GHOST: Global Hepatitis Outbreak and Surveillance Technology Feasibility Report

Team 23:
Raghav Kaul
Jeongsoo Kim
Ernest Lai
Sarthak Mohapatra
Lovissa Winyoto
Georgia Institute of Technology

In partnership with:
David S. Campo, Ph.D.
Centers for Disease Control (CDC)

Introduction

The Division of Viral Hepatitis at the Centers for Disease Control (CDC) has created a system, Global Hepatitis Outbreak and Surveillance Technology (GHOST), in which public health researchers can perform individual analyses based on collected and existing patient data. The current iteration of the user interface is a source of a number of usability and robustness issues. The interface is centered around a data visualization that forms the majority of the project. This visualization is written in one of several different frameworks that specialize in data visualizations. The two proposed alternatives are either improving upon the current code base or using the current model as a template to recreate it in a different framework. By factoring the alternatives through a set of various technical, managerial, economic, and political/cultural criteria, we determined that it would be best to refine and refactor the existing model of Global Hepatitis Outbreak and Surveillance Technology (GHOST) in order to avoid the significant costs of recreating the model.

Overview of Alternatives

As the project revolves around the creation of a data visualization that allows the user to manipulate, modify, and filter the data, the alternative solutions revolve around the framework that the visualization and its corresponding front end will be built in. The existing model currently operates using D3.js, and the alternatives to this are as follows:

A. Refine and refactor existing model
The current model using D3.js, a data visualization framework for Javascript, is a rough proof of concept and is lacking in basic functionality. Any additional features will be implemented in Javascript and appended to the current model's source code. All bug fixes and modifications will also be made directly to the current model's source code.

B: Implement new model using different framework
As the current model is still in its infancy, an alternative exists to rewrite the model using a different framework, namely Flare/Prefuse. Flare and Prefuse are Actionscript and Java APIs, respectively, for the same web graphics framework. Using the current UI as a guide, this alternative involves duplicating D3 graphics and animations in Flare for the dynamic force-directed visualization and Prefuse for the data processing.

Criteria

A. Technical

Rendering speed: The foremost criterion for the GHOST front end is the ability to quickly access, modify, and visualize the genome sequencing data. This criterion is a measurement of how quickly the framework renders large datasets inside a browser; the faster alternative provides better performance.

B. Management

Time for testing and deployment: The GHOST front end must be completed by the end of Spring 2016 due to the semester-long nature of the commitment. One way to evaluate deployment time is by finding the time required build/compile the project and run it in a browser. Note that this differs from rendering speed because it measures running time *before* visualization starts. The better alternative has the least complexity to achieve the best performance.

Interoperability with data: This criterion refers to the ability of the front end to handle data in the format the GHOST backend provides. Currently, GHOST outputs data in JSON (Javascript Object Notation). The better alternative provides more robust library functions to parse JSON strings as objects.

C. Economic

Costs to the CDC: This project has no funding and consequently has no budget. Therefore any expenses incurred will be paid out-of-pocket. This criterion measures the total costs of the project to the CDC; the alternative with the fewest costs is preferred due to the limitation in funding.

D. Political/Cultural

Development support: This criterion is a measure of how easily the code written in the chosen framework can be maintained by the CDC development team working on the project. One central component of maintainability is the availability of online help sources and tutorials in addition to the level of API documentation for the framework. The alternative with more online help sources and tutorials provides better development support in the long term.

Methods

A. Technical

Rendering speed: Performance of the visualization alternatives was evaluated by comparing the rendering time for datasets of size 500, 5000, 10000, and 20000. The two rendered visualizations were a scatterplot matrix and a stacked graph. For the main alternatives, D3.js and Flare, benchmarks were conducted as a component of an academic paper [1] presenting D3. These benchmarks were not conducted as a component of this report, but were publicly available performance comparisons of each technology, and were therefore appropriate for feasibility evaluation.

B. Management

Time for testing and deployment: The time required for testing and deploying code was compared by examining equivalent code samples produced with either technology running in a browser. It may also be estimated through the verboseness and complexity of writing code in each framework. Specifically, sample D3 and Flare codes for graph visualization were written to compare the number of lines of “boilerplate” code (code required for setup as opposed to implementation) required for each to be used in the browser. Then, Firefox Development Tools Web Profiler (a browser plugin) was used to calculate the respective loading times of the D3 and Flare alternatives. For the Flare alternative, the time required to compile the ActionScript code using the ActionScript compiler [2] was added. We will specifically use sample code found on Github using the force-directed layout libraries for each framework.

Interoperability with data: Research to find libraries or functions for parsing JSON data was conducted by reading the API documentation for D3 [3] and Flare [4].

C. Economic

Costs to CDC: The cost was calculated by visiting the manufacturer’s websites for both D3 [5] and Flare [6] and estimating the overall published costs for the features that the project would use, namely node-link diagram visualizations. Features needed for this project were found by searching the github repositories for D3 [7] and Flare [8].

D. Political/Cultural

Development Support: Extensive online research to find resources for both alternatives was conducted. For D3, we searched the D3 wiki [10], the D3 API [11], a sample code website, and an external help website for D3 [12]. For Flare, we searched the Flare API [13] and the Actionscript Help Website [14]. While searching these websites, the number of different features that were documented was also compared. In addition, we calculated the number of StackOverflow posts tagged with either framework.

Evaluation

Table 1. Results of Evaluation.

	D3.js (existing system)	Flare/Prefuse (new system)
Rendering speed	**Loads up to 3 times faster than Flare for arbitrary visualizations (graph and non-graph)**	**Increased load times due to plugin**
Time for testing and deployment	**First frame: 60-100ms Framerate: 60 fps (n<10), 10 fps(n>1000)**	**First frame: (Using command line argument "benchmark=True"): .1-.4 seconds Framerate: fully controlled during runtime/rendering**
Interoperability with data	**Yes [d3.json()]**	**Yes [JSON.parse()]**
Costs to CDC	**Free / open source**	**Free / open source**
Development Support	**Wiki documentation for in-depth explanation as well as API reference, all modules have multiple code samples, showing distinct features; StackOverflow:~16k for D3, ~1k for NVD3 (a D3 add-on), ~962k for Javascript**	**All documentation in form of API reference, ~⅔ of all modules have code samples; StackOverflow: ~30 for Flare & Prefuse, ~47k for Actionscript**

A. Technical

Rendering speed: This criterion was evaluated by reviewing existing literature comparing the rendering speed visualizations in D3.js and Flare/Prefuse. Specifically, the existing literature evaluated the time required to initialize a rendering of scatterplot matrices and stacked graphs. These criteria were measured with datasets of varying sizes. For the scatterplot matrix visualization, the initialization time for D3.js was 0.5 seconds; a 3x reduction over the Flare/Prefuse rendering time of 1.5 seconds as concluded in Table 1. The time reduction was similar for the stacked graph visualization: from ~0.7 seconds for Flash to .2 seconds for D3.

B. Management

Time for testing and deployment: This criterion was evaluated by running benchmarks on a sample code from node-link diagram visualizations, and evaluating boilerplate code. For D3.js,

only 2-3 lines of “boilerplate” Javascript are required to initialize the force module, and 2-3 lines of HTML for necessary imports. For D3.js, Javascript code is interpreted, not compiled, so the Firefox Developer Tools were used to measure a code sample’s loading time into the browser. The length of the process was found to be independent of dataset size, completing in between 600-100ms across 5 benchmarks with code samples from bl.ocks.org using d3.force() [15 - 19].

Actionscript, however, is compiled. The code samples required 10-20 lines of boilerplate import statements before any necessary code is written. The Actionscript compiler was benchmarked using code samples on the Flare/Prefuse Demo page, namely Job Voyager, Flare Dependency Graph, and Package Map. Using benchmark settings, the compiler reported times ranging from .1 - .4 seconds for all 3 samples [20]. Loading the visualization itself also took 100-300ms. Given these results, D3 was found to have lower times for testing and deployment, as concluded in Table 1.

Interoperability with data: This criterion was evaluated by searching both frameworks’ libraries for a method that “handles” data in JSON format. This search was performed on each module’s open source Github page. Here, “handle” is defined as being able to take text input in JSON format and convert it into an object representation of the data. Upon searching the source code for D3 and Flare/Prefuse, a function was found in each library to parse JSON. For D3.js, it was the d3.json function, found in the Requests module. For Flare, an entire library of functions for JSON was found under the JSON Serialization module. Specifically, the functions JSON.parse() and JSON.decode() met this requirement. Given these findings, the capabilities of D3 and Flare/Prefuse were found to be roughly equivalent as concluded in Table 1.

C. Economic

Costs to CDC: This criterion was evaluated by searching either framework’s libraries for an open-source node-link diagram visualization module. The search was performed on each’ module’s Github page [7][8]. The module needed to be open source in order to minimize the costs from our project to the CDC. For D3.js, the d3.force() module was found to meet our criteria. The module was open sourced by Michael Bostock, the developer of the entire D3.js library, under the BSD license. For Flare/Prefuse, the GraphDistanceFilter module was found to meet our criteria. This module was open sourced by the Apache foundation, also under the BSD license. Given these findings, both frameworks met our criteria because both frameworks are open sourced, as concluded in Table 1.

D. Political/Cultural

Developer support: This criterion was evaluated by consulting several online and offline help sources for either framework and the languages they are implemented in. For D3.js, help sources were found on the Github wiki page for D3.js [7] and bl.ocks.org (a website hosting code samples and visualizations in D3.js). Wiki documentations were found for each module on the Github page, along with an in-depth API reference. Each module had multiple code samples found on bl.ocks.org, demonstrating different features/options for the module.

For Flare/Prefuse, help sources were found on the Flare Help Forum [20], Sample Applications page [21] , API documentation [13], and Tutorial page [22]. While the Flare Help page had over 900 posts, the most recent one was made in Fall 2014, almost a year out of date. The Sample Applications page also only had 3 sample Flare applications with source code included. The most helpful source was the API documentation for Flare/Prefuse, which thoroughly detailed different classes and methods in Javadoc form. Finally, the Tutorial page provided instructions on how to set up an Actionscript environment, as well as a detailed walkthrough of a simple program in Actionscript using Flare/Prefuse.

Along with these sources, questions on StackOverflow.com about D3.js and Flare/Prefuse were found, as well as questions on Javascript and Actionscript, the languages underlying the respective technologies. Our search was conducted by searching for the relevant tags, and counting questions that had the same tag. Around 16,000 questions were found for D3.js, as well as another 1,000 for NVD3, a D3.js addon library. For Flare/Prefuse, only around 30 questions were found. Even counting the ~900 questions on the Flare/Prefuse Help site, the support for D3.js was clearly higher in quantity. A similarly large disparity existed in language support. Around 962,000 questions tagged “javascript” were found on StackOverflow, compared with around 47,000 questions for Actionscript. Given these findings, we concluded that D3.js would have substantially better support for the CDC developers, as concluded in Table 1.

Conclusions and Recommendations

The current model that exists works in the context of the GHOST project as a whole. While there is a time cost associated with understanding the source code of the current model, re-writing the code base in a different language would be far more time consuming, particularly since Flare does not allow for rapid iterations through post-hoc changes through a developer console like D3.js does.

While visualization possibilities are limited only by the creativity of the creator, there is a performance discrepancy when it comes to visualizing data sets that approach the upper limit. D3.js performs significantly better on load times, especially since the benchmark tests already have the Flare plugin preloaded. This impacts load speeds by up to a second, which is significant in the context of responsiveness. As a result, learning and refining the current code base is the superior option in terms of cost, both temporally and performance-wise. Therefore, D3.js is recommended for the use of this project.

Appendix

Links

[1] http://vis.stanford.edu/files/2011-D3-InfoVis.pdf
[2] http://opensource.adobe.com/wiki/display/flexsdk/Downloads
[3] https://github.com/mbostock/d3/wiki/API-Reference
[4] http://flare.prefuse.org/api/
[5] http://d3js.org
[6] http://flare.prefuse.org
[7] https://github.com/mbostock/d3/
[8] https://github.com/prefuse/Flare
[9] http://www.whiteboxtest.com/Halstead-software-science.php
[10] https://github.com/mbostock/d3/wiki/
[11] https://github.com/mbostock/d3/wiki/API-Reference
[12] https://www.dashingd3js.com/
[13] http://flare.prefuse.org/api/
[14] http://sourceforge.net/p/prefuse/discussion/757572
[15] http://bl.ocks.org/mbostock/1062288
[16] http://bl.ocks.org/mbostock/4062045
[17] http://bl.ocks.org/mbostock/3750558
[18] http://bl.ocks.org/mbostock/7882658
[19] http://bl.ocks.org/mbostock/2706022
[20] http://flare.prefuse.org/apps/index
[21] http://sourceforge.net/p/prefuse/discussion/757572
[22] http://flare.prefuse.org/tutorial