VisComposer: A Visual Programmable Composition Environment for Information Visualization

Category: Research

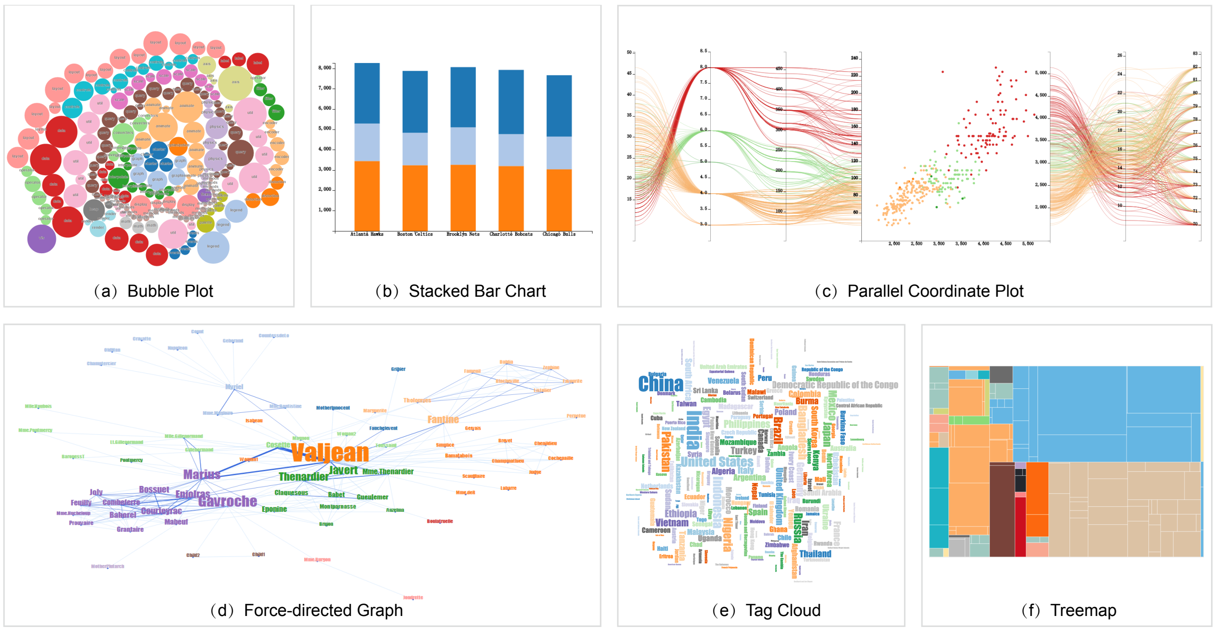


Fig. 1. Examples of the visualization designs.

Abstract—We present a programmable integrated development environment (IDE), VisComposer, that supports the development of expressive visualizations 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. We demonstrate the performance of VisComposer with a variety of examples.

1 Introduction

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 [6, 7, 9, 10, 18, 19]. Unfortunately, such frameworks demand comprehensive visualization and coding skills and require users to develop visualization from scratch. As such, these visualization programming toolkits and languages require a large amount of effort for developing appropriate visualization solutions, thereby making visualization nearly intractable for non-programming experts. Furthermore, conventional programming toolkits or languages seldom integrate with a What You See Is What You Get editor. Thus, they rarely support interactive configurations, thereby stifling collaboration between coders and visual artists in a shared and unified environment.

As such, a recent trend in information visualization is focused on creating interactive visualization design environments that require little to no programming [13, 30]. Tools such as Lyra [27] and iVisDesigner [23] have been created as visualization production tools that allow users to create sophisticated layouts and transformations that are enabled via transformation pipelines. Unfortunately, these visualization production tools offer support for only a small portion of visual forms, thereby greatly limiting the design space. As such, expressive visualizations for multivariate and heterogeneous datasets that can easily be created with visualization languages or toolkits like D3 [10] and Processing [7], can be difficult to realize in many current visualization production tools. For instance, Lyra [27] only operates on tabular datasets, and iVisDesigner [23], while being very efficient at generating coordinate systems, does not support the construction of recursive drawings such as treemaps. 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 data set becomes increasingly complicated or even intractable.

Unfortunately, the extremes of a fully programmable solution and a purely interactive non-programmable design environment have major shortcomings. In this paper, we propose a hybrid solution in the form of a novel programmable integrated development environment (IDE), VisComposer. Such hybrid solutions have a long history in the graphics community. For example, with the development of shading languages [14, 25] and programmable graphics hardware, shader-based programs replaced the traditional fixed rendering pipeline to achieve more comprehensive effects [15, 24]. To further improve performance and flexibility, many interactive shader development environments, such as RenderMonkey [28] and FX Composer [12], were created to provide graphic artists with an IDE that served as both a production tool and a programmable interface.

VisComposer 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. Our hybrid solution improves on previous work by enlarging the design space of visualization available to the user. This is enabled through a custom scene graph in which every component of the visualization process can be configured, edited, and shared. To facilitate this process, we modularize the visualization design process into components that can be manipulated individually and composed to ensure the efficient, customized, and synchronized production of information visualization. We also adopt the design concepts of shaders from computer graphics in the context of information visualization, where each node in our scene graph can be defined as a piece of program that creates a specific visualization effect. VisComposer not only provides a drag-and-drop visual editor for rapid prototyping of data-driven visualization, but also enables the customization of special effects through textual programming. Our system compares favorably with existing IDEs [23, 27] in that it allows for pre-defined visualization creation, while enhancing this through programmable operations, thereby creating a visualization composition language and a visual composer that supports the easy integration of various visualization components, as demonstrated with several examples shown in Figure 1. Our contributions include:

- A visualization composition model that modularizes the visualization design process and abstracts the resultant visualization as a scene graph, in which nodes and edges are editable and programmable.

- 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.js [10].

- An integrated visual composition environment that empowers users with the ability to directly program visualization effects and immediately view the results of their programming operations. In this way, we can reduce the time needed for development while still providing the flexibility for novel visualization design. Such an environment enables the quick iteration of novel designs that cannot be encapsulated in a non-programmable environment.

2 Related Work

Visualization Programming Frameworks and Languages Over the years, a variety of frameworks and languages have been developed with varying degrees of uptake [17, 33]. The Infovis toolkit [16] is an interactive graphics toolkit written in Java that provides a large set of visualization components and supports fast dynamic queries. A distinctive feature of the Infovis toolkit is that it enables many user interactions, e.g. fisheye lenses, dynamic labeling, etc. Improvise [32] is another attempt to provide developers with a fully-implemented Java software architecture with a declarative visual query language. It enables users to build and browse highly-coordinated visualizations through a visual interface. The focus of Improvise is on user-driven exploration of complicated datasets across multiple views. Flare [6] is an ActionScript library for creating visualizations that runs in the Adobe Flash Player. It supports data management, visual encoding, animation, and interaction of a list of visual forms. Prefuse [19] 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. [11]. Processing [7] is a programming language that was initially designed as a software sketchbook to teach computer programming fundamentals within a visual context. Due to its simplicity, it is widely used by programmers, designers and visual artists. ProtoVis [9, 18] is a declarative, domain-specific language for constructing interactive visualizations across platforms. A scenegraph is employed to organize the visual design. D3.js [10] 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 with varying degrees of expertise 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 programming languages. However, researchers have also recognized that there is a large overhead in using these programming languages to create visualizations from scratch.

Integrated Development Environments for Visualization The need for a faster visualization development time has led to the development of integrated environments for visualization [8, 21, 22]. Commercial information visualization software and tools, such as Tableau and Manyeyes [31], provides 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 [27] 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 [29] which then enables the sharing and reuse of the visualization product. Similarly, iVisDesigner [23] supports the interactive design of expressive visualization for heterogeneous datasets, covering a broad range of the information visualization design space. Other tools include Ellipsis [26], 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.

One main advantage of integrating programmability into the graphical development environments is its flexibility and expressiveness. For example, in computer graphics and scientific visualization, shader programs [15, 25] are typically written to apply transformations to a large set of elements, e.g., to each pixel in a window. This is very suitable for parallel processing, and most GPUs have multiple shader pipelines to facilitate this parallelism, vastly improving computation throughput. Our approach shares the same benefit in that a visualization operation written within the interactive visualization environment can be applied to many data points, avoiding individual specifications of the intended operations.

3 Design

Roughly speaking, the conventional visualization pipeline [11] 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, the user can view, navigate, and control the data states and operations, making a closed feedback loop in the interactive design.

Throughout the previous sections of this paper, we have often referred to inspiration from the computer graphics community with regards to IDE development and the use of programmable shaders. However, this is an imprecise analogy between the visualization pipeline (Figure 2 (a)) and the graphics pipeline (Figure 2 (b)). The

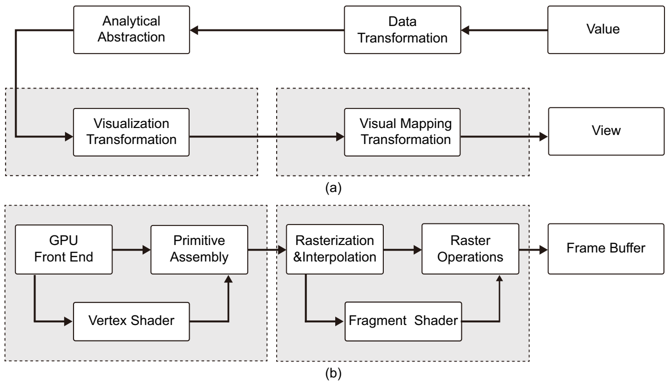


Fig. 2. (a) The conventional visualization pipeline; (b) The standard graphics rendering pipeline. Both pipelines can be seen as taking a list of primitives as the input, and then mapping them from data to a visual representation.

graphics pipeline defines an operation flow that converts a group of vertices, textures, buffers, and states into an image [14], while the visualization pipeline converts a wide variety of data into image. Although the modern programmable graphics pipeline [25] contains multiple stages (e.g., the input assembler, the vertex shader, the hull shader, the tesselation stage, the geometry shader, the rasterizer, the pixel shader, and the output merger), its main data flow is similar to that of the fixed-function graphics pipeline: transforming the vertices and shading the fragments, which conceptually corresponds to the data transformation and visual mapping stages in the visualization pipeline.

The computer graphics community has had a large amount of success in utilizing programmable shaders, which is again analogous to the use of scripting languages for creating information visualizations. 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 pipeline [12, 28]. 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. However, in the visualization community, existing IDEs are targeted to create visualizations without writing any code. While this has proven effective for a variety of visualization designs, more complicated data structures and visualization effects prove difficult in the current environments. For example, iVisDesigner [23] struggles with recursive visualization structures, such as treemaps.

We argue that designing a specialized visualization is analogous to the development of a shader program in terms of flexibility and expressiveness. As such VisComposer takes its cues from IDEs in the computer graphics community and can be seen as a programmable IDE for information visualizations. We introduce a new visualization composition model to support modularized management, specification and modification in designing visualizations (Section 4). We also formulate the design of expressive visualizations as an interactive composition of visualization operations bound to selected and filtered data (Section 5). Each operation can be exposed to the visualization authors for management, programming, and reuse, just as shaders enable programmability in the graphics pipeline. The implemented system, VisComposer, realizes these abstractions within an integrated visual composition environment, which includes a set of control widgets, visual data views, operations and designs, and text editors (Section 6).

We do not intend to design a visualization language. Instead, our effort is based on well-established visualization languages and toolkits and can be regarded as complimentary to existing systems.

4 The Visualization Composition Model

Designing a visualization for a particular dataset from scratch is an extremely complicated process. To accomplish this task, the designer needs to first choose or craft a visual representation that is suitable for the data and then map the data onto the graphical primitives that the visual form will be composed of. These choices, mappings and compositions can be thought of as a visualization composition process. In VisComposer, we define this process as a model consisting of the following modules:

The **Resource** module consists of four classes of information: the underlying data, a set of basic primitives, a set of composition operators and a list of visual forms. The *input dataset* is represented with a specified format which contains the data and extra information that defines the data types. The *primitives* denote the basic graphical presentations for constructing a visualization. They can be evaluated and rendered with respect to different platforms. For instance, our implementation employs the basic SVG primitives introduced in HTML5. Each primitive contains a name, a set of visual properties, and links to the bound data. Each *composition operator* defines the layout and organization of the primitives. A *visual form* refers to a design template that consists of a group of primitives and their compositions. The visual forms can be created, stored, and altered for fast prototyping and sharing.

The **Visualization** module denotes the result of applying a visualization pipeline or a visual form (template) to an input dataset. In principle, a visualization is composed of graphical primitives by means of the composition operators. Thus, composing the primitives with the assistance of the composition operators is the basic task in designing a visualization.

The **Scenegraph** module is a structural abstraction of a visualization. It employs a tree structure to encode how the primitives are organized in the drawn view. We employ the tree organization to allow for composition of multiple visual representations in a visualization. Each node of a tree is bound to selected data items and primitives while each link represents the data transformations and primitives arrangement of linked nodes. Nodes and links are created using primitives and composition operators of the **Resource** module, respectively. They are editable and programmable.

The **Transformation** module denotes a workspace that accommodates three types of user-controllable operations: filtering data, specifying the visual properties of primitives, and binding selected data items, primitives and composition operators to corresponding elements of a scenegraph.

Figure 3 (a) illustrates the relationships among the four modules. After a dataset is loaded, it is filtered, enumerated or selected into many groups of data items. In the meantime, the set of primitives can be selected and arbitrarily combined with the composition operators. The entire composition process includes two parts. The first part specifies or modulates the organization of the intended visualization within the **Scenegraph**, and the second part transforms, binds, and specifies the visual mapping of the scenegraph elements within the **Transformation**.

The proposed composition model is different from the ones used in iVisDesigner [23] or Lyra [27] which allow the user to directly manipulate the graphical elements in the visualization canvas. Our model allows for both the manipulation of elements on a canvas while providing an additional design space through the Scenegraph and Transformation modules. While the direct manipulation mode enables dynamic design and facilitates a What You See Is What You Get visual editor, it is inefficient to design complicated visual forms (e.g., recursive drawings) or specify visual styles for massive individual visual elements (e.g., specifying colors of 1000 primitives based on a certain criteria). Our model explicitly employs an editable abstraction to the visualization thereby enabling a flexible customization of special visualization effects and eliminates the workload of manual specification of visual designs with selected programmability.

We illustrate our model by creating a sample visualization of a 4D dataset containing 150 points (see Figure 5). In this example, we have chosen a scatterplot matrix to be the visual form. The Resource module inputs the dataset and passes the dimension number (i.e., 4) to the Scenegraph module. Two types of primitives (point and axis) and the composition operators (uniform subdivision) are selected for

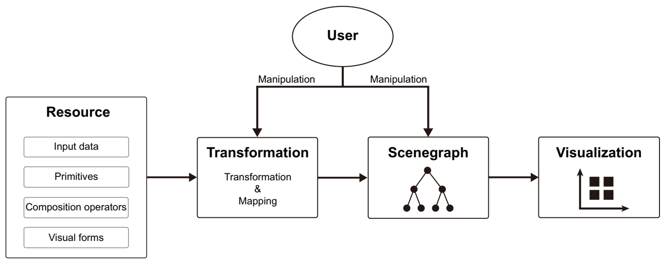


Fig. 3. The visualization composition model consists of four modules.

composing the visualization in the Scenegraph and Transformation modules. In particular, the Scenegraph module generates a 3-level tree: the root node represents the entire visualization, the middle level nodes encode $4 \times 4 = 16$ matrix cells, and the leaf nodes correspond to $4 \times 4 \times 150$ primitives. The links of the Scenegraph encode the 3-level embedding structure of the scatterplot matrix, and determine the layout, composition and coordinate systems of the visualization. The Transformation module binds the nodes and edges of the Scenegraph to selected and filtered data, and specifies the visual properties of the primitives.

5 The Visualization Composer

Our goal is to design a web-based system that facilitates interactive manipulation and customization of visualizations. For the sake of simplicity, we describe our visualization composer by using the notations of Javascript. We employ the rules of SVG elements in HTML5 as the graphical primitives which encode data with visual properties, such as shape, path, size, color, position, transparency and orientation. The primitives can be inherited and extended to construct complicated forms.

5.1 Operations

The composition of a visualization is accomplished with a non-linear timeline, in which the user can iteratively manipulate the data, primitives and visual designs across different modules. The required operations can be classified into six categories: **Filter**, **Abstract**, **Bind**, **Map**, **Ensemble**, **Draw**. Below we elaborate on each category of operation.

5.1.1 Filter

The *filter* operations filter the input data in the Transformation module with pre-defined operations, e.g., aggregating along one data dimension, or computing the variance of a set of data items. The operation parameters are user-adjustable. The filter operations can be automatically adaptable for different data types, i.e., numerical, categorical, ordinal or textual. Representative operations include item selection, arithmetics, attribute filtering and analysis. In particular, the *selection* operations choose arbitrary data attributes or subsets of data items. The *arithmetic* operations perform the numerical computations of data items. The *attribute filtering* operations support the extraction of qualified data items based on user-defined criteria. The *analysis* operations apply statistical, text analysis or data mining algorithms to selected data items. All these operations can be modulated by the user on-the-fly.

5.1.2 Abstract

The *abstract* operations construct a scenegraph structure for the intended visualization in the Scenegraph and Transformation modules. A scenegraph is instantiated as a tree with only the root corresponding to the entire visualization when a dataset is loaded. The construction process can be performed top-down, or bottom-up.

A node of a scenegraph contains two parts: the bound data and the primitives. A global coordinate system is dispatched to the root node, and its properties (e.g., the origin, the axes, the labels) can be specified either programmatically or interactively by the user. Then, the links and nodes are recursively specified. Typically, a link under a node is used to specify the organization form between the node and its child nodes. It is accomplished by specifying the composition operators of the Resource module. Possible composition and decomposition modes include: juxtaposition, superimposition, overloading, and nesting [20]. For instance, the dimensions (e.g., 4 and 4) of the Iris dataset [3] 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. Also, the decomposition can be uniform or non-uniform, depending on the user intention. Note that, only after the links under a node are bound to data and primitives can its child nodes can be specified. A local coordinate system may be built for each child node. The data items and primitives bound to the child nodes can be identical or varied. For instance, specifying the color of primitives associated with 1000 nodes can be very tedious; however, in this system, the nodes can inherit idenitical colors, or the user can specify various colors within a selected group of the primitives.

The constructed scenegraph can be stored as a template of a visual form, which can be restored and revised later. Therefore, loading or selecting a visual form for subsequential design can be regarded as a template-based implementation of the abstract operations.

5.1.3 Bind

The *bind* operations bind the nodes and links of the scenegraph with selected data and primitives in the Transformation module. Again, the binding process can be interactively or programmatically performed. Typically, the filter operations on selected data items are used prior to the bind operations.

5.1.4 Map

The *map* operations specify the visual properties of primitives (e.g., position, size and color) in the Transformation module. The specifications are enabled with a suite of adjustable palettes, together with ordinal and quantitative scaling tools (e.g., linear, log, exponential).

5.1.5 Ensemble

The *ensemble* operations evaluate a scenegraph and organize the attributed primitives for drawing in the Visualization module. They are invoked whenever a modification to the Scenegraph module, the Resource module or the Transformation module occurs. By recursively traversing the constructed scenegraph *top-down* with a depth-first mode, the bound primitives of nodes and links are evaluated and stored as a drawing list, which enumerates all graphical primitives with stacked transform information.

5.1.6 Draw

The *draw* operations show the designed visualization in the visualization canvas. It accepts the drawing list built by the Ensemble operations and renders the graphical primitives using platform-specific renderers, such as Vega [29].

5.2 The Composition Procedure

Our visualization composer encapsulates the operations in four modules (**Resource**, **Scenegraph**, **Transformation**, and **Visualization**) by leveraging an interactive design framework (see Figure 4). Similar to the graphics library OpenGL [25], in the visualization composer we employ a *state machine* mechanism to store and transfer the state of each element. The state of an element includes a set of variables that describes the visual properties, and a set of functions in the form of javascript that transforms the element. A state can be exposed to the user, edited in the programmable stage, and transferred across different modules. When an element is constructed or modified in the composition process, its state is automatically pushed into a global stack that is kept by the composer. The state of each element can be passed into other elements for inheritance. In the entire composition procedure, cross-module state synchronization and transfer are supported.

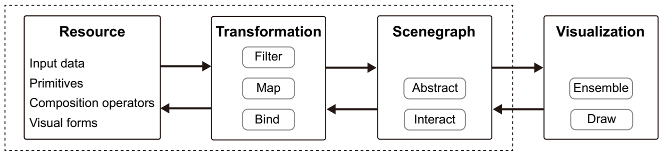


Fig. 4. The operational flowchart of the visualization composer.

Conceptually, the final visualization is composed by three components: the visual form, construction component, the rendering code generation component, and the final production component. In the first component, the visualization composer assembles all elements of the Scenegraph module and the Transformation module, and generates a topological organization that describes the directed input-output links between elements. In the second component, the elements of each module are translated into JavaScript code. 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. The third component, compiles the produced code within the web browser using the evaluate function (eval) of JavaScript and displays the visualization. The user can then search for errors and modify the results.

5.3 Programmable Visualization Shaders

We enable programmability by allowing the user to craft a piece of javascript code for each element of the Scenegraph module and the Transformation module. Specifically, the following schemes are employed to enable a visual programmable composition environment.
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 5 (f)).
- 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 5 (g)).
- 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 5 (h)). A bock-based shader offers the most flexible programmability, and can fulfill many kinds of special effects (e.g., a treemap layout algorithm).

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** The input state denotes the custom built designs of elements in previous composition steps. It is passed across four modules. The user can freely modulate an input state in writing a visualization shader. The input state is implicitly accessible if the user composes the visualization by direct manipulation in the Transformation, Scenegraph and the visualization modules.
- **The Output State** The modulated state in a visualization shader 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.
- **The Global State** A global state is instantiated when constructing the root node of the Scenegraphh 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.

6 INTERACTIVE VISUALIZATION DESIGN ENVIRONMENT

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

6.1 The Interface

The interface (Figure 5) 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 5 (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.

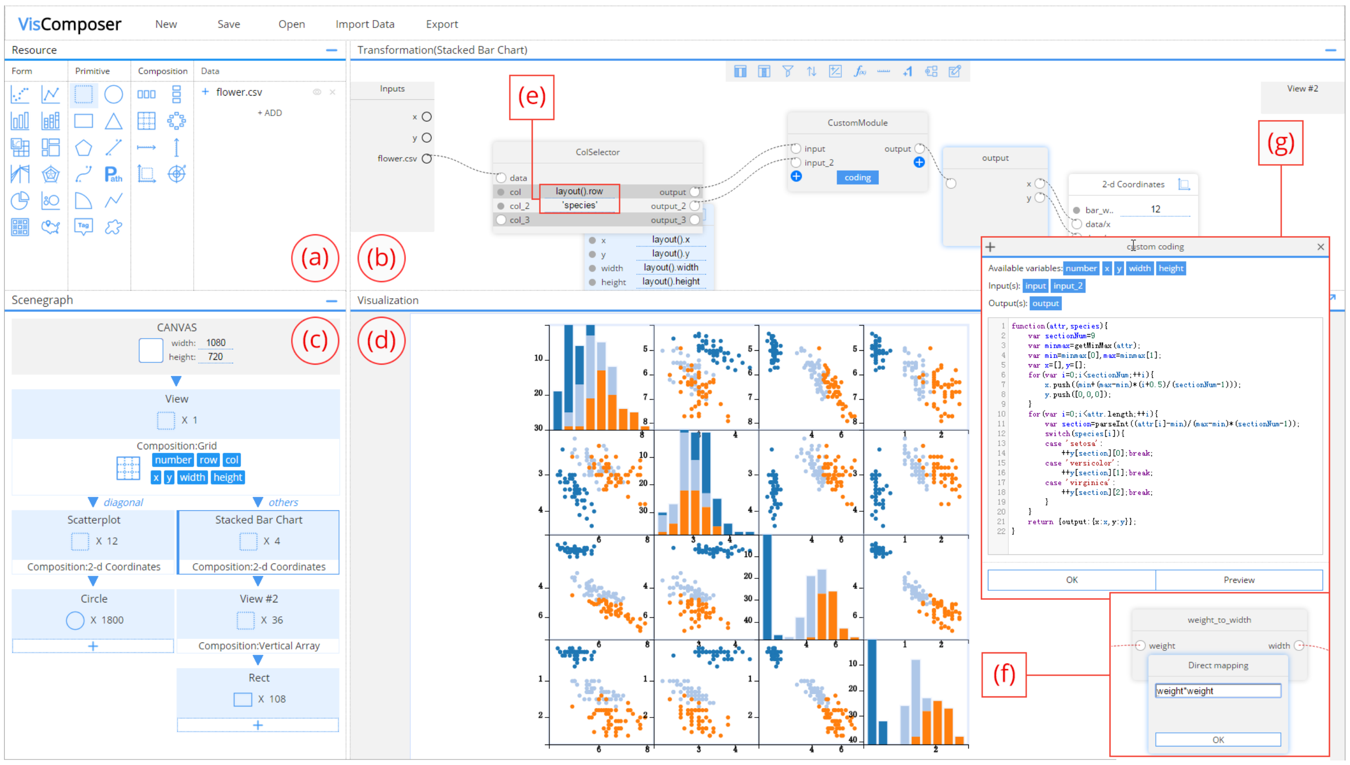
The **visualization** view (Figure 5 (b)) is the container of the visualization being designed. It is updated whenever a modification is made during the design process. To enable quick design iterations, the view is synchronized with the creation of the scenegraph. A proxy geometry or graphical glyph 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 propertie. Interactive specification and manipulation of the graphical primitives or forms in the view is allowed.

The **scenegraph** view (Figure 5 (c)) shows the scenegraph structure of the visualization and enables many user interactions that represent the abstract and interact operations defined in Section 5.1. 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 **transformation** view is the workbench for performing the bind, map and abstract operations. A suite of textual and configuration panels (Figure 5 (d)) are provided for user modulations. The user can craft transformations by editing ones provided by the system or write a visualization shader.

6.2 User Interactions

To compose a visualization, the user can perform the 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.
Selection: Two types of selection tools are provided to facilitate 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, the user can select the data items or primitives by double clicking on the nodes or links in the scenegraph view.



```
function(attr, species){
    var sectionNum=9
    var minmax=getMinMax(attr);
    var min=minmax[0],max=minmax[1];
    var x=[],y=[];
    for(var i=0;i<sectionNum;++i){
        x.push((min+(max-min)*(i+0.5)/(sectionNum-1)));
        y.push([0,0,0]);
    }
    for(var i=0;i<attr.length;++i){
        var section=parseInt((attr[i]-min)/(max-min)*(sectionNum-1));
        switch(species[i]){
        case 'setosa':
            ++y[section][0];break;
        case 'versicolor':
            ++y[section][1];break;
        case 'virginica':
            ++y[section][2];break;
        }
    }
    return {output:{x:x,y:y}};
}
```

Fig. 5. The interface of VisComposer: (a) the resource view, (b) the transformation view, (c) the scenegraph view, (d) the visualization view, and (e) the code editor window of a custom transformation. The scatterplot matrix representation of Iris Flower Dataset is displayed.

Drag-and-drop: To transfer selected items in or across different views, the user can drag the object from the original view to the destination.

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 the user quickly specify properties of a new visual structure and refine them if needed. A set of docking points are utilized for the purposes of positioning and designing composition modes.

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. The user can view the programmed effects by examining individual operations in the visualization view.

Annotations: The user can identify and analyze the results through free-style annotations. Iconic and textual annotations are supported.

6.3 Two 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 the user to build elements from scratch. Generally, the user follows the flowchart depicted in Figure 4 to craft a visualization. For modification, the user 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, the user can bypass the tedious setup steps and dive straightly into the visualization design process.

6.4 The Implementation Details

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 about the input data, such as the data types and dimensions, are kept and can be used in the composition process.

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 supports 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.

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 the user can quickly compose a visualization with user-friendly interactions and obtain real-time feedback in the visualization module.

Serialization: A visualization is composed of a list of JavaScript objects that are referenced to each other. Similar to iVisdesigner [23], 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 the user can only save part of the pipelines or a sub-tree of the scenegraph into an offline file. Other users can import the file to re-use the component as part of the current design.

Reuse and Export: VisComposer supports the reuse and exportation of visulaizations 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.

7 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.

7.1 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 the user easily specify a grid array in the second level of the scenegraph, and then specify the visualization of each cell individually. Figure 5 shows a scatterplot matrix representation of the Iris Flower Dataset [3]. 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.

7.2 Bubble Plot

The bubble plot representing the *Flare class hierarchy*[1] is shown in Figure 1 (a). Each bubble represents an entity in the Flare library. Because the bubble layout is complex and can be accomplished by interactively specifying interactive 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.js. Additional visual properties are added such as colors and text labels.

7.3 Stacked Bar Chart

A stacked bar chart is a bar chart that encodes an additional dimension by stacking it along certain axis (Figure 1 (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.

7.4 Parallel Coordinate Plot

VisComposer supports the construction of multiple coordinate systems, like the parallel coordinate plot and the radar chart. Figure 1 (c) presents an example of an advanced parallel coordinate visualization design [34]. The *Auto MPG Dataset*[2] 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 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.

7.5 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 1 (d) shows the character co-occurrence in the *Les Misérables Dataset*[3]. 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. VisComposer also supports the usage of the third-party libraries, which may speedup the design process. In this example, the link width is intended to encode a dimension of the input data. With VisComposer, the user 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.

7.6 Tag Cloud

VisComposer supports the visualization of a collection of words with the tag cloud techniques. Figure 1 (e) is an example of the tag cloud based on the *Countries and Dependencies by Population Dataset*[4]. 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.

7.7 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 1 (f) shows the visualization of the *Public Company Bankruptcy Cases Dataset* [5], 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.

8 EVALUATIONS AND DISCUSSIONS

All examples were created on a PC equipped with a Dell display (24-inch LCD with resolution of 1920× 1080 pixels) and Google Chrome 41.

8.1 User Evaluation

8.1.1 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. 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 brower. Feedback is collected with the online survey platform Kuiksurveys (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.

[1]Flare Class Hierarchy, `http://bl.ocks.org/mbostock/4063269`. (last accessed: 2015.03.31)

[2]Auto-MPG Data, `https://archive.ics.uci.edu/ml/datasets/Auto+MPG`. (last accessed: 2015.03.31)

[3]Victor Hugo's Les Misérables, `http://bl.ocks.org/mbostock/4062045`. (last accessed: 2015.03.31)

[4]Countries and dependencies by population, `http://en.wikipedia.org/wiki/List_of_countries_and_dependencies_by_population`. (last accessed: 2015.03.31)

[5]Public Company Bankruptcy Cases Opened and Monitored for Fiscal Year 2009, `http://catalog.data.gov/dataset`. (last accessed: 2015.03.31)

Visual forms	Datasets
Stacked Bar Chart	[3]
Parallel Coordinate Plot	[1]
Force-directed Graph	[5]
Squarified Treemap	[4]
Tag Cloud	[2]

Table 1. Tasks for user evaluation.

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 participant 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.

8.1.2 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 $\sim$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.

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

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

8.1.3 Quantitative Evaluation

We performed an additional quantitative evaluation to compare the performance and usability of VisComposer with other approaches. After the qualitative evaluation, we chose 3 of the 6 skilled participants and asked them to create a scatterplot matrix based on the Iris dataset [3]. They were asked to create the visualization in 3 different ways: hand-drawing the scatterplot in Adobe Illustrator, programming the scatterplot with D3.js and designing the scatterplot in VisComposer. We compared the time spent with two other tools in Table 3. Results indicate the high efficiency of VisComposer.

Participant ID	Tool	Approximate time
A	Adobe Illustrator	1 hour 10 minutes
B	D3.js	49 minutes
C	VisComposer	21 minutes

Table 3. Performance comparison for crafting a scatterplot matrix with three approaches.

8.1.4 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 the user 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.

8.2 Limitations

Although the programmability favors the design of all visual effects, there are several limitations caused by the design complexity and the capabilities of the implemented system.

- Our current implementation uses a tree-based scenegraph to structure the visualization. This limits the flexibility of the creation process. On one hand, the tree structure requires a directional transfer of the visualization design. However, 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.
- Animation and transition are not supported in VisComposer. Although VisComposer can achieve animation effects by adding extra data and controls to represent a gradual transform, it requires the design of an extremely complicated scenegraph. A potential solution is to provide additional widgets to allow for modify the Transformation and the Scenegraph modules in runtime. Unfortunately, it remains difficult to represent the needed operations efficiently.
- 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.
- Our current implementation does not provide history views and undo operations. This issue can be addressed by preserving global states and snapshots with pruning optimization techniques.

- In terms of performance, VisComposer employs SVG as the renderer to make it compatible with browsers. However, it results in a relatively slow performance compared with the means with Canvas or WebGL techniques. We plan to provide additional rendering options by leveraging Canvas and WebGL.
- It is challenging to construct complex forms using basic JSON primitives. Paths may help but using paths needs extra support like the SVG libraries in D3.js. Fortunately, the programmability of VisComposer allows easy importing from other JavaScript libraries.

8.3 Discussions

Existing interactive design environments [23] [27] 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.

On the other hand, Lyra and iVisdesigner support direct manipulation on the visualization canvas. VisComposer additionally abstracts the visualization with a scenegraph, which can provide an alternative means of designing visual transformations and visual mappings. This is particularly useful in situations where an intermediate result is needed for effective development.

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.

References

[1] Auto-mpg data, https://archive.ics.uci.edu/ml/datasets/Auto+MPG. (last accessed: 2015.03.31).

[2] Countries and dependencies by population, http://en.wikipedia.org/wiki/List_of_countries_and_dependencies_by_population. (last accessed: 2015.03.31).

[3] Iris data set, http://archive.ics.uci.edu/ml/datasets/Iris. (last accessed: 2015.03.31).

[4] Public company bankruptcy cases opened and monitored for fiscal year 2009, http://catalog.data.gov/dataset/public-company-bankruptcy-cases-opened-and-monitored-for-fiscal-year-2009. (last accessed: 2015.03.31).

[5] Victor hugo's les misérables, http://bl.ocks.org/mbostock/4062045. (last accessed: 2015.03.31).

[6] *Flare*. http://flare.prefuse.org, 2015.

[7] *Processing*. http://processing.org, 2015.

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

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

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

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

[12] I. Castno and D. Horowitz. State of the Art Cross Platform Shader Development with FX Composer 2. In *Proceedings of ACM SIGGRAPH Courses*, 2006.

[13] 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, July 2011.

[14] R. Cortes and S. Raghavachary. *The RenderMan Shading Language Guide*. Course Technology PTR, 2007.

[15] K. Engel, J. M. Kniss, M. Hadwiger, C. Rezk-Salama, and D. Weiskopf. *Real-time Volume Graphics*. AK Peters, 2006.

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

[17] P. E. Haeberli. Conman: a visual programming language for interactive graphics. In *Proceedings of ACM SIGGRAPH*, pages 103–111, 1988.

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

[19] J. Heer, S. K. Card, and J. A. Landay. Prefuse: a toolkit for interactive information visualization. In *ACM SIGCHI*, pages 421–430, 2005.

[20] W. Javed and N. Elmqvist. Exploring the design space of composite visualization. In *IEEE Pacific Visualization Symposium*, pages 1–8, 2012.

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

[22] B. A. Myers, J. Goldstein, and M. A. Goldberg. Creating charts by demonstration. In *Proceedings of ACM SIGCHI*, pages 106–111, 1994.

[23] D. Ren, T. Hollerer, and X. Yuan. iVisDesigner: Expressive Interactive Design of Information Visualizations. *IEEE Transactions on Visualization and Computer Graphics*, 20(12):2092–2101, 2014.

[24] C. Rieder, S. Palmer, F. Link, and H. K. Hahn. A Shader Framework for Rapid Prototyping of GPU-based Volume Rendering. *Computer Graphics Forum*, 30(3):1031–1040, July 2011.

[25] R. J. Rost. *OpenGL Shading Language (3rd Edition)*. Addison-Wesley, 2009.

[26] A. Satyanarayan and J. Heer. Authoring Narrative Visualizations with Ellipsis. *Computer Graphics Forum*, 2014.

[27] A. Satyanarayan and J. Heer. Lyra: An interactive visualization design environment. *Computer Graphics Forum*, 2014.

[28] N. Tatarchuk. RenderMonkey: An Effective Environment for Shader Prototyping and Development. In *Proceedings of ACM SIGGRAPH Sketch*, 2004.

[29] Trifacta. *The vega visualization grammar*. http://trifacta.github.io/vega, 2015.

[30] B. Victor. *Drawing Dynamic Visualizations*. http://vimeo.com/66085662, 2015.

[31] 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):1121C1128, Nov 2007.

[32] C. Weaver. Building highly-coordinated visualizations in improvise. In *IEEE Symposium on Information Visualization*, pages 336–343, 2004.

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

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