Inspecting and Debugging In-Browser D3 Visualizations

Elaine Xiao*
Massachusetts Institute of Technology

Kathryn Jin†
Massachusetts Institute of Technology

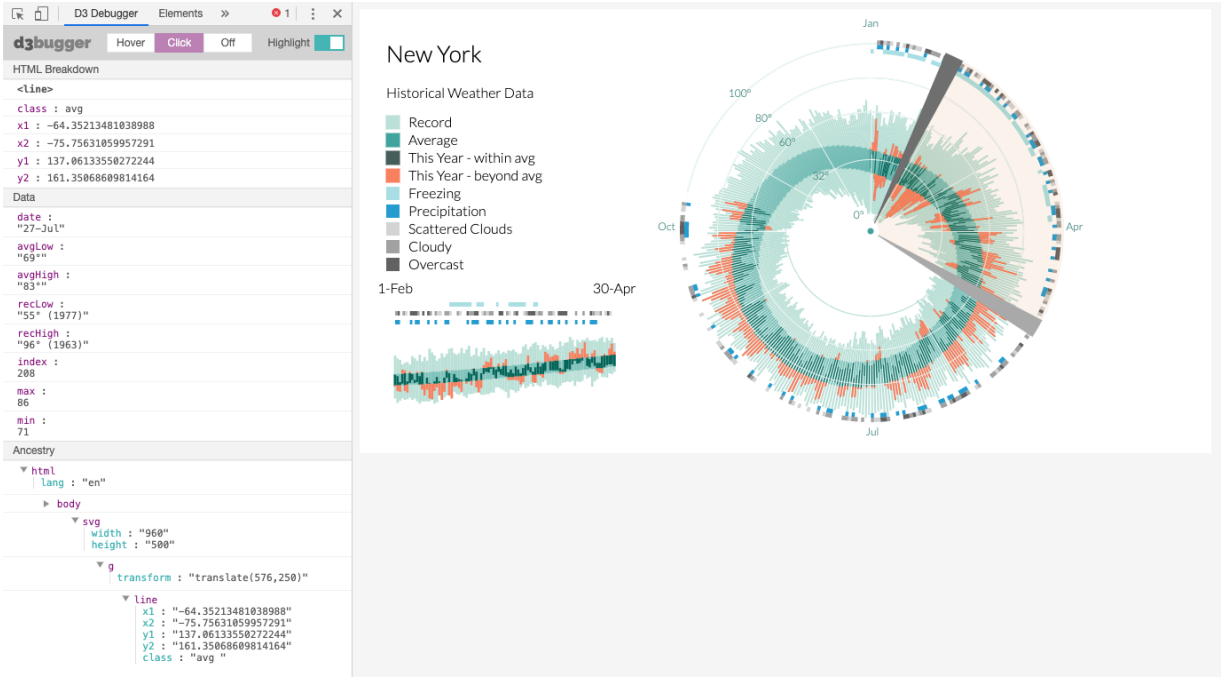

Figure 1: Demo of the Chrome extension debugging tool.

Abstract

The D3 JavaScript library has become the predominant tool for creating data visualizations using web standards. D3's adherence to the standard document object model (DOM) allows developers to use tools native to modern browsers, such as Chrome DevTools, to inspect and manipulate D3 visualizations. However, there are unique aspects of D3's representation in the DOM that are poorly handled by native tools. With d3bugger, we present a inspection tool that is custom-made for D3's DOM idiosyncrasies. Our inspection tool lets users select individual elements in a D3 visualization and easily view D3-relevant information. This tool is intended as an extension to native development tools, rather than a replacement.

Index Terms: Human-centered computing—Visualization—Visualization design and evaluation methods

1 Introduction

In the realm of data visualization toolkits, the D3 JavaScript library has become a ubiquitous tool for producing web-based visualizations. D3 is a dominant choice, as evinced by its use by major news and media companies (e.g. the New York Times), as well as the immense amount of D3 learning resources (e.g. code examples) available online.

A distinguishing characteristic of D3 is its direct manipulation of the document object model (DOM), and, correspondingly, its basis in the well-known web standards of HTML, CSS, JavaScript, and SVG [4]. As a result, D3 visualizations integrate with native, powerful developer tools such as Chrome DevTools and Safari's Web Inspector [1]. Despite the ease of integration, D3 visualizations present unique use cases of the DOM that are beyond the scope of native developer tools, which are designed for general DOM-based use cases. For example, a single SVG element in a D3 visualization could have hundreds of siblings, which obfuscates the DOM tree view of an inspected page. Beyond usability, there is also a need for improved prioritization of what information to display – certain properties of the DOM are more relevant for D3 visualizations than for typical web pages, but native tools' interface prioritizations reflect a much more general use. Thus, there is a need for a D3-specific tool that goes beyond the limitations of native development tools.

We introduce a web-based tool specifically for inspecting D3 visualizations. The design of our tool addresses the idiosyncrasies of D3 visualizations by refining the displayed information to what is most prioritized by D3 developers. Our tool takes the form of a Chrome extension that adds a panel to Chrome's native DevTools interface. Users can interact with the inspected page and see the D3-relevant properties appear in the panel. As such, this tool is useful both for exploring visualizations published online, as well as for examining and debugging work-in-progress visualizations in the D3 development workflow.

*eyxiao@mit.edu
†kjin@mit.edu

2 Related Work

2.1 Deconstructing D3 Visualizations

"D3 Deconstructor" is a tool developed and published by the VisLab at UC Berkeley [3]. The tool is a published Chrome extension that extracts data, marks, and mappings between them from D3 visualizations. The primary functionality of this tool is its "deconstruction" of any D3 visualization, where "deconstruction" means extracting the entire dataset of a given visualization into a downloadable data table. D3 Deconstructor does not support granular access to information about individual elements of the visualizations. Though this tool is useful for extracting relevant D3 information from the DOM, the gap between the extracted dataset and the visualization prevents it from meaningful use as a tool for inspection and debugging.

2.2 JavaScript Library Developer Tools

Several popular JavaScript libraries and frameworks such as React, Vue, and Grunt, and Ruby on Rails have library-specific developer tools published as Chrome extensions [2]. These extensions augment Chrome DevTools by adding an additional panel. In general, these panels provide a view of some form of DOM tree or list of components that is customized to provide library-relevant information. Tools for front-end libraries like React include inspectors in the form of mouse interactions with the inspected page. No such tool exists currently for D3.

3 Methods

At a high level, our extension injects a script into the webpage that extracts relevant information from the D3 elements in the page, then displays the information in a DevTools panel. In the following sections, we will describe our design of the DevTools panel, the details of the extension architecture and how it works, and additional implemented features.

3.1 DevTools Panel Design

We chose to display three types of information about the D3 element in our DevTools panel: a breakdown of its HTML attributes, the data bound to it, and its ancestry in the DOM. These three types of information are critical in debugging D3 visualizations, but are often difficult to efficiently locate given a specific D3 element. Thus, we decided to dedicate a section to each of these – HTML breakdown, data bound, and ancestry – in our DevTools Panel (see Figure 2).

In the HTML Breakdown section, the D3 element's SVGElement type is always shown in the first entry (e.g. rect) so users can easily locate it. The entries following that are the rest of the HTML attributes of the selected D3 element, with the attribute type colored in purple to prominently distinguish from the values on the right.

In the Data section, the entries are formatted similarly, with the data fields colored in purple to distinguish from the data values. The Data section displays all of the data bound to the D3 element, and supports nice formatting for nested data objects, as shown in Figure 2.

Finally, the Ancestry section displays all of the D3 elements' ancestors so users can quickly tell where in the DOM the D3 element is located. The HTML attributes of each of the ancestors can be accessed by toggling the arrow symbol next to the ancestor name, though they are initially not displayed (unless the user toggles the arrow) to reduce clutter in the section and allow users to clearly trace through the ancestry tree.

In the next section, we will describe the details of our implementation of the DevTools panel and the rest of the Chrome extension.

3.2 Extension Architecture Overview

The Chrome extension's architecture (see Figure 3 is based off of recommendations in the official Google Chrome Extension documentation [1]. It consists of three main components: the background script, the UI element (panel), and content scripts. Pieced together,

Figure 2: Layout of the DevTools Panel.

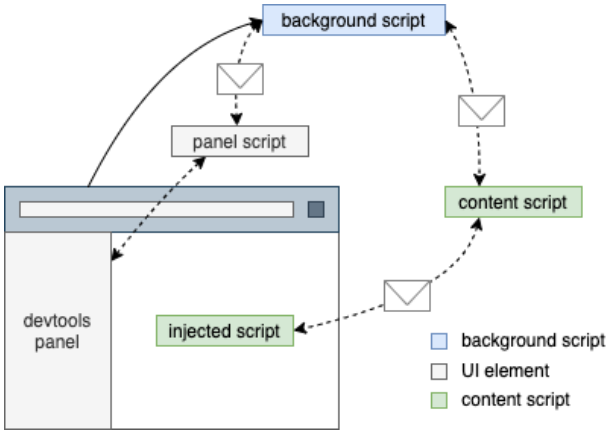

Figure 3: Overview of System Architecture.

these three components are able to get data from the webpage, process the data, then format and display it in the DevTools panel. In the next two paragraphs, we will describe these components in more detail, then give a concrete example of how exactly they work in our specific extension.

The first component is the background script. The background script is the extension's event handler and listens for browser events that are important to the extension. It also routes messages between the other components of the extension, acting as a central hub of communication between the DevTools panel and content scripts. The second component is the content script(s). Content scripts are scripts that execute in the context of the webpage – they can read and modify the DOM of the webpage. In our extension, there are two content scripts: the script injected into the webpage, and an intermediary script between the injected script and the background script. The intermediary script is necessary because scripts that are injected directly into the webpage cannot pass messages to the background script; it acts as a message channel between the injected script and the background script. Finally, the third component is the UI element, which in our extension is the DevTools panel. The DevTools panel displays the relevant information that was extracted from the webpage by the injected script.

3.3 Updating the DevTools Panel

With a little more background on the extension components, we can now delve into how the extension works. First, the extension injects a script into the webpage. This script finds all D3 elements in the webpage DOM and binds event listeners (e.g. hover, click) to these elements. When the user then hovers over or clicks a D3 element in the webpage, the event listener detects the event and triggers an event handler. The event handler extracts data from the D3 element that was hovered over or clicked, formats it into a message, and then passes the message to the intermediary content script. Once the intermediary content script receives the message, it passes the message to the background script. The background script in turn routes the message to the DevTools panel script, where the panel script processes the message, then updates the DevTools panel display to show the D3 element's information.

While the main route of information seems to flow in one direction, from injected script to intermediary content script to background script to panel script, other features in our extension that we will discuss in the next section require communication in the opposite direction, hence the double sided arrows in Figure 3.

3.4 Additional Features

To provide users with more flexibility in using the extension, we built off of the main functionality described in Section 3.3 and incorporated additional features into our extension.

3.4.1 Different Interaction Modes

In the first iteration of the extension, we started off only implementing hover interaction (i.e. using the extension, users would have to hover over a D3 element to show its information in the DevTools panel). However, we found that having hover as the only interaction mode could limit the extent to which a user can effectively utilize the extension – hover interaction makes it easy to switch to inspecting various D3 elements quickly and easily, but it also didn't suit certain types of visualizations and use cases. For example, if you want to move your cursor from the visualization to scroll through the DevTools panel, you would need to avoid hovering over any other D3 element before getting there, otherwise the panel would update with new information. In some dense visualizations, it is impossible to maneuver your way out of the visualization without hovering over other D3 elements, which makes hover interaction ill-suited for this purpose. To provide an alternative to hover interaction that would allow users to interact with D3 elements more selectively, we implemented a click interaction, and also a way to turn off the interactions completely.

We implemented switching between different interaction modes as follows. First, when a user clicks one of the interaction mode buttons in the DevTools panel, the panel passes a message to update the interaction mode to the background script, which passes the message to the intermediary content script, and then to the injected script. When the injected script receives the message, it iterates through the D3 elements in the webpage DOM and removes event listeners for the other interaction modes and adds event listeners for the selected interaction mode.

3.4.2 Highlight Toggling

Another feature that we added to the extension was highlight toggling. In the first iteration of the extension, we added a yellow highlight border around the element that the user was currently hovering over to help the user easily locate and identify the element selected. However, we realized that this could clash with the hover interaction in the original visualization – for example, if the visualization also added a highlight border over the hovered element, the highlight added by the extension might obscure the original visualization's highlighting. To solve this issue, we decided to provide the ability to toggle the extension's highlighting on and off, so that the user can choose whether or not to use the highlighting feature depending on their visualization and preferences. Highlight toggling was implemented similarly to switching between different interaction modes described in the previous section.

4 Results

After our first iteration of the extension, we received feedback and thoughts about the project from other D3 developers. Much of our feedback was positive, and many people were excited by the idea of an easily accessible debugging tool for D3. A large proportion of reviewers were interested mainly in the Data section of the DevTools panel, as the data bound to a D3 element is often the most difficult attribute to view. One of our next steps for this project would be to conduct a larger user survey, preferably among D3 developers who would use the extension while they are trying to actively debug a visualization, along with implementing additional features as described in Section 5.

5 Discussion & Future Work

There is a discrepancy between D3's integration in the DOM and the inspector tools provided by web browsers. d3bugger augments the built-in inspector and allows users to quickly explore all elements of their D3 visualization and receive relevant information to both exploratory and development purposes.

The primary benefit of this tool for D3 developers is increased efficiency. The development workflow is expedited by d3bugger, as it illuminates only relevant information in the panel interface, and its interaction modes are responsive. This tool reduces the need for D3 developers to tediously console.log() D3 selections, and saves developers time spent searching for the bound data attribute of their elements.

This D3 inspector tool also has the benefit of allowing for exploration, and even scrutiny, of published D3 visualizations. Allowing element-by-element inspection of visualizations and their bound data could expose data obfuscation or misleading encodings.

There are several possible features that we can experiment with adding to our extension in the future. For example, a feature that allows users to temporarily edit the D3 element's styles in the extension's DevTools panel, similar to the Chrome Inspector's styles panel, could be useful for users to experiment with styling. Another possible feature that takes inspiration from the Chrome Inspector is highlighting ancestor elements in the webpage when hovering over an ancestor name in the Ancestry section of the panel – this would

allow users to visualize exactly where the ancestors are located on the page. Finally, we could implement a search bar in the panel that allows users to search for and locate different types of SVGElements in the DOM for debugging purposes.

While there are many possible features that could be added to the extension, our immediate next steps would likely to be to conduct a larger user survey of our existing extension, as mentioned in Section 4, so that we can polish the existing extension and build additional features from there.

ACKNOWLEDGMENTS

Special thanks to Arvind Satyanarayan for teaching us how to design effective interactive visualizations and visualization tools (and for always having the most aesthetically pleasing lecture slides), Rupayan Neogy, Nava Haghighi, and Rishabh Chandra for their invaluable guidance and feedback on this project and throughout the semester, and the rest of the 6.894 class for their feedback and ideas on how to improve this project as well as all of the insightful discussions during lecture this semester.

REFERENCES

[1] Google Chrome. Extending devtools. Available at https://developer.chrome.com/extensions/devtools.

[2] Google Chrome. Featured devtools extensions. Available at https://developer.chrome.com/devtools/docs/extensions-gallery.

[3] Maneesh Agrawala Jonathan Harper. Deconstructing and restyling d3 visualizations. *UIST*, 2014.

[4] Jeffrey Heer Michael Bostock, Vadim Ogievetsky. D3 data-driven documents. *IEEE Transactions on Visualization and Computer Graphics*, 17, 2011.