Comparing Rendering Performance of Common Web Technologies for Large Graphs

Tom Horak* Ulrike Kister† Raimund Dachselt‡

Interactive Media Lab, Technische Universität Dresden, Germany

Abstract

In this work, we compared the established web-technologies SVG, Canvas, and WebGL regarding their performance for large visualizations. Specifically, we compared these technologies by analyzing the achievable frames per second (FPS) in exemplary implementations of a tree visualization with increasing number of elements. We found that SVG and Canvas almost perform on par, with performance drops starting at around 10,000 graphical elements, while WebGL performs slightly better when showing text elements and stays almost unaffected by increasing node quantities without text elements. Finally, we discuss additional strategies to improve the performance in certain situations.

Index Terms: Human-centered computing—Visualization—Visualization systems and tools;

1 Introduction & Background

Thanks to the continuous upward trend of cloud-based applications, web-based visualization interfaces are also getting increasingly popular. Besides specialized stand-alone visualizations, e.g., in journalism or research, also professional web-based business intelligence platforms, such as SAP Analytics Cloud[1], Microsoft Power BI[2], or Tableau Online[3], incorporate rich visualization views. In order to ensure a high usability, the interface must always run smoothly, which can be challenging when displaying large visualizations or many at the same time.

Although steadily improving, web browsers can easily experience performance drops, e.g., when hitting a high number of elements to render. Most of the established visualization libraries are based on SVG, which add DOM nodes for each graphical element and, thus, are heavily contributing to the overall DOM node count. As a result, displaying element-heavy visualizations like large graphs may not be possible in a performant way. Alternatively, other web technologies, such as Canvas or WebGL, can be used. These technologies promise a higher performance at the cost of an increased implementation effort (in respect to, e.g., rendering, event-handling, consistency).

In the context of information visualizations, these performance considerations are not new and had been considered before. While the focus was on using Flash some years ago [3], it shifted towards SVG and/or WebGL [1, 2, 5]. Four years ago, Andrews and Wright [1] already performed a similar comparison to the one in this paper: Using a parallel coordinates plot as reference visualization, they compared SVG, Canvas, and WebGL, finding that WebGL outperformed SVG and Canvas. Notably, they also reported that in their test SVG and Canvas never reached FPS rates above 30, which is an outdated value nowadays. More recently, WebGL is increasingly considered for rendering visualizations. Ren et al. [5] proposed a WebGL library which simplifies creating visualizations. Fang et al. [2] showcased a WebGL-based interface which is able to fluidly render visualizations with up to millions of nodes.

Figure 1: The implemented tree allowed for flexible node quantities (left, 100 nodes); each node showed an embedded bar chart (right).

However, there is not one technology that always should be used. Instead, each technology comes with different implementation efforts and challenges, thus, it is important to weigh effort against required performance. In the following, we are investigating the differences between SVG, Canvas, and WebGL regarding the resulting interface performance, expressed by frames per second (FPS), when (re-)rendering a visualization during user interactions. Additionally, we also discuss different strategies to improve performance.

2 Comparing Rendering Performances

Instead of pure rendering time (i.e., time until a view is loaded), we opted to observe the frames per second (FPS) during typical interactions to measure the performance: while a longer loading time may be acceptable, a slow-acting or even laggy interface is most certainly not. Therefore, FPS can more accurately represent the perceived effects for the user. As an example visualization, we utilized a tree visualization, similar to Value Driver Trees [4], as trees and graphs are realistic examples that both can (i) consist of a large number of nodes, and (ii) easily be scaled for testing.

Implementation. For each technology, we implemented the same tree visualization. The nodes of the tree consisted of both multiple graphical elements and multiple text elements (simulating an embedded bar chart), resulting in a total of 15 elements per tree node. The tree was layouted with a static, pre-defined algorithm; edges were shown as lines with a small box and an abstract label in the center (Fig. 1). The number of nodes, and thus the size of the tree, was set through a URL parameter. In total, each node caused around 20 graphical elements to be drawn (elements of node plus corresponding edges). We incorporated some libraries to simplify the creation of the chart: we used D3[4] for the SVG version, PixiJS[5] for the WebGL version, and no library for the Canvas implementation.

Procedure. We considered the FPS during zoom and pan interactions, which are common interactions for many visualizations, including large graph visualizations. In other words, we tested how many nodes we can display until the browser is no longer capable of fluidly translating (pan) or re-rendering (zoom) the current scene with 60 FPS. Concretely, after loading the graph, we applied a fixed 20-seconds-sequence of zoom and pan interaction to the graph. Using the MouseRecorder tool[6], the corresponding mouse events were recorded before and replayed for each trial. While replaying

*e-mail: tom.horak@tu-dresden.de
†e-mail: ulrike.kister@tu-dresden.de
‡e-mail: raimund.dachselt@tu-dresden.de

[1] sap.com/products/cloud-analytics, [2] powerbi.microsoft.com
[3] tableau.com/products/cloud-bi, [4] d3js.org, [5] pixijs.com
[6] http://www.mouserecorder.com/

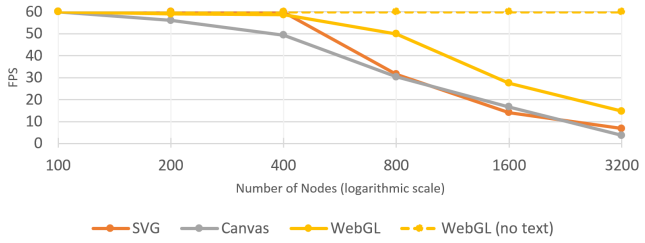

Figure 2: Performance decreased above 400 nodes, with SVG and Canvas performing similar and both worse than WebGL. For WebGL, the performance decrease only occurred when showing text labels.

this interaction sequence, we retrieved the frame rates through the built-in developer tools of Chrome or Firefox and logged the average FPS. All trials were run on the same laptop (Windows 10 64-bit, full-HD resolution, Intel i7-7500U CPU, integrated graphics, 16 GB RAM).

Findings. Our measurements show that performance losses started to occur above 400 nodes (ca. 8,000 elements) for all three technologies. SVG and Canvas perform almost equally, while the drop for WebGL is less extreme, The first aspect is interesting, as it is often assumed that Canvas is faster than SVG, since the overhead of adding DOM elements is removed. Although this could still be true for initial rendering time, there is no advantage regarding FPS—which is similar to the findings of Andrews and Wright [1] in 2014. Concerning WebGL, the performance drop only occurs when also displaying text. Without text elements, the FPS is not affected by the number of nodes; even in an extreme setup of 400,000 nodes (ca. 8 million elements) the visualization ran with 50 FPS. The reason for this difference is that in WebGL text can either be rendered and stored as an image (as in our example), which reduces performance, or must be laboriously put together with graphical primitives, which increases implementation effort.

Between Firefox and Chrome, we did not find notable differences regarding FPS, with the exception of SVG in Firefox: when starting to perform the interactions, the browser internally applied style changes, which blocked interface updates for a few seconds.

3 Strategies for Performance Improvements

Similar to improving the WebGL performance by avoiding text elements, there are several characteristics for each technique that can be exploited in order to gain additional performance benefits.

Flexible Level of Detail. As the the number of elements is the key performance driver, and in many situations not all elements are of interested to the user, one approach is to temporarily remove elements from the scene that are not required currently. For instance, when zoomed out, details of the nodes can be hidden (e.g., labels, embedded details, decorative elements), which then drastically reduces the number of graphical elements. Especially in SVG, this can easily be implemented by utilizing the CSS style property.

Asynchronous Tile Loading. Similar to web-based map applications, asynchronous tile loading can be used to split up the rendering effort to multiple threads and keep the interface responsible anytime. Traditionally, this approach involved a server that renders and sends the files to the client. However, thanks to improving web standards, this can also be done purely on client side. Specifically, the client has to start multiple threads (Webworker API[7]), where each of them initializes a headless browser rendering instance (Offscreen Canvas API[8]). These threads process than chunks of the scene and return the rendered tile. Especially the Canvas API can relatively easily be combined with this approach. With this approach, the interface always runs at 60 FPS (since all rendering is performed in separated threads) and the latency for loading tiles becomes the main performance indicator. In an early prototype, we were able to get a latency below 1 second for graphs with more than 400,000 nodes. In general, the latency itself is affected by the number of nodes but also strongly by the zoom level. Finally, it is important to note that the Offscreen Canvas feature is currently experimental, thus browser support is lacking and implementation details might change in the future[9].

Combined Approaches. To some extent, it is also possible to combine the different approaches as well as technologies. For instance, to avoid the issues with text in WebGL, a two-folded implementation could be realized, where WebGL is used for graphic elements and Canvas for text elements. Thereby, the main challenge is to keep the both scenes synchronized to make sure the text elements are at the right location.

4 Discussion and Conclusion

When implementing web-based visualizations, it is important to thing about the performance requirements of the interface. This requirement can mainly be derived from the number of elements that a visualization—or an interface in total—has to show while maintaining a FPS rate close to 60. For SVG and Canvas, a rough limit for a fluid interface is 10,000 graphical elements (~500 nodes in our example). Of course, this number can fluctuate when changing the setup, e.g., to mobile devices or ultra-high resolution displays. For most scenarios, this limit should be feasible. For instance, most business intelligence applications incorporate mainly simpler visualization, which consists of a few hundred graphical elements.

When on the edge, additional strategies for performance improvements can be considered. For instance, incorporating a flexible level of details can provide a notable performance boost with a low implementation effort for SVG. If a significantly higher performance is required, WebGL is the best solution, especially when keeping in mind its limitations of text rendering. As a promising alternative, the asynchronous tile loading is a future strategy that can be used as soon as browser support improves.

Acknowledgments

We want to acknowledge Robin Thomas for his work on this topic. Also, we thank SAP SE and specifically Thomas Beck for the valuable discussions on VDTs and web-based visualizations. This work was supported in part by DFG grant GEMS 2.0 (DA 1319/3-3).

References

[1] K. Andrews and B. Wright. Fluiddiagrams: Web-based information visualisation using javascript and webgl. In *EuroVis - Short Papers*. The Eurographics Association, 2014. doi: 10.2312/eurovisshort.20141155

[2] D. Fang, M. Keezer, J. Williams, K. Kulkarni, R. Pienta, and D. H. Chau. Carina: Interactive million-node graph visualization using web browser technologies. In *Proceedings of the 26th International Conference on World Wide Web Companion*, pp. 775–776. ACM, 2017. doi: 10.1145/3041021.3054234

[3] R. C. Hoetzlein. Graphics performance in rich internet applications. *IEEE Computer Graphics and Applications*, 32(5):98–104, sep 2012. doi: 10.1109/mcg.2012.102

[4] T. Horak, U. Kister, and R. Dachselt. Improving value driver trees to enhance business data analysis. In *Poster Program of the 2017 IEEE Conference on Information Visualization (InfoVis)*, 10 2017.

[5] D. Ren, B. Lee, and T. Hllerer. Stardust: Accessible and transparent GPU support for information visualization rendering. *Computer Graphics Forum*, 36(3):179–188, jun 2017. doi: 10.1111/cgf.13178

[7] html.spec.whatwg.org/multipage/workers.html#workers

[8] html.spec.whatwg.org/multipage/canvas.html#the-offscreencanvas-interface

[9] caniuse.com/#feat=offscreencanvas