Joshua Barnett

Registration number 5939968

An Evaluation of HTML5 as a Mobile Gaming Platform

Supervised by Professor Andy Day

University of East Anglia
Faculty of Science
School of Computing Sciences

Abstract

This is my abstract.

Acknowledgements

These are my achknowledgements.

Contents

List of Figures

1 Introduction

For the past five years it has been evident that mobile devices have risen as one of the foremost dominant ways in which the modern world consumes media and entertainment. Alongside this, a new web standard has been developing. *HTML5* (a popularly abused buzzword) is a new language specification with the primary purpose of succeeding where its predecessors failed, in delivering rich interactive content. Generally *"HTML5"* is used as an all encompassing term that includes a host of technologies, as it alone could not deliver such user experiences. Technologies such as *JavaScript* (JS), *Cascasing Style Sheets* (CSS), *Scalable Vector Graphics* (SVG) and *WebGL*. When used together in varying combinations and configurations these form a Web Platform[1] on which to convey content and information in new and interesting ways. However, it is currently the convention to refer to this new platform as *HTML5*.

"Linux" has similarly been distorted as an all encompassing term for Linux based operating systems such as *Debian*, *Fedora*, and *Ubuntu*. However, *"Linux"* alone is just the open-source kernel which lies at the heart of these distributions but on its own would not be of much use.

Original equipment manufacturers (OEMs) such as *Apple*, *LG*, *Nokia* and *Samsung* are becoming progressively competitive in producing better mobile devices for the global market. They offer higher resolution displays, more memory capacity, and increasingly faster processors. This escalation in performance has been a key instigator in the growing support and optimization of *HTML5* and *JavaScript* on mobiles. However, it has only just begun to reach the requirements needed for delivering interactive real-time graphics applications through *HTML5's* features, such as video games.

The purpose of this project is to evaluate the current state of the *HTML5* and its companion technologies on the mobile platform, and to assess whether they provide a suitable means to develop and deploy games. To assist my research I will be designing and creating my own game, expanding my personal understanding of the platform along the way.

[1] *http:// docs.webplatform.org/ wiki/ Main_Page*

2 HTML5

HTML5 is the fifth iteration of the hypertext mark up language. During its previous iterations such as *HTML4.01* there was minimal support for multimedia content. This resulted in a void that was promptly filled by third-party plugins such as *Adobe Flash* and *Microsoft Silverlight*. Such plugins allowed for easy presentation of media such as video, audio and even allowed for interactivity that was used to great effect in games.

Unfortunately many of these plugins are proprietary and closed-source which prevents developers from fixing and debugging inherent issues in the technology. Instead they are dependant on the technology owner for solutions to these problems while being limited to reporting them or finding "temporary" workarounds. Often such plugins also require some form of installation process as they are "add-ons" and exist outside of the *W3C* standard.

Jobs (2010) famously professed his concerns about Flash and presented sound reasoning as to why *Apple* had no intentions of supporting or integrating Flash into their *iOS* devices. Flash was also briefly available for Android but it was short lived and eventually discontinued by *Adobe*. Adobe is continuing to making strides into the mobile platform with their new runtime AIR[2]. *HTML5* is still currently a work in progress but the W3C (2012) have made plans to stabilize the specification and help it reach *Recommendation*[3] status as of this year.

2.1 The Document Object Model (DOM)

The *document object model* (DOM) is the core of what drives modern web applications. It provides a means to manipulate the structure and style of the original document loaded from a web server at run-time on the client side.

However, what makes *HTML* different from most compiled languages such as *C++* and *Java*, is its a interpreted language. Compiled languages are typically converted into a binary format that compresses their instructions into machine-code allowing for

[2] *https://www.adobe.com/aboutadobe/pressroom/pressreleases/201002/021510FlashPlayerMWC.html*

[3] *http://www.w3.org/2005/10/Process-20051014/tr.html#rec-publication*

quicker execution. Interpreted languages in contrast are often read line-by-line with their instructions parsed and executed at run-time.

When HTML is interpreted by a web browsers it is parsed into a DOM. The DOM is essentially a representation in memory of what was originally marked down in the static HTML document. This allows the browsers to interpret relations between elements and use them to render the document appropriately. While in memory the information is not longer static allowing for manipulation by other technologies such as JavaScript.

One example of where this manipulation is used to great effect is the *single-page application* (SPA). *SPAs* have significant advantages over that of traditional websites as they can provide a seamless user experience similar to that of desktop applications. Instead of navigating through links that loading separate *HTML* pages they use states to manage the flow, loading information dynamically when required or requested. This methodology shrinks factors such as load times, enables rich interactions, responsiveness while encouraging reuse. (Takada, 2012)

2.2 Web Browsers

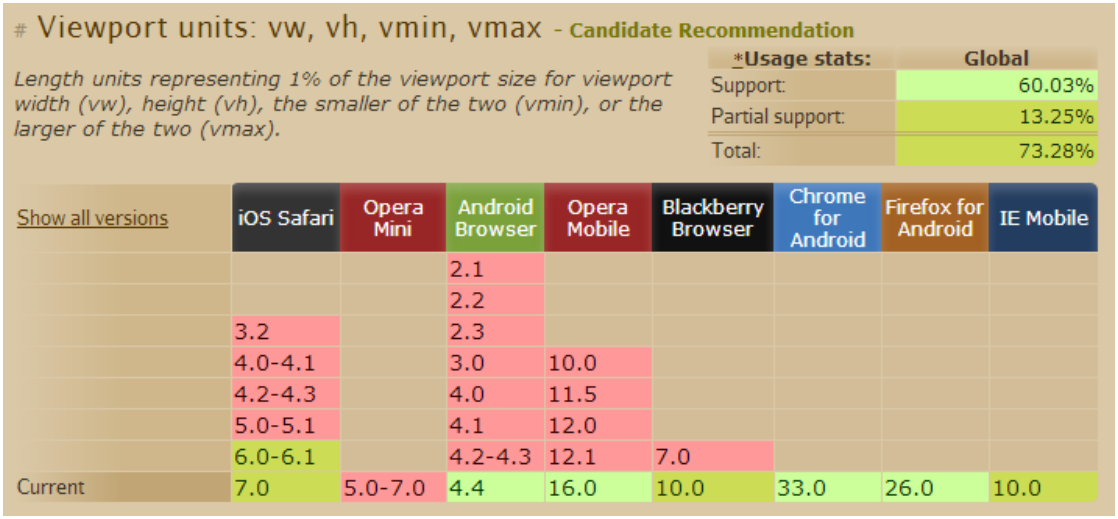
Web browsers are client side applications that render the information requested by the user delivered from a web server. Many of the popular browser vendors such as *Google (Chrome)*, *Apple (Safari)*, and *Mozilla (Firefox)* all have a mobile counterpart. However these mobile counterparts often support a subset of their desktop editions making much of the fringe *HTML5* features even less supported.

2.3 Cross Compatibility

Getting a single web application to render and function consistently across a range of browsers can be a huge undertaking. This task has grown exponentially with the introduction of mobile devices because of the endless combinations of hardware, operating systems, and mobile browser applications. Each combination of these variables has the potential to produce unique bugs and quirks specific to a particular test case. As the specification of *HTML5* was being developed much of the new features have been implemented in browsers at the discretion of the browser vendor. This has quickly led to

cross compatibility issues as older browsers will support less of these new features and not necessarily be consistent with the other browsers that were available at the time.

Depending on the project specification a developer can rapidly become limited by what features of *HTML* they can utilise. For instance one such feature I had planned on using when initially developing my mobile game was *viewport units*. This feature allows for the scaling of elements based upon the current viewport's dimensions which is especially useful on mobile devices for scaling visual elements relative to the screen's aspect ratio. However the stock web browsers on mobile devices have only recently added support for this feature with it being introduced in version 4.4 of *Android* and 6.0 of *iOS*.



Viewport units: vw, vh, vmin, vmax - Candidate Recommendation

Length units representing 1% of the viewport size for viewport width (vw), height (vh), the smaller of the two (vmin), or the larger of the two (vmax).

*Usage stats:	Global
Support:	60.03%
Partial support:	13.25%
Total:	73.28%

Show all versions	iOS Safari	Opera Mini	Android Browser	Opera Mobile	Blackberry Browser	Chrome for Android	Firefox for Android	IE Mobile
			2.1					
			2.2					
	3.2		2.3					
	4.0-4.1		3.0	10.0				
	4.2-4.3		4.0	11.5				
	5.0-5.1		4.1	12.0				
	6.0-6.1		4.2-4.3	12.1	7.0			
Current	7.0	5.0-7.0	4.4	16.0	10.0	33.0	26.0	10.0

Figure 2.1: A compatibility table for viewport units

In cases like this a developer can either limit their target platform range and make use of the fringe features available on mobile devices or use a polyfill also known as a shim. A polyfill is often referred to as code that substitutes the lack of a future *API* by utilising existing features to mimic that of newer fully fledged implementations. Lawson and Sharp (2012) I my case I had to polyfill the lack of *viewport units* in older mobile browsers. After searching for a solution I came across the slides of Kadrmas (2012) presentation that detailed the use of *em* units in *HTML5* games. *em* units were originally intended for scaling document elements relative to the current font size. I used this to

implement consistent scaling of my game across differing screen sizes by adjusting the root document element's font size relative to the screen width of the current device. This approach also made it easy to retain a specified aspect ratio, preventing any oddities when displaying on landscape displays.

However, even when polyfilling potential feature gaps in browsers, issues can still occur under a variety of circumstances. One of the most reliable ways in which to ensure compatibility is to actually possess all the mobile devices you plan to support. This solution most of the time is impractical, especially if the developers fund are limit such as in my case. The next best alternative is to either emulator such devices or use online tools such as BrowserStack[4]. Figure 2.2 shows thumbnails of *MindFlip* on *Android* and *iOS* browsers this is a helpful utility that assists in highlighting obvious incompatibility issues at a glance. The blank thumbnails quickly point out where my game is not loading, which allows me to select and prioritize what devices need debugging. However, with my limited workforce, budget, and time I could really only afford to debug devices I own such as the *Samsung Galaxy S4* and the *Google Nexus 7 (2012)*.

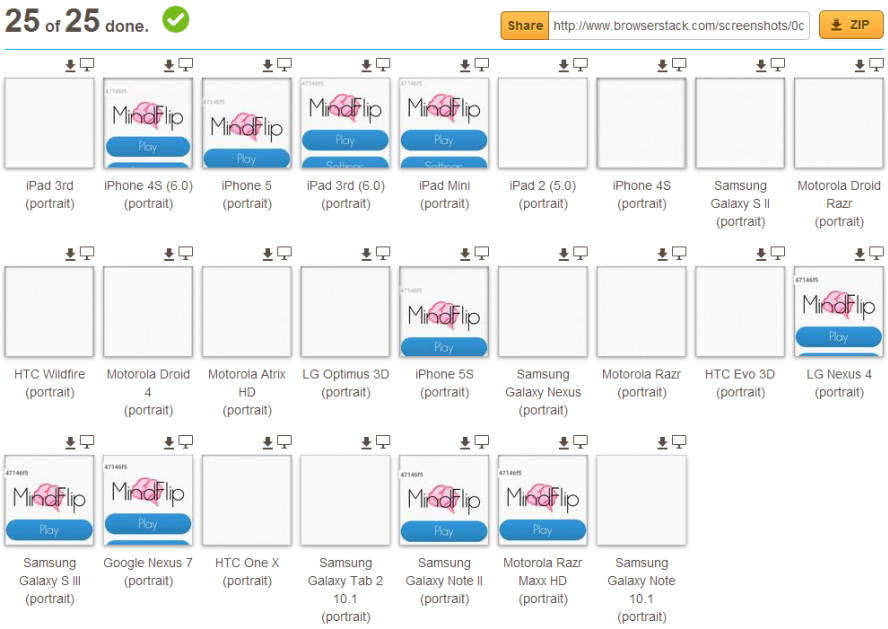


Figure 2.2: Screenshots of MindFlip on various mobile device browsers

[4] *http:// www.browserstack.com/*

3 Graphics

When it comes to rendering in HTML5 are I many different options with varying advantages and disadvantages.

3.1 Cascading Style Sheets (CSS)

The fundamentals of web development are that the content of a website is typically marked down in a `.html` document and the style of this content is embedded alongside or defined within a `.css`. The browser parses the rules found in the stylesheet identifying referenced document elements in the process. This information then instructs the browser how to render the selected document elements often improving their otherwise bland presentation. Although *HTML* has been getting a much needed update, this begs the question of what its companion *CSS* has been doing in order to enhance the web experience.

The latest iteration of *CSS* often referred to as *CSS3* provides new and interesting ways in which to style these elements. Notable features include rounded corners, shadows, gradients , transitions or animations , as well as new layouts like multi-columns , flexible box or grid layouts[5].

The support for these on mobile browsers has been around for quite awhile with much of the older versions supporting them through the use of vendor prefixes. These are a side effect of specification *CSS3* still being a draft as such features were classified as experimental at the time. However, these are starting to be phased out as browser vendors start to agree on specifics and it also provides a means in which to use them providing legacy support for older browsers.

3.2 Canvas

Canvas is probably the most significant feature that was introduced in *HTML5*, especially for game developers. It finally enables developers to do interactive and dynamic graphics natively in the browser, without third-party plugins.

[5] *https://developer.mozilla.org/en-US/docs/Web/CSS/CSS3*

Effectively *Canvas* is a new *HTML* element that can be used to draw graphics via a *JavaScript API*. Although this may seem a simply concept it has huge application opportunities. However, because the draw calls are being made by *JavaScript* they are also limited to the speed of the *JavaScript* virtual machine. Although there has been much development striving to improve the optimization and speed of *JavaScript* virtual machines such as *Google's V8 JavaScript Engine*[6], it only gets slower when it comes to mobile devices.

The performance of mobile devices has been on the increase in recent years with an emphasis on *GPUs* Lin (2014). The application of these faster processors in mobile browsers has only just started to become common place. There are many ongoing software projects with a focus on accelerating the *HTML5* canvas using the mobile device's *GPU*. Unfortunately there has been a lack of cross platform open-source projects that tackle this issue. *Ejecta* is an is one such project but it current only supports *iOS*. The hybrid mobile application framework *Apache Cordova*[7] also provides this feature through a plugin called *FastCanvas*[8].

3.3 WebGL

WebGL is a distillation of the popular *OpenGL* making *GPU* accelerated rendering in the browser now a reality. *WebGL*, a *JavaScript API* is based on the *OpenGL ES (Embedded Systems)* subset, and as the name implies it was designed and optimized for embedded systems such as mobile devices. One streamlined modification *OpenGL ES* made was the removal of the fixed-function *API* introduced in *OpenGL 1.0* enabling the use and compilation of modern shaders.

WebGL has been in development for the past three years which came to maturity as of early last year when the first a stable version was released. As this is a new technology it has only just begun surfacing in the wild, one notable example is the *PlayStation 4's* user interface. In a revealing presentation given by Olmstead (2014) it was declared that the move to *WebGL* was done to ensure cross platform support. This potentially hints at

[6] *https://code.google.com/p/v8/*
[7] https://cordova.apache.org/
[8] *http://plugreg.com/plugin/phonegap/phonegap-plugin-fast-canvas*

Sony's future plans to bring their games to multiple platforms including mobile devices through their *PlayStation Now* streaming service.

However, the support for *WebGL* through mobile browsers is essentially nonexistent to date. One company striving to change that is *Ludei*, who are in the early stages of rolling out their new technology *CocoonJS*. *CocoonJS* is a means of packaging and publishing *HTML5* applications on mobile devices. In amongst this framework is the facilitation of *WebGL* letting it tap into the onboard *GPU* available on most modern mobile devices. Many of the demos they provide demonstrate close to near native performance. This makes *WebGL* a feasible avenue for game development on mobile devices, with the added bonus of being cross platform ready.

4 Audio

a

4.1 `<audio>` Element

a

4.2 Web Audio API

a

4.3 Native Audio

a

5 Game Design

a

5.1 Player Interaction

a

5.2 Gameplay & Mechanics

a

5.3 Feedback

Throughout the development and prototyping of MindFlip it was important to take into consideration the reactions and feedback of players. So whenever the chance arose I would hand over my phone containing my latest stable build of the game and observe others playing it from a third-party perspective.

Preconceptions can be a double edged sword when used in game design. Often they can be used to draw parallels between activities in contemporary games such as my own. This acts as a shortcut when teaching a player new or similar gameplay mechanics. However, these can also be stumbled upon accidently from a developer's perspective as their own preconceptions about games will likely be vast by comparison to the average player.

This was the case with the initial draft of introductory level. In the beginning the first two card symbols the player encountered were a red nought and a blue cross on a three-by-three grid. After showing the game to my sister while providing no tutorial, the first actions she made were to produce a winning *Tic-tac-toe* game state. This made it apparent that there was a flaw in my game design, because the symbols are common to a pre-existing game namely *Tic-tac-toe* the player receives mixed signals about how to play the game from the offset leading to frustration and confusion. I later decided to go for simpler more abstract symbols such as circles, squares, and triangles in my initial set of levels do to their generic associations they imply simplicity leaving the player open to learn the game mechanics.

6 Development Stack

When creating a game targeting HTML5 it can be difficult to know what workflow to invest in. Developers each have their own preferences and beliefs as to what is the best way to work with their priorities often being similar but achieved in different ways.

6.1 Preprocessors & Alternatives

JavaScript in its current state (*ECMAScript 5.1*) is an underwhelming language, especially when compared to more mature languages such as *C++* or *Java*. It lacks native support for object-orientated features such as inheritance, generics, and abstraction. Originally *JavaScript* was never intended for use in large-scale development projects however, due to its inherent portability it has become widespread. Until its apparent deficiencies are rectified and standardised in its next iteration (*ECMAScript 6*), developers have to over utilise its strengths, such as being loosely typed. To help aleve *JavaScript's* growing pains many organisations have been developing alternatives and middle ware to deal with its short comings.

Heavily standardised languages such as *JavaScript* often have very long iteration cycles, making it difficult for them to adapt swiftly however, this does help to ensure and maintain a language's stability in the long term. By comparison *transpilers* (source-to-source compilers) can transform a better suited custom language into plain *JavaScript*, by shimming the missing functionality at compile time. They also make it easier for developers to transition to web development without having to learn the semantics of *JavaScript*. In figure 6.1 are the top thirty-nine most popular programming languages as of January 2014. These statistics were calculated based upon their frequency of occurrence on GitHub (x-axis) and Stack Overflow (y-axis); popular community websites with a focus on programming. By comparing it to one from last year[9] it is clear to see their has been a rise in popularity. Some such preprocessors include CoffeeScript[10],

[9] http://redmonk.com/sogrady/2013/07/25/language-rankings-6-13/
[10] http://coffeescript.org/

Clojure[11], Dart[12] and TypeScript[13]. The full graph[14] is cumbersome so this snippet is only of the top quadrant.

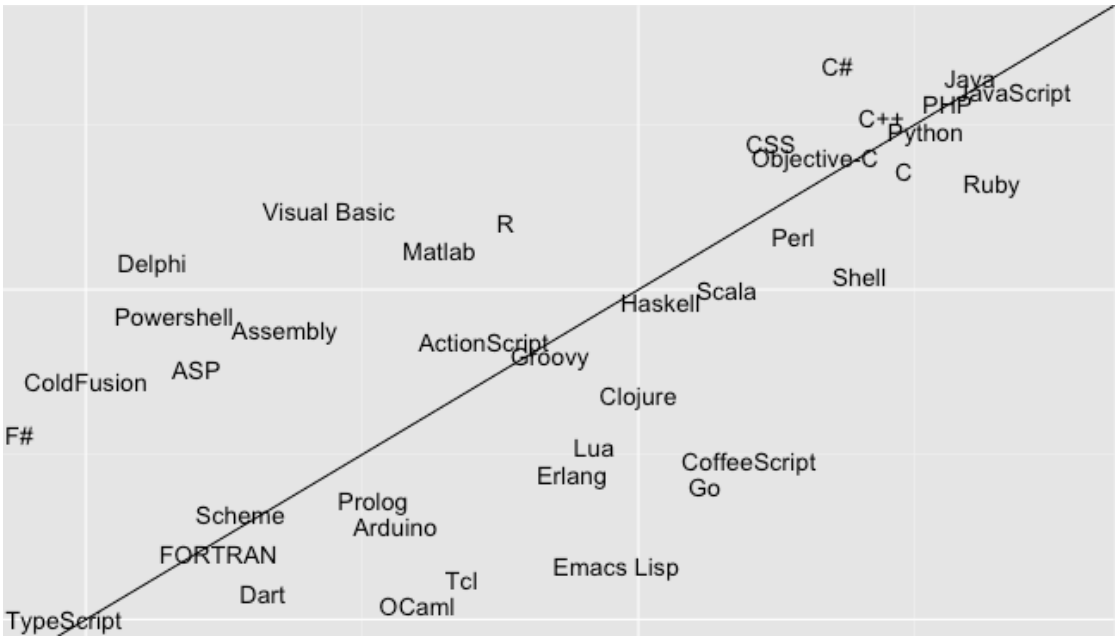


Figure 6.1: The RedMonk Programming Language Rankings: January 2014

The advantages of transpilers often mean that less can be written in order to achieve the same functionality in its targeted language. Simple common activities such as array iteration, class extension, and appending vendor prefixes can become unnecessarily ceremonious especially in the case of *JavaScript* when shimming non-existent functionality. These transpilers can offer an alternative means in which to achieve the same results with less code and effort required on the part of the developer.

6.1.1 Dart

Google's Dart "is a new platform for scalable web app engineering" which comes in the form of several tools. First there is *Dart* the language which is an alternative to writing

[11] http://clojure.org/
[12] https://www.dartlang.org/
[13] http://www.typescriptlang.org/
[14] http://redmonk.com/sogrady/2014/01/22/language-rankings-1-14/

plain *JavaScript* with a host of additional built-in and libraries language features akin to *Java* Fortuna and Cherem (2013). Second there is *DartVM* a virtual machine that comes as a standalone program and also happens to be embedded in the *Google Chrome* browser. The *DartVM* works similarly to a typical *JavaScript* virtual machine with the difference being it interprets `.dart` code instead of *.js* with significant performance advantages Schneider (2013). Lastly there is the *dart2js* compiler, which is effectively a backwards compatibility tool to ensure any application created on the *Dart* platform will still work through conventional means.

6.1.2 TypeScript

Microsoft's TypeScript by its own admission "is a typed superset of JavaScript that compiles to plain JavaScript." which makes it very flexible alternative. This effectively means that you can continue to write plain *JavaScript* and no restrictions will be imposed by the *TypeScript* transpiler. This helps greatly if a developer already has a large *JavaScript* codebase which they wants to utilise during the development of a large scale application. *TypeScript* offers some of the inherently missing language features of *JavaScript* such as type annotations, classes, interfaces and modules. There are also plans to support *ECMAScript 6* features once it becomes the standard. Another positive remark about the output produced by the *TypeScript* transpiler is it produces rather legible *JavaScript* making it easy to see the relationships between the source and target languages.

Because the compile target of these transpilers is *JavaScript* it is possible to take advantage of them when developing mobile web applications such as video games. When I first started developing my game I chose to use *TypeScript* because it was syntaxitcally similar to that of *ActionScript 3*, where my background lies. However, due to the immaturity of the project at the time and a lack of core *JavaScript* knowledge on my part, I felt it was best not to continue using it as I did not have the time to invest in fully learning its ins and outs. I also felt working at this level of abstraction could potentially lead to delays when debugging later in development. However, as of the 2nd of April 2014 close to the time of writing the specification for *TypeScript 1.0* has been finalised and its current status is *stable*. Going forward after this project I am eager to re-evaluate

it now that it has had time to mature.

6.2 JavaScript Modules

JavaScript is typically created as separate `.js` text files that are then referenced in a `.html` document by `<script>` tags. When the web browser finds scripts linked by these tags it will request them from the web server and inject them into the document once loaded. This presents a few problems for large and complex web applications which will often require many scripts each having to be requested and loaded synchronously. Every one of these request has an overhead, which is compounded when a vast quantity are being made. This can lead to suboptimal load times.

To combat this problem a developer could attempt to write all their code in one `.js` file. This file can quickly become colossal making it difficult to manage and refactor, especially on large scale projects. Alternatively the developer can split their code into several `.js` files that will all have to be referenced by a corresponding `<script>` tag. Therefore the more organised the codebase becomes the more requests the browser has to make. The developer also has to micromanage these `<script>` tags to ensure their ordering is correct. The ordering is important because one script could be referencing another that has not yet been loaded, thus causing runtime errors. This approach also presents issues when attempting to introduce software engineering best practices such as unit testing because the interdependencies between these scripts are not clearly defined.

One solution to this common problem is the *module* design pattern. *JavaScript* has an official name *ECMAScript*, and version *5.1* (ECMA-262[15]) is the current standard. However, this present language specification does not support *modules*. The 6th version[16] of the *ECMAScript* specification is currently being drafted with plans to include modules but until then developers must seek other alternatives.

There are a variety of libraries that offer solutions such as *RequireJS*[17], *HeadJS*[18]

[15] *http://www.ecma-international.org/ecma-262/5.1/*
[16] *https://people.mozilla.org/~jorendorff/es6-draft.html*
[17] *http://requirejs.org/*
[18] *http://headjs.com/*

and *yepnope.js*[19]. However, personally I found *Browserify*[20] to be the simplest and the most supported amongst the *Node.js* community. *Browserify* enables developers to write their *JavaScript* code in numerous files and when one file requires another it can be injected via the `require('example');` function. Once a developer has written all their code they can recursively bundle all their modules starting with a root module similar to that of a main class in conventional programming languages. This provides the developer with the best of both scenarios as all their code will be built into a single `.js` file (including their external libraries), while allowing for easy management and refactoring of their core codebase through individual source files.

Most might think that this will in turn cause complications when debugging. As when the code breaks it will do so in the colossal `.js` file that the browser is running. This is where source maps come in. Source maps provide linkage between the original `.js` source files and their bundled counterparts making breakpoints and stepping through code painless. Seddon (2012)

This process can also be made seamless by *Watchify* which automatically recompiles the *Browserify* bundle whenever a module file has been updated. This allows for the developer to work on their original source files and as soon as the file has been saved it have rebuilt the bundle ready to reloaded in the browser for testing.

After the modules have been bundled a further step called minification which involves taking legible *JavaScript* and compressing it down into its most minimal form by removing unnecessary characters such as a whitespace and shrinking instance names. This process can further decrease the `.js` file size making load times marginally faster while also partially obfuscating the original code from prying eyes.

Hanselman (2013) proclaimed that *JavaScript* has become akin to a assembly language for the web. Which is the impression a lot of people will get when taking a peek at the source of popular website such as *Google* or *Facebook*. This is some what understandable given the that *JavaScript* is as low-level as developers can get when programming for the web. However, a happy side effect of being closely tide with the web is it has become now one of the most portable languages to date, because every web

[19] *http:// yepnopejs.com/*
[20] *http:// browserify.org/*

enabled device has a browser, and every browser has a *JavaScript* virtual machine. This makes *JavaScript* a very cross-platform language to develop games with and reaching a wide audience is key when maximising chances for success.

6.3 Model View Controller (MVC)

a

6.4 Templates

a

6.5 Build Pipeline

a

7 Mobile Deployment

One of the challenges that faces mobile developers is how to package their applications for the mobile platform, which spans such a wide variety of hardware and operating systems (iOS[21], Android[22], Windows Phone[23], Ubuntu[24]), Feijoo et al. (2012). This poses a problem especially for the smaller companies as to develop native variants of an application for each platform while retaining a common vision is demanding on both financial and chronological frontiers. Larger companies that facilitate and accommodate such resources, do however frequently benefit from having a superior overall user experience. Most native applications built with their corresponding software development kits have direct access to device specific application programming intefaces (APIs). This often results in native applications having faster performance and better system user interface integration. Native applications compile to binary machine code

[21] *http://www.apple.com/ios/*
[22] *http://www.android.com/*
[23] *http://www.windowsphone.com/*
[24] *http://www.ubuntu.com/phone*

that is then interpreted via its relative hardware architecture making it vastly faster than JavaScript which is interpreted at run-time via a browser or web-view. This is most apparent when it comes to developing 3D gaming experiences as these can often be most the most performance taxing, Kulloli et al. (2013).

When comparing HTML5 based mobile applications to native there was another significant disadvantage as there was no way to access device APIs. Such APIs are required to make use of the most

hardware features on mobile devices such as the camera, device orientation and battery status. Especially considering that each API varies dependant on operating system and then again on hardware specification. For instance not all mobiles have front facing cameras or high-dpi (dots per inch) screens but a vast majority do. So it becomes quite a challenge when trying to build an application that adapts from phone-to-phone without excluding a particular sector of an audience from the core user experience, Charland and Leroux (2011).

A project with the sole purpose of remedying such issues is that of PhoneGap a mobile development framework originally created by a company called Nitboi. After PhoneGap began to make strides the company was purchased and enveloped by Adobe in 2011, Adobe (2011). Then after the project matured under Adobe's supervision the project was donated to the Apache Software Foundation. This was to ensure that the project was properly maintained when made open source under the Apache License Version 2.0[25]. A formality due to this hand over was that the open source variety of the project had to operate under a different name to avoid trademark ambiguity, Leroux (2012). Adobe's PhoneGap still functions as a separate entity as a distribution of Apache Cordova offering services such as cloud compilation[26].

Apache Cordova[27] is a platform for building native mobile applications using HTML, CSS and JavaScript. It achieves this by providing developers with a set of relative device APIs that allow for access of functions previous obfuscated from web technologies by the native system. It also assists with handling a lot of the cross-platform concerns previously mention when targeting multiple platforms and device hardware.

[25] *http://www.apache.org/licenses/LICENSE-2.0.html*
[26] *https://build.phonegap.com/*
[27] *http://cordova.apache.org/*

7.1 Packaging

a

7.2 Debugging

a

7.3 Profiling

a

References

Adobe (2011). Adobe Announces Agreement to Acquire Nitobi, Creator of PhoneGap. *Adobe Press Releases*. *http://www.adobe.com/aboutadobe/pressroom/pressreleases/201110/AdobeAcquiresNitobi.html*.

Charland, A. and Leroux, B. (2011). Mobile Application Development: Web vs. Native. *Communications of the ACM, Volume 54 Issue 5, May 2011*. *http://delivery.acm.org/10.1145/1970000/1968203/p20-charland.pdf*.

Feijoo, C., Gómez-Barroso, J.-L., Aguadoc, J.-M., and Ramos, S. (2012). Mobile gaming: Industry challenges and policy implications. *Telecommunications Policy*.

Fortuna, E. and Cherem, S. (2013). Dart: HTML of the Future, Today! *Google I/O 2013*. *http://youtu.be/euCNWhs7ivQ*.

Hanselman, S. (2013). JavaScript is Web Assembly Language and that's OK. *Scott Hanselman's Blog*. *http://www.hanselman.com/blog/JavaScriptIsWebAssemblyLanguageAndThatsOK.aspx*.

Jobs, S. (2010). Thoughts on Flash. *Apple*. *http://www.apple.com/hotnews/thoughts-on-flash/*.

Kadrmas, J. (2012). Building HTML5 Games. *cf.Objective()*. *http://dl.dropboxusercontent.com/u/21521496/cf.objective/index.html*.

Kulloli, V. C., Pohare, A., Raskar, S., Bhattacharyya, T., and Bhure, S. (2013). Cross Platform Mobile Application Development. *International Journal of Computer Trands and Technology (IJCTT), Volume 4, Issue 5, May 2013*.

Lawson, B. and Sharp, R. (2012). Introducing HTML5, Second Edition. *New Riders*.

Leroux, B. (2012). PhoneGap, Cordova, and what's in a name? *PhoneGap Blog*. *http://phonegap.com/2012/03/19/phonegap-cordova-and-what%E2%80%99s-in-a-name/*.

Lin, E. (2014). 2014 global mobile application processor market forecast. *DIGITIMES Research*. *http://www.digitimes.com/news/a20140129RS400.html*.

Olmstead, D. (2014). Optimizing WebGL Applications. *Google Developers*. *http://youtu.be/QVvHtWePQdA*.

Schneider, D. F. (2013). Compiling Dart to Efficient Machine Code. *Lecture at ETH*. *https://www.dartlang.org/slides/2013/04/compiling-dart-to-efficient-machine-code.pdf*.

Seddon, R. (2012). Introduction to JavaScript Source Maps. *HTML5 Rocks*. *http://www.html5rocks.com/en/tutorials/developertools/sourcemaps/*.

Takada, M. (2012). Adventures in Single Page Applications. *HTML5 Developer Conference*. *http://youtu.be/BqDJqKGfIiE*.

W3C (2012). Plan 2014. *World Wide Web Consortium (W3C)*. *http://dev.w3.org/html5/decision-policy/html5-2014-plan.html*.