration tasks. This dependency is even harder to predict when multiple views issue queries at different times and frequencies. Adaptive multiple query processing algorithms [63] may be one way to address some of these issues. At present, Improvise views independently filter, sort, then project data for rendering. With the help of query caching, this order of operations seems to perform reasonably well for the number of views and amount of coordination used in the Improvise visualizations created to date.

### 5.6.4 Visual Concurrency

Asynchronous query processing can lead to "visual concurrency" failures whenever expressions refer to variables that are changing interactively. In particular, the visual encoding of data records may depend on the time each record was processed relative to variable change events. Although there may be obscure cases in which this behavior is acceptable or even desirable, it is undesirable for the standard gallery of views and is likely undesirable in general.

To assure visual consistency, all expressions involved in a given query are rewritten by replacing variable operators with value operators using current variable values. Rewriting occurs at the same time as expression optimization, prior to issuing the query for asynchronous

processing and caching. Because query issuance occurs on the main interaction thread, replacement is effectively atomic, thereby avoiding coarser-grained visual concurrency failures in which individual expressions produce consistent results but entire queries do not. For instance, a scatter plot renders data in multiple layers defined by various info, projection, filter, and sort expressions. To build its query, the scatter plot atomically replaces all variables operators in these expressions with corresponding value operators, avoiding visual inconsistency that could result by using a different set of variable values for filtering than for projection.

This approach reduces but does not eliminate lapses in visual concurrency. In response to the stream of incoming coordination notifications, different views may issue queries involving different interactive states. Moreover, asynchronous processing means that the consistency of views with respect to the current interactive state of the visualization depends strongly on processing time as a function of query complexity. Nevertheless, individual views will appear self-consistent even if the visualization as a whole appears somewhat inconsistent because some views are more up-to-date with recent user interactions than others.

### 5.6.5 Screen Space

All visualization designers must contend with the problem of limited screen real estate. Larger and tiled screens [7] are two ways to use hardware to address the problem. Coordination flexibility and increasing hardware speed encourage more views, but the number of views is fundamentally limited by available space. Making all views smaller is a short-term solution limited by the ability to represent data in each view using fewer pixels.

The goal of rapid exploration encourages amodal designs in which building and browsing are integrated in time and space. During building, however, building dialogs compete with visualization frames for space. It is often necessary to move dialogs repeatedly to see the

effects of editing on visualizations. During browsing, moreover, users must know how views are coordinated in order to interact with them effectively. Like web browsers, Improvise uses a tabbed folder metaphor to provide more working space. Because coordination can occur across pages, however, designers and users may lose track of dependencies between views, and thus be surprised by unexpected changes when returning to a previous page.

## 5.7 Extensions

### 5.7.1 Standalone Browsing

One of the primary benefits of the integrated interface in Improvise is the ability of designers and skilled users to shift quickly and transparently between design, construction, testing, and exploration. Most users, however, are expected to lack the necessary design skills and experience to take advantage of the builder interface. Deployment of a standalone browser would involve providing a version of Improvise in which builder functionality is removed. Such a browser would provide a simpler user interface and have a much smaller deployment footprint (by stripping out unneeded code), but could open and save Improvise visualization documents in the existing XML format. Individual visualizations might also be deployed as self-contained browsers designed for specific analytical purposes.

### 5.7.2 Library Namespace and Versioning

Improvise manages library modules using a flat namespace mechanism. Although all modules are dynamically loaded, loading occurs once during program initialization using a hard-coded list of modules stored in the local file system (that is, on the Java class path). Developing large libraries of modules would benefit from a hierarchical namespace mechanism that allows

identification of local and remote modules from a variety of sources. Startup and runtime performance would improve by loading (and unloading) modules as needed when opening or building visualizations.

Aging of visualization documents in the face of ongoing module development is a related problem. The kinds of module improvements that occur frequently involve:

- changing the properties of controls, and
- changing or reordering the arguments of function operators (such as to create more flexible glyph constructors that expand possibilities for visual encoding).

Different versions of evolving modules could be differentiated using version numbers as part of the namespace mechanism. The development process would require keeping old versions for backward compatibility. Visualization document files would refer to specific versions of controls and operators. Absence of individual modules or entire libraries would result in graceful failure when opening or building visualizations.

## 5.8 Summary

Improvise consists of a unified builder and browser user interface on top of a modular library of visualization components. In Improvise, visualization designers rapidly create, lay out, and coordinate multiple views of data accessed from one or more flexibly specified sources. Visualization users explore data by browsing the resulting visualization interfaces. Because browsing occurs directly within the builder interface, users with sufficient motivation and skill can act as their own designers, making possible open-ended exploration and analysis.

# Chapter 6

# Coordination Patterns

## 6.1 Overview

Information visualization systems have matured into interactive development environments that enable users to build visualizations with multiple coordinated views rapidly. A variety of useful coordination techniques have been discovered by working in these environments. North and Shneiderman [104] have organized these techniques into a taxonomy of strategies for coordinating multiple views in terms of navigation and selection. Most visualization systems implement only a subset of these techniques, the goal being to provide users with a simple, elegant, easy-to-use means to visualize their data. Users coordinate views using a small set of predefined primitives that follow well-known *coordination patterns*.

Coordination patterns are inspired by design patterns [52, 37]. Just as design patterns are recipes for building software, coordination patterns are recipes for building visualizations using multiple coordinated views. Like other visualization systems, Improvise enables users to specify combinations of data, views, and coordinations interactively [107]. Unlike other systems, Improvise enables users to invent new combinations and refine existing ones. This chapter describes 28 coordination patterns and outlines how they can be instantiated in Improvise using examples from five typical visualizations, shown in figures 39–44. (The reader may safely choose to overlook the detailed points in these figures for now, as they will be discussed throughout the chapter in relation to specific patterns.)

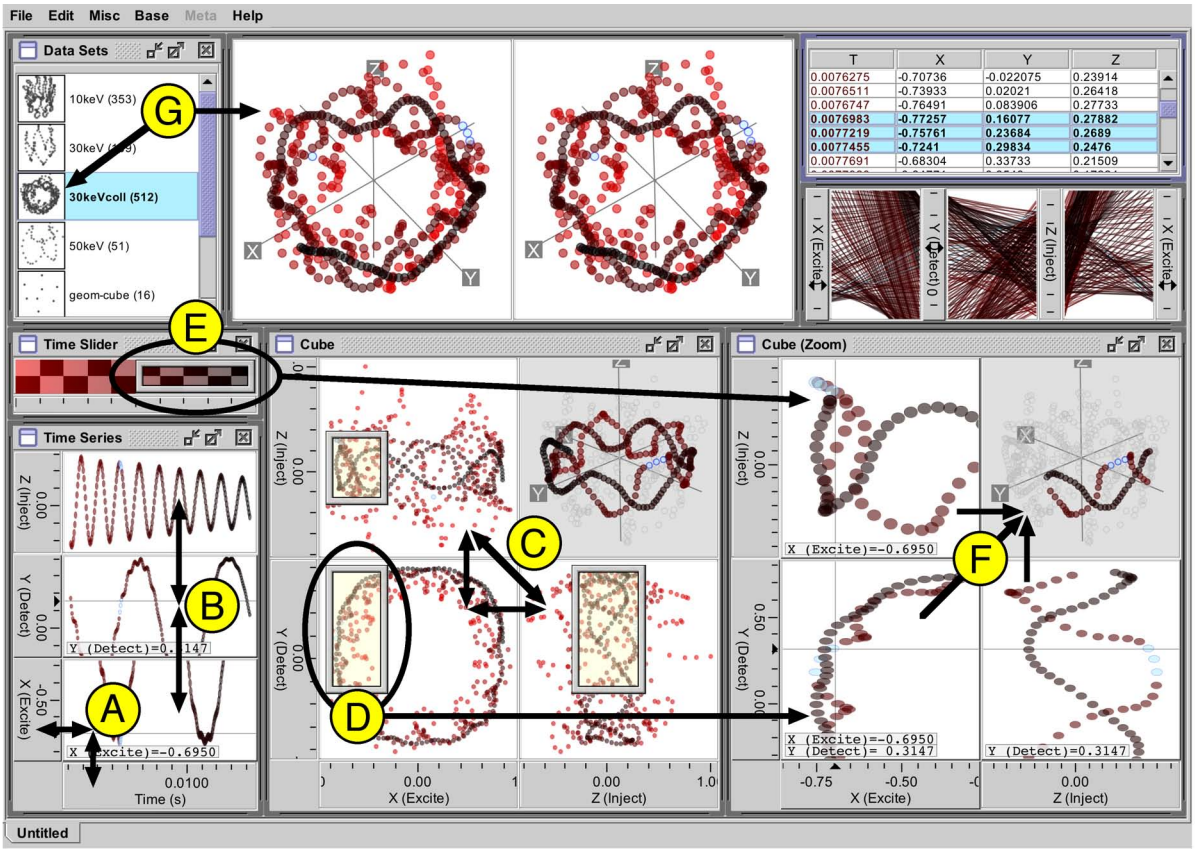



Figure 39: Visualization of simulated ion trajectories (see appendix A.2). (A) Axis controls label a scatter plot and provide a way to change X and time independently. (B) Horizontal synchronized scrolling coordinates three time series scatter plots showing the X, Y, and Z positions of ions over time. (C) A scatter plot matrix shows the trajectory as seen from three orthogonal sides of the ion trap. (D) An overview uses a portal (circled) to select the extent of a detail view. (E) A perceptual slider enables users to select a visible range of time using a translucent color gradient instead of numeric values. (F) Portions of the trajectory outside the cubic detail region are filtered out in the detail scatter plot matrix but are drawn in gray in a 3-D view. (G) The names of the available trajectory data sets are accompanied by nested views that are rotationally coupled with a stereoscopic pair of 3-D views.

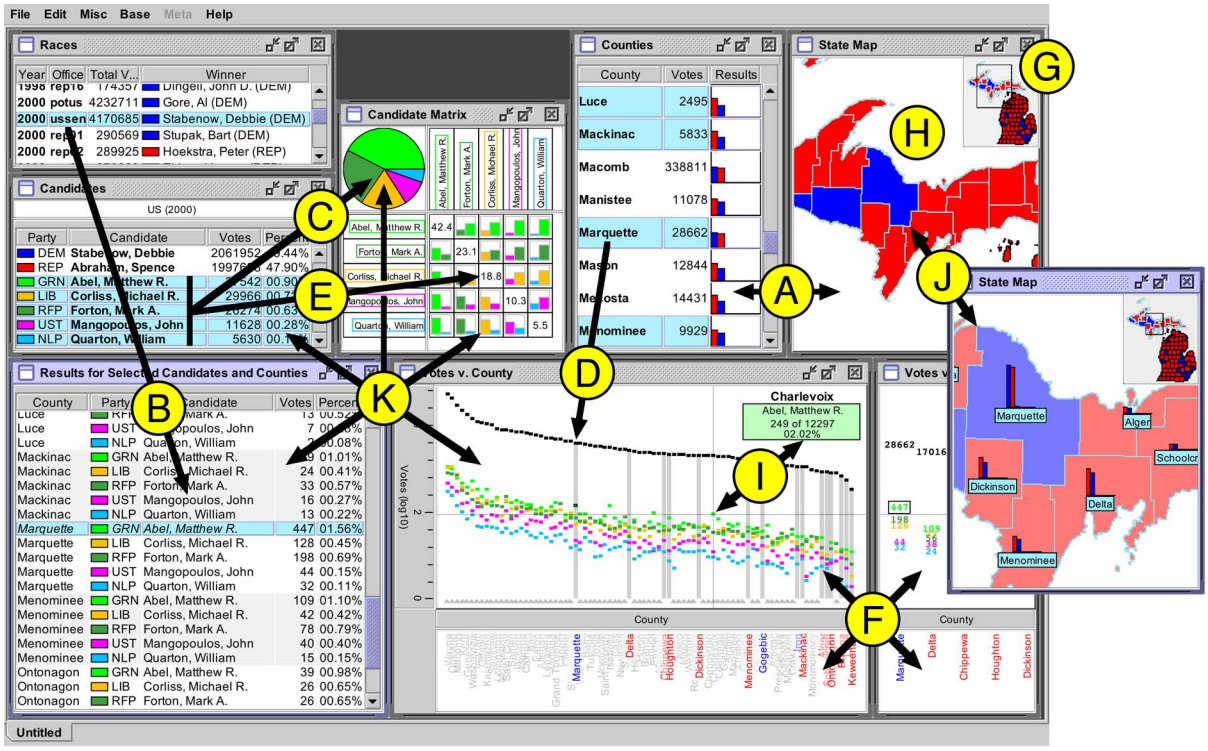



Figure 40: Visualization of election results in Michigan from 1998 to 2004 (see appendix A.3). (A) Shared selection of counties between a table view and a map. (B) Selecting a race causes the election results for that race to be loaded (from a file) and shown throughout the visualization. (C) A pie chart uses a filter to compare results for selected candidates only. (D) A scatter plot highlights selected counties with gray bars. (E) The diagonal of a grid view shows vote percentages considering only selected candidates. (F) Four scatter plots break down election results and winning candidate party color in decreasing order of total county votes, for all counties and for selected counties only. (G) An inset view summarizes county winners for the entire state. (H) A four-layer scatter plot colors counties by winning candidate party. (I) An additional layer displays inline detail for results at the current mouse location in a scatter plot. (J) Semantic zoom labels counties with nested bar plots at sufficient zoom. (K) Five views show the same election results in different forms.

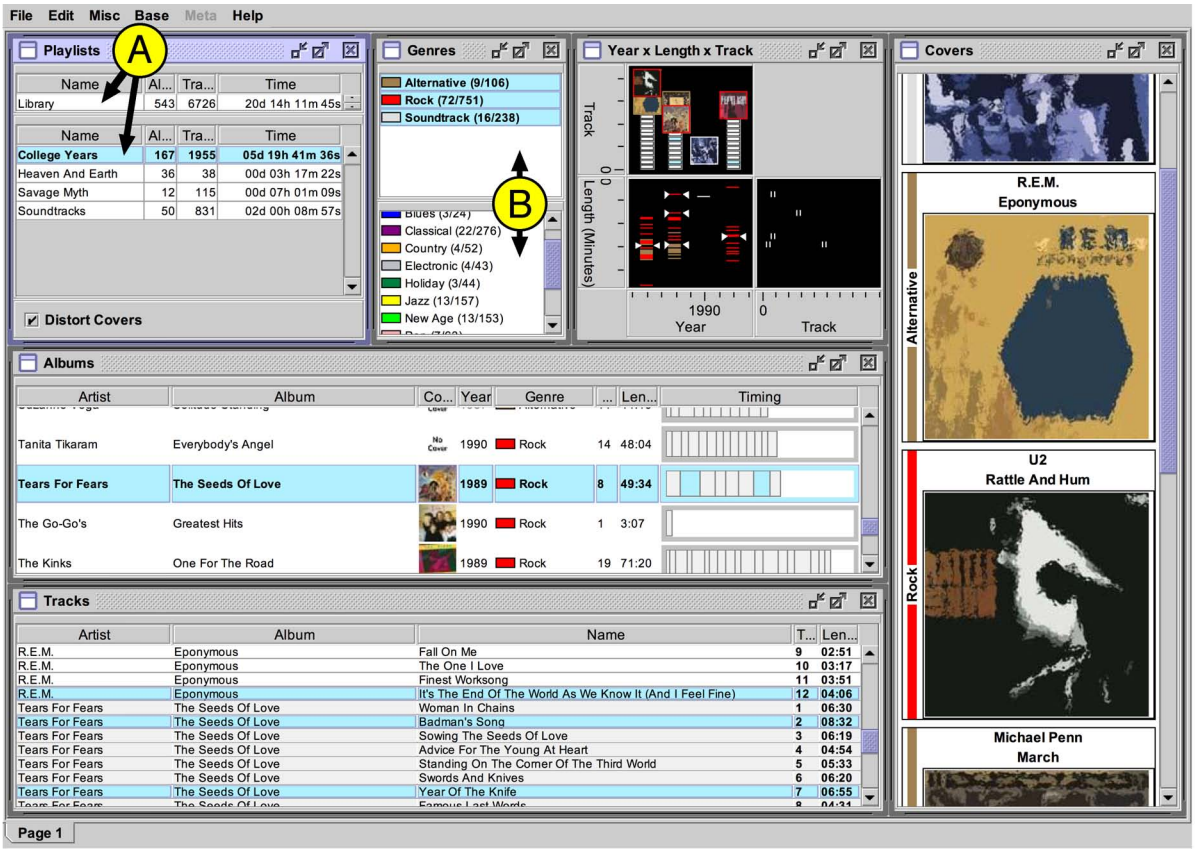



Figure 41: Visualization of iTunes playlists (see appendix A.7). (A) Split selection of a single playlist into two list views that isolate the master (library) playlist from the other playlists. (B) Magic selection shifts genres from the unselected list (bottom) to the selected list (top) and back again.

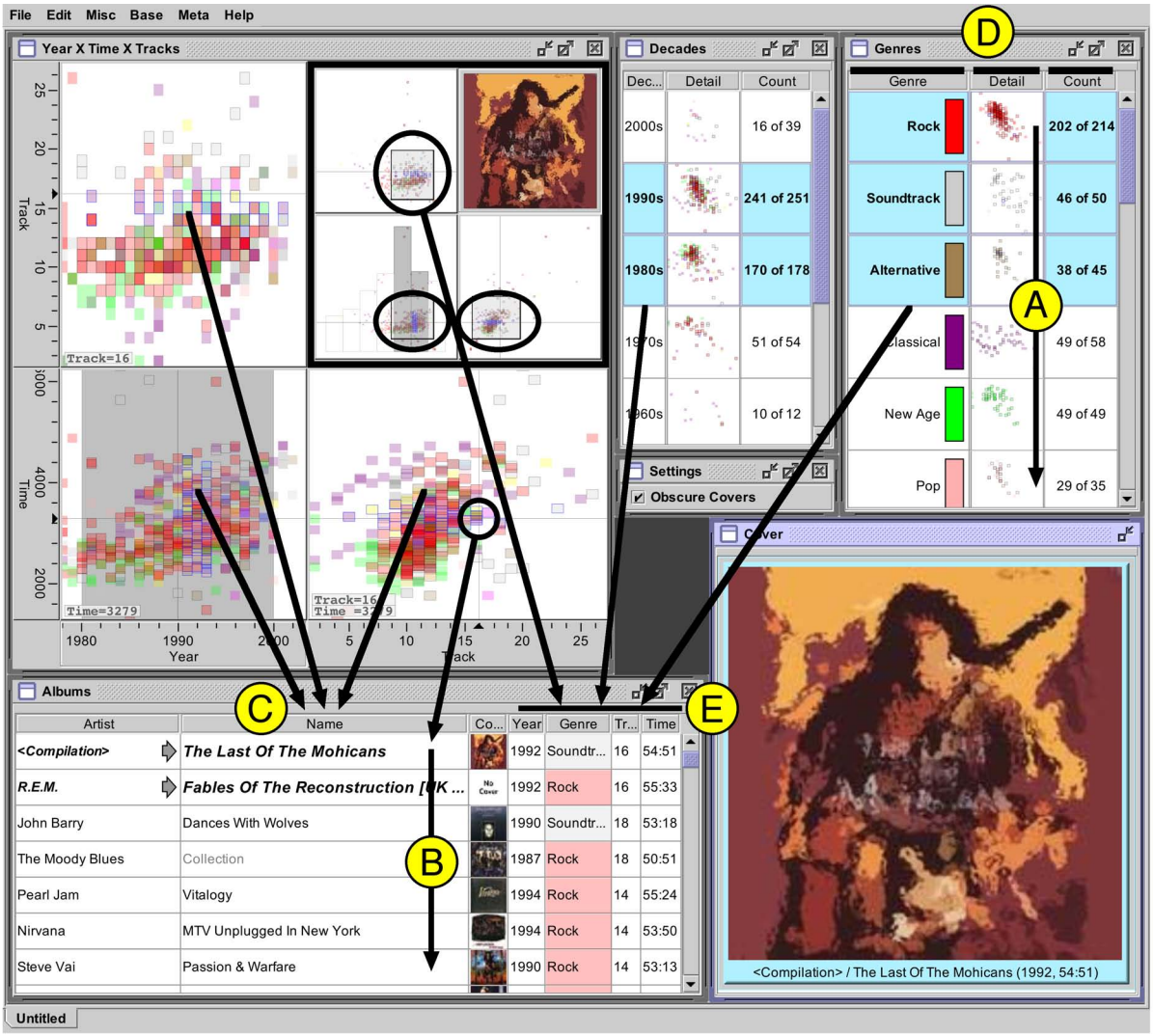



Figure 42: Visualization of music albums with cover art (see appendix A.7). (A) A table of musical genres shows top picks by moving selected rows to the top. (B) The albums table is sorted on increasing distance from the current mouse point in the `(track, time, year)` scatter plot matrix. (C) Compound brushing of albums. Names are drawn in black if selected in at least one scatter plot, and in larger italics if selected in all three scatter plots. (D) Small multiples summarize albums of each genre across three columns of a table view. (E) Navigation and selection in multiple controls (three portals and two table views) filter the album table on decade, genre, time range and track range.


Figure 43: Geovisualization of county-level census data (see appendix A.6). Three nested lenses draw county names (A), roads color-coded on type (B), and urban areas (C) on top of the map. (D) Overlapping the three lenses implicitly reveals multiple layers of detail. (E) An additional layer of the map explicitly draws road names inside the spatial intersection of the lenses.

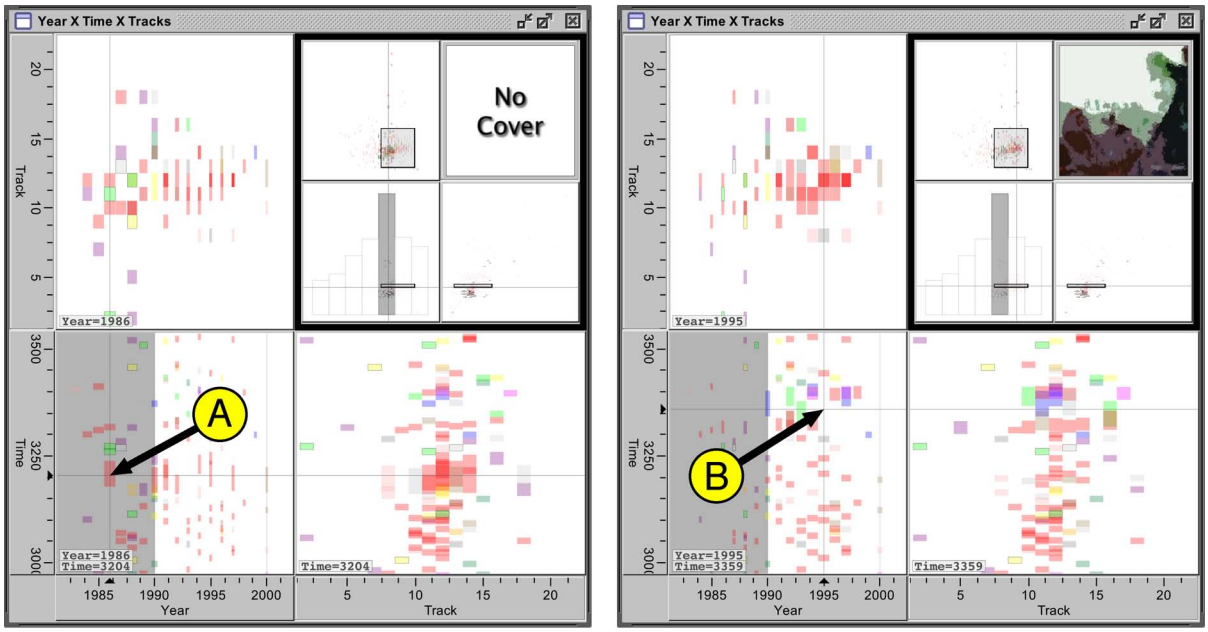



Figure 44: Item distortion in the visualization in figure 42. The size of each rectangle glyph is a function of its distance from the current mouse location. Moving the mouse pointer from (A) to (B) shifts the center of distortion. The other two scatter plots distort in one dimension.

The purpose of this chapter is to argue that a flexible and expressive approach to coordination expands the range of exploratory visualization. It does so by

- providing examples of expression syntax,
- applying aggregation, indexing, and grouping to visual queries,
- identifying useful patterns in a space of possible coordinations,
- providing context for each pattern in terms of prior research and applications,
- describing how to reproduce common coordinations available in existing visualization systems,
- outlining how to build custom coordinations as needed during data exploration, and
- suggesting useful pattern variations.

## 6.2 Navigation Patterns

Navigation is the interactive manipulation of points, ranges, and regions of space in views. In Improvise, navigation occurs by mapping mouse and keyboard inputs into points and ranges in a multidimensional data space displayed by views and other controls. Decoupling data from the space in which it is displayed allows for flexible display of multiple data sets in high-dimensional spaces across many views. Common coordination patterns involving navigation include:

- *Scatter Plot + Axes.* Range sliders control orthogonal cartesian dimensions.
- *Synchronized Scrolling.* Views show the same region of space.
- *Scatter Plot Matrix.* Two-dimensional views show an N-dimensional space.
- *Overview+Detail.* One view shows all data; the other, a portion.
- *Perceptual Slider.* Users navigate in non-spatial dimensions such as color.
- *Navigation-Dependent Encoding.* Spatial relationships affect item appearance.

### 6.2.1 Scatter Plot + Axes

Sliders and other controls are often useful for manipulating individual parameters of a visualization. In Dynamic Queries [2], non-spatial data attributes can be manipulated using range sliders. LinkWinds [73] provides controls that can coordinate with views for dynamic filtering.

Improvise users browse visualizations by interacting with views and non-data controls such as checkboxes, text fields, and sliders. Axis controls are independent of scatter plots, but perform the usual roles of marking, labeling, and handling interaction in one dimension. Figure 45 shows how horizontal and vertical axes can be coordinated with a scatter plot.

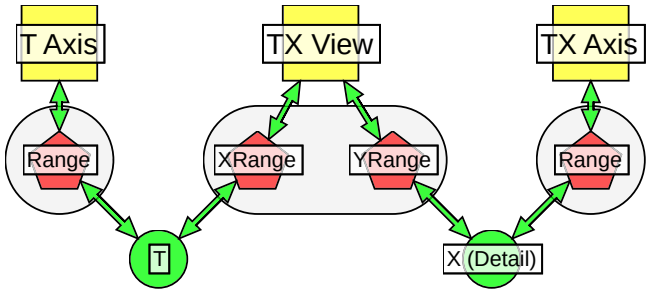



Figure 45: Coordination graph for a scatter plot with axis controls (see figure 39A). Panning or zooming in the `T` (or `X`) axis changes the value of the `T` (or `X`) range variable, which causes the plot to translate or stretch horizontally (or vertically). Manipulating the plot changes both variables, causing both axes to update appropriately.

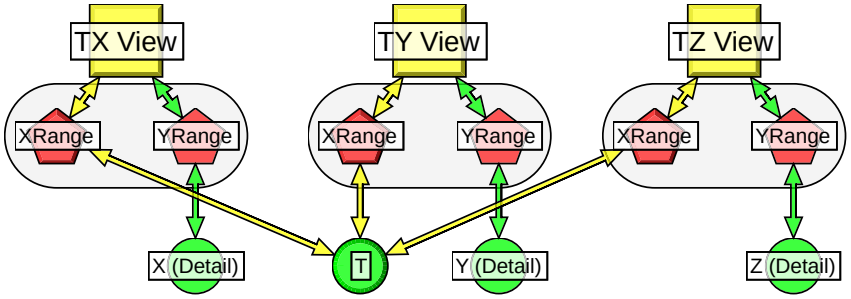



Figure 46: Coordination graph for three scatter plots with synchronized horizontal scrolling but independent vertical scrolling (see figure 39B). All three plots update in unison whenever the value of `T` changes.

### 6.2.2 Synchronized Scrolling

Views can also be coordinated with each other. Synchronized scrolling is a common form of coordination in which two views are constrained to show the same data items or the same region of a coordinate space. For instance, scatter plots in DEVise [90] can be coordinated through *visual links* to show the same range of `X` and/or `Y`. In Snap-Together Visualization [105], synchronous scrolling between lists of items is achieved by coordinating their *scroll actions*.

Plots can be coordinated with each other in the same way that they coordinate with axis controls: through their range properties. Figure 46 shows three plots in which scrolling is synchronized horizontally. This is done by binding the same range variable to the `T` range property of all three views. The flexibility of property-variable binding makes it simple to

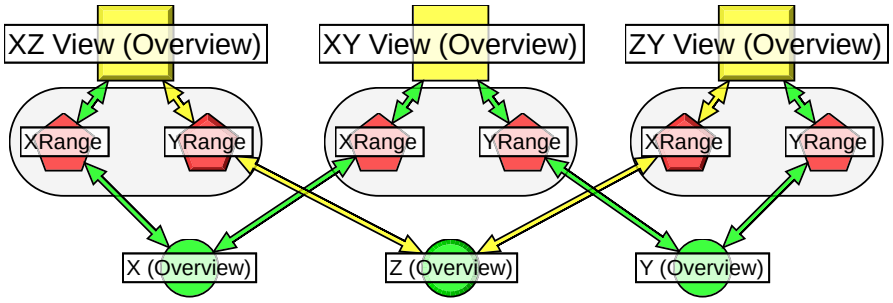



Figure 47: Coordination graph for a 3-D scatter plot matrix (see figure 39C). The shared `Z` variable synchronizes vertical navigation in the `XZ` scatter plot with horizontal navigation in the `ZY` scatter plot.

construct numerous variations of synchronized scrolling, including two-dimensional (sharing `X` and `Y` ranges), horizontal (sharing only the `X` range), vertical (sharing only the `Y`), and crossed (one view shows `XY`, the other `YX`).

### 6.2.3 Scatter Plot Matrix

Scatter plot matrices [9] show an N-dimensional space as a stairstep arrangement of 2-D scatter plots. Synchronized scrolling in this case is complicated by the need to invert the coordinates of some plots in order to produce the expected navigation behavior. Figure 47 shows how inverting the coordinates of a plot is a simple matter of swapping the range variables bound to its properties. Building scatter plot matrices is straightforward but tedious; Improvise provides shortcuts (section 5.4.5) for building scatter plot matrices for $N >= 2$.

### 6.2.4 Overview + Detail

Coordination using the overview+detail [129] technique differs from synchronized scrolling in that the entire area shown in a detail view is synchronized with a subarea of an overview. *Cursors* in DEVise are an example of this technique in which a selection box in a scatter plot has the same `X` and `Y` ranges as some other scatter plot.

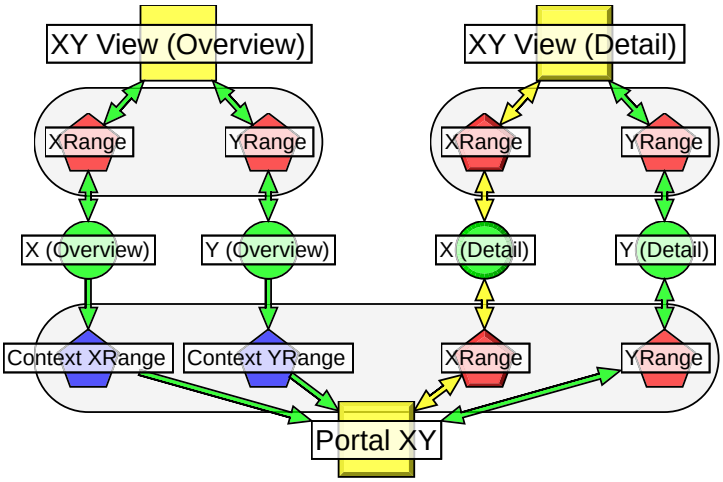



Figure 48: Coordination graph for overview+detail (see figure 39D). The portal covers the region in the overview (its context) that corresponds to the full region visible in the detail view.

In Improvise, *portals* (not to be confused with portals in DataSplash [110]) are draggable controls for selecting a rectangular region. Figure 48 shows how the `X` and `Y` ranges of a detail plot are coordinated with the ranges of a portal inside an overview plot. This construction can be chained to create multiple levels of detail (as in [117]). Omitting the two `X` (or two `Y`) range variables produces vertical (or horizontal) versions of overview+detail.

### 6.2.5 Perceptual Slider

Another use of one-dimensional portals is in *perceptual sliders*, which allow users to select data by thinking visually while acting spatially. Figure 49 shows how a plot is coordinated with a portal in a gradient view to create a perceptual slider based on color. The projection expression used by the plot visually encodes points along an ion trajectory by mapping (normalized) time into the same color gradient shown in the gradient view. The filter expression used by the plot elides points that would fall outside the range of color selected by the portal. Although the user perceives the position of the portal as a selection on color, the selection is actually on a range of time values. (Perceptual sliders are similar to visualization sliders [41], but present a set of

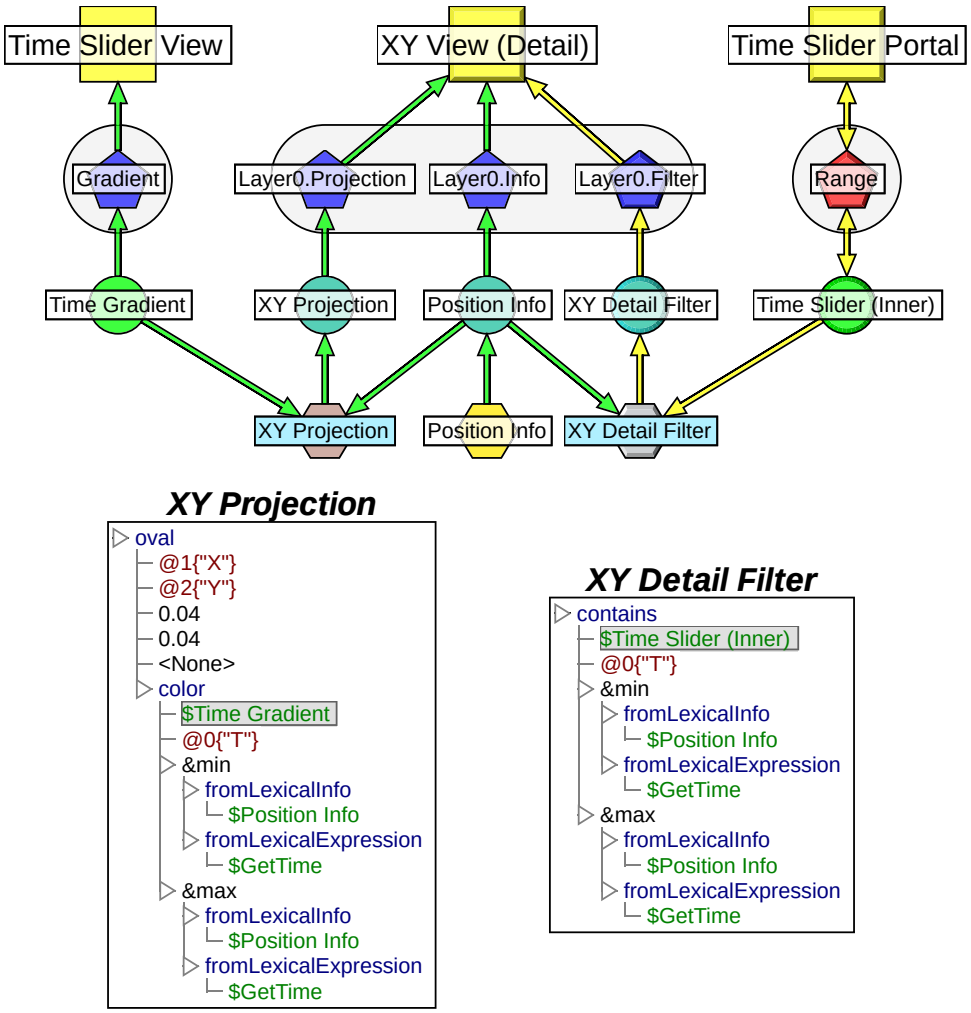



Figure 49: Coordination between a scatter plot and a gradient slider (see figure 39E). The scatter plot draws ovals colored by mapping time into a color gradient, relative to minimum and maximum values, but only for relative times in the range selected by the slider portal.

possible perceptual values instead of a distribution of values that actually occur in the data.)

### 6.2.6 Navigation-Dependent Encoding

Perceptual sliders are a special case of *navigation-dependent encoding*, in which the visual encoding of items in views is a function of spatial parameters of other views. DEVise uses this technique in the form of *record links* that cause a "destination" scatter plot to render only those records that are visible in a "source" scatter plot.

Figure 50 shows how a 3-D view can be coordinated with a scatter plot so as to visually differentiate between trajectory points that are and are not inside the rectangular region visible in the scatter plot. Visual differentiation can occur by filtering or projection; in this example the visualization uses projection. In the case of navigation-dependent filtering, the 3-D view elides trajectory points outside the cubic space defined by three range variables, two of which are bound to the `XY` scatter plot in the detail scatter plot matrix. In the case of navigation-dependent projection, the 3-D view draws the same points in gray.

Navigation-dependent filtering also occurs between the three scatter plots. The bounding rectangle of each plot naturally filters the trajectory in two dimensions; in the third dimension, an expression defined in terms of the corresponding third range variable elides trajectory points. The result is true three-dimensional overview+detail.

The expressiveness of Coordinated Queries makes it possible to impose complex geometric semantics during navigation-dependent encoding. For example, the filter and projection expressions could be modified to check for total or partial containment of the actual oval glyphs in the three scatter plots rather than containment of the infinitely small raw data points they represent. The subexpressions used to test containment would be similar to those used to define the bounds of the oval glyphs in the first place.

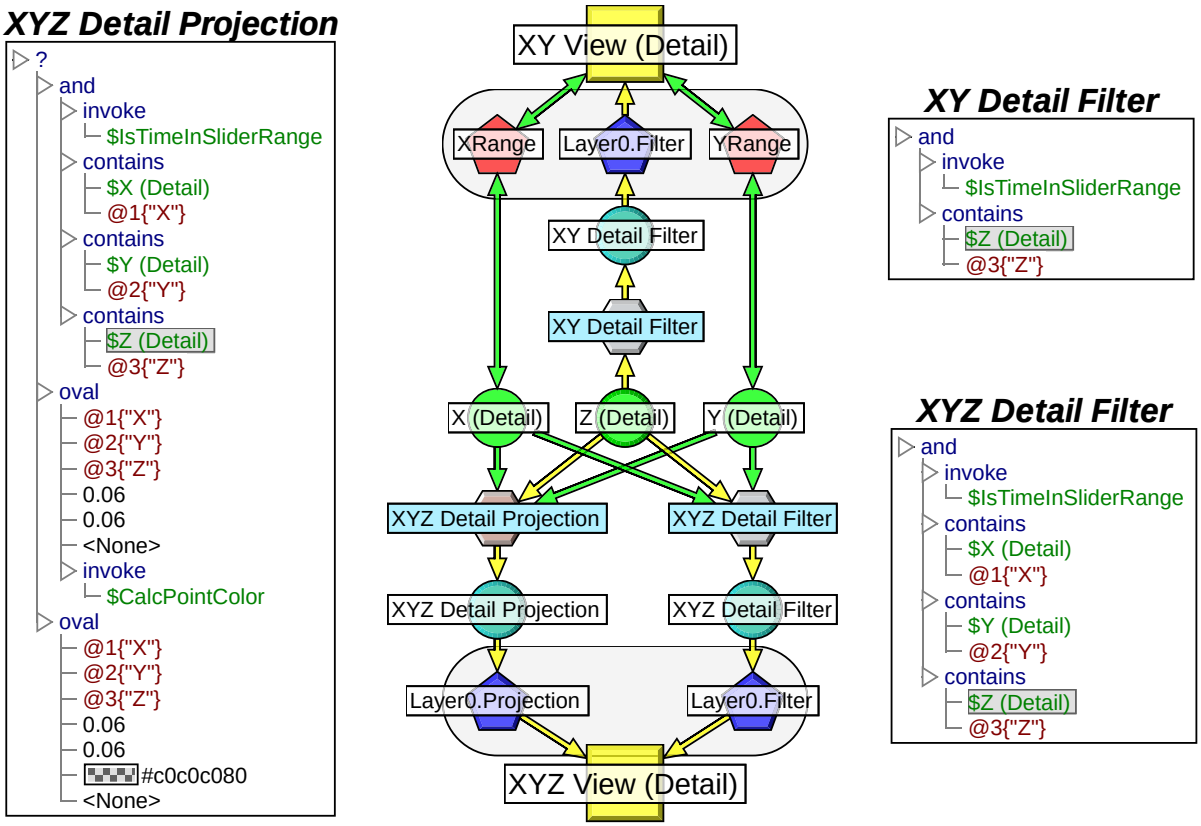



Figure 50: Coordinated query graph for navigation-dependent encoding (see figure 39F). In the 3-D detail view, trajectory points that fall within the cubic region visible in the three detail scatter plots are filtered out or differently projected from those outside the region. The three detail scatter plots similarly filter out such points; the bounds of each plot serves as a natural filter in two of the three dimensions.

## 6.3 Selection Patterns

Selection consists of user interactions that distinguish arbitrary data items from others. Views can visually differentiate selected items from unselected items in many ways, the most common being highlighting with color. In Improvise, a selection is an independent bitstring that indicates selected records by the integer identifiers assigned to them when data is accessed during visualization. Decoupling selections from data in this way (rather than representing them as additional data columns as in XmdvTool [145]) separates coordination of views on data from coordination of views on selections. This approach makes it possible to coordinate multiple views using multiple independent selections of the same data set in a single visualization. (However, it also makes it possible for designers to accidentally introduce semantic ambiguity by associating the same selection with different data sets). Useful coordination patterns involving selection include:

- *Shared Selection.* Views highlight the same data items.
- *Selection-Dependent Data.* Selected items represent whole data sets.
- *Selection-Dependent Encoding.* Selection affects item visibility and appearance.
- *Selection-Dependent Aggregation.* Aggregates exclude (un)selected data items.
- *Split Selection.* Multiple views split up selectable items into disjoint subsets.
- *Magic Selection.* Data items jump between views when selected.

### 6.3.1 Shared Selection

Shared selection is a form of brushing [9] that allows users to select items in views, and see the corresponding items in other views. In XGobi [21], users can brush items in multiple

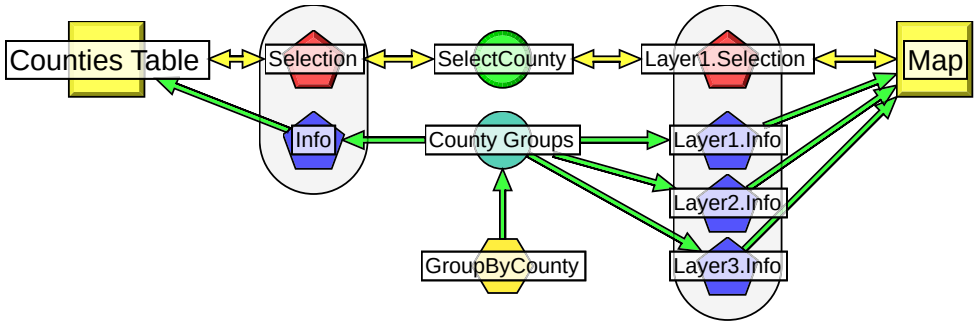



Figure 51: Coordination graph for shared selection between a table view and a scatter plot (see figure 40A). Selection of items in either view causes both views to redraw their shared data.

scatter plots of high-dimensional data. Brushing-and-linking in Snap-Together Visualization uses *select actions* to coordinate selections in two views of the same data.

Figure 51 shows how two Improvise views are coordinated to share a selection over data that describes the 83 counties in Michigan. The scatter plot draws counties as polygons read from shapefiles; the table view draws each county as a row of text with a nested bar plot. Selecting items in either view (by clicking shapes or rows) changes the selection variable, causing both views to redraw with the selected items highlighted in cyan.

### 6.3.2 Selection-Dependent Data

Users often want to select from multiple related data sets (or subsets of one large data set) in a single visualization, such as during analysis of a sequence of experiments. Selecting a data set in one view to show in other views is a form of drill-down. For instance, Snap-Together Visualization supports drill-down by coordinating a *select action* in one view with a *load action* in another view.

Selection-dependent loading of data in Improvise is performed using an expression that is defined in terms of (1) data that lists the names of (or otherwise identifies) loadable data sets, and (2) a selection on that data. In figure 52, the election results for each office are

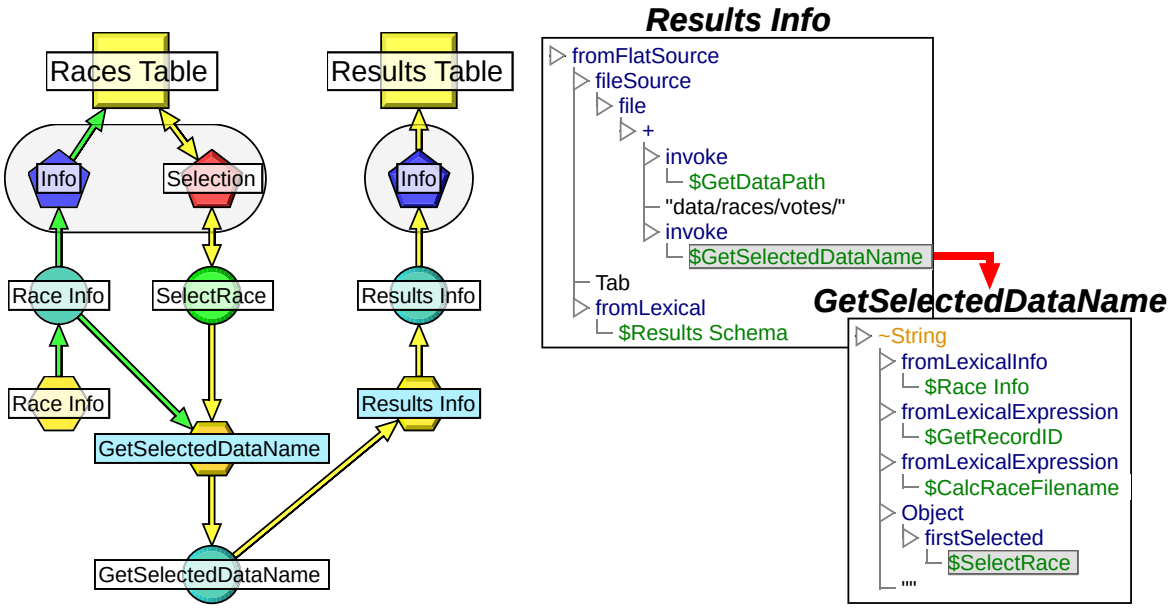



Figure 52: Coordinated query graph for selection-dependent loading of data (see figure 40B). An index on the races data set maps the record identifier of the first selected race into a filename. The `Results` view displays an info that accesses the corresponding file of voting results.

stored in separate files. The expression constructs the name of a file to load using the name of the selected office. Whenever the user selects an office, the visualization loads data from the corresponding file. Using expressions, the user can specify a file, URL, or database as the source of data to visualize.

### 6.3.3 Selection-Dependent Encoding

Selection-dependent filtering is an asymmetric version of shared selection in which the filtered view differentiates between selected and unselected items by not drawing unselected items instead of highlighting selected ones. Whereas selection-dependent filtering determines the visibility of items, selection-dependent projection determines the appearance of items. Most visualization systems can coordinate two views so as to highlight the items in one view that correspond with items selected in the other view. Highlighting is usually a fixed function of the type of view, typically implemented as a special background color. In XGobi, points and

lines in scatter plots can be brushed using glyphs as well as color.

By using expression-based projections to determine the entire visual encoding of items in views, highlighting in Improvise is a user-customizable visual differentiation of selected and unselected items. Highlighting of items can therefore appear as a special background color, reverse video, a special font, or just about any variation on color or other visual attributes the user can dream up. Customizable highlighting can also be used to avoid conflict with normal visual encoding of items. In figure 53, the `Candidate Shares` pie chart shows vote shares for candidates selected in the `Candidates` table view. Although both views display the same data, the filtered view elides unselected candidates using an expression defined in terms of the selection. The result is a kind of multi-item details-on-demand that allows comparison of details for selected subsets of items. Similarly, the `Votes v. County` scatter plot highlights counties based on whether they are selected in the `Counties` table view.

### 6.3.4 Selection-Dependent Aggregation

Many visualization systems enable users to explore large data sets in limited screen space by manipulating aggregates [54]. Visage [121] supports aggregation of data records into categories of raw and derived attributes. In breakdown visualization [36], categorical aggregation of tuples and functional aggregation of attributes allow drill-down in data *polyarchies* (multiple overlapping hierarchies). Improvise supports categorical aggregation through aggregate operators and group queries. Because both of these mechanisms are based on expressions, users can categorize records on arbitrary functions of attributes.

One way to think of selection is as a way for users to impose an arbitrary binary categorization scheme on items in a data set. Aggregates can be calculated over either category; that is, over selected items or unselected items. Aggregates can also be calculated over the

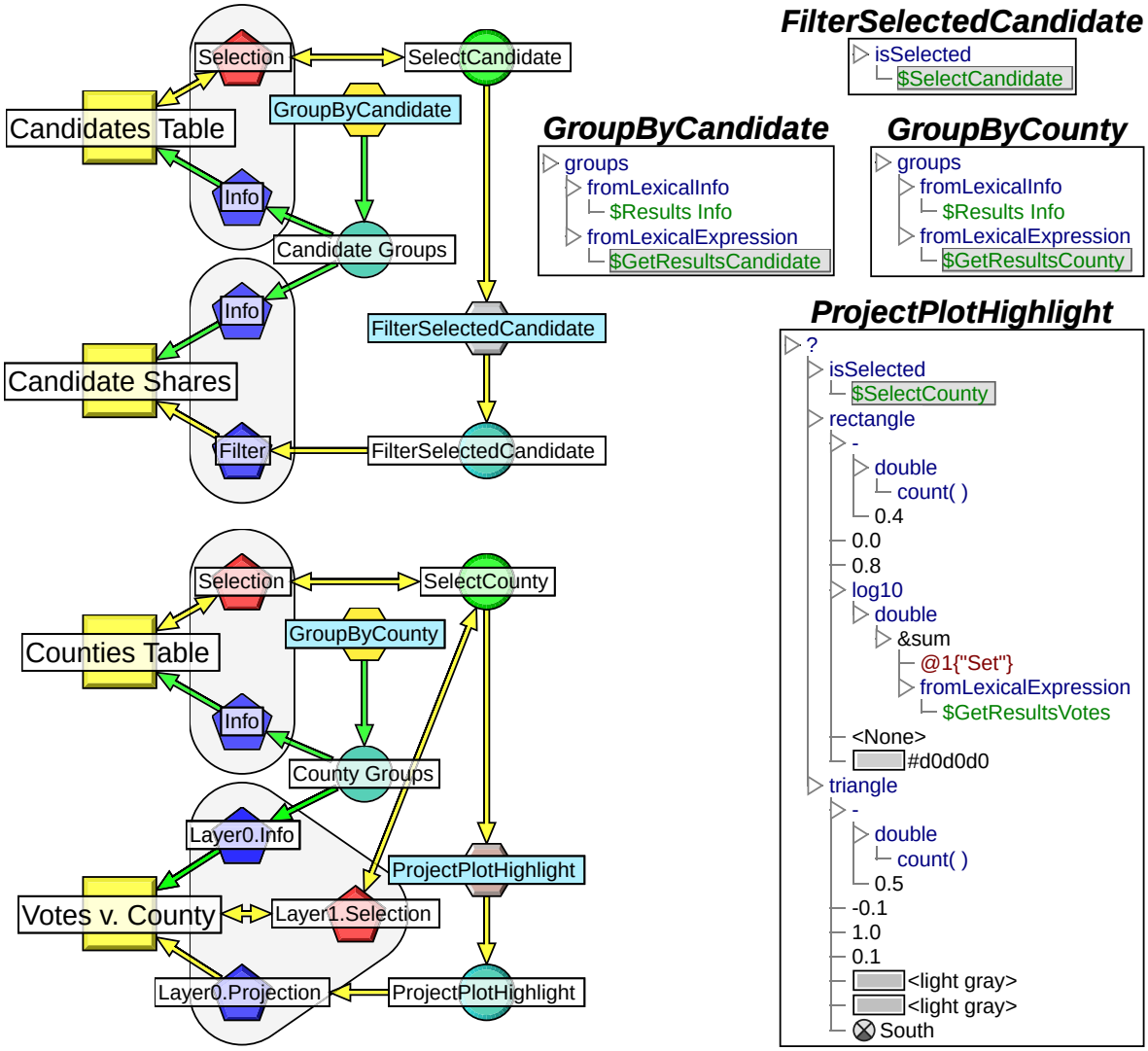



Figure 53: Views can be indirectly coordinated through filters or projections that depend on selection variables (see figure 40C, 40D). The filter expression states that "for each candidate, draw it only if it is selected." The projection expression states that "for each county, draw a rectangle if it is selected, a triangle otherwise." The height of each rectangle is an aggregate of the data set created by grouping the overall election results by the corresponding county.

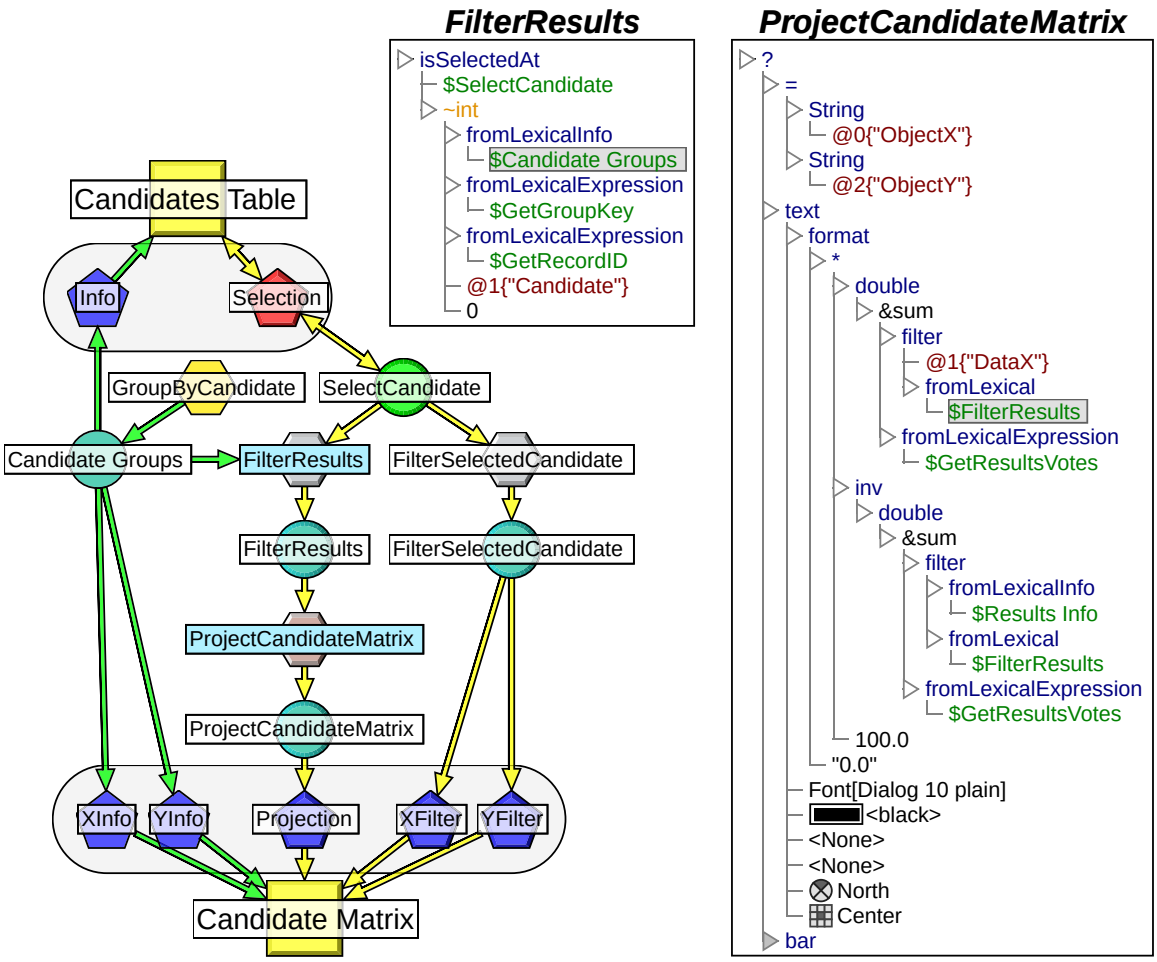



Figure 54: Coordinated query graph for selection-dependent aggregation (see figure 40E). A grid view displays nested bar plots for comparing candidates pairwise. The diagonal of the grid displays the percentage of votes for each selected candidate relative to the total votes for all selected candidates.

subset of items for which an attribute maps into a selected item in another data set. In figure 54, for example, the aggregate sums votes over records for which the candidate attribute maps to a selected candidate. Selection-dependent aggregation is essentially the application of selection-dependent filtering to a data set prior to aggregation. In fact, the filter expression in figure 54 has been simplified; the actual visualization filters the voting results data set on selected counties as well as selected candidates.

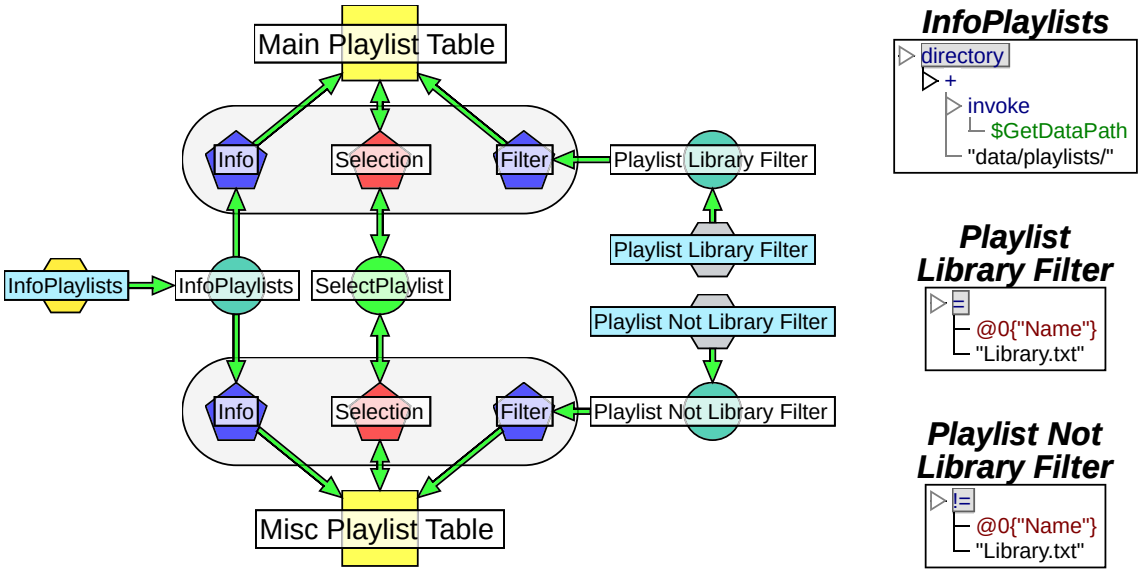



Figure 55: Coordinated query graph for split selection (see figure 41A). Complementary filters split the playlists data set between two table views that share a selection variable. Selecting a playlist in either table causes all other playlists in both views to be deselected.

### 6.3.5 Split Selection

Split selection is an extension of shared selection in which the records in a data set are divided into two or more non-overlapping subsets, each displayed in its own view. Division can be performed using a grouping query or by applying mutually exclusive filters. Selection of items occurs as if they are all contained in a single view. Figure 55 shows selection of music playlists split between two list views, one containing the full music library, the other containing all other playlists. Because both views are set to single selection mode, selection of any item in either list causes all other items in both lists to be deselected.

Unlike split panes in spreadsheets and word processors, splitting in Improvise is not limited to showing different regions of a single display space. The views that display each subset need not have the same visual encoding, or even be of the same type. For example, the second list view could show the distribution of genres in non-library playlists using nested bar plots.

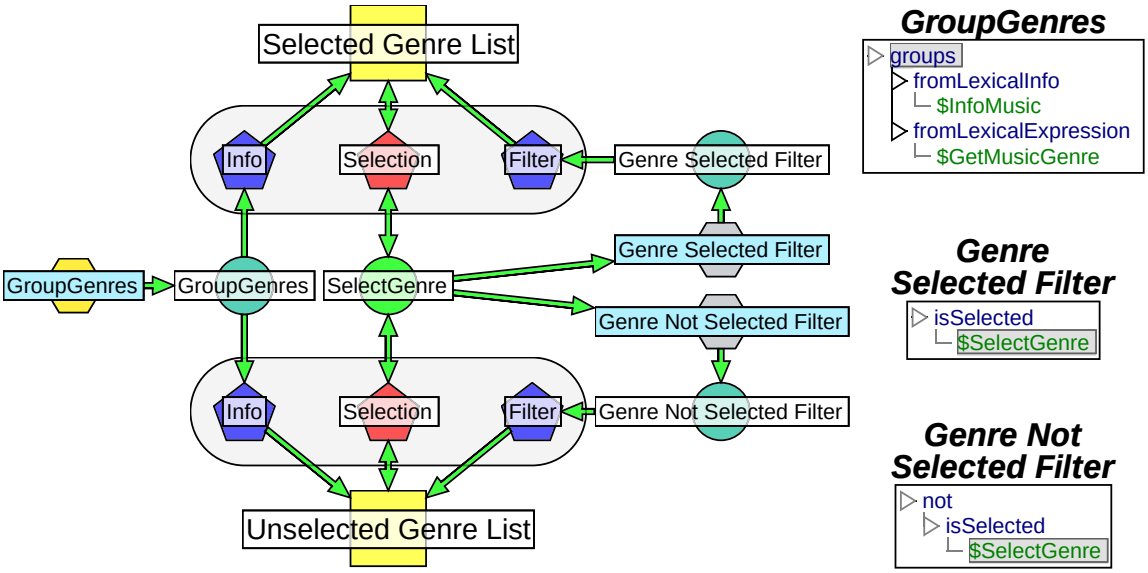



Figure 56: Coordinated query graph for magic selection (see figure 41B). Complementary filters split musical genres into selected and unselected lists. Clicking a genre in either list causes it to "magically" reappear in the other list.

### 6.3.6 Magic Selection

Magic selection is a combination of shared selection and selection-dependent filtering in which the items in a data set are split between two complementary views. One view shows selected items; the other shows unselected items. Clicking an item in either view toggles the corresponding bit in a selection variable, causing the item to "magically" switch views.

The magic selection pattern underscores how important it is to distinguish *selection*—interaction by users to divide items into arbitrary, complementary equivalence classes—from *highlighting*—visual encoding to indicate each item's equivalence class. In figure 56, selection gestures in the `Selected Genre List` view actually cause musical genres to be *deselected*; invisibility is the encoding used to "highlight" unselected items.

Coordinated Queries allows visualization designers to define complicated relationships between selection and filtering in views. However, the results are not always usable. If the magic selection pattern had only one view, for instance, clicking an item would cause it to disappear

from the visualization completely.

## 6.4 Ordering Patterns

Ordering patterns involve the visual prioritization of data items in views. The prioritization is usually spatial, determining the vertical, horizontal, or z-order placement of data items. There are many possible ways of coordinating views through data ordering, three of which are:

- *Shared Priorities.* Views display items from the same data set in the same order.
- *Top Picks.* Selected items appear before unselected items.
- *Proximity Sort.* Items closest to the mouse pointer come first.

### 6.4.1 Shared Priorities

The shared priorities pattern has the same purpose as brushing (shared selection): visual accentuation of certain records over others across multiple views. Whereas brushing accentuates items arbitrarily selected by the user, shared prioritization accentuates items according to the magnitude of some ordinal function of their attributes.

In Coordinated Queries, views prioritize records using sort expressions. By sharing a sort expression, multiple views can prioritize data in a common way. In figure 57, four scatter plots of the same data share an expression that causes counties to be visually ordered from left to right on decreasing total number of votes.

When views process data for display, logical ordering using sort expressions occurs prior to any visual ordering that occurs as a side effect of rendering. In some views, visual ordering is a function of view type (such as the top-to-bottom organization of rows in list views). In other views, visual ordering is determined by the visual encoding (such as $(X, Y)$ glyph position in

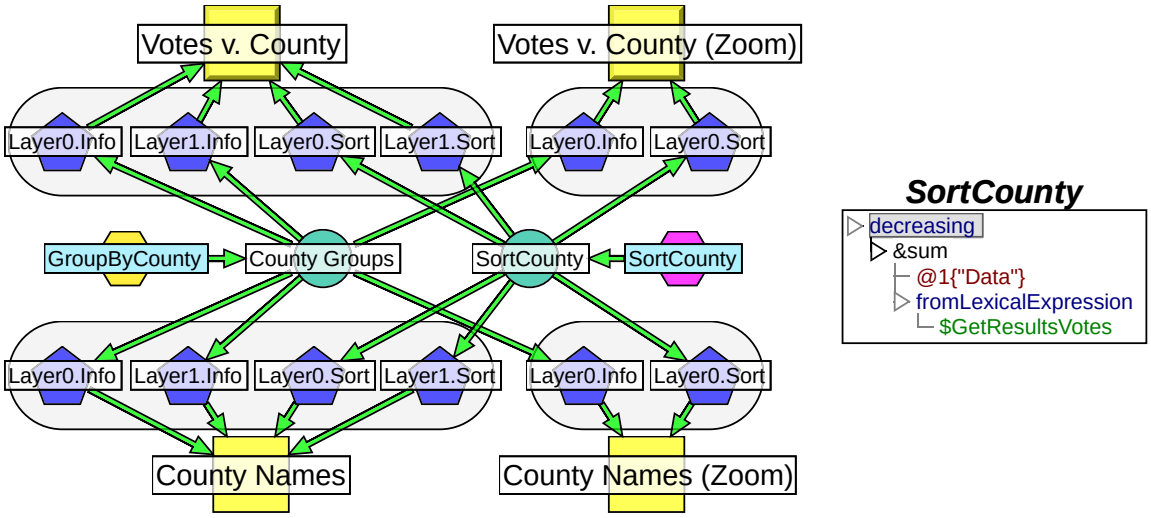



Figure 57: Coordinated query graph for shared priorities (see figure 40F). Four scatter plots arrange counties from left to right in order of decreasing number of total votes. Despite filtering out unselected counties prior to sorting, the two "zoom" scatter plots are able to use the same sort expression as the two unfiltered plots.

scatter plots). Prioritization of data in Coordinated Queries is flexible because logical ordering and visual encoding are both based on user-defined expressions.

### 6.4.2 Top Picks

Like projection and filter expressions, sort expressions can be defined in terms of selection variables. As a result, it is possible to create a variety of ways to coordinate selection and ordering across views. For instance, views can sort selected items before unselected ones, optionally subsorting on a derived attribute. In lists and other naturally ordered views, this approach is useful for visually grouping items of particular interest to users. In scatter plots and other spatial views, this approach is useful for assuring that selected items are not hidden underneath unselected items.

The top picks pattern is an example of selection-ordering coordination *in a single view*. Selecting items causes them to move "in front" of unselected items. In figure 58, selected

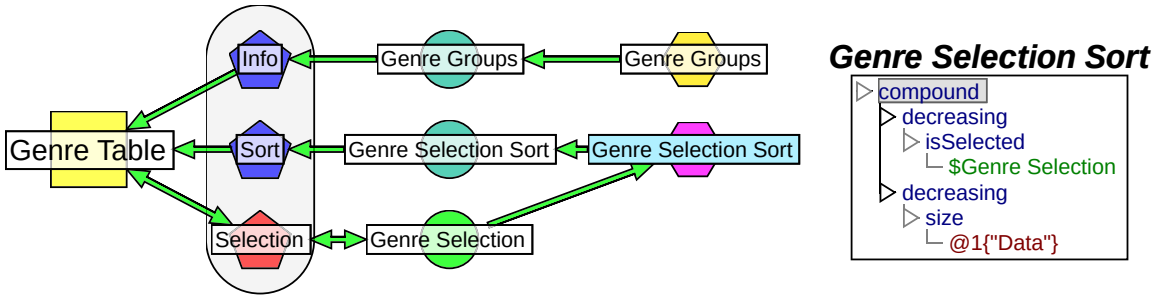



Figure 58: Coordinated query graph for the top picks pattern (see figure 42A). Selected genres appear at the top of a table view. Within selected and unselected rows, subsorting places genres with more albums closer to the top of the table.

musical genres appear at the top of a table view. Deselecting a genre returns it to its former position amongst the unselected items. The result is like magic selection, but confined to a single view.

### 6.4.3 Proximity Sort

Sort expressions can also be defined in terms of navigation variables. The resulting navigation-ordering coordinations can involve points, ranges, angles, and other interactive spatial parameters of scatter plots and other views. In the proximity sort pattern, items in a view are sorted in order of increasing distance from the current mouse position in a scatter plot. Proximity sort is related to a similar pattern, similarity sort, in which items in one view are ordered on a measure of their similarity to an item selected in another view.

Figure 59 shows how the order of musical albums in a table view is coupled with mouse movement inside three scatter plots in a 3-D scatter plot matrix. Although this navigation occurs in only two of three dimensions at any given moment—depending on which scatter plot contains the mouse cursor—the sort expression correctly calculates zero distance in the third dimension. Moreover, because the calculation takes place in view coordinates rather than screen coordinates, the expression scales each dimension to keep any one of them from

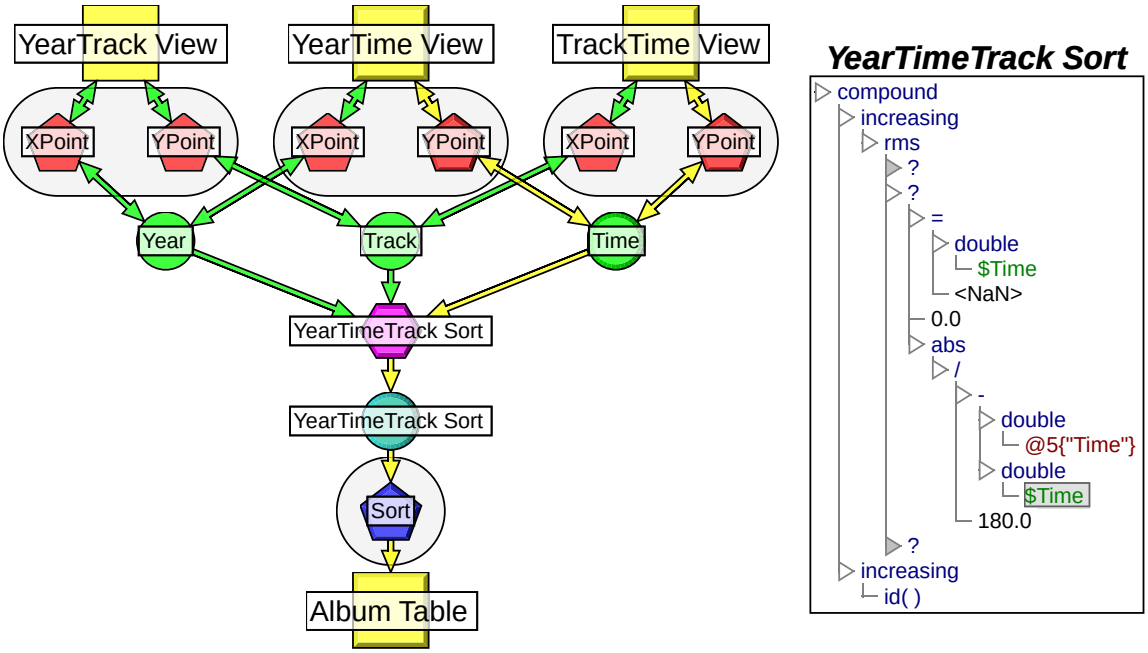



Figure 59: Coordinated query graph for the proximity sort pattern (see figure 42B). Moving the mouse over one of the views in the scatter plot matrix causes the albums table to be sorted in order of increasing root mean square distance (decreasing proximity) from the current `(year, time, track)` location of the mouse pointer. Album order is not affected when the values of the `Year`, `Track`, and `Time` range variables are all undefined, i.e. when the mouse is in none of the scatter plots.

dominating the distance measure.

## 6.5 Containment Patterns

Containment patterns involve the presentation of additional information in existing views. Using containment patterns, it is possible to display and summarize multiple data sets with multiple visual encodings in a single context. Useful containment patterns include:

- *Layered Views.* Stacked scatter plots with synchronized scrolling.
- *Inset Views.* Miniature overviews on top of detail views.
- *Inline Detail.* Additional layers that summarize data items.

- *Nested Views.* Glyphs that visually encode entire data sets.
- *Nested Lenses.* Transparent views in draggable, stretchable frames.

### 6.5.1 Layered Views

Layered views (such as *piles* in DEVise) enable users to visualize multiple data sets using different visual encodings in a single context. A common use of layering is to visualize a single data set using a layer to highlight items that have been selected in a lower layer. Because Improvise scatter plots have an adjustable number of layers each defined by its own info, projection, filter, and selection properties, they can also visually encode items using compound highlighting schemes.

Figure 60 shows how a four layer scatter plot draws a map using four different projections of two data sets. The bottom layer draws all counties. The top three layers fill, highlight, and label only the counties that are involved in the selected election. Drawing labels in the highest layer keeps them from being obscured by shapes in underlying layers. The combination of layering and compound glyphs provides extensive control over the z-order of items drawn in scatter plots.

### 6.5.2 Inset Views

Clever placement of views can increase their usability by suggesting how they are coordinated. In Improvise, visualization designers lay out views interactively in a layout editor (figure 32), in which they can specify: (1) containment of views in hierarchies of panels rooted in frames; (2) constraints that determine the relative placement of views in each panel.

The inset views pattern is a variation of the overview+detail pattern in which the overview is placed on top of the detail view. Positioning the inset in a corner of the detail view mimics

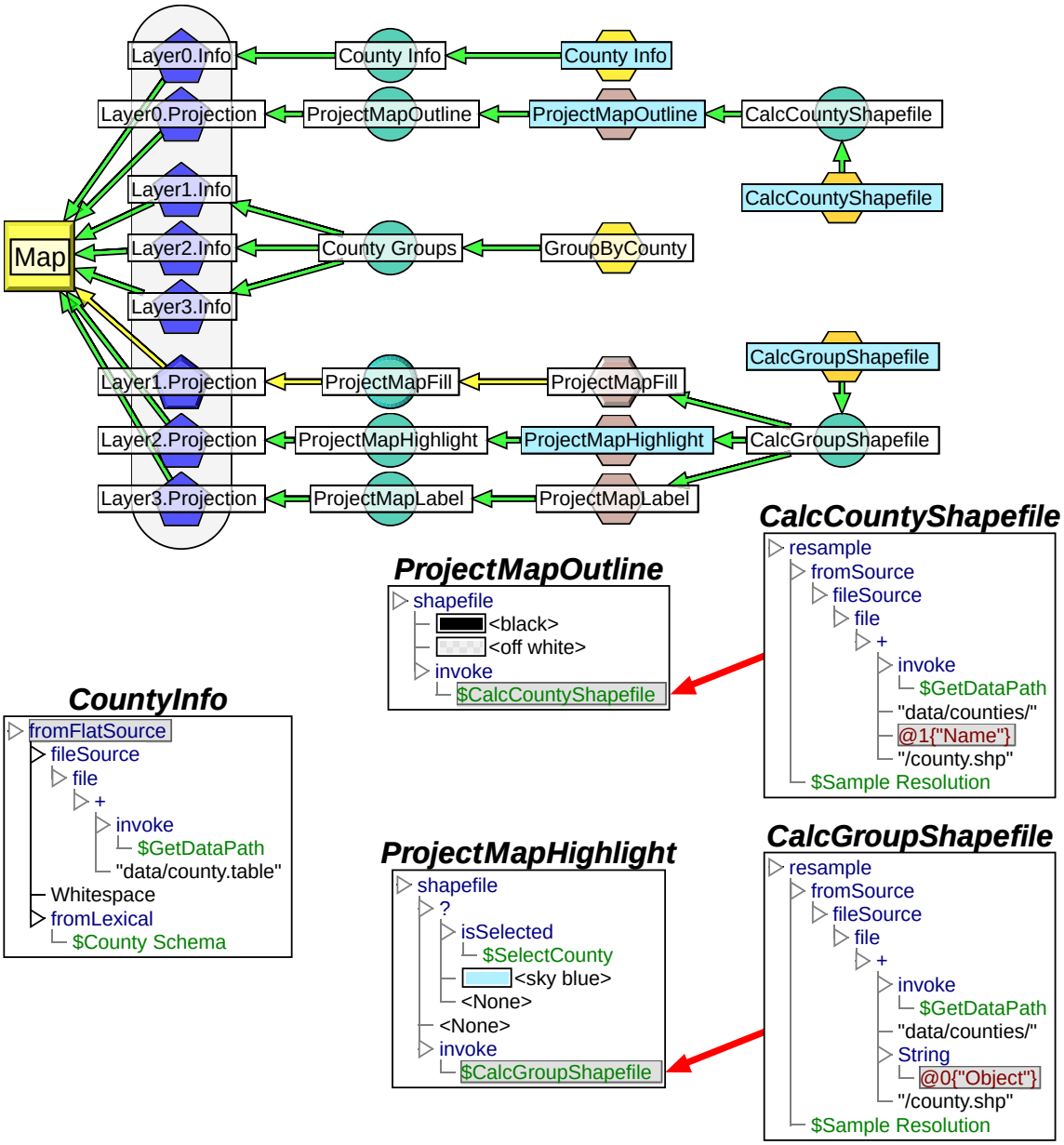



Figure 60: Coordinated query graph for a four layer scatter plot (see figure 40H). The bottom layer draws counties independent of voting results. The top three layers draw different projections of voting results for counties involved in the selected race. All four layers invoke expressions to load and downsample county shapefiles for drawing.

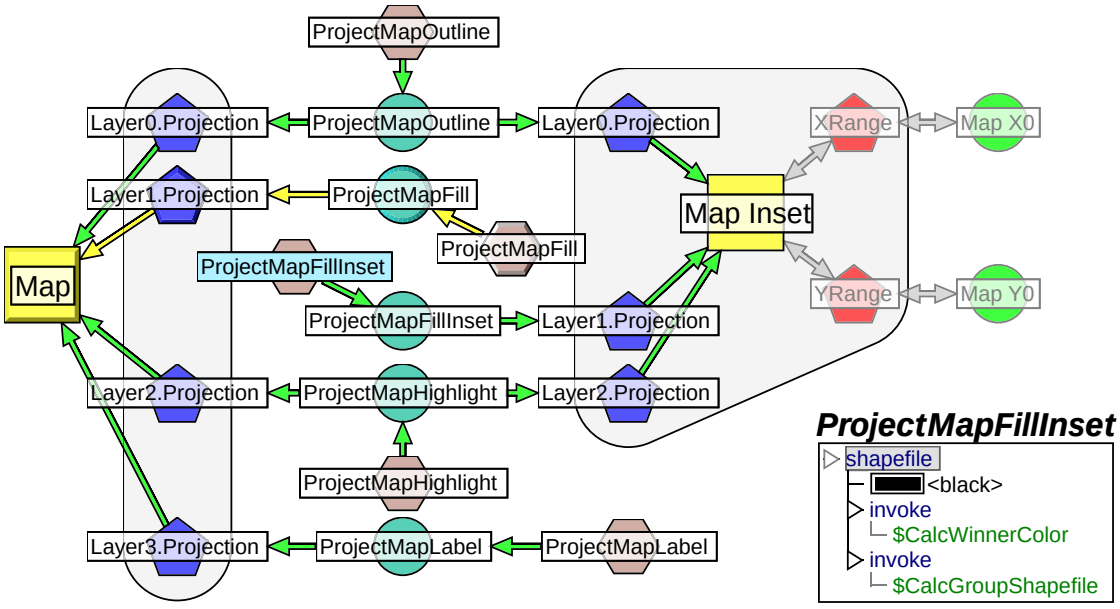



Figure 61: Coordinated query graph for inset views (see figure 40G). The layers of an inset scatter plot draw the same county-level map of Michigan as in the main scatter plot, sans county labels and with decreased detail in the fill layer. The `X` and `Y` ranges of the inset view are locked to prevent panning and zooming.

the appearance of insets in static geographic maps. In figure 61, an inset version of the state map fills each county with the winning party color; the main map draws detail as well, in the form of a nested bar chart summarizing relative candidate vote totals above each county name.

### 6.5.3 Inline Detail

In many visualization systems, moving the mouse cursor over a view raises popups containing details of data item(s) at that point. For instance, clicking a shape in a DEVise scatter plot creates a short-lived window that lists the raw attributes of the data item represented by the shape. In GeoVISTA *Studio* [137], clicking a glyph that represents an entire data set creates a long-lived window containing a regular view of that data.

In the inline detail pattern, an additional layer provides additional details about data item(s) at a particular point in a scatter plot. Figure 62 shows how the top layer summarizes voting

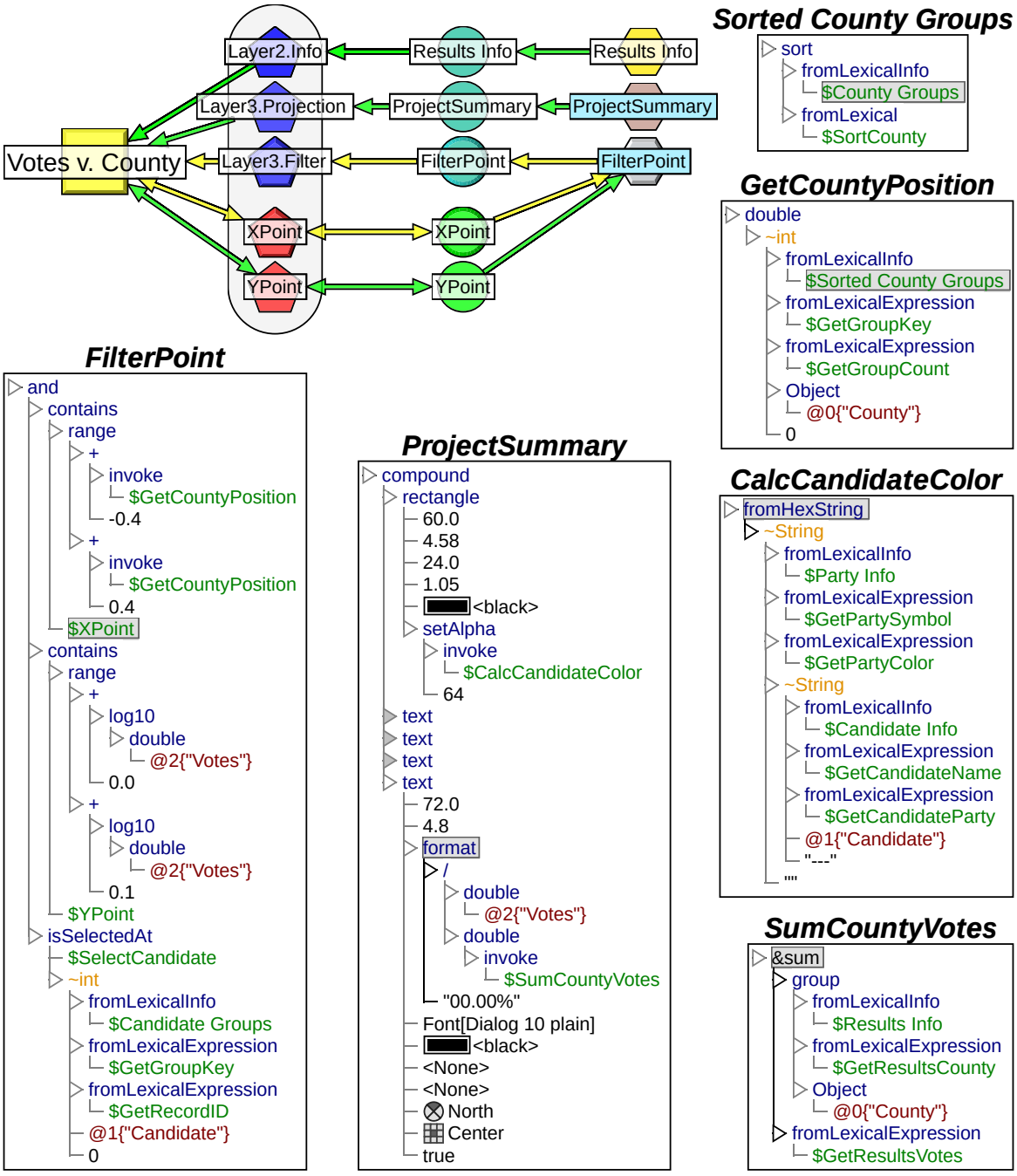



Figure 62: Coordinated query graph for the inline detail pattern (see figure 40I). The top layer of the `Votes v. County` scatter plot summarizes vote results for glyphs at the current mouse location, including winning candidate, total votes, and vote percentage.

results at the cursor location. Although the layer projects details for every data item into the same spot in the top right corner of the scatter plot, it filters out results not located at the current navigational point. As a result, inline detail of multiple results occurs only when they overlap, i.e. when they are from the same county or involve similar number of votes.

### 6.5.4 Nested Views

Nested views enable exploration of a group of related data sets by displaying each data set in its own view, all of which are contained in a larger view. In DataSplash, portals are clickable hyperlink windows into other data spaces. In Improvise, nested views are special glyphs in which the value being visually encoded is an entire data set. Because all Improvise views use projection expressions to generate glyphs, they all can contain nested views. In figure 63, a list visually encodes data files as a filename next to an icon that shows the data as 3-D points. The projection expression that draws each list item generates a nested 3-D view glyph by applying a second projection to the data from the corresponding file.

### 6.5.5 Nested Lenses

See-through tools [16] are draggable windows that modify the appearance of the user interface beneath them. In visualization, lenses are transparent controls that modify the visual encoding (including visibility) of data in underlying views [126]. In Improvise, a nested lens is a portal (section 6.2.4) that acts like a transparent scatter plot, translating and scaling its contents relative to its parent scatter plot. By moving and resizing the portal, the user can focus on additional aspects of data in a particular region of its parent view.

Figure 64 shows how a nested lens can be used to draw roads on top of a county-level map of census data. Because lenses are an extension of regular portals, they navigationally

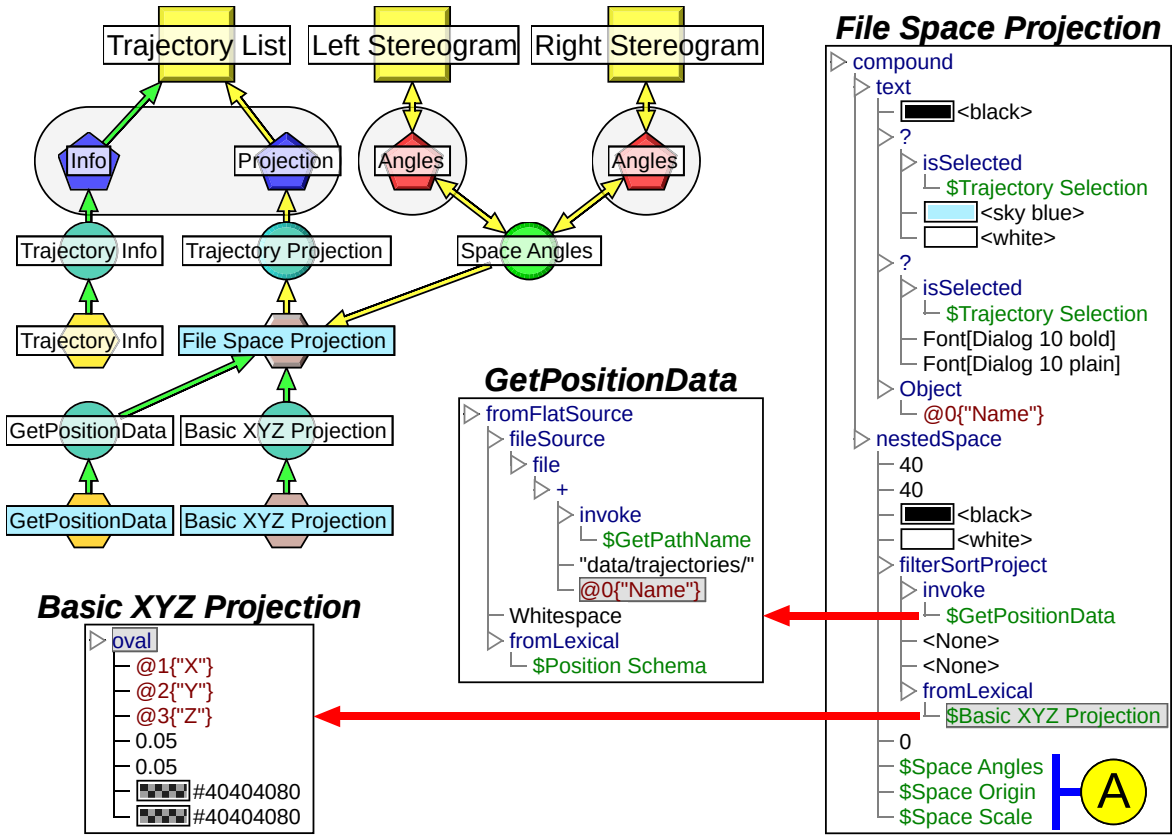



Figure 63: Coordinated query graph for nested views in a list of available data sets (see figure 39G). Each item in the list consists of a formatted file name and a nested 3-D plot. These plots are navigationally coordinated with the main 3-D stereogram through variables that define camera position and orientation (A).

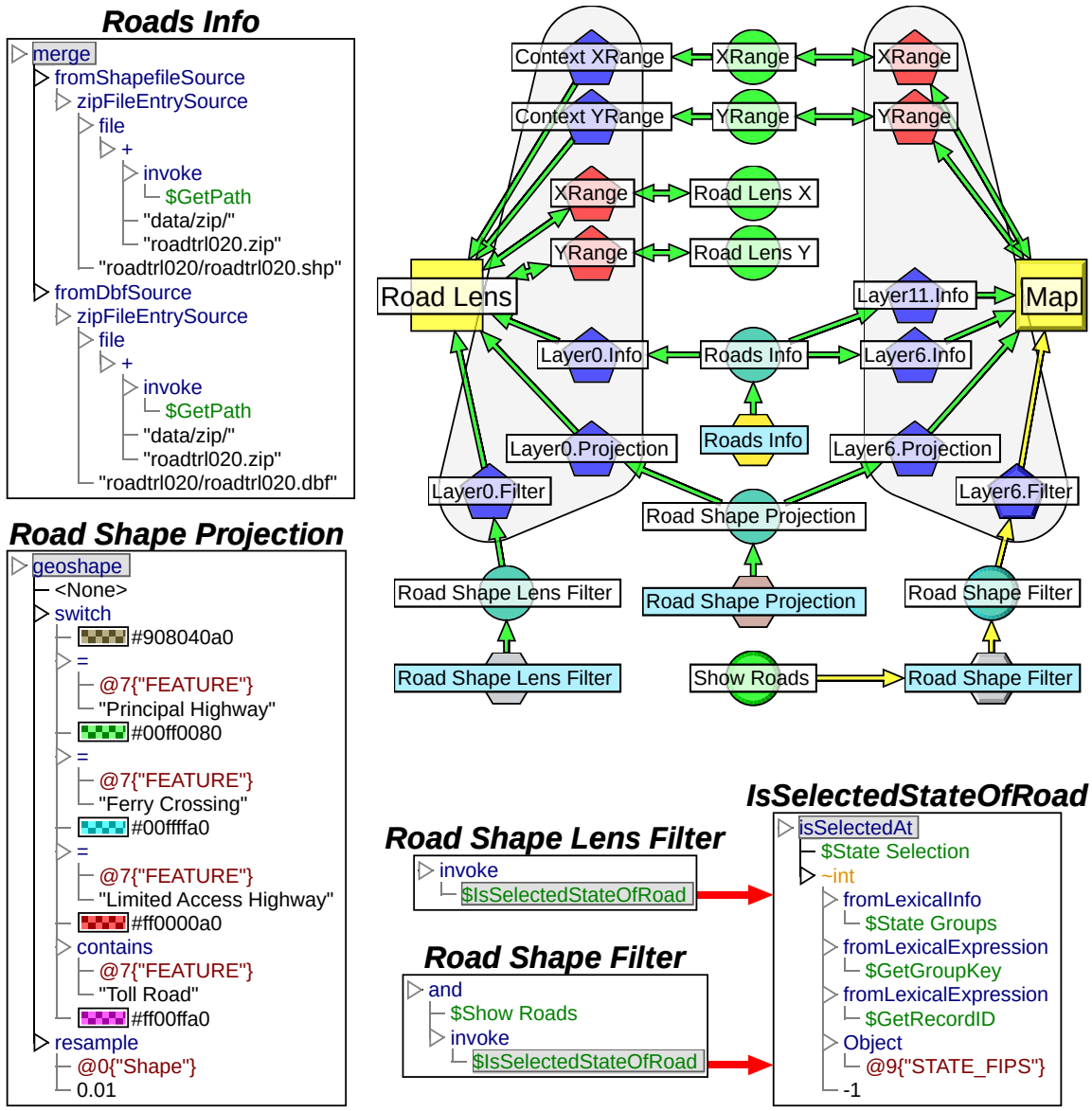



Figure 64: Coordinated query graph for the nested lenses pattern (see figure 43B). A layer of the scatter plot draws color-coded polylines for roads in selected states, but only if the `Show Roads` flag is true. A portal always draws roads with the same visual encoding, but only inside its frame. The portal can be dragged and stretched to reveal roads in different areas of the map.

coordinate with their parent views in the same way (see figure 48). Users can specify the data, filter, and visual encoding of nested lenses just like any other view. In this case, the lens replaces the road layer in the map.

## 6.6 Mutation Patterns

Mutation patterns involve radical alteration of visual encoding in response to navigation. When users indicate interest in a particular region of the displayed data space (by pointing, panning, zooming, etc.), mutation patterns differentiate data items in that region from those in other regions. As special cases of navigation-dependent encoding patterns, mutation patterns can add detail or transform spatial attributes (such as location, size, orientation, etc.) Two common mutation patterns are:

- *Semantic Zoom.* Visual scale affects item appearance.
- *Distortion Encoding.* Navigational location affects item size and shape.

### 6.6.1 Semantic Zoom

Zoomable user interfaces [12, 11] allow users to explore large data sets in a compact space. *Semantic zoom* [111] is a form of details-on-demand that lets users see different amounts of detail in a view by zooming in and out. For instance, the layer manager in DataSplash allows users to select the amount of detail by changing a view's "altitude". The view draws data using the visual encodings visible at the chosen altitude.

Semantic zoom in Improvise involves expressions that calculate glyphs as a function of a plot's own `X` and `Y` ranges. Figure 65 shows how a county map depends on two ranges, both directly and indirectly. Although this example demonstrates synchronized zoom between plot

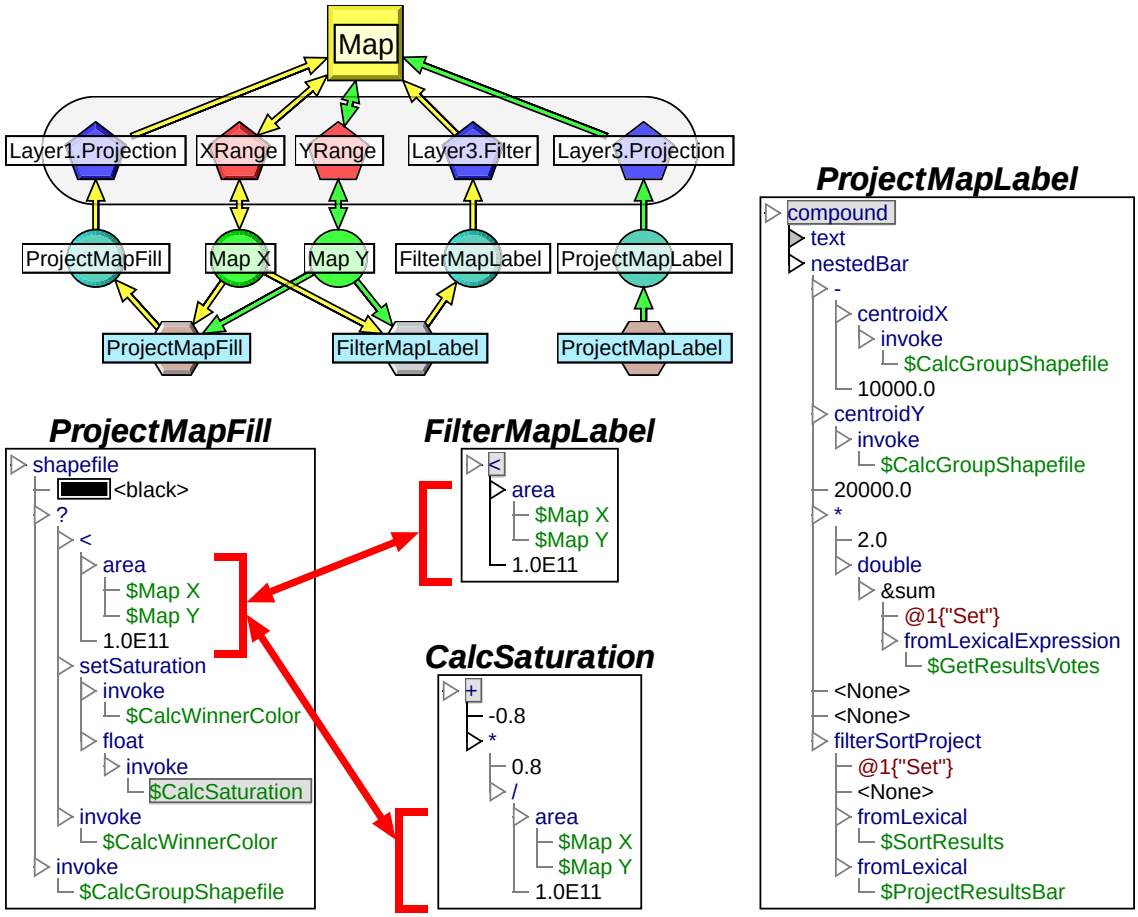



Figure 65: Coordinated query graph for semantic zoom in the county map (see figure 40J). At sufficient zoom, the top layer draws a centered label and a scaled, nested bar plot for all counties. To make the top layer easier to read, the fill layer reduces the saturation of the winning candidate's party color at the same zoom level.

layers, the expressions could be edited to make the layers change detail at different zoom levels.

One-dimensional zooming and multiple levels of detail are also straightforward.

### 6.6.2 Distortion Encoding

Focus+context approaches are another means to explore large data sets in limited space. Distortion techniques [24] such as fisheye lenses [49] transform the visual space in which data is drawn. In particular, spatial transformation functions have been applied to visualization of

graphs [86], tables [119], and geographic maps [122].

In Improvise, views apply location-specific visual encodings to distort data, as shown in figure 66. Expression-based visual encoding makes it possible to apply distortion to items in any kind of view. For instance, list views can have variant row height. However, because list views do not currently track mouse movement across rows, their visual encodings need to be defined in terms of navigational properties of other views. This is one example of how coordination can overcome omissions and limitations in the design of particular views.

## 6.7 Compound Patterns

Coordination in Improvise is not limited to binary relationships between views. Multiple views may simultaneously couple with a common view through a shared coordination. Moreover, any given pair of views may be interactively coupled through multiple coordinations. Compound coordination patterns involve intercoupling of multiple views of multiple data sets through multiple coordinations to create complex interface behaviors, including:

- *Compound Lenses.* Overlapping lenses that modify data appearance in concert.
- *Compound Brushing.* Multiple independent selections of the same data set.
- *Small Multiples.* Views that display slices of a data set in parallel.
- *Multiview+Detail.* High-dimensional overviews of multiple lower-dimensional detail views.
- *Multiform.* Multiple dissimilar but reinforcing presentations of the same data set.

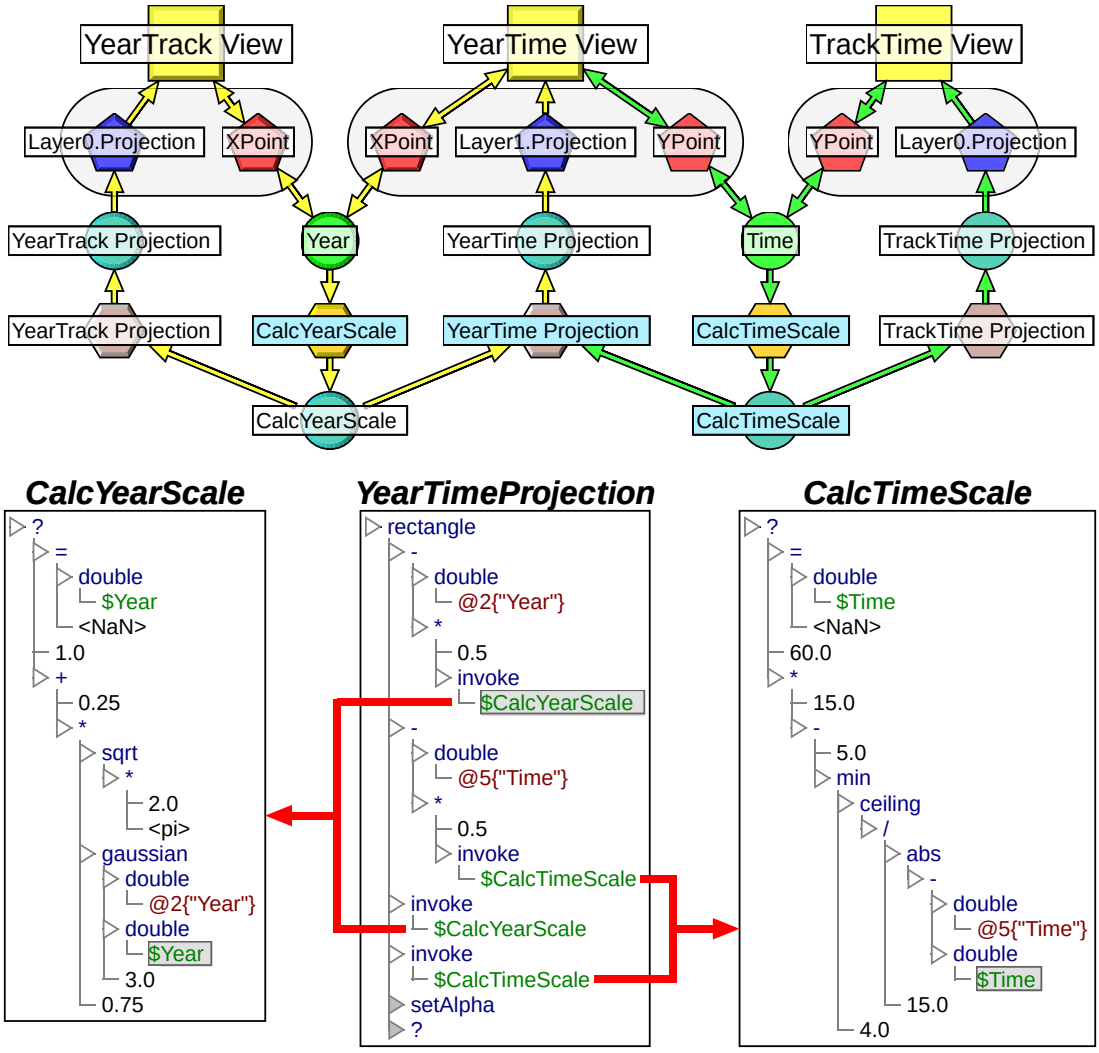



Figure 66: Coordinated query graph for distortion encoding (see figure 44). The width of rectangle glyphs in the `YearTime View` scatter plot is a gaussian function centered on the year at the current mouse location; the height is a five-tier step function centered on the album duration at the same location. Glyphs in the other two scatter plots are scaled similarly.

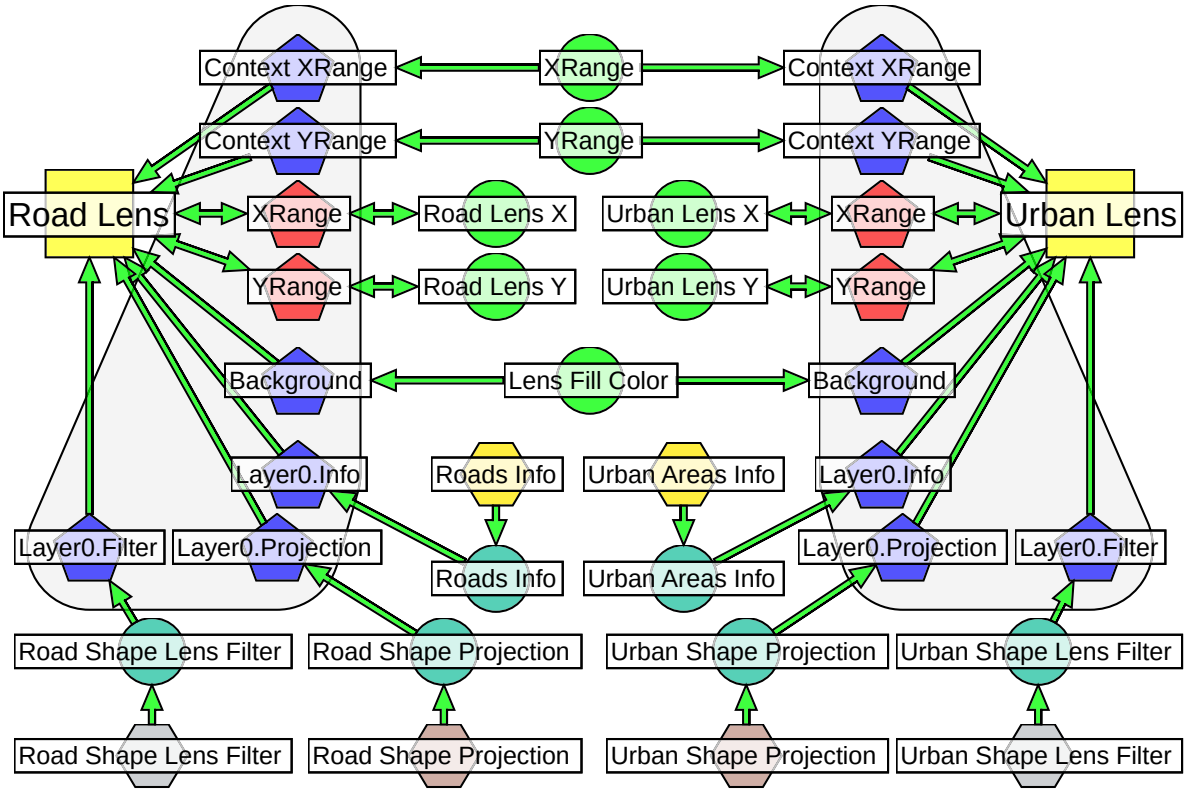



Figure 67: Coordinated query graph for implicit compound lenses (see figure 43D). One nested lens displays roads in selected states. A second lens displays urban areas. Roads are drawn over urban areas inside the spatial intersection of the two lenses.

### 6.7.1 Compound Lenses

Nested lenses can act in concert to modify the appearance of data in views, much like movable filters [48]. Composition of multiple lenses in a single view can be implicit or explicit. Implicit composition occurs when independent lenses overlap, coincidentally producing an additive graphical effect in their intersection. Explicit composition occurs when visual encoding of data in the view depends on the position of lenses.

Figure 67 shows implicit composition of two lenses. Although the lenses draw different data and move independently within the same context, they could share data, projections, filters, or ranges. Sharing a range would couple the lenses in one dimension, much like synchronized scrolling between views.

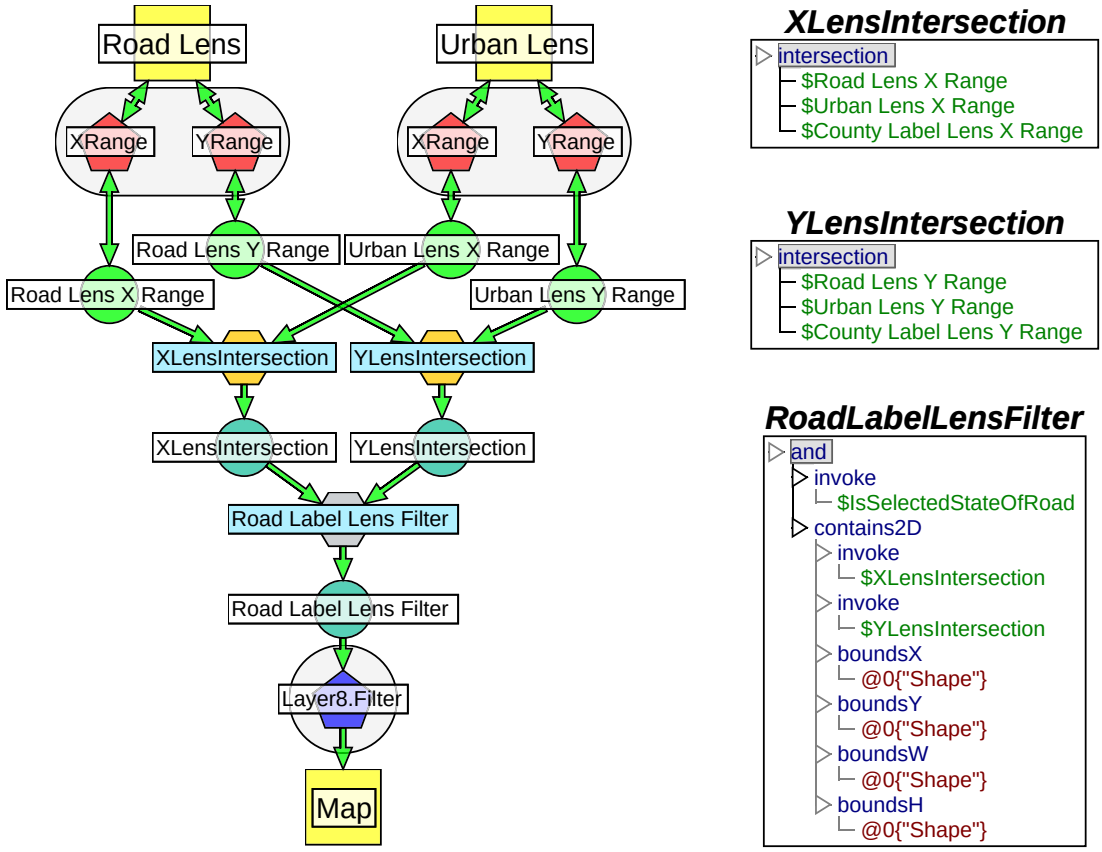



Figure 68: Coordinated query graph for explicit compound lenses (see figure 43E). The map scatter plot labels only those roads that are completely contained in the rectangular intersection of the lenses. (The county label lens is not shown in the graph.)

In explicit composition of lenses (figure 68), a layer of the containing scatter plot draws inside the intersection of the lenses, as calculated from their range properties. The coordination is similar to navigation-dependent filtering, except that navigation occurs in the immediate context of the filtered data. Moreover, other focus+context effects can be applied by customizing the coordination in terms of:

- *Visual encoding.* Like navigation- and selection-dependent encoding, compound lenses can involve projection rather than filtering.

- *Drawing locus.* Additional graphics can be drawn inside any or all of the lenses as well as in the view.

- *Layers.* Views and lenses can draw additional graphics in any layer, not just the top one. For instance, *cel lensing* involves drawing in the bottom layer of the view, behind the view's primary data items. (A *cel* is a sheet of transparent film used in the production of cartoons.)

- *Navigation.* Visual encoding can be defined in terms of the lenses' individual `X` or `Y` ranges, rather than both, creating visual effects in infinite vertical or horizontal strips.

- *Composition.* Filtering and projection can involve the union (or another set operator) rather than the intersection of lens regions.

### 6.7.2 Compound Brushing

In the selection-dependent encoding example (section 6.3.3), the filter and projection expressions depend on selections on either candidates or counties. It is straightforward to extend these expressions to depend on conjunctions or disjunctions of selections. The effect is similar to additive encoding of selection highlighting in interactive externalizations [143] and composite brushing in linked 2-D/3-D scatter plots [114].

Compound brushing is conjunctive or disjunctive highlighting across independent selections of the same items in different views. Figure 69 shows an example of compound brushing, in which the visual encoding of items in a table view depends both conjunctively and disjunctively on independent selection of items in three scatter plots. As this example suggests, compound brushing can be visually conjunctive—meaning that each selection affects a different visual attribute—as well as logically conjunctive.

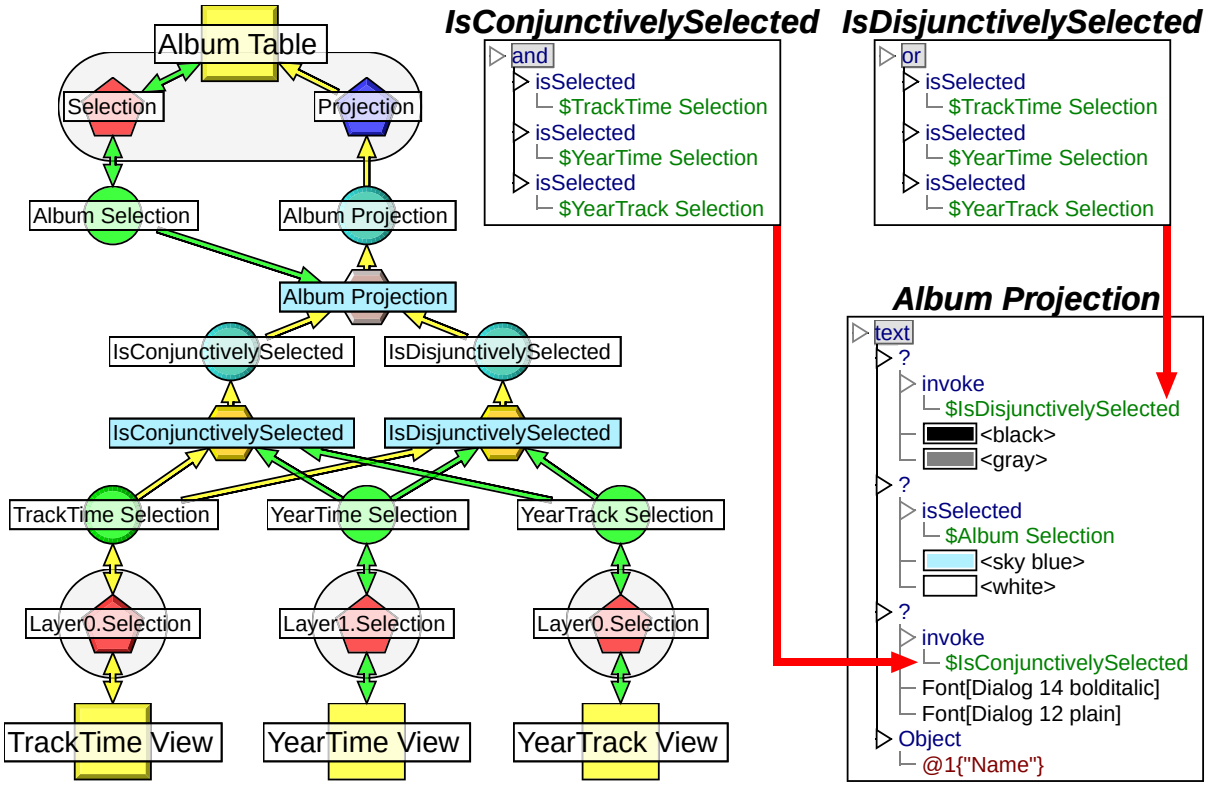



Figure 69: Coordinated query graph for compound brushing (see figure 42C). The visual encoding of album names in the table view is a function of independent selection in three scatter plots. Albums selected in at least one scatter plot are drawn darker; those selected in all three are drawn larger. Albums can also be selected in the table view, independent of their selection in the scatter plots.

### 6.7.3 Small Multiples

Small multiples [139] display slices of a single data set in parallel using the same visual encoding. An advantage of small multiples is that users can focus on differences in the data between slices rather than differences in presentation. In Improvise, small multiples can be created using individual top-level views (laid out vertically or horizontally in a frame, for example), or displayed in a single top-level view using nested views. Unlike the previous application of nested views, which displayed an arbitrary number of parallel data sets, small multiples involves splitting a single data set into a small number of categories or value ranges.

In the visualization in figure 42, the small multiples technique is used twice to show how the

distribution of album track count plotted against total album time varies from decade to decade and from genre to genre. Figure 70 shows small multiples as an application of nested views to a data set that has been grouped on an attribute with a small number of discrete categories, musical genre in this case. To create the other instance of small multiples, the albums data set is grouped into ten-year intervals (using a simple function on the year attribute as the key expression in the grouping query).

### 6.7.4 Multiview + Detail

The number of parallel perceptual channels available for simultaneous visual encoding limits the number of data dimensions that can be displayed in one view. As a result, a single view is often insufficient to provide a complete overview of high-dimensional data. The multiview+detail pattern addresses this problem by extending the overview+detail pattern to multiple overviews. The overviews visually encode overlapping subsets of data dimensions, acting together as a single "multiview".

Figure 71 shows an example of the multiview+detail pattern in which the multiview is made up of three scatter plots and two tables. Each overview coordinates with the detail view through navigation- or selection-dependent filtering. Together, the five overviews filter the data items shown in the detail view.

### 6.7.5 Multiform

In multiform visualization [120], multiple views of different types present the same data set in different ways simultaneously. By providing alternative avenues for navigation and selection, users can adopt their own exploration strategies. Variation in the display of a data set can be achieved using different view types or different visual encodings.

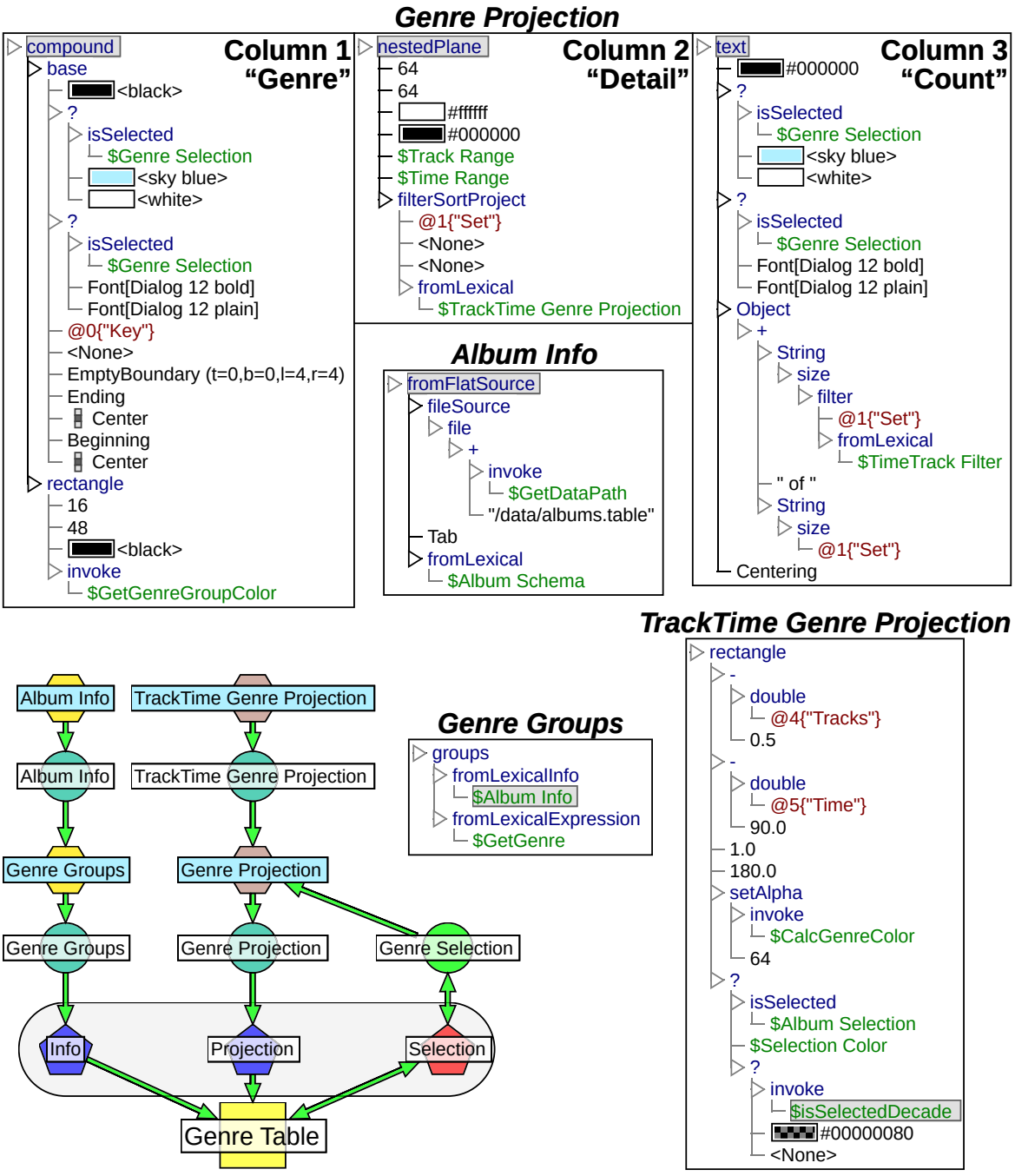



Figure 70: Coordinated query graph for small multiples (see figure 42D). Groups of albums of each musical genre are visualized in parallel in the rows of a table view. The table serves simultaneously as a key (color by name), an overview (album track and time information), and a summary (album count) of each genre.

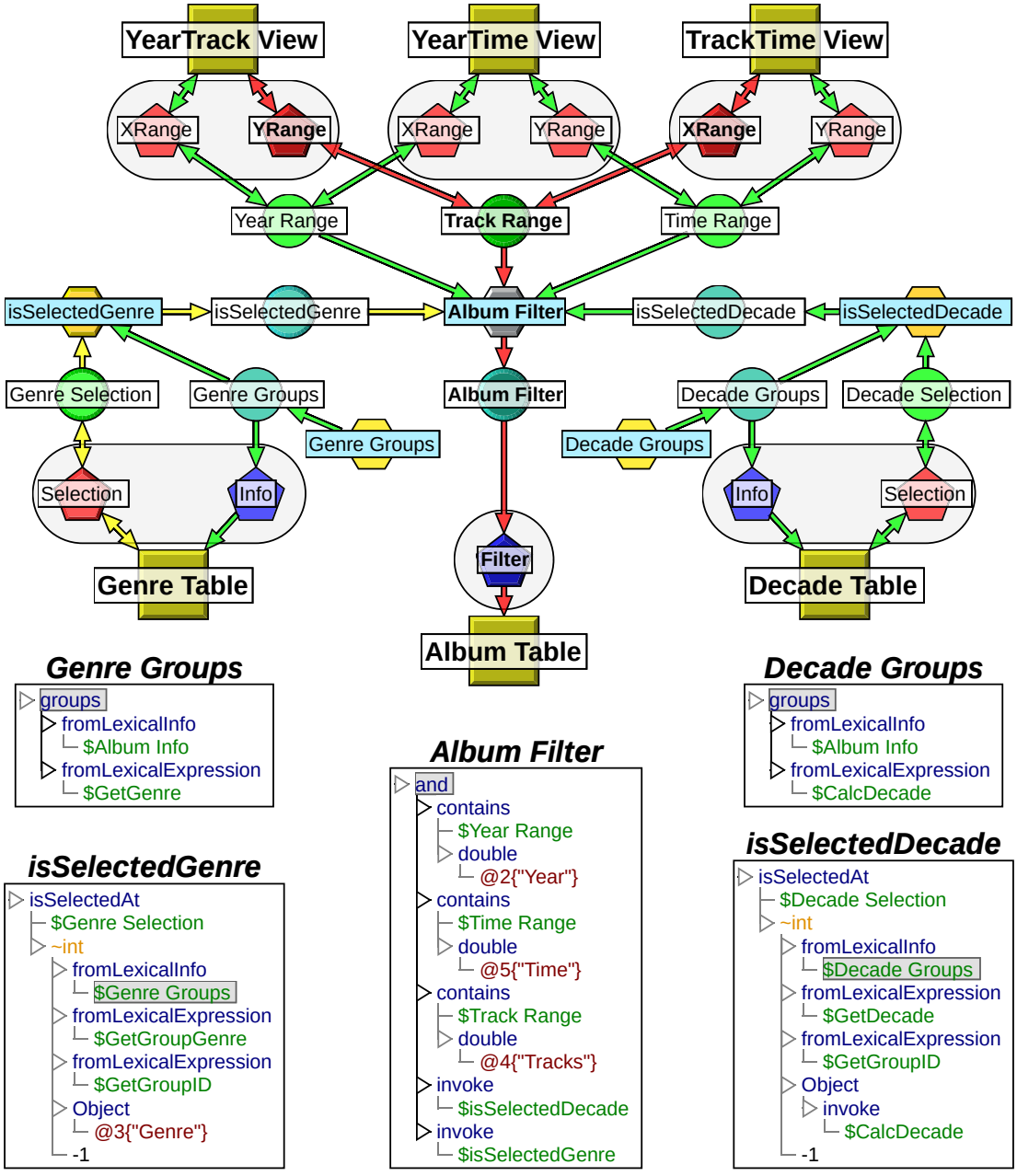



Figure 71: Coordinated query graph for multiview+detail (see figure 42E). The album table view shows only albums that are visible in all three views of the scatter plot matrix and whose genre and decade are selected in the other two table views.

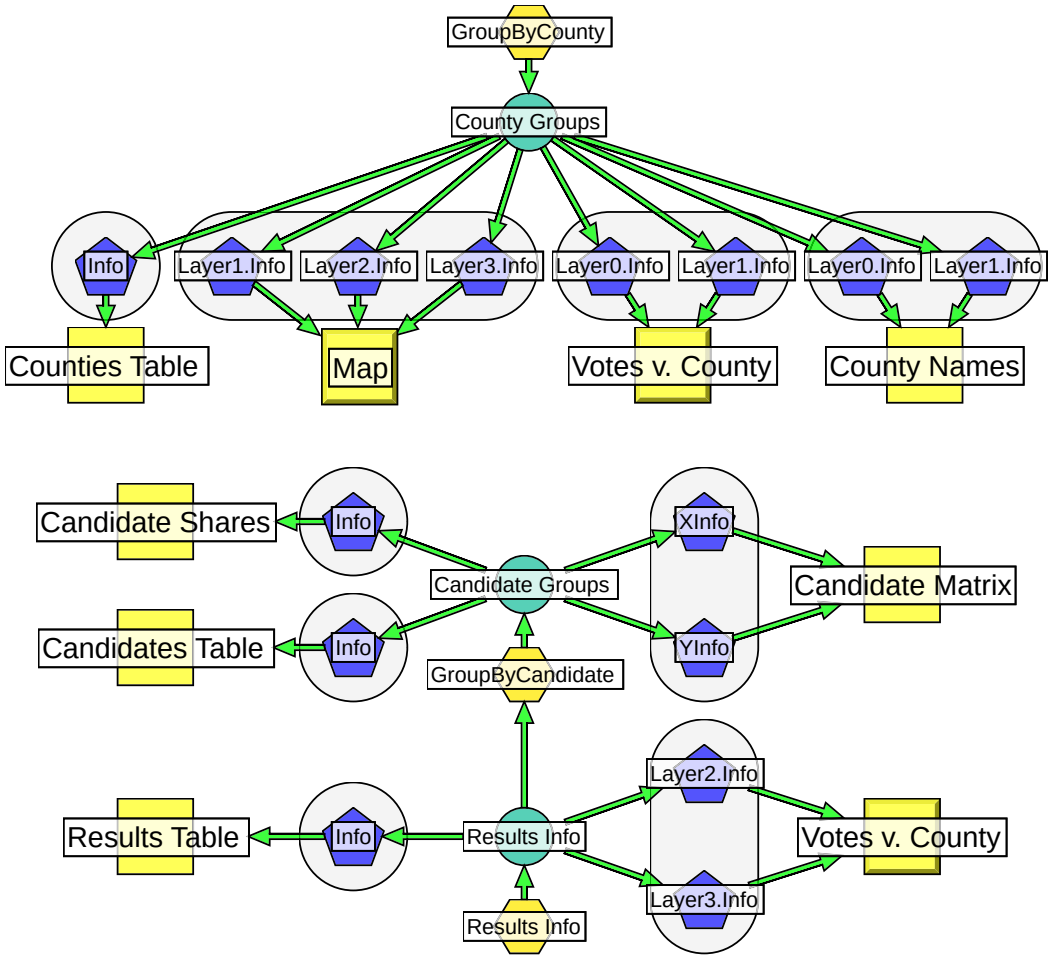



Figure 72: Coordinated query graphs for the multiform pattern (see figure 40K). A table view, map, and two scatter plots visualize the same county data set (top). Five views of different types process and display the same voting results data set in different ways (bottom).

Figure 72 shows two examples of multiform visualization. The data set of vote totals grouped on county is shown in four views as bar charts (`Counties` table), pie charts (`Map` scatter plot), colored text (`County Names` scatter plot), and vertically aligned sets of rectangles (`Votes v. County` scatter plot). The same data set grouped on candidate is shown in five views as a pie chart (`Candidate Shares`), bar charts (`Candidate Matrix`), rows of text (`Results Table`), aggregated rows of text (`Candidates Table`), and rectangles sorted on decreasing total county votes (`Votes v. County`). As these examples demonstrate, multiform visualization in Coordinated Queries is a simple matter of sharing variables

that have data sets as values.

## 6.8 Extensions

The previous patterns testify to the flexibility of coordination in Improvise. Nevertheless, the expressiveness of Coordinated Queries as a visualization language is limited in practice by the library of implemented components, including:

- *Views and controls.* Improvise provides common, simple view types such as scatter plots and axis controls. More esoteric views such as treemaps [74] and hyperbolic graphs [86, 98] have yet to be added. Moreover, the appearance and behavior of existing views is constrained by their implementation in terms of particular properties.

- *Data processing algorithms.* Coordinated Queries currently provides the equivalent of common relational operations (select, project, merge, group by, aggregate) except join. A vast assortment of visualization algorithms could be custom programmed as data processing modules.

- *Function operators.* Improvise provides about 1400 function operators defined in terms of about 100 object types. Function operators can be added in a modular fashion to access additional data sources and formats, apply newly added data processing algorithms, or perform domain-specific calculations.

Examples of coordination patterns that cannot be instantiated at present include *sliding layers* and *popup detail*. Both of these examples would be possible with minor additions to the Improvise implementation.

### 6.8.1 Sliding Layers

Sliding layers is a variation of layered plots in which the `X` and `Y` ranges of different scatter plot layers are decoupled. Users pan and zoom layers independently, allowing them to compare regions of spatially encoded data by visual overlap.

To add sliding layers to Improvise scatter plots, it would be necessary to: (1) add two range properties for each layer; (2) allow users to switch between layers quickly during navigation. Like layered plots, sliding layers could be coordinated with each other, allowing arbitrary sets of layers to be navigationally coupled on `X` and/or `Y`. Sliding layers could also be implemented as independent single-layer scatter plots with transparent backgrounds, arranged in a stack. Although this layout would preclude direct navigation in all but the topmost scatter plot, indirect navigation would be possible by coordination with views and controls outside the stack.

### 6.8.2 Popup Detail

Popup detail is a variation of overview+detail and inline detail in which clicking an item creates a window containing a new view. The window can be temporary or "sticky". A popup view can visually encode a single record or an entire data set, depending on what the clicked item represents. For example, clicking a region in a map view in GeoVISTA *Studio* creates a sticky window containing a scatter plot of the region's abstract data.

Popup detail views in Improvise would be similar to nested views. The contents of popup views could be defined in terms of attributes that specify the background color, ranges, data, and queries of the popup view. Mouse clicks in glyphs in the primary view would generate popup views from these attributes. Unfortunately, popup views created in this way would be second-class views, incapable of interaction or coordination. First-class popup views would require extension of Coordinated Queries to support indirection in expressions, such as to refer

to a range variable as a variable (for binding to a range property in a first-class popup scatter plot) rather than the range value assigned to it (for rendering the static contents of a second-class popup scatter plot).

## 6.9 Summary

Visualization systems generally provide a few common coordination types such as synchronized scrolling, brushing, overview+detail, and semantic zoom. As such, most systems support only a small subset of the 28 patterns described here. Many patterns are supported by at most a handful of systems, if any. Of these patterns, many were discovered unexpectedly while building visualizations in Improvise, including inline detail, distortion encoding, and compound lenses. The magic selection pattern is a rediscovery of an old user interface metaphor, the resource mover. All three ordering patterns are unique to Improvise.

Discovery of these patterns resulted from looking for useful and usable ways to display and interact with particular high-dimensional data sets. This experimental process consisted largely of mixing and matching variable operators in expressions, thereby manipulating the semantics of interactive dependencies between views by varying the dimensionality of coordination. This variability is what gives Coordinated Queries greater coordination flexibility for visual encoding over other coordination approaches. This variability is increased by allowing filtering and ordering of data to be coordinated as well. Although some visualization systems also allow coordinated filtering, coordinated ordering is a novel feature of Coordinated Queries.

A major goal for Improvise has been to increase coordination flexibility substantially without significantly decreasing ease-of-use, as compared to similar systems like DEVise and Snap; to make simple coordinations (like synchronized scrolling) easy, and complex coordinations

(like semantic zoom) possible. Although comparative user studies will be needed to determine if this goal has been achieved, the possibilities for highly-coordinated visualization in Improvise appear to be limited only by the creativity of visualization designers.

Well-known coordination patterns appear to be much easier to instantiate in Coordinated Queries than other coordination approaches because views are connected indirectly through navigational parameters, selections, data, and visual encodings that are shared objects which can be edited on the fly. There is no need to link views pairwise or in sequence to achieve complex coordination semantics. By way of example, the bottom half of the visualization in figure 39 contains ten scatter plots, four portals, and 13 axes but uses only eight numeric ranges for navigational coordination.

Future work might focus on adapting existing coordination architectures to work on top of a common coordination abstraction layer built around Coordinated Queries. An abstracted coordination architecture would offer users a choice of several "connect and move" styles, analogous to user-selected "look and feel" styles available in many current graphic user interface environments. Users might start with a simple coordination model, working up to more flexible coordination models as their experience increases and analysis requirements evolve. Expert users might choose to work with Coordinated Queries directly most of the time, yet still be able to prototype and refine visualizations rapidly when called for in time-pressured situations.

The numerous coordination patterns described in this chapter demonstrate many useful ways in which views can be interactively coupled. Projection, filtering, and ordering of data in views can be specified flexibly and precisely in terms of navigation and selection anywhere in a visualization. It is likely that many more useful coordination patterns have yet to be discovered.

# Chapter 7

# Metavisualization

## 7.1 Overview

Exploratory visualization environments allow users to build and browse coordinated multiview visualizations interactively. As the number of views and amount of coordination increases, conceptualizing coordination structure becomes more and more important for successful data exploration. Although numerous visualization systems implement ad hoc ways of displaying the structure of their visualizations, the systematic application of exploratory visualization techniques to visualize interactive structure is a new idea unique to Improvise.

Exploratory visualization of coordination and other interactive structure *in situ*, directly inside a visualization's own user interface, is called *integrated metavisualization*. This chapter presents a model of integrated metavisualization, describes the problem of capturing dynamic interface structure as visualizable data, and outlines three general approaches to integration. Metavisualization has been implemented in Improvise, using *metaviews*, *lenses*, and *embedding* to reveal the dynamic structure of its own highly-coordinated visualizations.

## 7.2 Motivation

Exploratory visualization systems have evolved into full-featured interactive environments that enable users to build highly-coordinated visualizations with many views rapidly and easily.

These systems follow a recent trend toward interactive construction approaches that employ combinations of well-known coordination patterns and view types. In these systems, users load data, create and layout views, specify visual abstractions, and establish coordinations as they explore their data.

Unfortunately, the more views and coordinations a visualization has, the harder it is to conceptualize. To make matters worse, coordinations rarely have an explicit graphical presence in the interface, appearing as a side effect of interaction, if at all. Users must resort to trial-and-error—or help from a visualization expert—to learn how a visualization works.

North and Shneiderman [105] have pointed out the difficulty users have understanding how coordinated multiple view (CMV) visualizations work. They suggested allowing users to edit and debug visualizations using a graph representation of visualization structure, an approach employed by data flow visualization systems like AVS [144], apE [40], and Data Explorer [92]. They further proposed showing coordinations in action by augmenting visualization appearance, but made no recommendations of how to do so.

Visualization systems such as LinkWinds [73], Tioga-2 [157], DEVise [90], Snap-Together Visualization [107], and the Infovis Toolkit [44] are designed for interactive exploration of relational data. Many of these systems support interactive editing of the views, coordinations, and visual abstractions that make up a visualization. DataSplash [110] provides a zoom layer manager for editing how tuples appear at different levels of magnification. Polaris provides zoom graphs [133] in which nodes are drawn in a graphical notation that describes a visual query at each vertex of a data cube. Visualization Schemas [103] is an extension of Snap that displays a coordination graph in which nodes represent views and edges represent coordinations. However, these features are built in and allow little or no user customization. As a result, the user's ability to examine visualization structure is limited to the needs anticipated by the visualization system developer—precisely the situation that exploratory visualization approaches

are meant to overcome.

In *integrated metavisualization*, the interactive structure of a coordinated visualization can be explored by visualizing it *in situ*, directly in the visualization itself. Metavisualization has been implemented in Improvise by reusing its existing coordination architecture and views to reveal the evolving structure of its own highly-coordinated visualizations during interactive data exploration.

## 7.3 Metavisualization

*Metavisualization* is a general term that refers to visualization of another visualization's structure and operation. Metavisualization targets a broad spectrum of visualization users. It is instructive to consider how metavisualization might benefit four overlapping groups into which visualization users generally fall:

- *Users* can use metavisualization to learn how to interpret and manipulate existing visualizations.
- *Designers* can use metavisualization to design, build and debug visualizations.
- *Developers* can use metavisualization to design, implement, evaluate, and optimize visualization systems.
- *Researchers* can use metavisualization to explore, evaluate, and formalize visualization techniques.

Members of each group might be interested in different aspects of visualization, from views (along with sliders and other controls) and coordinations (navigation and selection), to processing (querying and rendering) and interaction (events and traces). While users would typically

be interesting in learning basic operation, designers would be interested in interaction and processing for debugging purposes. Historical information about views—such as frequency of use—could help developers target particular views for improved design and optimization. Metavisualization could also be used by developers to compare visualizations in terms of usability. Researchers might metavisualize views, coordination, and interaction across multiple visualizations and visualization systems in the hope of finding patterns that help them model existing techniques and identify new ones.

A metavisualization and its visualization need not run at the same time or place. *Standalone metavisualizations* run independently of their visualizations and display a static data representation, as in [153]. *Dynamic metavisualizations* run at the same time as their visualizations, but in a different place (such as on a remote screen). *Integrated metavisualizations* run at the same time and place as their visualizations. Integration makes it possible for a metavisualization to access and visualize a rich data representation of its visualization, especially if both run in the same process or are otherwise able to share objects or memory directly.

Figure 73 depicts an important special case in which the integrated metavisualization is a CMV visualization. The data being metavisualized: (1) happens to represent the interactive structure of another CMV visualization, and (2) dynamically reflects interaction in that visualization. Four research and engineering problems that must be solved are:

- representing visualization structure as visualizable data (research),
- inventing/adapting visual techniques and views for metavisualization uses (research),
- incorporating metavisual views into visualizations (engineering), and
- notifying the metavisualization of visualization changes (engineering).

The following sections describe how these problems have been addressed in Improvise.

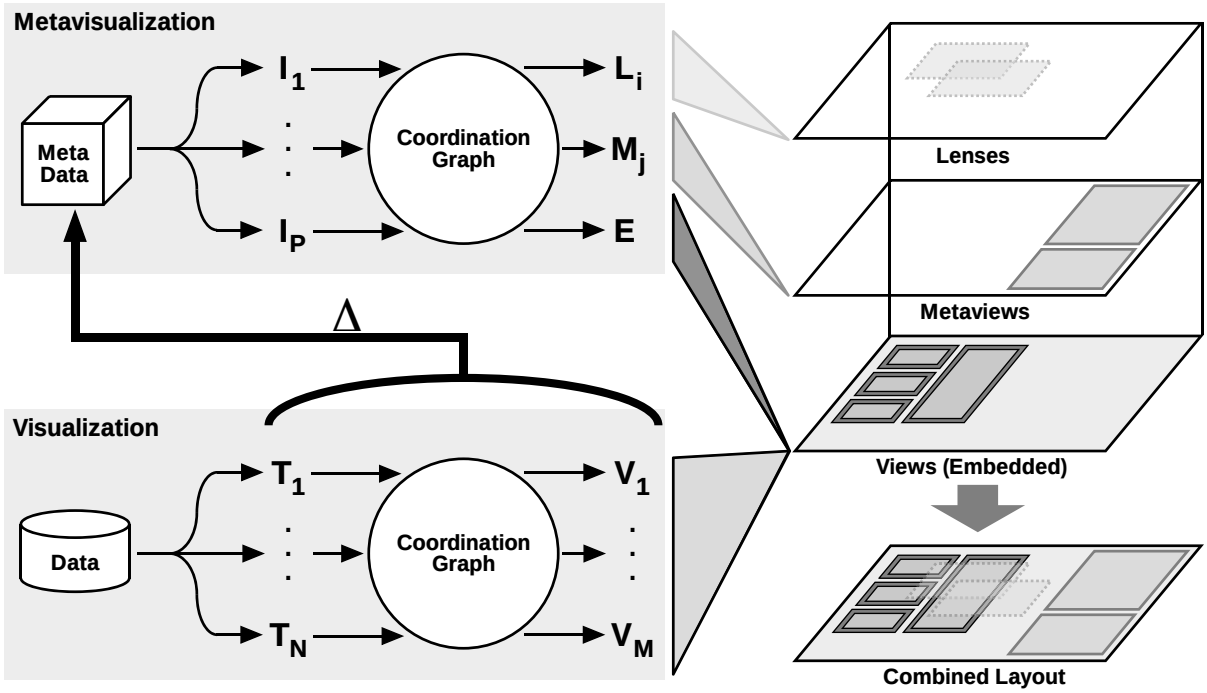



Figure 73: Model of integrated metavisualization. A metavisualization consists of coordinated lenses (L) and metaviews (M) that visualize a dynamic representation (I) of another visualization's views (V), data (T), coordinations, and screen layout. Lenses and metaviews appear directly within a visualization above all regular views. Each view is embedded (E) in a metacontrol that visualizes local information about that view.

## 7.4 Metavisualization in Improvise

The combination of a simple, direct coordination mechanism (Live Properties) with a more powerful, indirect coordination mechanism (Coordinated Queries) in Improvise gives users fine-grained control over the appearance of visualized data. This combination enables users to define complex interactive dependencies between views and other controls in metavisualizations as well as visualizations.

Controls coordinate using a shared object update and notification mechanism (figure 74). Controls interpret these objects as data, data processing operations, limits to abstract spatial extent, and basic aspects of appearance. Visual abstractions are created by projecting and

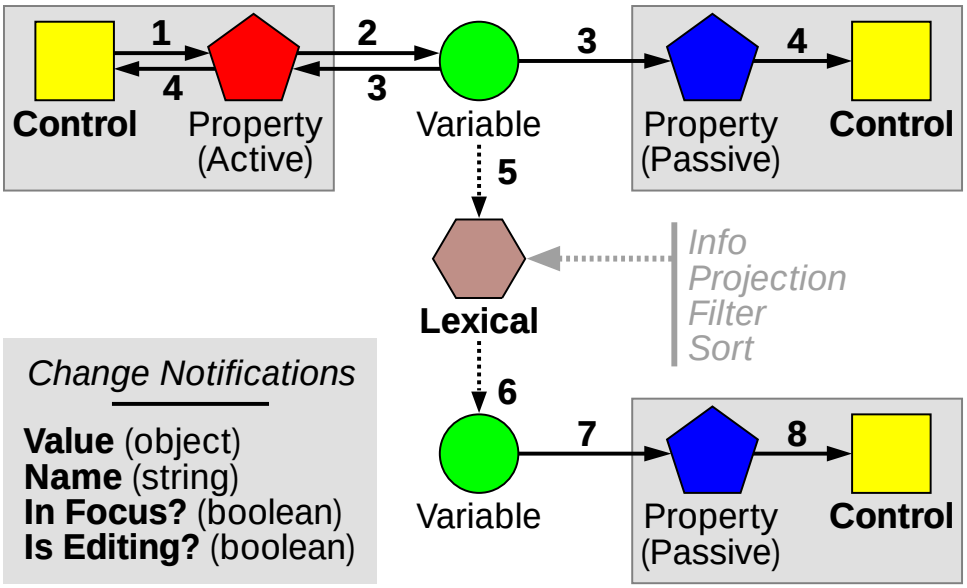



Figure 74: Model of coordination. In response to interaction, a control modifies the value of one of its active properties (1), which assigns the new value to its bound variable (2). The variable sends a change notification to all properties bound to it (3), each of which notifies its control of the change (4). The variable also notifies all lexicals (query objects) whose expressions refer to it (5). Each lexical notifies any variables to which it is assigned as a value (6). When a view receives notification through one of its properties that a lexical value has changed (7, 8), it updates itself by processing the modified query.

filtering data using expressions that can themselves be defined in terms of shared objects. Users interactively edit these expressions to specify what data to draw, how to draw it, and where to draw it in a visualization.

An Improvise (meta)visualization may be thought of as an instance of the Model-View-Controller architecture [83] in which the coordination graph is both controller and model. Acting as a multiple query model, views pull processed data from it to update themselves. Acting as a control flow graph, it pushes change notifications to views. Coordination graphs in Improvise piggyback a kind of "distributed focus" on top of the control flow mechanism (section 3.2.2). Interactive focus in a view causes dependent objects throughout the graph to also be in focus. Distributed focus makes it possible to metavisualize coordinations in action.

Each Improvise visualization occupies a single top-level desktop frame that presents the

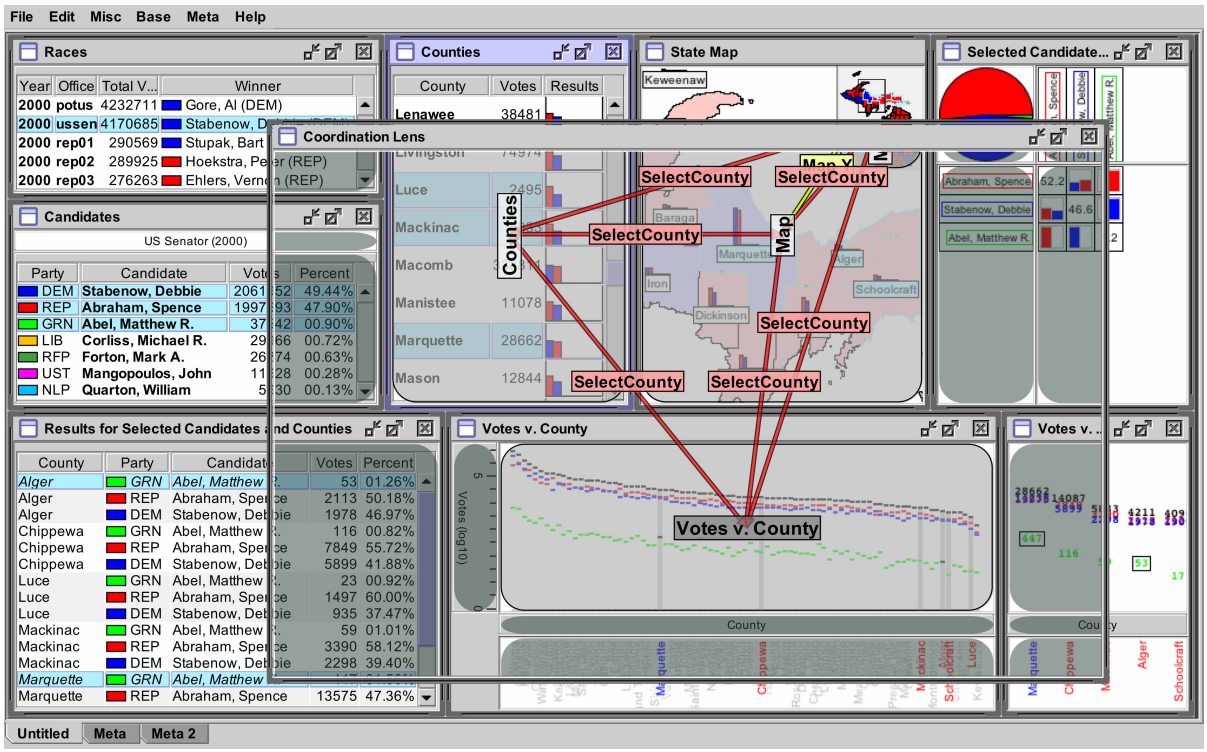



Figure 75: A visualization of county-level election results for the State of Michigan from 1998 to 2004 (see appendix A.3). A tinted lens highlights views, using labeled arrows to reveal coordination on the user's selection of counties in the `Votes v. Counties` scatter plot.

user with a tabbed page metaphor (figure 75). Internal frames contain a tree of panels in which views and other controls are the leaves. Metavisualizations are stored in the same XML file format as regular Improvise visualization documents. When the user loads a metavisualization from the menu, its internal frames are positioned in tab pages beside those of the visualization.

Integrating the components of a metavisualization directly into a visualization allows the user to work with both simultaneously. Integration minimizes the number and duration of "visual context switches" between the visualization and metavisualization, and reinforces the metavisualization because its appearance and behavior continuously reflect the current interactive state of its visualization. However, limited screen real estate often reduces these benefits by forcing the metavisualization designer to place metaviews in separate tab pages from the visualization's own views.

The problem of limited screen space is mitigated by using *metavisual lenses* and *embedding* to modify visualization appearance. Lenses are transparent metaviews that cover all or part of the visualization. Embedding inserts metavisual graphics directly into individual views.

Embeddings, lenses, and metaviews draw in, on, and near the visualization. Like regular visualization views, all three kinds of metavisual display are completely customizable in terms of the projections and filters they use to display data. Improvise users build and browse metavisualizations in exactly the same way as they do visualizations, except that the data sets being visualized are generated internally rather than accessed externally.

### 7.4.1 Metavisual Data Representation

To represent the structure of a visualization as relational data, its views, coordinations, and screen layout must be translated into tables of records. The data representation also needs to capture the ever-changing interactive state of the visualization. Numerous theoretical treatments of visualization techniques provide insight into the problem of capturing a data representation of coordination structure and view layout. Taxonomies of visualized data types [129], interaction semantics [30], and the visualization design space [22] are helpful in choosing which aspects of visualization interface structure to capture as data for metavisualization. Studies of screen layout include Myers' taxonomy of window layouts in user interfaces [100], Kandogan and Shneidermans' evaluation of task-dependent multiple window layout [75], and North and Shneidermans' enumeration of multi-window coordination strategies [104]. The data state model [28] is a universal formalization of visual data processing that encompasses the entire coordination structure of CMV visualizations.

Improvise maintains an interdependent collection of data sets for each running visualization, as shown in figure 76. These data sets enumerate the different kinds of objects that make

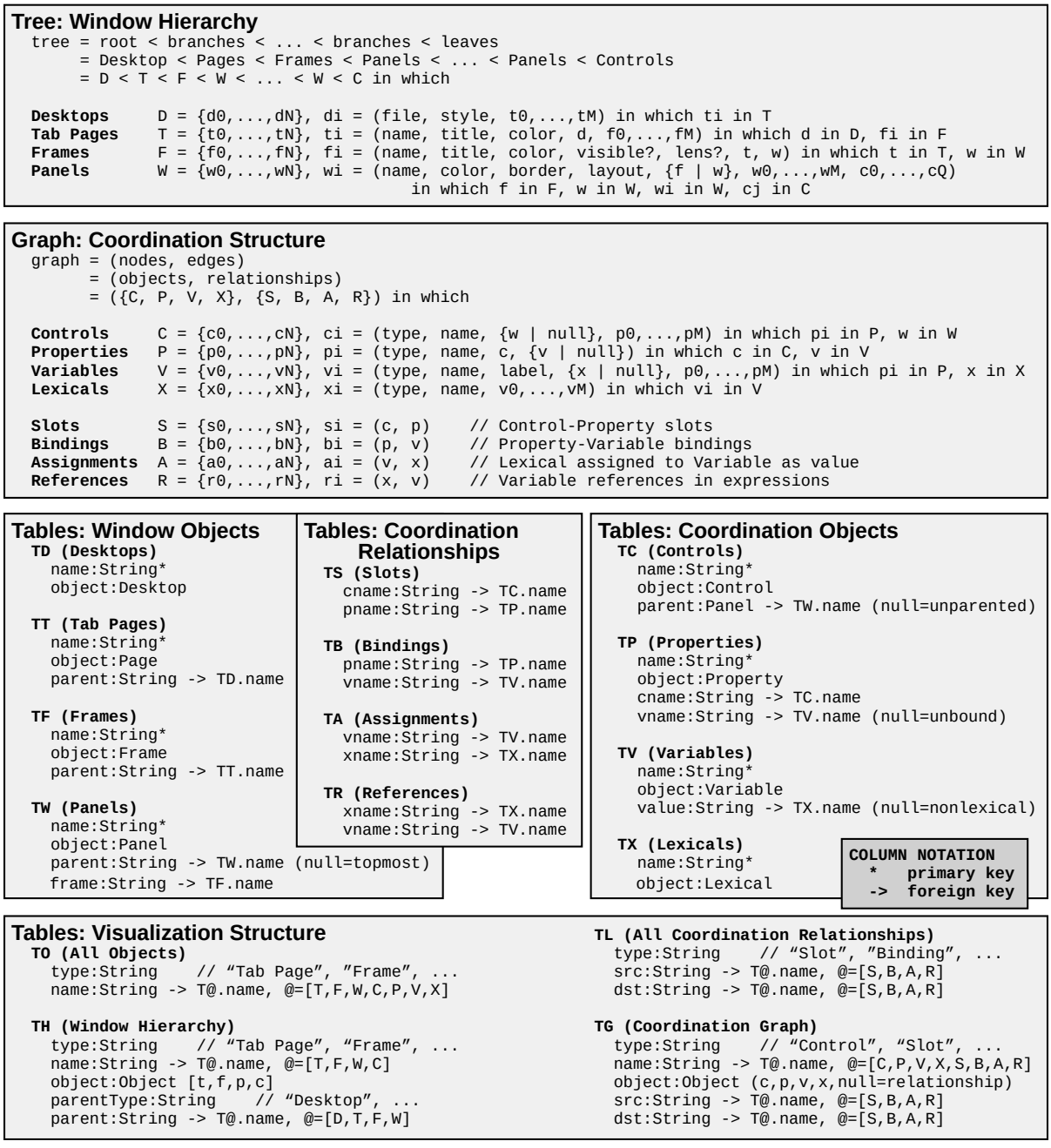



Figure 76: Model of metavisual data representation for Improvise visualizations. Tabular data sets encode the window containment tree, coordination graph, object lists, and coordination relationships of a visualization.

**Window Hierarchy**

| Desktop | Tab Page | Frame | Panel |
|---|---|---|---|
| | create/destroy | create/destroy | create/destroy |
| modify name (document file) | modify name | modify name | modify name |
| modify settings (tab style) | modify setting (title, color) | modify settings (title, style, lens?) | modify settings (color,border, layout) |
| move/resize (on screen) | | move/resize/ show/hide | move/resize/ show/hide |
| add/remove page | add/remove frame | add/remove panel | add/remove subpanel/control |

**Coordination Structure**

| Control | Property | Variable | Lexical |
|---|---|---|---|
| create/destroy | create/destroy | create/destroy | create/destroy |
| modify name | | modify name | modify name |
| *change state (focused, editing)* | *change state (focused, editing)* | *change state (focused, editing)* | *change state (focused, editing)* |
| move/resize/ show/hide | modify value/label | modify value/label | edit expressions/ select data source |
| add/remove property | bind/unbind variable | bind/unbind property | assign to variable |

Figure 77: Interactive events that affect the data representation of Improvise visualizations. Shaded entries indicate dependent changes. For example, hiding a frame recursively hides the panels and controls inside it.

up a visualization, as well as the relationships between them. To support metaviews that draw tree- and graph-structured data, the entire window hierarchy and coordination graph are each encoded in tabular format (as in the Infovis Toolkit), using primary and foreign keys to capture structural relationships.

The contents of these data sets are kept up-to-date by adding, deleting, or modifying records in response to editing and interaction (figure 77). Callbacks are used to monitor changes in native data structures throughout the visualization. The entire data representation can be affected by frequent, redundant changes caused by input as fine-grained as a single mouse movement. To maintain interactivity, updates to the data representation are aggregated on a 50-millisecond time scale. Whenever possible, tables are “touched” rather than rebuilt. These performance

enhancements help keep the metavisualization up-to-date with its visualization during interaction. Liveness in metavisualization is highly desirable because it reinforces the associations that users perceive between the apparent interactive structure in the visualization itself (as seen in its views and layout) and how that structure is displayed in the metavisualization.

Improvise also maintains a history of high-level (variable change, query, render) and low-level (keyboard, mouse, painting) interaction events in the running visualization. Discovering effective ways to explore this data using integrated metavisualization has been left as future work.

### 7.4.2 Metaviews

Metaviews are simply views that display metavisual data. The appearance of metaviews differs from regular views only because the data they display happens to represent the visualization they accompany. The behavior of metaviews differs from views in that views reflect interaction in the visualization, but metaviews reflect interaction in the metavisualization as well as interaction (and other structural changes) in the visualization.

All of the views and other controls used in Improvise visualizations can be reused, unchanged, for metavisualization. Figures 78 and 79 show a metavisualization that contains several kinds of views:

- *Tables*. A projection maps data records into rows, using a column for each field. A filter eliminates rows. In metavisualization, tables are good for summarizing multiple aspects of interactive state in an equivalence class of interface objects, such as all the controls or variables in a visualization.

- *Lists*. Like tables, but a projection maps data records into graphical attributes of a single column. In metavisualization, lists are good for enumerating views or coordinations in a

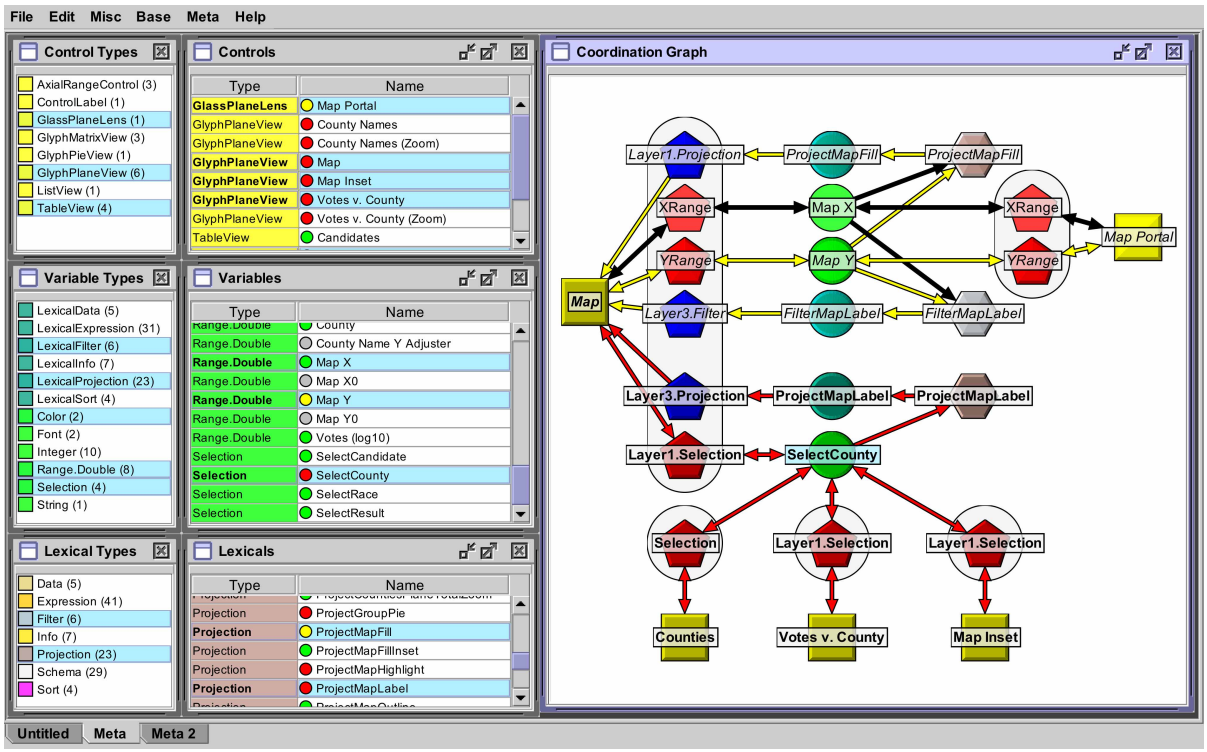



Figure 78: A metavisualization of the elections visualization in figure 75, consisting of seven coordinated metaviews. Three lists enumerate the types of controls, variables, and query objects in the visualization. Three tables show the interactive state (green for inactive, yellow for focused, red for editing) of items of the selected types. The coordination graph shows selected items and the relationships between them.

perceptually natural way, such as to show the interactive state of coordinations.

- *Grids*. Displays a join of two data sets in a grid. A projection is used to render all grid elements. Rows and columns are filtered separately. Using only one data set produces a single row or column that can be used to label columns or rows in other grid views. In metavisualization, grid views are good for showing relationships between objects, e.g. showing which controls are bound to which variables.

- *Graphs*. Two data sets are decoded as nodes and edges. Two projections determine graph structure; two more projections determine node and edge appearance. Two filters elide nodes and edges; dependent edges of elided nodes are always elided. Nodes can be

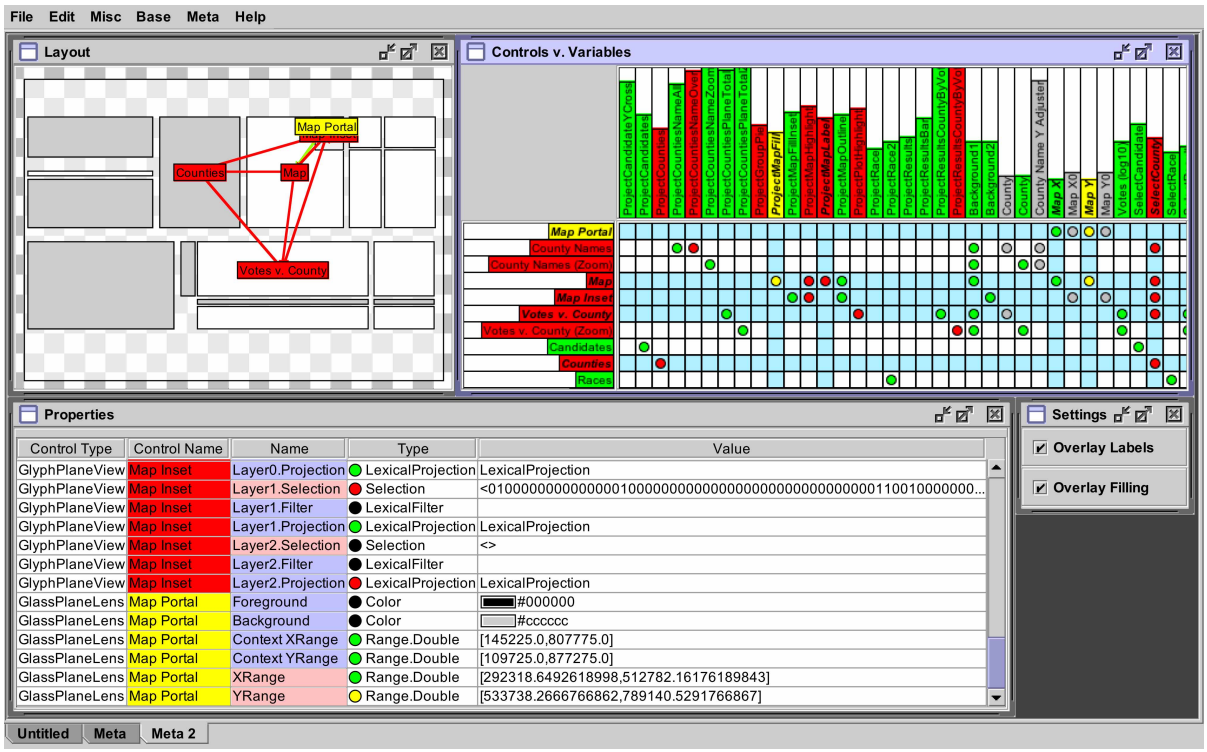



Figure 79: Continued metavisualization of the elections visualization in figure 75, consisting of five additional metaviews. A scatter plot shows a miniature version of the visualization layout, overlaid with the coordinations between active views. A table shows the interactive states and values of all properties of active views. Three grid views show dependencies between controls and variables in the visualization.

positioned using force-directed or manual layout. In metavisualization, graphs are good for showing coordination structure.

- *Scatter plots*. Two numeric ranges determine the visible rectangular region of the cartesian coordinate plane. Scatter Plots can have multiple layers. In each layer, a projection maps records into scalable glyphs. A filter determines which glyphs to draw. In metavisualization, scatter plots are good for showing the screen layout of a visualization in miniature.

Metavisualization designers specify the appearance and behavior of metaviews in exactly the same way as visualization designers specify the appearance and behavior of views, using

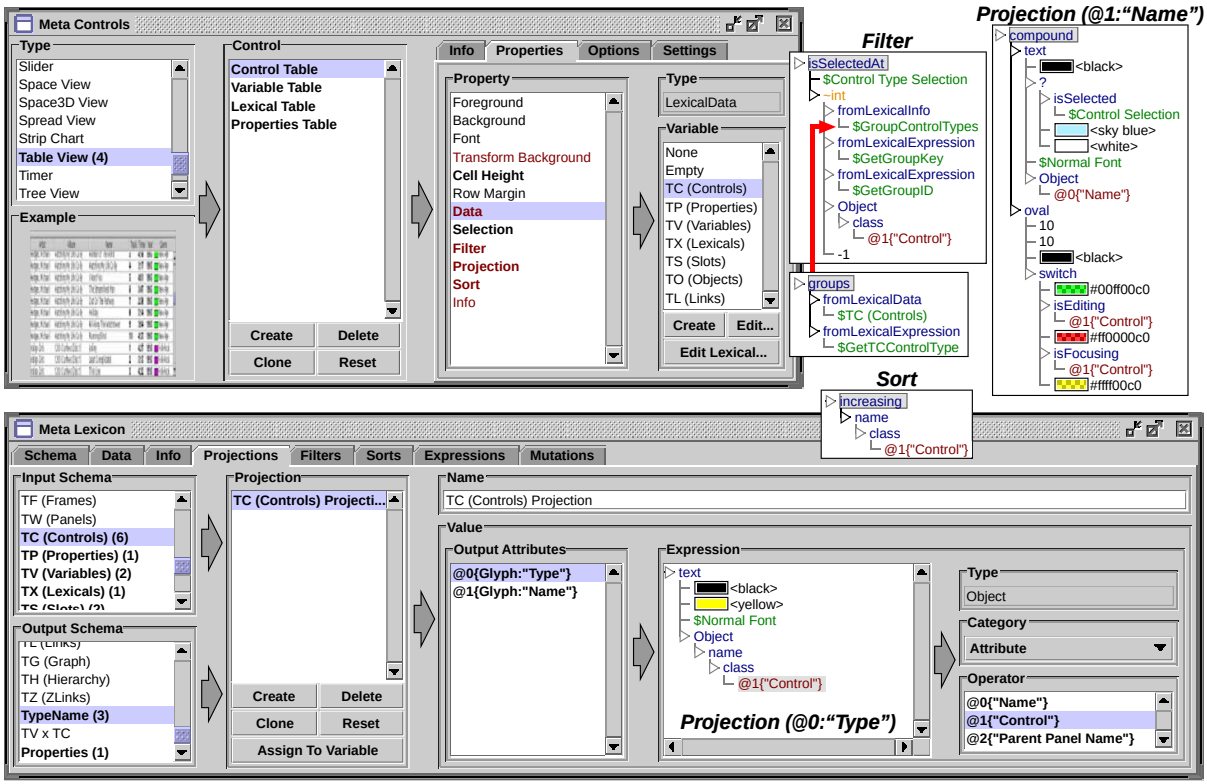



Figure 80: Editing the projection and filter expressions used to draw the table of controls shown in figure 78.

editor dialogs in the existing user interface to build coordinated query graphs (figure 80).

### 7.4.3 Lenses

Metavisual lenses are transparent metaviews that modify the appearance of the underlying visualization. By drawing on top of other views, lenses reveal hidden interactive structure. As with other see-through tools [16], using a metavisual lens is like looking at a visualization through a painted sheet of (possibly tinted) glass. Unlike movable filters [48], which aim to modify the appearance of a visualization's contents, the purpose of a lens is to augment the appearance of a visualization's interface structure.

In Improvise, lenses are implemented as transparent scatter plots that translate their contents relative to their location above the underlying visualization. Lenses are contained in a frame, like any other metaview. By moving and resizing the frame, the user can focus on a small part of the underlying visualization. Lens frames can also cover the entire visualization by extending beyond its bounds.

Figure 75 shows a multilayer lens on top of a visualization of election results. In the bottom layer, the projection calculates the screen bounding box of each control in the controls data set. This box is filled with a translucent color keyed to the control's current interactive state. The name of the control is shown in a label centered in the box. The top layer shows how the currently active control shares variables with other controls using a second projection that draws colored, labeled lines between the centers of controls.

Unfortunately, application of distortion techniques to metavisual lenses is restricted by the need to maintain visual correspondence between the screen location of underlying views and the graphics drawn in the lens. Nevertheless, visualization systems that employ an infinite, scrollable desktop might use distortion to show the hidden sides of the desktop, such as in the perspective wall [96]. Lenses could also be used to alter visualization appearance on a pixel-by-pixel basis, such as by adjusting brightness/contrast to highlight or downplay particular views, or changing color channel values to indicate the degree of staleness in views during querying and rendering.

### 7.4.4 Embedding

Metavisual *embedding* augments the appearance and behavior of individual views in order to indicate their relationship to visualization structure as a whole. Embeddings fall into four categories:

- *Emergent embeddings* are a side effect of tightly coupled coordination, such as synchronized scrolling and other forms of navigation. Frequent updates during interaction reveal the existence of (and, to a lesser extent, the behavior of) coordination between views. Loosely coupled coordination produces a weaker form of emergent embedding, appearing once at the completion of interaction.

- *Implicit embeddings* suggest how views are related based on screen layout, such as horizontal and vertical scrollbars on the sides of a text document. Implicit embedding promotes conceptualization of interactive structure by providing strong hints as to how components are grouped. However, the bounded two-dimensional nature of graphic displays severely limits the amount of implicit embedding. Moreover, poor design choices can result in counterintuitive implicit embedding.

- *Explicit embeddings* draw persistent graphics in views to indicate coordination. Examples include common background colors and labels. Explicit embedding can reinforce implicit embedding, and reduce the effect of counterintuitive layouts. Consistent combination of emergent embedding with implicit embedding can also reinforce user expectations: the implicit embeddings *look* right; the emergent embeddings *act* right.

- *Reactive embeddings* draw temporary graphics (such as custom cursors, tooltips, and navigational guides) in views in response to transient interactions. Reactive embedding can reveal coordination, such as by drawing a colored border around coordinated views while the user manipulates one of them.

CMV visualizations have a limited amount of embedding built-in "for free", in the form of tight coupling (emergent) and matched labels (explicit) between coordinated views. Implicit embedding is a function of design, and can be achieved through careful placement of views regardless of the visualization system used.

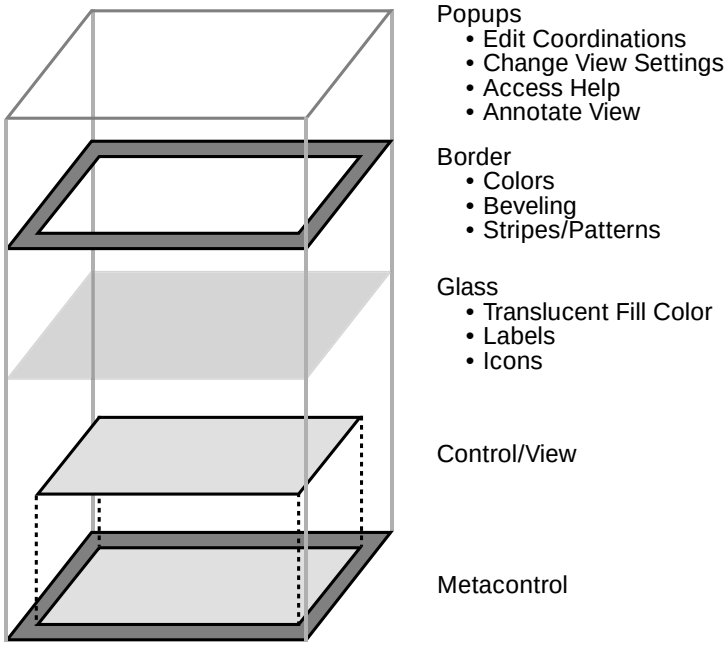



Figure 81: Model of embedding. Each control is wrapped in a metacontrol that surrounds it with a border and draws other graphics on top of it. Popups allow visualization designers to edit the control's coordinations and other characteristics.

Improvise allows for additional embedding by placing every view inside a *metacontrol* (figure 81) that: (1) manages the view's links, (2) provides a popup menu for editing coordinations, and (3) draws metavisual graphics on top of the view. The embedding inherent in every view is enhanced by drawing over it without its knowledge. This approach results in a well-integrated visualization appearance despite independent implementation of visualization and metavisualization components.

All metacontrols in a given visualization share a common projection and filter. Depending on the projection and filter, the embedding produced can be either explicit or reactive. The default projection uses a subtle button-like beveling effect to indicate whether each control is inactive, in focus, or being edited. Custom projections can be used to fill each control with a translucent color or to draw a label over it, such as in figure 82.

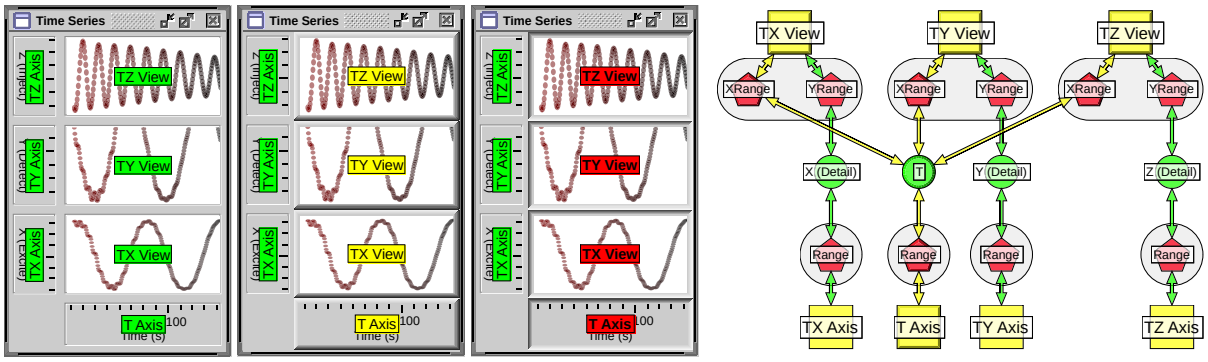



Figure 82: Example of embedding. The graph metaview shows how three time-series scatter plot views and their axis sliders are coordinated through the shared time range variable. Synchronized scrolling, vertical layout, colored labels, and beveled borders are emergent, implicit, explicit, and reactive embeddings that metavisually differentiate three interactive states (inactive, in focus, and editing, from left to right).

Unlike metaviews and lenses, which can be defined in terms of the complete metavisual data representation, metacontrols draw embeddings by projecting and filtering a single one-field data record containing a reference to the embedded view. Metacontrols are much more responsive than lenses because they draw over only the active parts of the visualization. Unlike lenses, however, they cannot draw outside the bounds of individual views, such as to draw arrows to represent coordinations.

## 7.5 Summary

Improvise is designed for visualization of relational data. By reusing the existing coordination mechanism, adding metavisualization to Improvise was a straightforward process of representing each running visualization as a collection of dynamically-maintained tabular data sets. Because this data changes frequently in response to interaction, however, achieving satisfactory interactivity during metavisualization remains a major challenge.

Implementing integrated metavisualization in Improvise using the existing CMV infrastructure provides three major benefits. First, visualization users only need to learn one coordination

approach to work with both visualizations and metavisualizations. Second, metavisualizations are reusable, and can be loaded into any Improvise visualization. Third, metavisualizations are composable. Multiple independent metavisualizations can be loaded into a visualization, each showing different aspects of coordination structure. Taken together, these qualities mean that the effort of constructing useful metavisualizations can be left to visualization system developers, while still allowing customization by visualization designers.

Although the work described here focuses on coordination structure in CMV visualizations, integrated metavisualization appears to be well-suited for other kinds of visualization, as well as for use in visualization usability and performance studies. Metavisualization also appears to have potential for application to non-visualization interfaces that possess similarly complex interactive structure. Web pages might be metavisualized by drawing on top of images and hyperlinks [76], complimenting visualizations of multiple pages [23] and entire web sites [102]. In grid views, metavisualization might be used to reveal cell activity and interdependencies, similar to the visualization of spreadsheet dynamics described by Igarashi [70].

The interactive structure of CMV visualizations can be metavisualized using existing visualization techniques. Nevertheless, there appears to be great potential for development of new techniques specific to metavisualization, particularly metaviews and lenses that better reveal the structure hidden just beneath the interface. Metavisualization holds promise to help users design, build, debug, analyze, and explore highly-coordinated visualizations.

# Chapter 8

# Visualizing Coordination in DEVise

## 8.1 Overview

DEVise [90] is an interactive editor and browser for visualizing large relational data sets using multiple coordinated scatter plots that have customizable visual encodings. As one of the earliest systems to support construction of coordinated multiple view visualizations, DEVise is the source of much of the inspiration that went into the development of Improvise. In particular, both Live Properties and Coordinated Queries can be clearly traced back to coordination and visual encoding techniques employed in DEVise. Conversely, Improvise has been used to analyze coordination structure and screen layout in DEVise visualizations. The process has been a self-reinforcing one in which better understanding of applications of coordination led to improved coordination models, which led to wider-ranging applications of coordination. This chapter recounts this process in the form of a case study that considers:

- the coordination and visual abstraction models implemented in DEVise,
- the process of building and browsing DEVise visualizations,
- pre-Improvise research efforts to understand coordination in DEVise,
- application of two Improvise visualizations to exploration of DEVise visualizations, and
- how exploration of DEVise linking structure influenced the design of Improvise.

## 8.2 DEVise

In DEVise, visualizations consist of scatter plots that act as visual queries on data. Each view shows the rendered records of a single table filtered by the virtual bounds of the view. Figure 83 shows a typical DEVise visualization.

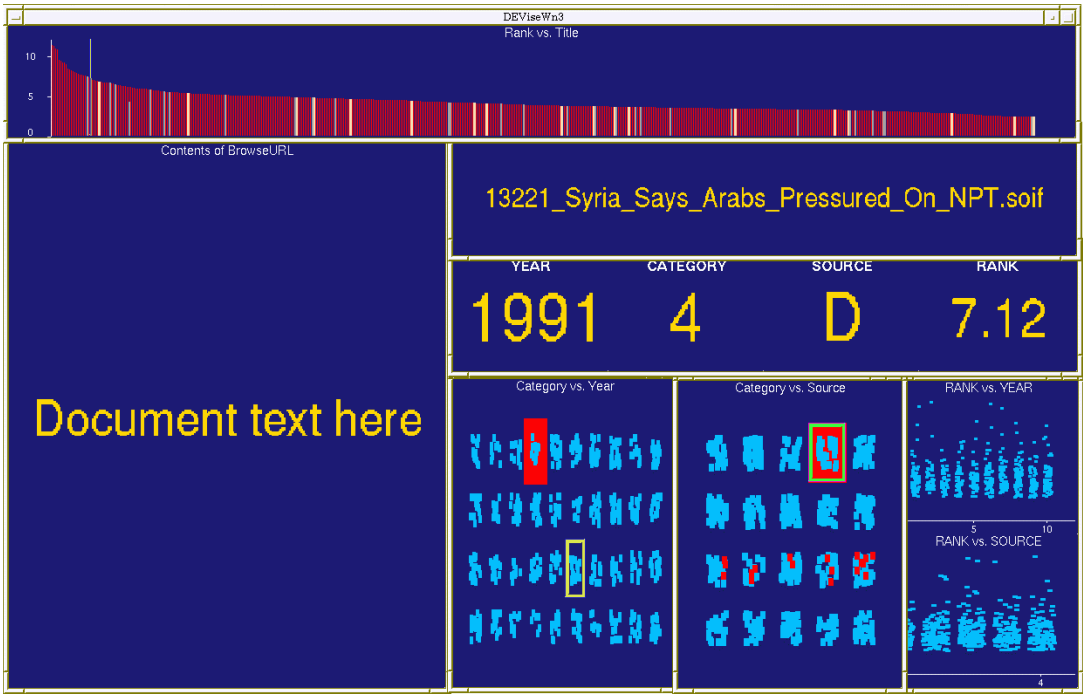



Figure 83: DEVise visualization of ranked news articles (actual article text omitted).

DEVise has three kinds of coordination: *visual links*, *record links*, and *cursors*. A visual link constrains two views to render identical `X` and/or `Y` ranges. A record link renders in a destination view only those records that are visible in a source view. A cursor is a selection box in a view that has the same `X` and `Y` ranges as some other view. (Another kind of DEVise primitive, *piles*, are visual links in which the linked views happen to be stacked on the screen.)

Building visualizations in DEVise is a three-step process. First, the user creates views. Second, the user defines *mappings* that project data attributes into graphical attributes. Each view uses a mapping to render the shapes within it. Third, the user links views and lays them

out in order to create visual and logical relationships between them. These three steps may be repeated and interleaved as desired. For instance, the user might create a view with a mapping that renders a bar chart of attribute `Y` vs `X` using attribute `Z` as the bar color. The user might then create a second view with a mapping that renders a bar chart of attribute `W` vs `X` in the same color scheme. Finally, the user could establish a horizontal-only visual link between the two views so that they always show the same range of `X` values.

Once a DEVise visualization has been built, the user can browse the data interactively by panning and zooming views. However, the user must understand how the visualization works in terms of coordination in order to analyze patterns and relationships in the data being displayed. Although DEVise links exhibit an elegant combination of simplicity and flexibility, they are hard to deduce from the appearance of a visualization alone. Learning (and relearning) to interact with a DEVise visualization often requires trial-and-error, forcing the user to construct a mental model of the linking structure one link at a time. Cyclic chains of links occur frequently, making this task even harder.

The motivation for visualizing the coordination structure of DEVise visualizations is to provide a clear answer to a simple question: "how does it work?" It is vital to convey coordination semantics to the user as completely and unambiguously as possible, during the visualization design and construction stages as well as the exploration and analysis stages that follow.

## 8.3 Early Metavisualization Efforts

### 8.3.1 DEVise Layout Manager

The DEVise Layout Manager [87] facilitates design and construction of DEVise visualizations by providing a visual mockup for the creation, layout, and linking of views. The screen layout

of views in DEVise is limited to a two-level window hierarchy in which unconstrained top-level windows contain views in an NxM array. In the Layout Manager, views may be unconstrained or connected by *struts* and *rivets*. Views may also contain views in an arbitrary hierarchy.

The Layout Manager uses lines and arrows to represent DEVise linking primitives. These shapes are drawn on top of the windows and views in the visual mockup. For instance, when two views are visually linked on X (that is, they share a range of values on the X axis), a bidirectional yellow arrow indicates the linkage. Figure 84 shows the layout of a DEVise visualization with overlaid link arrows.

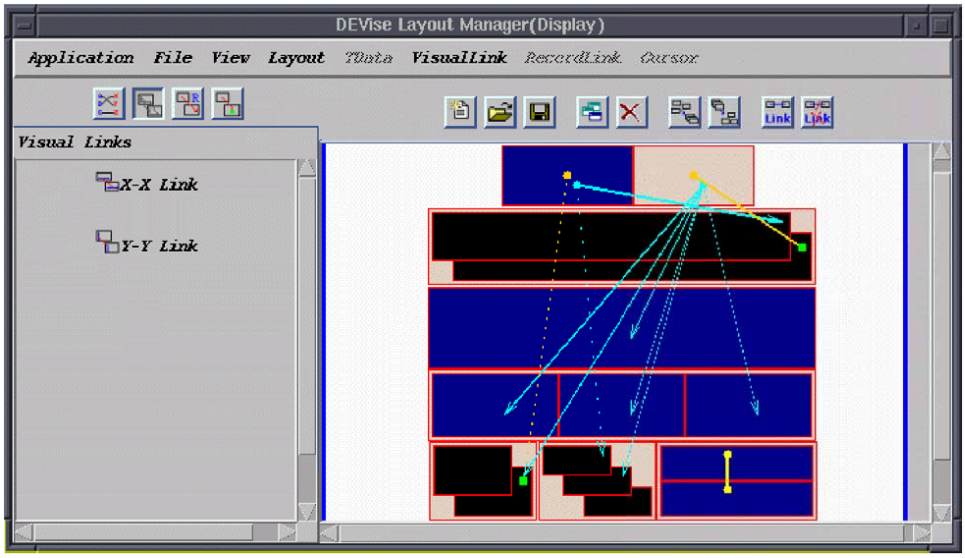



Figure 84: The DEVise Layout Manager, showing an approximation of the screen layout of the visualization in figure 83 with overlaid arrows representing visual links and piles.

As a tool for metavisualization of DEVise visualizations, the Layout Manager has shortcomings. For all but the simplest DEVise visualizations, the overlaid graphics obscure each other and the layout beneath. Moreover, because the two-dimensional location of views constrains how and where overlays can be drawn, overlays cannot depict clearly multidimensional structure typical of DEVise visualizations. These problems are somewhat mitigated by the ability to select which types of links are drawn as overlaid arrows at any given time.

Despite these problems, working with the Layout Manager produced a critical insight: displays of view coordination can be separated from displays of view layout. Whereas visualizations of *physical layout* display on screen spatial relationships between views, visualizations of *logical layout* display coordination structure in ways that do not depend on the on screen spatial relationships between views.

### 8.3.2 Logical Layout Editor

The Logical Layout Editor was created for the purpose of experimenting with logical layout techniques with the goal of implementing a flexible tool for the visual analysis of coordination structure in DEVise visualizations. As such, the visual representation of DEVise coordination graphs changed over multiple successive versions of the Logical Layout Editor.

The first version uses nodes for views and edges for visual links, record links, and cursors. Nodes are drawn as circles and edges are drawn as arrows. In this representation, each set of visually linked views appears as a fully connected subgraph of nodes. Similar graph structures emerge from the semantics of record links and cursors: two-layer trees for record linked views, directed pairs for views connected by cursors. The second version adds *packs* as a way to visually cluster nodes in such graph structures. Packs are drawn as balloons—enlarged, filled convex hulls—around the nodes they contain. Around groups of nodes that represent visually linked views, packs reveal group containment and overlap better than closed sets of pairwise edges, especially for groups with more than four views in which edges are forced to cross.

Seeing logical layouts drawn using packs instead of arrows motivated the third version of the editor (figure 85), in which nodes represent all DEVise primitives (views, visual links, record links, cursors) and edges indicate dependencies between primitives rather than the primitives themselves. The graph substructures that result are strikingly different from those in

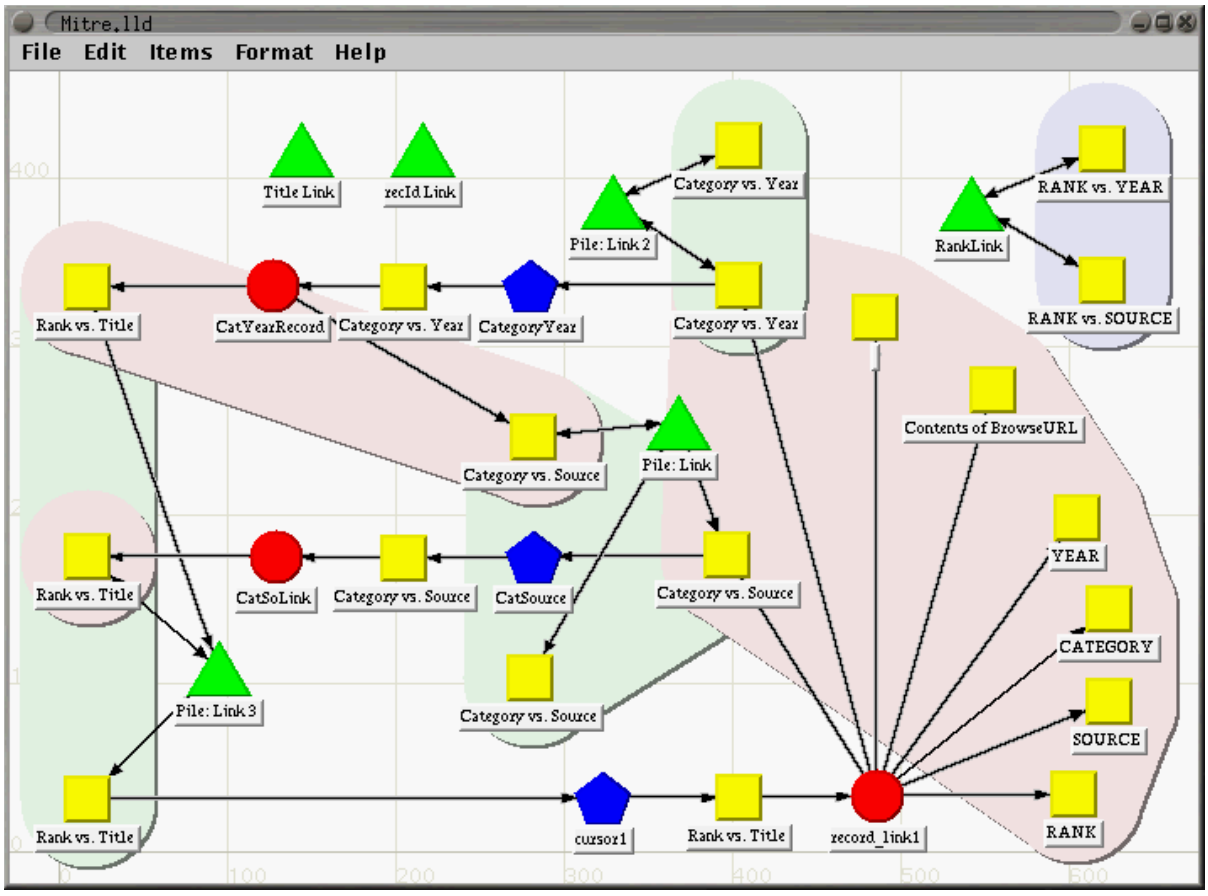



Figure 85: Logical layout graph of the visualization in figure 83. Yellow squares represent views. Green triangles, red circles, and blue pentagons represent visual links, record links, and cursors, respectively. Groups of linked nodes are contained in packs drawn in corresponding colors. (The blue pack at top right represents views visually linked on X only.)

earlier versions. A set of visually linked views appears as a set of view nodes radiating from a central visual link node. A set of record linked views has a similar appearance, but with one view node connected toward the record link node and the remaining view nodes connected away. A cursor appears as a

$$View1 \rightarrow Cursor \rightarrow View2$$

triple. The graph also reveals the presence of a fourth frequent structure, a *record cursor*, having form

$$View1 \rightarrow Cursor \rightarrow View2 \rightarrow RecordLink \rightarrow View3$$

In this structure, $View2$ is an DEVise implementation artifact that must be hidden off screen to avoid visual distraction.

Logical layout can successfully depict the linking structure of even the most complicated DEVise visualizations. For instance, the visualization in figure 83 is difficult to comprehend even with assistance from an experienced user. Its coordination structure contains cyclic dependencies that are hard to deduce from seeing physical layout but are obvious in a logical layout graph. The graph also readily reveals all four kinds of coordination substructure as well as the resulting dynamics of coordination. The directedness of edges representing structural dependencies depict how manipulation of the spatial extent of views or cursors leads to visual changes in the spatial extent and data contents of other views.

Logical layout is a useful way to show how to manipulate a DEVise visualization by showing how its views collectively respond to interaction. However, the various versions of logical layout are merely hard-coded special cases of graph visualization. The first step in building a more flexible tool for exploratory metavisualization of DEVise visualizations is to recreate the logical layout graph using the general-purpose graph view available in Improvise. The second step is to coordinate the graph view with other views so as to reveal additional relationships between DEVise primitives.

## 8.4 Metavisualization of DEVise Visualizations in Improvise

### 8.4.1 TGraph

#### Interface

The *tgraph* Improvise visualization, shown in figure 86, statically metavisualizes DEVise visualizations using a graph and a table. The graph displays the coordination structure (logical

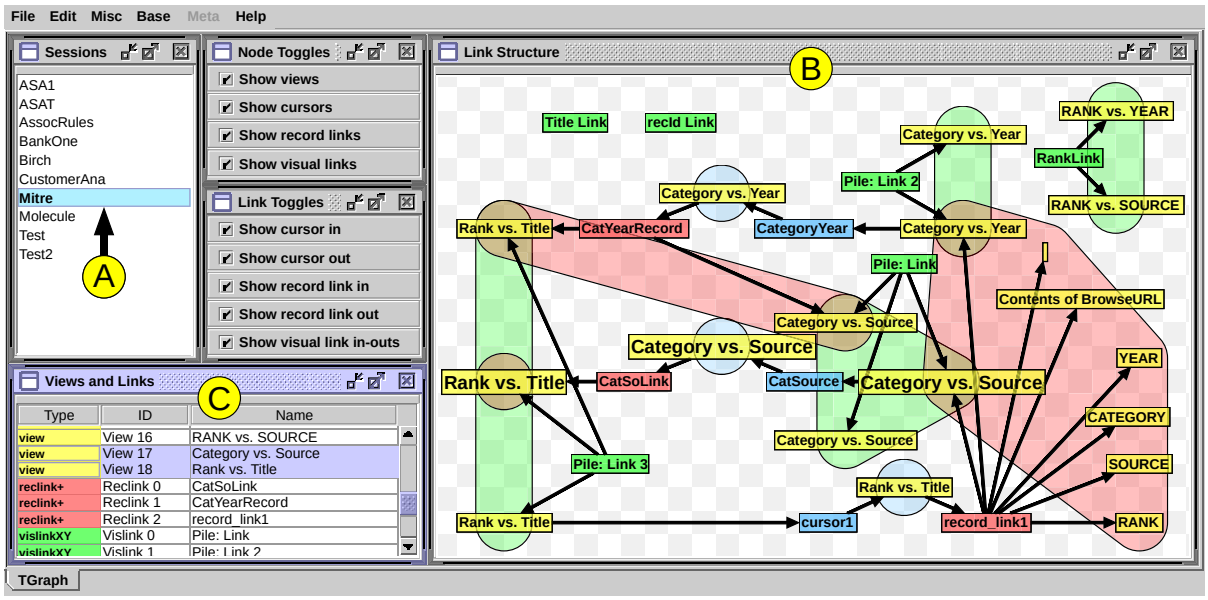



Figure 86: The *tgraph* Improvise visualization (see also appendix A.9). In a list of available visualizations, selecting the DEVise visualization from figure 83 (A) loads its coordination structure into a graph of relationships (B) and a list of objects (C).

layout) of the visualization using nodes to represent DEVise views and links, and packs to surround views that are related to each other through common links. To provide perceptual coherence, views, visual links, record links, and cursors are uniformly colored throughout the visualization using yellow, green, red, and blue, respectively.

The graph and table views share a lexical filter variable that filters which types of DEVise primitives are shown. The filter is defined using four boolean variables, which the user can toggle on and off using checkbox controls. In the graph view, the lexical filter variable is bound to the `node filter` property. The `link filter` property is bound to a second lexical filter variable, defined in terms of five boolean variables that determine which types of dependencies between DEVise primitives are visible as edges in the graph view. In this way, the user can rapidly ascertain the presence of and relationships between different coordination substructures. In figure 87, for example, the graph has been filtered to show three record cursors by toggling the appropriate checkboxes.

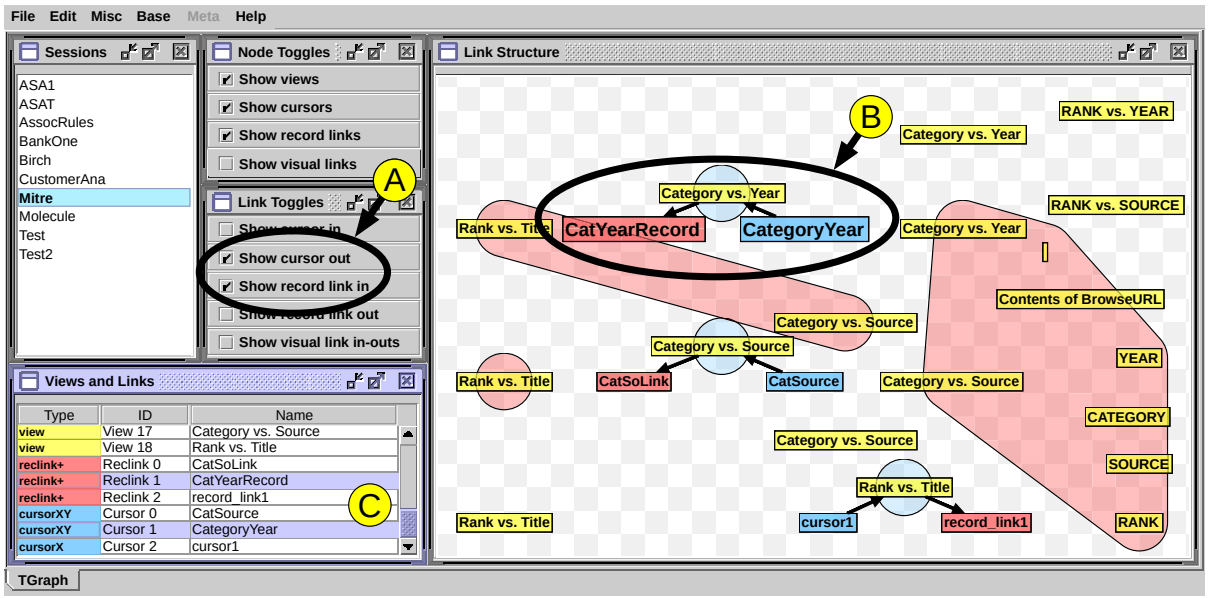



Figure 87: Category filtering in the *tgraph* visualization. Checkboxes (A) toggle boolean variables used to filter objects and relationships in the graph (B) and list (C) views.

## Data

The *tgraph* visualization accesses data using a "session dump" feature in DEVise that writes a node-and-edge graph representation of the DEVise visualization's structure to a file in a tabular format in which records are of the schema shown in table 10.

| *type* | *name* |
|---|---|
| `string` | `ID` |
| `string` | `Name` |
| `string` | `Type` |
| `int` | `X` |
| `int` | `Y` |

Table 10: Schema of records that describe DEVise objects and relationships between them.

Dumped files contain records for dependencies between DEVise primitives as well as records for the primitives themselves. Pairwise dependencies between a view and another primitive—a visual link, record link, or cursor—are encoded into the fields of records as

$$\{LinkID, srcPrimitiveID, dstPrimitiveID, 0, 0\}$$

For example, `View17` is the source view of `Reclink0` in the record `{Link0, View17, Reclink0, 0, 0}`. Dependencies between groups of views—such as destination views in record links and transitively closed sets of views in visual links—are encoded as

$$\{PackID, PackName, ViewID, 0, 0\}$$

For records that represent the primitives themselves, the `X` and `Y` fields are used to determine the initial location of each primitive's node in the graph view.

Figure 88 shows how the *tgraph* visualization accesses and processes data. A master data set that lists available DEVise session dumps is generated from a directory in the local file system. Selecting an item from the master list loads a particular session dump file from that directory. Grouping queries classify records from the file into node, edge, and pack categories that are used by the graph view to build graph topology.

### Coordination and Visual Abstraction

Figure 89 shows the lexical expressions used by the *tgraph* visualization to determine the structure, visibility, and appearance of nodes and edges in the graph view. Graph structure is determined by naming edge endpoints and pack members using unique node identifiers. Visibility of graph elements depends on the state of relevant checkboxes via boolean variables. Node and pack glyphs are colored according to the type of DEVise primitive they represent. Edge glyphs are drawn as black arrows, and do not depend on data attributes to determine their appearance.

Figure 90 shows how the graph view depends on the session list view and checkbox controls via paths through the *tgraph* visualization's coordinated query graph. Selecting a particular DEVise visualization in the session list causes the graph to apply the structure projection expressions to the corresponding set of node, edge, and pack data sets (red paths). Toggling the boolean value of the `Visual Link Flag` variable (using the corresponding checkbox)

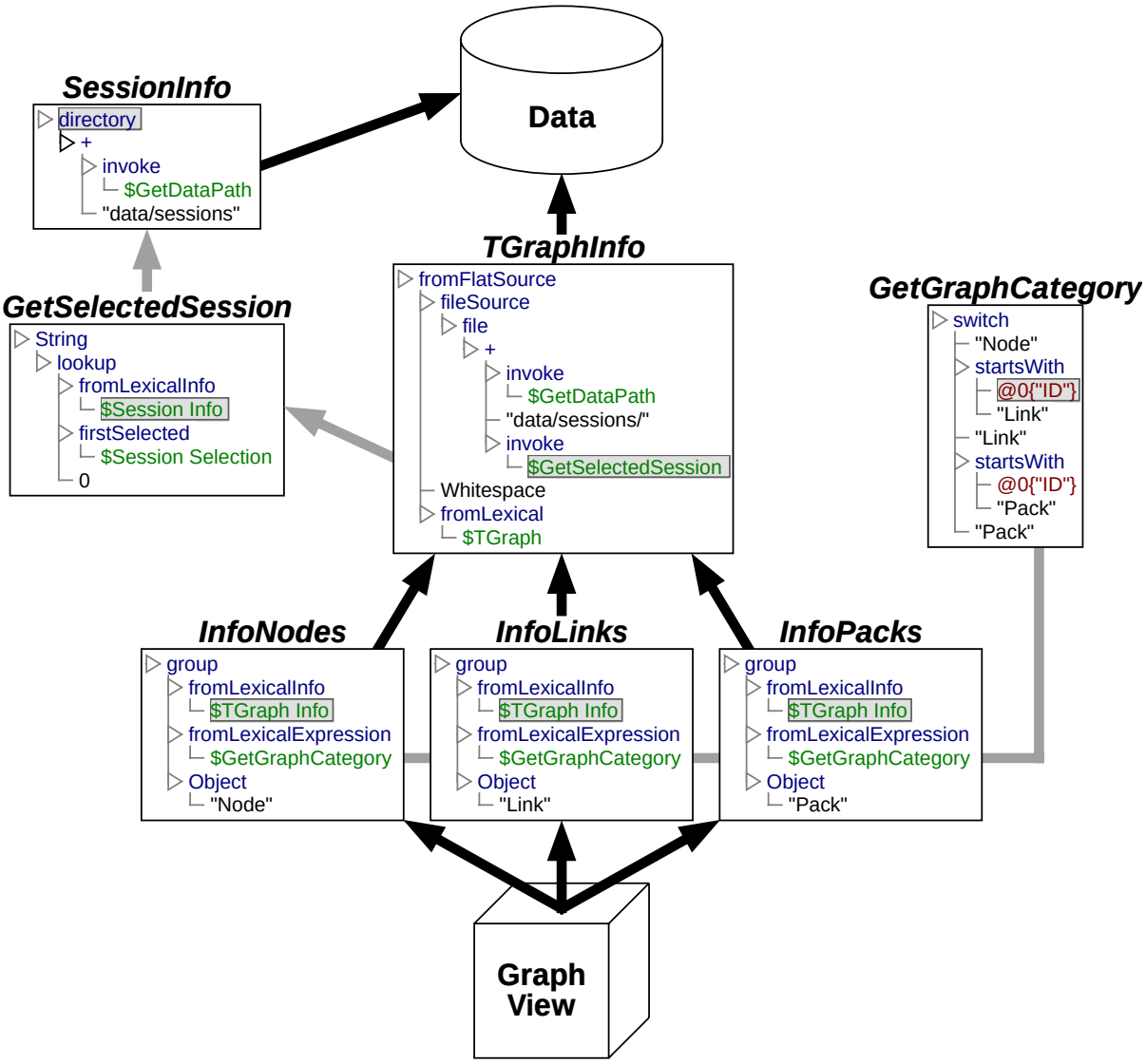



Figure 88: Query processing in the *tgraph* visualization.

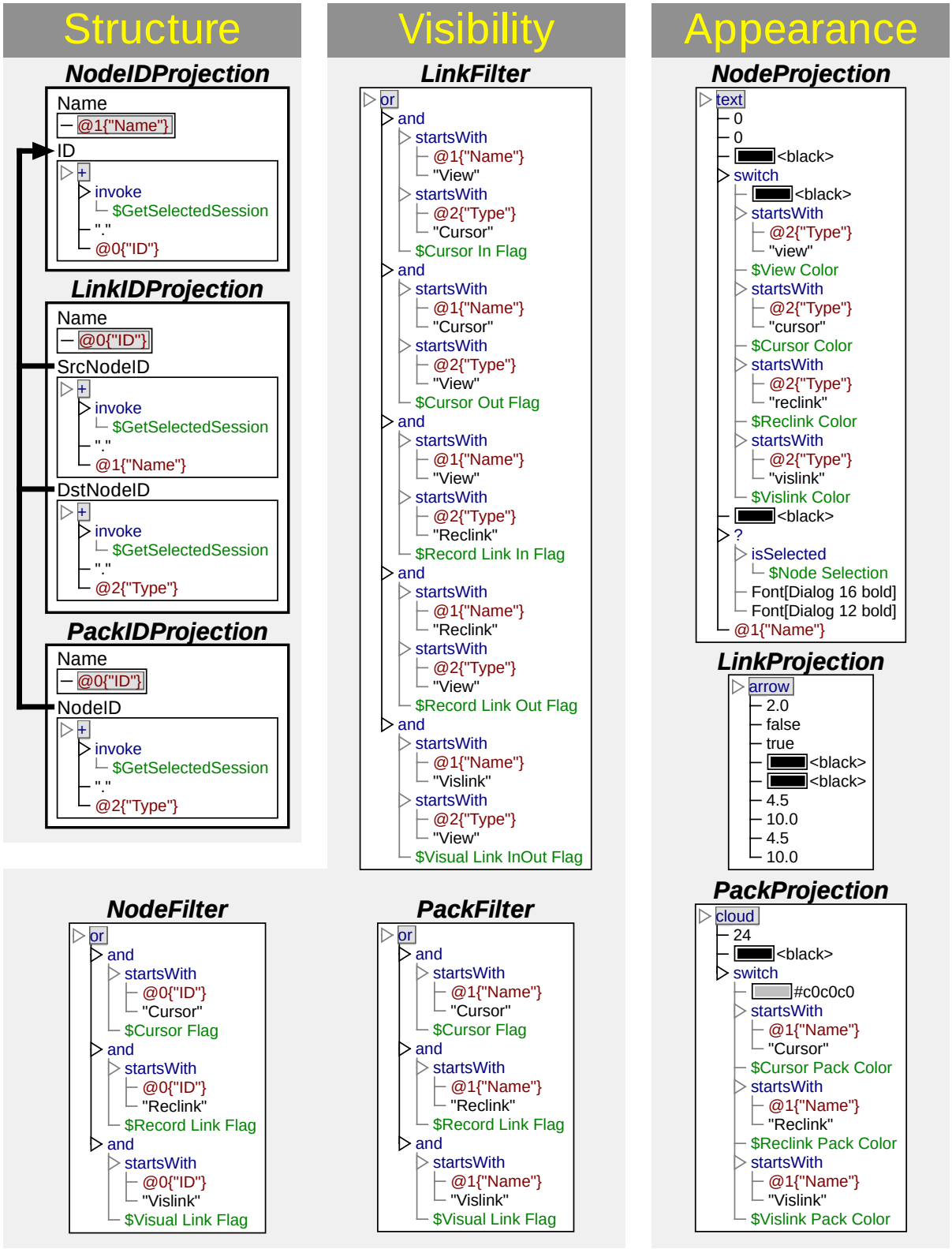



Figure 89: Visual abstraction in the *tgraph* visualization.

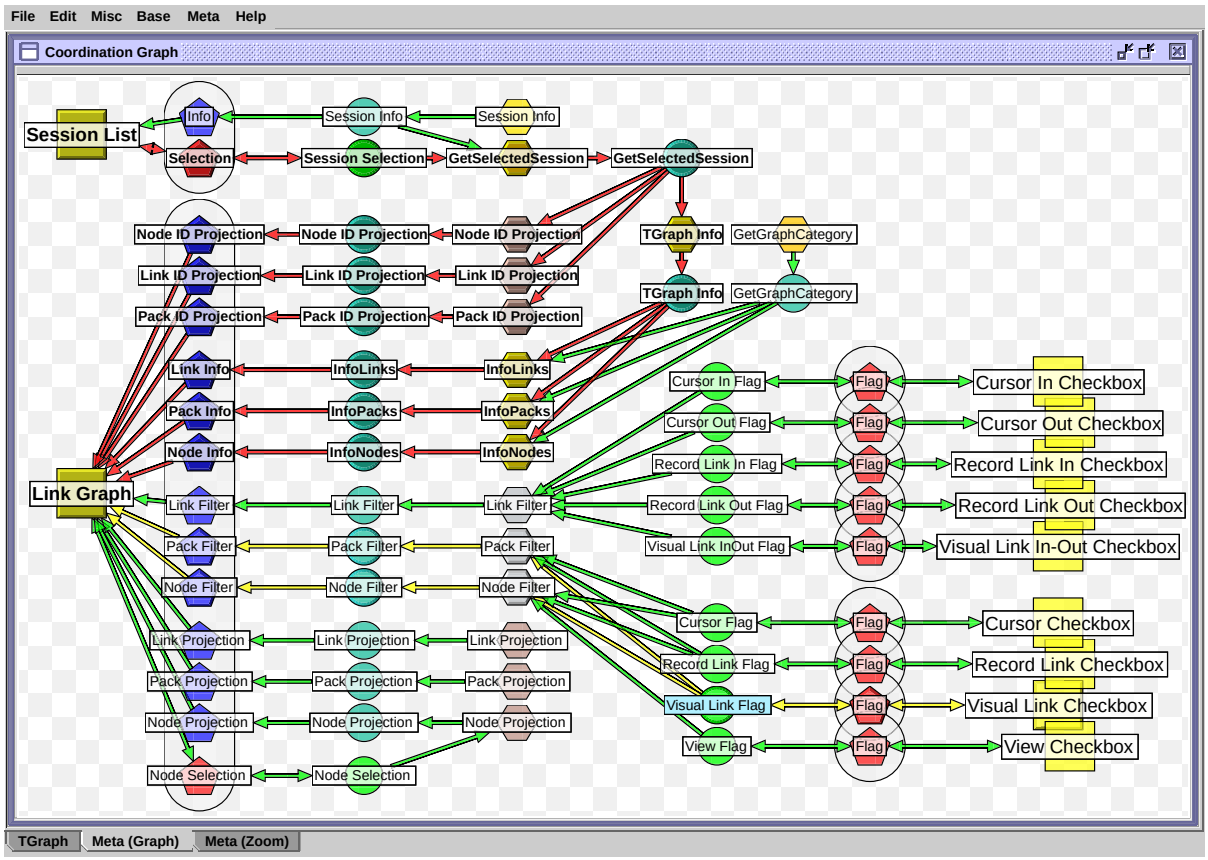



Figure 90: Coordinated query graph for the *tgraph* visualization as it appears in figure 87.

affects filtering of edges and packs in the graph view (yellow paths).

Addition of the node and edge visibility toggles, including layout of the checkboxes and modification of the lexical filters, took only a few minutes. This ability to interactively modify a metavisualization to explore the structure of visualizations using Coordinated Queries is a substantial improvement over logical layouts and other metavisualizations that are custom-programmed for a particular visualization system.

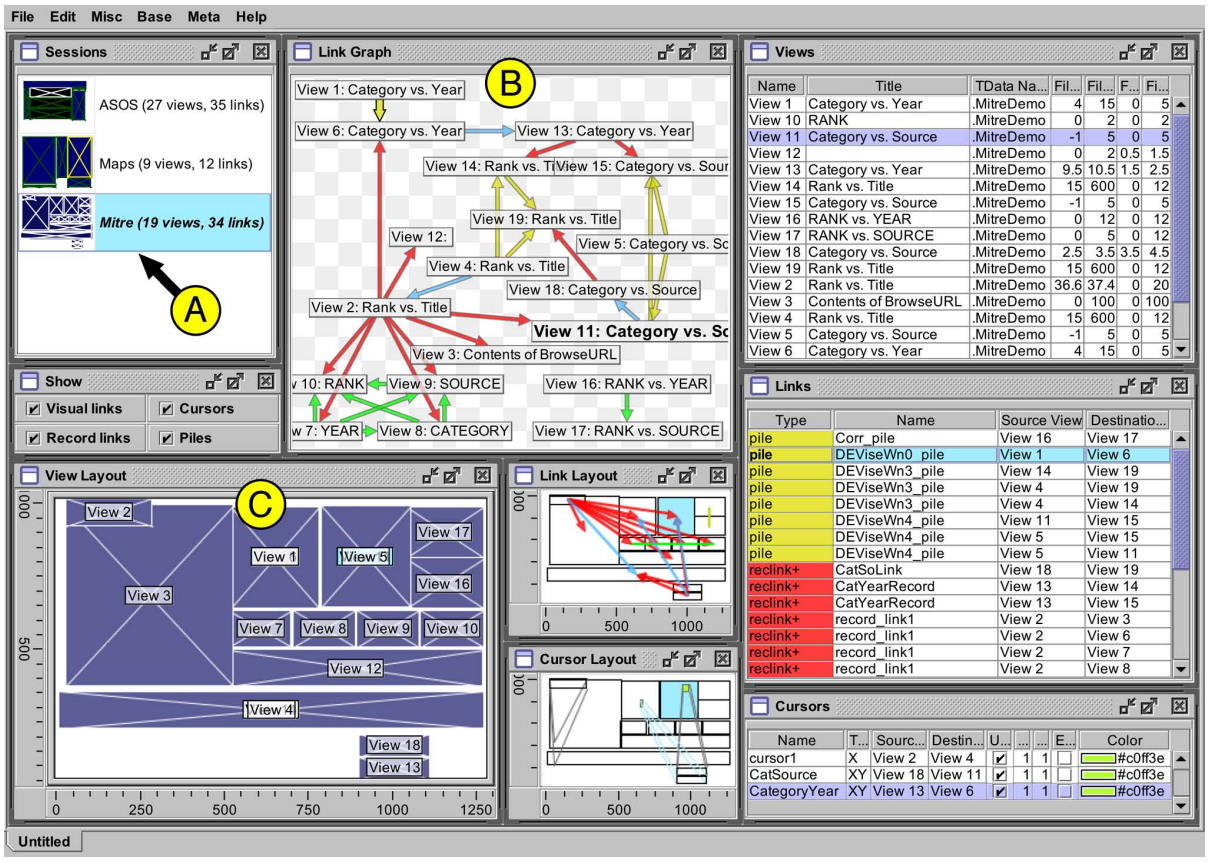



Figure 91: The *devise-mv* visualization (see also appendix A.9). In a list of available visualizations (shown in miniature), selecting the DEVise visualization from figure 83 (A) loads its coordination structure into a graph (B) and its screen layout into a scatter plot (C).

## 8.4.2 DEVise-MV

### Interface

The *devise-mv* Improvise visualization, shown in figure 91, statically metavisualizes DEVise visualizations using a graph, scatter plot, and several tables. The graph displays the coordination structure of the visualization using nodes to represent DEVise views and edges to represent links. The scatter plot draws a miniature version of DEVise screen layout using position information in the records that describe DEVise views. To provide perceptual coherence, DEVise

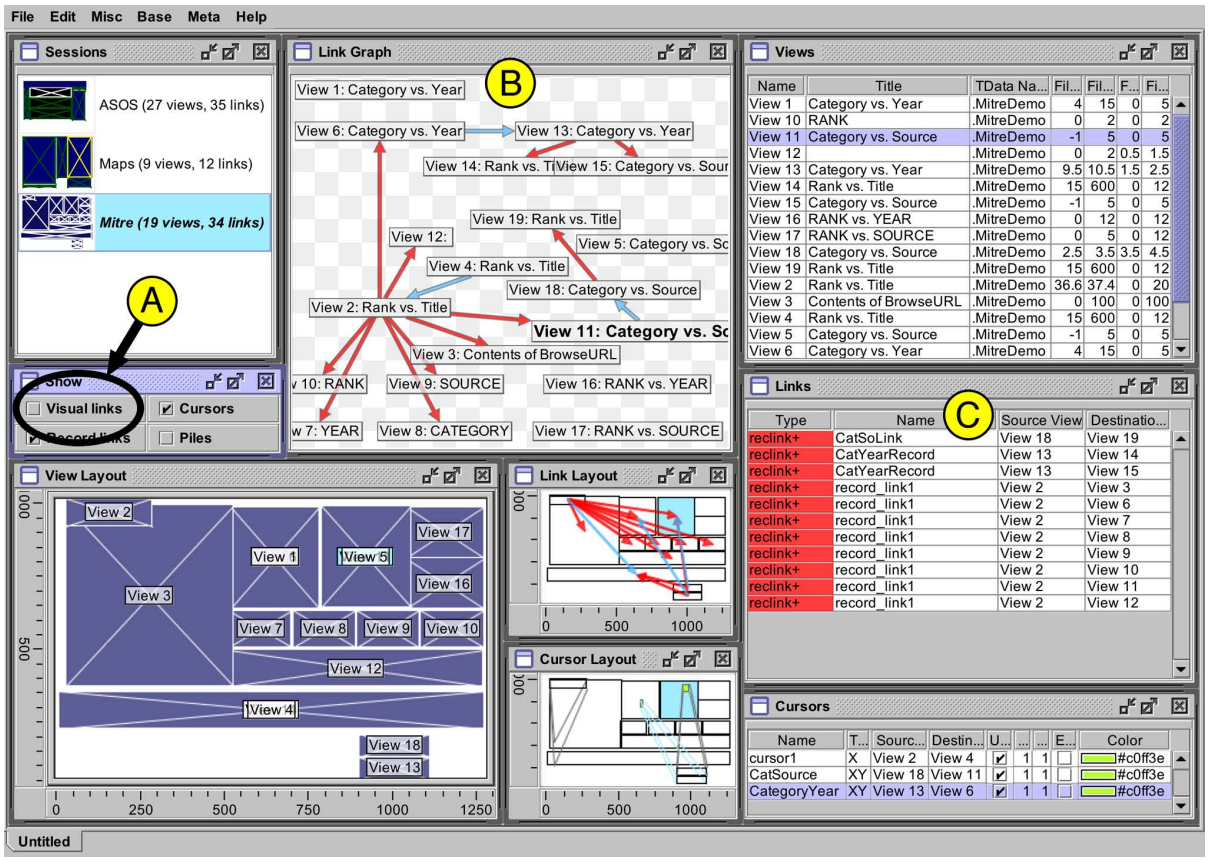



Figure 92: Category filtering in the *devise-mv* visualization. Checkboxes (A) toggle boolean variables that affect the visibility of different kinds of DEVise links that are represented as edges in the graph (B) and rows in the links table (C).

primitives are uniformly colored throughout the visualization similarly to the *tgraph* visualization, but using yellow for piles and white for views.

The graph view and the `Links` table view are filtered using (different) lexical filters defined in terms of five boolean variables. The user can toggle these variables using checkboxes (figure 92). The graph view and the `Views` table view are further filtered to show only the DEVise views that are at least partially visible in a portal in the main scatter plot. In this way, the user can identify which links affect which parts of the screen in the visualization.

The main scatter plot portal is navigationally coupled with the link and cursor scatter plots, resulting in a three-way coordination that combines aspects of overview+detail, synchronized

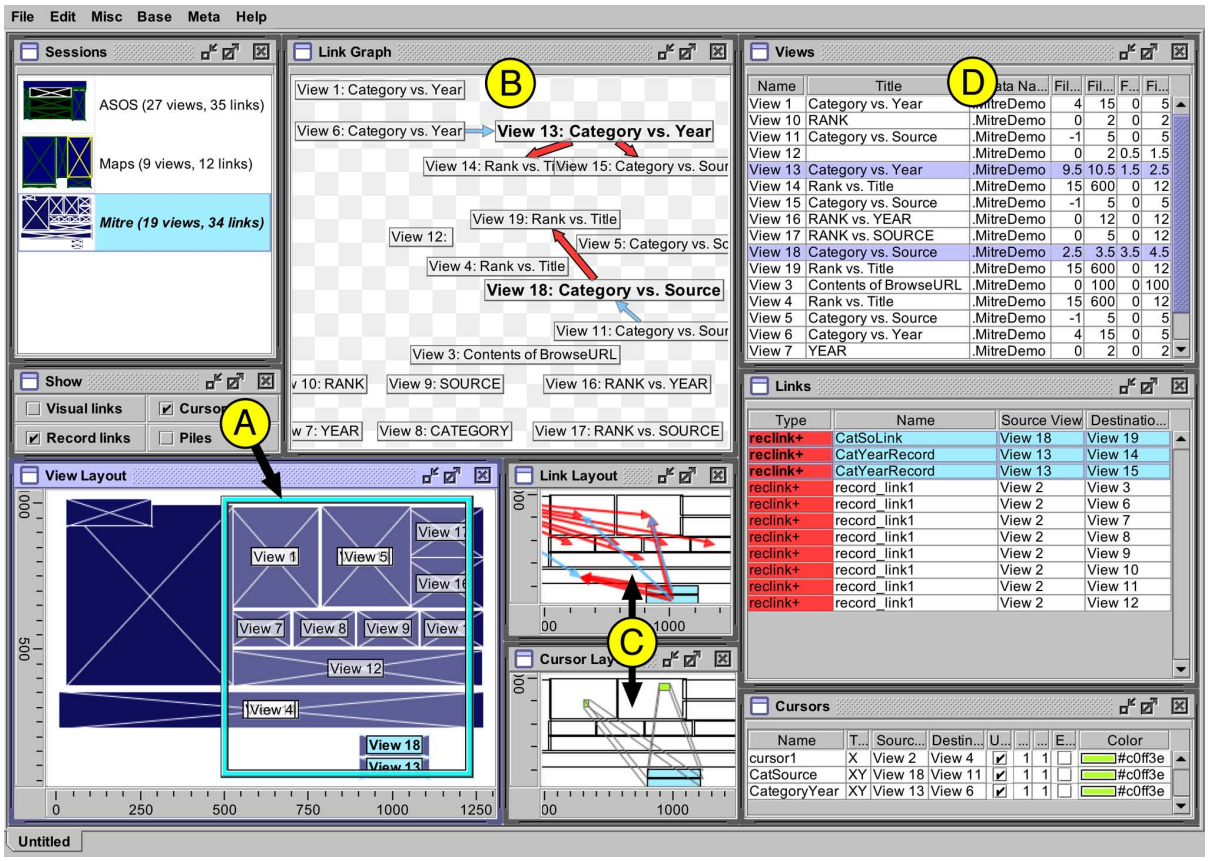



Figure 93: Spatial filtering in the *devise-mv* visualization. Moving a portal above the screen layout (A) causes the graph (B) and views table (D) to show only views inside the portal bounds. The portal is navigationally coordinated with the link and cursor scatter plots (C).

scrolling, and small multiples. In figure 93, the portal has been panned to display the right half of the visualization screen layout. The visual links and piles have been made invisible. The resulting metavisualization reveals details about extraneous views that were created to support two record cursors, including their artifactual presence just beneath other views on the screen.

## Data

The *devise-mv* visualization accesses data generated using a second version of the DEVise "session dump" feature that writes a substantially more detailed description of views, visual links, piles, record links, and cursors in a session. Although the dumped file is written in a

| *type* | *name* |
|---|---|
| string | Name |
| string | Title |
| string | Parent View |
| color | Foreground Color |
| color | Background Color |
| string | Pile |
| int | Dimensions |
| string | TData Name |
| int | X |
| int | Y |
| int | Width |
| int | Height |
| double | Filter X Lo |
| double | Filter X Hi |
| double | Filter Y Lo |
| double | Filter Y Hi |

Table 11: Schema of records that describe DEVise views.

| *type* | *name* |
|---|---|
| string | Name |
| string | Type |
| string | Source View |
| string | Destination View |
| boolean | Use Grid |
| int | Grid X |
| int | Grid Y |
| boolean | Edge Grid |
| color | Color |

Table 12: Schema of records that describe DEVise cursors.

| *type* | *name* |
|---|---|
| string | Name |
| string | Type |
| string | Source View |
| string | Destination View |

Table 13: Schema of records that describe DEVise record links, visual links, and piles.

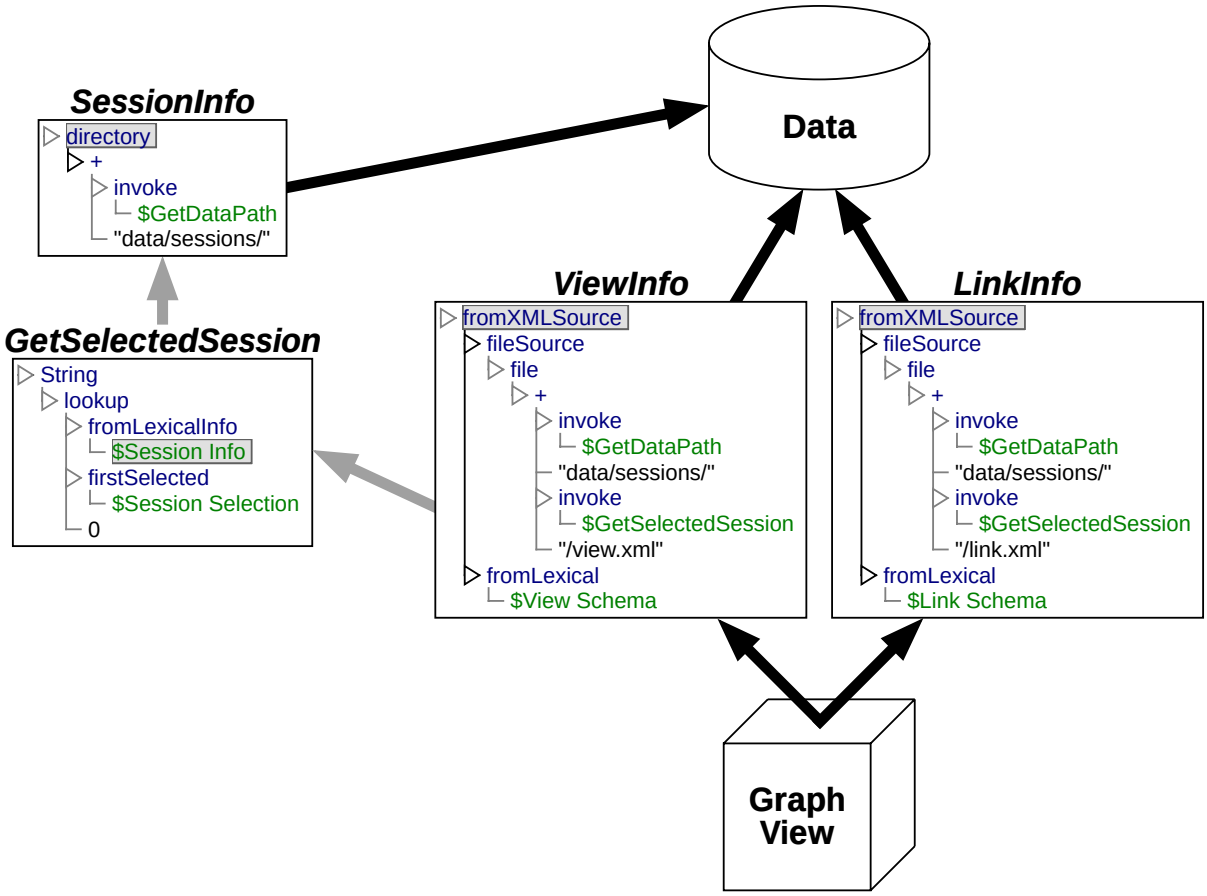



Figure 94: Query processing in the *devise-mv* visualization.

record-oriented format, each kind of DEVise primitive has a different description, necessitating variant records. Records of three different schemas, shown in tables 11–13, are written to the file in the session dump. The records of each schema are separated into two XML-formatted files, one describing views and one describing links.

Figure 94 shows how the *devise-mv* visualization accesses and processes data. A master data set that lists available DEVise session dumps is generated from a directory in the local file system. Selecting an item from the master list loads its pair of XML-formatted files from that directory. The graph view uses the resulting data sets to construct the node-and-edge topology of its graph. The same data sets are separately displayed in the view and link tables.

### Coordination and Visual Abstraction

Figure 95 shows the lexical expressions used by the *devise-mv* visualization to determine the structure, visibility, and appearance of nodes and edges in the graph view. Graph structure is determined by naming edge endpoints using unique node identifiers. The direction of each edge is determined by the type of DEVise link it represents. Node visibility depends on whether or not the views they represent are contained within the portal in the main scatter plot. Edge visibility depends on the state of relevant checkboxes via boolean variables. Each node glyph displays the name and type of the view it represents. Each edge glyph is drawn as an arrow colored according to the type of the DEVise link it represents.

Figure 96 shows how the graph view depends on the session list view, main scatter plot, and checkbox controls via paths through the *devise-mv* visualization's coordinated query graph. Selecting a particular DEVise visualization in the session list causes the graph to apply the structure projection expressions to that session's data set; it also causes the main scatter plot and portal to display the newly selected data set (red paths). Dragging and stretching the portal horizontally affects filtering of nodes in the graph view (top yellow path). Toggling the boolean value of the `Visual Link Flag` variable (using the corresponding checkbox) affects filtering of edges in the graph view (bottom yellow path).

Although the absence of explicit joins in Improvise originally precluded the kind of complex filters used in this metavisualization, index operators are a sufficient substitute. The `Links` and `Cursors` table views show only the primitives that link views visible in the scatter plot. To do this, it is necessary to filter on screen position information (in the views data set) indexed using source and destination view names (in the links and cursors data sets). A similar technique is used to draw the topmost layer in the `Links` scatter plot, by drawing links and cursors as colored arrows between the centers of the rectangles that represent views,

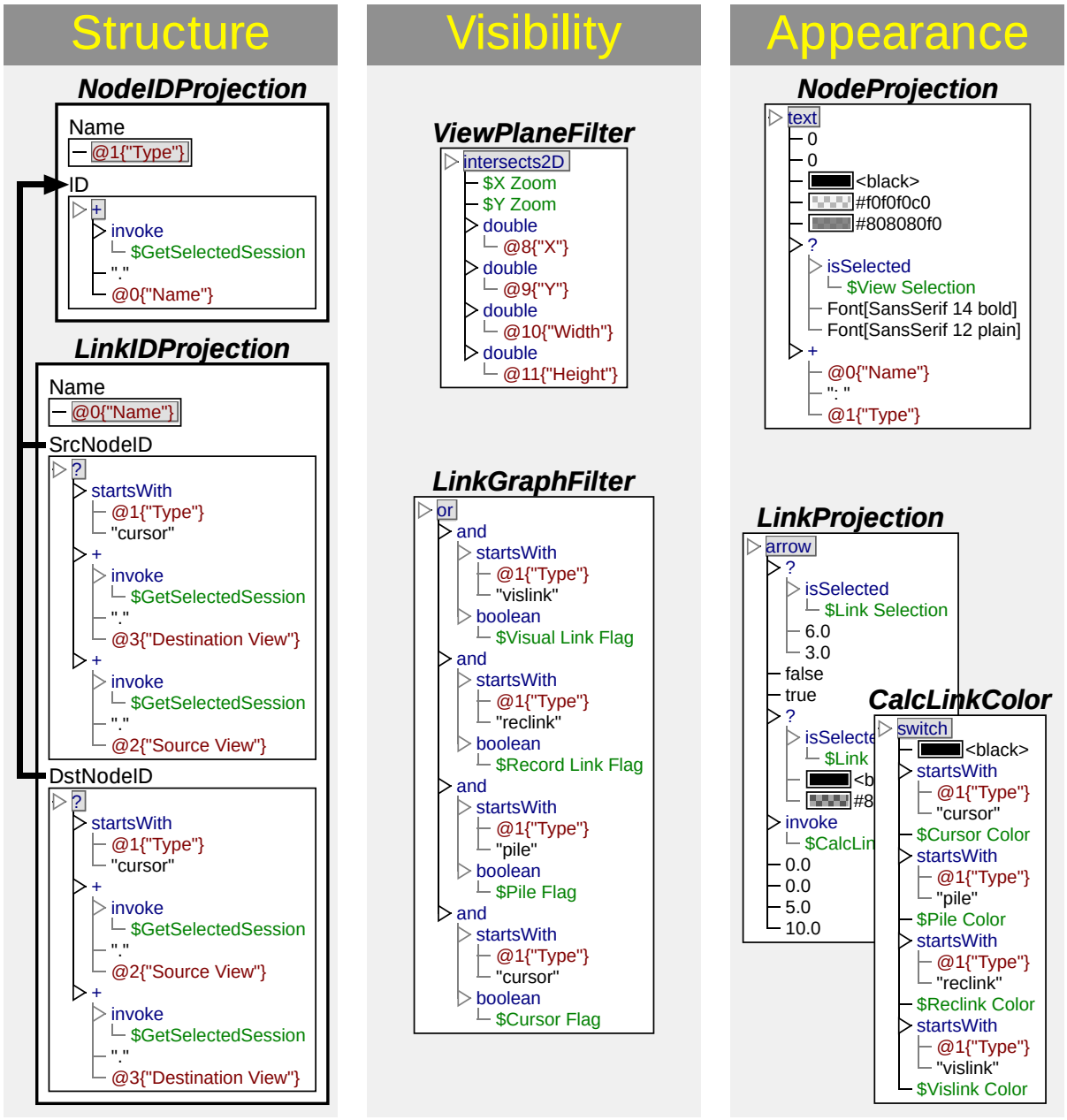



Figure 95: Visual abstraction in the *devise-mv* visualization.

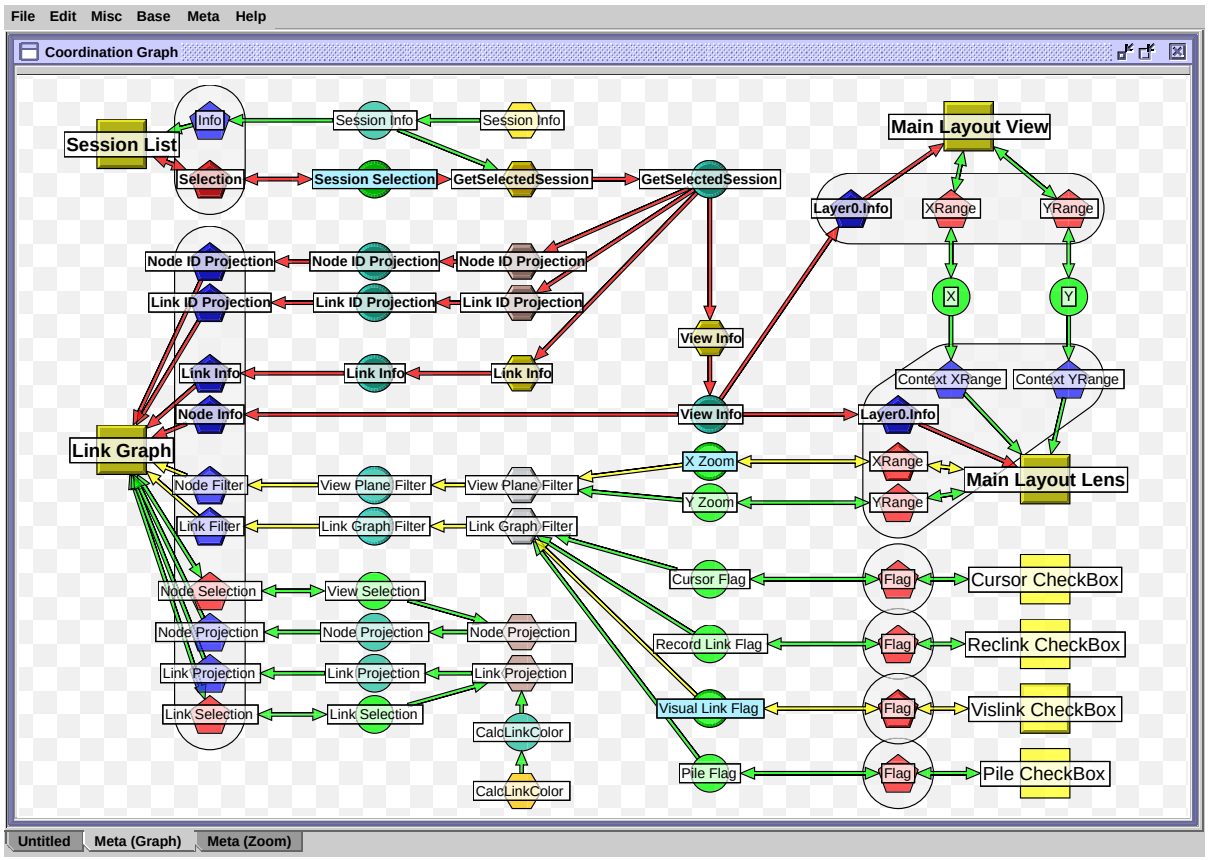



Figure 96: Coordinated query graph for the *devise-mv* visualization as it appears in figure 93.

thereby recreating the appearance of the DEVise Layout Manager.

The process of building the *tgraph* and *devise-mv* visualizations reveals a limitation in the current design of Improvise graphs: the inherent structure of accessed data sets (like the session dumps) limits the ways in which those data sets may be projected into nodes and edges in graphs, and how those nodes and edges may be filtered. This is why the *tgraph* visualization shows all DEVise primitives as nodes and dependencies between them as edges, while the *devise-mv* metavisualization shows DEVise views as nodes and other primitives as edges.

## 8.5 Evolution of DEVise Linking Structure

The earliest attempts at metavisualization involved both hand-drawn and custom-coded views of DEVise coordination structure. Despite the crudeness of these approaches, they led to the discovery that DEVise views and links can be translated into a simpler, more flexible set of coordination and visual abstraction components. These components are the basis of Live Properties and Coordinated Queries. Figure 97 shows translation stages for visual links, cursors, record links, and view mappings.

Translation of visual links into Coordinated Queries can be broken down into four stages:

1. In DEVise, visual links ($L_{12}, L_{13}, L_{23}$) form a transitive closure of symmetric connections between scatter plots ($V_1, V_2, V_3$).

2. The transitive closure can be treated as a single coordination ($L$) connecting multiple views.

3. The coordination is a special case of synchronized scrolling in which views always display the same rectangular region of the cartesian plane. This rectangular region can be defined as a pair of ranges on `X` and `Y` ($R_X \times R_Y$).

4. In Improvise, multiple views bind to a pair of independent range variables ($R_X, R_Y$).

Translation of visual links reveals that synchronized scrolling can be treated as two distinct coordinations involving separable navigation in orthogonal (horizontal and vertical) dimensions. This separation of scrolling dimensions increases coordination flexibility by allowing views to synchronize horizontally, vertically, or in dimensionally flipped fashion, as well as in the usual two-dimensional manner.

Translation of cursors into Coordinated Queries can be broken down into five stages:

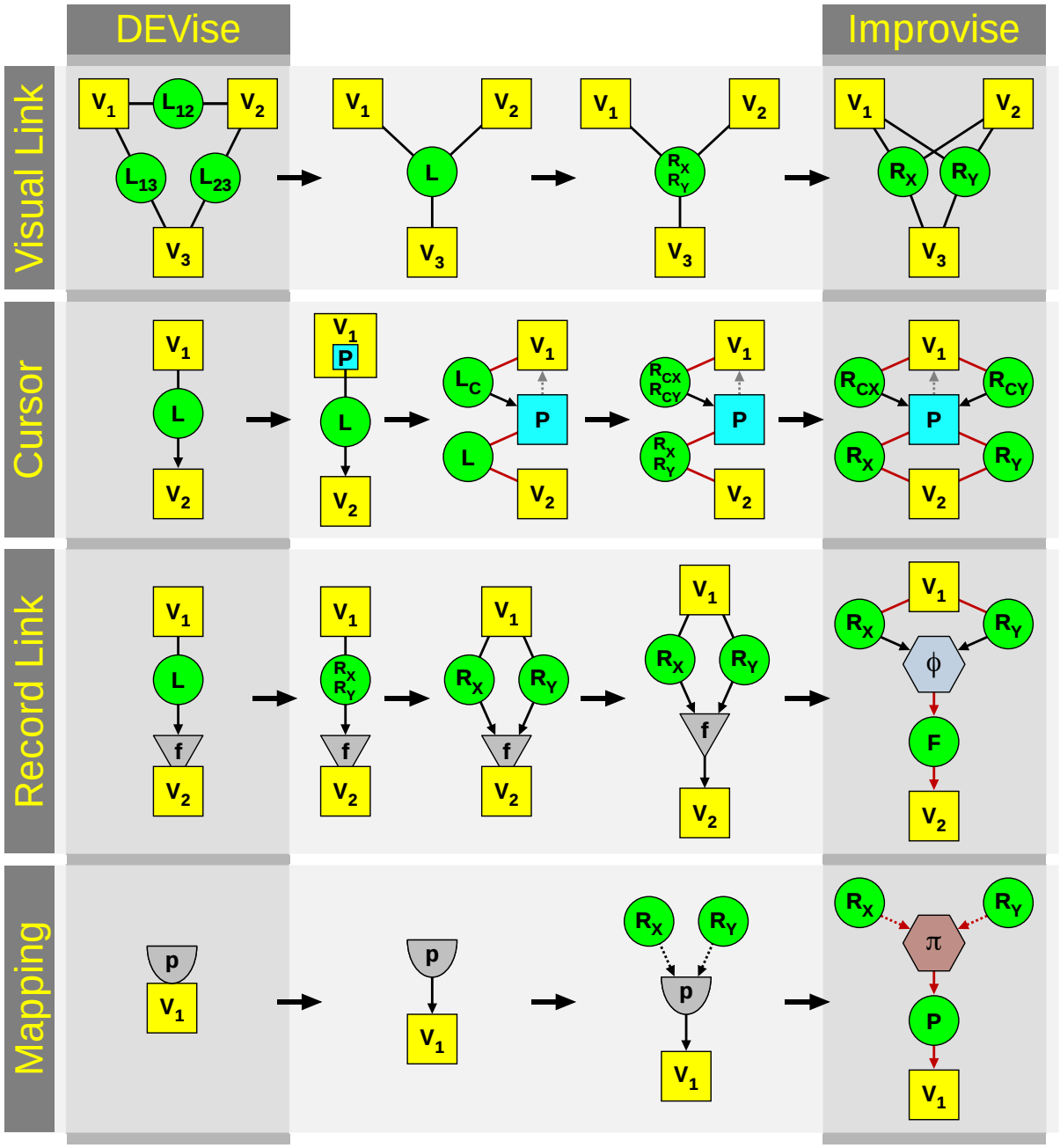



Figure 97: Translating DEVise coordination and visual abstraction primitives into Improvise.

1. In DEVise, a cursor ($L$) connects the full extent of one view ($V_2$) to an arbitrary rectangular subregion of another view ($V_1$).

2. The connection involves two relationships: (1) containment of a movable portal ($P$) inside $V_1$, and (2) the equivalent of a visual link between $P$ and $V_2$.

3. The combination of relationships is a special case of overview+detail in which the detail view ($V_2$) coordinates with a control nested inside the spatial context of the overview ($V_1$) rather than with the overview itself.

4. The rectangular overview and detail regions can be defined as two pairs of ranges on `X` and `Y` ($R_{CX} \times R_{CY}$ and $R_X \times R_Y$, respectively).

5. In Improvise, a rectangular portal ($P$) control is nested in a scatter plot ($V_1$). Two contextual range properties bind to the overview's bound range variables ($R_{CX}, R_{CY}$). Two additional range properties bind to the range variables ($R_X, R_Y$) shared with the detail view ($V_2$).

Translation of cursors reveals that overview+detail can be treated as two distinct navigation coordinations: (1) between a parent view and a nested child control, and (2) between the child control and other views. Moreover, doing so increases coordination flexibility by enabling several useful variations on overview+detail (as described in section 6.2.4).

Translation of record links into Coordinated Queries can be broken down into four stages:

1. In DEVise, record links ($L$) filter ($f$) the contents of one view ($V_2$) to show only those records visible in another view ($V_1$).

2. The rectangular extent of $V_1$ can be defined as a pair of ranges on `X` and `Y` ($R_X \times R_Y$).

3. The filter can be expressed as a function of two range variables ($R_X, R_Y$). The filter passes only those records for which $x \in R_X$ and $y \in R_Y$.

4. The filter can be separated from its view. As a result, filters can be reused by multiple views.

5. In Improvise, views filter records using lexical filter expressions ($\phi$) that can be defined in terms of the ranges bound to other views.

Translation of mappings into Coordinated Queries can be broken down into four stages:

1. In DEVise, views ($V_1$) apply mapping projections ($p$) to visually encode raw data attributes into graphical attributes of shapes.

2. The projection can be separated from its view. As a result, projections can be reused by multiple views.

3. DEVise uses implicit *visual filtering* to render only projected shapes that fall inside a view's rectangular bounds. The rectangular extent of $V_1$ can be defined as a pair of ranges on `X` and `Y` ($R_X \times R_Y$).

4. In Improvise, views render records using lexical projection expressions ($\pi$). Although Improvise scatter plots do not automatically filter unseen glyphs, explicit visual filtering can be done by defining the projection expression so as to make invisible all glyphs that fall outside the view's own ranges.

Translation of record links and mappings reveals two things. First, the manner in which scatter plots filter and visually encode data can be specified externally, regardless of whether of not data processing occurs internally. Second, these specifications can be defined in terms of ranges that determine the visible extent of those scatter plots. These revelations have several

interrelated implications that motivated the design choices behind Live Properties and Coordinated Queries:

- Filters and visual encodings are merely special cases of language fragments that specify data processing operations.
- Data and data processing operations can be shared by multiple views.
- As sharable objects, data and data processing operations are themselves interactive parameters that may depend on other interactive parameters (including other data processing operations).
- Interactive parameters can be of any type, not just ranges.
- Coordination can involve multiple dimensions of different types.
- Coordination can occur between views of any type, not just scatter plots.

Together, these observations suggest that coordination models can be based on *symmetric sharing of interactive parameters between multiple views* rather than on *unidirectional and bidirectional interactive dependencies between pairs of views*. Moreover, interaction in multiple coordinated views can be thought of as *transitions between graphical states in a high-dimensional space of interactive parameters*, thereby reinforcing Chi's contention that Data State Models and Data Flow Models are functionally equivalent ways of constructing coordinated multiple view visualizations [29].

Using Improvise to metavisualize DEVise coordination structure has resulted in an enormous improvement in speed, usability, and usefulness over earlier approaches. Whereas the original custom-coded metavisualizations took weeks to implement, building the *tgraph* and *devise-mv* visualizations each took hours. Moreover, each of these visualizations provides a

complete exploratory interface using multiple coordinated views. These views were added quickly in the Improvise builder interface, but would have been costly in time and effort to implement as custom code. Most importantly, both visualizations have improved understanding of interaction in DEVise by enabling rapid exploration of coordination structure with the goal of identifying and classifying key substructures.

Visual links, cursors, and record links can all be reproduced using the 1-D and 2-D synchronized scrolling, overview+detail, and navigation-dependent filtering coordination patterns described in chapter 6. Because DEVise coordination structures can be translated easily into Improvise coordination structures, it would be straightforward to recreate all DEVise visualizations in Improvise. The resulting visualizations might then be redesigned or extended with the goal of improving their usability and usefulness by taking advantage of the flexibility of coordination and visual abstraction in Improvise.

Conversely, some simple Improvise coordination structures could be implemented as new DEVise primitives, such as one-dimensional cursors (section 6.2.4) and dimensionally-flipped visual links (section 6.2.2). An optional interface might be added to DEVise to enable advanced users to create additional, custom coordinations using Coordinated Queries directly inside DEVise. This approach could be followed in other visualization systems as well. Similarly, Improvise might incorporate the user interfaces of other visualization systems in the form of abstraction layers that would allow visualization designers and users to interact with their preferred visualization architecture while actually manipulating Improvise views and coordinations.

## 8.6 Summary

DEVise uses a relational data model to coordinate multiple views of large data sets. Users can create, destroy, coordinate, and specify the contents of scatter plots interactively using a small number of elegant coordinations. However, reproducing common visualization constructions in DEVise frequently involves convoluted chains of linked views, some of which are undesirable artifacts that must be intentionally hidden off screen.

Two different Improvise visualizations employ standalone metavisualization techniques to support exploration of coordination and screen layout in DEVise visualizations. Both metavisualizations display multiple data sets that represent the statically captured interactive structure of several DEVise visualizations, but do so at different levels of detail. Coordination graphs reveal that all four DEVise link types can be reproduced by treating the `X` and `Y` ranges of scatter plots as shared objects or as dynamic parameters in simple query expressions. Moreover, exploration and analysis of these graphs uncovered useful variations of the four link types that were later implemented in DEVise. These discoveries suggested a more general approach to coordination that led to the design and implementation of Live Properties and Coordinated Queries in Improvise.

Coordination in DEVise consists of a few well-known patterns similar to those supported in other visualization systems. Although the details of these patterns vary from system to system, all coordinated multiview visualizations share the same general structure: a graph of views and coordinations. As such, standalone metavisualization techniques are readily applicable across a variety of multiview coordination architectures. These techniques are also a useful fallback when it is impractical to implement integrated metavisualization (chapter 7) in existing visualization systems.

# Chapter 9

# Conclusion

Improvise is a fully implemented software architecture and user interface that enables users to build and browse highly-coordinated visualizations interactively. By coupling a shared-object coordination model with a declarative visual abstraction language, users can create a potentially infinite number of useful variations on common coordinations such as synchronized scrolling, overview+detail, brushing, drill-down, and semantic zoom. Visualization developers can extend Improvise by adapting their views and query processing algorithms using a simple plug-in architecture. Visualization researchers can use Improvise as a platform for exploring patterns of coordination in visualizations that have many views.

## 9.1 Contributions

This dissertation presents the following thesis: *By combining coordination, visual abstraction, and data querying in a single coordinated query language, it is possible and practical to build visualizations with more views and richer interactive appearance and behavior than in visualization systems in which these functions are independent.* Substantiation of this thesis consists of the following six innovations:

- *Conceptual Model.* A formal model of visualization coordination based on the sharing of objects between views for purposes of navigation and selection.

- *Declarative Language.* A visual query and abstraction language for specifying how the display of relational data in views depends on interaction in other views, in terms of elements of the conceptual model.

- *Software Architecture.* An architecture for coordinating visualizations using the conceptual model and declarative language.

- *User Interface.* A user interface to the software architecture, for constructing highly-coordinated visualizations.

- *Implementation.* An implemented system that realizes the model, language, architecture, and user interface.

- *Classification.* A classification of coordinations into patterns in terms of the model and language.

Together, these innovations improve visualization construction over previous approaches by substantially increasing the flexibility, expressiveness, speed, and accessibility of visual data exploration and analysis without sacrificing the benefits of an integrated, iterative, fully interactive design process. Improvise has been used to create useful visualizations of election results, particle trajectories, network loads, county maps, music collections, chemical elements, layout and coordination in DEVise visualizations, and even the dynamic coordination structure of its own visualizations during construction and data exploration, as well as numerous other visualizations currently in various stages of development. As such, the coordination and visual abstraction approaches used in Improvise appear sufficient to realize a large fraction of all information visualization interfaces. The components of the coordination and visual abstraction language were developed as needed in order to fill specific analytic requirements during the design and implementation of these visualizations, suggesting necessity as well.

The following publications describe the research results of this dissertation:

- Chris Weaver. Building highly-coordinated visualizations in Improvise. In Proceedings of the IEEE Symposium on Information Visualization 2004, pages 159-166, Austin, TX, October 2004. IEEE. [148]
- Chris Weaver. Visualizing coordination in situ. In Proceedings of the IEEE Symposium on Information Visualization 2005, pages 165-172, Minneapolis, MN, October 2005. IEEE. [149]
- Chris Weaver. Metavisual exploration and analysis of DEVise coordination in Improvise. In Proceedings of the 4th International Conference on Coordinated & Multiple Views in Exploratory Visualization, London, UK, July 2006. [In Press.] [150]

## 9.2 Benefits

Improvise provides many benefits for visualization researchers, developers, designers, and users. For visualization researchers, Improvise:

- Reconceptualizes known coordinations as instances of particular coordination patterns.
- Provides a working environment for studying coordination patterns.
- Enables identification of useful variations of known patterns. Developers can incorporate these variations into their own visualization systems.
- Makes possible discovery of novel patterns.

For visualization developers, Improvise:

- Provides a collection of reusable views and controls.

- Eliminates the need to implement coordinations.
- Eliminates the need to implement data loading and querying.
- Exposes a compact API for implementing new views and controls as plug-ins. Developers can focus on the local appearance and behavior of their views and controls.

For visualization designers, Improvise:

- Provides a self-contained user interface based on the desktop metaphor.
- Offers dynamic construction in which accessing data, creating views, laying out views, editing coordinations, and querying data are all flexible and interactive.
- Offers persistence of complete and partial visualizations as XML files.
- Enables construction of highly-coordinated visualizations with many views.

For visualization users, Improvise:

- Provides user interfaces for viewing databases without programming.
- Enables access to visualizations as regular document files.
- Provides feedback about coordinations during interaction.
- Enables exploratory data analysis. Users can act as designers, switching rapidly between building and browsing.

## 9.3 Future Work

### 9.3.1 Non-Tabular Data Structures

Although Improvise focuses on visualization of tabular data, its query and visual abstraction language is only loosely related to the relational model. The way Coordinated Queries processes data is fundamentally record-based rather than table-based. As a result, extension to trees, graphs, and other non-tabular record-containing data structures would be a straightforward matter of adding appropriate modules—lexical objects, function operators, and views designed to handle non-tabular data structures—to the existing architecture. Modules could also be added to support more complex forms of structured data such as regular and irregular grids (e.g. [59]) and various space-time models (e.g. [112]). However, Coordinated Queries does not appear well suited for semi-structured and unstructured data, such as text corpora, in which records are a poor means of representing complexly interrelated data elements.

### 9.3.2 Space-Time Indexing

The speed of query operations is critical for achieving satisfactory interactive performance in highly-coordinated visualizations, particularly in applications such as geovisualization that involve a large number of spatial, temporal, and abstract data dimensions across multiple data sets. Although Improvise has sufficient functionality to visualize such queries, performance is generally poor, particularly in the common case in which the amount of geographic data exceeds available working memory.

One possibility would be to develop a custom database rendering engine to act as a common high-performance data source. Implemented on top of one of the many available database libraries (such as BerkeleyDB for Java [132]), the database engine would perform adaptive

multiple query optimization by building appropriate single- and multidimensional indexes on viewed data: hash table object indexes on abstract attributes, B-trees [8] on temporal dimensions and event attributes, R-trees [58] and quadtrees [47] on spatial dimensions, and so on. An experimental implementation of quadtrees in Improvise scatter plots currently enhances interactive performance for large data sets by minimizing the number of glyphs that must be generated and drawn during tile-based rendering updates.

### 9.3.3 Interactive Performance

Improvise was originally designed for visualization of static data sets. By reusing the existing coordination and visual abstraction architecture, adding metavisualization to Improvise was a straightforward process of representing the running visualization as dynamically changing data. Because this data changes frequently, however, ensuring satisfactory metavisualization responsiveness is a major challenge. Visual correctness demands interface consistency between a visualization and a metavisualization that are locked in interactive competition for the same (or at least significantly overlapping) computational resources.

In building Improvise to be a fully functional but research grade interactive visualization construction environment, the expressiveness of coordination and visual abstraction has been a much higher priority than interactive performance. Nevertheless, early work on Live Properties resulted in several performance-enhancing strategies for use in coordinated multiview visualizations [152], both with and without metavisualization.

### 9.3.4 Evaluation

Usability studies are a common component of visualization research. Even when the complexity of a particular visualization system or visual programming environment precludes controlled experimental approaches [116], as is the case with Improvise, studies based on Cognitive Walkthrough [56], Cognitive Task Analysis [115] and other less formal approaches often provide valuable qualitative knowledge about usability and usefulness (e.g. [5]). Amar and Stasko have recently developed a set of precepts to guide the design and evaluation of information visualization tools [6].

Improvise visualizations are currently undergoing preliminary evaluation at the Penn State Department of Geography using the HERO e-Delphi system [113]. Like other evaluation approaches, e-Delphi involves small groups of participants who may or may not have experience with visualization software. The Delphi approach [88] follows the philosophy of anonymity, asynchronicity, controlled feedback, and statistical response to achieve emergent collaborative consensus about complex problems. In this case, the e-Delphi web portal allows a moderator-led group of users to provide qualitative feedback about particular Improvise visualizations and Improvise in general, in response to both prescribed analysis tasks and free form exploration.

### 9.3.5 Unified Visualization Infrastructure

Improvise is designed to be a general-purpose interactive visualization construction system that is modular in terms of accessing, processing, and displaying data. Focusing on coordination and visual abstraction rather than graphical techniques and data manipulation has resulted in a visualization system that is strong in terms of architecture but weak in terms of components. Its library of implemented views and algorithms is relatively small, having evolved over time as needed to build example visualizations that demonstrate key uses of coordination and visual

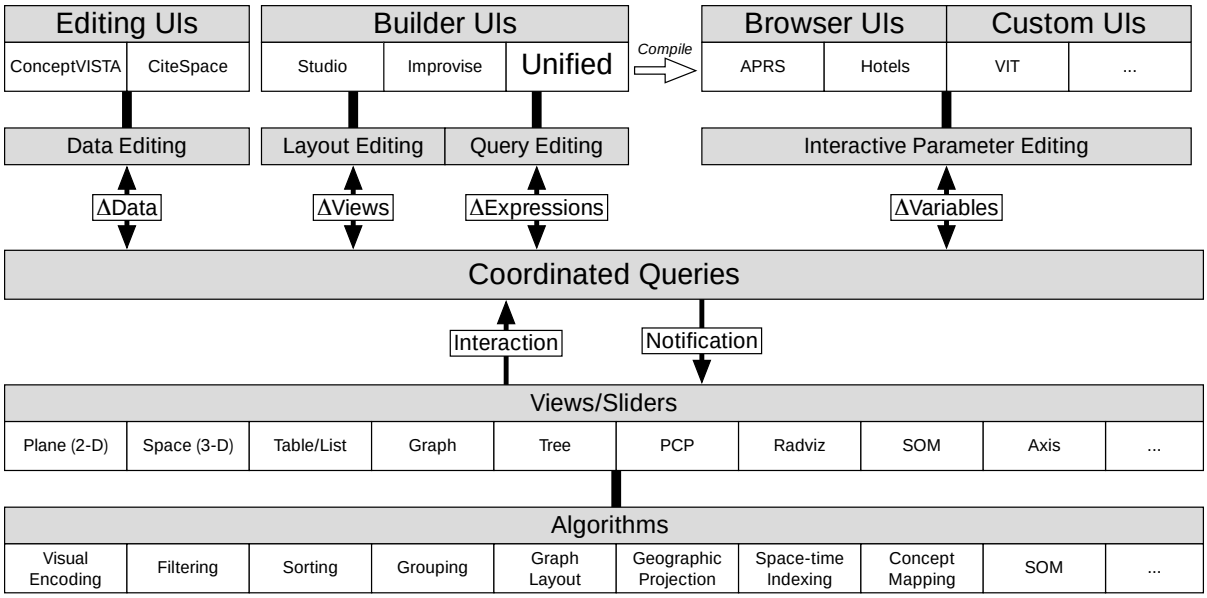



Figure 98: Unified visualization software architecture built around Coordinated Queries.

abstraction. Conversely, other visualization systems and toolkits implement a wide variety of sophisticated views and data processing algorithms, but typically require much more time and effort to combine them into useful visualizations, often requiring additional software development.

Combining the coordination and visual abstraction architecture of Improvise with the views and data processing algorithms of other systems and toolkits would result in a unified visualization software infrastructure having the advantages of both. As a part of ongoing research at the new North-East Visualization and Analytics Center (NEVAC) at Penn State, planning is currently underway to adapt GeoVISTA *Studio* [137], ConceptVISTA [94], and CiteSpace [26] to work with Improvise (figure 98).

### 9.3.6 Heterogeneous Collaborative Visualization

Like many other exploratory visualization systems, Improvise is designed for individual interaction. Collaborative visualization interfaces generally have been limited to custom implementations focused on particular knowledge domains and analysis scenarios [57]. Perhaps the most difficult problem in collaborative visualization is helping users maintain visual "common ground" during group analysis [31], particularly of geospatial information [123]. To solve current and future problems in visual analytics, it will be necessary for develop visualization hardware and software architectures that embrace heterogeneity in terms of *interaction* (coordination, collaboration), *display* (input capabilities, output devices), *information* (access, processing, delivery), *location* (distance, environmental conditions), *roles* (needs, skills, responsibilities, clearance), and *organization* (management/command, tech support, oversight).

Heterogeneous collaborative visualization would extend current exploratory visualization tools to multiple users of varying skill using different equipment in a range of environments. Potential benefits of heterogeneous collaborative visualization include:

- concurrent visualization by multiple users for emergency response, intelligence analysis, mission support, and scientific collaboratories,
- group visual exploration and analysis by mixed-role teams of users (responders/analysts), designers (tech support), and managers (command), and
- ultra-rapid direction, development and deployment of exploratory visualization tools in emergencies, investigations, and other time-critical situations.

The coordination and visual abstraction models implemented in Improvise appear to be well-suited for extension to heterogeneous collaboration (figure 99, bottom half). Improvise views follow a fine-grained, asynchronous notification and update protocol in which they act

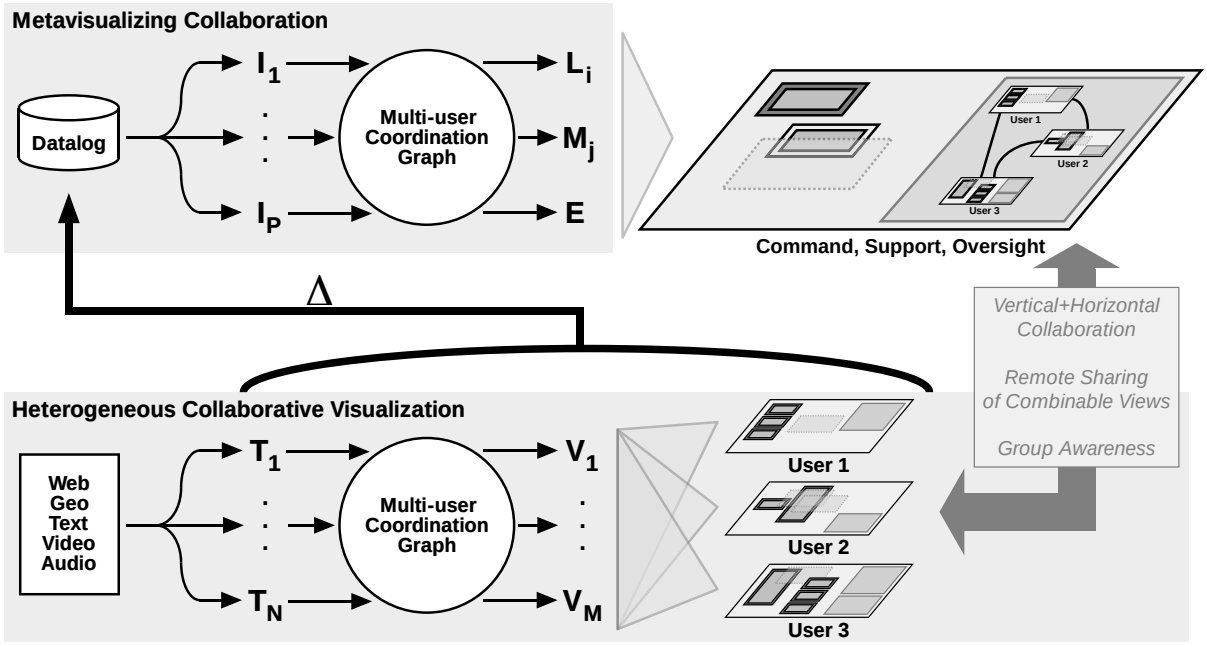



Figure 99: Model of heterogeneous collaborative visualization, with integrated metavisualization of the entire collaborative process.

as independent producers and consumers of interactive parameter changes. Because remote coordination would be expected to be functionally the same as local coordination, collaborative visualization in Improvise would involve a straightforward extension of coordinated multiview visualization in which multiple users share a distributed coordinated query graph.

### 9.3.7 Metavisualizing Collaboration

The metavisualization techniques described in chapters 7–8 focus on a tiny fraction of possible scenarios in which metavisual exploration and analysis focuses on a few current or past interactive states in a local visualization interface controlled by a single user. Metavisualization could be extended to support visual communication, training, management and oversight of the entire collaborative visual analytics process (figure 99, top half). Such a collaborative metavisualization system would need to:

- model the entire collaborative social process as a living, persistent information source,
- instrument the collaborative visualization system for continuous logging of all events,
- allow real-time and time-shifted visualization of the complete collaboration record,
- support rapid scrubbing of reconstructed, fully interactive interface states over multiple users, and
- integrate directly into the primary collaborative visualization system.

Potential benefits of metavisualizing collaboration include economical reuse of the visual analytics infrastructure for self-analysis (much as integrated metavisualization in Improvise reuses the existing visualization model and implementation), real-time visual management of the collaborative visual analytics process, and a visually immersive record for evaluation, training, and oversight.

## 9.4 Conclusions

McCormick [97] described one of visualization's greatest strengths: "It enriches the process of scientific discovery and fosters profound and unexpected insights." Interactive visualization construction tools further enrich this process by making visualization more widely accessible.

My principle goal has been to create an interactive visualization construction system in which coordinations, visual abstractions, and data queries can be combined and customized quickly and flexibly by visualization designers with modest training and little or no programming experience. I believe Improvise realizes this goal. In doing so, it opens up numerous possibilities for future research and development in information visualization and its application to visual analytics.

# Bibliography

[1] ESRI shapefile technical description, July 1998.

[2] Christopher Ahlberg and Ben Shneiderman. Visual information seeking: Tight coupling of dynamic query filters with starfield displays. In *Proceedings of CHI Conference: Human Factors in Computing Systems*, pages 313–317, 479–480, Boston, MA, April 1994. ACM.

[3] Christopher Ahlberg, Christopher Williamson, and Ben Shneiderman. Dynamic queries for information exploration: An implementation and evaluation. In *Proceedings of CHI: Human Factors in Computing Systems*, pages 619–626, Monterey, CA, May 1992. ACM.

[4] Christopher Ahlberg and Erik Wistrand. IVEE: An information visualization & exploration environment. In *Proceedings of the IEEE Symposium on Information Visualization*, pages 66–73, Los Alamitos, CA, 1995. IEEE Computer Press.

[5] Kenneth Allendoerfer, Serge Aluker, Gulshan Panjwani, Jason Proctor, David Sturtz, Mirjana Vukovic, and Chaomei Chen. Adapting the cognitive walkthrough method to assess the usability of a knowledge domain visualization. In *Proceedings of the IEEE Symposium on Information Visualization*, pages 195–202, Minneapolis, MN, October 2005. IEEE.

[6] Robert A. Amar and John T. Stasko. Knowledge precepts for the design and evaluation of information visualizations. *IEEE Transactions on Visualization and Computer Graphics*, 11(4):432–442, July 2005.

[7] Robert Ball, M. Varghese, A. Sabri, E. D. Cox, C. Fierer, M. Peterson, B. Carstensen, and Chris North. Evaluating the benefits of tiled displays for navigating maps. In *Proceedings of the IASTED International Conference on Human-Computer Interaction*, Phoenix, AZ, November 2005.

[8] Rudolf Bayer and Edward M. McCreight. Organization and maintenance of large ordered indices. *Acta Informatica*, 1:173–189, 1972.

[9] Richard A. Becker and William S. Cleveland. Brushing scatterplots. *Technometrics*, 29(2):127–142, 1987.

[10] Richard A. Becker, P. J. Huber, William S. Cleveland, and A. R. Wilks. Dynamic graphics for data analysis. *Stat. Science*, 2, 1987.

[11] B. B. Bederson, J. Meyer, and L. Good. Jazz: An extensible zoomable user interface graphics toolkit in Java. In *Proceedings of User Interface Software Technology (UIST)*, pages 171–180, San Diego, CA, 2000.

[12] Ben B. Bederson and James D. Hollan. PAD++: A zooming graphical user interface for exploring alternate interface physics. In *Proceedings of User Interface Software Technology (UIST)*, pages 17–27, 1994.

[13] Thomas Berlage. Using taps to separate the user interface from the application code. In *Proceedings of User Interface Software Technology (UIST)*, pages 191–198, Monterey, CA, November 1992. ACM.

[14] Jacques Bertin. *Semiology of Graphics: Diagrams, Networks, Maps*. University of Wisconsin Press, Ltd., Madison, WI, 1967/1983.

[15] Jacques Bertin. *Graphics and Graphic Information Processing*. Walter de Gruter & Co., Berlin, 1977/1981.

[16] Eric A. Bier, Maureen C. Stone, Thomas Baudel, William Buxton, , and Ken Fishkin. A taxonomy of see-through tools. In *Proceedings of CHI*, pages 358–364, Boston, MA, April 1994. ACM.

[17] Alan Borning. The programming language aspects of ThingLab, a constraint-oriented simulation laboratory. *ACM Transactions on Programming Languages and Systems*, 3(4):353–367, October 1981.

[18] Alan Borning and Robert Duisberg. Constraint-based tools for building user interfaces. *ACM Transactions on Graphics*, 5(4):345–374, October 1986.

[19] Nadia Boukhelifa and Peter Rodgers. A model and software system for coordinated and multiple views in exploratory visualization. *Information Visualization*, 2(4):258–269, December 2003.

[20] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, and Franois Yergeau, editors. *Extensible Markup Language (XML)*. World Wide Web Consortium, 3rd edition, February 2004.

[21] Andreas Buja, Dianne Cook, and Deborah F. Swayne. Interactive high-dimensional data visualization. *Journal of Computational and Graphical Statistics*, 5(1):78–99, 1996.

[22] Stuart K. Card and Jock D. Mackinlay. The structure of the information visualization design space. In *Proceedings of the IEEE Symposium on Information Visualization*, pages 92–99, 125. IEEE, 1997.

[23] Stuart K. Card, George G. Robertson, and William York. The WebBook and the Web-Forager: An information workspace for the world-wide web. In *Proceedings of the ACM CHI Conference: Human Factors in Computing Systems*, pages 111–117, New York, NY, 1996. ACM.

[24] M. S. T. Carpendale, D. J. Cowperthwaite, and F. D. Fracchia. Making distortions comprehensible. *Visual Languages*, pages 36–45, 1997.

[25] David Andrew Carr. *A Compact Graphical Representation of User Interface Interaction Objects*. PhD thesis, University of Maryland, 1995.

[26] Chaomei Chen. CiteSpace II: Detecting and visualizing emerging trends and transient patterns in scientific literature. *Journal of the American Society for Information Science and Technology*, 57(3):359–377, 2006.

[27] Chaomei Chen, Fidelia Ibekwe-SanJuan, Eric SanJuan, and Chris Weaver. Visual analysis of conflicting opinions. In *Proceedings of the Symposium on Visual Analytics Science and Technology*, Baltimore, MD, October 2006. IEEE [Accepted].

[28] Ed H. Chi. A taxonomy of visualization techniques using the data state reference model. In *Proceedings of the IEEE Symposium on Information Visualization*, pages 69–75, Salt Lake City, UT, October 2000. IEEE.

[29] Ed H. Chi. Expressiveness of the data flow and data state models in visualization systems. In *Proceedings of Advanced Visual Interfaces (AVI)*, pages 375–378, Trento, Italy, May 2002. ACM Press.

[30] M. C. Chuah and S. F. Roth. On the semantics of interactive visualizations. In *Proceedings of the IEEE Symposium on Information Visualization*, pages 29–36. IEEE, 1996.

[31] Mei C. Chuah and Steven F. Roth. Visualizing common ground. In *Proceedings of the Seventh International Conference on Information Visualization (IV)*, pages 365–372, London, UK, July 2003.

[32] William S. Cleveland. *Visualizing Data*. Hobart Press, Summit, NJ, 1993.

[33] William S. Cleveland. *The Elements of Graphing Data*. AT&T Bell Laboratories, September 1994.

[34] Diane Cluxton, Stephen G. Eick, and Jie Yun. Hypothesis visualization. In *Proceedings of the IEEE Symposium on Information Visualization (Posters Compendium)*, pages 9–10, Austin, TX, October 2004. IEEE.

[35] E. F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, June 1970.

[36] Nathan Conklin, Sandeep Prabhakar, and Chris North. Multiple foci drill-down through tuple and attribute aggregation polyarchies in tabular data. In *Proceedings of the IEEE Symposium on Information Visualization*, page 131, Washington, DC, 2002. IEEE Computer Society.

[37] James O. Coplien and Douglas C. Schmidt, editors. *Pattern Languages of Program Design*, New York, NY, USA, 1995. ACM Press/Addison-Wesley Publishing Co.

[38] Allen Cypher. Eager: Programming repetitive tasks by example. In *Proceedings of CHI: Human Factors in Computing Systems*, pages 33–39, New Orleans, LA, May 1991. ACM.

[39] David Duce, editor. *Portable Network Graphics (PNG) Specification*. World Wide Web Consortium, 2nd edition, November 2003.

[40] D. Scott Dyer. A dataflow toolkit for visualization. *Computer Graphics and Applications*, 10(4):60–69, July 1990.

[41] Stephen G. Eick. Data visualization sliders. In *Proceedings of User Interface Software Technology (UIST)*, pages 119–120, Monterey, CA, November 1994. ACM Press.

[42] Stephen G. Eick. Interactive data visualization at AT&T bell laboratories. In *CHI Companion*, pages 17–18, Denver, CO, May 1995. ACM.

[43] Stephen G. Eick. Engineering perceptually effective visualizations for abstract data. In G. Nielson, H. Mueller, and H. Hagen, editors, *Scientific Visualization Overviews, Methodologies and Techniques*, pages 191–210. IEEE Computer Science Press, February 1997.

[44] Jean-Daniel Fekete. The infovis toolkit. In *Proceedings of the IEEE Symposium on Information Visualization*, pages 167–174, Austin, TX, October 2004. IEEE.

[45] Jean-Daniel Fekete and Catherine Plaisant. Interactive information visualization of a million items. In *Proceedings of the IEEE Symposium on Information Visualization*, pages 117–124, Boston, MA, 2002. IEEE.

[46] Jan Ferraiolo, Jun Fujisawa, and Dean Jackson, editors. *Scalable Vector Graphics (SVG) 1.1 Specification*. World Wide Web Consortium, January 2003.

[47] Raphael A. Finkel and Jon L. Bentley. Quad trees—a data structure for retrieval on composite keys. *Acta Informatica*, 4(1):1–9, 1974.

[48] Ken Fishkin and Maureen C. Stone. Enhanced dynamic queries via movable filters. In *Proceedings of the ACM CHI Conference: Human Factors in Computing Systems*, pages 415–420, Denver, CO, May 1995. ACM.

[49] G. W. Furnas. Generalized fisheye views. In *Proceedings of the Conference on Human Factors in Computing Systems (CHI)*, pages 18–23, 1986.

[50] Mark Gahegan, Xiping Dai, James Macgill, and Sachin Oswal. From concepts to data and back again: Connecting mental spaces with data and analysis methods. In *Proceedings of the Seventh International Conference on GeoComputation*, Southampton, UK, September 2003.

[51] Mark Gahegan, Monica Wachowicz, Mark Harrower, and Theresa-Marie Rhyne. The integration of geographic visualization with knowledge discovery in databases and geocomputation. *Cartography and Geographic Information Society*, 28(1), January 2001.

[52] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison Wesley, 1st edition, October 1994.

[53] Nahum Gershon. What storytelling can do for information visualization. *Communications of the ACM*, 44(8):31–37, August 2001.

[54] Jade Goldstein and Steven F. Roth. Using aggregation and dynamic queries for exploring large data sets. In *CHI '94: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, number 0-89791-650-6, pages 23–29, Boston, MA, April 1994. ACM Press.

[55] Michael M. Gorlick and Alex Quilici. Visual programming-in-the-large versus visual programming-in-the-small. In *Proceedings of the IEEE Symposium on Visual Languages*, pages 137–144, St. Louis, MO, October 1994. IEEE.

[56] T. Green and M. Petre. Usability analysis of visual programming environments: A 'cognitive dimensions' framework. *Journal of Visual Languages and Computing*, 7:131–174, 1996.

[57] Ian J. Grimstead, David W. Walker, and Nick J. Avis. Collaborative visualization: A review and taxonomy. In *Proceedings of the Ninth International Symposium on Distributed Simulation and Real-Time Applications*, pages 61–69. IEEE, October 2005.

[58] Antonin Guttman. R-trees: A dynamic index structure for spatial searching. In *Proceedings of SIGMOD International Conference on Management of Data*, pages 47–57, Boston, MA, 1984. ACM Press.

[59] Robert B. Haber, Bruce Lucas, and Nancy S. Collins. A data model for scientific visualization with provisions for regular and irregular grids. In *Proceedings of the IEEE Conference on Visualization*, pages 298–305, San Diego, CA, October 1991. IEEE.

[60] P. O. Haeberli. ConMan: A visual programming language for interactive graphics. *Proceedings of ACM SIGGRAPH*, 22(4):103–111, August 1988.

[61] G. Hamilton, editor. *JavaBeans*. Sun Microsystems, Inc., 1997.

[62] Jeffrey Heer, Stuart K. Card, and James A. Landay. Prefuse: A toolkit for interactive information visualization. In *Proceedings of CHI*, Portland, OR, April 2005. ACM.

[63] Joseph M. Hellerstein, Michael J. Franklin, Sirish Chandrasekaran, Amol Descpande, Kris Hildrum, Sam Madden, Vijayshankar Raman, and Mehul A. Shah. Adaptive query processing: Technology in evolution. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 23(2), 2000.

[64] William Hibbard, Charles R. Dyer, and Brian E. Paul. Display of scientific data structures for algorithm visualization. In *Proceedings of the IEEE Conference on Visualization*, pages 139–146, Boston, MA, 1992. IEEE.

[65] William Hibbard, Curtis Rueden, Tom Rink, John Anderson, Steve Emmerson, and David Fulker. The VisAD java class library for scientific data and visualization. In *Proceedings of Scientific Visualization Conference*, pages 115–123, 1997.

[66] Ralph D. Hill. The abstraction-link-view paradigm: Using constraints to connect user interfaces to applications. In *Proceedings of CHI*, pages 335–342, Monterey, CA, May 1992. ACM.

[67] Ralph D. Hill, Tom Brinck, Steven L. Rohall, John F. Patterson, and Wayne Wilner. The Rendezvous architecture and language for constructing multiuser applications. *ACM Transactions on Visualization and Computer Graphics*, 1(2):81–125, June 1994.

[68] Matthew E. Hodges, Russell M. Sasnett, and Mark S. Ackerman. A construction set for multimedia applications. *IEEE Software*, 6(1):37–43, January 1989.

[69] Dugald Ralph Hutchings and John Stasko. Revisiting display space management: Understanding current practice to inform next-generation design. In *Proceedings of the Conference on Graphics Interface*, pages 127–134, London, Ontario, Canada, 2004.

[70] Takeo Igarashi, Jock D. Mackinlay, Bay-Wei Chang, and Polle T. Zellweger. Fluid visualization of spreadsheet structures. In *Proceedings of the 14th IEEE Symposium on Visual Languages*, Halifax, Nova Scotia, Canada, September 1998. IEEE.

[71] R. Ihaka and R. Gentleman. R: A language for data analysis and graphics. *Journal of Computational and Graphical Statistics*, 5(3):299–314, September 1996.

[72] Alfred Inselberg. The plane with parallel coordinates. *The Visual Computer*, 1:69–91, 1985.

[73] Allan S. Jacobson, Andrew L. Berkin, and Martin N. Orton. LinkWinds: Interactive scientific data analysis and visualization. *Communications of the ACM*, 37(4):43–52, April 1994.

[74] Brian Johnson and Ben Shneiderman. Tree-maps: A space-filling approach to the visualization of hierarchical information structures. In *Proceedings of the IEEE Conference on Visualization*, pages 284–291, San Diego, CA, October 1991. IEEE.

[75] Eser Kandogan and Ben Shneiderman. Elastic windows: Evaluation of multi-window operations. In *Proceedings of CHI*, pages 250–257, Atlanta, GA, March 1997. ACM.

[76] T. Alan Keahey and Stephen G. Eick. Visual path analysis. In *Proceedings of the IEEE Symposium on Information Visualization*, pages 165–168, Boston, MA, October 2002. IEEE.

[77] Daniel A. Keim. Scaling visual analytics to very large data sets. In *Workshop on Visual Analytics*, June 2005.

[78] Daniel A. Keim and Hans-Peter Kriegel. Visualization techniques for mining large databases: A comparison. *IEEE Transactions on Knowledge and Data Engineering*, 8(6):923–938, December 1996.

[79] Kevin B. Kenny. Tclsolver: An algebraic constraint manager for tcl. In *Proceedings of the Fourth USENIX Tcl/Tk Workshop*, Monterey, CA, July 1996.

[80] Bernard Kerr. THREAD ARCS: An email thread visualization. In *Proceedings of the IEEE Symposium on Information Visualization*, pages 211–218, Seattle, WA, October 2003. ACM.

[81] Donald E. Knuth. *The Art of Computer Programming, Volume 1: Fundamental Algorithms*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, 3rd edition, 1997.

[82] John A. Kolojejchick, Steven F. Roth, and Peter Lucas. Information appliances and tools in Visage. *Computer Graphics and Applications*, 17(4):32–41, July 1997.

[83] G. E. Krasner and S. T. Pope. A cookbook for using the model-view-controller user interface paradigm in Smalltalk-80. *Journal of Object-Oriented Programming*, 1(3):26–49, August 1988.

[84] Ravi Krishnamurthy and Moshé Zloof. RBE: Rendering by example. In *Proceedings of the 11th International Conference on Data Engineering*, pages 288–297, Taipei, Taiwan, March 1995.

[85] Harsha Kumar, Catherine Plaisant, and Ben Shneiderman. Browsing hierarchical data with multi-level dynamic queries and pruning. *International Journal of Human-Computer Studies*, 46(1):103–124, January 1997.

[86] John Lamping, Ramana Rao, and Peter Pirolli. A focus + context technique based on hyperbolic geometry for visualizing large hierarchies. In *Proceedings of ACM CHI Conference: Human Factors in Computing Systems*, pages 401–408, New York, NY, 1995. ACM.

[87] Shilpa Lawande. A meta-visualization framework for DEVise. Master's thesis, University of Wisconsin–Madison, December 1997.

[88] Harold A. Linstone and Murray Turoff, editors. *The Delphi Method: Techniques and Applications*, Reading, MA, 1975. Addison-Wesley.

[89] Yunping Liu, Mark Gahegan, and James Macgill. Increasing geocomputational interoperability: Towards a standard geocomputation API. In *Proceedings of GeoComputation*, Ann Arbor, MI, 2005.

[90] Miron Livny, Raghu Ramakrishnan, Kevin Beyer, G. Chen, Donko Donjerkovic, Shilpa Lawande, Jussi Myllymaki, and Kent Wenger. DEVise: Integrated querying and visualization of large datasets. In *Proceedings of SIGMOD*, pages 301–312, Tucson, AZ, 1997. ACM.

[91] Miron Livny, Raghu Ramakrishnan, and Jussi Myllymaki. Visual exploration of large datasets. In *Proceedings of SPIE Vol. 2657*, pages 301–312, San Jose, CA, January 1996.

[92] Bruce Lucas, Gregory D. Abram, Nancy S. Collins, David A. Epstein, Donna L. Gresh, and Kevin P. McAuliffe. An architecture for a scientific visualization system. In *Proceedings of the IEEE Conference on Visualization*, pages 107–114, Boston, MA, October 1992. IEEE.

[93] Alan M. MacEachren. *How Maps Work : Representation, Visualization, And Design*. The Guilford Press, June 2004.

[94] Alan M. MacEachren, Mark Gahegan, William Pike, Isaac Brewer, Guoray Cai, Eugene Lengerich, and Frank Hardisty. Geovisualization for knowledge construction and decision support. *IEEE Computer Graphics and Applications*, 24(1):13–17, January 2004.

[95] Jock D. Mackinlay. Automating the design of graphical presentations of relational information. *ACM Transactions on Graphics*, 5(2):110–141, April 1986.

[96] Jock D. Mackinlay, George G. Robertson, and Stuart K. Card. The perspective wall: Detail and context smoothly integrated. In *Proceedings of CHI*, pages 173–176, New Orleans, LA, 1991. ACM Press.

[97] B. H. McCormick, T. A. DeFanti, and M. D. Brown. Visualization in scientific computing. *Computer Graphics*, 21(6), November 1987.

[98] Tamara Munzner and Paul Burchard. Visualizing the structure of the world wide web in 3d hyperbolic space. *Computer Graphics*, pages 33–38, December 1995.

[99] Brad A. Myers. *Creating User Interfaces by Demonstration*. Academic Press, San Diego, CA, 1988.

[100] Brad A. Myers. Window interfaces: A taxonomy of window manager user interfaces. *Computer Graphics and Applications*, 8(5):65–84, September 1988.

[101] Brad A. Myers, D. Giuse, R. B. Dannenberg, Brad Vander Zanden, D. Kosbie, E. Pervin, A. Mickish, and P. Marchal. Garnet: Comprehensive support for graphical, highly-interactive user interfaces. *IEEE Computer*, 23(11):71–85, November 1990.

[102] David A. Nation. WebTOC: A hierarchical browser for websites. In *Proceedings of the 3rd Conference on Human Factors and the Web*, 1997.

[103] Chris North, Nathan Conklin, and Varun Saini. Visualization schemas for flexible information visualization. In *Proceedings of the IEEE Symposium on Information Visualization*, pages 15–22, Boston, MA, October 2002. IEEE.

[104] Chris North and Ben Shneiderman. A taxonomy of multiple window coordinations. Technical Report CS-TR-3854, University of Maryland Department of Computer Science, 1997.

[105] Chris North and Ben Shneiderman. Snap-together visualization: A user interface for coordinating visualizations via relational schemata. In *Proceedings of the Working Conference on Advanced Visual Interfaces (AVI)*, pages 128–135, New York, NY, USA, May 2000. ACM Press.

[106] Chris North, Ben Shneiderman, and Catherine Plaisant. User controlled overviews of an image library: A case study of the visible human. In *Proceedings of the 1st ACM International Conference on Digital Libraries*, pages 74–82, Bethesda, MD, March 1996. ACM.

[107] Christopher Loy North. *A User Interface for Coordinating Visualization Based On Relational Schemata: Snap-Together Visualization*. PhD thesis, University of Maryland, 2000.

[108] Lucille Terry Nowell. *Graphical Encoding for Information Visualization: Using Icon Color, Shape, and Size To Convey Nominal and Quantitative Data*. PhD thesis, Virginia Polytechnic Institute, Blacksburg, VA, November 1997.

[109] Chris Olston, Michael Stonebraker, Alexander Aiken, and Joseph M. Hellerstein. VIQING: Visual Interactive QueryING. In *Proceedings of the 14th IEEE Symposium on Visual Languages*, Halifax, Nova Scotia, Canada, September 1998. IEEE.

[110] Chris Olston, Allison Woodruff, Alexander Aiken, Michael Chu, Vuk Ercegovac, Mark Lin, Mybrid Spalding, and Michael Stonebraker. DataSplash. In *Proceedings of SIGMOD*, pages 550–552, Seattle, WA, June 1998. ACM.

[111] Ken Perlin and David Fox. Pad: An alternative approach to the computer interface. In *Proceedings of SIGGRAPH*, pages 57–64, Anaheim, CA, August 1993.

[112] Donna J. Peuquet. *Representations of Space and Time*. The Guilford Press, June 2002.

[113] William Pike, Brent Yarnal, Alan M. MacEachren, Mark Gahegan, and C. Yu. Infrastructure for human-environment collaboration: Building a prototype for the future of science. *Environment*, 47(2):8–21, 2005.

[114] Harald Piringer, Robert Kosara, and Helwig Hauser. Interactive focus+context visualization with linked 2D/3D scatterplots. In *Proceedings of Coordinated and Multiple Views (CMV)*, London, UK, July 2004.

[115] Peter Pirolli and Stuart Card. The sensemaking process and leverage points for analyst as identified through cognitive task analysis. In *International Conference on Intelligence Analysis*, McLean, VA, May 2005.

[116] Catherine Plaisant. The challenge of information visualization evaluation. In *Proceedings of ACM Conference on Advanced Visual Interfaces (AVI)*, pages 109–116, Gallipoli, Italy, May 2004. ACM.

[117] Catherine Plaisant, David Carr, and Ben Shneiderman. Image browser taxonomy and guidelines for designers. *IEEE Software*, 12(2):21–32, March 1995.

[118] Catherine Plaisant, Anne Rose, Brett Milash, Seth Widoff, and Ben Shneiderman. LifeLines: Visualizing personal histories. In *Proceedings of CHI Conference: Human Factors in Computing Systems*, pages 221–227, 518, New York, NY, 1996. ACM.

[119] Ramana Rao and Stuart K. Card. The table lens: Merging graphical and symbolic representations in an interactive focus context visualization for tabular information. In *Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI)*. ACM Press, 1994.

[120] Jonathan C. Roberts. Multiple-view and multiform visualization. In Robert Erbacher, Alex Pang, Craig Wittenbrink, and Jonathan Roberts, editors, *Proceedings of SPIE (Visual Data Exploration and Analysis VII)*, volume 3960, pages 176–185. SPIE, January 2000.

[121] Steven F. Roth, Peter Lucas, Jeffrey A. Senn, Cristina C. Gomberg, Michael B. Burks, Philip J. Stroffolino, John A. Kolojejchick, and Carolyn Dunmire. Visage: A user interface environment for exploring information. In *Proceedings of the IEEE Symposium on Information Visualization*, pages 3–12. IEEE, 1996.

[122] Wendy A. Schafer. *Supporting Spatial Collaboration: An Investigation of Viewpoint Constraint and Awareness Techniques*. PhD thesis, Virginia Polytechnic Institute, Blacksburg, VA, April 2004.

[123] Wendy A. Schafer, Craig H. Ganoe, Lu Xiao, Gabriel Coch, and John M. Carroll. Designing the next generation of distributed geocollaborative tools. *Cartography and Geographic Information Science*, 32(2):81–100, April 2005.

[124] W. J. Schroeder, W. E. Lorensen, G. D. Montanaro, and C. R. Volpe. VISAGE: An object-oriented scientific visualization system. In *Proceedings of the IEEE Conference on Visualization*, pages 219–226. IEEE, 1992.

[125] Jinwook Seo and Ben Shneiderman. Knowledge discovery in high dimensional data: Case studies and a user survey for an information visualization tool. *IEEE Transactions on Visualization and Computer Graphics*, 2006.

[126] Christopher D. Shaw, James A. Hall, David S. Ebert, and D. Aaron Roberts. Interactive lens visualization techniques. In *Proceedings of the IEEE Conference on Visualization*, pages 155–160, 521, San Francisco, CA, October 1999. IEEE.

[127] Ben Shneiderman. Direct manipulation: A step beyond programming languages. *IEEE Computer*, 16(8):57–69, August 1983.

[128] Ben Shneiderman. Dynamic queries for visual information seeking. *IEEE Software*, 11(6):70–77, November 1994.

[129] Ben Shneiderman. The eyes have it: A task by data type taxonomy for information visualizations. In *Proceedings of the IEEE Symposium on Visual Languages*, pages 336–343, Boulder, CO, September 1996. IEEE.

[130] Ben Shneiderman. *Designing the User-Interface: Strategies for Human-Computer Interaction*. Addison-Wesley, Reading, MA, 3rd edition, 1998.

[131] Simon Buckingham Shum, Victoria Uren Gangmin Li, John Domingue, and Enrico Motta. Visualizing internetworked argumentation. In Paul A. Kirschner, Simon J. Buckingham Shum, and Chad S. Carr, editors, *Visualizing Argumentation: Software Tools for Collaborative and Educational Sense-Making*, pages 185–204. Springer–Verlag, December 2002.

[132] Sleepycat Software. Berkeley DB java edition.

[133] Chris Stolte, Diang Tang, and Pat Hanrahan. Multiscale visualization using data cubes. In *Proceedings of the IEEE Symposium on Information Visualization*, pages 7–14, Boston, MA, October 2002. IEEE.

[134] Chris Stolte, Diang Tang, and Pat Hanrahan. Polaris: A system for query, analysis, and visualization of multi-dimensional relational databases. *IEEE Transactions on Visualization and Computer Graphics*, 8(1):52–65, January 2002.

[135] Deborah F. Swayne, Dianne Cook, and Andreas Buja. User's manual for XGobi, a dynamic graphics program for data analysis. Technical report, Bellcore, 1992.

[136] Deborah F. Swayne, Duncan Temple Lang, Andreas Buja, and Dianne Cook. GGobi: Evolving from XGobi into an extensible framework for interactive data visualization. *Computational Statistics & Data Analysis*, 43:423–444, 2003.

[137] Masahiro Takatsuka and Mark Gahegan. GeoVISTA Studio: A codeless visual programming environment for geoscientific data analysis and visualization. *Computational Geoscience*, 28(10):1131–1144, 2002.

[138] James J. Thomas and Kristin A. Cook, editors. *Illuminating the Path: The Research and Development Agenda for Visual Analytics*. IEEE Computer Society, August 2005.

[139] Edward Tufte. *The Visual Display of Quantitiative Information*. Graphics Press, Cheshire, CT, 1983.

[140] Edward Tufte. *Envisioning Information*. Graphics Press, Cheshire, CT, 1990.

[141] Edward Tufte. *Visual Explanations: Images and Quantities, Evidence and Narrative*. Graphics Press, Cheshire, CT, 1997.

[142] Lisa Tweedie. Describing interactive visualization artifacts—DIVA. In *Proceedings of CHI*, pages 73–74. ACM, May 1995.

[143] Lisa Tweedie. Characterizing interactive externalizations. In *Proceedings of CHI*, pages 375–382, Atlanta, GA, March 1997. ACM.

[144] Craig Upson, Thomas Faulhaber, Jr., David Kamins, David Laidlaw, David Schlegel, Jeffrey Vroom, Robert Gurwitz, and Andries van Dam. The application visualization system: A computational environment for scientific visualization. *Computer Graphics and Applications*, 9(4):30–42, July 1989.

[145] Matthew O. Ward. XmdvTool: Integrating multiple methods for visualizing multivariate data. In *Proceedings of the IEEE Conference on Visualization*. IEEE, 1994.

[146] Colin Ware. *Information Visualization: Perception for Design*. The Morgan Kaufmann Series in Interactive Technologies. Morgan Kaufmann, 2nd edition, April 2004.

[147] Martin Wattenberg. Arc diagrams: Visualizing structure in strings. In *Proceedings of the IEEE Symposium on Information Visualization*, pages 110–116, Boston, MA, October 2002. ACM.

[148] Chris Weaver. Building highly-coordinated visualizations in Improvise. In *Proceedings of the IEEE Symposium on Information Visualization*, pages 159–166, Austin, TX, October 2004. IEEE.

[149] Chris Weaver. Visualizing coordination in situ. In *Proceedings of the IEEE Symposium on Information Visualization*, pages 165–172, Minneapolis, MN, October 2005. IEEE.

[150] Chris Weaver. Metavisual exploration and analysis of DEVise coordination in Improvise. In *Proceedings of the 4th Internation Conference on Coordinated & Multiple Views in Exploratory Visualization*, London, UK, July 2006.

[151] Chris Weaver, David Fyfe, Anthony Robinson, Deryck W. Holdsworth, Donna J. Peuquet, and Alan M. MacEachren. Visual analysis of historic hotel visitation patterns. In *Proceedings of the Symposium on Visual Analytics Science and Technology*, Baltimore, MD, October 2006. IEEE [Accepted].

[152] Christopher E. Weaver and Miron Livny. Improving visualization interactivity in Java. In *Proceedings of SPIE Vol. 3960 (Visual Data Exploration and Analysis VII)*, San Jose, CA, January 2000. SPIE.

[153] Christopher E. Weaver and Miron Livny. Interactive poster: Metavisualization of dynamic queries. In *Proceedings of the IEEE Symposium on Information Visualization*, Austin, TX, October 2002. IEEE.

[154] Leland Wilkinson. *The Grammar of Graphics*. Springer–Verlag, August 1999.

[155] Christopher Williamson and Ben Shneiderman. The Dynamic HomeFinder: Evaluating dynamic queries in a real-estate information exploration system. In *Proceedings of ACM SIGIR Conference*, pages 338–346, Copenhagen, Denmark, 1992.

[156] Allison Woodruff, James Landay, and Michael Stonebraker. Constant information density in zoomable interfaces. In *Proceedings of Advanced Visual Interfaces (AVI)*, pages 57–65, L'Aquila, Italy, May 1998.

[157] Allison Woodruff, Alan Su, Michael Stonebraker, Caroline Paxson, Jolly Chen, Alexander Aiken, Peter Wisnovsky, and Cimarron Taylor. Navigation and coordination primitives for multidimensional visual browsers. In S. Spaccapietra and R. Jain, editors, *Proceedings of the 3rd IFIP 2.6 Working Conference on Visual Database Systems*, pages 360–371, Lausanne, Switzerland, March 1995. Chapman & Hall.

[158] William Wright, David Schroh, Pascale Proulx, Alex Skaburskis, and Brian Cort. Advances in nSpace – the sandbox for analysis. In *International Conference on Intelligence Analysis*, McLean, VA, May 2005.

# Appendix A

# Examples

## A.1 Overview

Improvise has been used to create highly-coordinated visualizations of information from a wide variety of knowledge domains, including chemistry, politics, sociology, musical entertainment, geography, and computer science. Examples of visualized data sets include:

- simulated ion trajectories in a cubic ion trap,
- county-level federal and state election results in Michigan,
- physical properties of elements in the periodic table,
- roads and hydrography features in Michigan,
- demographics from the 2000 United States Census,
- iTunes playlists with cover art,
- the X Windows color table, and
- screen layout and coordination in DEVise visualizations.

The information displayed in these examples includes text, numbers, colors, images, and geographic regions accessed as both tabular and graph-structured data from a variety of sources in a variety of formats.

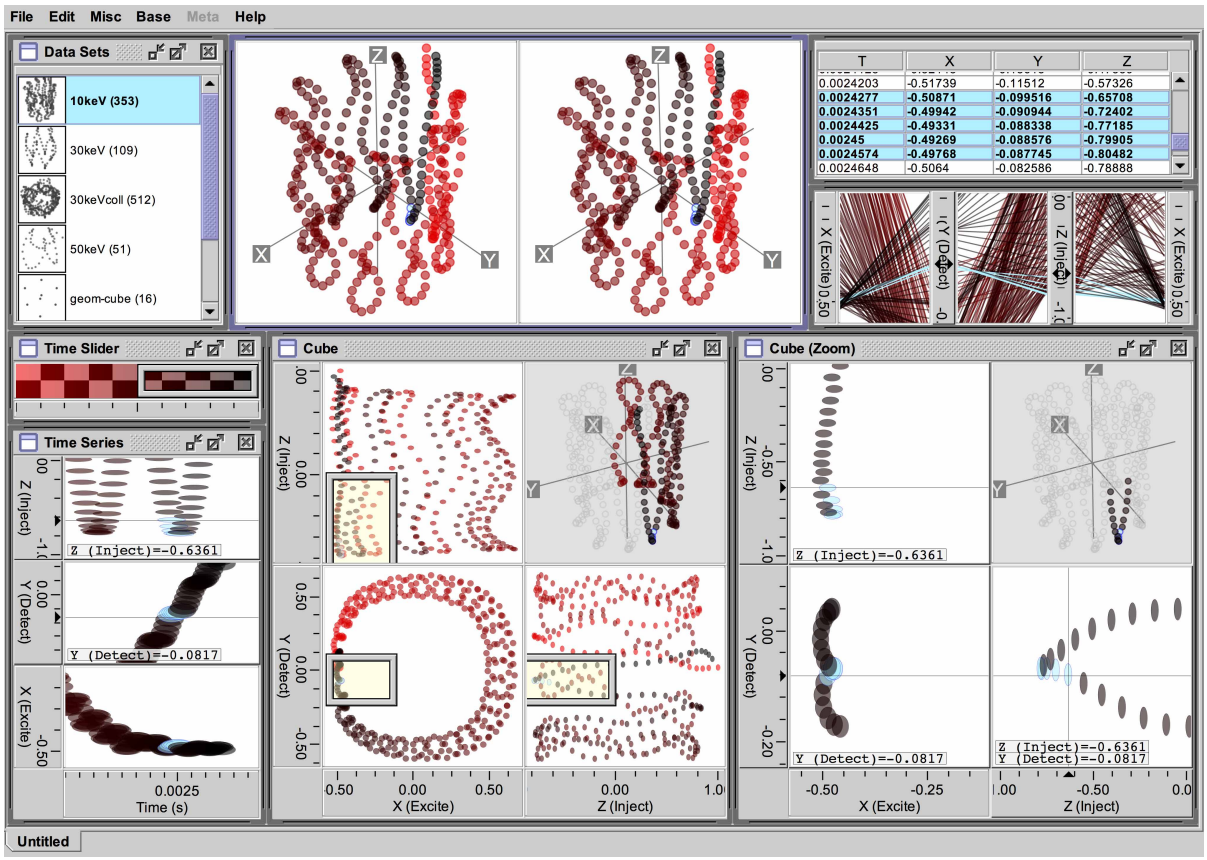



Figure 100: FT/ICR/MS ion trajectories (`ions.viz`)

## A.2 Simulated FT/ICR/MS Ion Trajectories

Figure 100 shows a visualization of the simulated trajectory of an ion in a cubic ion trap under difference excitation energies, taken from Fourier Transform Ion Cyclotron Resonance Mass Spectrometry (FT/ICR/MS) research at the National High Magnetic Field Lab (NHMFL)[1].

The data was calculated using the parametric equations for cyclotron/magnetron motion of ion clouds in a one-inch trap placed in the center of a large (3+ tesla) magnetic field generated by a cylindrical superconducting magnet. In each experiment, ions are injected along the `Z` axis of the trap/magnet, energized by a temporary electrical charge on the excite (`X` axis) trap plates, then detected by measuring charge on the detect (`Y` axis) trap plates induced by the motion of

[1] http://www.nhmfl.gov/science/cimar/icr/

the ion cloud. The trajectory shown in the visualization consists of the motion which occurs during the detection phase of an experiment.

Users start by selecting a trajectory from a list that displays each trajectory in miniature. The selected trajectory is displayed in a variety of ways by the other views in the visualization:

- A left-right stereogram pair shows the trajectory in space. These views are navigationally coordinated with the miniature views in the list.
- A table view shows the actual $\{T, X, Y, Z\}$ positions of points along the trajectory.
- Three time series scatter plots show ion motion over time relative to each pair of trap plates (excite, detect, inject).
- A 3-D matrix of overview scatter plots shows the entire trajectory as seen from three sides of the ion trap.
- A second 3-D matrix of detail scatter plots shows the subtrajectory that intersects the space selected by the portals in the overview scatter plots.
- Three parallel coordinate plots show the same subtrajectory.
- Monoscopic stereo views show the trajectory and selected subtrajectory next to the corresponding scatter plot matrices.

All views map time into a translucent red-to-black color gradient. Using the time slider, the user can select a range of times perceptually rather than by specific values. All views except the overview scatter plots and stereo view are filtered to show only the selected range of colors.

Although the data sets displayed in this visualization are small (512 records for the longest trajectory in the list), most of the views are coordinated with each other through eight range variables (`XOverview`, `YOverview`, `ZOverview`, `XDetail`, `YDetail`, `ZDetail`,

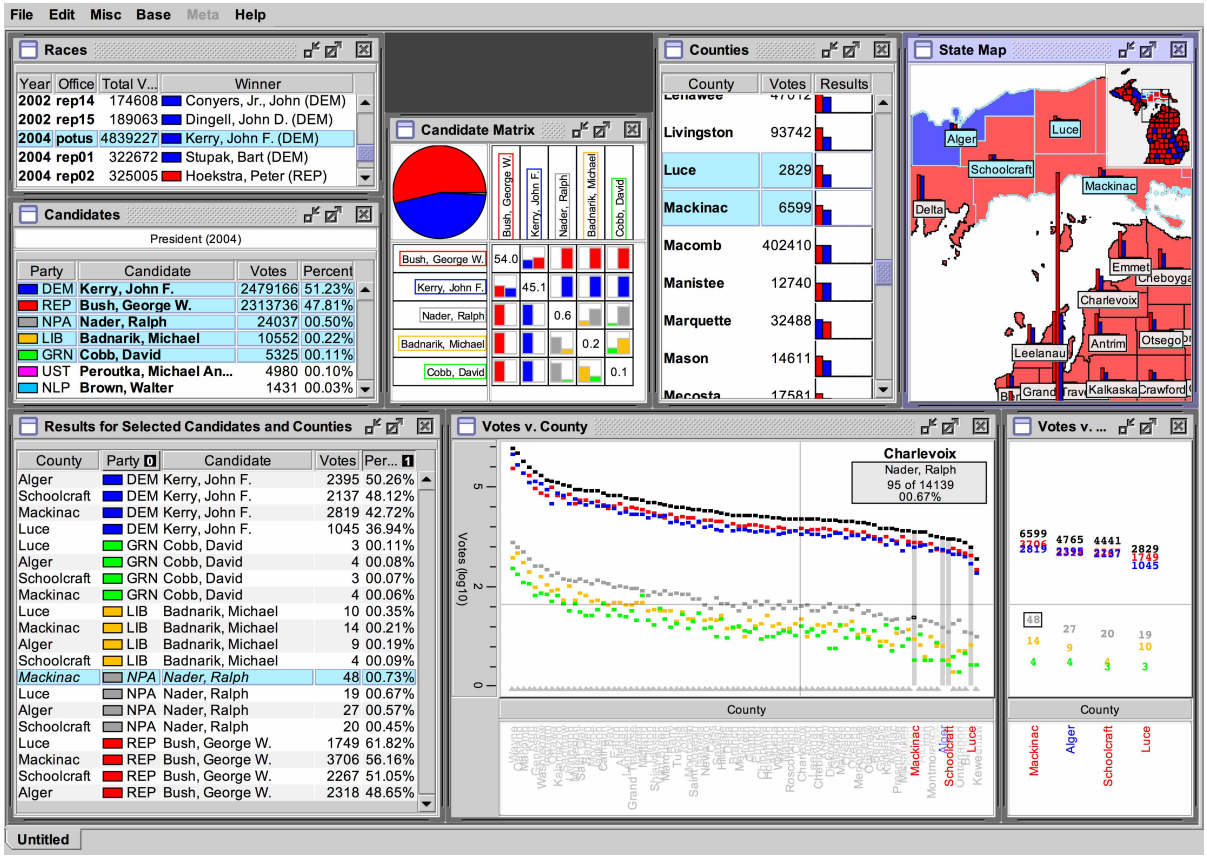



Figure 101: Michigan election results, 1998-2004 (`elections.viz`)

`VisibleTime`, `SelectedTime`). This means that even small mouse interactions result in the data set being projected, filtered, and rendered multiple times. For instance, when a mouse drag or keystroke in a view or axis changes the `XDetail` range, nine views update by processing $9 \times 512 = 4608$ records. Four axes and two portals also update to reflect the change.

## A.3 Michigan Elections 1998-2004

The visualization in figure 101 shows county-level election results in Michigan. The data consists of vote totals in regular elections of federal and state offices from 1998 to 2004 (source:

Michigan Secretary of State[2]). The visualization also generates a state map by accessing shapefiles which describe the borders of the 83 counties in Michigan (source: Michigan Center for Geographic Information[3]). The voting data was preprocessed by aggregating about 60MB of precinct-level voting data using MySQL, then storing the results for each race into separate files totalling about 160KB. The county shapefiles loaded into the visualization take up about 3MB. (Although it is possible to visualize the full 60MB data set directly, precinct-level maps are not available.)

Users start by selecting a race from a table view that displays summary information for each race, including total votes and winning candidate/party. Users can then select subsets of the candidates and counties involved in that race, in two additional table views. The candidates table view shows each candidate with their party and the votes they received. The counties table view shows the total votes cast in each county, alongside nested bar charts that display the proportion of votes cast for each candidate. A fourth table view shows a breakdown of the votes for each candidate in each county. Rapid sorting of this table (by clicking in the column headers) allows the user see ordered vote totals for specific counties, parties, or candidates.

A scatter plot shows the same county information in the form of a pannable, zoomable map; a second scatter plot acts as a map inset, showing the entire state. When users zoom in, the main map adds nested bar plots over the winning party color used to fill each county shape. A portal in the inset map indicates the region shown in the main map. Users can select counties by clicking inside the corresponding shape boundaries in the map or inset.

A pie chart grouped with three grid views shows overall and pairwise breakdown of vote percentages for the selected candidates in the selected counties. These views allow users to compare candidates in various geographic regions, as well as to explore the hypothetical effect

[2]http://www.michigan.gov/sos/
[3]http://www.mcgi.state.mi.us/mgdl/

of the absence of candidates/parties and counties on overall election results.

The `Votes v. County` scatter plot uses a logarithmic scale to show vote totals for both major and minor party candidates, from largest to smallest county vote totals. The scatter plot draws all counties but highlights only selected ones. A second scatter plot uses the same ordered, logarithmic presentation to show the number of total votes for selected counties only.

## A.4 Periodic Table of the Elements

This visualization shows the chemical elements and their physical properties. The data consists of a text file (17KB) containing the atomic number, name, symbol, crystal structure and several physical properties of each element (collated from numerous sources). Additional files contain utility data for: (1) mapping atomic numbers into positions in the periodic table; (2) looking up the units of each property. The visualization also accesses about 18K of GIF image files containing small icons which depict various crystal structures.

The first page of the visualization (figure 102) displays the well-known layout of elements (commonly referred to as the periodic table). Each element is shown in the layout as a rectangle (color-coded by family) containing the atomic number, symbol, crystal structure (icon), property value, and color-mapped property value (triangle). Users can choose the displayed property in a combo box. Selecting families and crystal structures in the two lists cause the layout view to restrict display of the relevant information to conforming elements. The table view displays information for elements in the selected range of atomic numbers, sorted on the selected property. Users can override the ordering by clicking in one or more column headers.

Users can select elements (here, tungsten) for highlighting throughout the visualization by clicking rectangles in the layout. Whenever the mouse is inside one of the rectangles, a summary of the properties of the corresponding element appears above the transition metals,

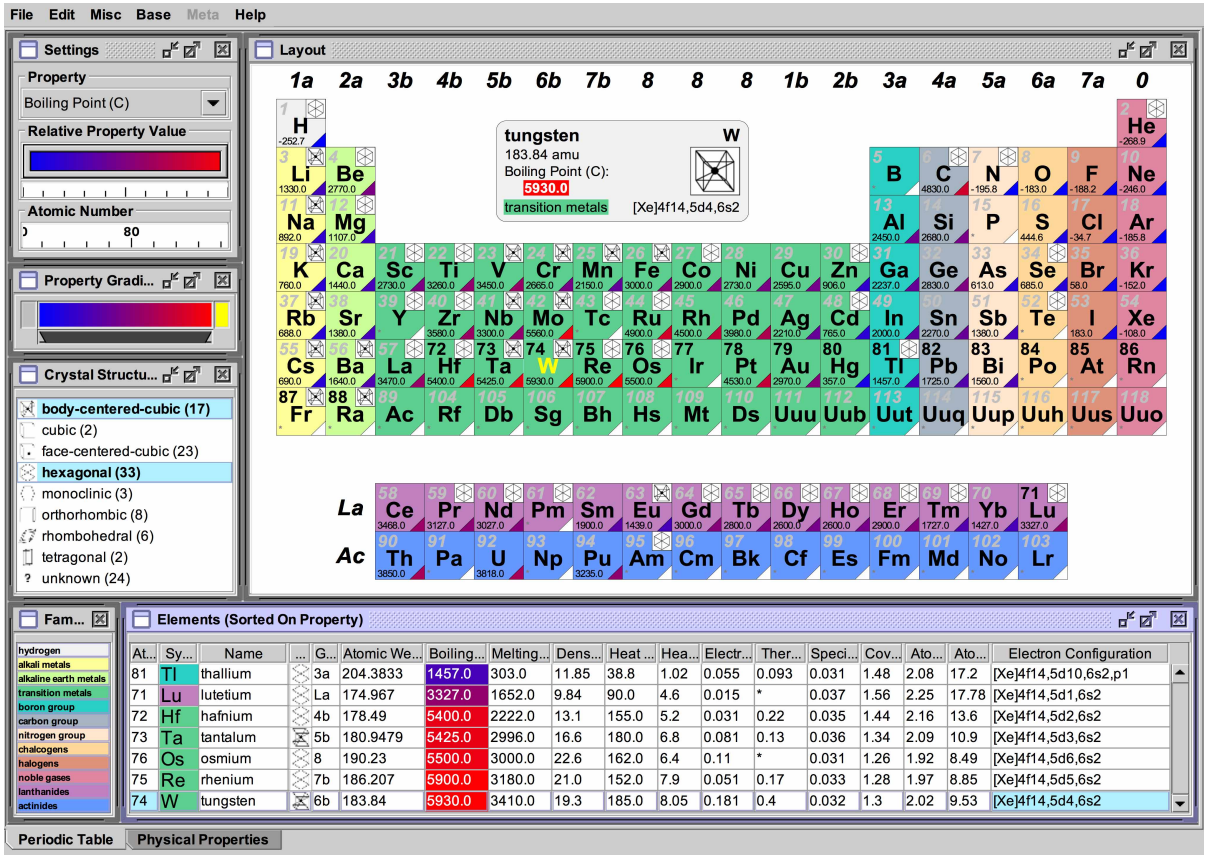



Figure 102: Periodic table of the elements (`elements.viz`, first page)

and the element's symbol is highlighted. (Different highlight colors are used to differentiate selected elements from the element the mouse is over at any given time.)

The second page of the visualization (figure 103) allows comparison of physical properties in various scatter plots and corresponding parallel coordinate plots. A stack of four views plot six properties against atomic number. The bottom two plots have two layers that show heat of vaporization (blue) with heat of fusion (red) and boiling point (blue) with melting point (red). A header plot shows the family of each element as text and a color-coded rectangle. Horizontal interaction in any of these views changes the range of atomic numbers shown in all of them; when the user zooms out enough, the views show each element as simple rectangles rather than each element's symbol.

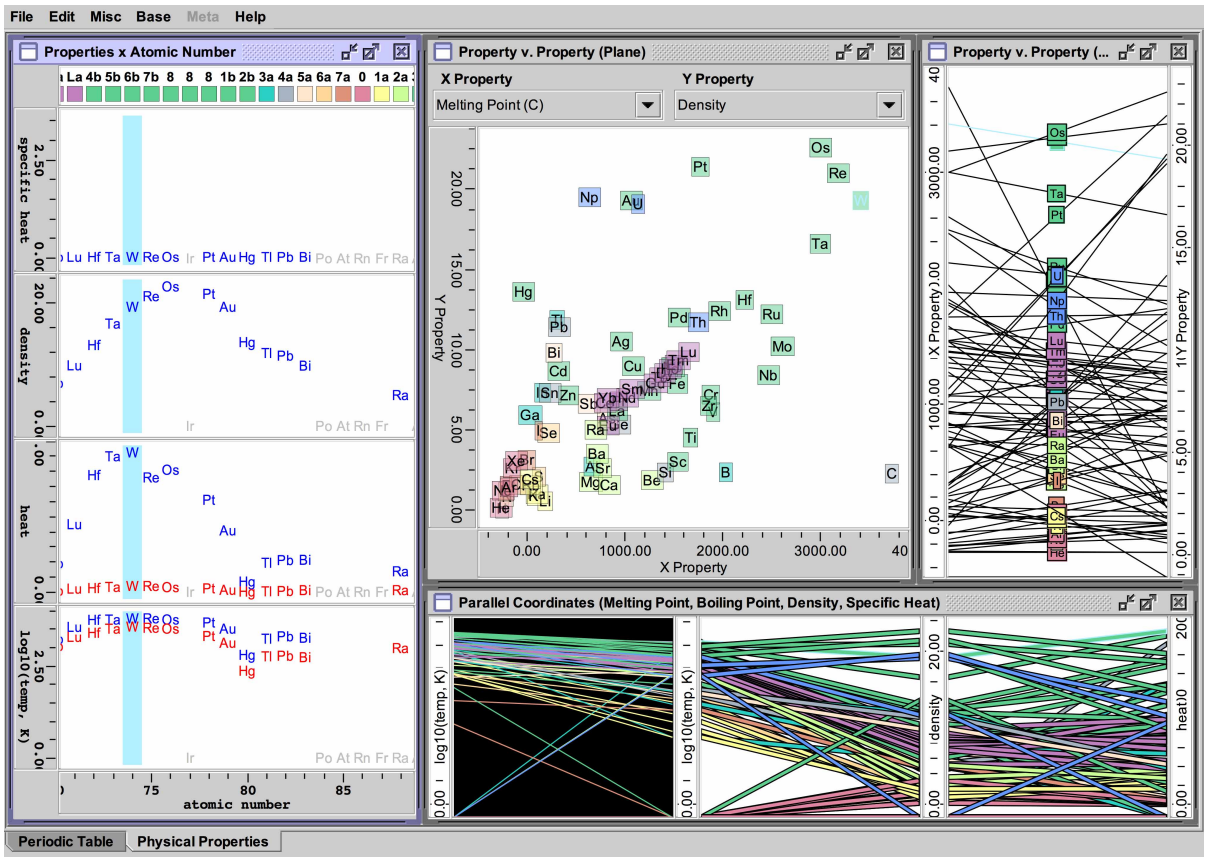



Figure 103: Periodic table of the elements (`elements.viz`, second page)

The `Property v. Property` group of views contains two combo boxes for the user to select properties to plot against each other. Each element is visually encoded as a textbox using the values of the two properties to determine symbol location and the family to determine box color. The same encoding is used to label lines in the equivalent parallel coordinate plot.

## A.5 Michigan County Roads and Hydrography

Figure 104 shows a visualization of the counties, cities, roads, and hydrological features of the 83 counties in Michigan. The data consists of shapefiles and DBF database files (totalling approximately 575MB) obtained from the Michigan Center for Geographic Information[4].

[4]http://www.mcgi.state.mi.us/mgdl/

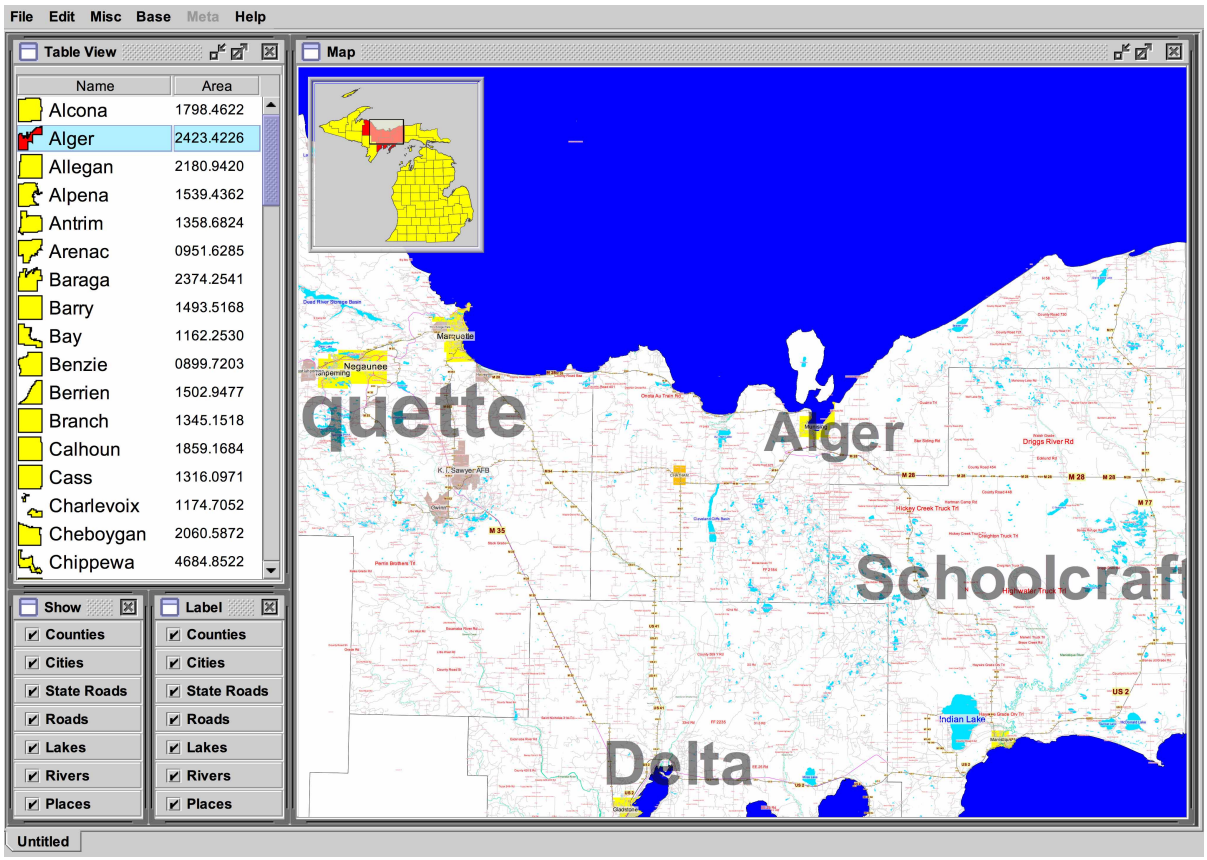



Figure 104: Michigan roads and hydrographic features (`map.viz`)

The main map is a pannable, zoomable scatter plot, inset with a second scatter plot that draws only the outline of each county. The main map draws cities, roads, lakes and rivers for the selected counties. The user can select counties by clicking inside the corresponding region boundary in the map or inset.

The table view displays the outline, name, and land area of each county. To view features, users select one or more counties in the table view, then pick which features to show (and possibly label) using the checkboxes. To avoid crowding labels, the projection used to render the main map only draws labels when the corresponding features are sufficiently large on the screen. All labels are scaled to the size of the corresponding geographic feature (e.g. area for lakes, length for roads).

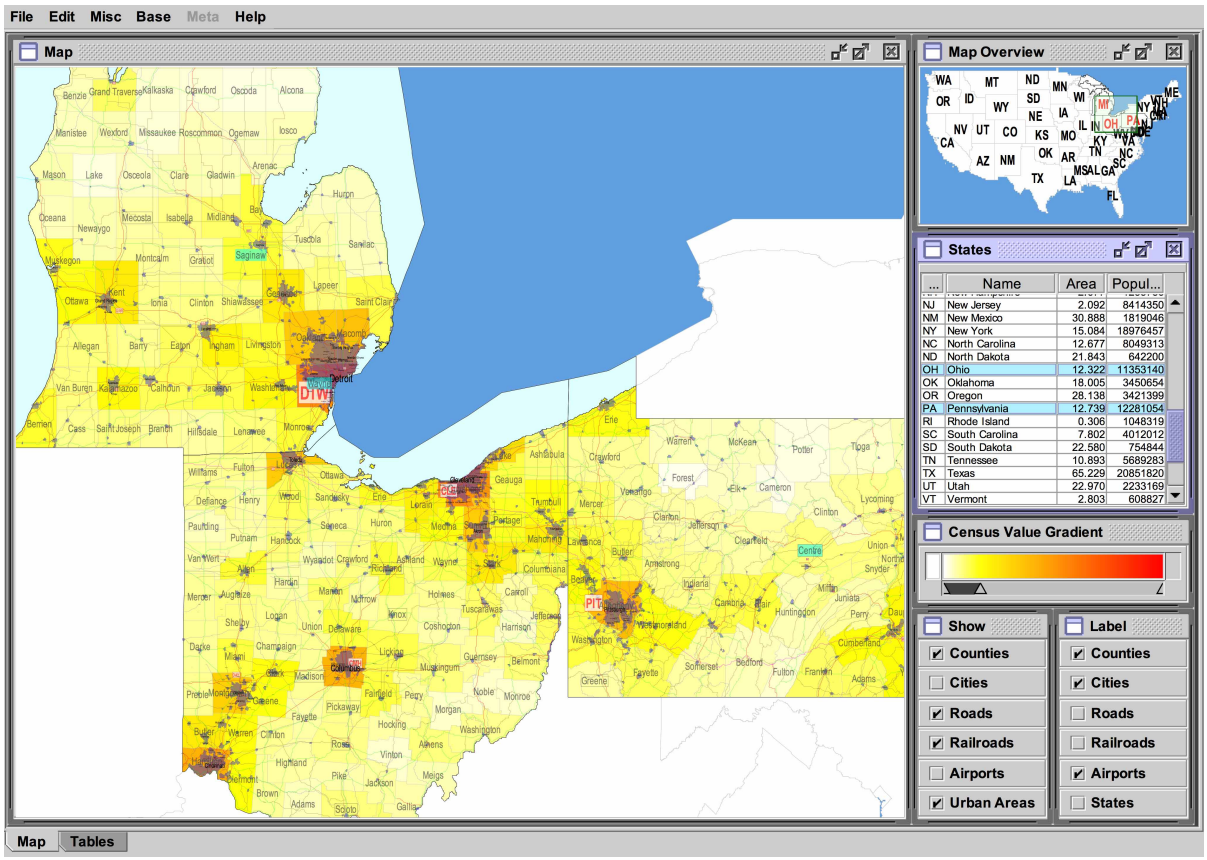



Figure 105: Year 2000 population density by county (`census.viz`, first page)

## A.6 2000 United States Census

This geovisualization shows county-level demographic data from the 2000 United States Census. The data consists of shapefiles and DBF database files (totalling approximately 20MB zipped) obtained from the National Atlas[5], describing states, counties, cities, urban areas, roads, railroads, and airports. Demographics for each county include population, density, genders, median age, percent change since 1990, and proportions of major ethnic groups.

The first page of the visualization (figure 105) displays overview and detail maps alongside a list of states and territories. The detail map draws and labels natural and artificial areas of selected states, as determined by the checkboxes. The color used to fill each county is

[5]http://www.nationalatlas.gov/atlasftp.html

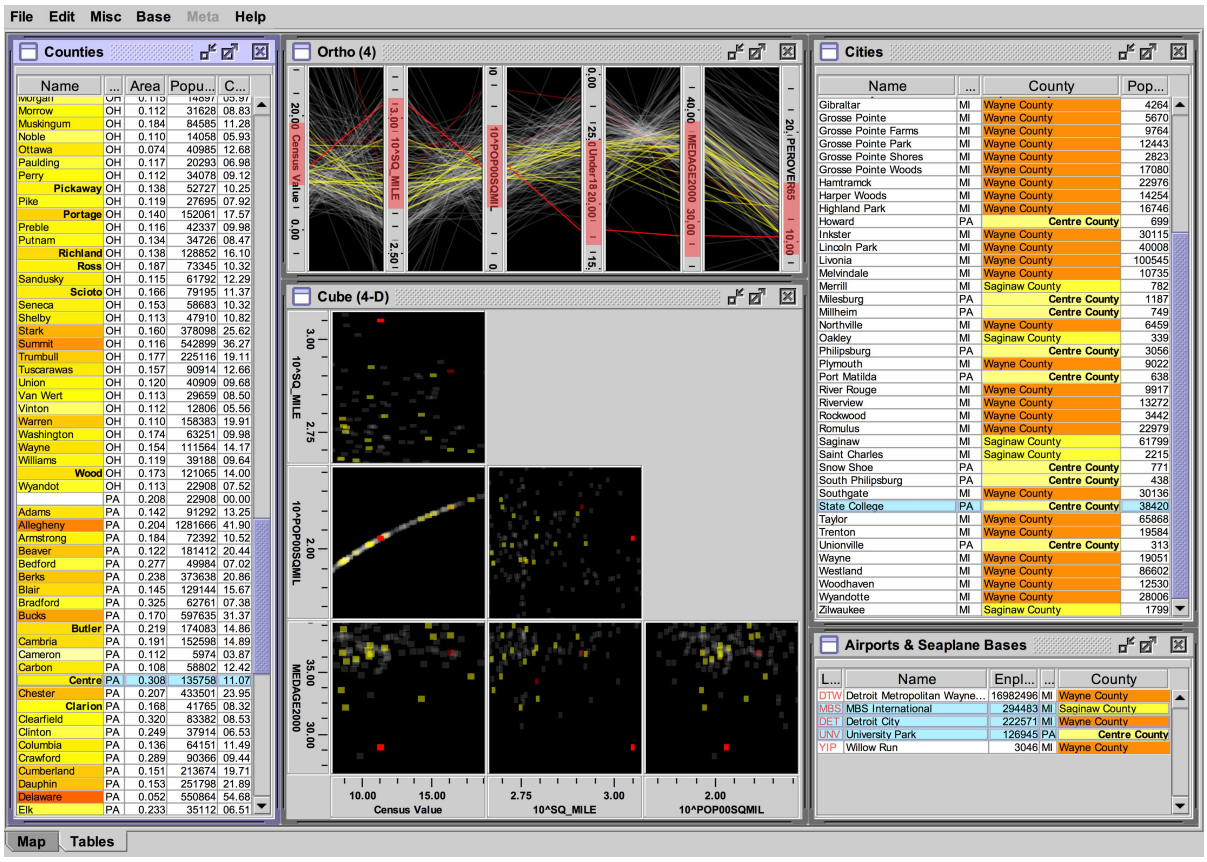



Figure 106: County and city population density (`census.viz`, second page)

calculated by mapping a function of county-level demographic attributes—in this case, the square root of population density—into a color gradient. Users can change the color gradient and edit the demographic function dynamically (the latter by modifying the corresponding expression in the lexicon editor).

The second page of the visualization (figure 106) allows comparison of the demographic function with population density and other demographic attributes, using parallel coordinate plots and the corresponding scatter plot matrix. A table view lists the counties of the selected states, using the same color-coding as the detail map. Two more table views list the cities and airports in selected counties. Users can drag and stretch portals (shown as translucent red rectangles) in the parallel coordinate axes to highlight certain counties in all views. Counties

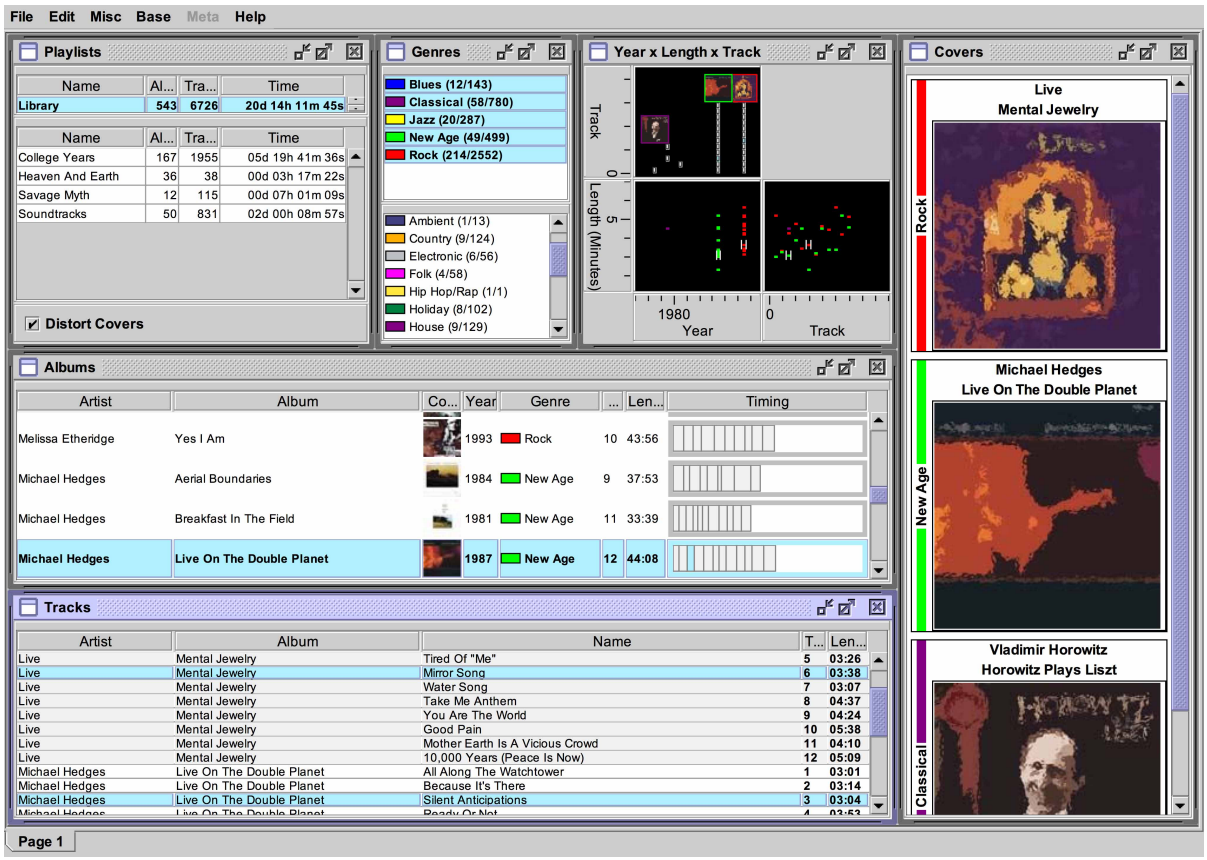



Figure 107: iTunes playlists (`music.viz`)

for which all demographic values fall within the corresponding portal ranges are shown as bright shapes (in plots) or right-justified, boldface names (in tables). In plots, counties inside all portal ranges are shown in red; counties inside at least one range are shown in yellow.

## A.7 iTunes Playlists

Figure 107 shows a visualization of a music collection. The data consists of the ID3 tags of music tracks in several playlists, generated using the *Export Playlist* feature in iTunes[6]. The visualization also accesses (intentionally distorted versions of) cover art scanned from the booklets of the compact discs that make up my personal collection. The example shown here

[6]http://www.apple.com/itunes/

visualizes just under 10000 tracks in five playlists (about 3MB total) with about 500 JPEG album covers (about 26MB total).

Two table views summarize the entire library and individual playlists, including the number of albums, number of tracks, and total play time. When users select one of the playlists, the visualization performs grouping queries to determine the genres and albums represented in that playlist. Two list views display each genre with its color-coding and the number of albums/tracks of that genre in the selected playlist; selected genres appear in the top list, unselected genres in the bottom list.

The albums table view lists the artist, name, cover, year, genre, track count, length, and timing breakdown of each album in the playlist. The timing breakdown is a nested strip view that shows the scaled length of tracks in the order in which they appear on the album.

When users select one or more albums in the album table view, the large versions of their covers are shown in the covers list view. The tracks on the selected albums are detailed in the tracks table view. Of particular interest is how the tracks selected in this view are also highlighted in the timing breakdown column of the album table view.

A scatter plot matrix of the year, length, and positions of tracks on selected albums allows the user to compare albums spatially. Text in the year, length, and track number columns of the album and track table views is drawn in bold for values visible in all three scatter plots.

Figure 108 shows a visualization of the same music collection, aggregated into albums. A 3-D scatter plot matrix (with overview matrix) draws rectangles for the year, length, and number of tracks of each album, color-coded on genre. The decades table summarizes the albums from each decade. The genres table summarizes the albums of each genre. Nested scatter plots in both tables show the distribution of time versus number of tracks for each particular decade/genre. The albums table summarizes albums of selected decades and genres. All three tables are filtered to show only the albums visible in the scatter plot matrix.

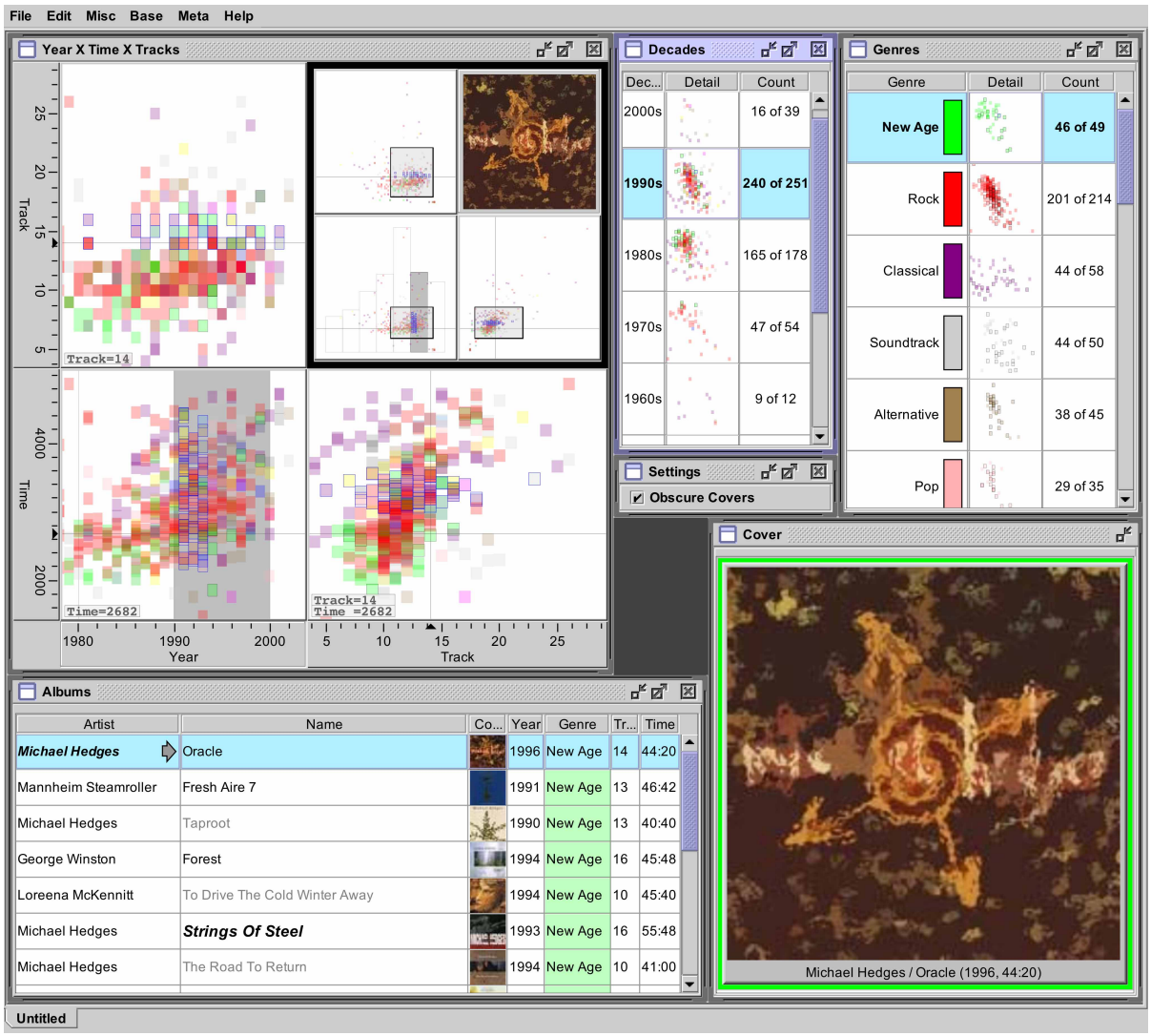



Figure 108: Music albums with cover art (`album.viz`)

In all scatter plots, albums of the selected decades and genres are highlighted by drawing the sides of the corresponding rectangles in black. Albums selected in the albums table are filled with cyan rather than the album's genre color. As the user moves the portals in the three overview scatter plots over albums, their names are displayed differently in the albums table: bold-italic for albums in all three portals, normal for albums in at least one portal, and gray for albums outside the bounds of all three portals.

Moving the mouse over the three detail scatter plots causes the frontmost album under the

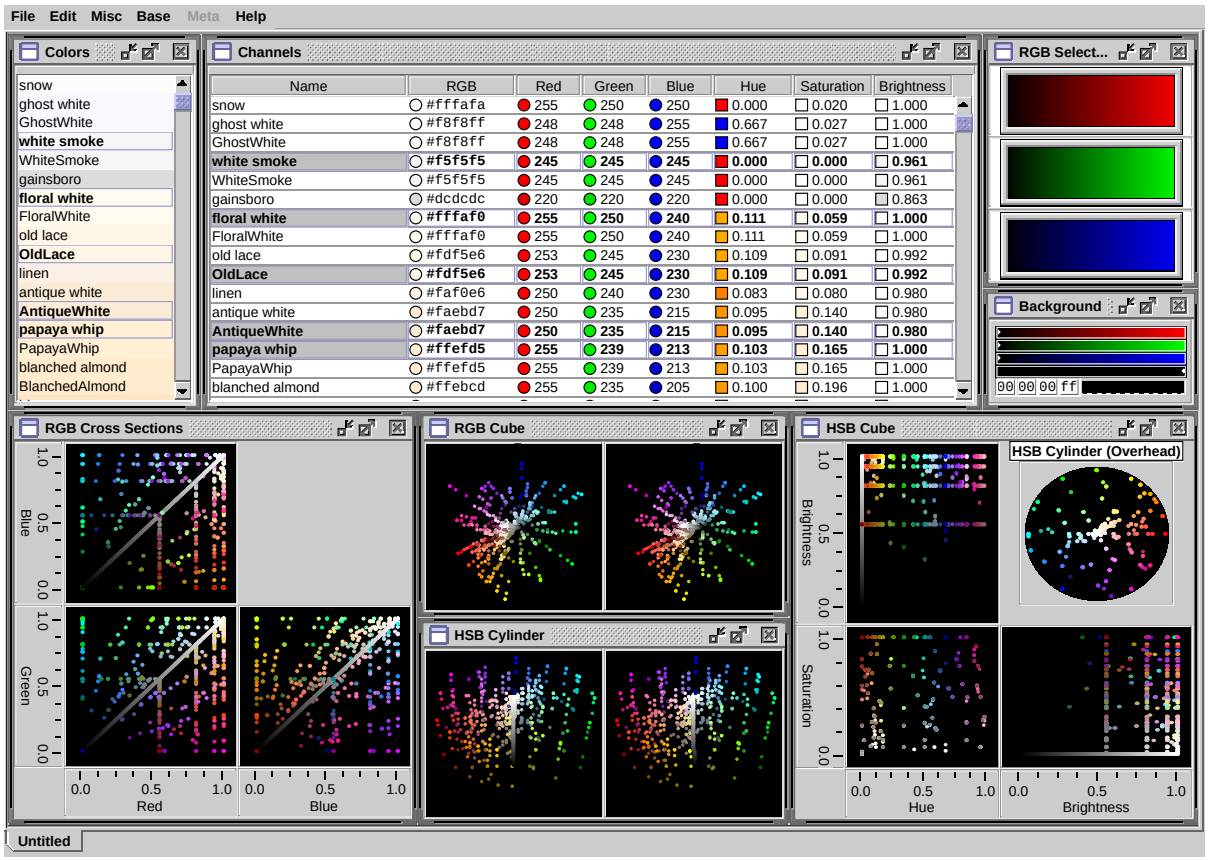



Figure 109: Colors in X Windows (`xrgb.viz`)

mouse to be displayed in the cover view. The album table is sorted in increasing order of distance from the `(year, time, track)` coordinates of the mouse.

## A.8 X Windows Color Table

The visualization in figure 109 shows the standard table of RGB colors in X Windows. The data is taken from the file `/usr/X11R6/lib/X11/rgb.txt` on systems with an X11 installation.

The main view in the visualization is a table that lists each color with its name, hexadecimal RGB value ($\#RRGGBB$), and integer values of its red, green, blue, hue, saturation,

and brightness color channels. Each cell in the last seven columns of the table fills small circles/squares with the color appropriate to the color or color channel value shown in it.

Two scatter plot matrices display the RGB color cube and HSB color cylinder by drawing colored points at positions corresponding to normalized color channel values, e.g. red is drawn at $RGB = \{1.0, 0.0, 0.0\}$, $HSB = \{0.0, 1.0, 1.0\}$. Two left-right stereogram pairs display the RGB cube and HSB cylinder similarly.

Three sliders let the user select ranges on the red, green, and blue color channels independently. All other views filter their contents to show only colors in the selected ranges. The background color control allows the user to pick the background color shown in the scatter plots and stereo views, in order to make certain color points easier to see.

## A.9 DEVise Layout and Coordination

DEVise[7] uses a relational data model to coordinate multiple views of large data sets. Users can create, destroy, coordinate, and specify the contents of views interactively. Its only view—the scatter plot—and few coordination types—*cursor*, *visual link*, *record link*, and *set link*—are quite powerful. However, reproducing common visualization constructions in DEVise frequently involves convoluted chains of linked scatter plots (many of which are undesirable artifacts that must be intentionally hidden off screen). Coordination graphs of DEVise visualizations reveal that all four coordination types can be reproduced by treating the `X` and `Y` ranges of scatter plots as shared objects or as dynamic parameters in simple query expressions. This discovery motivated the design of Live Properties and Coordinated Queries in Improvise. Incremental development of the two visualizations shown here was part of this discovery process.

[7] http://www.cs.wisc.edu/~devise/

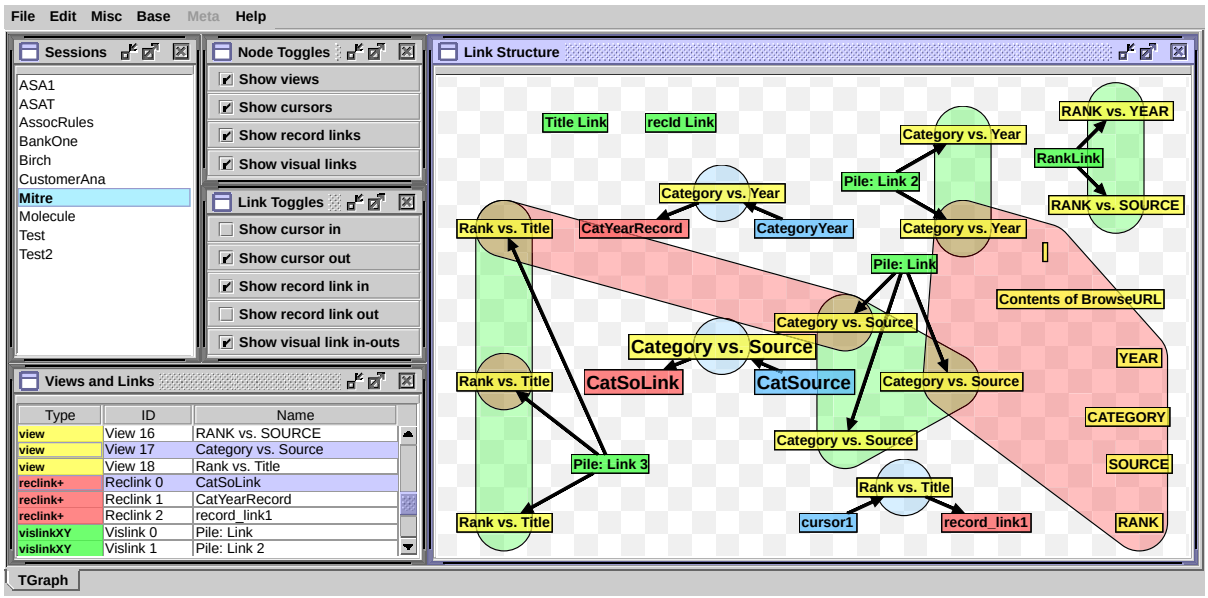



Figure 110: Coordination graphs of DEVise visualizations (`tgraph.viz`)

The visualization in figure 110 shows the coordination structure of DEVise visualizations (called *sessions*). The data is generated using a feature in DEVise that allows users to export a text description of the views and links of a session. Users start by selecting a session from the list. The table view shows all the views and links in the session, color-coded on type. The graph view displays a coordination graph of the session, showing views and links as nodes and the dependencies between them as edges. Packs surround groups of nodes that represent views which mutually depend on a particular link. Checkboxes let users pick which kinds of nodes and edges to show in the graph.

The visualization in figure 111 shows the screen layout and coordination structure of sessions. The data consists of session descriptions created using the *Session Dump* feature in DEVise. Users start by selecting a session from the list. The list displays the name, the number of views and links, and a miniature screen layout of each session. The layout of the selected session is shown in a large scatter plot with a portal lens that labels each of the DEVise views. Two smaller scatter plots, showing the same screen subregion as the portal, display how links

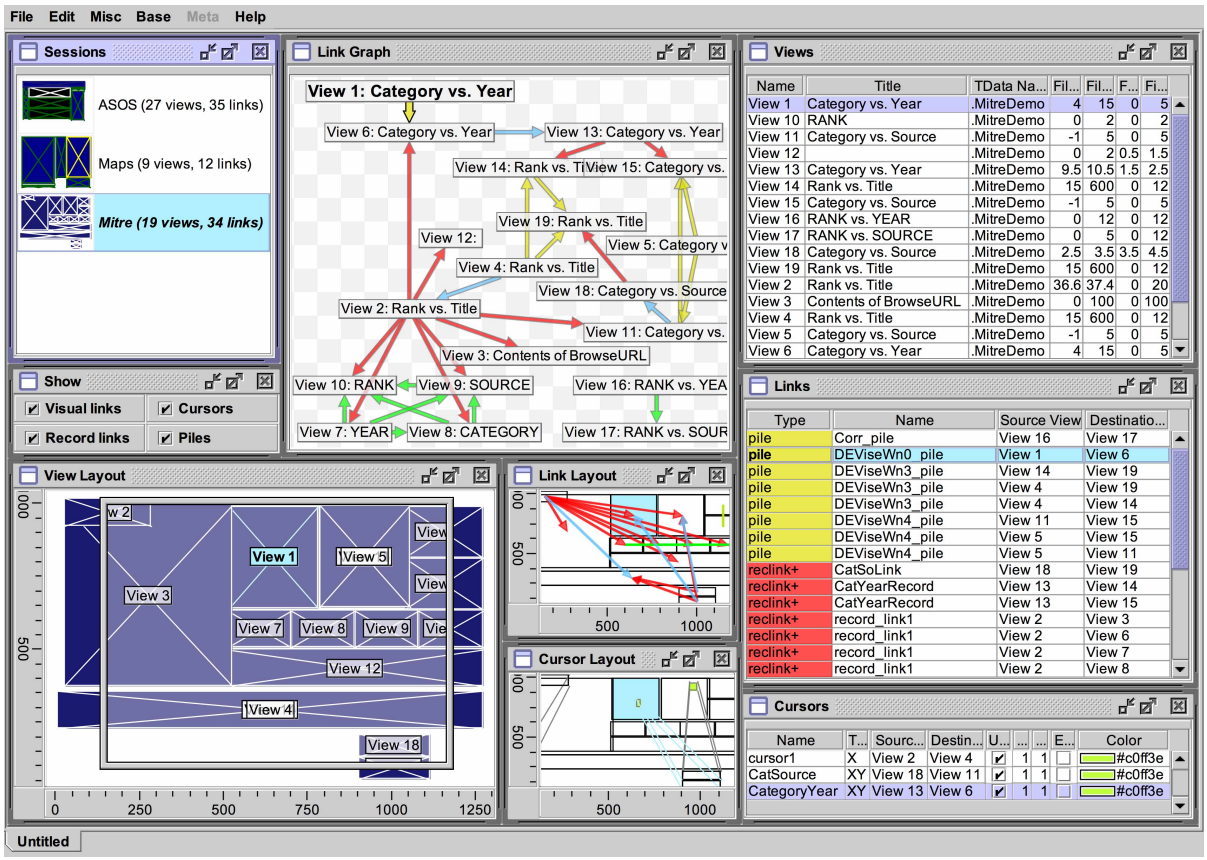



Figure 111: Screen layouts and coordination graphs of DEVise visualizations (`devise.viz`)

and cursors coordinate views in the layout.

The graph view displays a coordination graph of the session, showing DEVise views as nodes and the links that connect them as color-coded edges. Three table views summarize the views, links, and cursors in the session. Records representing DEVise views can be selected (and are highlighted) in the three scatter plots, the graph view, and the table views. Records representing DEVise links can be selected (and are shown highlighted or as fatter arrows) in these views. The checkboxes allow the user to select which types of DEVise links are visible in the smaller scatter plots, the graph view, and the links table view.

## A.10 Summary

Improvise enables interactive construction like other visualization systems, but with substantially increased flexibility that would otherwise require custom programming. The growing use of multiple coordinated views in information visualization systems has been driven by the increasingly sophisticated exploration needs of knowledge domain experts. These examples demonstrate how Improvise can be applied to meet these needs in a variety of knowledge domains.