Visual Informatics 2 (2018) 71–81

Contents lists available at ScienceDirect

Visual Informatics

journal homepage: www.elsevier.com/locate/visinf

VisComposer: A Visual Programmable Composition Environment for Information Visualization

Honghui Mei[a], Wei Chen[a,*], Yuxin Ma[a], Huihua Guan[a], Wanqi Hu[a]

[a] *State Key Lab of CAD&CG, Zhejiang University, Hangzhou, China*

ARTICLE INFO

Article history:
Received 11 December 2017
Received in final form 23 February 2018
Accepted 12 March 2018

Keywords: Information Visualization, Visualization authoring, Interactive development environment

ABSTRACT

As the amount of data being collected has increased, the need for tools that can enable the visual exploration of data has also grown. This has led to the development of a variety of widely used programming frameworks for information visualization. Unfortunately, such frameworks demand comprehensive visualization and coding skills and require users to develop visualization from scratch. An alternative is to create interactive visualization design environments that require little to no programming. However, these tools only supports a small portion of visual forms.

We present a programmable integrated development environment (IDE), VisComposer, that supports the development of expressive visualization using a drag-and-drop visual interface. VisComposer exposes the programmability by customizing desired components within a modularized visualization composition pipeline, effectively balancing the capability gap between expert coders and visualization artists. The implemented system empowers users to compose comprehensive visualizations with real-time preview and optimization features, and supports prototyping, sharing and reuse of the effects by means of an intuitive visual composer. Visual programming and textual programming integrated in our system allow users to compose more complex visual effects while retaining the simplicity of use. We demonstrate the performance of VisComposer with a variety of examples and an informal user evaluation.

1. Introduction

Programming frameworks for information visualization (Bostock and Heer, 2009; Bostock et al., 2011; Heer and Bostock, 2010; Heer et al., 2005) have been created to enable users to construct visual effects for various demands. However, the coding skills required when using such frameworks hinder users from effective construction. As such, a recent trend in information visualization is focused on creating interactive visualization design environments that require little to no programming (Claessen and Van Wijk, 2011; Victor, 2013). Tools such as Lyra (Satyanarayan and Heer, 2014b) and iVisDesigner (Ren et al., 2014) have been created as visualization production tools that allow users to create sophisticated layouts. Unfortunately, these visualization production tools often support for only a small portion of visual forms, thereby greatly limiting the design space. Expressive visualization for multivariate and heterogeneous datasets that can easily be created with visualization languages or toolkits like D3 (Bostock et al., 2011), can be difficult to realize in many current visualization production tools. Furthermore, specifying customized visual designs in these tools is only feasible when the interaction workload is moderate, such as mapping selected data dimensions to appropriate visual channels, or modulating the color scheme for a group of selected data items. The task of managing and customizing detailed designs on a large dataset becomes increasingly complicated or even intractable.

In this paper, we propose a novel programmable integrated development environment (IDE), VisComposer. VisComposer

*Corresponding author: Tel.: +86-571-88206681-529; fax: +86-571-88206680;
e-mail: meihonghui@zju.edu.cn (Honghui Mei), chenwei@cad.zju.edu.cn (Wei Chen), mayuxin@zju.edu.cn (Yuxin Ma), ghh@zju.edu.cn (Huihua Guan), huwanqi@zju.edu.cn (Wanqi Hu)

https://doi.org/10.1016/j.visinf.2018.04.008

has been designed with the goal of making visualization design and optimization easier by providing an intuitive user interface to customize effects with rich authoring controls. VisComposer provides a hybrid solution of constructing visual effects by combining interactive creation of viusal forms with visual and textual programming. It not only provides a drag-and-drop visual editor for rapid prototyping of data-driven visualization but also enables customization of special effects through visual and textual programming. This is enabled through a custom scenegraph in which every component of the visualization process can be configured, edited and shared. It allows for predefined visualization creation while enhancing this through visual programming and manual coding. Our hybrid solution improves on previous work by enlarging the design space of visualization available to the users.

Our contributions include: First, an abstraction over visualization pipeline that modularizes the resultant visualization as a scenegraph, in which nodes and edges are editable and programmable. Second, a visualization composer that provides a set of operations to enable intuitive and programmable visualization design. Its implementation is built upon the JavaScript language and is compatible with JavaScript-based visualization programs, e.g. D3 (Bostock et al., 2011). Third, an integrated visual composition environment that empowers users with the ability to directly program visualization effects and immediately views the results of their programming operations.

2. Related Work

2.1. Visualization Programming Frameworks and Languages

Over the years, a variety of frameworks and languages have been developed with varying degrees of uptake (Haeberli, 1988; Wilkinson, 2005; Mei et al., 2017). Well-known examples include the Infovis Toolkit (Fekete, 2004), Improvise (Weaver, 2004), and Processing[1]. Prefuse (Heer et al., 2005) is a software framework for producing dynamic visualizations with structured or unstructured data. The abstractions provided by Prefuse follow the data visualization pipeline proposed in Card et al. (1999). ProtoVis (Bostock and Heer, 2009; Heer and Bostock, 2010) is a declarative, domain-specific language for constructing interactive visualizations across platforms. A scenegraph is employed to organize the visual design. D3 (Bostock et al., 2011) inherits the basic abstractions from ProtoVis and improves the browser compatibility by directly binding the input data to standard document object model (DOM).

Each of these frameworks and languages has enabled analysts to create and share interactive data visualizations. The power of these languages is in their extendability to create both traditional and novel visualizations. The ubiquity of visualizations created by these libraries shows that there is a need for these visualization frameworks. However, researchers have also recognized that there is a large overhead in using these programming languages to create visualizations from scratch.

Some other textual tools such as Vega (Satyanarayan et al., 2016), Vega-Lite (Satyanarayan et al., 2017) and ECharts[2] abstract over visualization pipeline and specifications and let users author configurations (written in JSON) to reduce the user familiarity of visualization expertise and the programming skills required. However, the learning curve of these tools are still steep and the obscurity of underlying procedure makes users hard to make further customization.

2.2. Integrated Development Environments for Visualization

The need for a faster visualization development time has led to the development of integrated environments for visualization (Bavoil et al., 2005; Lee et al., 2013; Myers et al., 1994). Commercial information visualization software and tools, such as Tableau (Stolte et al., 2002) and Manyeyes (Viegas et al., 2007), provide users with a drag-and-drop interface for creating visualizations from the scratch. However, the resultant visualization designs are typically adopted from canned visual forms, making it difficult (or in some cases impossible) to create novel customized visualization or effects.

Recent work has sought to improve the user experience and allow for a variety of expressive visualization creations. For example, Lyra (Satyanarayan and Heer, 2014b) seeks to improve the expressiveness by mapping the conventional data visualization pipeline into a visual editor. The input data is interactively bound to the properties of graphical marks, and the author can design a new visualization which is represented as a specification in Vega (Trifacta, 2015; Satyanarayan et al., 2016) which then enables the sharing and reuse of the visualization product. Similarly, iVisDesigner (Ren et al., 2014) supports the interactive design of expressive visualization for heterogeneous datasets, covering a broad range of the information visualization design space. Other tools include Ellipsis (Satyanarayan and Heer, 2014a), which implements a model of storytelling abstractions and a domain-specific language (DSL) within a graphical interface for effective story authoring.

While these systems advance previous solutions by enabling visualization customization without writing code, the design space available to the visualization authors is still limited. On the other hand, the language based approach of building each visualization from scratch also has considerable drawbacks. As such, it seems natural to integrate programming capability within an IDE for Visualization. VisComposer presents a hybrid solution combining options for programmability into an IDE for visualization. By enabling users to directly add code segments for customizable and extensible visualizations, our work is able to reduce the programming overhead common among visualization languages while still providing much of their flexibility.

2.3. Visual Programming Tools

Visual programming improves the accessibility for non-programmers by graphical representation and user interactions instead of textual coding. Such idea has already been explored

[1] http://processing.org

[2] http://echarts.baidu.com

for many years and many tools have been developed (Myers, 1990; Boshernitsan and Downes, 2004). Dataflow systems are sorts of visual programming tools where users can specify data processing through dataflow diagrams. Typical dataflow systems (Gurd et al., 1987; Najjar et al., 1999) are designed for general computation. Commercial data analytics platforms such as IBM SPSS Modeler[3] and KNIME[4] enable users to deploy computational nodes for data mining and machine learning.

Dataflow systems can also be used to create visualizations. VisFlow (Yu and Silva, 2017) implements a subset flow model modified from the dataflow model(Roberts, 1998) and allows users to create custom visualization by manipulating a flow diagram. iVoLVER (Méndez et al., 2016) provides an infinite canvas to perform data extraction from images, specify the dataflow, and present visualization results. Similar applications can be found widely used in the field of computer graphics. In order to improve the efficiency of creating visual effects in computer graphics, textual programming editors (or IDEs) have been widely adopted in the programmable graphics pipelines (Castno and Horowitz, 2006; Tatarchuk, 2004). These editors provide both a visual composition component in which authors can create graphics effects from canned modules and add novel effects through programmable shaders.

3. Abstraction Over Visualization Pipeline

In VisComposer, a scenegraph is constructed. Resources including data and visual primitive are added into the scenegraph and then their transformations and visual mappings are specified before the final visualization result is generated.

In this section, we propose our scenegraph abstraction over visualization pipeline for creating information visualization in a programmable way. First, we introduce the input data to process and then give the definition of our scenegraph abstraction over information visualization pipeline. At last, the idea of programmable weaving is proposed.

3.1. Data

Tabular data is the most widely used data in creating information visualizations. We define a set of tabular data as the cartesian product of attributes and records. Each column of the table representing an attribute of the data and each row of the table is corresponding to a record. The value in a cell shows a specific attribute of one record, according to the column and row it is located. For other structured data, we treat them as tabular data with special types of items in each cell. For instance, a set of data which is used in drawing a node-link graph typically consist of nodes and links, each of which is a tabular data. Such a dataset can be represented as a table with only one row and two columns, which the two columns are nodes and links. Both values in the two cells are tables instead of simple numbers or strings. Through such kind of representation, we can construct hierarchical dataset as tables located in table cells.

[3] https://www.ibm.com/us-en/marketplace/spss-modeler
[4] https://www.knime.com/knime-analytics-platform

3.2. Pipeline Abstraction

Roughly speaking, the conventional visualization pipeline (Card et al., 1999) consists of two stages: the data transformation stage where data is cleaned and filtered, and the visual mapping stage where data is mapped to an appropriate visual structure (e.g., size, color). Throughout the pipeline, users can view, navigate, and control the data states and operations, making a closed feedback loop in the interactive design.

For an in-practice visualization, the actual pipeline is more complicated. In our work, we modified the data-hierarchy of sub datasets introduced by Roberts (1998). As a result, we combine the **dataflow** representation of data transformation and visual mapping in visualization pipeline and the **scenegraph** abstraction of visualizations.

3.2.1. Attributes transformation and mapping

We treat the process of data transformation and visual mapping as a dataflow model. Conventionally, the dataflow is represented by a node-link diagram. A node is a module that deals with data transformations and generations of visual structures. The input and output ports of a module represent the required data accepted by the module and outgoing data generated in the module. Each link connecting two ports specifies the data transfer during data transformation stage or represents a visual mapping linking the data to the visual channel.

Although we put all data transformation and visual mapping in a uni.ed dataflow model, there are several di.erent types of modules performs the different functionalities:

Data modules (Figure 1 (a)) hold data for other modules to gain as input. The data they hold may come from the original dataset or output data of other modules. The decomposition of columns may occur at this step.

Calculation modules (Figure 1 (b)(c)(d)) process and filter data before they are mapped to visual structures.

Primitive modules (Figure 1 (e)) are the abstractions of visual structures. The input ports of primitive modules represent visual channels to specify.

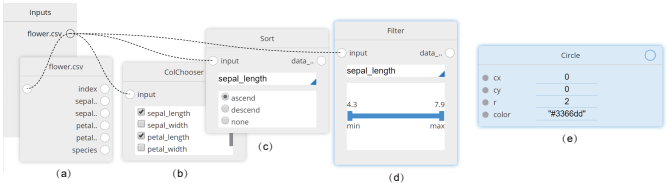

Fig. 1. Some different kinds of modules: (a) data module, (b) module for attributes filtering, (c) module for sorting, and (d) module for filtering.

A typical dataflow of a visualization may consist of all three kinds of modules and links among them, sequentially. Data modules specify the data to visualize. The data are then be filtered and processed by calculation modules, before finally mapping to visual channels provided by input ports of primitive modules (Figure 3 (b)).

3.2.2. Graph construction

In order to improve the customization degree, we decompose the resulting visualization into a more detailed structure for specifying. We decompose the visualization according to a *scenegraph* abstraction (Figure 2).

A scenegraph represents the structural abstraction of a created visualization. It employs a tree structure to encode how the primitives are organized in the drawn view. In a scenegraph, the root node corresponds to the entire graph and the child nodes of a node corresponding to a decomposition of the parent node, each of which holds and draws a subset of the dataset held by the parent node. A node of a scenegraph contains three parts: the bound data, dataflow, and primitives. Bound data is filtered, processed and mapped to visual channels in terms of the dataflow specification. After all, resulting primitives are generated. The edges under a node specify the organization form between the node and its child nodes. We call such an organization form a composition.

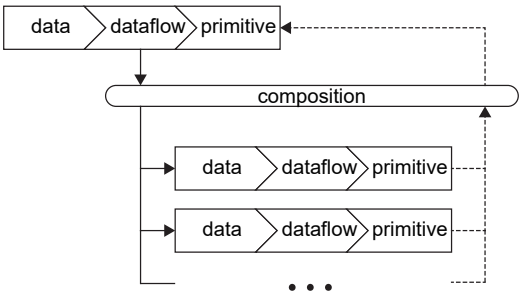

Fig. 2. A scenegraph consist of hierarchical sub datasets, dataflows and primitives organized by the composition.

Possible composition modes include juxtaposition, superimposition, overloading, and nesting (Javed and Elmqvist, 2012). For instance, the dimensions of the Iris dataset[5] can be used to compute a two-dimensional (e.g., 4×4) spatial subdivision of the hierarchy of the node, yielding a two-dimensional juxtaposition layout (e.g., a scatterplot matrix). Besides the allocation of data, the compositions also have e.ect on the way geometries are generated and drawn. At the very beginning, a global coordinate system is dispatched to the root node. During the specifying of compositions, a local coordinate system may be built for each child node. For instance, each scatterplot of the scatterplot matrix mentioned above has its own two-dimensional space weaved by the two axes. The origin point of each scatterplot is determined by the overall juxtaposition layout.

In such a scenegraph and composition model, a complex visualization graph can be decomposed into several simple visualization pipelines, each of which contains only a uniform dataset, together with its corresponding transformations and visual mappings. The composition and decomposition of complex data structures make the specification of dataflow as simple as possible.

[5] `http://archive.ics.uci.edu/ml/datasets/Iris`

3.3. Programmable Weaving

Based on the abstraction over visualization pipeline introduced above, we can easily construct a flexible way to specify visualization graphs, which consist of two main parts: dataflow specification and hierarchical composition.

3.3.1. Dataflow specification

The specification of dataflow involves visual programming methods. Users may drag and drop to create modules and bind ports to perform data transformations and visual mappings. Moreover, manual programming is also a solution to deal with complex computations and specific layout algorithms.

3.3.2. Hierarchical composition

During the composition process, users may specify the way data is decomposed for computation and rendering at a more detailed level. Such compositions can be automatically generated from typical modes or specified by users.

4. The Visualization Composer

In this section, we briefly describe how the programmable model works. For the simplicity and flexibility, in our implementation, we choose JavaScript as the programming language.

4.1. Programmable Visualization Shaders

We enable programmability by allowing users to manipulate the scenegraph nodes, compositions, and dataflows. Moreover, the users can also craft a piece of JavaScript code for manual coding of scenegraph compositions and dataflow modules. Specifically, the following schemes are employed to enable a visual programmable composition environment.

4.1.1. The visualization shader

We denote a complete piece of custom built code as a *visualization shader*. Such a shader is used to create visual effects for specialized visualization elements. The effects achieved by visualization shaders are comprehensive, such as specifying visual properties, organizing graphical primitives, and designing a special visual form. A visualization shader can be directly used to customize a visualization element and can be seamlessly integrated into the intermediate program that is generated by the system during the composition process. When the composition process is completed, a set of javascript code associated with the Scenegraph and Transformation modules is synthesized and is run to generate a visualization. We introduce three kinds of visualization shaders: expression, statement, and block.

- An *expression* is a mathematical or evaluation expression that is used to set data binding operations (e.g., "`parent().width / 4`" set a primitive's width to be a quarter of its parent). It refers to a control widget in the interface (see Figure 3 (e)).
- A *statement* is a set of algebraic functional equations to specify data transformation or visual mapping operations (e.g., "`normalized_x = (x - min(x)) / (max(x) - min(x))`"). It refers to a control widget in the interface (see Figure 3 (f)).

- A *block* denotes an editable function with an input state and an output state. A block is shown and edited in a visual text editor (see Figure 3 (g)). A bock-based shader offers the most flexible programmability, and can fulfill many kinds of special effects (e.g., a treemap layout algorithm).

4.1.2. Cross-module state transfer

To guarantee the user-generated program matches the associated elements and operations in composing a visualization, the states transferred from other module or inherited from other elements can be exposed, used or modulated in a block-type shader. We classify the states into three categories:

The **input state** denotes the custom-built designs of elements in previous composition steps. It is passed through scenegraph edges or dataflow links. The user can freely modulate an input state in writing a visualization shader. The input state is implicitly accessible if users compose the visualization by direct manipulation in the Transformation, Scenegraph, and the visualization modules.

The **output state** is modulated in a visualization shader and can be transferred to other visualization elements. The transfer can be specified in the shader, or specified by user interactions. The transferred state can be used for further transformation or bound to other visualization elements.

A **gobal state** is instantiated when constructing the root node of the Scenegraph and is used to specify functions (e.g., `getCol`: get a specific column of a table data) and global variables (e.g., the size of the visualization canvas). The global state can be accessed by all elements during the customization of a visualization shader.

4.2. Code Assembly and Graph Generation

In order to enable the running of user-defined dataflows and code pieces, the process of generating a visualization result is divided into three steps.

First, after the scenegraph, dataflows and modules are specified by users, they are collected and assembled. The visualization composer assembles all elements of the Scenegraph and the Transformation and generates a topological organization that describes the directed input-output links between elements.

Then, the elements of each module are translated into JavaScript code, including both native ones and custom ones. By solving conflicts between variable names and incorporating exception handling and other environment settings into the program, the production code is produced. The code can then be used for visualization and interactive user modulation.

After all, data are distributed and all codes are executed according to a breadth-first order of the scenegraph nodes to generate the resulting graph. The produced codes are compiled within the web browser using the evaluation function (eval) of JavaScript and display the visualization. During the execution, the environment of variables and the structure of geometries are dynamically created and updated.

5. Interactive Visualization Design Environment

In this section, we will describe our web-based visualization design environment, VisComposer.

5.1. The Interface

The interface (Figure 3) contains four main views corresponding to four modules of the visualization composition model. All widgets and views are designed to be collapsible so that more screen space can be preserved for the main view.

The **resource** view (Figure 3 (a)) shows the resources being used in designing visualizations. The view contains four main components that list the names, structures and properties of data and the data primitives, composition operators and visual forms. Detailed information of selected items (e.g., the input dataset) can be further explored using additional popup windows. Our system supports the loading and composition of multiple datasets and visual forms.

The **transformation** view (Figure 3 (b)) is the workbench for performing the bind, map and abstract operations. A suite of textual and configuration panels (Figure 1) are provided for user modulations. The user can craft transformations by editing ones provided by the system or write a new visualization shader.

The **scenegraph** view (Figure 3 (c)) shows the scenegraph structure of the visualization and enables many user interactions that represent the abstract and interact operations defined in the next section. The user can create nodes, links and compose the visualization within this view. The tree shows the basic information of each node and link, such as the type of primitives, the underlying data selectors and the composition operators. The user can edit nodes and links interactively or load from stored forms as a prototype template.

The **visualization** view (Figure 3 (b)) is the container of the visualization being designed. It is updated whenever a modification is made during the design process. To enable quick design iterations, the view is synchronized with the creation of the scenegraph. A proxy geometry or graphical glyph is shown if a node or link of the underlying scenegraph has not been bound, or if a primitive has not been mapped to a visual propertiy. Interactive specification and manipulation of the graphical primitives or forms in the view is allowed.

5.2. User Interactions

To compose a visualization, users can perform operations of the visualization composition model by interactively manipulating data, primitives, composition operators, visual forms, and intermediate structures, such as the scenegraph. The composition process begins only after a dataset is loaded. User interactions with the VisComposer interface can be defined as follows.

5.2.1. Selection

Two types of selection tools are provided to facilitate the specification of points of interest. The lasso selection tool supports brushing a rectangular or an arbitrary shaped region within the visualization or transformation view to select data items or primitives directly. Alternatively, users can select the data items or primitives by double clicking on the nodes or links in the scenegraph view.

5.2.2. Drag-and-drop

To transfer selected items in or across different views, users can drag the object from the original view to the destination.

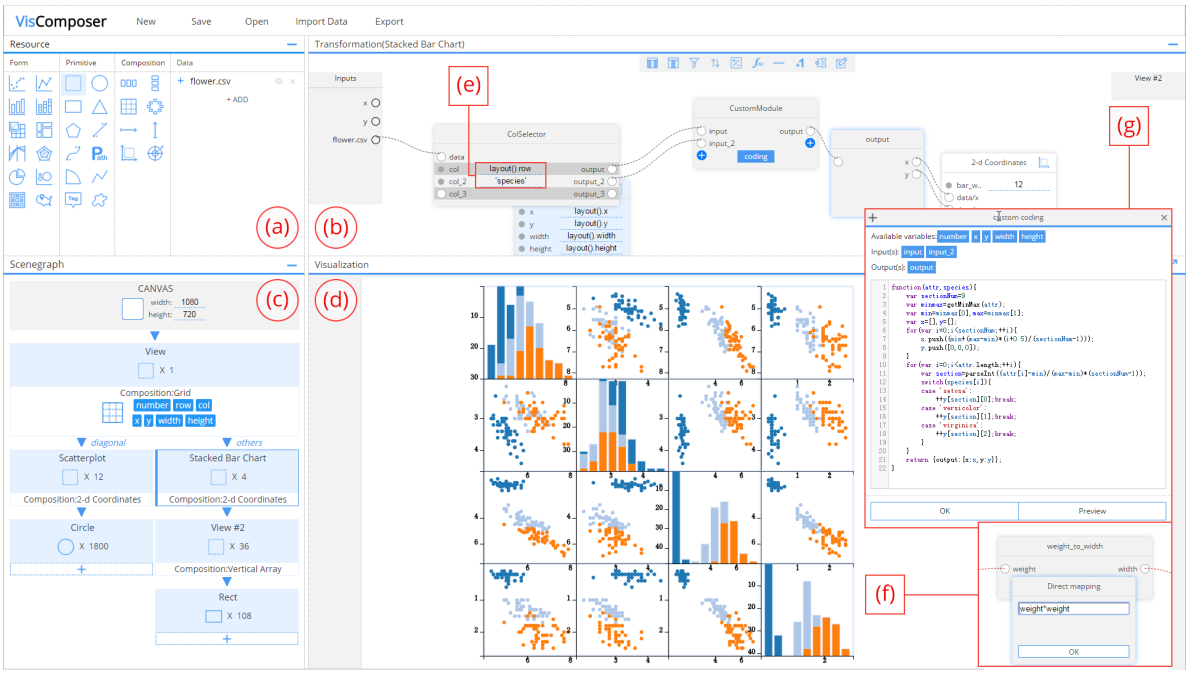

Fig. 3. The interface of VisComposer: (a) the resource view, (b) the transformation view, (c) the scenegraph view, and (d) the visualization view. (e) ~ (h) Editors for the three types of programmable shaders. The scatterplot matrix representation of Iris Flower Dataset is displayed.

5.2.3. *Painting*

The user can freely draw an axis or other geometry in the visualization canvas in order to directly specify the visual design. The paint tools can help users to quickly create and refine new visual structure. A set of docking points are utilized for the purposes of positioning and the primitive module with visual channels bound is generated for further programming (Figure 4).

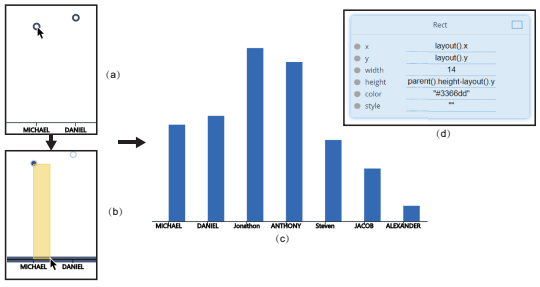

Fig. 4. Painting with docking points on (a) data points and (b) axes to create a (c) bar chart. (d) shows the generated module and bound visual channels.

5.2.4. *Programming*

In the transformation view and the scenegraph view most of the elements are programmable. The user can bind data by using a short expression in an input box, or create a functional modifier to assign a custom visual mapping. The user can also trigger a code editor and write scripts in JavaScript. Quick debugging of code is enabled by the interactive output of intermediate results in the visualization view.

5.3. *Design Modes*

VisComposer enables both high-level and low-level control features for use by technical developers as well as purely visual controls to enable visualization design by non-technical artists. Two design modes are provided: design from scratch or template-based design.

Design from scratch: This mode requires users to build elements from scratch. For modification, users can flexibly erase, restore, and restart an ongoing composition by means of the interface. Any element can be edited by either programming or pre-defined interactions if necessary.

Template-based design To simplify the customization of effect and share visualization, a high-level manipulation mode is supported by leveraging the visual forms. By default, a visual form can be formulated based on a template representation. When a template is selected or loaded, the environment automatically instantiates the scenegraph structure. All the components in a template are editable and programmable. In this way, users can bypass the tedious setup steps and dive straightly into the visualization design process.

Additionally, as the system provides real-time preview of the resultant visualization, it can be used to perform some simple visual analytics tasks. For instance, after creating a scatterplot with modules which filter data by their attributes' values, users can drag the slider (Figure 1 (d)) to watch the distribution of the dataset within different attributes' range.

5.4. *The Implementation Details*

5.4.1. *Data Format*

VisComposer stores and manipulates data as JavaScript Objects. Data sources, such as CSVs, JSONs, can be loaded and converted to objects. VisComposer provides a flexible data type control to improve data import. Profiles of the input data, such as the data types and dimensions, are kept and can be used in the composition process.

5.4.2. Preparing

Before actually drawing the graph, a test evaluation with no rendering of geometries may be performed. Such a test evaluation is used to check the states of running data and to prepare interfaces for interactive specification. For instance, getting the min and max values of an attribute can help setting the range of the slider in a filter module (Figure 1 (d)).

5.4.3. Drawing

VisComposer outputs a structured representation of primitives. Then these primitives are sent to the renderer in a batch mode. In general, there are two modes for rendering the primitives: creating and displaying SVG nodes which support mouse and keyboard interaction from the browser; drawing in the canvas to generate a static result. To support interactive development, VisComposer utilizes the SVG mode during the visualization composition process. Exporting the result in to HTML5 canvas is also supported when publishing the design.

5.4.4. Interaction

VisComposer provides rich user controls which are synchronized among different views. For example, moving an element in the visualization view leads to the changes of its x and y coordinates in the Transformation view; creating a visual mapping in the transformation view modifies the scenegraph and the visualization. In this way, users can quickly compose a visualization with user-friendly interactions and obtain real-time feedback in the visualization module.

5.4.5. Serialization

A visualization is composed of a list of JavaScript objects that are referenced to each other. Similar to iVisdesigner (Ren et al., 2014), VisComposer serializes all objects and maintains references to enable external storage, loading and reuse of components. VisComposer assigns a universally unique identifier (UUID) to each object that represents references out of memory. The object collection of a visualization is processed in depth-first order and objects are stored when they first occur. Specifically, we employ an enhanced serialization scheme that users can save part of the pipelines or a sub-tree of the scenegraph into an offline file.

5.4.6. Reuse and Export

VisComposer supports the reuse and exportation of visualizations in three ways: 1) The designed visualization can be exported as a bitmap image or vector graph; 2) The workspace can be exported and saved as a template file for future reuse; 3) The design can be packed with a runtime library, which can be embedded into an HTML page.

6. Examples

In order to demonstrate our work, we show the creation of a variety of visualization using VisComposer. Examples range from network and hierarchical datasets to structured and unstructured data. In each example explained below, the flexibility of VisComposer is highlighted. Please view the supplementary video for more details.

6.1. Scatterplot Matrix

Typically, a scatterplot matrix consists of neatly arranged scatterplots, each of which represents two of the dimensions. Because each of the diagonal cells has the same dimension bound to the x and y coordinates, the resultant visualization of those individual scatterplots will always be a straight line. As such, we can use visual forms other than scatterplot in the diagonal cells. Unfortunately, creating this type of customized scatterplot is non-trivial in many current tools. VisComposer allows users to easily specify a grid array in the second level of the scenegraph, and then specify the visualization of each cell individually. Figure 3 shows a scatterplot matrix representation of the Iris Flower Dataset[6]. The row and column number of the grid is set to be the dimension of the input dataset. The categorical attribute is bound to the point color. A data selector is used to select the diagonal cells and replace them with a bar chart visualization. Each bar chart shows the distribution of the corresponding dimension over its domain, which is accomplished by an 1D coordinate array of the sized bar with values segmented and counted in the transformation module.

6.2. Bubble Plot

The bubble plot representing the *Flare class hierarchy*[7] is shown in Figure 5 (a). Each bubble represents an entity in the Flare library. Because the bubble layout is complex and can be accomplished by interactively specifying mappings, a visualization shader is employed in the transformation module to compute the coordinates of bubbles. In this example, the positioning algorithm is adopted from `d3.layout.pack` in D3. Additional visual properties are added such as colors and text labels.

6.3. Stacked Bar Chart

A stacked bar chart is a bar chart that encodes an additional dimension by stacking it along a certain axis (Figure 5 (b)). The hierarchical structure of the scenegraph favors the construction of this kind of nested coordinate systems. First, the height of the bar is computed by aggregating the values in each category. Then a scenegraph node is created to represent a basic bar chart. In each bar, a local 1D coordinate system is employed to stack the additional dimension. Note that the scale of the local coordinate system equals to the scale of the y-axis in the global system.

6.4. Parallel Coordinate Plot

VisComposer supports the construction of multiple coordinate systems, like the parallel coordinate plot and the radar chart. Figure 5 (c) presents an example of an advanced parallel coordinate visualization design (Yuan et al., 2009). The *Auto MPG Dataset*[8] is used. In designing this example, the axes and the links between axis pairs refer to two separated nodes in the scenegraph. To enable the scatteplot visualization between a

[6] `http://archive.ics.uci.edu/ml/datasets/Iris`
[7] `http://bl.ocks.org/mbostock/4063269`
[8] `https://archive.ics.uci.edu/ml/datasets/Auto+MPG`

Fig. 5. Examples of visualization designs.

pair of adjacent axes, a data selector is used to specify two data dimensions and create an individual node of the scatter plot in the scenegraph.

6.5. Force-directed Graph

The force-directed layout is one of the most commonly-used layout algorithm for graph drawing. VisComposer makes it easy to create a force-directed graph visualization. Figure 5 (d) shows the character co-occurrence in the *Les Misérables Dataset*[9]. For the node layout, we construct a programmable transformation to generate the coordinates of nodes. The user can write a shader with JavaScript language and instantly view the effect. In this example, the link width is intended to encode a dimension of the input data. With VisComposer, users can simply specify a link in the Visualization module and bind the width with the data dimension in the Transformation module with a non-linear transformation.

6.6. Tag Cloud

VisComposer supports the visualization of a collection of words with the tag cloud techniques. Figure 5 (e) is an example of the tag cloud based on the *Countries and Dependencies by Population Dataset*[10]. Here, the font size of the words is proportional to the word frequency, which can be achieved by binding the word frequency with the color with a linear mapping in the Transformation module. Additionally, a visualization shader is created based on a randomized greedy strategy to layout the words. The font color can be interactively bound to a categorical dimension of the dataset.

6.7. Squarified Treemaps

VisComposer is capable of crafting recursive layouts, like the treemap. A treemap consist of a recursive drawing layout which is difficult to design with a static scenegraph structure. To handle the recursion, a custom composition can be created by using a visualization shader in the transformation module. Figure 5 (f) shows the visualization of the *Public Company Bankruptcy Cases Dataset* [11], and the associated scenegraph and workflow. It should be noted that the squarified layout is implemented without using any existing JavaScript visualization libraries. To support the color assignment on the rectangles, a color mapping transformation is created afterwards.

7. Evaluations

All user studies were perfomred on a PC equipped with a Dell display (24-inch LCD with resolution of 1920× 1080 pixels) and Google Chrome 64.

7.1. Evaluation Setup and Procedure

We conducted an informal user evaluation to collect feedback on VisComposer. We recruited 15 participants (10 were male) aged from 21 to 28 years old, in which 7 participants were graduate students and 8 participants were undergraduate students. All the participants are computer science college students. 6 participants are very skilled at visualization and have been trained on visualization design and coding. 9 participants have only passing knowledge on visualization.

All subjects used the Google Chrome browser. Feedback is collected with the online survey platform Kuiksurveys

[9] http://bl.ocks.org/mbostock/4062045

[10] http://en.wikipedia.org/wiki/List_of_countries_and_dependencies_by_population

[11] http://catalog.data.gov/dataset

(http://kwiksurveys.com). In general, five tasks were assigned to each participant. The tasks were designed to test the capability of VisComposer in different aspects Tasks were carried out in ascending order in terms of the degree of difficulty. Table 1 lists the details of the tasks.

Table 1. Tasks for user evaluation.

Visual forms	Datasets
Stacked Bar Chart	Iris Dataset[12]
Parallel Coordinate Plot	Auto-MPG Data[13]
Force-directed Graph	Victor Hugo's Les Misérables[14]
Squarified Treemap	Company Bankruptcy[15]
Tag Cloud	Countries' population[16]

The user study was performed for each participant separately. The entire process consists of five steps and costs 60 to 90 minutes. The evaluation procedure for participants consisted of four steps: **1) Instruction:** The participant watched a video demonstration that teaches them how to use the system and how to create a scatterplot matrix. The participant was then given hands on training by an experienced tutor of VisComposer. **2) Training:** The participants spent 30 minutes exploring the system and were allowed to ask the tutor questions. **3) Evaluation:** The participant was asked to perform the tasks in Table 1 sequentially. The result of each task was output to a standard visualization effect file. **4) Questionnaire:** After the tasks were finished, the participant was asked several short questions, and the answers and task results were evaluated and counted.

7.2. Qualitative Evaluation

We designed a questionnaire that contains 9 questions in two categories:

- Subjective feelings: VisComposer is 1) expressive; 2) easy to use; 3) easy to understand; 4) useful.
- Feasibility: VisComposer is feasible for 5) basic visualization; 6) programmable visualizations; 7) tree and graph visualization; 8) text visualization.

All questions were encoded with a 5-degree Likert Scaling (from -2 = lowest to 2 = highest). Table 2 presents the average score of each question. On average the feedback is positive: most scores are in the range of 1 ∼2. However, the scores on "easy to use" and "easy to understand" are relatively low. The question "feasibility for programmable visualizations" receives the highest average score when compared to other questions, which verifies that the programmability of VisComposer is effective and highly appreciated.

[12] http://archive.ics.uci.edu/ml/datasets/Iris

[13] https://archive.ics.uci.edu/ml/datasets/Auto+MPG

[14] http://bl.ocks.org/mbostock/4062045

[15] http://catalog.data.gov/dataset/public-company-bankruptcy-cases-opened-and-monitored-for-fiscal-year-2009

[16] http://en.wikipedia.org/wiki/List_of_countries_and_dependencies_by_population

Table 2. Average scores of 8 questions in a qualitative evaluation.

Question	Average Score
1) expressive	1.53
2) easy to use	0.73
3) easy to understand	1.07
4) useful	1.80
5) basic visualization	1.73
6) programmable visualizations	1.93
7) tree and graph visualization	1.47
8) text visualization	1.73

7.3. Quantitative Evaluation

We performed an additional quantitative evaluation to compare the performance and usability of VisComposer with other similar tools, including Tableau, Lyra, and iVisDesigner. After the qualitative evaluation, we chose 4 of the 6 skilled participants and asked them to create a scatterplot matrix based on the Iris dataset, which is shown in Figure 3. They were asked to create the visualization in 3 steps. First, they are asked to create a simple scatterplot with two of the dimensions with color encoded. Then, they need to construct a scatterplot matrix with 4×4 scatterplots. The last step is to replace the scatterplot in diagonal cells with a histogram. The participants are also asked to use the tools to get familiar with them for one hour before starting the evaluation.

Table 3. Performance comparison for different tools. Time spent on each steps are recorded.

Tool	Scatterplot	Scatterplot matrix	Replace diagnal cells
VisComposer	1 minute	8 minutes	21 minutes
Tableau	<1 minute	<1 minute	unable
Lyra	1 minute	(13 minutes)	unable
iVisDesigner	<1 minute	(5 minutes)	unable

We compared the time spent with four tools in Table 3 and time spent on each step is recorded. The third step is unable for some tools to finish. To be mentioned, Lyra and iVisDesigner do not support scatterplot matrix but it is possible to draw it by place several scatterplots side by side. Tableau is the best when creating the scatterplot and scatterplot matrix, but cannot deal with the replacement of diagonal cells. The participant using VisComposer, Lyra, and iVisDesigner spent similar time when creating the scatterplot and the scatterplot matrix. VisComposer is the only tool that can finish all three steps.

7.4. User interview

After the evaluation, we did an interview with the participants. Some participants noticed the separation of the visual design structure and the visualization and thought it is advantageous because it allows users to focus on the design of the visual transformation and visual mapping. One user stated: "the workflow and the scenegraph present the entire structure of my visual design, which is quite necessary for compound and complex visual design." Two participants extremely appreciated the programmability as they have experiences on both

graphics shading programming and web visualization development. They commented that "it is very useful to apply the visual mapping and layout codes and craft a visualization without following a fixed template. Also plenty of redundant work can be avoided because VisComposer incorporates rich controls. All I need is to focus on the core design.". However, half of the participants claimed that the user interface is complicated because it requires many user interactions for specifying data transformations and visual mappings.

8. Discussions

8.1. Expressiveness

We aim to fill the gap between textual coding and interactive specification of visualization design. Existing interactive design environments (Ren et al., 2014; Satyanarayan and Heer, 2014b) have demonstrated the feasibility of interactive visualization design. With VisComposer, comprehensive visualization effects can be made easy by exposing the programmability in each component. The interactivity of the program enables faster development and quick design iterations, thus improving the final effect. This hybrid development scheme facilitates both artists and programmers in visually authoring special effects. We believe that our hybrid approach can improve the expressiveness of visual effects controlled by users, while remaining intuitive user interface and high usability.

However, the balance between high flexibility and easy-to-use interface is not completely satisfied. For instance, when designing a complex visualization, the layout may totally depends on custom programming blocks and JavaScript codes. In such a case (e.g. the bubble plot example), our VisComposer performs no more than a textual editor.

8.2. Accessibility

VisComposer provides different design modes, including top-down and bottom-up processes, for users to choose. Users can paint on visualization, drag-n-drop, or manually code to specify data transformations and visual mappings. However, users may be accustomed to traditional control panels and want to use dropdown menu to make specifications. One possible solution is that we may collect the options on all nodes in a dataflow and show them at a same panel.

Another issue is the abstraction of geometries rendering. Our primitives come from the definition of SVG elements. However, simple ones such as circle and rect are easy to use, while many complex shapes are all rely on SVG paths. We may need an encapsulation of special shapes, such as arcs, pies and parametric curves, like what D3 do (e.g. `d3.pie`).

8.3. Comparison to Other Similar Tools

There are several ways in which tools allows users to specify data transformations and visual mappings, including textual programming, direct manipulation through control panels, visual programming, and surrogate objects (e.g. docking points in VisComposer) (Mei et al., 2017). Generally speaking, textual programming is the most flexible way but requires programming and visualization expertise, while direct manipulation is easy-to-use but lack of flexibility. Visual programming falls in between. Surrogate objects allows users to intuitively see what they are manipulating but is sometimes arbitrary.

Table 4. Comparison of VisComposer, Tableau, Lyra, and iVisDesigner. The ways in which the tool allows users to specify customizations include (T) textual programming, (D) direct manipulation through control panels, (V) visual programming, (S) surrogate objects, and (✗) not supported.

Tool	Data transformation	Visual mapping	Interaction &animation
VisComposer	**T,V,D**	**T,V,D,S**	✗
Tableau	**D**	**D**	✗
Lyra	**T,D**	**D,S**	✗
iVisDesigner	✗	**D**	✗

Table 4 summarizes the way to specify customizations using VisComposer and other similar tools, including Tableau, Lyra, and iVisDesigner. It can be seen that VisComposer supports diverse ways for customizations, as we focus on balancing the capability gap between expert coders and visualization artists. However, all these tools do not support customizations on interactions and animations.

8.4. Limitations

A typical scenegraph is a graph rather than a tree as data and some properties can be shared among nodes. But VisComposer uses a tree-based scenegraph to structure the visualization, as we bind the decomposition of data with the tree-based visual structures (e.g. DOM tree of SVG elements). It is similar to what D3 does. However, in D3, users can simply share data through variables defined but it not works for VisComposer as it is hard to represent in the scenegraph view and dataflow diagram. To reduce users' confusion, we limit the scenegraph to a tree structure. This limits the flexibility of the creation process. On one hand, the tree structure requires a directional transfer of the visualization design so that the resources (e.g., data and primitives) can only be transferred from top to bottom. The information transfer in sibling nodes or upwards is infeasible. We plan to address this problem by implicitly passing values using the global state, which is hard to maintain in terms of the system architecture. On the other hand, the scenegraph is tightly coordinated with the data transformation, each of which influences the other subject on a modification. Once the scenegraph is constructed, it is hard to modify the visual design, which restricts the flexibility of the user control.

Moreover, some interactions for visual programming is arbitrary. One possible solution is to create more templates for users to load and reuse. Additionally, although interactive design and code-based design can be synchronized, they are not fully compatible in our current implementation. It is easy to generate editable code from an interactive design while the reverse is difficult. This makes some programming operations irreversible. This problem can be solved by refining the interface of the shader creation. A better solution is to create a Domain Specific Language based on javascript and write a compiler to support the user-customized program.

Another main limitation is that the specification of animation and interactions is not supported by our system. However, it is easy to implement some simple and fixed interactions, such as

selection and linked brush, as we can attach meta data to existing data tables by the global states when interactions occur. The meta data can be used to specify colors or other visual encodings to indicate the selection. But still the way to specify more complicated interactions remain unknown. It is similar for animation, that VisComposer can achieve animation effects by adding extra data and controls to represent a gradual transform. But it requires the design of an extremely complicated scenegraph.

9. Conclusions

This paper presents a visual authoring environment that enhances the conventional visualization pipeline with an abstractive design structure and programmable scheme. The contribution is a novel visual programmable design scheme that visually represents the modularized visualization pipeline and allows for an interactive management and composition of the visualization process. The implemented system not only allows easy visualization design and development, but also provides a mechanism for managing the designs of the associated visual resources in a single environment. One of its distinctive features is its inclusion of textual programming to support the programmable development of novel visualization effects. The interactivity of the program enables fast content development, quick design iteration, and customizable effects. Finally, the results made with our system can be easily integrated into the workflow of other visualization programs.

Acknowledgments

This work is supported by National 973 Program of China (2015CB352503), National Natural Science Foundation of China (61772456, U1609217).

References

Bavoil, L., Callahan, S.P., Crossno, P.J., Freire, J., Scheidegger, C.E., Silva, C.T., Vo, H.T., 2005. Vistrails: Enabling interactive multiple-view visualizations, in: Visualization, 2005. VIS 05. IEEE, IEEE. pp. 135–142.

Boshernitsan, M., Downes, M.S., 2004. Visual programming languages: A survey. Citeseer.

Bostock, M., Heer, J., 2009. Protovis: A graphical toolkit for visualization. IEEE Transactions on Visualization and Computer Graphics 15.

Bostock, M., Ogievetsky, V., Heer, J., 2011. D^3: Data-driven documents. IEEE Transactions on Visualization and Computer Graphics 17, 2301–2309.

Card, S.K., Mackinlay, J.D., Shneiderman, B., 1999. Readings in information visualization: using vision to think. Morgan Kaufmann.

Castno, I., Horowitz, D., 2006. State of the art cross platform shader development with fx composer 2, in: Proceedings of ACM SIGGRAPH Courses.

Claessen, J.H., Van Wijk, J.J., 2011. Flexible linked axes for multivariate data visualization. IEEE Transactions on Visualization and Computer Graphics 17, 2310–2316.

Fekete, J.D., 2004. The infovis toolkit, in: IEEE Symposium on Information Visualization, IEEE. pp. 167–174.

Gurd, J., Bohm, W., Teo, Y.M., 1987. Performance issues in dataflow machines. Future Generation Computer Systems 3, 285 – 297.

Haeberli, P.E., 1988. Conman: a visual programming language for interactive graphics, in: ACM SigGraph Computer Graphics, ACM. pp. 103–111.

Heer, J., Bostock, M., 2010. Declarative language design for interactive visualization. IEEE Transactions on Visualization and Computer Graphics 16, 1149–1156.

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

Javed, W., Elmqvist, N., 2012. Exploring the design space of composite visualization, in: Visualization Symposium (PacificVis), 2012 IEEE Pacific, IEEE. pp. 1–8.

Lee, B., Kazi, R.H., Smith, G., 2013. Sketchstory: Telling more engaging stories with data through freeform sketching. IEEE Transactions on Visualization and Computer Graphics 19, 2416–2425.

Mei, H., Ma, Y., Wei, Y., Chen, W., 2017. The design space of construction tools for information visualization: A survey. Journal of Visual Languages & Computing .

Méndez, G.G., Nacenta, M.A., Vandenheste, S., 2016. ivolver: Interactive visual language for visualization extraction and reconstruction, in: Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems, ACM. pp. 4073–4085.

Myers, B.A., 1990. Taxonomies of visual programming and program visualization. Journal of Visual Languages & Computing 1, 97–123.

Myers, B.A., Goldstein, J., Goldberg, M.A., 1994. Creating charts by demonstration, in: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, ACM. pp. 106–111.

Najjar, W.A., Lee, E.A., Gao, G.R., 1999. Advances in the dataflow computational model. Parallel Computing 25, 1907–1929.

Ren, D., Höllerer, T., Yuan, X., 2014. ivisdesigner: Expressive interactive design of information visualizations. IEEE Transactions on Visualization and Computer Graphics 20, 2092–2101.

Roberts, J.C., 1998. Waltz: an exploratory visualization tool for volume data using multiform abstract displays, in: Visual Data Exploration and Analysis V, International Society for Optics and Photonics. pp. 112–123.

Satyanarayan, A., Heer, J., 2014a. Authoring narrative visualizations with ellipsis, in: Computer Graphics Forum, Wiley Online Library. pp. 361–370.

Satyanarayan, A., Heer, J., 2014b. Lyra: An interactive visualization design environment, in: Computer Graphics Forum, Wiley Online Library. pp. 351–360.

Satyanarayan, A., Moritz, D., Wongsuphasawat, K., Heer, J., 2017. Vega-lite: A grammar of interactive graphics. IEEE Transactions on Visualization and Computer Graphics 23, 341–350.

Satyanarayan, A., Russell, R., Hoffswell, J., Heer, J., 2016. Reactive vega: A streaming dataflow architecture for declarative interactive visualization. IEEE Transactions on Visualization and Computer Graphics 22, 659–668.

Stolte, C., Tang, D., Hanrahan, P., 2002. Polaris: A system for query, analysis, and visualization of multidimensional relational databases. IEEE Transactions on Visualization and Computer Graphics 8, 52–65.

Tatarchuk, N., 2004. Rendermonkeytm: An effective environment for shader prototyping and development .

Trifacta, 2015. the vega visualization grammar. `http://trifacta.github.io/vega`.

Victor, B., 2013. Drawing dynamic visualizations. `http://vimeo.com/66085662`.

Viegas, F.B., Wattenberg, M., Van Ham, F., Kriss, J., McKeon, M., 2007. Manyeyes: a site for visualization at internet scale. IEEE Transactions on Visualization and Computer Graphics 13.

Weaver, C., 2004. Building highly-coordinated visualizations in improvise, in: Information Visualization, 2004. INFOVIS 2004. IEEE Symposium on, IEEE. pp. 159–166.

Wilkinson, L., 2005. The Grammar of Graphics. Springer.

Yu, B., Silva, C.T., 2017. Visflow - web-based visualization framework for tabular data with a subset flow model. IEEE Transactions on Visualization and Computer Graphics 23, 251–260.

Yuan, X., Guo, P., Xiao, H., Zhou, H., Qu, H., 2009. Scattering points in parallel coordinates. IEEE Transactions on Visualization and Computer Graphics 15, 1001–1008.