Web User Interface Implementation Technologies: An Underview

Antero Taivalsaari[1], Tommi Mikkonen[2], Kari Systä[3] and Cesare Pautasso[4]
[1]*Nokia Technologies, Tampere, Finland*
[2]*Department of Computer Science, University of Helsinki, Helsinki, Finland*
[3]*Tampere University of Technology, Tampere, Finland*
[4]*Software Institute, Faculty of Informatics, USI, Lugano, Switzerland*
antero.taivalsaari@nokia.com, tommi.mikkonen@helsinki.fi, kari.systa@tut.fi, cesare.pautasso@usi.ch

Keywords: Web User Interfaces, Web Programming, Web Rendering, Single Page Web Applications, Web Application Architectures.

Abstract: Over the years, the World Wide Web has evolved from a document distribution environment into a rich development platform that can run compelling, full-fledged software applications. However, the programming capabilities of the web browser – designed originally for relatively simple scripting tasks – have evolved organically in a rather haphazard fashion. Consequently, there are many ways to build applications on the Web today. Depending on one's viewpoint, current standards-compatible web browsers support three, four or even five built-in application rendering and programming models. In this paper, we provide an "underview" of the built-in client-side web application UI implementation technologies, i.e., a summary of those rendering models that are built into the standards-compatible web browser out-of-the-box. 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.

1 INTRODUCTION

The World Wide Web has become such an integral part of the human society that it is often forgotten that the Web did not even exist thirty years ago. The original design sketches related to the World Wide Web date back to the late 1980s. The first web browser prototype for the NeXT computer was completed by Tim Berners-Lee in December 1990. The first version of the Mosaic web browser was released in February 1993, followed by the first commercially successful browser – Netscape Navigator – in late 1994. Widespread commercial use of the Web took off in the late 1990s (Berners-Lee and Fischetti, 2000) when the web browser became the most commonly used computer program, sparking a revolution that has transformed not only commerce but communication, social life and politics as well.

In desktop computers, today nearly all the important tasks are performed using the web browser. Even mobile applications can be viewed merely as "mirrors into the cloud" (Charland and Leroux, 2011). While native mobile apps still offer UI frameworks and widgets that are better suited to the limited screen size and input modalities of the devices, valuable content has moved gradually from mobile devices to the cloud, thus reducing the original role of the mobile apps considerably.

Interestingly, the programming capabilities of the Web have largely been an afterthought – designed originally for relatively simple scripting tasks. Due to different needs and motivations, there are many ways to make software run on the Web – many more than people generally realize. Furthermore, over the years these programming capabilities have evolved in a rather haphazard fashion. Consequently, there are various ways to build applications on the Web. Without considering any extensions, frameworks and add-on libraries, depending on one's viewpoint, the Web browser natively supports three, four or five different built-in application rendering and development models. Thousands of libraries and frameworks have then been implemented on top of these built-in models.

Furthermore, in addition to architectures that partition applications more coarsely between server and client side components (Gallidabino and Pautasso, 2017), it is increasingly possible to fine-tune the application logic by moving code flexibly between the

client and the server (Meijer, 2007). In this context, the rendering capabilities of the web browser are crucial in creating the presentation layer of the applications.

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*, there are few if any papers comparing the built-in user interface development models offered by the browser itself. The choice between these alternative development models has a significant impact on the overall architecture and structure of the resulting web applications. The choices are made more difficult by the fact that the web browser offers a number of overlapping features to accomplish even basic tasks, reflecting the historical, organic evolution of the web browser as an application platform.

This 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/). 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 (Bitworking.org, 2014). However, even the "vanilla" browser offers a cornucopia of choices when it comes to application development.

This paper is an extended version of an earlier conference paper (Taivalsaari et al., 2017). The extensions over the earlier work include extended background and motivation, more detailed description of the different technologies, together with sample code, and a more elaborated discussion on the broader architectural implications.

The structure of this paper is as follows. In Section 2, we provide an overview on the evolution of the web browser as an application platform. In Section 3, we dive into the built-in user interface development and rendering models offered by modern web browsers: (1) DOM, (2) Canvas, (3) WebGL, (4) SVG, and (5) Web Components. In Section 4, we provide a comparison, followed by a broader architectural discussion in Section 5. Finally, Section 6 concludes the paper.

2 EVOLUTION OF THE WEB BROWSER AS AN APPLICATION PLATFORM

Over the past twenty-five years, the World Wide Web has evolved from its humble origins as a *document sharing system* to a massively popular hypermedia application and content distribution environment – in short, the most powerful information dissemination environment in the history of humankind. This evolution has not taken place in a fortnight; it has not followed a carefully designed master plan either. Although the World Wide Web Consortium (W3C) has seemingly been in charge of the evolution of the Web, in practice the evolution has been driven largely by dominant web browser vendors: Mozilla, Microsoft, Apple, Google and (to a lesser degree) Opera. Over the years, these companies have had divergent, often misaligned business interests. While browser compatibility has improved significantly in recent years, the browser landscape is still truly a mosaic or cornucopia of features, reflecting organic evolution – or a tug of war – between different vendors over time.

Before delving into more technical topics, let us briefly revisit the evolution of the web browser as a software platform (Taivalsaari et al., 2008; Taivalsaari and Mikkonen, 2011; Anttonen et al., 2011).

Classic Web. In the early life of the Web, web pages were truly *pages*, i.e., page-structured documents that contained primarily text with interspersed images, without animation or any interactive content. Navigation between pages was based on simple *hyperlinks*, and a new web page was loaded from the web server each time the user clicked on a link. There was no need for asynchronous network communication between the browser and the web server. For reading user input some pages were presented as *forms*, with simple textual fields and the possibility to use basic widgets such as buttons and radio buttons. These types of "classic web" pages were characteristic of the early life of the Web in the early 1990s.

Hybrid Web. With the introduction of *DHTML* – the combination of HTML, Cascading Style Sheets (CSS), the JavaScript language (Flanagan, 2011), and the Document Object Model (DOM) – it became possible to create interactive web pages with built-in support for more advanced graphics and animation. The JavaScript language, introduced in Netscape Navigator version 2.0B almost as an afterthought in December 1995, made it possible to build animated interactive content by scripting directly the web browser.

In the second phase, web pages became increasingly interactive. Web pages started containing animated graphics and plug-in components that allowed

richer, more interactive content to be displayed. This phase coincided with the commercial takeoff of the Web during the dot-com boom of the late 1990s when companies realized that they could create commercially valuable web sites by displaying advertisements or by selling merchandise and services over the Web. Plug-in components such as Adobe Flash, RealPlayer, Quicktime and Shockwave were introduced to make it possible to construct web pages with visually enticing, interactive multimedia, allowing advanced animations, movie clips and audio tracks to be inserted in web pages.

In this phase, the Web started moving in directions that were unforeseen by its original designer, with web sites behaving more like multimedia presentations rather than static pages. Content mashups and web site cross-linking became popular and communication protocols between the browser and the server became increasingly advanced (Daniel and Matera, 2014). Navigation was no longer based solely on hyperlinks. For instance, Flash apps supported drag-and-drop and direct clicking/events on various types of objects, whereas originally no support for such features existed in browsers.

The Web as an Application Platform. In the early 2000s, the concept of Software as a Service (SaaS) emerged. Salesforce.com pioneered the use of the Web as a Customer Relationship Management (CRM) application platform in the early 2000s, demonstrating and validating the use of the Web and the web browser as a viable target platform for business applications. At that point, people realized that the ability to offer software applications seamlessly over the Web and then perform instant worldwide software updates could offer unsurpassed business benefits.

As a result of these observed benefits, people started to build web sites that behave much like desktop applications, for example, by allowing web pages to be updated partially, rather than requiring the entire page to be refreshed. Such systems often eschewed link-based navigation and utilized direct manipulation techniques (e.g., drag and drop features) borrowed from desktop-style applications instead. Interest in the use of the browser as an application platform was reinforced by the introduction of Ajax (Asynchronous JavaScript and XML) (Holdener, 2008). The key idea in Ajax was to use *asynchronous network communication* between the client and the server to decouple user interface updates from network requests. This made it possible to build web sites that do not necessarily block when interacting with the server and thus behave much like desktop applications, for example, by allowing web pages to be updated asynchronously one user interface element at a time, rather than requiring the entire page to be updated each and every time something changed. Although Ajax was primarily a specific technique rather than a complete development model or platform, it fueled further interest in building "Web 2.0" applications that could run in a standard web browser. This also increased the demand for a full-fledged programming language that could be used directly from inside the web browser instead of relying on any external plug-in components.

After the introduction of Ajax and the concept of *Single Page Applications* (SPAs) (Jadhav et al., 2015), the number of web development frameworks on top of the web browser has exploded. Today, there are over 1,300 officially listed JavaScript libraries (see http://www.javascripting.com/).

Server-Side JavaScript. The use of client-side web development technologies has spread also to other domains. For instance, after the introduction of Google V8 high-performance JavaScript engine, the use of the JavaScript language has quickly spread into server-side development as well. As a result, Node.js (https://nodejs.org/) has become a vast ecosystem of its own. In fact, the NPM (Node Package Manager) ecosystem has been growing even faster in recent years than the client-side JavaScript ecosystem. According to npmjs.com, there are more than 700,000 NPM packages at the time of this writing.

As already mentioned earlier, in this paper we shall focus only on client-side technologies and only on those technologies that have been included natively in standards-compatible web browsers. We feel that this is an area that is surprisingly poorly covered by existing research.

Non-Standard Development Models and Architectures. For the sake of completeness, it should be mentioned that over the years web browsers have supported various additional client-side rendering and development models. For instance, Java applets were an early attempt to include Java language and Java virtual machine (JVM) support directly in a web browser. However, because of the immaturity of the technology (e.g., inadequate performance of early JVMs) and resistance by some vendors, applets never became an officially supported browser feature.

In the late 2000s, so called *Rich Internet Application* (RIA) platforms such as Adobe AIR or Microsoft Silverlight were very much in vogue. RIA systems were an attempt to reintroduce alternative programming languages and libraries in the context of the Web in the form of browser plug-in components that each provided a complete platform runtime. For a comprehensive overview of RIA systems, refer to Castelyn's survey (Casteleyn et al., 2014). However, just as it

was predicted in (Taivalsaari and Mikkonen, 2011), the RIA phenomenon turned out to be rather short-lived. The same seems to be true also of various attempts to support native code execution directly from within the web browser. For instance, Google's Native Client offers a sandbox for running compiled C and C++ code in the browser, but it has not become very popular. Mozilla's classic NPAPI (Netscape Plugin Application Programming Interface) – introduced originally in 1995 by Netscape – has recently been removed from all the major browsers; for instance, Google Chrome stopped supporting it in 2015. Although there are some interesting ongoing efforts in this area – such as the W3C WebAssembly effort (http://webassembly.org/), it is now increasingly difficult to extend the programming capabilities of the web browser without modifying the source code of the browser itself.

3 CLIENT-SIDE WEB USER INTERFACE RENDERING TECHNOLOGIES

As summarized above, the history of the Web has undergone a number of evolutionary phases, reflecting the *document-oriented* – as opposed to *application-oriented* – origins of the Web. Nearly all the application development capabilities of the Web have been an afterthought, and have emerged as a result of divergent technical needs and business interests instead of careful planning and coordination.

As a result of the browser evolution that has occurred in the past two decades, today's web browsers support a mishmash of complementary, partially overlapping rendering and development models. These include the dominant "holy trinity" of HTML, CSS and JavaScript, and its underlying Document Object Model (DOM) rendering architecture. They also include the *Canvas 2D Context API* as well as *WebGL*. Additionally, there are important 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 mentioned above introduce their own distinct programming models and approaches that the developers (and development frameworks built on top of these base technologies) 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 on top of the underlying development model. The DOM-based approach is by far the most popular and most deeply ingrained, but the other technologies deserve a fair glimpse as well.

Below we will dive more deeply into each technology. We will start with the DOM, Canvas and WebGL models, because these three technologies can be regarded more distinctly as three separate technologies. We will then examine SVG and Web Components, which introduce their own programming models but which are closely coupled with the underlying DOM architecture at the implementation level.

3.1 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. Document Object Model is the foundation for *Dynamic HTML* – the combination of HTML, CSS, and JavaScript – that allows web documents to be created and manipulated using a combination of declarative and imperative 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. Programmatic interfaces are provided for manipulating the contents of the DOM tree from HTML, CSS and JavaScript.

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 adding, removing and modifying its nodes; the browser will then decide how to optimally lay out and render the display after each change.

Over the years, the capabilities of the DOM have evolved significantly. The evolution of the DOM has been described in a number of sources, including Flanagan's JavaScript "bible" (Flanagan, 2011). In this paper we will not go into details, but it is useful to provide a summary since this evolution partially explains why the browser offers such a cornucopia of overlapping functionality.

- *DOM Level 1* specification – published in 1998 – defines the core HTML (and XML) document models. It specifies the basic functionality for document navigation.
- *DOM Level 2* specification – published in 2000 – defines the stylesheet object model, and provides methods for manipulating the style information attached to a document. It also enables

traversals on the document and provides support for XML namespaces. Furthermore, it defines the *event model* for web documents, including the event listener and event flow, capturing, bubbling, and cancellation functionality.

- *DOM Level 3* specification – released as a number of separate documents in 2001-2004 – defines document loading and saving capabilities, as well as provides document validation support. In addition, it also addresses document views and formatting, and specifies the keyboard events and event groups, and how to handle them.
- *DOM Level 4* specification refers to a "living document" that is kept up to date with the latest decisions of the WHATWG/DOM working group (https://dom.spec.whatwg.org/).

DOM attributes can be manipulated from HTML, CSS, JavaScript, and to some extent also XML code. 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 CSS animation capabilities without writing a single line of imperative JavaScript code.

Below is a "classic" DHTML example that defines a text paragraph and an input button in HTML. The input button definition includes an `onclick` event handler function that – when clicked – hides the text paragraph by changing its visibility style attribute to `'hidden'`.

```
<!DOCTYPE html>
<html><body>
<p id="text">This is a piece of text.</p>

<input type="button" value="Hide text"
onclick="document.getElementById('text').style.visibility
    ='hidden'">

</body></html>
```

In practice, very few developers nowadays use the raw, low-level DOM interfaces directly. 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.

3.2 Canvas

The 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 (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 data structure representing 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 code snippet below provides a minimal example of Canvas API usage. In this example, we first instantiate a 2D canvas graphics context of size 100x100 after declaring the corresponding HTML element. We then imperatively draw a full circle with a 40 pixel radius in the middle of the canvas using the Canvas 2D Context JavaScript API.

```
<!DOCTYPE html>
<html><body>

<canvas id="myCanvas" width="100" height="100">
<script>
var c = document.getElementById("myCanvas");
var ctx = c.getContext("2d");
ctx.beginPath();
ctx.arc(50,50,40,0,2*Math.PI);
ctx.stroke();
</script>

</body></html>
```

Note that in these simple examples we are mixing HTML and JavaScript code. In real-world examples, it would be a good practice to keep declarative HTML (and CSS) code and imperative JavaScript code in separate files. We will discuss programming style implications later in Section 4.

The event handling 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.

The Canvas element was was initially introduced by Apple in 2004 for use inside their own Mac OS X WebKit component in order to support applications such as Dashboard widgets in the Safari browser. In 2005, the Canvas element was adopted in version 1.8 of Gecko browsers and Opera in 2006. The Canvas API was later standardized by the Web Hypertext Application Technology Working Group (WHATWG).

The adoption of the Canvas API was hindered by Apple's intellectual property claims over this API. From a technical viewpoint, adoption was also slowed down by the fact that the Canvas API expressiveness is significantly more limited than the well-established, mature immediate-mode graphics APIs that were available in mainstream operating systems already a decade or two earlier. Microsoft's DirectX API – originally introduced in Windows 95 – is a good example of a substantially more comprehensive API.

3.3 WebGL

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. 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-in components. WebGL is integrated into major web browsers, enabling Graphics Processing Unit (GPU) accelerated usage of physics and image processing and effects in web applications. WebGL applications consist of control code written in JavaScript and shader code that is typically executed on a GPU.

WebGL is widely supported in modern desktop browsers. However, its availability, performance and usability is dependent on various factors such as the GPU supporting it.

Just like the Canvas API discussed above, 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. For instance, setting up typical view transformation shaders (e.g., for view frustum), loading scene graphs and 3D objects in the popular industry formats can be very tedious and requires writing a lot of source code.

Given the verbosity of shader definitions, we do not provide any code samples here. However, there are excellent WebGL examples on the Web. For instance, the following link contains a great example of an animated, rotating, textured cube with lighting effects: http://www.sw-engineering-candies.com/snippets/webgl/hello-world/.

Because of the complexity and the low level nature of the raw WebGL APIs, many JavaScript convenience libraries have been built or ported onto WebGL in order to facilitate development. Examples of such libraries include *A-Frame, BabylonJS, three.js, O3D, OSG.JS, CopperLicht* and *GLGE*.

3.4 SVG

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* (W3C, 2011) 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.

The code snippet below provides a simple example of an SVG object definition that renders an automatically scaling, graphical W3C logo to the screen (https://dev.w3.org/SVG/tools/svgweb/samples/svg-files/w3c.svg).

```
<div id="w3clogo">
<svg xmlns='http://www.w3.org/2000/svg' viewBox="0 0 131
    76">
  <path d="M36,5l12,41l12-41h33v41l-13,21c30,10,2,69-21,28
    l7-2c15,27,33,-22,3,-19v-41l2-20h-15l-17,59h-11
    -13-42l-12,42h-11-20-67h91l2,41l8-28l-4-13h9"
    fill='#005A9C'/>
  <path d="M94,53c15,32,30,14,35,71-1-7c-16,26-32,3-34,0
    M122,16c-10-21-34,0-21,30c-5-30,16,-38,23,-21l5
    -10l-2-9"/>
</svg>
</div>
```

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. Just like with 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.

The code sample below serves as an example of interactive SVG that defines a circle object that is capable of changing its size in response to mouse input.

```
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
  "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">

<svg width="6cm" height="5cm" viewBox="0 0 600 500" xmlns
     ="http://www.w3.org/2000/svg" version="1.1">

  <!-- Change the radius with each click -->
  <script type="application/ecmascript">
    function circle_click(evt) {
      var circle = evt.target;
      var currentRadius = circle.getAttribute("r");
      if (currentRadius == 100) {
        circle.setAttribute("r", currentRadius*2);
      } else {
        circle.setAttribute("r", currentRadius*0.5);
      }
    }
  </script>

  <!-- Define circle with onclick event handler -->
  <circle onclick="circle_click(evt)" cx="300" cy="225" r
       ="100" fill="blue"/>
</svg>
```

As illustrated in the example, the SVG scene graph enables event handlers to be associated with objects, so a circle object may respond to an `onClick` event or other events. To get the same functionality with canvas, one would have to implement the code to manually match the coordinates of the mouse click with the coordinates of the drawn circle in order to determine whether it was clicked.

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 (Taivalsaari et al., 2008). 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, considering its tight integration within the DOM and its ability to embed HTML snippets as a foreign Object, 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.

3.5 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 already used by the page 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 (Mikkonen and Taivalsaari, 2008). Over the years various tools and libraries have been invented to 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 parallel "shadow" 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 (Parnas, 1972). 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 the `<style>` attributes. This is how shadow DOM achieves CSS style scoping.

The following listing presents a minimal web component example that creates a text editor that automatically resizes itself as text is entered in the text area. Note that this example only describes the *use* of the component – not its definition.

```
<!DOCTYPE html>
<html><head>
<link rel="import" href="basic-autosize-textarea.html">
</head><body>
<p>Automatically resizing text input component:</p>
<basic-autosize-textarea>Edit me!
</basic-autosize-textarea>
</body></html>
```

Up until recently, many browsers did not support Web Components. Rather, they had to be emulated in the form of *polyfill libraries* that implement the missing functionality (http://webcomponents.org/polyfills/). For latest implementation status, refer to http://caniuse.com/#feat=shadowdom/.

4 COMPARISON

Table 1 provides a comparison of the technologies introduced in the previous section. 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.

In the original version of this paper (Taivalsaari et al., 2017), we discussed the topics presented in Table 1, as well as examined the characteristics and typical use cases of the presented technologies in more depth. Refer to the earlier paper for more details.

Table 1: Comparison of Built-In Client-Side Rendering Technologies.

	DOM / DHTML	**Canvas**	**WebGL**	**SVG**	**Web Components**
Year of Introduction	1998	2004	2011	2001	2011
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	No	No	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	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	No	No	Yes	Yes
Mobile Browser Support	Yes	Yes	Not in Android (add-ons required)	Yes	Not in iOS (polyfill add-ons required)

5 BROADER ARCHITECTURAL DISCUSSION

According to MacLennan's classic software design principles (Schummer et al., 2009), some of the most fundamental principles in engineering aesthetics are *simplicity* and *consistency*: There should be a minimum number of concepts with simple rules for their combination; things that are similar should also look similar, and different things should look different (MacLennan, 1999). Unfortunately, the web browser violates these and several other key principles in a number of ways, as evidenced by the above observations.

Overlapping Capabilities. Ideally, in a software development environment there should be only one, clearly the best and most obvious, way to accomplish each task. However, in web development – even in a generic web browser without add-on components or libraries – there are several overlapping ways to accomplish even the most basic rendering tasks. It is not easy to provide recommendations on specific technologies to use, except for those tasks in which immediate-mode graphics are required (in which case either the Canvas or WebGL API will have to be utilized). In most cases, developers will end up using the basic DOM/DHTML approach, complemented with various libraries.

Mismatching Development Styles. When composing web applications even using the basic

DOM/DHTML approach, the developers commonly face a mixture of declarative and imperative programming styles. Recent trends (e.g., CSS in JS) attempt to reduce everything to JavaScript at the expense of the conciseness of the declarative approaches. They may also have to use a combination of retained and immediate-mode graphics especially when aiming at applications that are usable across different screen sizes – following responsive web design (Marcotte, 2011).

Incompatible and Incoherent Abstractions. The abstractions and programming patterns supported by Canvas and WebGL APIs are very different from DOM/DHTML and SVG programming. Web Components introduce yet another abstraction layer that has been patched on top of the DOM/DHTML. In general, the features supported by the browser reflect organic evolution of features over the years rather than any carefully master-planned architectural design.

Nevertheless, given the organic evolution of the web ecosystem, it is fairly safe to predict that we will not go back to a less diverse web ecosystem or have a chance to radically simplify the feature set of the web browser. For example, the latest versions of the JavaScript language – ECMAScript 6, 7 and 8 – have introduced a lot of new language functionality (promises, generators and decorators, to list a few), thus ensuring that library rewriting and evolution will be swift in the coming years. This will further fuel some of the most problematic characteristics of web development, addressed in the following.

Fashion-driven Development. Over the past years there has been a notable trend in the library area towards fashion-driven development. By this we refer to the developers' tendency to surf on the wave of newest and most dominant "alpha" frameworks. For instance, the once hugely popular *Prototype.js* and *JQuery.js* libraries are nowadays mostly forgotten, replaced by *Knockout.js* and *Backbone.js* in 2012. Back in 2014, *Angular.js* was by far the most dominant alpha framework, while in 2018 it is the *React.js + Redux.js* ecosystem that seems to be capturing the majority of developer attention. As witnessed by the somewhat unfortunate recent evolution of the Angular ecosystem, the alpha frameworks have a tendency to evolve very quickly once they get developers' attention, leading into compatibility issues. To make matters worse, once the next fashionable alpha framework emerges and hordes of developers start jumping ship onto the new one, it becomes questionable to what extent one can build long-lasting business-critical applications and services, e.g., for the medical industry in which products must commonly have a minimum lifetime of twenty years. With the present pace of upgrades, the browser and the web server as the runtime environment would be almost completely replaced by patches, upgrades, and updates; similarly, most of the libraries would be replaced several times by newer, fancier ones.

Opportunistic Design and "Cargo Cult" Programming. In web development there has historically been a strong tradition of *mashup-based development*: searching, selecting, pickling, mashing up and glueing together disparate libraries and pieces of software (Hartmann et al., 2008). Often such development has the characteristics of *cargo cult programming*: ritually including code and program structures that serve no real purpose or that the programmer has chosen to include because hundreds of other developers have done so – without really understanding why. While this approach can save a lot of work and open up interesting opportunities for large-scale code reuse (Salminen and Mikkonen, 2012), this approach does not foster development of reliable, long-lasting applications, because even the smallest changes in the constituent components – each of which evolves separately and independently – can break applications (Salminen et al., 2010).

Violation of Established Software Engineering Principles. Although many web developers may not realize this, the web browser violates many established software engineering principles, including the lack of *information hiding*, lack of *manifest interfaces*, lack of *orthogonality*, and lack of (aforementioned) simplicity and consistency (MacLennan, 1999) (Mikkonen and Taivalsaari, 2008). The absence of these principles is easy to understand given that the web browser was originally designed to be a document distribution environment rather than an application execution environment. However, the current popularity of the Web as a software platform makes it unfortunate that these important principles have been ignored. Currently Web Components are the best – and arguably also the only – chance to reintroduce some of these important principles to the heart of the Web.

In the broader picture, the deficiencies of the web browser as a software platform are being tackled with an abundance of libraries. As of this writing, there are more than 1,300 officially listed JavaScript libraries in javascripting.com, with new ones being introduced at a rapid pace. Although many of the libraries are domain-specific, a lot of them are aimed squarely at solving the architectural limitations of the web browser, e.g., to provide a consistent set of manifest interfaces to perform all the programming tasks. Over the years, JavaScript libraries have evolved from mere

convenience function libraries to full-fledged Model-View-Controller (MVC) frameworks providing extensive UI component sets, application state management, network communication and database interfaces, and so on. In general, these will not help in tackling the above characteristics but rather add a new layer of complexity on top of them.

6 CONCLUSIONS

Web development today presents a cornucopia of choices on many fronts. Both on the client side and the server side, there exist a large number of competing, overlapping technologies, and new libraries and tools become available almost on a daily basis. 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.

In this paper, we have investigated one of the perhaps most overlooked areas in web development: the client-side web rendering architectures that have been built into the web browser. We summarized five built-in rendering and application development models (DOM/DHTML, Canvas, WebGL, SVG, and Web Components), followed by some broader architectural discussion.

REFERENCES

Anttonen, M., Salminen, A., Mikkonen, T., and Taivalsaari, A. (2011). Transforming the Web into a Real Application Platform: New Technologies, Emerging Trends and Missing Pieces. In *Proceedings of the 2011 ACM Symposium on Applied Computing*, pages 800–807. ACM.

Berners-Lee, T. and Fischetti, M. (2000). *Weaving the Web: The Original Design and Ultimate Destiny of the World Wide Web by its Inventor*. HarperInformation.

Bitworking.org (2014). Zero Framework Manifesto: No More JS Frameworks. https://bitworking.org/news/2014/05/zero_framework_manifesto.

Casteleyn, S., Garrigós, I., and Mazón, J.-N. (2014). Ten Years of Rich Internet Applications: A Systematic Mapping Study, and Beyond. *ACM Trans. Web*, 8(3):18:1–18:46.

Charland, A. and Leroux, B. (2011). Mobile Application Development: Web vs. Native. *Communications of the ACM*, 54(5):49–53.

Daniel, F. and Matera, M. (2014). *Mashups: Concepts, Models and Architectures*. Springer.

Flanagan, D. (2011). *JavaScript: The Definitive Guide, 6th Edition*. O'Reilly Media.

Gallidabino, A. and Pautasso, C. (2017). Maturity Model for Liquid Web Architectures. In *17th International Conference on Web Engineering (ICWE 2017)*, volume 10360, pages 206–224, Rome, Italy. Springer.

Hartmann, B., Doorley, S., and Klemmer, S. R. (2008). Hacking, Mashing, Gluing: Understanding Opportunistic Design. *IEEE Pervasive Computing*, 7(3):46–54.

Holdener, A. T. (2008). *Ajax: The Definitive Guide: Interactive Applications for the Web*. O'Reilly Media.

Jadhav, M. A., Sawant, B. R., and Deshmukh, A. (2015). Single Page Application using AngularJS. *International Journal of Computer Science and Information Technologies*, 6(3).

MacLennan, B. J. (1999). *Principles of Programming Languages: Design, Evaluation, and Implementation, 3rd edition*. Oxford University Press.

Marcotte, E. (2011). *Responsive Web Design*. Editions Eyrolles.

Meijer, E. (2007). Democratizing the Cloud. In *Companion Proc. of OOPSLA'07*, pages 858–859.

Mikkonen, T. and Taivalsaari, A. (2008). 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.

Parnas, D. L. (1972). On the Criteria to be Used in Decomposing Systems into Modules. *Communications of the ACM*, 15(12):1053–1058.

Salminen, A. and Mikkonen, T. (2012). Mashups: Software Ecosystems for the Web Era. In *IWSECO@ ICSOB*, pages 18–32.

Salminen, A., Mikkonen, T., Nyrhinen, F., and Taivalsaari, A. (2010). Developing Client-Side Mashups: Experiences, Guidelines and the Road Ahead. In *Proc. 14th Int'l Academic MindTrek Conference: Envisioning Future Media Environments*, pages 161–168. ACM.

Schummer, J., MacLennan, B., and Taylor, N. (2009). Aesthetic Values in Technology and Engineering Design. In *Philosophy of Technology and Engineering Sciences*, Handbook of the Philosophy of Science, pages 1031 – 1068. North-Holland, Amsterdam.

Taivalsaari, A. and Mikkonen, T. (2011). The Web as an Application Platform: The Saga Continues. In *37th EUROMICRO Conference on Software Engineering and Advanced Applications*, pages 170–174. IEEE.

Taivalsaari, A., Mikkonen, T., Ingalls, D., and Palacz, K. (January 2008). Web Browser as an Application Platform: the Lively Kernel Experience, Sun Labs Technical Report TR-2008-175.

Taivalsaari, A., Mikkonen, T., Systä, K., and Pautasso, C. (2017). Comparing the Built-In Application Architecture Models in the Web Browser. In *International Conference on Software Architecture (ICSA)*. IEEE.

W3C (2011). Scalable Vector Graphics (SVG) Specification 1.1 (Second Edition). https://www.w3.org/TR/SVG/.