Comparing Interactive Web-Based Visualization Rendering Techniques

Daniel E. Kee*
Tufts University
iRobot Corporation

Liz Salowitz†
Tufts University

Remco Chang‡
Tufts University

Abstract

As the need to display complex visualizations on the web has increased, a wide variety of rendering techniques have emerged. However, no single implementation has become standard, leaving developers with many options to choose from and no clear way of selecting the best one for their problem. We implemented a 2D parallel coordinates visualization, using a variety of common rendering techniques, and empirically tested them for rendering speed and interactivity on increasingly large data sets. The HTML5 canvas with a WebGL context performed best although the developer cost was highest. The performance requirements of a project may dictate a given implementation, but some options clearly do better than others, even for relatively large amounts of data.

1 Introduction

Although there are many web-based visualization tools, there is no consensus as to which technique is the best or which is most appropriate for any given purpose, especially when working with data sets considered relatively large for web visualizations. We compared and characterized each technique in order to be able to choose the appropriate one for a given project. We tested SVG, the HTML5 Canvas with an invisible SVG overlay and either a 2D or WebGL [1] rendering context, Processing.js, and KineticJS.

2 Interaction Techniques

Rendering methods were chosen for testing based on their prevalence in the web community. SVG and HTML5 Canvas are accepted standardized cross-browser methods to generate graphics and are therefore widely used. WebGL adds the benefit of hardware accelerated performance within an HTML5 Canvas, making it extremely attractive to those implementing complex visualizations. Processing.js allows developers familiar with it to easily create web apps that are compliant with web standards, and KineticJS is a library with event-based interactions so it can be used without additional interaction-detecting tools.

For both of the canvas methods tested, an invisible SVG was placed on top of the canvas in order to detect which shape mouse movements were over because the canvas cannot do this alone. Although this is a similar concept to an HTML map tag, the map tag cannot be used in conjunction with the canvas and only allows very basic shapes. Manually constructing an invisible SVG on top of the canvas has the advantage of allowing event-based interaction without manipulating the DOM, resulting in better performance. However, if the visualization marks are not static (i.e. zooming or panning is permitted), some DOM manipulation would still be required. The visualization would still perform well as long as direct interaction was not necessary during animation.

*e-mail: dkee01@tufts.edu
†e-mail: liz.salowitz@gmail.com
‡e-mail: remco@cs.tufts.edu

SVG Scalable Vector Graphics (SVG) is commonly used on the web in large part because it is based on the HTML5 DOM and uses graphics tags. This makes it easy for developers who are familiar with HTML5 and HTML5 events to create complex visualizations. Using the Data-Driven Documents (D3) library [2], data can easily be mapped to DOM elements. Because HTML5 (and thus SVG) is a declarative language, the developer can't control when rendering occurs, which may be a disadvantage in trying to improve the performance of an interactive visualization.

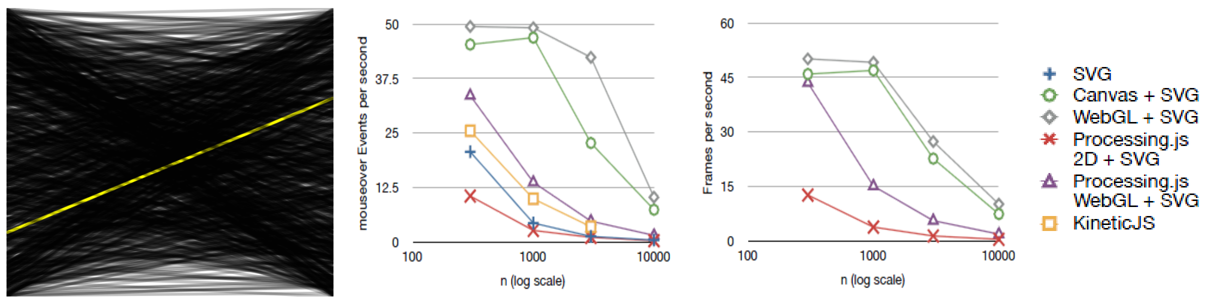
HTML5 2D Canvas with SVG The HTML5 canvas allows developers to choose a rendering context to create and manipulate content within the canvas. Currently 2D is the only context available on all canvas-supporting browsers. We placed an SVG overlay on top of the canvas in order to detect user interactions. A canvas with a 2D context works in Internet Explorer, unlike a canvas with a WebGL rendering context. The canvas is currently the most commonly used low-level method for displaying graphics in JavaScript libraries, probably because it has the most cross-browser support and high performance because browser implementers have optimized it. It also gives the developer control over when rendering occurs.

HTML5 WebGL Canvas with SVG This technique is very similar to the canvas with 2D context except that the rendering context has been changed to WebGL, which works directly with the graphics card using OpenGL ES [3]. It is available in some browsers, including Chrome and Firefox, but is disabled by default in Opera and Safari. As before, we used an SVG overlay to precisely detect user interaction. Both SceneJS [4] and Three.js [5] allow rendering with WebGL. This technique may be more difficult to implement for those unfamiliar with OpenGL.

Processing.js As a port of the Processing programming language [6] to JavaScript, Processing.js [7] can use a 2D or WebGL rendering context. This makes it easy to use for those who have experience with Processing. It allows the user to choose a different context without changing the visualization code, so we tested both. Processing.js uses an HTML5 canvas to render at a low level, so we used an SVG overlay to track user interaction. It has an explicit rendering loop which the user can control using a redraw function, though we did our tests with the default setting.

KineticJS Both at the library and developer level, KineticJS [8] uses various layers of HTML5 canvases in order to create a single image. The library sets the frame rate based on the CPU demand of the user's computer and therefore does not have an explicit rendering loop from which to measure the frame rate [9].

Other Techniques Other potential visualization techniques that we did not test include OpenGL-style picking, which would likely have the fastest performance but may be tricky to implement for those unfamiliar with OpenGL. We also did not test the performance of SceneJS, an abstraction for OpenGL-style picking.



(a) Visualization Used for Performance Testing (b) mouseover Events vs. Data Points (c) Frames per Second vs. Data Points

Figure 1: Performance testing visualizing (a) two-dimensional parallel coordinates in order to measure (b) the number of mouseover events per second and (c) the rendering frame rate for different data set sizes. The visualizations can be found at http://www.cs.tufts.edu/ dkee01/webvistest

3 Experiment Design

The same test visualization was implemented using each of the six techniques in order to isolate performance. We selected a visualization using two-dimensional parallel coordinates because it can be easily scaled for larger amounts of data and provides a simple yet clear visual test of the differences in speed shown by highlighting the selected line. The user can interact with the parallel coordinates through brushing, and feedback is given by highlighting the line being hovered over (see Figure 1(a)).

All recorded data is from testing done on a MacBook Pro with an Intel Core i5 dual core 2.4GHz processor and 4GB of RAM. It used an nVidia GeForce GT 330M graphics card with 256MB of graphics memory. The computer was running Mac OSX 10.7 and tests were conducted in Google Chrome 19.0.1084.56. Each implementation was tested for functionality in Firefox, Internet Explorer, and Opera. They all worked in each browser except for WebGL-based implementations, which did not work in Internet Explorer.

4 Testing Metrics

Although the difference in speed between many of the implementations is clearly visible to the naked eye, we created and measured more precise metrics to better compare each interaction technique.

4.1 Performance Considerations

Across each technique we measured the number of frames per second that the visualization was able to render along with the number of mouseover events per second that the interaction style was able to detect. We compared this data to the specified number of marks used in each visualization in order to see how data set size affects the performance of each technique.

4.2 Developer Considerations

The cost to the developer is frequently a key factor in choosing a solution, so we measured the number of lines of code required to implement each technique (see Table 1). Each implementation uses HTML5 and JavaScript, however techniques using WebGL require the developer to use OpenGL ES, which may be difficult if she is unfamiliar with it. And, of course, each library-based implementation requires some familiarization with that library. This information is useful to know and understand because of the direct impact it has on developer's implementation decisions.

5 Results

The visualization using an HTML5 canvas with a WebGL rendering context consistently performed best in both measurements. It detected the most mouseover events per second (see Figure 1(b)), followed closely by the 2D HTML5 canvas implementation.

The two canvas techniques performed even more similarly in the number of frames they were able to render each second (see Figure 1(c)), with the WebGL context rendering slightly more frames per second. Although Processing.js with WebGL was able to render a similar number of frames per second for small data sets, the frame rate quickly dropped with larger data sets to perform similarly to the 2D Processing.js implementation. Even with larger data sets the canvas implementations outperformed the others tested.

Lines of Code	Implementation
29	SVG
56	Canvas with 2D Context
205	Canvas with WebGL Context + 23 lines of shader code
75	Processing.js
49	KineticJS

Table 1: Lines of Code Written for Each Implementation. Note: the SVG code is included in the counts of other techniques using SVG.

6 Conclusion

For interactive visualizations requiring high performance, the best implementation of those tested would be a WebGL HTML5 canvas with SVG. However, there is a definite incentive to choose a canvas with a 2D context and SVG if performance is not a critical concern because of the difference in cost to the developer. The library-based implementations have a similar developer cost while greatly decreasing performance, and the implementation with the fewest lines of code, a simple SVG, performs very poorly compared to the other implementations we tested. Both canvas implementations outperformed the library-based implementations across all data set sizes, making them the clear winners for interactivity and rendering speed among web-based visualization techniques.

References

[1] www.khronos.org/webgl/. Retrieved on 6/26/12.
[2] d3js.org. Retrieved on 6/26/12.
[3] www.khronos.org/opengles/. Retrieved on 6/26/12.
[4] scenejs.org. Retrieved on 6/26/12.
[5] https://github.com/mrdoob/three.js/. Retrieved on 6/26/12.
[6] processing.org. Retrieved on 6/26/12.
[7] processingjs.org. Retrieved on 6/26/12.
[8] www.kineticjs.com. Retrieved on 6/26/12.
[9] www.kineticjs.com/how-it-works.php. Retrieved on 6/26/12.