A Block Based Language for Easy D3 Visualizations

Saketh Kasibatla

VPRI Memo M-2015-001

Viewpoints Research Institute, 1025 Westwood Blvd 2nd flr, Los Angeles, CA 90024 t: (310) 208-0524

A Blocks Based Language for Easy D3 Visualizations

Saketh Kasibatla

This writeup is the outcome of an internship with Viewpoints Research from April 1, 2015 to June 30, 2015. My work was supervised by Aran Lunzer and Alex Warth.

This quarter, I sought to outline the basics of a block based programming language in order to foster data literacy, and provide non-expert programmers with a useful set of tools to create effective data visualizations. Through various explorations and prototypes, we have arrived at a design which, I believe, with a little more fleshing out, will be a good start for the project.

All my code and mocks for the project can be found at https://gitlab.com/cdg/d3-examples, and should be accessible to any users registered on CDG's private gitlab servers.

Principles, Goals, and a bit of the Story So Far

Initially, we sought to follow a few basic principles, which we believe will facilitate user understanding of data and effective presentation of said data via interactive visualizations. We chose to base our language on D3, as it provided several interesting concepts (e.g. the data join and a declarative style of programming), along with a well architected set of utility libraries which had made creating data visualizations in JavaScript much easier. We hope to abstract the essence of D3 into a visual language, freeing it from some of the considerations which its implementation in JavaScript imposed on it.

We chose to implement our initial experiments in Snap, as it was easily extensible and was the most fully featured block based language available. We felt that we could take advantage of some of the visual error prevention features (i.e. blocks that do not fit together since their types are not compatible) and other aspects of both Snap and Scratch which had made them useful for teaching purposes, in order to make our language more accessible to non-experts.

In order to focus our exploration, we adopted a few guiding principles, which I will describe briefly.

Interactivity

Users must be able to establish a feedback loop quickly between the visual output and the code which produces that output. One of the difficulties of developing programs that are even relatively large is the difficulty of holding a model of an algorithm in one's mind. As programmers, we are trained to be able to develop discipline to be able to do so, but this is a skill that novices have trouble with. We seek to offload as much of that work as possible to our system in order to allow the user to focus on

the task at hand.

Easy Experimentation

The user may not always have a clear visualization goal in mind when using our language. Thus, we seek to make it easy for users to play with their data and make visualizations which are easy to change. We don't want the user to commit to a path that isn't fruitful too early and make an inappropriate visualization for their data.

Expressiveness

The user must not be overly constrained in the visualizations which they can produce using our tool. The main aspect of our system which distinguishes it from tools like Plotly and Tableau is the fact that the user creates data visualizations using code. We believe this gives the user the advantage of being able to make visualizations that do not follow pre-selected templates if they so choose. While we will provide templates such as bar charts to get the user started quickly, the user should be able to make a significant subset of the visualizations they could make using D3 in our language.

A Gentle Slope

While we seek to produce a tool which is expressive, so as to not constrain the user, we acknowledge that it may be difficult to bring the entirety of D3's features and richness into Snap. The solution we propose is the gentle slope. Instead of providing a small subset of D3's functionality easily, and throwing the user to the wind for the rest, we seek to make a few common cases extremely easy to work with, and add concepts or abstractions to progressively increase the discipline necessary to create more complex ideas. The user should be able to create a simpler visualization quickly, and should still be able to make a more complicated visualization if they are willing to learn a more complex system. The end of this gentle slope may be to write a visualization in plain JavaScript and D3.

Approach

We chose to flesh out ideas from two directions—both from the top down and from the bottom up. In the bottom up approach, we implemented visualizations we had written in D3 in Snap, with the code relatively close to our original reference, in order to create running prototypes. In the top down approach, we separated ourself from all implementation details and sought to imagine what our environment would ideally look like.

Some of our low level prototypes include SnapMinder, which embeds an svg element in the Snap environment, and uses blocks written entirely in Snap (using its JavaScript function block) to create a visualization in the style of the Trendalyzer, created by GapMinder; and Riemann Sum, which used Snap blocks compiled to ES6 in order to replicate a visualization of the Riemann Sum method for calculating in-

tegrals.

Both these visualizations taught us a great deal about the capabilities of Snap and proved useful in our effort to gain an understanding of the types of designs for languages we could use Snap to create, but our prototypes resembled the original JavaScript code too closely, and required far too many blocks to get the most basic visualizations up and running. This hindered interactivity, so we decided to ensure that each block added produced a noticeable visual change, so as to make the user's job easier.

As a first experiment with the high level design of the system, I outlined a system which would use programming by example in order to set up a basic skeleton which could then be expanded by the user. We quickly found a great deal of overlap with these designs and the work of many others who seek to bridge the expertise gap between easy to use tools like Plotly and excel, and tools for experts like D3, including Bret Victor and Toby Schachman. Upon realizing this, we sought to narrow our area of exploration and look more closely at the concept of a block language for data visualization, leaving integration with more complex user interfaces as further work.

The Design

The design for the system in its current form is the product of a second iteration on the high level design, and contains many elements inspired by the modular UI architecture of Facebook's “React” frontend library. Other sources of inspiration for this design are cited in the bibliography. It is important to note that as of right now, none of the mentioned aspects of the proposed system have been implemented. In some cases, I propose an implementation strategy or refer to another project which implements parts of what is mentioned. These mentions are cues towards a future prototype implementation.

This language consists of three parts—a data manipulation language, a language for visual composition, and a language to add events and behavior. Editing the code results in live changes to an environment which can either be overlaid over the Snap environment or viewed in a different window. The editor itself is organized into tabs, each for different elements of a visualization.

Data Manipulation

The data manipulation language consists of a set of blocks for relational algebra operations, basic data import and cleaning, and data manipulation via map, filter and reduce. These blocks operate on data in the form of tables. In the data tab, the user composes these operations to create basic views on data pulled from sources. For example, the countries table in figure 1 is composed of three tables joined together to form one table with many properties. These views provide a common organization for the data, which visual glyphs then use in order to display data in an orderly fashion.

Figure 1: An example of basic data views setup in the data tab

Users develop these views using a UI similar to that of Bags, a language based on Snap, which seeks to enable students to work with relational algebras. The system features a side pane which displays tables and the results (in tables) of operations when blocks are clicked on. With such a UI, a basic feedback loop should be established early between the user and the system, enabling them to begin exploring their data early. This exploration should enable users to both better understand their data and format it in a manner which will prove useful when they seek to access the data within the context of visual composition.

Visual Composition

In the current architecture, visualizations are composed of glyphs, or visual elements. These glyphs are composed of attribute settings, bindings between types of elements contained in the glyph and data, state (unique for each instantiation of a glyph), and event handlers.

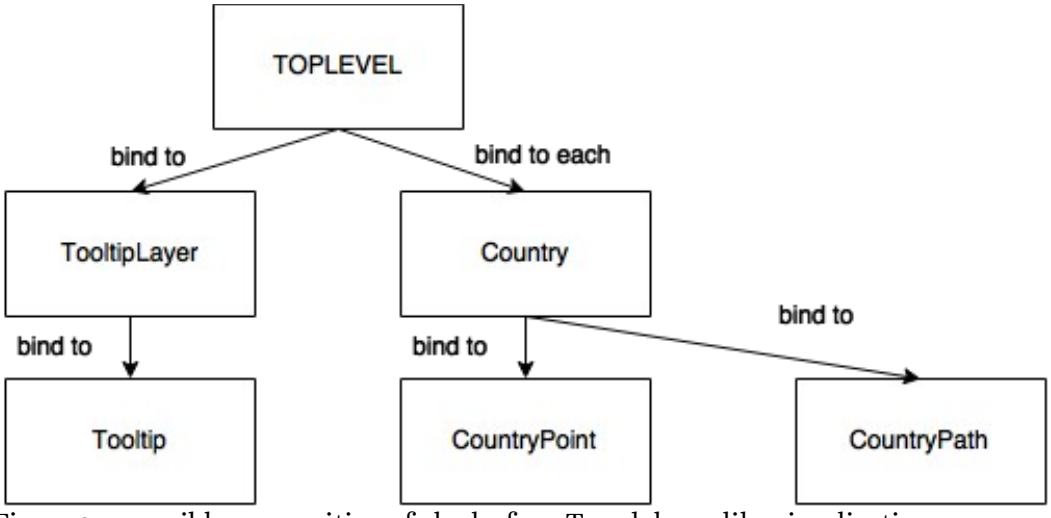

Figure 2: a possible composition of glyphs for a Trendalyzer-like visualization

In the above diagram, we see an example composition of glyphs in an implementation of our Trendalyzer-like visualization. Here, the TOPLEVEL container holds multiple countries and a global TooltipLayer, which displays tooltips at a given position. Each Country consists of a CountryPoint, a circle indicating the current posi-

tion of the country, and a CountryPath, indicating the country's position over time. Each glyph is associated with a datum, the value to which the current instance is bound. In the case of 'Country', datum consists of a table with columns 'year', 'GDP', 'lifeExpectancy', and 'population'. As shown below, 'CountryPoint' is bound to a single row of this table, the row which is indicated by yearSelector. In addition, the logic of a glyph can include events which then result in messages and visual changes to other glyphs in the composition.

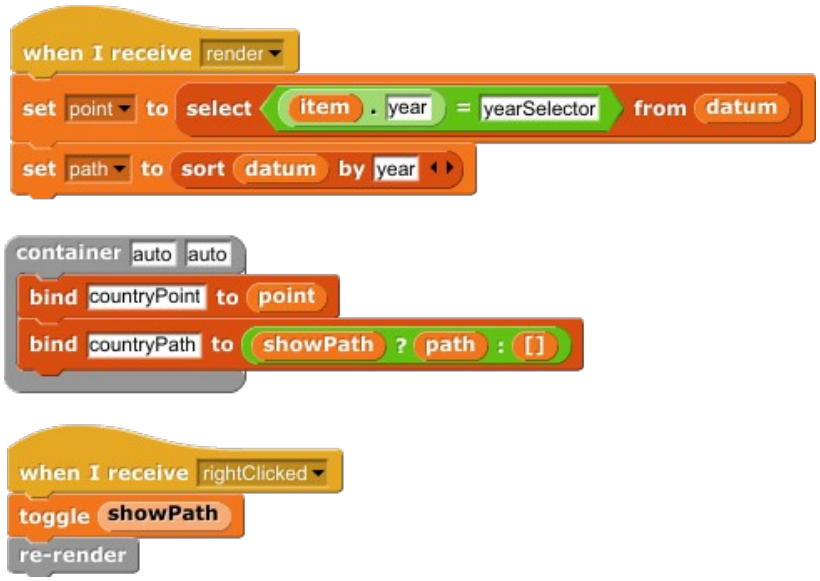

Figure 3: Sample implementation of the Country glyph from figure 2.

Above is an example of a glyph definition. Here, point and path are aspects of the state of the glyph and are treated as local variables in the Snap environment. These are updated on the synthetic render event, which is triggered by the re-render block, and when the visualization initially renders. After sending the render message, the block labeled 'container' is executed. This block contains declarative attributes of the glyph in question, including children, expressed by the bind block. Note that 'bind to' takes the provided data and passes it to a single instance of the specified glyph. 'bind to each' passes each element of the provided data array to a different glyph, in the manner of a data join in D3.

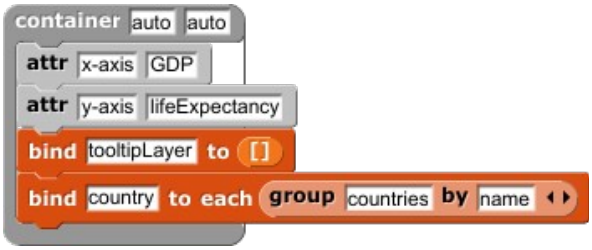

Figure 4: an example of a glyph with both attributes and children

In the above figure, we see another example of specified attributes and children. Here, 'x-axis' and 'y-axis' are both custom attributes which specify how to setup D3 scales for the x and y axes of this container, as well as what aspects of the data they correspond to. These serve to set default values for the 'x' and 'y' attributes of the container's children. Another important detail to note is that while the container element (a custom version of the D3 group) has many attributes, only two are specified here. This is because, wherever possible, we seek to infer information or to provide a reasonable default so as to allow the user to get a visualization running with minimal code.

Events and Behavior

Events are created using a scenario based programming model in a spirit similar to that outlined by Gordon, Marron and Meerbaum-Salant (2012). Upon the receipt of an event within a glyph (e.g. 'rightClick', and 'mouseOut'), the user may invoke procedural code, which has the option to dispatch further synthetic messages to other glyphs which can produce visual effects via re-rendering.

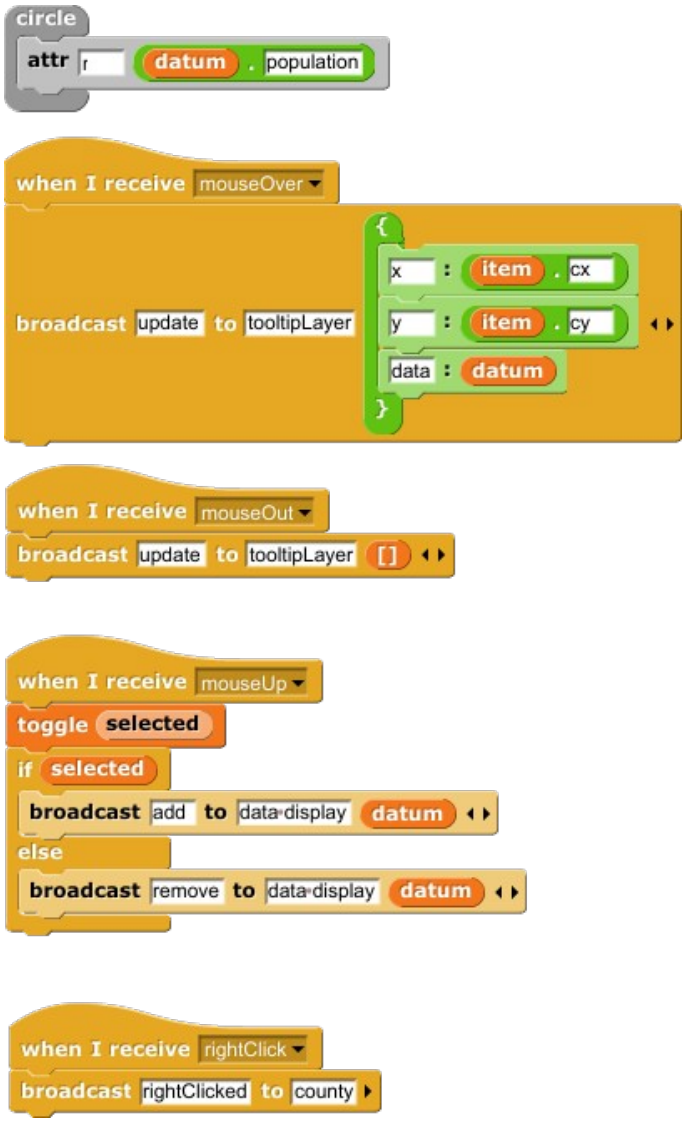

Figure 5: the code for the countryPoint glyph from our mock of a Trendalyzer-like visualization

In Figure 3, we see many such events being received by a circle, which represents a country in the Trendalyzer-like visualization. Upon the receipt of each event, a simple procedure is carried out, which may modify state and dispatch events to other glyphs so as to effect visual changes beyond the current glyph. For example, When a countryPoint receives a 'rightClick' event, it dispatches a different event ('rightClicked') to the country glyph (in this case, the one which binds it), which then may change the visibility of the path, which denotes changes in the country over time.

Visualization Environment

In order to produce interactivity with the visual environment where the effects of the user's code are being rendered, we send an AST of all the created scripts to said environment, which interprets and uses this information to render the visualization. This environment should automatically diff new ASTs with the old ones which it was rendering in order to identify a set of data joins and updates which produce the expected results. If the changes are too complex, the environment should save the old visualization on a stack of previous versions and render a fresh visualization. In addition, the user must also be able to specify old revisions when they wish so as to provide easy iteration, and history, providing a mechanism for both interactivity and avoidance of early commitment in design.

Future Work

Build a Basic Prototype

In order to continue iterating with both language and user interface ideas, we must build out the current design (or a minimal iteration on it) as a prototype. This can be accomplished by extending Snap as our programming environment, and having it communicate with a custom rendering environment of our creation. Inter-tab communication can be used to send the AST of the current project to the rendering environment, and, with minimal UI changes, we can support tabs for different glyphs.

The blocks would be compiled to JavaScript objects in a manner similar to that of the React system, with Snap code defining functions that are called by a larger system. This system would connect event handlers, provide defaults, call the rendering logic of the user's code, and automatically generate data joins based on binds.

Easing Common Approaches, Implementing the Gentle Slope

As of right now, a great deal of the user's work is being inferred, including unspeci-

fied attributes, unspecified event handlers, and specific logic for data joins. It would be valuable to allow the slow revelation of some of this inferred complexity for the user. This is accomplished in the attributes and event handlers by allowing the user to specify any given attribute or event handler. But, it is not obvious how one would go about revealing the nuance of the data joins which are being inferred via bind calls for a user who would like to perform more complex operations. This would be a useful area to explore towards our goal of a gentle slope.

Building a Custom UI/Blocks Language

While, for the foreseeable future, we will be implementing our language in Snap, as it provides a fair amount of support for our goals, is fully featured, and extensible, it may be useful to construct a blocks language which is designed specifically for visualization. This could allow us to take advantage of some of the 'visual error checking' which seems to be lost in Snap due to our non-standard semantic meanings for custom blocks.

Integrating with Others' Research

In the very long term, we may want to look at integrating our language into the wider area of tools for easy data visualization. I feel that our language serves as a useful form of semitextual interface, which allows the user to interact with an interface much closer to natural language in the pursuit of providing epistemic actions for the understanding of the user as outlined in Sedig and Parsons (2013). It could be useful to have the language back a visual interface such as Toby Schachman's Apparatus or the one presented by Bret Victor in "Drawing Dynamic Visualizations".

Bibliography

Kamran Sedig, Paul Parsons. "Interaction Design for Complex Cognitive Activities with Visual Representations: A Pattern-Based Approach". Transactions on Human-Computer Interaction, 2013.

Michal Gordon, Assaf Marron, Orni Meerbaum-Salant. "Spaghetti for the Main Course? Observations on the Naturalness of Scenario-Based Programming". Proceedings of the 17th ACM annual conference on Innovation and technology in computer science education, 2012.

Lars Grammel, Chris Bennett, Melanie Tory, Margaret-Anne Storey. "A Survey of Visualization Construction User Interfaces". Eurographics Conference on Visualization, 2013.

Lars Gramel, Melanie Tory, Margaret-Anne Storey. "How Information Visualization Novices Construct Visualizations". IEEE Transactions on Visualization and Computer Graphics, 2010.

William A. Pike, John Stasko, Remco Chang, Theresa A. O'Connell. "The Science of Interaction". Information Visualization, 2009.