Comparing the Built-In Application Architecture Models in the Web Browser

Antero Taivalsaari[1], Tommi Mikkonen[2], Cesare Pautasso[3] and Kari Systä[4]
[1]Nokia Technologies, Finland, [2]University of Helsinki, Finland
[3]USI Lugano, Switzerland, [4]Tampere University of Technology, Finland

***Abstract*—Depending on one's viewpoint, a generic standards-compatible web browser supports three, four or five built-in application rendering and programming models. In this paper, we provide an overview of the built-in client-side web application architectures. While the dominance of the base HTML/CSS/JS technologies cannot be ignored, we foresee Web Components and WebGL gaining popularity as the world moves towards more complex and even richer web applications, including systems supporting virtual and augmented reality.**

***Keywords*—Web development, web application architectures**

I. Introduction

As a result of the web browser evolution that has occurred in the past two decades, today's web browsers support a number of complementary, partially overlapping rendering and development models. These models include the dominant Document Object Model (DOM) rendering architecture. They also include the *Canvas 2D Context API* as well as *WebGL*. Additionally, there are important and emerging technologies such as *Scalable Vector Graphics* (SVG) and *Web Components* that complement the basic DOM architecture.

The choice between the rendering architectures can have significant implications on the structure of client-side web applications. Effectively, all of the technologies introduce their own distinct programming models that the developers are expected to use. Furthermore, all of them have varying levels of framework, library and tool support available to simplify the actual application development work. The DOM-based approach is by far the most popular and most deeply ingrained, but the other technologies deserve a glimpse as well.

In this paper, we provide a comparison of the *built-in client-side web application architectures*, i.e., the programming capabilities that the web browsers provide out-of-the-box before any additional libraries are loaded. This is a topic that has received surprisingly little attention in the literature. While there are countless papers on specific web development technologies, and hundreds of libraries have been developed *on top of the browser*, we are unaware of any papers comparing the built-in development models of the browser itself.

The paper is motivated by the *recent trend toward simpler, more basic approaches in Web development*. According to recent studies, the vast majority (up to 86%) of web developers feel that the Web and JavaScript ecosystems have become far too complex (http://stateofjs.com/2016). There is a movement to go back to the roots of web application development by building directly upon what the web browser can provide without the added layers introduced by various libraries and frameworks. The recent "*zero framework manifesto*" crystallizes this desire for simplicity [1]. However, even the "vanilla" browser offers a cornucopia of choices when it comes to application development, reflecting the historical, organic evolution of the web browser as a software platform.

II. Client-Side Web Rendering Architectures

DOM / DHTML. In web parlance, the *Document Object Model* (DOM) is a platform-neutral API that allows programs and scripts to dynamically access and update the content, structure and style of web documents. DOM is the foundation for *Dynamic HTML* (DHTML) – the combination of HTML, Cascading Style Sheets (CSS) and JavaScript – that allows web documents to be manipulated using a combination of declarative (CSS, HTML) and imperative (JavaScript) development styles. Logically, the DOM can be viewed as an *attribute tree* that represents the contents of the web page that is currently displayed by the web browser.

In the web browser, the DOM serves as the foundation for a *retained (automatically managed) graphics architecture*. In such a system, the application developer has no direct, immediate control over rendering. Rather, all the drawing is performed indirectly by manipulating the DOM tree by modifying its nodes; the browser will then decide how to optimally lay out and render the display after each change.

DOM attributes can be defined from HTML, accessed and evaluated from CSS and manipulated from JavaScript. As a result, a number of entirely different development styles are possible, ranging from purely imperative usage to a combination of declarative styles using HTML and CSS. For instance, it is possible to create impressive 2D/3D animations using the latest version of CSS animation capabilities without writing a single line of imperative JavaScript code.

In practice, very few developers use the raw, low-level DOM interfaces directly nowadays. The DOM and DHTML serve as the foundation for an extremely rich library and tool ecosystem that has emerged on top of the base technologies. The manipulation of DOM attributes is usually performed using higher-level convenience functions provided by popular JavaScript / CSS libraries and frameworks.

Canvas. Canvas (officially known as the *Canvas 2D Context API*) is an HTML5 feature that enables dynamic, scriptable rendering of two-dimensional (2D) shapes and bitmap images

978-1-5090-5729-0/17 $31.00 © 2017 IEEE
DOI 10.1109/ICSA.2017.23

(https://www.w3.org/TR/2dcontext/). It is a low level, imperative API that does not provide any built-in scene graph or advanced event handling capabilities. It that regard, Canvas offers much lower level graphics support than the DOM or SVG APIs that will automatically manage and (re)render complex graphics elements.

Canvas objects are drawn in *immediate mode*. This means that once a shape such as a rectangle is drawn using Canvas API calls, the rectangle is immediately forgotten by the system. If the position of the rectangle needs to be changed, the entire scene needs to be repainted, including any objects that might have been invalidated (covered) by the rectangle. In the equivalent DOM or SVG case, one could simply change the position attributes of the rectangle, and the browser would then automatically determine how to optimally re-render all the affected objects.

The user interaction capabilities of the Canvas API are minimal. A limited form of event handling is supported by the Canvas API with *hit regions* (https://developer.mozilla.org/en-US/docs/Web/API/Canvas_API/Tutorial/Hit_regions_and_accessibility).

Conceptually, Canvas is a low level API upon which a higher-level rendering engine might be built. Although canvas elements are created in the browser as subelements in the DOM, it is entirely possible to create just one large canvas element, and then perform all the application rendering and event handling inside that element. There are JavaScript libraries that add event handling and scene graph capabilities to the canvas element. For instance, with *Paper.js* (http://paperjs.org/) or *Fabric.js* (http://fabricjs.com/) libraries, it is possible to paint a canvas in layers, and then recreate specific layers, instead of having to repaint the entire scene manually each time. Thus, the Canvas API can be used as a full-fledged application rendering model of its own.

WebGL (http://www.khronos.org/webgl/) is a cross-platform web standard for hardware accelerated 3D graphics API developed by Khronos Group, Mozilla, and a consortium of other companies including Apple, Google and Opera. The main feature that WebGL brings to the Web is the ability to display 3D graphics natively in the web browser without any plug-in components. WebGL is based on OpenGL ES 2.0 (http://www.khronos.org/opengles), and it leverages the OpenGL shading language GLSL for shader definition. A comprehensive JavaScript API is provided to open up OpenGL programming capabilities to JavaScript programmers.

In a nutshell, WebGL provides a JavaScript API for rendering interactive, immediate-mode 3D (and 2D) graphics within any compatible web browser without the use of plug-ins. WebGL enables Web applications to take advantage of the Graphics Processing Unit (GPU) to accelerate complex rendering, image processing and visual effects. WebGL applications consist of control code written in JavaScript and shader code written in GLSL that is typically executed on a GPU.

WebGL is widely supported in modern desktop browsers. However, its availability and usability is dependent on various factors such as the GPU supporting it. Today, many of the mobile browsers (e.g., the Android browser) still do not support WebGL by default. Furthermore, even in many desktop computers WebGL applications may run poorly unless the computer has a graphics card that provides sufficient capabilities to process OpenGL efficiently.

Just like the Canvas API, the WebGL API is a rather low-level API that does not automatically manage rendering or support high-level events. From the application developer's viewpoint, the WebGL API may in fact be too cumbersome to use directly without utility libraries, such as *A-Frame, BabylonJS, three.js, O3D, OSG.JS, CopperLicht* and *GLGE*. For instance, setting up typical view transformation shaders, loading scene graphs and 3D objects in the popular industry formats can be very tedious and requires writing a lot of source code.

Scalable Vector Graphics (SVG) is an XML-based vector image format for two-dimensional graphics with support for interactivity, affine transformations and animation. The *SVG Specification* [5] is an open standard published by the World Wide Web Consortium (W3C) originally in 2001. Although bitmap images were supported since the early days of the Web (the `<IMG>` tag was introduced in the Mosaic browser in 1992), vector graphics support came much later via SVG.

While SVG was originally just a vector image format, SVG support has been integrated closely with the web browser to provide comprehensive means for creating interactive, resolution-independent content for the Web. Similar to the HTML DOM, SVG images can be manipulated using DOM APIs via HTML, CSS and JavaScript code. This makes it possible to create shapes such as lines, Bezier/elliptical curves, polygons, paths and text and images that be resized, rescaled and rotated programmatically using a set of built-in affine transformation and matrix functions.

Just like with the HTML DOM, SVG support in the web browser is based on a *retained (managed) graphics architecture*. Inside the browser, each SVG shape is represented as an object in a *scene graph* that is rendered to the display automatically by the web browser. When the attributes of an SVG object are changed, the browser will calculate the most optimal way to re-render the scene, including the other objects that may have been impacted by the change.

In the earlier days of the Web, SVG was the only means to implement a scalable, "morphic" graphics system, which is why the SVG DOM API was used as the foundation for graphics implementation, e.g., in the original Lively Kernel web programming system [4]. The following link provides a reference to a more comprehensive, "Lively-like" example of an SVG-based application that includes interactive capabilities (image rescaling and rotation based on mouse events) as well: https://dev.w3.org/SVG/tools/svgweb/samples/svg-files/photos.svg/.

In general, it is important to summarize that in the context of the Web, SVG is *much more than just an image format*. Together with event handling capabilities, affine transformations, gradient support, clipping, masking and composition features, SVG can be used as the basis for a full-fledged, standalone graphical application architecture or windowing system.

Web Components. Web Components (https://www.w3.org/TR/#tr_Web_Components) are a set of features added to the HTML and DOM specifications to enable the creation of reusable widgets or components in web documents and applications. The intention behind web components is to bring component-based software engineering principles to the World Wide Web, including the interoperability of higher-level HTML elements, encapsulation, information hiding and the general ability to create reusable, higher-level UI components that can be added flexibly to web applications.

An important motivation for web components is the *fundamentally brittle nature of the Document Object Model*. The brittleness comes from the global nature of elements in the DOM created by HTML, CSS and JavaScript code. For example, when you use a new HTML `id` or `class` in your web application or page, there is no easy way to find out if it will conflict with an existing name used by the page already earlier. Subtle bugs creep up, style selectors can suddenly go out of control, and performance can suffer, especially when attempting to combine code written by multiple authors [2]. Over the years various tools and libraries have been invented circumvent the issues, but the fundamental brittleness issues remain. The other important motivation is the *fixed nature of the standard set of HTML elements*. Web components make it possible to extend the basic set of components and support dynamically downloadable components across different web pages or applications.

Web components are built on top of a concept known as the *Shadow DOM*. In technical terms, the Shadow DOM introduces the concept of nested subtrees in the Document Object Model. These subtrees can be viewed conceptually as "icebergs" that expose only their tip while the implementation details remain invisible (and inaccessible) under the surface. Unlike regular branches in the DOM tree, shadow trees provide support for *scoped styles* and *DOM encapsulation*, thus obeying the well-known separation of concerns and modularity principles that encourage strong decoupling between public interfaces and implementation details [3]. Utilizing the Shadow DOM, the programmer can bundle CSS with HTML markup, hide implementation details, and create self-contained reusable components in vanilla JavaScript without exposing the implementation details or having to follow awkward naming conventions to ensure unique naming.

At the technical level, a shadow DOM tree is just a normal DOM tree with two differences: 1) how it is created and used, and 2) how it behaves in relation to the rest of the web page. Normally, the programmer creates DOM nodes and appends those nodes as children of another element. With shadow DOM, the programmer creates a *scoped DOM tree* that is attached to the element but that is separate from its actual children. The element it is attached to is its *shadow host*. Anything that the programmer adds to the shadow tree becomes local to the hosting element, including `<style>`. This is how shadow DOM achieves CSS style scoping.

Note that up until recently, many browsers did not support web components yet. Therefore, they had to be emulated in the form of *polyfill libraries* that implement the missing functionality (http://webcomponents.org/polyfills/). As of this writing, native support for the Shadow DOM is available both in desktop and Android versions of Google Chrome, in desktop version of Opera, as well as in Apple Safari desktop version 10. Mozilla Firefox currently supports web components only as a developer option.

III. Comparison, Use Cases and Predictions

An overview and a summary of the different approaches is presented in Table I. The table covers topics such as the overall development paradigm (imperative vs. declarative), rendering architecture (retained/managed vs. immediate), information hiding support, primary intended usage domain and current popularity. We also provide impressions on more subjective factors such as technology maturity, abstraction level and ease of code reuse. Finally, the table summarizes whether each technology provides support for defining animations in a declarative fashion (as opposed to having to write lengthy JavaScript timer scripts to drive animations), as well as whether the technology is supported by mobile browsers.

Primary Use Cases. The following bullets provide a basic characterization on the primary use cases for each technology.

- *DOM/DHTML*. HTML was originally developed as a declarative markup language for creating static documents and forms. Over the years, the use of DOM/DHTML has expanded to almost every imaginable use case. Today, DOM-based development approach dominates the web development landscape. This approach is declarative in nature, so the browser largely decides about rendering; this simplifies the development of web sites that look like documents, but can complicate the creation of sites that should behave like desktop applications or require control of the display at pixel level.
- *Canvas*. The Canvas API was introduced at a time when there was no other way to render lines, circles, rectangles or other low-level graphics imperatively in the browser. Currently, the Canvas API is utilized primarily by game developers. It is also used sometimes in regular web pages for drawing custom graphical content.
- *WebGL*. From technical viewpoint, WebGL is basically a thin JavaScript wrapper over native OpenGL interfaces for providing a programmatic API inside the web browser to achieve hardware-accelerated (GPU) rendering. As a result, the use cases of WebGL are a direct derivative of the OpenGL use cases, including (especially) game development, computer-aided design (CAD), scientific visualization, flight simulation, virtual reality, or any other case in which advanced 3D (or 2D) graphics rendering capabilities are needed. WebGL is an imperative, low-level API that places a lot of requirements on developer skills. Today, the use of WebGL is still marginal, but we foresee it gaining importance as the need to render VR/AR content in the web browser increases.
- *SVG*. In the context of the web browser, SVG has a dual role. First and foremost, SVG is a vector image format for rendering scalable graphics content on web pages. However, SVG can also be used as a rich, generic graphics

Table I
COMPARISON OF BUILT-IN CLIENT-SIDE RENDERING TECHNOLOGIES

	DOM / DHTML	Canvas	WebGL	SVG	Web Components
Development Paradigm	Declarative and imperative	Imperative	Imperative	Declarative and imperative	Declarative and imperative
Rendering Architecture	Retained	Immediate (explicit repainting required)	Immediate (explicit repainting required)	Retained	Retained
Information Hiding	No	Not applicable (no namespace support)	Not applicable	No (except when creating multiple SVG images)	Yes (Shadow DOM encapsulation and scoped styles)
Primary Usage Domain	Documents and forms	2D graphics (e.g., in games)	3D/2D graphics especially in games and VR/AR	2D image rendering	Web applications and graphical user interfaces
Popularity	Ubiquitous	Popular in specific use cases	Limited	Popular in specific use cases	Growing
Technology Maturity	Mature	Mature	Mature (implementation underway)	Mature	Emerging (standardization underway)
Abstraction Level	Medium	Very low	Low	Medium	High
Ease of Code Reuse	Low to medium	Low	Medium (shaders)	Low to high (high as an image format)	High
Declarative Animation Support	Yes (with CSS)	No	No	Yes	Yes
Mobile Browser Support	Yes	Yes	Not in Android (add-ons required)	Yes	Not in iOS (polyfill add-ons required)

context to drive scene graph based applications with support for complex event handling, affine transformations (rotation, zooming, scaling, shearing), gradients, clipping, masking and object composition.

- *Web Components*. Web components are the "dark horse" in web development – they are still little known to most developers, and it is difficult to place betting odds on their eventual success. Web components reintroduce well-known (but hitherto missing) software engineering principles and practices into the web browser, including modularity and the ability to create higher-level, general-purpose UI components that can be flexibly added to web applications. Web components cater to nearly any imaginable use case but they are especially well-suited to the development of full-fledged Web applications that require an extensible set of GUI widgets.

Predictions. Out of the technologies discussed in this paper, DOM/DHTML will likely maintain its dominant role as the base technology. However, we foresee especially WebGL gaining more popularity in the future, as the world moves towards richer media experiences, including virtual and augmented reality. WebGL enables browser-based, installation-free high-performance applications for viewing VR/AR content. WebGL will be also increasingly important for game developers. In fact, the most dramatic impact of WebGL is that it will effectively eliminate the "last safe bastion" of traditional binary applications, allowing the creation of portable high-performance applications in the context of the web browser.

Regarding web components, it is still too early to declare victory or failure. Since web components offer a more disciplined approach to DOM/DHTML programming, reintroduce established software engineering principles, and generally alleviate the "spaghetti code" issues that have resurfaced with the Web, we would certainly like to see them succeed. In reality, the main obstacle to the wider adoption of web components are the predominant JavaScript libraries that provide additional abstraction layers on top of the underlying DOM and basic browser features.

We also believe that the Web would benefit from a high-performance, low-level 2D graphics API that would provide a more comprehensive feature set and direct drawing capabilities without any historical baggage of the Canvas API. Apple's recent WebGPU API proposal [6] is an interesting step towards addressing both 2D/3D rendering capabilities.

IV. CONCLUSIONS

Web development today presents a cornucopia of choices on all fronts. In this paper, we have studied one of the perhaps most overlooked areas in web development: the client-side web rendering architectures that have been built into the generic web browser. The rapid pace of innovation has put the developers in a complex position in which there are numerous ways to build applications on the Web – many more than most people realize, and also arguably more than are really needed.

ACKNOWLEDGMENTS

This work is supported by Academy of Finland (project 295913) and Nokia Technologies.

REFERENCES

[1] Bitworking.org. Zero Framework Manifesto: No More JS Frameworks, 2014. https://bitworking.org/news/2014/05/zero_framework_manifesto.
[2] T. Mikkonen and A. Taivalsaari. Web Applications — Spaghetti Code for the 21st Century. In *Proc. Int'l Conf. Software Engineering Research, Management and Applications (SERA'2008, Prague, Czech Republic, August 20-22, 2008)*, pages 319–328. IEEE Computer Society, 2008.
[3] D. L. Parnas. On the Criteria to be used in Decomposing Systems into Modules. *Communications of the ACM*, 15(12):1053–1058, 1972.
[4] A. Taivalsaari, T. Mikkonen, D. Ingalls, and K. Palacz. Web Browser as an Application Platform: the Lively Kernel Experience, Sun Labs Technical Report TR-2008-175. January 2008.
[5] W3C. Scalable Vector Graphics (SVG) Specification 1.1 (Second Edition), 2011. https://www.w3.org/TR/SVG/.
[6] WebGPU. WebGPU API Proposal by Apple, Inc. https://webkit.org/wp-content/uploads/webgpu-api-proposal.html, Jan 30, 2017.