Visualizing Vega's Scenegraph and User Interaction

Jane Hoffswell
University of Washington
Seattle, Washington
jhoffs@cs.washington.edu

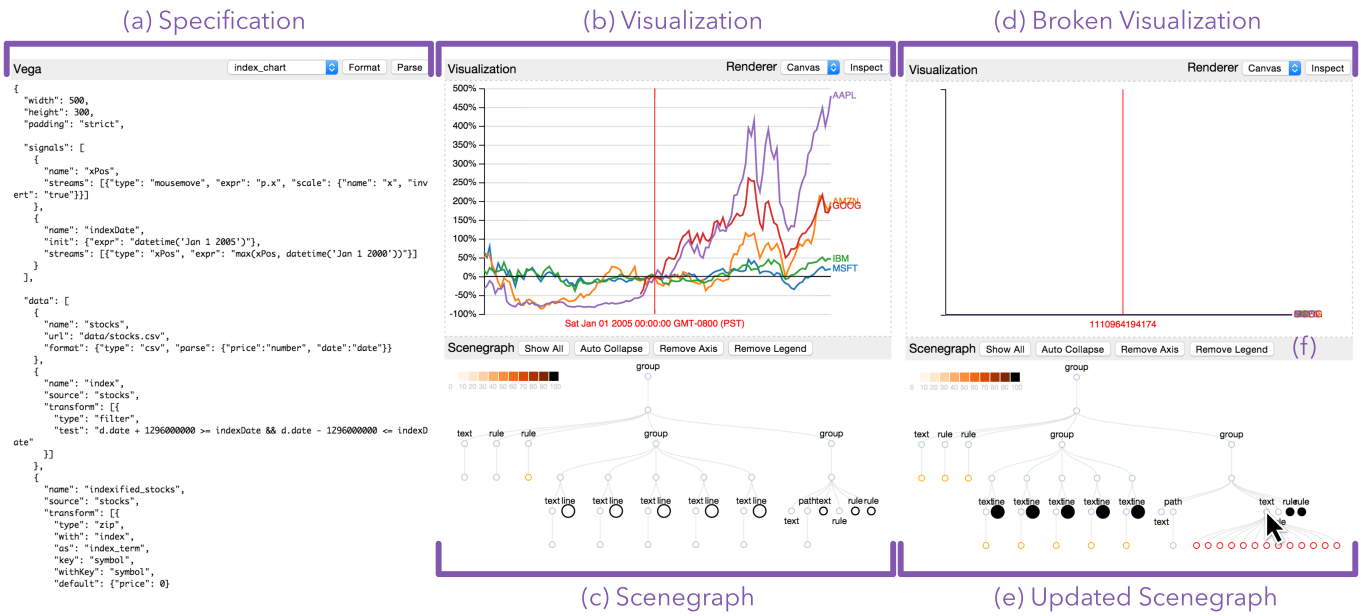

Figure 1: The modified Vega development environment including the (a) specification, (b) visualization, and (c) scenegraph representation of an index chart. (d) The broken visualization for the index chart after user interaction and (e) the updated scenegraph showing all nodes for the axis removed after expanding the scenegraph node. (f) Control buttons for updating the scenegraph visualization.

ABSTRACT

Vega is a declarative visualization language that enables rapid iteration while allowing designers to focus on visual encoding instead of low-level implementation details. However, the separation of specification from execution inhibits debugging effectiveness due to obfuscation of the underlying code. Visualizing Vega's scenegraph enables rapid exploration of the visualization structure and underlying data. As end-users interact with the visualization, the scenegraph updates dynamically to show changes in the underlying data. Inspection of the visualization enables users to identify related elements in the scenegraph. Visually surfacing system internals enhances user understanding and facilitates the identification of errors that would otherwise require complex debugging cycles.

ACM Classification Keywords

H.5.2. Information Interfaces and Presentation (e.g. HCI): User Interfaces

INTRODUCTION

Declarative languages accelerate the development process by decoupling specification from execution, but this often comes at the cost of effective debugging strategies since the underlying execution is obfuscated [4]. Vega is a declarative visualization grammar that builds on D3 with a high-level JSON-based specification language. The simplified specification enables rapid iterative exploration of the design space and enables designers to focus on the visual encoding rather than low-level implementation details. Declarative languages promote code reuse since components of a specification can easily be retargeted to incorporate new datasets or different visual encodings. Vega has been extended to support declarative design of end-user interactions and retargeting enables reuse and modification of interaction techniques [7]. By separating specification from execution, language developers are free to implement new optimizations without inhibiting the designer's process. However, this separation obfuscates the underlying program state thus inhibiting the developer's ability to evaluate and debug the output.

D3 partially addresses this issue by using a declarative framework to map attributes to the document object model [1]. By leveraging the browser's native representation, D3 can support common web-based debugging strategies. Immediate evaluation simplifies debugging by surfacing errors as they occur instead of propagating them through a hidden control flow. However, these forms of direct inspection can require additional domain expertise. While Vega's delayed evaluation often complicates the debugging process, the advantages of the simplified JSON representation outweigh this added complexity.

To address Vega's complicated debugging cycle, this work shows how a visualization of the scenegraph can facilitate exploration of the underlying data and identification of errors. Dynamic updates to the scenegraph show how the data changes based on end-user interactions. Inspection of the visualization helps users identify corresponding areas within the scenegraph. This paper discusses an example use case of the scenegraph for identifying an error in an implemented Vega specification for an index chart. Interviews with real-world developers help motivate design decisions and suggest areas for future work.

RELATED WORK

As noted above, D3 attempts to address some of the tradeoffs of declarative visualization languages by utilizing a declarative framework within the document object model in order to leverage the benefits of the browser's native representation [1]. However, these advantages come at the cost of a more complex representation that requires additional domain expertise. Vega builds on D3 to provide a higher-level specification language, but in separating specification from execution adds a layer of obfuscation that limits debugging effectiveness [8]. The separation of specification and execution is a problem for other declarative languages beyond the field of visualization, and Perfopticon starts to address this issue for database queries [6].

Visual representations of program state can provide developers with relevant context to better understand and interact with their code. Visualization of program behavior is a common strategy in education for demonstrating how code is executed. Automatic processes such as [3] step through program execution and visualize how data is changing. However, standard visualization techniques for visualizing program behavior often lack scalability to real world scenarios. Recent work has addressed how to record and replay program execution in the browser for more effective debugging [2]. Other visualization techniques target real world use cases but limit the range of questions that a developer can answer about their code [5].

METHODS

Vega is a declarative visualization language which separates specification from execution, but limits debugging effectiveness due to obfuscation of the underlying code. Vega parses a JSON specification into a data flow graph representing the execution pipeline. Data tuples are pushed through the data flow graph to generate a scenegraph that is rendered into the final visualization (see Figure 2). The designer can use the JavaScript console to access some information about the underlying system, but such inspection requires knowledge about Vega's proprietary internal structure. For designers without this expertise, the specification and visualization are the only resources for debugging (see Figure 3). This project introduces the scenegraph into the designer's development cycle to provide insight into the underlying structure and data transformations.

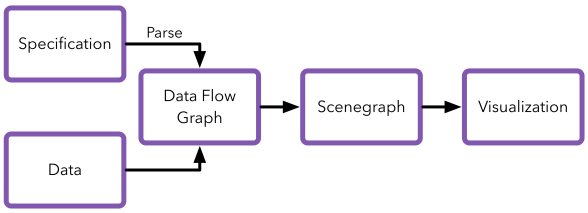

Figure 2: The execution pipeline of Vega.

Figure 3: The execution pipeline of Vega according to external users.

Data Flow Graph

The data flow graph represents Vega's internal execution structure and is directly connected to the user-defined specification. Nodes in the graph correspond to different components of the specification and are responsible for transformations to or propagation of data tuples. Though the data flow graph was not used for this project, it is a relevant component of future work.

Scenegraph

Vega constructs and updates the scenegraph based on user interaction in the online editor. The visualization extracts the scenegraph each time an update happens and processes the difference by comparing the nodes with the previous data. Calls to update are throttled until the user stops interacting with the visualization in order to limit the number of necessary computations. The difference calculation compares the scenegraph from the start of the interaction with the scenegraph representing the visualization in its final state. Once the data has been extracted and updated, the scenegraph is drawn using D3's tree layout. Inspection of elements in the visualization is handled using internal Vega functions for comparing the bounds of scenegraph items with an input point.

RESULTS

In this section I outline the features of the scenegraph visualization and briefly discuss areas of improvement and existing limitations.

Scenegraph Tree Visualization

The scenegraph is a tree structure that describes how to render the output visualization. Prior to this work, users could explore the scenegraph tree via a textual, nested representation in the console. However, the tree representation enables more direct inspection into nodes of interest within the underlying structure.

Tree Interactions

The primary interaction on the tree is the ability to expand and collapse nodes. When the scenegraph is first displayed, the structure is simplified by collapsing all nodes where the number of children exceeds some predefined threshold (see Figure 1c). Future work should examine techniques to provide an improved smart default representation. When a node is collapsed in the tree, the stroke of the node is drawn in black and the size of the node represents the number of hidden children within the collapsed node. Buttons in the "Scenegraph" bar allow the user to more rapidly expand nodes and otherwise simplify the display of the tree layout (see Figure 1f).

Data on Demand

Right clicking nodes in the scenegraph prints the node to the JavaScript console and binds it to the variable `global`. This form of interaction is the main entry point for accessing the internal data for the scenegraph.

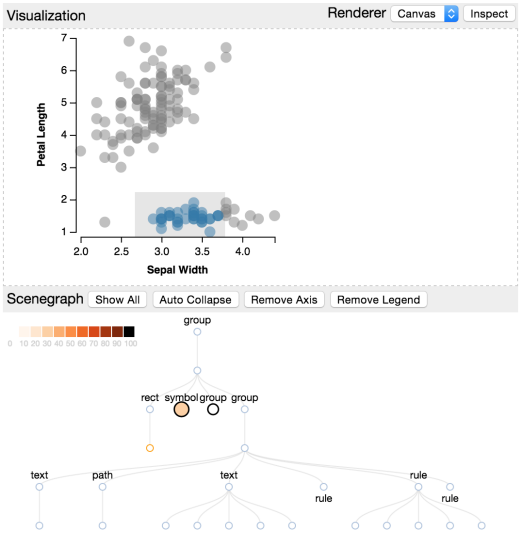

Figure 4: Between 20%-30% of nodes changed based on the user selection of points.

Color Encoding

Completing end-user interactions on the visualizations or reparsing the specification updates the scenegraph to show how data has changed. Nodes have four possible statuses after an update (added, removed, modified, or none). The status of a particular node is currently encoded by the stroke color of that node. However, based on feedback from the poster session, a symbol encoding may better represent this data and remove confusion with the percentage color encoding described next; For collapsed nodes in the scenegraph, the fill color of the node represents the percentage of descendants that have been updated in some way (see Figure 4).

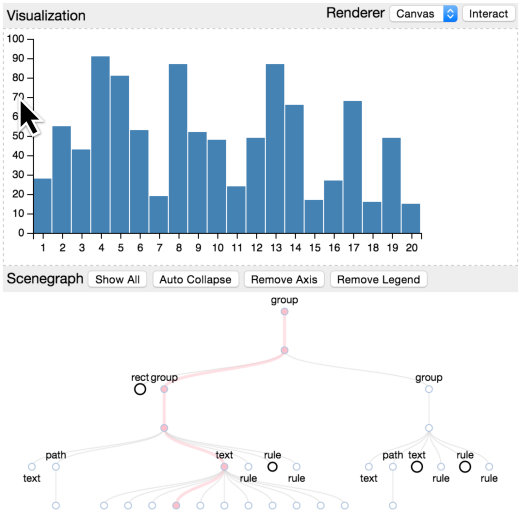

Figure 5: Selecting the "70" text label on the visualization highlights the corresponding components of the visualization.

Inspection

Changing to the "Inspect" mode allows users to select elements in the visualization and see corresponding components in the scenegraph (see Figure 5). This form of inspection uses the bounds of the scenegraph nodes to determine if the point of interest is contained within the element. Vega's `pick` function did not seem to select components beyond visual marks (i.e. it did not identify axis elements) and only identified a single node, not the entire path. Currently, this form of inspection does not always work as expected and therefore requires additional modification to resolve bugs as future work.

DISCUSSION

Before implementing the scenegraph visualization I conducted interviews with external Vega users to gain a better understanding of the problems faced by current users. Based on the results of these interviews, I implemented the scenegraph and interaction techniques described above. Finally, I used the scenegraph to explore an actual error in a Vega specification in order to discuss the advantages and limitations of this visualization.

User Interviews

Prior to the start of this project, I conducted interviews with external Vega users to identify common debugging strategies and areas of the development cycle that would benefit from improvement. Vega's JSON representation facilitates programmatic generation so we hypothesized that most specifications would be programmatically generated rather than written by hand. In interviewing Vega users, I observed that many users do in fact generate specifications within their external system. However, when a specification has an error beyond a simple syntax error, the user often switches to the

online editor to manually debug the result. There are two debugging strategies that are employed at this point: *iteration* and *inspection*. The scenegraph visualization presented in this paper augments both types of debugging strategies.

For the *iterative* debugging strategy the user manually updates the specification and parses it to examine changes in the result. This strategy can result in a simplified specification where the user iterates to find the smallest specification that produces the error. This strategy requires a significant amount of manipulation of the specification by hand, which can be a tedious process for complicated visualizations. To better support iterative debugging, the scenegraph tracks changes when the user re-parses a specification. Updates are shown in the scenegraph and thus allows the user to quickly identify the impact of a change.

In the *inspection* debugging technique, the user accesses the underlying representation via the JavaScript console. In some cases, the JavaScript console will contain an error that provides insight into the source of the bug. Otherwise, users will use the console to explore the scenegraph or data flow graph. However, most users noted that they did not take much advantage of this information in the debugging cycle. Though inspection is a common debugging strategy for the developers of Vega, the required domain expertise makes it hard for external users to utilize this information. However, by visually representing the scenegraph, Vega users can more rapidly access information that was otherwise hidden in the proprietary internal format.

In these interviews I also elicited feedback on possible improvements to the overall development process. The two most largely requested features were brushing & linking between specification, visualization, and underlying execution structures and a representation of how data changes over time. The scenegraph attempts to address these concerns by enabling linking between the scenegraph and visualization and by enabling inspection of the underlying data through nodes in the scenegraph.

Example Use Case

To demonstrate the advantages of the visual scenegraph representation, I used it to help diagnose an actual error in a Vega specification for an index chart. An early iteration of Vega's index chart contained a bug that caused the visualization to improperly display the lines (see Figure 1d). Prior to diagnosing this error, we identified the following lines as the potential source of the bug:

```
{ "name": "index",
  "source": "stocks",
  "transform": [{
    "type": "filter",
    "test": "d.date + 1296000000 >= indexDate &&
             d.date - 1296000000 <= indexDate"
  }]
}
```

Making this hypothesis required close inspection of the specification and sufficient familiarity with the example to understand the intended behavior of this code. Our hypothesis was that this calculation was not correctly binning the data since the hard coded date offset (`1296000000`) would not work correctly for certain months. We hypothesized that using Vega functions such as `month` would help to resolve the problems with this binning calculation.

To test this hypothesis, I used the scenegraph to explore changes in the data while interacting with the index chart. I started by finding a point where the visualization stopped behaving correctly, as in Figure 1d. It is evident from both the visualization and scenegraph that all the lines still exist in the visualization at this point, though we can observe that all the axis marks have in fact been removed (see Figure 1d, e). I then selected the index point text mark and determined the date of this point: `Wed Jan 16 2002`. I compared the index point to the difference calculation to observe that the bounds where `indexDate` was the center would extend from the `1st` to the `31st` for this point. Thus, the hard coding appears to be creating a buffer of two weeks on either side of the input date.

To put this date in context, I identified additional points in the visualization where the data was incorrect and compared the index date at those points. In making this comparison, I noticed that many of the dates that caused the visualization to break were the `16th`.

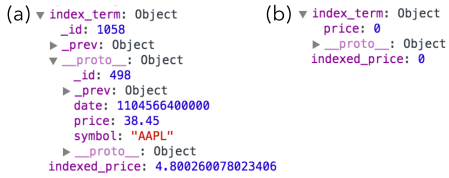

Figure 6: (a) Data printed to the console from nodes in the scenegraph. (b) Data from the same node at the broken index point.

At this point, I expanded one of the lines and started inspecting the data of the points. By comparing the internal data with the data at points where the visualization was working correctly, I noticed that the data of the lines was not being correctly set beyond the default (see Figure 6). This observation suggested that there was an error in how the `index_term` was calculated, which is related to the specification code identified above.

In examining the input data file, I noticed that every data point was for the first of the month, so this calculation should be binning the data by month. At this point, I tested the following update to the specification using Vega's `month` function:

```
{ "name": "index",
  "source": "stocks",
  "transform": [{
    "type": "filter",
    "test": "month(d.date) == month(indexDate) &&
             year(d.date) == year(indexDate)"
  }]
}
```

This update to the specification removed the error. However, I still wanted to get a firm diagnosis as to why the previous specification was broken. Returning the specification to the

previous state, I identified the point before the broken index date. I selected the first node of the first line and grabbed the index date for that point: `Tue Jan 01 2002 00:00:00 GMT-0800 (PST)`. Next, I computed the range of values using the original formula and compared them to the broken index point:

```
date       = Tue Jan 01 2002 00:00:00 GMT-0800 (PST)
high       = Wed Jan 16 2002 00:00:00 GMT-0800 (PST)
low        = Mon Dec 17 2001 00:00:00 GMT-0800 (PST)
indexPoint = Wed Jan 16 2002 08:42:42 GMT-0800 (PST)
```

In comparing these points, we actually see that the index point does *not* fall within the bounds as expected since the time of the `indexPoint` throws off the calculation that should otherwise be based on month and year.

This example shows that the main advantage of the scenegraph is that it provides an entry point for the underlying data. Though we were able to identify the source of the error prior to debugging it with the scenegraph visualization, accessing the data was useful for diagnosing why the specification was incorrect. We can hypothesize that the process mentioned above would have facilitated identification of the error had we not already identified a potential source. However, future work is required to sufficiently test the effectiveness of this debugging strategy by external Vega users.

FUTURE WORK

The interviews with external Vega users identified two areas that would benefit from future work: brushing & linking and representations of data changes. Interviewees noted that brushing and linking specification, visualization, and visual representations would be beneficial as it would shrink the gap between components thus enabling more rapid exploration and manipulation of the specification. The inspection mode of the visualization starts to implement this idea, but many individuals at the poster session and through interviews have noted that linking to the specification would be even more beneficial for implementing changes. This form of linking would have been particularly useful for the use case explored above because it could have directly shown the connection between the data in Figure 6 and the part of the specification responsible for calculating the `index_term`.

The most requested feature during the interviews was the ability to observe changes to the underlying data. This request was supported by the use case above in which we observed that the data was the most relevant component for debugging the visualization. While the scenegraph acts as an entry point to the underlying data, future work should explore more direct ways to present this information. The internal data comes in multiple forms (data driving the visualization and data representing the visualization), and the data is transformed by the data flow graph and through end-user interaction. Both types of data are potentially relevant for debugging broken visualizations, so it will be necessary to design a representation that can display changes to both types of data.

The data flow graph may provide insight into how best to link the specification to the visualization. Furthermore, the data flow graph may provide an entry point for examining how data transformations update the data. However, future work is required to identify how best to utilize the data flow graph in the development process. Finally, additional user studies need to be conducted to determine the effectiveness of these debugging strategies in real world development processes.

CONCLUSION

While Vega is useful because it enable rapid iteration and allows designers to focus on visual encoding over implementation, it obfuscate the underlying execution thus limiting debugging effectiveness. Visualizing Vega's underlying scenegraph provides users with an entry point for exploring internal data and making relevant connections between components of the execution process. Interacting with a dynamic scenegraph can provide useful insights into how data is manipulated, thus allowing users to more easily diagnose errors in their specification.

ACKNOWLEDGMENTS

The author thanks Jeffrey Heer, Dominik Moritz, Jeff Snyder, Arvind Satyanarayan, and the students of CSE512 for their feedback and support of this project. The author also thanks the interviewees for their insights on the Vega development process.

REFERENCES

1. Bostock, M., Ogievetsky, V., and Heer, J. D3: Data-driven documents. *IEEE Trans. Visualization & Comp. Graphics (Proc. InfoVis)* (2011).
2. Burg, B., Bailey, R., Ko, A. J., and Ernst, M. D. Interactive record/replay for web application debugging. In *Proceedings of the 26th ACM Symposium on User Interface Software and Technology* (St. Andrews, UK, October 8–11, 2013), 473–484.
3. Guo, P. J. Online Python Tutor: Embeddable web-based program visualization for CS education. In *Proceedings of the 44th ACM Technical Symposium on Computer Science Education*, SIGCSE '13, ACM (New York, NY, USA, 2013), 579–584.
4. Heer, J., and Bostock, M. Declarative language design for interactive visualization. *IEEE Trans. Visualization & Comp. Graphics (Proc. InfoVis)* (2010).
5. Lieber, T., Brandt, J. R., and Miller, R. C. Addressing misconceptions about code with always-on programming visualizations. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '14, ACM (New York, NY, USA, 2014), 2481–2490.
6. Moritz, D., Halperin, D., Howe, B., and Heer, J. Perfopticon: Visual query analysis for distributed databases. *Computer Graphics Forum (Proc. EuroVis) 34*, 3 (2015).
7. Satyanarayan, A., Wongsuphasawat, K., and Heer, J. Declarative interaction design for data visualization. In *ACM User Interface Software & Technology (UIST)* (2014).
8. Vega: A Visualization Grammar. **http://trifacta.github.io/vega**, April 2014.