Scalable Scalable Vector Graphics: Automatic Translation of Interactive SVGs to a Multithread VDOM for Fast Rendering

Michail Schwab, David Saffo, Nicholas Bond, Shash Sinha, Cody Dunne, Jeff Huang, James Tompkin, and Michelle A. Borkin

Abstract—The dominant markup language for Web visualizations—Scalable Vector Graphics (SVG)—is comparatively easy to learn, and is open, accessible, customizable via CSS, and searchable via the DOM, with easy interaction handling and debugging. Because these attributes allow visualization creators to focus on design on implementation details, tools built on top of SVG, such as D3.js, are essential to the visualization community. However, slow SVG rendering can limit designs by effectively capping the number of on-screen data points, and this can force visualization creators to switch to Canvas or WebGL. These are less flexible (e.g., no search or styling via CSS), and harder to learn. We introduce Scalable Scalable Vector Graphics (SSVG) to reduce these limitations and allow complex and smooth visualizations to be created with SVG. SSVG automatically translates interactive SVG visualizations into a dynamic virtual DOM (VDOM) to bypass the browser's slow 'to specification' rendering by intercepting JavaScript function calls. De-coupling the SVG visualization specification from SVG rendering, and obtaining a dynamic VDOM, creates flexibility and opportunity for visualization system research. SSVG uses this flexibility to free up the main thread for more interactivity and renders the visualization with Canvas or WebGL on a web worker. Together, these concepts create a drop-in JavaScript library which can improve rendering performance by 3–9× *with only one line of code added*. To demonstrate applicability, we describe the use of SSVG on multiple example visualizations including published visualization research. A free copy of this article, collected data, and source code are available as open science at osf.io/ge8wp.

Index Terms—Visualization systems, SVG, performance, virtual DOM, rendering, D3.js

1 Introduction

Scalable Vector Graphics (SVG) is the dominant markup language for visualizations on the web. The format is open and comparatively easy to learn, and the SVG document object model (DOM) is accessible, searchable, and easy to customize with CSS. As data is represented via individual elements, it is easy to add interaction to these elements, and to select and debug them when necessary. The data visualization community has built infrastructure around SVG with tools like D3.js [7], which enable fast development because data are bound to visual mark objects like `<circle>` and `<rect>` in the SVG DOM. The success of this open format is in part due to Web browser tools that help developers inspect and debug element appearance and behavior. With the help of these tools, it is easy to learn from and adapt existing SVG markup, leading to many visualizations being created using SVG. In these respects, SVG is essential to the visualization community.

However, this flexibility comes at a price: as dataset size increases and the corresponding number of DOM nodes grows, SVG-based visualizations become slow. Even a minor DOM update triggers the browser rendering pipeline which includes full re-calculation of styles, layout, updating the DOM layer tree, painting, and compositing. This overhead can lead to slow performance for animation and interaction even with only a few hundred nodes, which acts as a significant limitation on the space of possible visualization designs and datasets.

For rendering and interaction performance, the community turns to lower-level pixel-based interfaces such as Canvas and WebGL. For instance, in one 2009 Firefox benchmark [37], Canvas is $\approx 4\times$ faster than SVG for rendering 600 circles. These gains are wrought by bypassing the DOM and requiring the user to provide their own higher-level object (and data binding) abstraction, but this approach is harder to learn, causes more difficult implementation and debugging, loses the useful searchable and CSS customizable properties of SVG, and makes element-wise interactions more difficult to implement. While object-level abstraction libraries have been proposed to simplify the development process for Canvas and WebGL [31], none have yet gained a critical mass within the visualization community. Similarly, browsers have recently added support for the new OffscreenCanvas feature [40], which can

- *Michail Schwab, David Saffo, Nicholas Bond, Cody Dunne, and Michelle A. Borkin are with the Northeastern University, Boston, MA 02115 USA. E-mail: {schwab.m, saffo.d, bond.n, c.dunne, m.borkin}@northeastern.edu.*
- *Shash Sinha, Jeff Huang, and James Tompkin are with the Brown University, Providence, RI 02912 USA. E-mail: {ssinha11, jeff_huang, james_tompkin}@brown.edu.*

Manuscript received 28 Sept. 2020; revised 14 Jan. 2021; accepted 28 Jan. 2021. Date of publication 15 Feb. 2021; date of current version 1 Aug. 2022. (Corresponding author: Michail Schwab.) Recommended for acceptance by M. Hadwiger. Digital Object Identifier no. 10.1109/TVCG.2021.3059294

1077-2626 © 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

Authorized licensed use limited to: Brown University. Downloaded on September 04,2022 at 03:14:38 UTC from IEEE Xplore. Restrictions apply.

Fig. 1. Viegas & Wattenberg's popular wind map visualization is written for the Web using Canvas for its rendering speed (http://hint.fm/wind/). Implementation via the more-familiar SVG and D3.js is comparatively simple and enables element-level events for interaction, styling with CSS, and element inspection, but results in slow rendering at 7 frames per second (FPS). Using SSVG, via one line of additional code, we reach 36 FPS. This is faster than the original Canvas implementation. Live at ssvg.io/examples/windmap.

parallelize drawing work and increase performance. But so far, few visualizations take advantage of these options for higher performance because they are challenging to use. Community investment in SVG continues as it is easier and more flexible for visualization prototyping and development; yet the performance problems remain.

The research question we address is how the SVG specification of visualizations — which is important to the community — may be de-coupled from the browser-based SVG rendering — which does not sufficiently scale. To answer this question, we explored multiple new visualization system ideas by analyzing the state of the art and the background relevant to data visualization. We discovered several shortcomings and identified research opportunities for data visualization system research. *First*, we studied whether and how JavaScript function calls meant to populate and update the DOM can be intercepted and redirected to instead reliably and efficiently target a virtual DOM (VDOM). Specifically, we focused on a common scenario in data visualization: updating attributes of many elements at once. *Second*, we explored communication strategies to effectively maintain a VDOM on a worker thread including SharedArrayBuffers. Again, we optimized for common data visualization usage patterns of SVGs. *Third*, we introduced the usage of the new OffscreenCanvas element to offload the rendering work from the main thread to the worker thread. This effectively provides the first accessible multi-thread solution for web-based information visualization. *Fourth*, we enabled element-level interactivity common in SVGs even when rendering Canvas visualizations. We do this by performing hitbox testing on the VDOM and re-triggering JavaScript events on the hidden DOM events so that JavaScript event listeners can capture them.

We developed, implemented and bundled these concepts into Scalable Scalable Vector Graphics (SSVG): a drop-in JavaScript library for D3.js which can 'scale' SVG performance to that of a native Canvas reimplementation. Depending on code complexity, SSVG works with as little as one line of code. SSVG is the first system able to continuously render a dynamic SVG as Canvas or WebGL, which dramatically improves rendering performance and reduces DOM overhead. In addition, SSVG is the first web-based visualization system to take advantage of parallel rendering. This approach frees the main thread from rendering and asks it only to process simulation computations, interaction handling, and VDOM updates. For example, adding SSVG helps the computationally-intensive calculation of a force-directed graph layout converge much faster while maintaining an interactive visualization (see Figs. 12 and 11).

We evaluated the robustness of SSVG by applying it to 25 visualizations taken from bl.ocks.org. We find that SSVG works on 24 of them without any code changes, and that changing 2 lines of code allows the one remaining visualization to work with SSVG. Further, we evaluated SSVG's efficiency on four rendering-heavy visualizations. SSVG improves rendering performance by $3-9\times$, and shows a speedup in network layout visualization stabilization from 16 seconds to 5 seconds. We show that SSVG increases its performance twice as much as SVG given an increase in CPU power. We also demonstrate the increased ease of development for complex data visualizations with a remake of a famous wind map visualization (Fig. 1) and a node-link visualization of the IEEE Information Visualization conference co-author network (Fig. 11). Finally, we show SSVG's applicability in visualization research in a real-world research project that uses SSVG published at CHI 2020 by Klamka, Horak and Dachselt, researchers unaffiliated with the authors [23].

We release SSVG as open source software for interactive visualization developers to use at ssvg.io and osf.io/ge8wp. Beyond the SSVG system, we contribute:

- The concept of automated translation from interactive DOM to dynamic VDOM, de-coupling SVG specification and rendering,

Authorized licensed use limited to: Brown University. Downloaded on September 04,2022 at 03:14:38 UTC from IEEE Xplore. Restrictions apply.

- A communication technique to efficiently maintain a dynamic VDOM on a worker thread,
- An optimized rendering approach that batches the drawing of many elements and that uses OffscreenCanvas without additional burdens for visualization creators,
- Recommendations for visualization system creators based on our lessons learned, including research opportunities in the areas of accessibility, expressivity and responsiveness which take advantage of the availability of a dynamic VDOM.

2 A Motivating Example

Jasmine is a student who has taken a data visualization course and learned basic Web development and D3.js. She sees Viegas & Wattenberg's wind map [39] and feels inspired to better understand the chosen encodings by interactively editing them. She right-clicks the map and looks for "Inspect" to change the map's color via CSS, but is surprised to find instead "Save image as...": the visualization is rendered via Canvas, which only allows exporting as an image and has no DOM to inspect. Jasmine investigates to find that the Canvas visualization is over 1,000 lines of pure JavaScript.

Undeterred, Jasmine re-makes the visualization as a learning exercise. Using D3.js, she binds the wind data to line elements in her SVG and, with 200 lines of JavaScript code, creates the visualization seen in Fig. 1. With moderate effort, she has implemented it in a format open to inspection, CSS manipulation, and modification in editors such as Inkscape or Illustrator. However, she quickly realizes why Canvas was used, as Jasmine's SVG windmap has three problems: First, the rendering happens at such a low frame rate (7 FPS) that she sees individual images rather than continuous motion. Second, the browser becomes slow to respond to interaction input like click or scroll. Third, the wind particle paths contain visible corners, which is an artifact of not being able to compute the wind paths smoothly without causing additional render time.

When Jasmine hears about SSVG, she visits ssvg.io and simply copies a one-line script tag into her HTML file. With no other changes, the visualization rendering performance increases to 36 FPS and browser interaction performance increases sevenfold. The browser's main thread is also now free enough to quadruple particle computations for smooth path traces. If she now wishes to change the appearance via CSS, she disables SSVG momentarily and tweaks the CSS in the browser developer tools. Once she decides on a new style, she re-enables SSVG for higher performance.

3 Background

3.1 SVGs in Information Visualization Research

Online information visualization relies heavily on SVG. We compiled a list of the software technology used in Web visualizations from the 2018 proceedings of the IEEE Information Visualization conference (InfoVis), available in the supplemental material, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TVCG.2021.3059294, by reading paper descriptions, reviewing online demos, and contacting authors. We started with all 47 InfoVis papers, excluded work without interactive web-based visualizations, and learned the technology of 27 papers. Of 27 examples with known technology, 23 used SVG. Of the remaining four publications, two used Canvas and two used WebGL. For example, Wang *et al.* [38] "used SVG for visualizing the animations, as it is easier to operate on the SVG elements and the efficiency of SVG is acceptable for our project". On the other hand, Jo *et al.* [22] wrote their application with "the HTML5 Canvas API to support visualizing tens of thousands of nodes and edges." This trade-off is understood within the community: researchers generally prefer to use SVG unless they are forced to switch technologies for performance reasons. We aim to enable researchers to investigate complex data visualizations with their preferred technology.

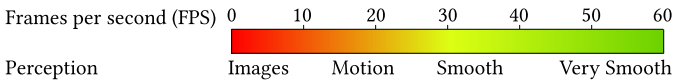

Fig. 2. Perception of frame rates between 0 and 60 FPS. At least 24 FPS are required to "tolerably" perceive motion [3] as opposed to individual images, and 30 FPS and above are perceived as smooth.

3.2 Metrics

To help describe browser SVG performance and our alternative approach, we introduce three performance metrics: rendering performance, interaction delay, and compute time.

Rendering performance describes the speed at which a visualization is re-rendered. This speed is measured in frames per second (FPS), where FPS of ≈30 are perceived as continuous motion (Fig. 2). FPS can affect task performance, e.g., below 30 FPS, performance decreases in first-person games [11]. Google's Web Fundamentals performance model states: "Users are exceptionally good at tracking motion, and they dislike it when animations aren't smooth. They perceive animations as smooth so long as 60 new frames are rendered every second" [16]. The question we seek to answer is how rendering performance can be increased as much as possible while retaining SVG's flexibility.

Interaction delay is the response time of the browser and website to user input, such as clicking. High interaction delay (feeling "laggy") takes control away from the user and stops a visualization from feeling interactive. According to Google, websites should respond to user input, such as a click, within 100ms [16], and numerous studies recommend 40–400ms depending on task [18], [19], [27], [32]. Bostock, Ogievetsky, and Heer write that "a sufficient frame rate is necessary for fluent interaction and animation", and that "results also indicate that browser vendors still have some distance to cover in improving SVG rendering performance" [7]. This is supported by literature on human psychology: for brushing and linking, for example, visual feedback is recommended to appear within 50–100 ms to support human perception [4], [10], [26].

Compute time is the time required to complete calculations necessary for a visualization. Force-based network visualizations, for example, require many iterations to compute a stable layout for user interpretation. A long compute

Authorized licensed use limited to: Brown University. Downloaded on September 04,2022 at 03:14:38 UTC from IEEE Xplore. Restrictions apply.

Fig. 3. Chrome's rendering pipeline performs these steps on the main thread at its tick rate after any JavaScript code updates the DOM: Style, Layout, Paint and Composite. In some cases, these steps take just as much time as the visualization's JavaScript code itself even though that work is often unnecessary for visualization, such as re-applying CSS rules when styles have not changed or positioning and layering HTML elements even though SVG does not support these features. SVGs are much simpler to render than HTML, yet browser rendering does not take advantage of their simplicity. Figure adopted from [17].

time until the visualization is 'ready' can turn away users, as shown by Liu and Heer [26].

3.3 The Browser and SVG

In this subsection, we analyze the browser setup commonly used on web-based data visualizations. We identify usability issues that we recommend visualization system researchers to tackle.

3.3.1 Processes and Threading

Consider Google Chrome as an example of a modern browser. Chrome runs a top-level 'browser process' with its own dedicated threads to handle user input and draw UI elements. Each Website instance runs in its own 'render process', such as in individual tabs. Each render process has a main thread, responsible for everything shown and interacted with on the page, from rendering the page contents itself to computing which DOM object the pointer clicked on.

Apart from low level operations on the graphics card, SVG visualizations are rendered on the render process main thread. To respond to input requests such as clicks, the main thread must finish rendering a frame before it can react to user input. In other words, visualization rendering performance is typically the same as interaction performance; 4 FPS rendering would delay user input by 250 milliseconds. Likewise, any computation required for a visualization's simulation would happen every 250 milliseconds.

Challenge 1. *Rendering, interaction, and simulation computation are performed on the main thread, which does not take advantage of multi-core CPUs. This slows rendering, hampers interactivity, and increases compute times — even if other CPUs are idle. We explore strategies to decouple these processes and parallelize work.*

3.3.2 Render Process Pipeline

The render process pipeline governs a significant portion of the rendering time of DOM-based visualizations like SVG—typically up to half. This is supported by Google's Web Fundamentals documentation: "Each of those frames has a budget of just over 16ms (1 second / 60 = 16.66ms). In reality, however, the browser has housekeeping work to do, so all of your work needs to be completed inside 10ms. When you fail to meet this budget the frame rate drops, and the content judders on screen. This is often referred to as jank, and it negatively impacts the user's experience" [17].

After JavaScript is executed to perform changes to the DOM (including any SVG DOM changes), the browser must translate this markup into pixels visible on the screen. On Google Chrome, the steps which are performed on the main thread are shown in Fig. 3:[1] Style, Layout, Paint, and Composite. The Style step applies rules specified in CSS, the Layout step lays out elements on the website, the Paint step decides which areas get filled with what color, and the Compositing step resolves layering questions and requests output. These modular steps happen one after another, each taking the output of the previous step as input.

While this ordering always ensures correct rendering, it also loses context which could make the rendering faster. For example, when only the position of one circle element is changed by JavaScript, the style rules are still re-parsed and re-applied to every SVG node in the DOM because the Style step requires the complete DOM as input. In fact, at the browser's tick rate, the whole pipeline is triggered any time anything on the DOM is accessed or edited. While browsers try to keep track of "dirty" elements and regions to avoid unnecessary work, their to-spec implementation has to be conservative and is more optimized for HTML than for SVG. Further, while in HTML, developers can specify nodes that should be assigned individual compositing layers for joint transformations of nested elements, such optimizations are not enabled for SVG. With a sufficient number of DOM nodes, the time spent in unnecessarily performed work in the render process pipeline is often greater than the time spent in the visualization's JavaScript code.

Challenge 2. *The expensive render pipeline is triggered for any SVG DOM updates, which can slow down rendering with re-applying previously completed work on unchanged nodes. We use the knowledge of what DOM changes have occurred to avoid unnecessary work and explore additional strategies that minimize work by communicating information across rendering steps.*

3.4 The OffscreenCanvas HTML Element

Newly added to the document of Web standards—the HTML Living Specification—in Section 4.12.5.3 [40] is the OffscreenCanvas element: "an HTMLCanvasElement but with no connection to the DOM. This makes it possible to use Canvas rendering contexts in [web] workers." To date, Chromium-based browsers support OffscreenCanvas, such as Google Chrome and Microsoft's new Edge browser, as well as Opera [30]. Mozilla Firefox allows enabling OffscreenCanvas in the settings, with official support coming soon.

OffscreenCanvas implies that rendering can happen on a separate thread without blocking the main thread and without triggering the render process pipeline for DOM updates. The OffscreenCanvas is linked to a Canvas DOM element, and changes on the OffscreenCanvas appear instantly in the browser window – no manual transfer of the OffscreenCanvas image is needed. This is an *opportunity* for the data visualization community with many potential applications. Simulation-intensive visualizations would not slow down a separate rendering thread, e.g., with expensive node position calculations. Likewise, drawing-intensive visualizations can render in a separate thread to free up the main thread for faster interaction performance and

1. Mozilla is currently developing and rolling out GPU-based threaded painting and compositing for Firefox, called WebRender [2]. For simplicity of exposition and due to its popularity, we use Google Chrome as our example browser.

Authorized licensed use limited to: Brown University. Downloaded on September 04,2022 at 03:14:38 UTC from IEEE Xplore. Restrictions apply.

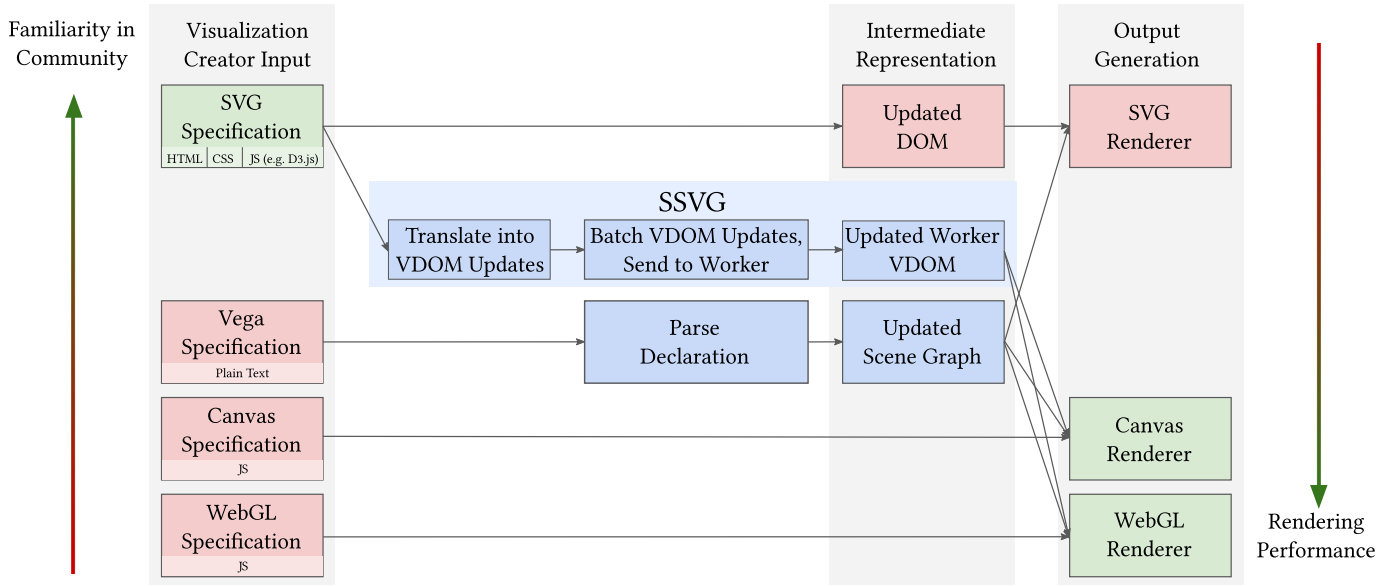

Fig. 4. A high level comparison of how visualization creators specify and define data visualizations in different languages and frameworks, and how these specifications are then rendered by the browser. Green denotes specifications familiar to the data visualization community and fast renderers, red denotes specifications less familiar to the community and slow renderers. Bridging the gap between the familiar SVG Specification and the fast Canvas and WebGL Renderers is a crucial challenge for data visualization system research to allow the community to use their preferred language even on large data sets. Translating the SVG Specification for automatic application to a Canvas or WebGL Renderer is a difficult process that SSVG tackles with a VDOM.

simulation. Finally, removing rendering from the main thread could improve the overall interaction performance of the browser, which currently must wait for a response from the render process before responding to general UI requests which relate to the page.

Challenge 3. *To achieve parallelization through rendering on an OffscreenCanvas, the main thread needs to efficiently communicate to the render thread what to render—effectively replacing the DOM analysis functions provided by the render process pipeline. Conventionally, visualization creators wishing to use the OffscreenCanvas element would need to implement their custom logic for rendering the specific visualization, for communicating updates from the main thread to the renderer thread, and to support interactivity. Interactivity is particularly challenging because knowing what visualization elements were interacted with requires position information that typically lives on the rendering thread. Further, if a lot of data must be updated, such as in a large node-link visualization, communicating the new node positions is costly, and the overhead needs to be managed. To our knowledge, OffscreenCanvas has never been used by the information visualization research community to improve visualization render performance. We find a way for visualization creators to easily take advantage of the new OffscreenCanvas element by streamlining communication between threads and by minimizing changes to visualization development for increased adoption.*

3.5 Related Work

3.5.1 Canvas and WebGL Libraries

Significant progress has been made in simplifying the use of Canvas and WebGL for information visualization. Vega [34], shown in Fig. 4, uses a custom declaration to specify the visualization and create a scene graph, which can be rendered as SVG or WebGL. Three.js [9] is a general-purpose WebGL library which abstracts some low-level implementation details of WebGL programming. PixiJS [14] is a 2D WebGL rendering library which simplifies the complexities of WebGL scenes, cameras, and GPU sprite loading. Proton [21] is a WebGL library which comes with many predefined behaviors and physics simulations for particle rendering. For network visualizations, vis.js [1] and Cytoscape [12], [36] provide fast-rendering and configurable visualizations. These libraries significantly simplify parts of the challenges of developing visualizations with WebGL. However, they have a higher barrier of entry than SVG for designing custom visualization since they require either Canvas/WebGL knowledge or knowledge of these less familiar libraries, and there is little compatibility with existing visualizations and workflows compared to the open standard of SVGs.

Other libraries propose paradigms closer to the DOM. Stardust [31] is a library which provides a data-binding abstraction for WebGL visualizations and reduces the required effort to create 2D data visualization in WebGL. Two.js [8] has a similar objective, and replicates some DOM advantages such as CSS-like styling. Paper.js [24] defines a programming interface to more easily define Canvas visualizations that are also statically exportable as SVG.

While promising, these tools are not yet widely used in the information visualization community. Two possible reasons are that Canvas and WebGL are less flexible to style (no CSS) and are harder to debug in the browser. Another reason is that each framework proposes its own application programming interface (API), and a consensus on which to standardize has yet to form within the community. We believe that switching to one of these APIs with their own language is counterproductive to the sharing and extending tools necessary for a vital visualization ecosystem. In contrast, SVG is already an open standard with widespread support and widely-used infrastructure such as D3.js. None of these tools aim to improve existing visualizations and

Authorized licensed use limited to: Brown University. Downloaded on September 04,2022 at 03:14:38 UTC from IEEE Xplore. Restrictions apply.

support the established and preferred infrastructure. While changes in visualization creators' workflows are sometimes desirable, e.g., for more expressivity, we see a switch from the open standard of SVGs to a custom API due to performance reasons as detrimental to the ecosystem. We advocate for extending the rendering capabilities of SVGs to strengthen this open and expressive format.

3.5.2 Canvas and WebGL with D3.js

D3.js [7] can render to Canvas by specifying a custom namespace, appending so-called custom elements, and then defining a custom Canvas draw function for each element type such as circles and rectangles. This provides visualization creators some familiarity within the Canvas environment and helps manage data binding to virtual elements. However, implementing the draw functions requires time and familiarity with Canvas, and the use of D3.js in this way does not provide the advantages of styling via CSS.

Some projects convert SVG visualizations into Canvas renderings, e.g., by extracting the rendered SVG as pixels and pasting the bitmap or rendering a textured quad into a Canvas. This does not increase performance as more work must be completed. Other projects, such as canvg [25], render SVG elements individually in a Canvas. Canvg supports many SVG features, including definition-based complex gradients. However, canvg is optimized for static SVGs and one-time rendering.

3.5.3 Virtual DOM

The concept of VDOMs is widely known and well adopted to improve DOM update performance, particularly in frontend JavaScript frameworks such as React [13], Vue.js [41], Angular [15], and many others. These frameworks use optimized VDOM implementations to efficiently update the DOM but, to our knowledge, no frameworks exist to intercept DOM changes to populate a VDOM. Furthermore, efficiently maintaining a VDOM on a worker thread requires effective communication that is hard to achieve because the worker thread has no access to the DOM. One work-in-progress project, WorkerDOM [5], implements synchronization of the DOM across web workers to support a small number of compute-heavy operations; however, it does not consider rendering nor our scenario of coping with updates on many thousands of visualization elements per frame.

3.5.4 Summary

To our knowledge, prior work has yet to enable rendering of interactive or animated SVG visualizations as Canvas, smooth or not. Likewise, existing solutions do not maintain the valuable properties of SVG like CSS styling, and do not provide consistency with the SVG API. Existing VDOM implementations require custom implementations and are not useful for multi-threaded visualization, and no current visualization systems support rendering on a web worker. This is a large set of challenges and opportunities for data visualization system research, as data visualization can benefit from progress in all these areas.

We build upon existing technology and infrastructure which is already in use by the information visualization community, and avoid custom APIs while still increasing SVG performance. To take advantage of multi-threading, we create a VDOM implementation optimized for communication of SVG properties across threads simply based on JavaScript function calls to the DOM.

In Fig. 4, we display a high-level overview comparison of different visualization creator inputs and the corresponding rendering outputs across different visualization systems. The SVG specification as input allows visualization creators to design with HTML, CSS, and JavaScript, such as D3.js, but typically requires going through the browsers' slow DOM pipeline and SVG renderer. Other systems, such as Vega, Canvas and WebGL require less familiar and less open languages to specify the visualization, but allow usage of the faster rendering performance. We propose for visualization systems to bridge this gap between desired visualization creator input and desired rendering engine. SSVG achieves this by parsing SVGs and JavaScript DOM calls, translating them into well-defined VDOM updates, sending batched VDOM updates to a worker, and rendering the VDOM as Canvas or WebGL on the worker.

4 Goals

Drawing from our insights on the browser's rendering pipeline and lessons learned from related work, we arrive at the following goals for Scalable Scalable Vector Graphics (SSVG):

G1: SVG Format. As shown in Section 3.1, the information visualization community relies on SVGs and is familiar with D3.js. We believe that SVG is a natural fit for data binding because of the availability of elements (to which data can be mapped). We also believe that since SVG allows inspection and is customized more easily in browsers' developer tools as well as with CSS, reviewing existing SVG code is more conducive to learning than Canvas implementations. We want SSVG to be SVG-based.

G2: Interactivity. Interactivity plays a big role in data visualizations. Enabling listeners on elements rather than absolute positions eases interactivity implementation, as Canvas-based visualizations have to manually infer which data point was hovered or clicked. In SSVG, we wish to enable element-based listeners as they exist on SVG.

G3: Rendering Performance. To address SVG's main shortcoming, we wish to make the technology scalable to more elements so that more data can be displayed. For a smooth user experience, the rendering performance should be at least 20–30 FPS, see Section 3.2.

G4: Load on Mzain Thread. To aid scalability, slow rendering should impact the user experience as little as possible. We wish to perform rendering on a separate thread such that the main thread can prepare the next frame concurrently. Little free time on the main thread can also lead to slow visualization layouts, such as network visualizations. We aim to keep the load on the main thread low to maximize browser interaction performance, and for quick layout computations.

5 System

In this section we describe Scalable Scalable Vector Graphic (SSVG)'s core concepts, its architecture, implementation, browser support, features, limitations, future work, and

Authorized licensed use limited to: Brown University. Downloaded on September 04,2022 at 03:14:38 UTC from IEEE Xplore. Restrictions apply.

how these address our design goals. We also include technical information to help the reader understand the system's inner workings, but this is not required for understanding later sections of the paper.

5.1 Overview and Alternative Strategies

At a high level, SSVG circumvents browsers' render process, as we have identified this pipeline to be too slow for high interactivity and rendering performance (G2 & G3). Any system designed to circumvent the DOM for custom rendering needs a VDOM as an internal representation of the visualization so that it can be rendered. Flexibility exists in the approach of obtaining a VDOM, in the format of the VDOM, and in the rendering technique.

SSVG obtains a VDOM by intercepting JavaScript calls which would normally affect the SVG DOM. SSVG's approach causes *no* interaction with the Webpage's own DOM, and avoids the costs associated with browsers' render pipeline. SSVG uses the DOM function calls to maintain a VDOM and records associated styles from relevant CSS. This approach has the benefit of not introducing a new API and not requiring workflow changes for visualization creators, but comes with some challenges to interpret the JavaScript calls correctly. The much more common approach of obtaining a VDOM is via custom APIs, or via templates such as in React [13] or Angular [15]. This approach is possible for visualization systems, but reduces compatibility with existing systems such as D3.js.

SSVG draws the VDOM on an OffscreenCanvas in its own render thread, but also allows single-thread use. This requires us to automatically communicate all required VDOM information across the separate main and rendering threads. We implement efficient communication via SharedArrayBuffers: a low-level technology for sharing integers across JavaScript workers. Our implementation focuses on canvas rendering and optimizes the canvas drawing calls. An alternative strategy is to use WebGL for rendering. WebGL rendering is fast when the information needed to render is largely contained within the graphics card, and little communication is needed to update the image. This is unfortunately not the case for general-purpose rendering systems, as they have no information about what is causing the data updates. Therefore, for cases like this, the WebGL performance is expected to be similar to Canvas. Our choice to use OffscreenCanvas is in line with our goal to improve interactivity and to reduce the load on the main thread (G2 & G4), but other systems could reduce our communication overhead and complexity with single-thread rendering at the cost of some interactivity and rendering performance.

Put together, SSVG is a system for turning main-thread-rendered SVG visualizations into multi-threaded visualizations. It achieves higher rendering performance, low interaction delays, and low computation times with minimal code changes — many existing visualizations are accelerated simply by importing `ssvg.js` with one line of code. For other cases, we describe limitations and workarounds in Section 5.5.

5.2 Description

To support interactive visualizations built with SVG (see G1 and G2 in Section 4), we want to enable fast rendering on visualizations implemented with D3.js with as few changes as possible. To achieve fast rendering (G3), we address challenges 1 and 2 identified in Section 3.3 by taking the rendering work off the main thread and using a tight connection between the rendering steps in Fig. 3 to avoid unnecessary work.

We address these issues with the design of SSVG by maintaining an up-to-date virtual DOM (VDOM), only reapplying CSS for new or changed elements, and using the VDOM to render the visualization in a separate worker. In Fig. 5, we explain these concepts through an example of a node-link visualization:

Init (top arrow, blue). To set up the VDOM and the Canvas element, SSVG parses the SVG and extracts any potential contained elements and copies them to the VDOM. Stylesheet rules are parsed. The original SVG element is hidden and replaced with a new Canvas element of the same size at the same position. Control over the Canvas contents is then passed to the Web Worker, which is responsible for rendering.

Update (middle arrow, orange). When a visualization attempts to access the DOM, the operation is re-routed to the VDOM. This is achieved by capturing JavaScript function calls such as `appendChild()`, `setAttribute()`, and `getAttribute()`. In this way, visualizations do not modify the DOM when calling `setAttribute()`. Meanwhile, the visualization remains naive to these processes and continues to operate as usual. This is possible because `getAttribute()` also accesses the VDOM instead of the DOM. As a performance improvement, we intercept D3.js' `attr` function which operates on a selection of nodes instead of individual nodes. This helps minimize the overhead of mapping DOM to VDOM. But because this is only a performance improvement, SSVG works with other frameworks such as Vega and React. Calls to `appendChild()` determine the position of an element in the VDOM, but are also used to fake hierarchy in the original elements so that visualizations can continue to rely on the DOM structure, such as with `element.parentNode`. When accessing the DOM outside the SVG, SSVG passes on the information to the native functions so as to not disrupt any non-SVG JavaScript on a website.

Interaction (bottom arrow, purple). In SVG visualizations, event listeners are attached to DOM elements. With SSVG, no elements are appended to the DOM so event handling must work differently. To support element-based interactions, SSVG manually re-triggers events on the original SVG elements after they occur on the Canvas element. To find the correct SVG element on which to trigger an event, the matching VDOM element must be identified based on the event position. SSVG achieves this with manual hitbox testing, but for speed it performs this only on elements which received a listener function from the visualization. To allow event listeners to receive events without delay, hitbox testing needs to occur on the main thread. To support hitbox testing on the main thread, the main thread needs to have a partial copy of the VDOM that contains position information of elements. For a smooth user experience (G2), it is important to minimize discrepancies between the main thread VDOM, which users interact with, and the render VDOM, which is used to display the visualization. SSVG

Authorized licensed use limited to: Brown University. Downloaded on September 04,2022 at 03:14:38 UTC from IEEE Xplore. Restrictions apply.

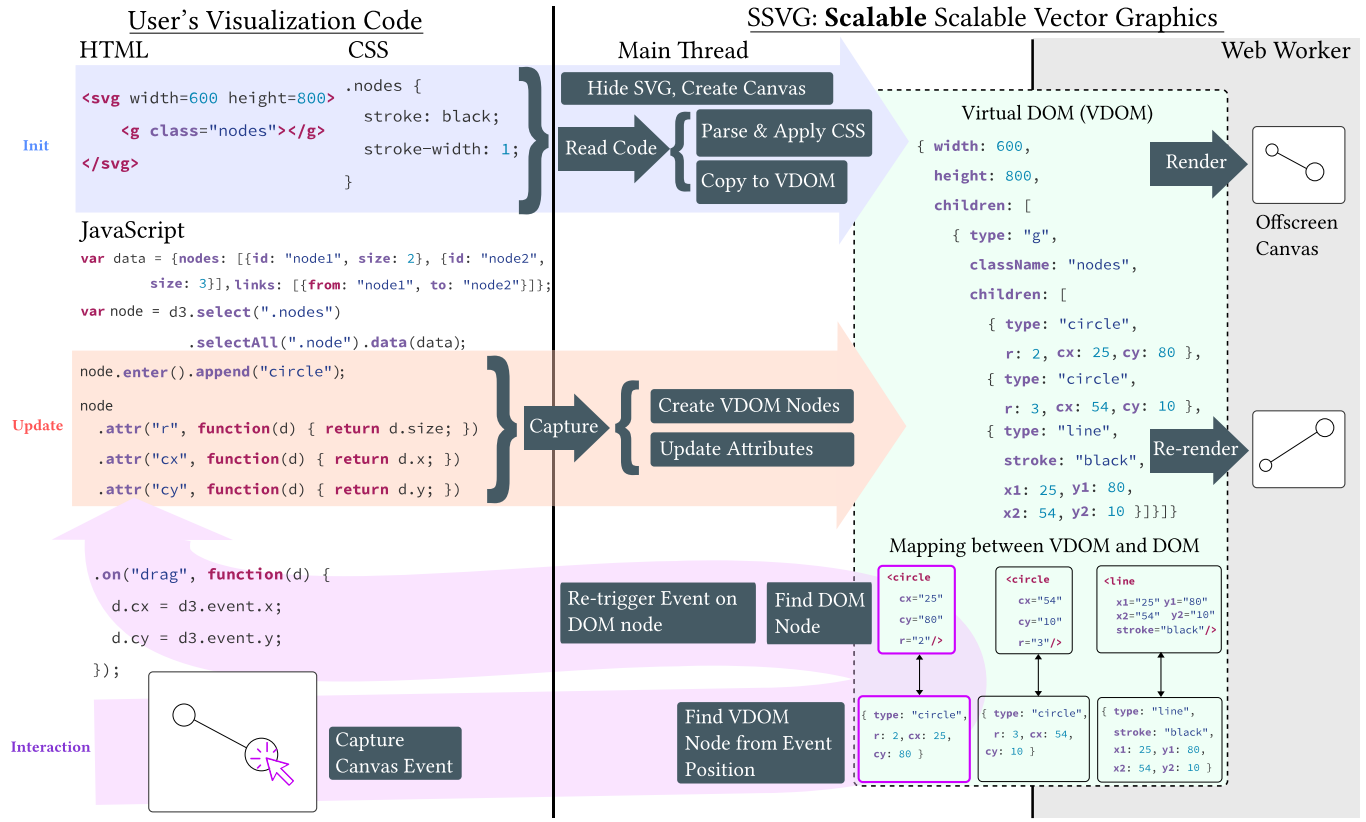

Fig. 5. Overview of SSVG's architecture and how it enables a conventional nodelink diagram written with D3.js to be rendered smoothly. From left to right, the diagram shows the user's visualization code, SSVG's main thread tasks, it's internal representation of the DOM, and the Canvas rendering in a web worker. Methods that read and write on the DOM are overridden to instead operate on the VDOM, which is used to render the Canvas. The browser rendering pipeline is largely skipped because DOM updates are avoided.

minimizes the discrepancy by only applying changes to the main thread VDOM when the render thread is done rendering and ready to update its VDOM for rendering a new frame. SSVG's fast parallel rendering also contributes to an immediate response to user interaction.

5.3 Cross-thread Communication

To allow the worker thread to render the visualization, the main thread has to communicate all VDOM data and keep the worker VDOM up to date by sending updates when they occur. To prevent race conditions, the main thread saves the local VDOM updates until the rendering thread informs the main thread that it is done rendering the last frame and is ready to receive updates for a new frame. We identify three main requirements for the communication:

Lightweight Communication. Serializing large messages for JavaScript's `postMessage` is expensive. To minimize delay, message sizes need to be small even when communicating updated data for thousands of node attributes.

Reliable Identification. Because messages are serialized, no VDOM node references can be transferred, and changed nodes have to be identified based on a specified message protocol. However identifiers, such as element selectors, can be unreliable; the communication happens asynchronously and the changes may effect the DOM and VDOM structure. For example, the commands "*move the fifth SVG element to the right, then delete the fourth SVG element, then move the fifth SVG element down*" need to be executed in the correct order so that the element that is moved to the right and the element that is moved down are not the same element.

Efficient Computation. It is important to consider the computational cost of generating the update messages on the main thread as well as decoding and applying the update on the worker thread. Importantly, the main thread has to generate a message that uniquely assigns the new values to specific elements. If element selectors are used to identify nodes, efficient ways to generate these selectors and intelligent ways of caching and updating these selectors are needed. On the worker, the updates have to be applied quickly without the need to look up nodes on an individual basis, and fast iterations over the elements are desired.

To address these three requirements, visualization systems aiming to support multi-thread VDOM updates for many nodes need to explore different batching strategies for update messages. Before switching to our latest strategy, SSVG used hierarchy-based lists of updates similar to how D3.js applies attribute changes to a selection of elements with a common parent. Within the parent, elements can be identified simply by their child index, and the worker can iterate over all children instead of looking them up individually. We paired this strategy with a selector-based method of identifying the parent element. Positive aspects of this approach are that it is similar to D3.js, lists are only as big as the number of elements within a parent, and the worker can iterate over the nodes efficiently if they have the same parent. On the other hand, the selector-based approach is

Authorized licensed use limited to: Brown University. Downloaded on September 04,2022 at 03:14:38 UTC from IEEE Xplore. Restrictions apply.

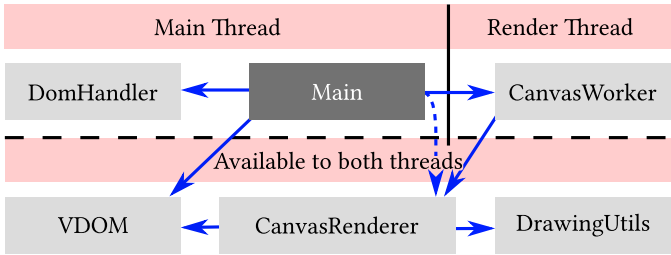

Fig. 6. SSVG's internal architecture. The main file sets up the Canvas and passes it to the CanvasRenderer through the CanvasWorker if OffscreenCanvas is supported, or directly otherwise. The main file intercepts function calls such as `setAttribute()` to DOM objects and captures the initiated attribute changes in the DomHandler. These changes are propagated to the local VDOM as well as to the renderer VDOM. The renderer uses the up-to-date VDOM and included utility functions to draw the Canvas.

difficult to implement reliably because of the asynchronous communication and simultaneous structural changes. The biggest issue with this earlier approach is that it relies on a flat hierarchy for its performance: if many nested elements are changed simultaneously, e.g., with `d3.selectAll('g').select('circle').attr()`, they have to be identified individually. This can result in slow performance.

We decided in our final implementation on a strategy that identifies all elements by a global element index within the visualization, assigned upon creation of the element. This strategy avoids referring to elements by their selectors, which makes this approach more reliable. New attribute values are simply communicated via an array, and the index within the array refers to the global element ID. This approach minimizes context that needs to be communicated within the update messages, but can lead to larger arrays. The biggest benefit of this array-based approach is that the computation on both threads is straightforward, and the worker can simply iterate over the array and assign the values to the elements with the corresponding global element IDs. Therefore, for each type of attribute changed, one single array is needed to communicate the new values.

To additionally improve communication performance, we use SharedArrayBuffers [29]. These are part of the ECMAScript 2017 standard and were added to browsers recently. SharedArrayBuffers are low-level arrays which can *only* contain integer values, to prevent any harmful content from being transmitted. We currently use SharedArrayBuffers to communicate `cx`, `cy`, `x1`, `x2`, `y1`, `y2`, `x`, and `y` attributes. These are frequently-updated position attributes, e.g., to move circles, rectangles, and lines. Since more than integer-level precision is required to avoid flickering of node positions, the message also contains information about the encoding of the values in the SharedArrayBuffers. We typically multiply position values by a factor of 100 before converting to integers. This information is then used by the worker to decode the values and to receive float values.

Combining these approaches, our update messages follow the format shown below. The portion of the data that is not sent via SharedArrayBuffers is sent as part of the "raw" data and supports arbitrary values such as strings. For the rest, pointers to the SharedArrayBuffers are transferred, along with the information of what types of attributes their values refer to, such as "cx".

```
const updateData = {
  raw: {fill: ['red','red','blue'],
    class: ['group1', 'group1', 'group2']},
  shared: {cx: bufferX, cy: bufferY}};
```

5.4 Implementation

SSVG is implemented in TypeScript, which is compiled and bundled to JavaScript. One JavaScript file defines the main thread behavior, and one file is created for the web worker. To avoid loading multiple files, the main library file includes the worker file as a Blob [28]. The internal architecture of SSVG is shown in Fig. 6. SSVG is modular, and different types of renderers can be used. We provide our own optimized Canvas renderer as well as a basic implementation of a WebGL renderer using Stardust [31]. When overlaps are not an issue, the order of drawing calls is not important. For such cases, SSVG's default rendering behavior is to group elements within the same parent by color, begin a rendering path, follow the path of all these elements, and then apply a single stroke and fill for them all. This approach dramatically speeds up rendering. As a fallback for visualizations in which the order within a parent is important, this option can be turned off and the Canvas renderer draws each element individually.

When OffscreenCanvas is available for rendering, SSVG creates a worker which uses the renderer on a separate thread. For browsers which do not yet support OffscreenCanvas, the Canvas renderer is used directly. As a consequence, SSVG does not require OffscreenCanvas to work. We show a performance comparison between single-thread usage of SSVG and SSVG with two threads in Section 7.

The DomHandler manages the mapping between DOM elements and the corresponding VDOM elements so that interactions can be propagated to the correct elements. The DomHandler lives on the main thread since the Web Worker has no access to the DOM. The CanvasRenderer only uses the VDOM to render since it has no access to any elements if it is being called from the renderer thread.

5.5 Features and Limitations

SSVG supports JavaScript features frequently used in SVG information visualization such as adding and removing elements, setting and reading attributes, setting CSS classes, and setting styles via CSS and via the style attribute. At michaschwab.github.io/ssvg-tests/, we track SSVG's rendering quality by comparing it with SVG's rendering output across a growing set of examples. This allows us to work on supporting more advanced SVG and CSS features while maintaining the established capabilities. The full list of supported and experimental features is available on the testing website. SSVG does not yet support SVG definitions such as arrow heads, advanced path settings such as stroke-dasharray, or SVG animations. D3.js functions with complex DOM operations, such as nested data selections or reordering SVG elements in the DOM, have limited support within SSVG. We aim to add implementations for these features in the future to support more use cases. Generally, visualizations can achieve

Authorized licensed use limited to: Brown University. Downloaded on September 04,2022 at 03:14:38 UTC from IEEE Xplore. Restrictions apply.

the same DOM changes through simpler operations that are already supported by SSVG, such as D3.js transitions instead of SVG animations, and single-depth data selections with nested elements instead of nested data selections. SSVG has a growing implementation of CSS and supports rule priorities. We will support more features in the future based on prioritized importance for the visualization community.

6 Using SSVG

SSVG currently works on visualizations with D3.js version 3, 4, and 5. In most settings, users simply have to include a new script tag on their website. The automatic version of SSVG detects SVG visualizations on the website and automatically enables SSVG with default settings:

```
<script src="//unpkg.com/ssvg/ssvg-auto.js">
</script>
```

If more control is desired, visualization creators can use the manual version with `new SSVG()`. This can be used to create separate instances of SSVG for multiple SVGs, in which case separate web workers would be created for each visualization. The manual version also allows passing settings:

```
<script src="//unpkg.com/ssvg/ssvg.js"></script>
<script>new SSVG({
  safeMode: false, useWorker: true,
  getFps: function(fps) { console.log(fps); }
});</script>
```

The settings (see ssvg.io/docs) include:
safeMode (default false): With safeMode enabled, SSVG renders all elements individually. This is closest to SVG-based rendering.

useWorker (default true): If supported, use a Web Worker.

getFps: A function which receives the current rendering performance. The FPS drops to 0 for visualizations with no animation to avoid unnecessary work.

We recommend minimizing the use of transformations for x and y positions, as these are difficult to optimize. In general, little work is needed to enable SSVG on a D3.js visualization.

6.1 Debugging With SSVG

While a VDOM is maintained in memory, one drawback of SSVG's Canvas rendering is that the VDOM elements are not readily available for browser inspection. This can make development more difficult, and does not allow easy access to the elements for users of the visualization. To support continued SVG debugging even on SSVG-rendered visualizations, we provide a toggle to switch back to SVG rendering for development purposes. Once SSVG is included, reloading a website with an `svg` parameter will temporarily disable SSVG. For example, if `vis.com` loads a SSVG visualization, then `vis.com?svg` disables SSVG and allows traditional browser-based inspection. If unfamiliar users attempt to inspect a SSVG visualization, a comment in the HTML explains that this is an SSVG visualization and provides the URL to enable SVG inspection. Thus, the workflow is to debug in SVG but toggle SSVG to monitor performance. Final presentation is in SSVG; should another user 'Inspect' the visualization, they will see instructions to toggle SSVG and view the SVG DOM as normal.

7 Utility

We demonstrate the utility of SSVG for visualization in practice and in research, and evaluate it on common visualizations and visualization sizes. SSVG works out of the box on 24 of 25 D3.js (v3, v4, v5) visualizations found on bl.ocks.org. In four benchmarks, we show that SSVG renders 3–9× faster than SVG. While performance varies, in one benchmark we find SSVG performs similarly to the speed of a native Canvas implementation. Further, we show that SSVG can be used for current research in two examples. First, we re-create an example publication from IEEE VIS 2018 [35] with SSVG which was originally implemented with canvas, demonstrating that work which typically happens offline or in complex Canvas or WebGL environments can be implemented and shared in the D3.js and SVG format which is better understood by the community. Second, we show how SSVG was used in a CHI 2020 visualization publication to render visualizations on wrist bands of smart watches [23]. Finally, on a IEEE VIS authorship network visualization, we demonstrate that expanding the scope of SVG enables visualization creators to focus on visual marks during development.

7.1 Bl.ocks.org

To test the general applicability of SSVG to D3-based data visualizations on the Web, we selected a set of 25 examples from bl.ocks.org to try SSVG on without any code changes. We selected the visualizations based on interactivity, animations, and amount of data to test SSVG's stability across challenging conditions.

With the only change to the visualizations being to load SSVG, SSVG fully worked on 24 out of the 25 visualizations. Twelve of the correctly-working visualizations are shown in Fig. 8. The one visualization that did not work correctly actually renders correctly, but only very slowly. An analysis revealed that the visualization unnecessarily deleted and re-created many of its elements in each frame. Because updates occur much more frequently in data visualizations than node creations, SSVG is more optimized for updating attributes, and creating an element involves the additional cost of transferring the information across threads. A two-line change in the visualization code prevented the unnecessary deletion and re-creation of elements, and cause SSVG's rendering to be smooth. With this change, all 25 interactive visualizations work with SSVG.

While we are working on continuously growing our support for SVG features, we demonstrated here that SSVG works a large subset of D3.js visualizations to be of immediate use to the visualization community. At ssvg.io/examples, we present a list of SSVG-rendered visualizations. At osf.io/ge8wp, we document the evaluation of the 25 example visualizations with testable demos.

7.2 Benchmark

To measure our success in achieving goal G3: high performance rendering, we take a D3.js SVG data visualization and compare its performance against SSVG rendering. We

Authorized licensed use limited to: Brown University. Downloaded on September 04,2022 at 03:14:38 UTC from IEEE Xplore. Restrictions apply.

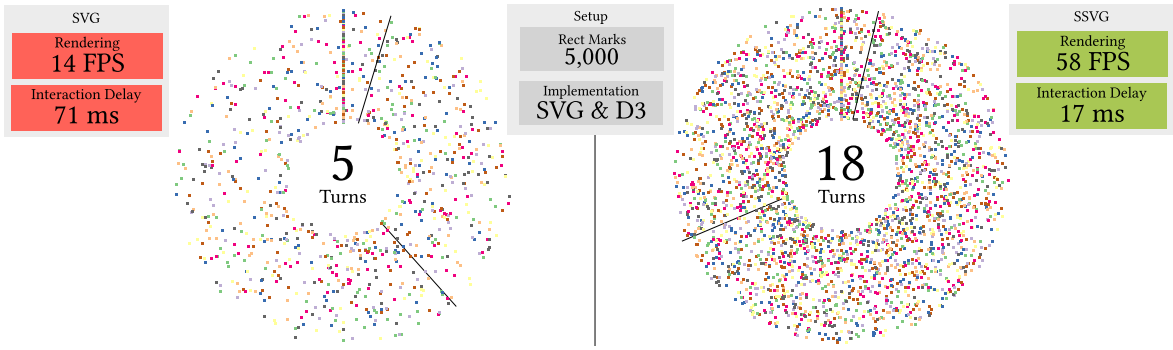

Fig. 7. A sample visualization which counts the completed turns for particles moving in a circle, rendered with SVG (left) and SSVG (right) over one minute each. While the particles have made 18 turns with SSVG, only 5 turns were completed with SVG. The number of particles is gradually increased. SSVG's rendering contains more particles for a given framerate, demonstrating its higher performance. Live on ssvg.io/examples/donut.

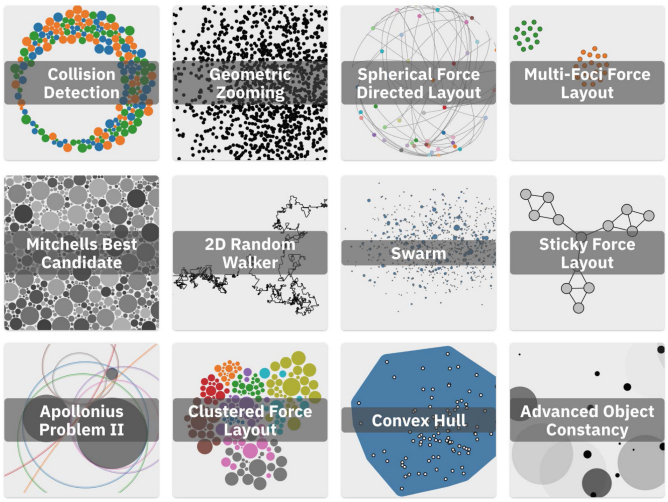

Fig. 8. A set of 12 SSVG data visualizations. The visualizations were taken as-is from bl.ocks.org and were not modified for SSVG usage beyond including a script tag. Live at ssvg.io/examples.

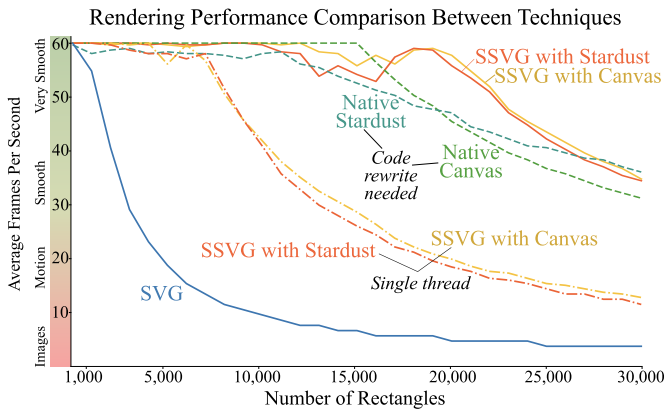

Fig. 9. Rendering performance comparison between SVG (blue), SSVG (yellow), SSVG with Stardust using WebGL (orange), custom native implementations of the same visualization for Canvas (green) and Stardust (cyan), and a single-threaded version of SSVG and SSVG with Stardust (yellow and orange, dash-dotted) that shows how much multithreading contributes to SSVG's performance. At 20,000 nodes, the SVG only renders at about 6 FPS, leading to jank and a bad user experience. Meanwhile, both SSVG implementations render at about 55 FPS, on par with and even outperforming custom native implementations of the same visualization in Canvas and WebGL. Data and implementations at osf.io/ge8wp.

create a simple visualization with many marks but little computationally-heavy work (Fig. 7). Data points move in a circle at varying radii and speeds (also visible on ssvg.io).

Visualization. For the benchmark, we use a simplified version of the circular visualization: All elements are gray and begin moving immediately. We gradually increase the number of data points with a step size of 100 nodes and average the rendering performance over a time period of 6 seconds at each step. We create a native Canvas implementation for the same visualization to compare performance.

Setup. For software, we use Google Chrome version 78. For hardware, we use a Apple MacBook Pro (16-inch, 2019) with 32 GB memory, an 2.4 GHz 8-Core Intel Core i9 processor and AMD Radeon Pro 5500M 8 GB graphics card, capped at a 60 FPS tick rate by the OS and hardware display. Performance will vary across devices due to varying hardware and software. Additional results for other hardware, including smart phones, is provided at osf.io/ge8wp and briefly described below. Also note that the frame rate results also imply corresponding interaction delay: for single-thread systems, such as SVG and Canvas, the interaction delay is equivalent to the FPS because interactions can only be picked up when the rendering process is done. For SSVG's multi-threading, interaction performance is at least as high as rendering performance.

Results. The performance results for rectangles are shown in Fig. 9. At 15,000 nodes, SSVG renders about 9× faster than SVG, and overall about on par with a custom native implementation of the visualization for Canvas. To stay above a rendering performance of 30 FPS, SVGs limit data visualization creators to about 2,500 data points, whereas SSVG supports about 35,000 data points at the same frame rate without any code changes; a 14-fold increase. Even for browsers that do not yet support the multi-thread setup, SSVG still allows up to 14,000 data points. This shows that the bigger contributor to SSVG's performance improvement over SVG is the ability to render dynamic SVGs as Canvas, side-stepping the browser's rendering pipeline by intercepting JavaScript function calls without any changes to the SVG visualization code.

CPU Scaling. To compare performance across different CPU powers, we measure SVG and SSVG performance on a computer using full CPU clock speed, as well as using only half the CPU's clock speed. Given twice the CPU performance, SVG's performance increased by 50 percent. However, SSVG increased its performance by 100 percent. As there is less overhead, SSVG can better take advantage of the

Authorized licensed use limited to: Brown University. Downloaded on September 04,2022 at 03:14:38 UTC from IEEE Xplore. Restrictions apply.

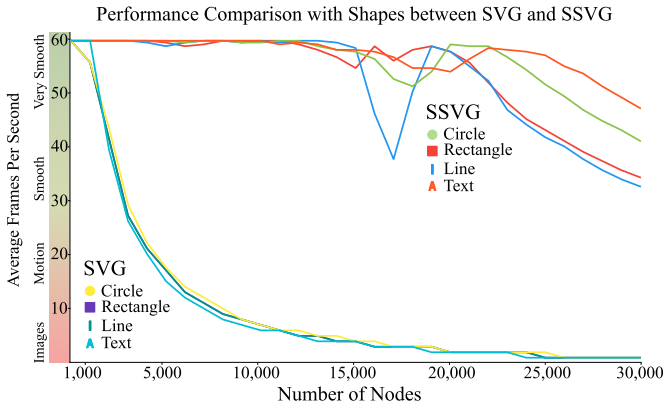

Fig. 10. Rendering performance comparison between SVG and SSVG for text, circle, rectangle, and line primitives. SSVG outperforms SVG across all tested shapes. Around 17,000 lines, SSVG's performance is lower than with more and fewer nodes. This anomaly requires further investigation.

faster CPU. On a fast computer, SSVG performs even faster than the native Canvas implementation because multi-threading is more advantageous there. This data is included at osf.io/ge8wp.

Smart Phones. Initial benchmarks on smart phones show average of a 55–60 percent rendering performance increase for 1,000–10,000 rectangles on an iPhone SE iOS 11 for Chrome 73 and Safari 11, and an average of 180 percent speedup for a Pixel 3 device.

WebGL. When evaluating our basic Stardust WebGL renderer (see Section 5.4), we found similar performance compared to the Canvas-based worker. Based on these results, our impression is that the Canvas renderer is already quite optimized for this use case and little improvement can be made by switching to WebGL.

Conclusion. These benchmarks show how that a speedup of up to $9\times$ is possible, and our results in Fig. 10 show that this speedup is consistent across different shapes. In the next sections, we demonstrate real-world applicability: We analyze the utility of SSVG for a previously-published VIS paper technique, describe how SSVG was used in a research project published at CHI by researchers unaffiliated with the authors, and discuss more examples.

7.3 Probabilistic Graph

Faster Web rendering with SSVG would allow our community to share more complex designs or data with online demos. We could then collectively learn from these online demos because they are created and shared in a familiar format. To demonstrate this aspect of SSVG's utility to the data visualization research community, we present an example case where SSVG could be used to make available online recent research which was completed offline.

In 2017, Schulz *et al.* [35] developed probabilistic graph layouts including uncertainty, drawing thousands of nodes and marking clusters. Unfortunately, the work was completed offline, making the results difficult to share. Here, we create a SVG-based visualization with similar visual encoding to demonstrate feasibility of rendering similarly structured networks with many nodes. Our implementation is a two-layer force layout where the node groups repel each other, and individual nodes are pulled toward the node group centers. Our recreated visualization contains 6,000 elements (Fig. 12). SSVG increases rendering performance from 9 FPS to 23 FPS and decreases the time to stabilize the network layout from 79 to 27 seconds. Our demonstration shows that rendering this probabilistic graph using a simple D3.js and SSVG implementation is feasible even with large datasets. While other factors, such as availability of clustering algorithms, may have played a role in their decision to implement their research offline, we want rendering performance to not dictate whether work can be shared online.

7.4 Network Visualization

In this example, we demonstrate SSVG's utility with the ability to focus on design even in complex network visualizations. This is a domain which typically requires heavy attention on implementation due to the many visualized data points. Using data from Isenberg *et al.* [20], we explore the co-authorships of researchers depending on location. We use multiple forces to determine authors' positions on the visualization, for which D3.js provides excellent support: A collision force prevents node overlaps and a geoposition force determines an author's position on the map. Co-authors are linked and pulled toward each other, whereas researchers without collaboration are repelled. A combination of these forces moves authors close to their affiliations, but allows collaborations to have a significant impact on their position (Fig. 11). The visualization is available online at ssvg.io/examples/infovis and has been presented as a IEEE VIS 2019 poster [33].

We created this visualization by starting with an example network graph on bl.ocks.org [6], added a map background, and defined custom forces to move nodes to the desired locations. We did not have to learn the API of a network visualization library nor implement custom drawing functions. Instead, we were able to benefit from forces and colors provided by D3.js, and SSVG is able to render the visualization smoothly (44 FPS instead of 15 FPS) and quickly produces a stable layout (5 seconds instead of 16 seconds). In this familiar workflow, we can add other forces provided by D3.js to experiment with different layouts, or style nodes differently with CSS. We believe maintaining the D3.js design process is helpful to data visualization research.

7.5 Windmap

We developed a remake of Fernanda Viegas and Martin Wattenberg's popular wind map [39], as illustrated in Section 2 and Fig. 1, to demonstrate that particle visualizations are feasible with SVG when SSVG is used for rendering. Our new implementation and performance results are shown in Fig. 1. Wind is visualized as 3,000 traces at a given time using four line segments with decreasing opacity, or 12,000 individual SVG elements total. Opacities and colors are simply set with CSS, and the lines are drawn with D3.js. The SVG visualization renders too slowly and causes low browser interactivity with this much data ($\approx$7 FPS). Including the SSVG library immediately results in an improved performance of 36 FPS. This new windmap visualization can now be customized easily, such as styling path colors with CSS, and data binding makes the code's structure clear. In Fig. 13 we compare the SVG and SSVG rendering pipeline,

Authorized licensed use limited to: Brown University. Downloaded on September 04,2022 at 03:14:38 UTC from IEEE Xplore. Restrictions apply.

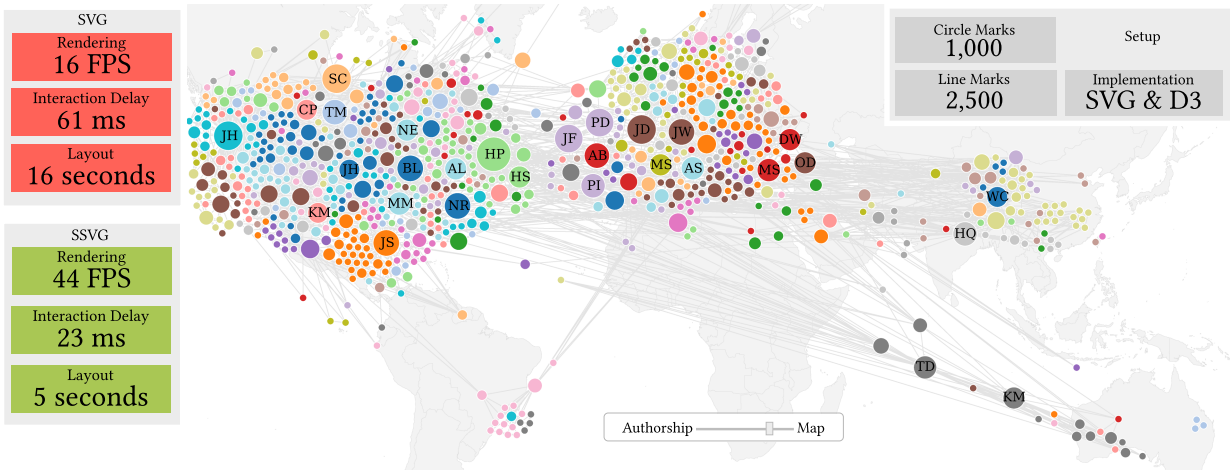

Fig. 11. IEEE VIS Information Visualization paper authors of the last 10 years, visualized by a combination of geolocation and co-authorship of papers [33]. Author nodes are sized by number of papers and colored by affiliation. Authors with at least 8 papers are shown with initials, such as Hanspeter Pfister (HP) or Sheelagh Carpendale (SC). Visualization researchers on the US East coast often collaborate with Europe, and researchers from Australia tend to collaborate with the US and Europe. Several authors are placed far from their institution. For example, Tim Dwyer (TD) and Huamin Qu (HQ) are placed far from Australia and Hong Kong, respectively, due to co-authorships with collaborators in the US and Europe. Without SSVG, this visualization's layout takes 16 seconds to compute, whereas with SSVG, the layout only takes 5 seconds and the visualization renders smoothly.

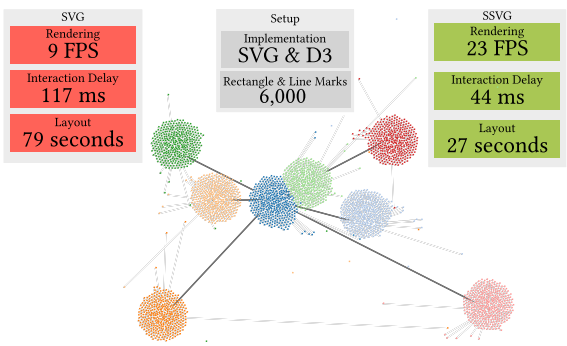

Fig. 12. A graph layout with 6,000 elements rendered with SSVG. This could be used as a starting point to make Schulz' work [35] open and accessible on the Web. SVG renders at 9 FPS, and layout takes 79 seconds to stabilize. SSVG renders at 23 FPS and arrives at a stable layout in 27 seconds. Available at ssvg.io/examples/probabilistic.

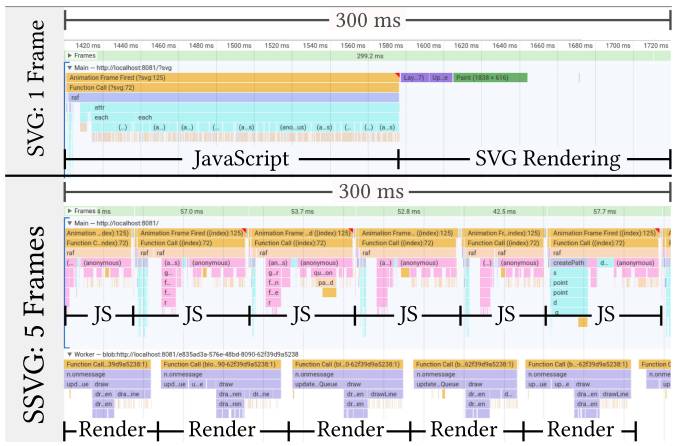

Fig. 13. Chrome's performance analysis on the windmap visualization for SVG (top) and SSVG (bottom). In 300ms, SVG renders one frame, whereas SSVG renders five frames. Parallelization is a strong performance advantage, as well as saving costly DOM access and avoiding the expensive browser rendering pipeline.

demonstrating that SSVG performs faster both by executing the rendering pipeline more quickly, as well as preparing the next frame while rendering via the multi-thread setup.

7.6 e-Ink Data Visualization

SSVG is open source and has already been used in research projects published at CHI. Klamka et. al used SSVG to efficiently render data visualizations on e-Ink devices [23], which can render images but not SVG. The researchers implemented their data visualizations using SVG and D3.js, their preferred technologies, and use SSVG to render images on the devices. We canvassed their opinion via personal correspondence: "I don't know any libraries for Canvas that are as powerful as D3.js, and using D3.js allows us to re-use previously created visualizations. With SSVG, we don't need to implement hit testing or give up inspecting DOM elements. SSVG can be a useful tool because researchers often have to decide between smooth rendering and the flexible experimentation with the relatively easy D3.js implementation that they are used to."

They noted that SSVG is easy to use: "The first steps are super easy, and there are good examples on the website [..]. You need a little bit of time to get used to debugging, switching back to the SVG, but even that is pretty straightforward. If things don't work, you can check if the normal SVG worked to check if the issue is in your code or in how SSVG translated it to Canvas. [..] Even our new students are getting the hang of it and are able to use SSVG well. It's working out nicely." However, some shortcomings exist: "Not yet all features of D3.js are supported. [..] I try to avoid D3.js' nested selection groups, as SSVG seems to struggle with that".

8 Discussion

SSVG consists of multiple techniques that are executed in a pipeline to take data visualizations created with SVG, CSS, JavaScript, and D3.js, and render them smoothly in a

Authorized licensed use limited to: Brown University. Downloaded on September 04,2022 at 03:14:38 UTC from IEEE Xplore. Restrictions apply.

separate thread. Below, we discuss each step before discussing high level points.

1. DOM to VDOM. This research demonstrates that it is possible to populate a VDOM based on the DOM, and keep it up to date by intercepting and redirecting JavaScript DOM function calls. The ability to automatically obtain an interactive VDOM, and de-coupling the SVG specification from SVG rendering, opens up many opportunities for data visualization system researchers that want to minimize changes to the visualization creation workflow based on SVG and D3.js while being able to customize visualization output. In this work, the VDOM is used for fast rendering of thousands of nodes. Other works may focus on extending the capabilities of SVGs, such as introducing `z-index` attributes to manage the layering of elements, which is not possible with the current SVG standard. Other possible extensions include the simplification of existing features, such as the direct use of gradients or arrow heads within SVG elements, as opposed to requiring separate SVG definition elements to be created. We also hope that other work will take advantage of a dynamic VDOM to target more devices, such as done in the published e-Ink data visualization project, or to automatically reposition or rescale elements to fit different screen sizes such as smart phones. Lastly, the VDOM may also be useful for systems that can make systematic accessibility adjustments for visualizations, such as to improve contrast or systematically avoid colors that are hard to distinguish by red-green colorblind users. Because of the ability to automatically obtain a dynamic VDOM for most DOM-based visualizations, such systems would be applicable to a wide set of visualizations, often without necessary changes, and a consistent set of accessibility settings could be applied across these visualizations. To summarize, *the ability to obtain a dynamic VDOM allows visualization system researchers to customize visualization rendering to improve aspects such as performance, expressivity, responsiveness, and accessibility*.

2. VDOM to Worker. Our work showcases that a VDOM can be automatically synchronized to a worker thread to parallelize work. Our approach bundles VDOM updates and sends the data to the worker thread via JavaScript worker messages and SharedArrayBuffers. Clearly defined update messages, such as ones with unique element identifiers, have the advantage of synchronizing the VDOM on the worker more robustly, but come with significant overhead in message size and work to update the VDOM element-by-element, rather than allowing grouped attribute changes. To support fast rendering of thousands of elements, we minimize lookups of individual elements and batch attribute changes. This comes with challenges of uniquely identifying elements if elements are removed and added at the same time as they are being modified, and we do not yet support nested D3.js selections that further complicate the update structure. These trade-offs between synchronization approaches should be explored in future visualization systems aiming to enable parallel work.

3. Canvas Renderer. Our renderer connects the paths of all elements within the same parent element that have the same color. This speeds up rendering, but the grouping changes the order in which elements are painted. In some cases, this can change the layering of elements compared with the original SVG. SSVG comes with a fallback option to render elements individually to more reliably arrive at the same result as SVG rendering. Ideally, an intelligent renderer would identify which elements can be grouped into batched paint calls without layering issues, and which elements need to be rendered individually. To further speed up rendering and to increase reliability, data visualization system researchers should explore strategies to intelligently group Canvas drawing calls.

4. OffscreenCanvas Rendering. SSVG demonstrates that multi-thread rendering using OffscreenCanvas is possible for data visualizations without additional technical burdens on visualization creators. To achieve this, SSVG synchronizes the SVG VDOM to the worker thread. Other possible architectures could make multi-threading easier for canvas visualizations, e.g., by sending Canvas drawing calls to the worker, as opposed to sending a virtual DOM. Future visualization system research should explore these and other strategies to make OffscreenCanvas useful and feasible for data visualization creators. Beyond OffscreenCanvas specifically, our work shows that giving visualization researchers the ability to customize rendering allows us to benefit from new technologies. This may be advantageous for data visualization when further future technologies are released. For example, Vulcan is a new graphics and compute API that is currently starting to be supported by browsers, and the ability to customize rendering allows the data visualization community to take advantages of new developments such as these while also benefiting from the open format of SVGs.

Overall, SSVG combines several existing and new ideas to achieve Canvas- and WebGL-based rendering on a worker thread of traditional SVG and D3.js visualizations. Each step — DOM, CSS and JS → VDOM → worker VDOM → Canvas/WebGL drawing — comes with opportunities and trade-offs. SSVG focuses on replicating SVG and DOM capabilities most efficiently with optimized communication and rendering, and other systems may focus on other aspects, such as added expressivity and accessibility.

Custom APIs. The benefits of by-passing the browser's rendering pipeline can also be achieved by using Canvas directly, or by using one of the available frameworks that provide a custom API to design visualizations. While this solution makes sense in special cases, we believe it is important for the data visualization ecosystem to continue to invest in the open, inspectable, and flexible format of SVG, and build on established works like D3.js.

Advantages Over Browser-Based Rendering. The performance gains of SSVG are achieved by replacing the normal browser rendering process, by batching attribute changes, by avoiding work on unchanged nodes, and by taking advantage of new multi-thread technology. In each step, we take advantage of how data visualizations are typically created: CSS rules typically do not change over time, many SVG features are rarely needed, and many nodes' attributes are often changed at once. This domain knowledge of how SVGs are typically used in the data visualization domain allows us to make assumptions, simplifications, and performance improvements that are not possible for browser vendors without breaking backward compatibility. Through public discussions and personal communication with the Google Chrome SVG team, we have advocated for

Authorized licensed use limited to: Brown University. Downloaded on September 04,2022 at 03:14:38 UTC from IEEE Xplore. Restrictions apply.

systematic improvements to SVG rendering. In some cases, we were successful: as part of the discussion surrounding our suggestion to add a "bulk setAttribute" function on many nodes at crbug.com/978513, improvements in the existing setAttribute function were identified, implemented and deployed in Chrome that speed up its execution by about 20 percent. We continue to work with the Chrome team to share our lessons learned. However, from our discussions, it is clear that substantial improvements, such as the 3–9× speedup provided by SSVG, are not feasible for browser vendors because they can not make the assumptions and simplifications that we can make within the data visualization community. Therefore, we argue that community-driven visualization systems are a useful path toward greater SVG performance in data visualization.

9 Conclusion

We contribute Scalable Scalable Vector Graphics (SSVG): a JavaScript library for faster rendering and interaction of Scalable Vector Graphics (SVG). As this is a foundational technology for the visualization community, we hope SSVG will enable the development of new and complex data visualizations on the Web. We show that SSVG automatically enables a 3–9× performance boost on four rendering-intensive visualizations. We use an online collection of data visualizations to demonstrate compatibility.

We believe that enabling visualization creators to use their generally-preferred technology—SVG—allows them to increase the number of iterations within the design process because it is a high-level specification with CSS customizability. This may aid researchers to focus on the visual design and interpretation, rather than on Canvas and WebGL implementation details. With SSVG, we encourage continued SVG-based visualization research because the result is open, meaning it can be easily customized, analyzed and learned from—crucial elements for research. To visualization system designers, we show how Canvas rendering and multi-threading can be integrated into users' visualization without many changes in their way to create visualizations, and hope that this approach will inspire more work that adds to existing infrastructure. Ultimately, we hope to contribute to the success of the visualization ecosystem by making Scalable Vector Graphics more scalable.

Acknowledgments

The authors would like to thank Alex Ahmed, Sara Di Bartolomeo, Aditeya Pandey, Lulu Liu, and Laura South for experimenting with SSVG and providing useful feedback and advice. This work was supported by the Khoury College of Computer Sciences, Northeastern University.

References

[1] B. V. Almende, "vis.js: A dynamic, browser based visualization library," 2013. Accessed: Mar. 29, 2019. [Online]. Available: http://visjs.org

[2] D. Anderson, "Mozilla GFX team blog: Off-main-thread painting," 2017. Accessed: Mar. 31, 2019. [Online]. Available: https://mozillagfx.wordpress.com/2017/12/05/off-main-thread-painting/

[3] P. Bakaus, "The illusion of motion," 2014. Accessed: Mar. 29, 2014. [Online]. Available: https://paulbakaus.com/tutorials/performance/the-illusion-of-motion/

[4] D. H. Ballard, M. M. Hayhoe, P. K. Pook, and R. P. N. Rao, "Deictic codes for the embodiment of cognition," *Behav. Brain Sci.*, vol. 20, pp. 723–767, 1995.

[5] K. Baxter, "WorkerDOM: JavaScript concurrency and the DOM," 2018. Accessed: Dec. 4, 2019. [Online]. Available: https://github.com/ampproject/worker-dom

[6] M. Bostock, "Force-directed graph," 2012. Accessed: Mar. 29, 2019. [Online]. Available: https://observablehq.com/@d3/force-directed-graph

[7] M. Bostock, V. Ogievetsky, and J. Heer, "D3 data-driven documents," *IEEE Trans. Vis. Comput. Graphics*, vol. 17, no. 12, pp. 2301–2309, Dec. 2011.

[8] J. Brandel, "Two.js: A renderer agnostic two-dimensional drawing api for the web," 2012. Accessed: Mar. 29, 2019. [Online]. Available: https://two.js.org

[9] R. Cabello, "Three.js: 3D Javascript library," 2010. Accessed: Mar. 29, 2019. [Online]. Available: https://threejs.org

[10] S. K. Card, A. Newell, and T. P. Moran, *The Psychology of Human-Computer Interaction*, Hillsdale, NJ, USA: L. Erlbaum Associates Inc., 1983.

[11] M. Claypool, K. Claypool, and F. Damaa, "The effects of frame rate and resolution on users playing first person shooter games," *Proc. SPIE - Int. Soc. Optical Eng.*, vol. 6071, pp. 1–11, 2006, doi: 10.1117/12.648609.

[12] Cytoscape Consortium, "Cytoscape: Network data integration, analysis, and visualization in a box," 2001. Accessed: Mar. 29, 2019. [Online]. Available: https://cytoscape.org

[13] Facebook, "React," 2013. Accessed: Dec. 4, 2019. [Online]. Available: https://reactjs.org/

[14] Goodboy Digital Ltd., "Pixijs: The HTML5 creation engine," 2013. Accessed: Mar. 29, 2019. [Online]. Available: http://www.pixijs.com

[15] Google, "Angular," 2010. Accessed: Dec. 4, 2019. [Online]. Available: https://angular.io/

[16] Google, "Web fundamentals: Measure performance with the RAIL model," 2015. Accessed: Mar. 29, 2019. [Online]. Available: https://web.dev/rail/

[17] Google, "Web fundamentals: Rendering performance," 2020. Accessed: Aug. 1, 2020. [Online]. Available: https://developers.google.com/web/fundamentals/performance/rendering/

[18] W. D. Gray and D. A. Boehm-Davis, "Milliseconds matter: An introduction to microstrategies and to their use in describing and predicting interactive behavior" *J. Exp. Psychol. Appl.*, vol. 6, no. 4, pp. 322–335, 2000.

[19] H. Gueziri, M. J. McGuffin, and C. Laporte, "Latency management in scribble-based interactive segmentation of medical images," *IEEE Trans. Biomed. Eng.*, vol. 65, no. 5, pp. 1140–1150, May 2018.

[20] P. Isenberg *et al.*, "vispubdata.org: A metadata collection about IEEE visualization (VIS) publications," *IEEE Trans. Vis. Comput. Graphics*, vol. 23, no. 9, pp. 2199–2206, Sep. 2017.

[21] A. Jie, "Proton: JavaScript particle engine," 2016. Accessed: Mar. 29, 2019. [Online]. Available: https://drawcall.github.io/Proton/

[22] J. Jo, F. Vernier, P. Dragicevic, and J. Fekete, "A declarative rendering model for multiclass density maps," *IEEE Trans. Vis. Comput. Graphics*, vol. 25, no. 1, pp. 470–480, Jan 2019.

[23] K. Klamka, T. Horak, and R. Dachselt, "Watch+strap: Extending smartwatches with interactive strapdisplays," in *Proc. ACM CHI Conf. Human Factors Comput. Syst.*, 2020, pp. 1–15.

[24] J. Lehni and J. Puckey, "Paper.js — the swiss army knife of vector graphics scripting," 2011. Accessed: Dec. 4, 2019. [Online]. Available: http://paperjs.org

[25] G. Lerner, "canvg: JavaScript SVG parser and renderer on Canvas," 2014. Accessed: Mar. 29, 2019. [Online]. Available: https://github.com/canvg/canvg

[26] Z. Liu and J. Heer, "The effects of interactive latency on exploratory visual analysis," *IEEE Trans. Vis. Comput. Graphics*, vol. 20, no. 12, pp. 2122–2131, Dec. 2014.

[27] R. B. Miller, "Response time in man-computer conversational transactions," in *Proc. Dec. 9–11, 1968, Fall Joint Comput. Conf. Part I*, 1968, pp. 267–277.

[28] Mozilla, "MDN web docs: Blob," 2013. Accessed: Mar. 29, 2019. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/Blob

[29] Mozilla, "MDN Web Docs: SharedArrayBuffer," 2016. Accessed: Mar. 29, 2016. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/SharedArrayBuffer

Authorized licensed use limited to: Brown University. Downloaded on September 04,2022 at 03:14:38 UTC from IEEE Xplore. Restrictions apply.

[30] Mozilla Developer Network, "The OffscreenCanvas element," 2016. Accessed: Sep. 24, 2016. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/OffscreenCanvas/OffscreenCanvas#Browser_compatibility
[31] D. Ren, B. Lee, and T. Höllerer, "Stardust: Accessible and transparent GPU support for information visualization rendering," *Comput. Graph. Forum*, vol. 36, no. 3, pp. 179–188, Jun. 2017.
[32] W. Ritter, G. Kempter, and T. Werner, "User-acceptance of latency in touch interactions," in *Proc. Int. Conf. Universal Access Human-Comput. Interact. Access Interaction*, 2015, pp. 139–147.
[33] D. Saffo, M. Schwab, M. A. Borkin, and C. Dunne, "GeoSocialVis: Visualizing geosocial academic co-authorship networks by balancing topology- and Geography- Based Layouts," in *Proc. IEEE Vis. Conf.*, 2019, doi: 10.31219/osf.io/ykwah.
[34] A. Satyanarayan, K. Wongsuphasawat, and J. Heer, "Declarative interaction design for data visualization," in *Proc. 27th Annu. ACM Symp. User Interface Softw. Technol.*, 2014, pp. 669–678.
[35] C. Schulz, A. Nocaj, J. Goertler, O. Deussen, U. Brandes, and D. Weiskopf, "Probabilistic graph layout for uncertain network visualization," *IEEE Trans. Vis. Comput. Graphics*, vol. 23, no. 1, pp. 531–540, Jan. 2017.
[36] P. Shannon *et al.*, "Cytoscape: A software environment for integrated models of biomolecular interaction networks," *Genome Res.*, vol. 13, no. 11, pp. 2498–2504, 2003.
[37] B. Smus, "Performance of canvas versus SVG," 2009. Accessed: Mar. 29, 2009. [Online]. Available: https://smus.com/canvas-vs-svg-performance/
[38] Y. Wang, D. Archambault, C. E. Scheidegger, and H. Qu, "A vector field design approach to animated transitions," *IEEE Trans. Vis. Comput. Graphics*, vol. 24, no. 9, pp. 2487–2500, Sep. 2018.
[39] M. Wattenberg and F. Viégas, "Wind map," 2012. Accessed: Mar. 29, 2019. 2012. [Online]. Available: http://hint.fm/wind/
[40] World Wide Web Consortium, "HTML living standard: The OffscreenCanvas interface," 2016. Accessed: Mar. 29, 2019. [Online]. Available: https://html.spec.whatwg.org/multipage/canvas.html#the-offscreencanvas-interface
[41] E. You, "Vue.js," 2016. Accessed: Dec. 4, 2016. [Online]. Available: https://vuejs.org

Michail Schwab received the graduate degree in physics from the University of Konstanz, and the PhD degree in computer science from Northeastern University. He is currently an engineer and researcher with Google. His research interests include scalability of the data visualization ecosystem, working to allow visualizations to display more data on more devices and be useful to more people. He works in the areas of interaction, rendering, collaboration, and responsiveness for mobile visualizations and other form factors.

David Saffo is working toward the PhD degree with the Khoury College of Computer Sciences, Northeastern University. Researching in the Visualization @ Khoury lab, his work centers around immersive analytics, virtual reality, and collaboration. This work is motivated by his passion for applying human-computer interaction research to new and emerging interdisciplinary areas.

Nicholas Bond received the bachelor's degree in computer science and interaction design from Northeastern University. He is a software engineer with Voltus where he creates tools for visualizing data related to power grids and energy markets.

Shash Sinha received the BE (Hons) degree in software engineering from the University of Waikato, New Zealand, in 2017, and the ScM degree in computer science from Brown University, in 2020. He is currently a employed at Amazon in Seattle, WA, where he incorporates modern visualization techniques in his work regularly.

Cody Dunne received the MS and PhD degrees in computer science from the University of Maryland, in 2009 and 2013. He is currently an assistant professor with Northeastern University. His research focuses on applying data visualization and human-computer-interaction principles to real-world problems and generalizing the results — often via design studies. He generally works with problems that combine showing network topology, position in space, values of attributes, changes to these over time, and how changes or events happen in sequence.

Jeff Huang received the graduate and master's degrees in computer science from the University of Illinois at Urbana-Champaign, and the PhD degree in information science from the University of Washington in Seattle. He is currently an associate professor in computer science with Brown University. His research interests include human-computer interaction focuses on behavior-powered systems, spanning the domains of mobile devices, personal informatics, and web search.

James Tompkin is currently an assistant professor of computer science with Brown University. His research at the intersection of computer vision, computer graphics, and human-computer interaction helps to develop new visual computing tools and experiences. His doctoral work at University College London on large-scale video processing and exploration techniques led to creative exhibition work in the Museum of the Moving Image in New York City. Postdoctoral work at Max-Planck-Institute for Informatics and Harvard University helped create new methods to edit content within images and videos. Recent research has developed new machine learning techniques for view synthesis for VR, image editing and generation, and style and content separation.

Michelle A. Borkin received the MS and PhD degrees in applied physics from Harvard University in Cambridge, MA, in 2011 and 2014. She is currently an assistant professor with the Khoury College of Computer Sciences, Northeastern University, Boston, MA. Her research interests include data visualization, human-computer interaction, and application work across domains including astronomy, physics, and medical imaging.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/csdl.

Authorized licensed use limited to: Brown University. Downloaded on September 04,2022 at 03:14:38 UTC from IEEE Xplore. Restrictions apply.