iVisDesigner: Expressive Interactive Design of Information Visualizations

Donghao Ren, Tobias Höllerer and Xiaoru Yuan

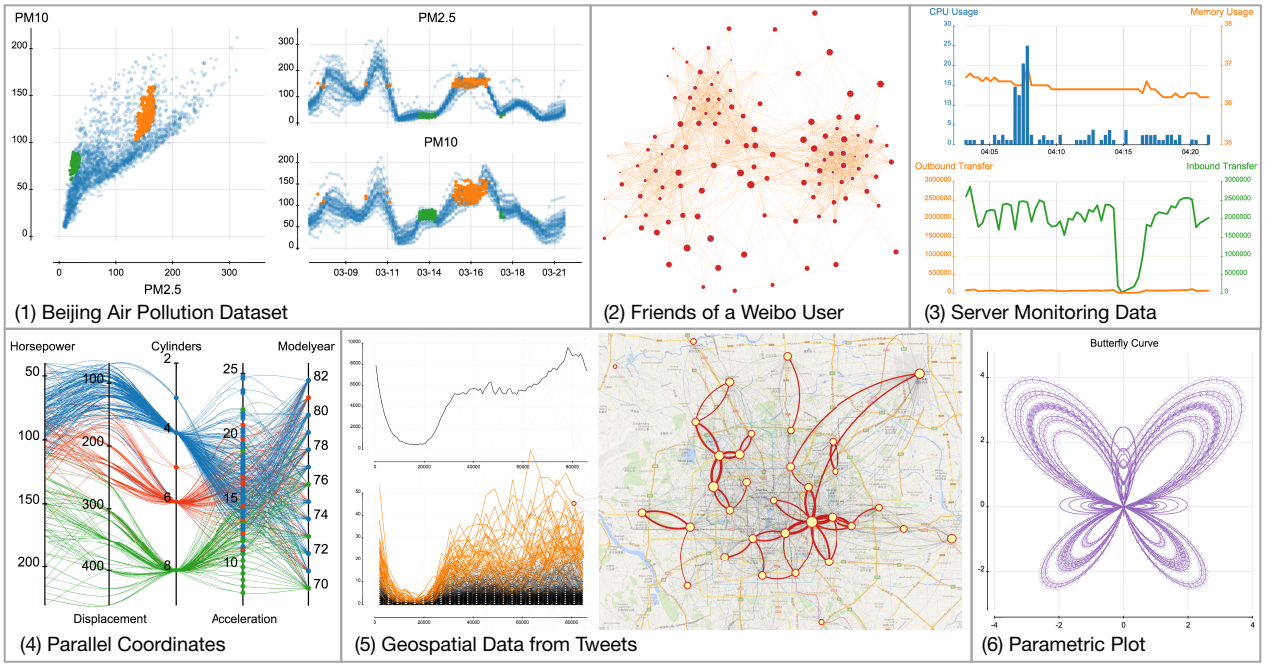

Fig. 1. Example visualization designs created in iVisDesigner.

Abstract— We present the design, implementation and evaluation of iVisDesigner, a web-based system that enables users to design information visualizations for complex datasets interactively, without the need for textual programming. Our system achieves high interactive expressiveness through conceptual modularity, covering a broad information visualization design space. iVisDesigner supports the interactive design of interactive visualizations, such as provisioning for responsive graph layouts and different types of brushing and linking interactions. We present the system design and implementation, exemplify it through a variety of illustrative visualization designs and discuss its limitations. A performance analysis and an informal user study are presented to evaluate the system.

Index Terms—Visualization design, Interactive Design, Interaction, Expressiveness, Web-based visualization.

1 Introduction

Programming frameworks for information visualization such as Processing [2], Prefuse [21], ProtoVis [20] or D3.js [7] provide very useful abstractions for visualization designs that make programming easier and more elegant. Frameworks can utilize existing programming languages, such as Javascript in the case of D3.js, or new programming languages, as in the case of Processing. However, they all require textual programming, which limits use to a population of coders, or otherwise imposes a fairly steep learning curve. Most of these frameworks require iterating back and forth between programming and execution stage, and thus adjustment of parameters can be cumbersome.

On the other hand, some information visualization toolkits support interactive ways of creating visual designs [17], from early pipeline-based systems [32] to more unified approaches [10, 18, 28]. These systems are easy to use, and generally offer a What You See Is What You Get editing experience, which greatly assists parameter tuning. However, compared to textual programming, they are generally less expressive. For example, the Flexible Linked Axes toolkit [10], which inspired parts of our visualization functionality, only covers axis-based designs for multidimensional data visualization, arguably a small portion of the whole design space. There is a need for highly flexible toolkits that support a wide spectrum of visualization designs.

In this paper, we present iVisDesigner, a web-based system that enables users to design information visualizations for heterogeneous datasets interactively, without the need for textual programming. Compared with other approaches such as Flexible Linked Axes [10] and Gold [25], and commercial software such as Microsoft Excel and Tableau, iVisDesigner focuses on expressiveness and modular visual-

- *Donghao Ren is with the Department of Computer Science, University of California, Santa Barbara and was formerly with the School of EECS, Peking University. E-mail: donghaoren@cs.ucsb.edu.*
- *Tobias Höllerer is with the Department of Computer Science, University of California, Santa Barbara. E-mail: holl@cs.ucsb.edu.*
- *Xiaoru Yuan is with the Key Laboratory of Machine Perception (Ministry of Education) and School of EECS, Peking University. E-mail: xiaoru.yuan@pku.edu.cn.*

Manuscript received 31 Mar. 2014; accepted 1 Aug. 2014; date of publication xx xxx 2014; date of current version xx xxx 2014.
For information on obtaining reprints of this article, please send e-mail to: tvcg@computer.org.

ization design flexibility, covering a wider range of the information visualization design space.

Expressiveness in iVisDesigner is supported by its underlying framework and user interaction provisions. The framework utilizes a flexible internal representation of visualization designs, which is carefully exposed in a unified user interface. Users specify designs via a mouse or pen-based user interface in a web browser, utilizing drag and drop, sketching, and context menu elements. The system supports visual analytics tasks, such as brushing and linking, and visualization customizations, both during visualization design and interactive exploration of data in completed designs. Users can embed the designed visualizations into existing websites or web-based applications by inserting a piece of Javascript provided by iVisDesigner.

The main contribution of this work is an expressive framework to represent visualization designs for different types of data, allowing users to interactively arrange visual elements in different ways, combining and linking different types of visualizations. We discuss our design decisions, implementation choices, as well as system limitations, and demonstrate the expressiveness of our system by presenting a variety of example applications on different types of datasets. We provide evaluation of our system in form of a performance analysis and an informal user study. Our prototype system exhibits high expressiveness compared with existing systems, while maintaining good performance and usability.

The paper is organized as follows: After discussing related work, we present the design of the framework and user interaction, followed by notable implementation details. Next we discuss example applications to exemplify coverage of the information visualization design space. We then present system evaluation in the form of performance measurements and an informal user study. Finally, we discuss overall results and limitations and conclude the paper with an outlook on future work.

2 Related Work

Our work builds on top of a rich research landscape for information visualization toolkits and systems. We structure our overview of related work by separating discussion of influential theoretical background, programming frameworks, toolkits, and systems that are related to our work.

Theory: Any expressive system facilitating flexible mappings of data to visual variables owes a lot to the semiological research of Bertin [4, 5]. Mackinlay [24] provided further foundation and presented automated tools for powerful modular visualization design. Shneiderman [30] analyzed the data types and tasks in information visualization, and presented a taxonomy for them. Card et al. [8] organized previous visualization designs, and presented categorization of data types and visual mappings. In our demonstrations and evaluations of iVisDesigner, we will highlight its coverage of the information visualization design space.

Data Flow Systems such as ConMan [19], AVS [32], IRIS Explorer [15] and VisTrails [3] use a pipelined approach and flow diagrams to represent the progress from data to visualization. These systems are particularly good at defining data transformations, but not very flexible for defining interlinked mappings from data to graphical elements. In addition, pipeline-based systems focus on representing the pipeline itself, there is little screen estate left for displaying and editing the visualizations. Our work provides an integrated representation and manipulation of graphical mappings, with all graphical elements created, presented and edited in the same canvas which dominates the user interface, allowing for better understanding of the overall visualization design.

Programming Frameworks and Languages: Drawing APIs and toolkits such as OpenGL, Java2D, HTML5 Canvas, and Processing [2] define programming interfaces to draw low-level elements. Even for experienced programmers, creating visualizations with these APIs directly is not straightforward. Thus visualization frameworks have been created for better abstraction. The InfoVis Toolkit [13] and Improvise [36] provide a set of basic widgets. Chi et al. [9] proposed a spreadsheet approach. "behaviorism" [14] uses three graphs to represent dynamic visualizations. To create novel designs, users need to create new widgets or inherit existing ones. Heer et al. proposed Prefuse [21, 1], a toolkit for interactive visualization. It first transforms abstract data into visualizable form by a *filtering* process, and renders the visualizable form by using a *view* process. It allows for advanced integration of existing operators to create novel techniques, but typically users will need to define new operators in the process.

Declarative models and languages for information visualization have been presented [38, 37]. Protovis [20, 6] provides a declarative language for information visualization, designed and implemented in Java and Javascript. Bostock et al. designed D3.js [7], a Javascript library for creating web-based visualization designs. It facilitates the manipulation of DOM elements with data. All of these programming frameworks require users to write programs to combine visualization components. The framework of our system is similar to D3. Both operate by defining and parameterizing mappings from data items to graphical elements. D3 takes a programming-oriented approach, while our system takes an interactive design approach: users create visualization designs, and provision for end-user interaction via the web-based user interface.

Interactive Toolkits: Vector-based drawing software, such as Adobe Illustrator, is widely used for graphical design, and we drew some inspiration for our user interface from such products. While it is possible to create visualizations with such design tools, there is no support for parameterizable mapping from data to graphics. Graphical items have to be created individually. On the other hand, a lot of commercial software has the functionality to create visualization designs for data interactively, for example the chart feature in Microsoft Excel and similar spreadsheet software. Web-based systems such as ManyEyes [34], Sense.us [22], or CommentSpace [39] focus on collaborative visual analysis. Most of these systems are focused on using and customizing several predefined templates.

Tableau is a highly sophisticated state-of-the-art commercial visualization system, providing good flexibility for visualization designs. However, it is still predominately template-based, which increases ease of use for beginners but limits free-style design explorations. In contrast, we focus on more fine-grained control and flexible combination of graphical elements in the pursuit of novel design combinations.

For multivariate data, Claessen et al. [10] allow users to position axes and put scatterplots and links between them interactively. However, it only supports multivariate data and axis-based visualization designs. Sketch-based interactions, like in SketchStory [23] and SketchInsight [35] have been explored. While sketch-based interactions are very intuitive, these systems currently only support a very limited set of designs. Bret Victor presented a tool [33] that allows users to define drawing procedures with geometrical constraints, which are parametrized in an interactive canvas. It requires procedural thinking, where users define loops to draw simple visualizations such as a scatterplot. In our system, we use a declarative approach, where loops are defined implicitly by *data selectors*. SageBrush [27] uses "partial prototypes" to define spatial properties for "graphemes", and supports editing of primitive properties. Our system expands on this theme by enabling data transformation and generation, and supporting interaction with designed visualizations. Lyra [28] is a very recent addition to the interactive visualization design landscape. Based on the JSON-based declarative visualization grammar Vega [31], it allows users to define visualizations interactively by constructing scales, guides and marks. Sophisticated layouts and transformations are enabled via transformation pipelines. Lyra and Vega only operate on tabular datasets, while our work also supports hierarchical datasets with a fixed schema and references between data items. Lyra is more oriented towards designing a single piece of visualization, while our system focuses on canvases that allow users to draw and link different designs. Furthermore, our system supports designing interactions such as brushing and linking.

3 Design Philosophy

The framework of iVisDesigner is designed to represent visualizations that support interactive user manipulation, all within a web-based in-

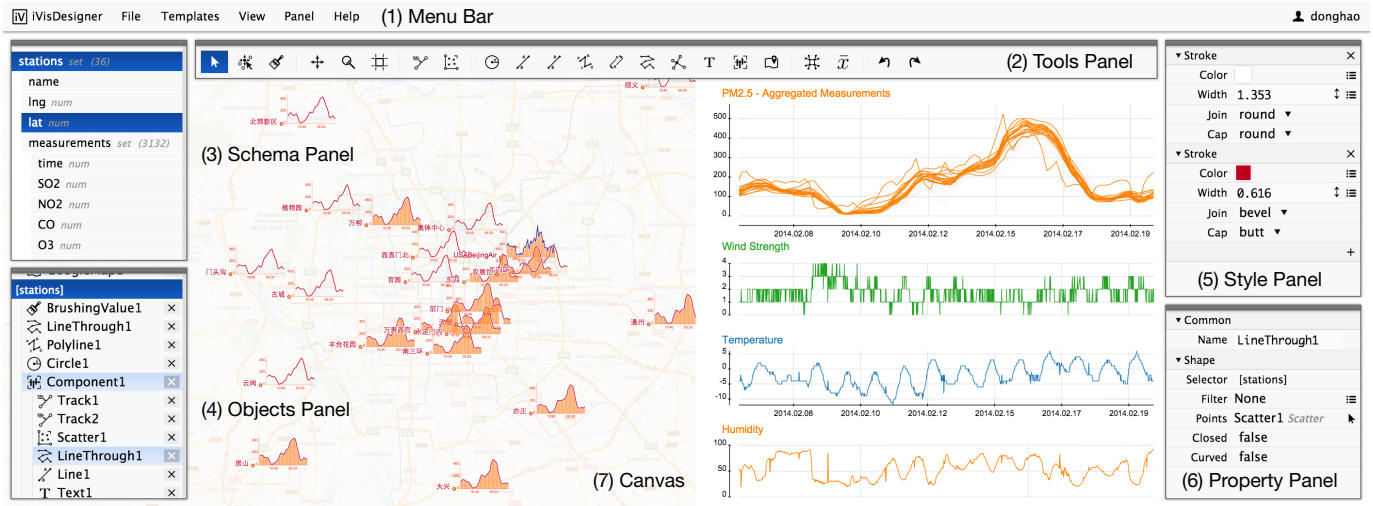

Fig. 2. The interface of iVisDesigner. (1) Menu Bar: Commands for loading or saving visualization designs, view settings, login and logout. (2) Tools Panel: Tools for moving objects around, creating new objects and changing the view. Grouped into different categories. (3) Schema Panel: Shows the structure of the dataset. Allows selection. (4) Objects Panel: Shows the objects currently in the visualization design. Allows selection. (5) Style Panel: Adjust graphical styles for currently selected objects. (6) Property Panel: Adjust properties of currently selected objects. (7) The canvas to draw the visualization. In this example, a visualization of Beijing Air Pollution data is presented. There are two linked views, the left view shows the timeline plots of PM2.5 indexes for each measurement station on top of a map, the right plots shows the trends of the PM2.5 indexes, wind strength, temperature and humidity. This visualization is designed solely through user interaction with iVisDesigner, without textual programming.

terface and canvas. The high-level design choices of iVisDesigner are based on the following idea: Allowing for interactive visualization creation, editing and interaction in an unified interface, where the space is dominated by a canvas showing the emerging visualization. We focus on enabling users to freely place graphical elements and links between them, instead of simply designing a chart or template visualization.

Our overall approach can be characterized as introducing support for data influx and manipulation to the common usage paradigm of interactive vector-based drawing software. By allowing users to define mappings from data to graphical elements, we enable them to directly create and manipulate groups of elements simultaneously, which tremendously reduces the amount of work to create visualizations. Transformed, aggregated, or otherwise generated data can be attached to the dataset, providing more capabilities, such as histograms and graph layouts. Graphical elements can be manipulated via dragging and brushing, which affects the underlying data attributes, enabling the design of interactive visualizations.

4 Design

In this section, we present the workflow and design details of iVisDesigner. We first give an overview of our framework, which is designed to represent visualizations, render them, and allow users to manipulate them interactively. We then discuss our user interface design, explaining how users can create and edit the different components.

4.1 Framework

The framework of our system is illustrated in Figure 3. Data is loaded from the *Data Source*, transformed into an internal representation, and then enumerated or decomposed into individual elements by various *Data Selectors*. The decomposed elements are then passed into *Objects* for visual mapping or data generation. For visual mapping, the resulting graphical elements are simply rendered on the canvas. For data generation, the results are additional data attributes (e.g., a histogram) that can be attached back to the data representation. In addition, users can create and attach new data from scratch, for example, creating a range of integers from 1 to 100 for numbering purposes. Multiple mappings and transformations can co-exist in the same visualization, and can refer to each other.

Our data representation is based on a hierarchical model similar to JSON, but allows for references among objects. Data items are a set of

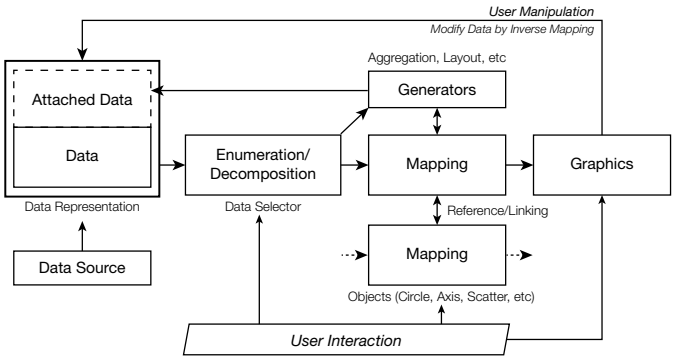

Fig. 3. The framework of iVisDesigner. Data is first being enumerated by user-defined data selectors, and then passed to different mapping objects. These objects may refer to each other, and finally generate the graphics or attach generated data back to the data representation. Users can also manipulate the graphics, and modify the underlying data if permitted.

key-value pairs. Each value can be a single data item, an array of data items, a primitive value (number or string), or a reference to another data item. The structure of the dataset is defined by a fixed *Schema*, which stores a definition of the data structure. This requires the items in a single array to be homogeneous, i.e. they must be of the same type and structure. This ensures that we can perform mappings from each array of objects to graphical elements in a unified way. Given that the arrays in a given dataset are homogeneous, it is easy to construct a schema for a dataset automatically.

This dataset definition is relatively more expressive than tabular structures. For example, in Figure 2, the depicted dataset contains a set of air-probing stations, each with a set of measurements, which are visualized individually on the map, and collectively on the timeline plot. Users can place small visualization designs for inner-level data within a larger plot.

Users can create *attached data* by creating special objects in the system. These can, for example, compute basic statistics or run a force-directed layout algorithm.

Data Selectors, which are automatically created from user interac-

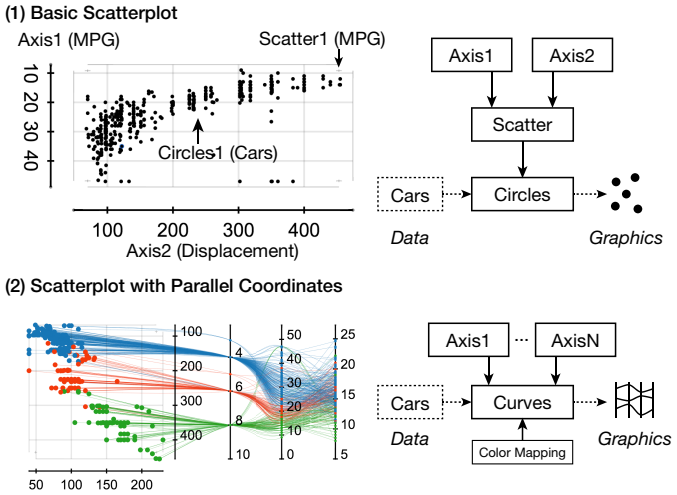

Fig. 4. Two basic visualization designs: (1) Scatterplot (2) SPPC, as in [40]. The scatterplot consists of two *Axes*, a *Scatter* and a *Circles* object, with *Axes* and *Scatter* objects providing location information for the *Circles* object, which maps cars to circles.

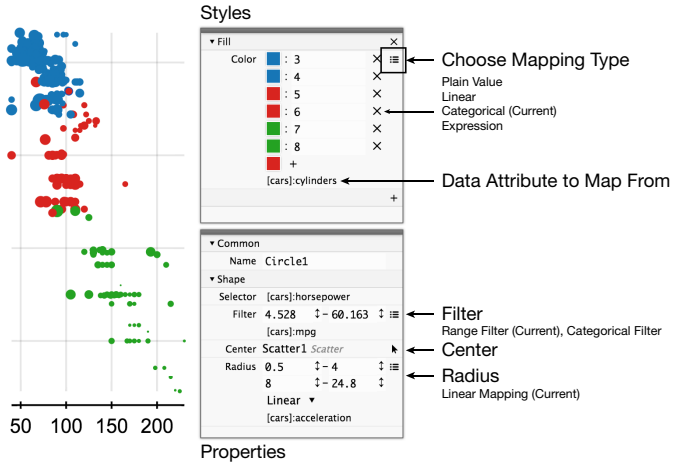

Fig. 5. The properties and styles of the objects can be defined in the corresponding panels. Styles are drawing actions, such as Fill and Stroke; properties define the shape of the object, such as Center and Radius. Both, properties and styles can be defined as plain values or mapped from data via the UI.

tion, are used to select a set of data items or values from the dataset. While Data Selectors are created via the UI (e.g. by clicking on an entry of the Schema panel or selecting from a dropdown menu in the Property panel), they also do have a syntax (string representation), where `[array]` means an enumeration of all the elements in the array, and `field` means a particular field from the current object. The Data Selectors can be specified in a path-like manner, joined by ":". For example, `[cars]:acceleration` means an enumeration of all (say n) cars in the dataset, taking the value of the acceleration field for each, resulting in an array of n numbers.

Reference fields can also be selected in the Data Selectors, for example, `[edges]:&source` will select the source nodes for the edges in a graph (nodes are stored in a separate array), and `[edges]:&source:value` will select the `value` attributes of the source nodes.

A visualization consists of a set of *Objects*, which define mappings from data to graphical elements, or generate new data attributes and attach them back to the data representation (as shown in Figure 3, Mapping and Generators). Objects in our system can be of various types, specifically, they encompass *Graphical objects*, *Guide objects*, and *Generator objects* as discussed below.

A *Graphical object* represents a mapping from a set of data items to a set of graphical elements. Examples are *Circles*, *Lines*, *Polylines*, *Arcs*, and *LineThroughs*. Each Graphical object has a Data Selector associated that specifies the set of data items to map from, and each item in this set is rendered as a graphical element.

The properties of a Graphical object, such as the location and radius for the *Circles* or end points for the *Lines*, are provided by *Guide objects*. These objects transform data values to various properties such as location, width and color. Examples are *Axes*, *Scatters*, and *Maps*.

Generator objects attach derived data to the dataset. For example, they can calculate the average value for a group of data attributes (*Statistics* object), group them into bins (*Aggregator* object), compute an expression on a set of data items (*Expression* object), or perform a force-directed layout algorithm on a graph and give each node a position (*ForceLayout* object). Generator objects mainly perform data transformations, they attach generated values to the dataset. Guide objects are displayed and edited on the canvas, and mainly deals with visual properties. There is no exclusive separation between the two categories, and one object might be of both kinds at the same time.

Generators can also accept user interaction. A *BrushingValue* object accepts brushing actions on a given visualization, and attaches corresponding data attributes to a data item that got brushed. These generated or derived values are attached to the dataset, which can then further be used in other parts of the visualization. For example, they can be used to construct brushing and linking functionality for a visualization, enabling basic visual analytics.

Objects can be nested into a *Component object*, which is bound to one Data Selector and which provides local coordinates for the objects inside it. Since our data representation allows for a hierarchical structure, users can design sub-visualizations and scatter them around. For example, one can create a component to design a small glyph for representing different data items. In section 6, we show an example that uses components to create small timeline plots for each item (cf. Figure 2), and Figure 7 showcases some custom glyphs.

Note that object types are not mutually exclusive. One object might have several types simultaneously. E.g. an *Axis* is at the same time a *Guide object* and a *Graphical object* showing tick markers. Force-directed layout is classified as a generator object, because it generates coordinates, but it can also be a guide object at the same time (users indicate a rectangle for the layout region).

Our framework can be extended via programming by defining additional custom objects, ranging from primitive graphical elements to complex visual designs (e.g. a special type of glyph).

A comprehensive list of currently supported objects and their properties, as well as the interactions to create each of them, can be found in the supplementary material.

4.2 Interface and Interaction

We now discuss the user interactions in iVisDesigner. The overall user interface is shown in Figure 2, together with an example design (detailed in subsection 6.2). It consists of a menu bar, a set of panels, a status bar, and a drawing canvas. There are five flexible panels that users can freely move around, resize, minimize or hide: The **Tools panel** presents a set of tools, including select/move, drag, brushing, pan/zoom, and object creating tools. The **Schema panel** shows the schema (hierarchical structure) of the dataset; users can select arrays or fields in the dataset. The **Object panel** shows a list of all graphical objects in the visualization and allows users to select, reorder or remove them. The **Style panel** is used to define graphical styles (series of drawing actions) for graphical objects. The **Property panel** lets users edit properties of selected objects. In addition, a data inspector panel is available, it can be shown when users need to examine the actual data values.

After an object is selected in the object panel or directly from the canvas, the property and style panels let the user edit the properties and drawing actions.

We aimed for a uniform interface, flexible for different tasks, and

intended to maximize the space for the drawing canvas, resulting in the five panels above. The tools panel helps switch between different mouse tasks (select, interact, create). The schema panel and object panel are for selecting data elements and visualization objects, while the style panel and property panel allow users to edit the selected objects in an uniform way.

In the following sections, we will discuss typical steps to use our system, including actions to create a visualization, edit an existing design, and interact with them. Since the system is flexible, users could go back and forth among these steps at will.

4.2.1 Creating Visualizations

A visualization is created by adding graphical objects to the canvas. Typical steps to create a graphical object include: (1) Selecting a desired set of data items from the schema panel. (2) Selecting the desired type of object from the tools panel. (3) Providing initial positioning. Other properties of the newly created object will be set to default values, for later adjustment.

While in other systems, such as SageBrush, Tableau or Lyra, one directly assigns data properties to marker properties, and the system will automatically determine scales and their positioning (that might be changed later), we require users to create guide objects (such as Axes, Scatters) explicitly. Since our canvas is virtually infinite, and there might be multiple existing visualization parts on it, automatically creating axes is not as straightforward. Therefore, to create a visualization, users first need to create guide objects as a frame, then add graphical elements. For example, to create a scatterplot, users need to first create two orthogonal axes, add a scatter between them, and then put, e.g., circles on the scatter.

References in the dataset can be utilized. For example, in a node-link graph visualization, the edges are defined as two references to nodes, `source` and `target`. Suppose we already visualized the nodes as a scatterplot. We now want to create lines for the edges to make a node-link visualization. The lines are bound to the `[edges]` array, so each edge is drawn as a line. To specify the two end points for each edge, we need to use the locations provided by the nodes' scatterplot. In this case, the user first highlights the "ref" button next to the `source` reference field, indicating she/he is going to use the *node referenced by that field*, and clicks on the scatterplot to specify the first end point. Then the user highlights the "ref" button beside the `target` reference field, and clicks on the scatterplot to specify the second end point and lines are created. In short, the scatterplot defines a mapping from nodes to locations, and the reference field tells the scatterplot which node to use for the mapping. This is perhaps the most difficult-to-understand interaction in our system. While users in our evaluations were able to easily follow our guidance to create node-link graphs, it proved challenging for some of them to create other forms, such as the matrix and arc-based graph visualizations in Figure 9, given just a short amount of learning time.

To reduce the burden of complex interactions to create common designs like a scatterplot, we defined a small set of templates (scatterplot, timeline plot, node-link graph), that allows users to create such visualizations simply by selecting the data attributes and dragging a rectangle on the canvas. The template will create the required graphical objects and guide objects for the user automatically, and the user can adjust the design later. This also allows novices to get started with the system more easily.

4.2.2 Editing Properties and Drawing Styles

After having created objects, users can further modify their properties. This is done via the property panel and the style panel. The property panel shows a grouped list of property-editing UI components. Most of the properties for an object can be set as mappings, such as linear mapping or categorical mapping, which are guide objects created implicitly by the property editor, allowing users to assign mappings from data attributes to actual properties of each graphical element represented by the object. For example, the "radius" property of the circles object can be assigned as a constant value, or as a linear mapping of some data attribute. The "center" property of the circles objects can be set by picking a guide object or a static point on the canvas. Properties can be copied and pasted among objects. Pasting can be done either by value or by reference. If pasted by reference, changing one of them will cause the others to change as well. For example, we might design a set of parallel coordinates and a scatterplot, and share the color mapping for the parallel coordinates and the scatterplot. Through copy and paste, one can reuse the properties. However, one drawback is that there is no easy way to indicate what objects share the same mapping in the interface. Currently, the user has to remember that. Another limitation is that our current version does not provide a straightforward way to visualize the scales of the mappings. In future versions, we would like to allow users to drag the mapping properties out onto the canvas and draw them as interactive scales. This would solve both of the above problems.

The style panel works similar to the property panel, showing a list of drawing actions for a graphical object. In the rendering process, each graphical object generates a graphical path, for example, lines, circles, Bezier curves or composites of these, and this path is rendered to the canvas by performing the drawing actions specified here. For example, the "Stroke" action will stroke the path, and it has four properties: width, color, line join and line cap. The "Fill" action will fill the path with a user-defined color. Users can add/remove actions, and reorder them in the style pane. The properties for drawing actions in the style panel can also be mappings, as in the property panel. The user can add or remove these actions in the style panel. Styles can be considered a special set of properties for a graphical object frequently used in visualization designs, and they commonly consists of multiple actions. This is the main reason we separated them from the property panel into an individual panel. Currently we only have stroke and fill actions, in the future, we would like to support more actions, including such that alter the path (e.g., distortion, outline, smoothing), similar to those in Adobe Illustrator.

Basically, we employ a property-editor based approach, which provides a set of uniform editing steps for each type of properties. The editing interfaces are automatically generated according to the properties *declared* in the object types, which is particularly useful for implementing new types of objects. From the users' perspective, a uniform editing experience helps them to learn the system effectively and speedily.

4.2.3 Interacting with Visualizations

In addition to creating and editing visualizations, our system allows a certain set of interactions to be designed. There are two specific tools for interactions on the visualization design.

Moving Elements: Users can move graphical elements in a designed visualization, and as a consequence, some corresponding data attributes might be changed. This should be performed very carefully, as one could easily produce fake findings, mislead others or get confused when changing the original data recklessly. As a default setting, we don't allow the original data attributes to be changed. However, certain properties can be changed without danger, such as the locations produced by a force-directed layout algorithm. Another example consists of users attaching a single attribute to the dataset, and using an axis and a circle to build a "slider" to control it. Such slider-controlled attributes could be used in different situations, such as, e.g., defining the range of a filter property, resulting in an adjustable filter.

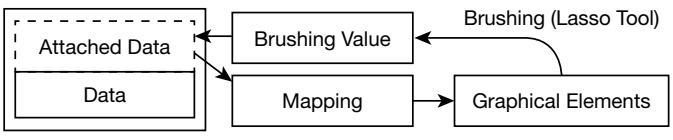

Fig. 6. The internal process of brushing. When the user brushes over a set of graphical elements with the lasso tool, they are collected and passed to the BrushingValue object (which can be created and activated by the user). The BrushingValue object then attaches data back to the data representation, and the attached data can be used by mapping objects, which affect the graphics. Users can define and combine multiple ways of brushing in our system.

Brushing: The lasso brushing tool is paired with the *Brushing-Value* object, which attaches a value to each data item, and once data items are brushed by selecting their graphical elements, their values will change. Users can then use the values to define graphical mappings. This implements brushing and linking functionalities. For example, one could set the fill color of a scatterplot as the brushed value, and use the lasso tool to color the points. Other examples are discussed in section 6.

Our goal is to enable the design of interactions, not just supporting a fixed set of interactions. This is achieved by allowing users to modify attached data attributes in two classical ways (moving and brushing), and we show that dragging, filtering, and brushing and linking can be supported from these atomic actions. There are possibilities to define more complex interactions beyond these, but as the complexity goes up, users have to create several related objects such as axes and expressions, which makes the process a little more complicated.

4.3 Limitations

While the framework of our system is inherently modular and object-oriented, allowing new types of objects to be added easily, it still has several limitations in terms of expressiveness. The framework is based on designing and parameterizing graphical mappings from original, transformed, or user-generated data. This approach has two fundamental limitations. (1) Designing adaptive markers, such as automatically determining the width of bars in a barchart based on the number of bars and the chart width, requires writing specific mathematical expressions, because this involves dividing the chart width by the number of bars, which is not a property of any single data item. In general, our framework does not address the dependencies among graphical objects, but rather performs mappings individually, so in order to accommodate higher-level layout constraints such as overlap avoidance, special objects have to be designed in programming. (2) Our system cannot, without using specific custom layout objects, design recursive drawings such as tree maps, and as aforementioned, the system doesn't support recursively defined data structures directly. One might argue that tabular structure can represent graphs and trees as well, but our data selectors can only enumerate arrays of items in the data hierarchy. It cannot follow references (such as running a graph/tree traversal). This is also true for other declarative approaches (e.g., ProtoVis, D3, Vega and Lyra), which are also resolved to using specific modules for each kind of layout. Abstractions such as [29] might be considered in the future. These limitations currently exclude a range of possible visualization designs.

There are also shortcomings that are more easily solvable within the current framework. (1) Our system currently is constrained in terms of the type of coordinate systems. Positional mappings are done via axes and scatters (the map with Mercator projection is the only exception). It does not currently accommodate circle-based visualizations, nor are polar coordinates currently supported. In the future, we will seek to support different coordinate systems. (2) Axes and their scales are currently intrinsically linked. For simple scatterplots and parallel coordinates this works fine, but it becomes tedious, although not impossible, to share the same scale for different data attributes. There is currently no way to use axes for numerical properties such as widths or radii (and this would be useful for designs such as error bars). This could be solved by additional property-editing interactions and better separation of axes and scales in the future. (3) There is currently no way to specify the order of data item enumeration, which would be useful if we want to stack data items or link through them in different manners. This could be solved by adding a sorting attribute to the data selectors. (4) The system currently lacks a way to specify more general graphical paths. The *LineThrough* object can draw paths through data points, which is somewhat restrictive: if we want to fill the area below a timeline plot, or between two of them, we need a more flexible way to define paths to connect static points and sequences of points together (similar to the "Pen" tool in Adobe Illustrator). The consequence is that our system is less expressive at defining shapes, and more oriented towards line-based visualizations. A general "pen" tool would be desirable.

Despite these limitations, our system can still support an extensive set of visualizations. In section 6, we showcase a variety of examples.

5 Implementation

The system is implemented in HTML5 using jQuery and other open-source libraries. A backend server written in Python Django is used to store the metadata for the datasets and saved visualizations. Here, we discuss some notable aspects of the implementation.

Input Format: The datasets are loaded as JSON objects, where the references are stored as the referenced items' ID (each data item having a unique ID). For our current system, we did not focus on supporting multiple data formats, but conversions from CSV or Excel-like data sources are trivial (without performing join operations). Dataset structure will influence how well certain designs will be supported. For example, if the dataset from subsection 6.2 had been stored as a flat table instead of a hierarchy, we would not be able to create that visualization, unless we added a special "Grouping" generator object, similar to Lyra's approach).

Rendering: We employ multiple layers of HTML5 Canvas. The renderer maintains the status of these canvases, and executes the visualizations on them. There are four layers in our current prototype. The Main layer contains the graphical elements, the Front layer shows selected elements, the Overlay layer shows temporary markers, such as alignment indicators during user interaction, and the Back layer displays the background color and grid. By using layers, we eliminated the need to redraw the entire visualization when the user selects a single element, achieving better responsiveness. In addition, the renderer also manages a viewport, allowing users to move or zoom the visualization, or to export the current view as PNG or SVG files.

Serialization: The visualization is stored as a set of Javascript objects internally; to save a visualization, we need to serialize them to a storable format. We implemented a general Javascript object serializer to support this task, which is capable of maintaining references between objects and retaining type information, which are critical for correctly restoring a visualization. To enable this, we assign a unique identifier (UUID) for each object, and store object references as UUIDs. Type information is preserved by recording an identifier for each registered object type, and restoring the "constructor" attribute for each object. The benefit is that we do not need to write a pair of serializing and deserializing functions for each object class, which reduces programming effort and the possibility of bugs.

Backend Server: The system operates mainly in the browser, but like every web application, it requires a backend server to provide data and store information. The backend server manages user accounts, datasets and visualization designs. It was implemented in Python Django and Twisted. The Django part is responsible for managing user credentials, storing the metadata of all datasets, and saving and loading of visualization designs. The metadata of datasets consists of the data description, data schema, and a URL for the data content. We used the WAMP protocol (based on WebSockets) in a Twisted server, which is connected to a Redis database. It provides real-time updates for changing datasets. Changes in the dataset can be posted to the web-based interface, and the system will update the visualization with the changed data. One can also write scripts that collect data from the web, and send it to iVisDesigner (either replacing original or providing incremental updates). For example, Figure 1 (3) illustrates real-time monitoring of a server's CPU, RAM, and network usage.

Embedding: Users can export their designs and embed them into their own websites or web applications. The datasets and visualization designs could either be retrieved from a server, or embedded statically.

6 Example Applications

In this section, we present a set of visualization design examples on different datasets, with the goal of illustrating the flexibility and expressiveness of our system. Different types of datasets are chosen, including multidimensional data, time series data, and graph data. We also demonstrate a design for Sina Weibo (A Chinese microblog service similar to Twitter) user data, and even some artistic designs without an underlying dataset.

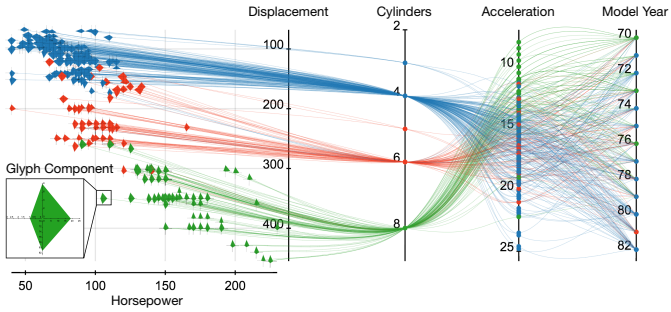

Fig. 7. Glyph-embedded Multidimensional Data Visualization. This design is based on the SPPC design [40], connecting a scatterplot with parallel coordinates. The points in the scatterplot are replaced by a set of glyphs, showing four attributes for each item.

6.1 Multidimensional Data

iVisDesigner can flexibly arrange axes like Flexible Linked Axes [10], whose design space is subsumed by our system. Figure 4 (2) is an example of linked-axes and scatter-plot-based visualization designs. The dataset used there is the 1983 ASA Data Exposition Cars dataset [11]. iVisDesigner emphasizes expressiveness. For example, we can design mini-glyphs for Cars with the *component* object, draw the glyphs on a scatterplot, and link them to a set of parallel coordinates (Figure 7).

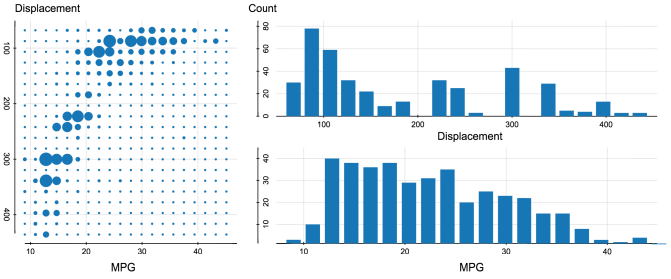

Fig. 8. *Aggregator* objects group values into numerical fields, which can be used to draw a histogram of a particular data attribute. In this example, we plotted the histograms of MPG and Displacement attributes in the Car Data, and also a 2D histogram to show their joint distribution.

Generator objects can be used to compute the statistics of a data attribute, including basic statistics such as min/max, mean, and more involved ones such as histogram. In Figure 8, we show an example using the histogram generator, an *Aggregator* object. This object generates an array of bins to form a histogram of the selected data attribute, which can be displayed in various ways.

6.2 Time Series Data

In Figure 2, we presented a visualization of the Beijing Air Pollution Dataset, crawled from two websites that update hourly. The dataset contains 36 stations, each of which has a name, a geographical location, and a time series of two weeks of measurements. The visualization consists of two different charts. The left chart is a set of timeline plots on a map, each representing a station's measurements. It shows the measurements for each station, allowing comparison between stations at different locations. The right chart is a single timeline plot, which contains the timelines for all of the stations. This chart shows the main trend of all the stations, while revealing some outliers. The left-hand visualization consists of a *Map* object for the geographical coordinates, two *Axis* objects, a *Scatter* object and a *LineThrough* object, connecting all the points in sequence.

6.3 Graph Data

A graph visualization with both, node-link diagram and adjacency matrix representations, is presented in Figure 9. The graph is based on

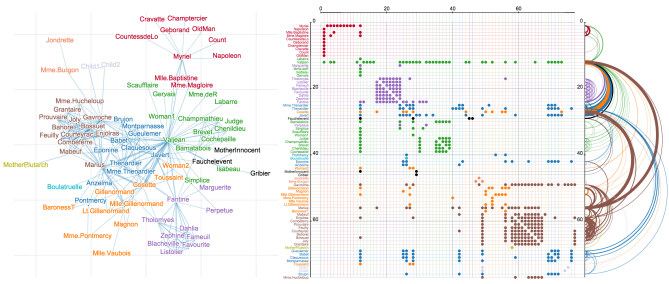

Fig. 9. Visualization design for the co-occurrence network graph from *Les Misérables*. Left: Node-link diagram with force directed layout. Right: adjacency matrix design. A brushing and linking mechanism for the graph edges is designed into this visualization. When the user selects a set of edges from either the left view or the right view, these edges will be highlighted in both views.

character co-occurrence in Hugo's *Les Misérables*[1]. The dataset contains a set of nodes and a set of edges, each edge referencing source and target nodes. The node-link diagram is constructed by first creating a *ForceLayout* object, which runs the Fruchterman-Reingold algorithm [16] to compute the layout, and then attaching the resulting coordinates (x and y values) to the nodes. The nodes are then drawn as a scatterplot of the attached x and y values. The edges between nodes are drawn using references to the node scatterplot.

The matrix representation is created by first assigning an index for each node by the Expression object, then the edges are scattered as *Circles* with the source node's index as the x axis, and the target node's index as the y axis. Since the index attached by the *Expression* object can be changed, users can use the "MoveElement" tool to drag the labels on the left of the matrix to re-order the nodes.

This visualization design also supports brushing and linking. We added a *BrushingValue* object for the edges, so users can select a set of edges in the node-link diagram or in the matrix, and get them highlighted in both views. Since the *BrushingValue* object supports brushing both numbers and colors, we can color the edges or change their widths by brushing. What graphical attribute (color, width) to brush is up to the user. With our system, users are free to design their way of brushing and linking, and have end-users perform it interactively.

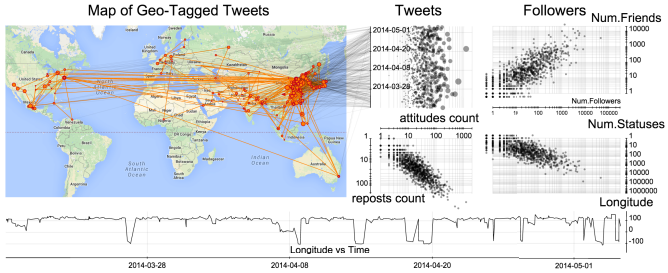

Fig. 10. Weibo User Visualization. The datasets were crawled from Sina Weibo, including the metadata of the recent tweets of a selected user, their followers and friends. This visualization shows the user's trajectory in a map view, and links the map view with the time axis. Scatterplots of four joint distributions are shown. Due to privacy concerns, the data shown here is synthetic, roughly modeled on observed distributions only for illustrating the visualization design. Readers should not draw any conclusion about Weibo users from this visualization.

6.4 Social Network Data

Next, we present an example with flexible linking between different views. We set up a data connection with WeiboEvents [26], which crawls data from Sina Weibo for iVisDesigner. The user can enter

[1]Dataset compiled by Donald Knuth, retrieved from http://bl.ocks.org/mbostock/4062045

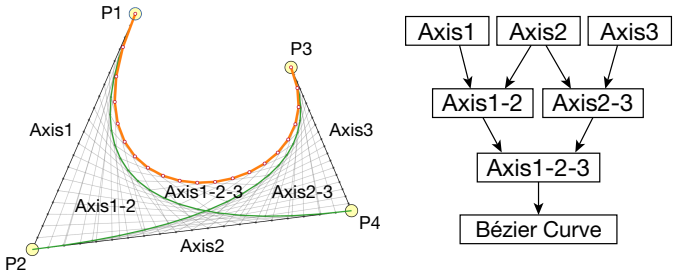

Fig. 11. Interactive Bézier-curve illustration: users can drag the four control points to change the shape of the curve.

an account name in Weibo, then the crawler will crawl the account's tweets, friends and followers, and send the resulting dataset to iVisDesigner, where users can create visualization designs. Figure 10 shows a visualization designed for such a dataset. It consists of a map showing the account's trajectory by connecting all of its geo-tagged tweets on the map. The map is linked with a time axis, revealing where the user was located during each time period. The right side contains several scatterplots, showing the statistics of the account's followers. Users could employ our system to create and connect various components, for example, they could move the time axis around, and see the connections more clearly, or link the map to the bottom timeline.

6.5 Generating Data from Scratch

In iVisDesigner, users can also create graphical designs without using an underlying dataset. This can be useful, e.g., to illustrate some mathematical concepts. In Figure 1 (6), we plotted the Butterfly Curve [12] on the canvas. This is done by first creating a *Range* generator object, which creates a range of numbers from 0 to 24π. Then, we created two *Expression* objects, each of which takes the generated numbers in the range, computes the parametric expressions for the x and y coordinates, and attaches the values to the *Range's* items. Finally, we created two *Axis* objects, a *Scatter* and a *LineThrough* object to visualize the function. The radii of the *Circles* are also bound to the function value in this case. After initial setup, users can interactively change the *Range* and the *Expression* as well, or define other mappings such as the color of the *Circles*.

In Figure 11, we created an interactive illustration of Bézier curves. We first created a *Range* of numbers from 0 to 1 as the t parameter in the curve, then created four *Circles* as the control points. Next we created three *Axis* objects connecting the four control points, then two *Axis* objects between the previous three, one *Axis* between the previous two, and finally the *LineThrough* that connects all the points in the Bézier curve. After the configuration, users can move the control points freely, the curve will change according to the user's interaction.

For these examples, we didn't employ any dataset. In our system, users can not only create data visualizations, but also create mathematical illustrations or even artistic designs, using a set of graphical objects and generator objects.

7 Evaluation

In this section, we first present a performance evaluation of our system and then show the results of an informal user study we conducted.

7.1 Performance Evaluation

We analyzed the overhead added by iVisDesigner's mechanism for rendering visualizations. Our experiments were done on a MacBook Pro with 2.6GHz Intel Core i7 processor, 8GB RAM, running MacOS X 10.9.2. The browser used was Google Chrome 33.0.1750.146. We compared the rendering performance of our system with hard-coded Javascript and D3.js.

We created three visualization designs for the test: (1) Scatterplot with uniform size: a scatterplot for the Cars dataset (406 cars), showing MPG and Displacement, with circle size 5. (2) Scatterplot with mapped size: the same scatterplot, but the circle radius is mapped as the number of cylinders. (3) Timeline: A timeline plot showing minimum temperature in a particular place over 115 days. The times to render these visualizations are 7.3ms, 7.6ms, 0.3ms respectively. The hardcoded version runs 5 to 7 times faster (1.7ms, 1.5ms, 0.04ms) than our system. This is because we have an extra layer of data enumeration and mapping, which involves a lot of function calls in the code. D3.js is around 6 to 30 times slower (61.3ms, 52.2ms, 11.1ms) than our system. Since our system uses HTML5 Canvas as the rendering engine, it is not a fair comparison with D3.js, which renders graphical elements as SVG elements, but since D3.js is a successful programming-based visualization framework that is seen as reasonably efficient, we see this result as encouraging. We chose Canvas as the rendering engine because it is very fast, and we do not rely on the simplified mouse events provided by SVG, since our system itself is responsible for handling mouse events.

We also measured the amount of time to render the visualizations in our design examples. The Graph in Figure 9 with 77 nodes and 254 edges, takes 43.54ms to render, the SPPC in Figure 4 (2) with 406 cars takes 28.61ms, and the Beijing Air Pollution visualization in Figure 2 with 3132 measurements for 36 stations takes 90.49ms to render.

The results of the performance evaluation shows that our prototype system is able to handle visualizations with hundreds or a few thousands of graphical items at real-time or at least interactive frame rates. The performance can be further improved by incorporating optimization techniques, such as reducing the number of function calls and auxiliary objects in the rendering process.

7.2 Informal User Study

We conducted an informal user study for our system. The user study was designed to solicit feedback from real users of the system, after showing and teaching them the basic principles. We recruited 8 users, 4 male and 4 female, ranging in age from 24 to 32 years, with a median of 25, most of whom had Computer Science backgrounds and were familiar with computer-based visualization concepts. We also recruited one of them for a supervised study in the style of a thinking-out-loud cognitive walkthrough.

The informal user study was conducted on an online web interface, with participants performing various tasks, and optionally asking the supervisor questions. After a short introduction of the iVisDesigner tool and the goal of the study, users watched an 8-minute video tutorial (similar to the supplementary video) explaining basic steps in creating a scatterplot for the Cars dataset, a graph visualization of the character co-occurrence data, and some brushing and linking interactions on existing designs. Next, they were asked to try the system by following some steps from the video to get an initial sense of the logic and interactions of the system, and then to try and create their own designs from what they have learned. Finally, the users were asked to complete a survey with Likert-scale (2 = Strongly Agree, 1, 0, -1, -2 = Strongly Disagree) questions. All in all, users spent on the order of an hour on the user study. These are the average results for the questionnaires: "iVisDesigner is {expressive (1.75), easy to use (0.63), easy to understand (1.13), useful (1.88) }", "iVisDesigner is good for { basic visualizations (1.75), novel visualizations (1.25), multidimensional data (1.75), graph data (1.50), time-series data (1.63), visual analytics (1.00), overview (1.75), artistic designs (1.00) }". Most of the ratings are towards the top of the scale (2 or 1), with a few lower scores on "easy to use" and "easy to understand". Participants believed that our system is very expressive and useful, and good for designing visualizations for different types of data. One participant said that the system is very flexible, he could *"make combinatory data/feature selection, for example, linking 2D and 1D elements together to create a polyline"*. From their textual comments, we observed that participants made use of templates very well; basic visualization designs could be created without difficulty. Ease of use and ease of understanding is more of a challenge for our system; as the participants pointed out, they had to carefully watch the video to learn the interactions. They also requested a more comprehensive user guide and tutorial. Usability could be improved by fine-tuning the user interactions. For example, one participant suggested we could enhance the highlighting when users select

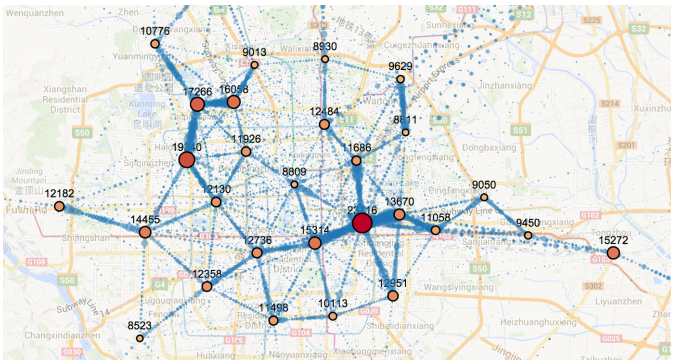

Fig. 12. A visualization design by the participant from the supervised study. Each circle shows a cluster of tweets, dots between circles show the time-dependent movement pattern between clusters.

an element, making it more visible for better guidance. The tool-tip text in the status bar was also recommended to be moved upwards to attract more attention.

In addition, we recruited one participant, a researcher analyzing Twitter feeds for social feature extraction, for a supervised study. We presented three datasets for the participant to explore. The participant was asked to construct visualization designs and try to understand the datasets though these visualizations. During this process, the participant could ask for help on how to use the toolkit and he also commented on his mental processes and considerations.

The supervised study was informative and successful in the sense of user appreciation for the expressiveness, speed, and stability of the system. The participant experimented with the Weibo user dataset (discussed in subsection 6.4). Scatterplots were created to show the correlation of different attributes of a user's friends, and the participant made use of the brushing feature between two plots. Also, he tried to bind the radius of the points to the number of bidirectional followers. *"I could easily discover bot / celebrity clusters. Also, by varying the radius of each circle proportional to the number of bidirectional followers, I was able to locate commercially used / institutional accounts. I was also able to locate individual / organizational accounts by looking at the logarithmic scatterplots by having different types of attributes of users. The labeling is more flexible than other visualization frameworks."*. The participant also experimented with the Weibo geographical dataset, which contains a set of users, each having authored a series of geo-tagged tweets. Before the experiment, the locations in the dataset were clustered by a K-means algorithm, and the trajectories of users were grouped as edges between clusters. The user produced the visualization shown in Figure 12. *"I'm able to identify usage of Weibo on each geographical area in the city. By applying timeline stamps on each edge between adjacent nodes (clusters), we can track the user movements between different locations over time. It is very interesting to see significant amount of communication and movement between adjacent nodes which can perhaps be the reflection of the physical proximity between the users." "I would have spent 1–2 hours to create this visualization by programming, it was done in a few minutes using iVisDesigner."* The user also tried our system on one of his own Twitter datasets for a timeline plot, and identified some previously undiscovered date-time conversion problems in his data preprocessing. In summary, with some supervision, the user gradually understood the general process to create visualization designs in our system, and was able to apply his knowledge to create visualization designs and understand the datasets.

8 Discussion

In this work, we have presented iVisDesigner, a novel expressive interactive information visualization construction toolkit. iVisDesigner is able to cover a wider spectrum of possible visualization designs than previous interactive (non-programming) toolkits. We already discussed its limitations in terms of expressiveness in section 4. Here we discuss usability concerns and possible future improvements.

As we increase expressiveness, the interactions to build a visualization design become more complex than required by more single-purpose toolkits such as [10], because we need to allow specification of extra design parameters, which other toolkits predefine. Compared to, e.g. the Flexible Linked Axes work, we need to specify what data to map from and what types of graphical elements to use, in addition to the axes and scatterplots. There is clearly a tradeoff between expressiveness and complexity. A simple way to improve user accessibility is to add more templates for existing designs. Users could start with a common template, and then modify it to satisfy their own needs. During the design and evaluation of the system, we have observed that designing from scratch is much more involved than modifying an existing design; providing templates certainly helps lower the barrier to entry. Another possible direction is to automate some design decisions by trying to predict what the user may want to show, filling in suggested informed default values for more complex parameters.

The learning curve of our system is not low. One contributing factor is that we haven't yet optimized online help and error reporting, but we are steadily improving on that front. However, when users have to learn a whole set of new concepts, such as axes, references and components, it will inevitably take some time for them to embrace the possibilities and fully utilize their potential for creative designs.

In the framework of iVisDesigner, we did not yet fully consider the interaction among graphical elements. For example, when drawing a graph, there might be multiple edges between two nodes, depending on the dataset. In this case, users might want to define some rules other than just placing two lines in the same place, for example, double the thickness, or use a different color. These types of designs are not feasible in our current framework (also not in D3.js). We could insert a new step in the pipeline of our system, after the mapping stage. Once we have all the graphical elements, we can allow users to define interactions among graphical items before they get rendered.

Dynamic visualization design is another future direction. Up to now, we have dealt predominantly with static visualizations, with the exception that we are able to re-render the visualization when the dataset is changed. However, users cannot define how a graphical element appears or disappears when a corresponding data item is added or removed. This could be achieved by adding a property on graphical elements that would let users specify various types of transitions.

9 Conclusion

We presented iVisDesigner, an expressive interactive web-based information visualization construction toolkit. Our system was designed to be a flexible tool for interactively creating information visualizations, inspired by interactive vector-based drawing tools and established information visualization principles. The system allows users to freely place graphical elements, and links between them, on a large central canvas. We chose a declarative approach to avoid reliance on familiarity with programming and for keeping the usage simple and straightforward. Our unified editing interface allows users to create and edit graphical, guide, and generator objects, enabling the interactive design of complex visualizations. We presented example applications to illustrate the breadth of design possibilities, discussed the limitations and future improvement possibilities of our approach, and reported the results of a performance evaluation and an informal user study. We hope that this work can inspire further contributions in the field of interactive information visualization design.

The source code of iVisDesigner is available on Github: https://github.com/donghaoren/iVisDesigner.

Acknowledgments

The authors wish to thank the anonymous reviewers for their suggestions and criticisms. We acknowledge the participants in our user study for their valuable comments. This work originated as undergraduate thesis work at Peking University and was supported by NSFC No. 61170204. At UCSB, the work was partially supported by the U.S. Army Research Office under MURI grant No. W911NF-09-1-0553 and the Office of Naval Research under contract N00014-14-1-0133.

REFERENCES

[1] Flare. http://flare.prefuse.org/, Jan. 2014.

[2] Processing. http://processing.org/, Jan 2014.

[3] L. Bavoil, S. Callahan, P. Crossno, J. Freire, C. Scheidegger, C. Silva, and H. Vo. Vistrails: enabling interactive multiple-view visualizations. In *Visualization, 2005. VIS 05. IEEE*, pages 135–142, Oct. 2005.

[4] J. Bertin. *Sémiologie Graphique*. Editions Gauthier-Villars, Paris, France, 1967.

[5] J. Bertin. *Semiology of graphics*. W. Berg, transl., University of Wisconsin Press, Madison, Wisconsin, 1983.

[6] M. Bostock and J. Heer. Protovis: A graphical toolkit for visualization. *IEEE Transactions on Visualization and Computer Graphics*, 15(6):1121–1128, Nov. 2009.

[7] M. Bostock, V. Ogievetsky, and J. Heer. D3 data-driven documents. *IEEE Transactions on Visualization and Computer Graphics*, 17(12):2301–2309, Dec. 2011.

[8] S. K. Card and J. Mackinlay. The structure of the information visualization design space. In *Proceedings of IEEE Symposium on Information Visualization*, pages 92–99, 1997.

[9] E. H.-H. Chi, P. Barry, J. Riedl, and J. Konstan. A spreadsheet approach to information visualization. In *Information Visualization, 1997. Proceedings., IEEE Symposium on*, pages 17–24, 1997.

[10] J. H. T. Claessen and J. J. van Wijk. Flexible linked axes for multivariate data visualization. *IEEE Transactions on Visualization and Computer Graphics*, 17(12):2310–2316, Dec. 2011.

[11] CMU StatLib. Car data. http://lib.stat.cmu.edu/datasets/cars.data.

[12] T. H. Fay. The butterfly curve. *The American Mathematical Monthly*, 96(5):pp. 442–443, 1989.

[13] J. Fekete. The infovis toolkit. In *Information Visualization, 2004. INFOVIS 2004. IEEE Symposium on*, pages 167–174, 2004.

[14] A. G. Forbes, T. Hollerer, and G. Legrady. "behaviorism": a framework for dynamic data visualization. *Visualization and Computer Graphics, IEEE Transactions on*, 16(6):1164–1171, 2010.

[15] D. Foulser. Iris explorer: A framework for investigation. *ACM SIGGRAPH Computer Graphics*, 29(2):13–16, 1995.

[16] T. M. J. Fruchterman and E. M. Reingold. Graph drawing by force-directed placement. *Software-Practice and Experience*, 21(11):1129–1164, 1991.

[17] L. Grammel, C. Bennett, M. Tory, and M.-A. Storey. A survey of visualization construction user interfaces. In *EuroVis-Short Papers*, pages 19–23. The Eurographics Association, 2013.

[18] H. Guo, N. Mao, and X. Yuan. Wysiwyg (what you see is what you get) volume visualization. *Visualization and Computer Graphics, IEEE Transactions on*, 17(12):2106–2114, Dec. 2011.

[19] P. E. Haeberli. Conman: a visual programming language for interactive graphics. In *ACM SigGraph Computer Graphics*, volume 22, pages 103–111. ACM, 1988.

[20] J. Heer and M. Bostock. Declarative language design for interactive visualization. *IEEE Transactions on Visualization and Computer Graphics*, 16(6):1149–1156, Nov. 2010.

[21] J. Heer, S. K. Card, and J. A. Landay. Prefuse: a toolkit for interactive information visualization. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 421–430. ACM, 2005.

[22] J. Heer, F. B. Viégas, and M. Wattenberg. Voyagers and voyeurs: supporting asynchronous collaborative information visualization. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, CHI '07, pages 1029–1038, New York, NY, USA, 2007. ACM.

[23] B. Lee, R. Kazi, and G. Smith. Sketchstory: Telling more engaging stories with data through freeform sketching. *Visualization and Computer Graphics, IEEE Transactions on*, 19(12):2416–2425, 2013.

[24] J. Mackinlay. Automating the design of graphical presentations of relational information. *ACM Transactions Graphics*, 5(2):110–141, Apr. 1986.

[25] B. A. Myers, J. Goldstein, and M. A. Goldberg. Creating charts by demonstration. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '94, pages 106–111, New York, NY, USA, 1994. ACM.

[26] D. Ren, X. Zhang, Z. Wang, J. Li, and X. Yuan. Weiboevents: A crowd sourcing weibo visual analytic system. In *Pacific Visualization Symposium (PacificVis), 2014 IEEE*, pages 330–334, March 2014.

[27] S. F. Roth, J. Kolojejchick, J. Mattis, and J. Goldstein. Interactive graphic design using automatic presentation knowledge. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 112–117. ACM, 1994.

[28] A. Satyanarayan and J. Heer. Lyra: An interactive visualization design environment. In *Computer Graphics Forum*, volume 33. Wiley Online Library, 2014.

[29] H.-J. Schulz, Z. Akbar, and F. Maurer. A generative layout approach for rooted tree drawings. In *Visualization Symposium (PacificVis), 2013 IEEE Pacific*, pages 225–232, Feb 2013.

[30] B. Shneiderman. The eyes have it: A task by data type taxonomy for information visualizations. In *In Proceedings of the 1996 IEEE Symposium on Visual Languages*, pages 336–343, 1996.

[31] Trifacta. The vega visualization grammar. http://trifacta.github.io/vega/.

[32] C. Upson, T. A. Faulhaber Jr, D. Kamins, D. Laidlaw, D. Schlegel, J. Vroom, R. Gurwitz, and A. Van Dam. The application visualization system: A computational environment for scientific visualization. *Computer Graphics and Applications, IEEE*, 9(4):30–42, 1989.

[33] B. Victor. Drawing dynamic visualizations. http://vimeo.com/66085662, Feb. 2013.

[34] F. B. Viegas, M. Wattenberg, F. van Ham, J. Kriss, and M. McKeon. Manyeyes: a site for visualization at internet scale. *IEEE Transactions on Visualization and Computer Graphics*, 13(6):1121–1128, Nov. 2007.

[35] J. Walny, B. Lee, P. Johns, N. Riche, and S. Carpendale. Understanding pen and touch interaction for data exploration on interactive whiteboards. *Visualization and Computer Graphics, IEEE Transactions on*, 18(12):2779–2788, 2012.

[36] C. Weaver. Building highly-coordinated visualizations in improvise. In *Information Visualization, 2004. INFOVIS 2004. IEEE Symposium on*, pages 159–166, 2004.

[37] H. Wickham. *ggplot2: Elegant Graphics for Data Analysis*. Springer Publishing Company, Incorporated, 2nd edition, 2009.

[38] L. Wilkinson. *The grammar of graphics*. Springer, 2005.

[39] W. Willett, J. Heer, J. Hellerstein, and M. Agrawala. Commentspace: structured support for collaborative visual analysis. In *Proceedings of the 2011 annual conference on Human factors in computing systems*, CHI '11, pages 3131–3140, New York, NY, USA, 2011. ACM.

[40] X. Yuan, P. Guo, H. Xiao, H. Zhou, and H. Qu. Scattering points in parallel coordinates. *IEEE Transactions on Visualization and Computer Graphics*, 15(6):1001–1008, Nov. 2009.