University of Derby

Department of Electronics, Computing & Mathematics

A project completed as part of the requirements for

BSc (Hons) Computer Games Programming

entitled:

Is WebAssembly the Future Development Target for Web Applications?

By

Kieran Warren

May 2018

# Abstract

The purpose of this study is to prove or disprove whether WebAssembly offers significant performance benefits over previous web technologies such as JavaScript and ASM.js. The study concentrates on heavy workloads with the aid of Unreal Engine and performance of demos are recorded and compared to find performance differences between the different technologies. Some parts of the study are dedicated to smaller components rather than entire platforms and whether small changes can have a significant performance benefit.

# Acknowledgements

I would like to thank Mr. Carlton McDonald for guidance and support for the entirely of this project.

# Table of Contents

[Abstract 1](#_Toc513641909)

[Acknowledgements 2](#_Toc513641910)

[1 Table of Contents 3](#_Toc513641911)

[2 Introduction 6](#_Toc513641912)

[2.1.1 Project Rationale 6](#_Toc513641913)

[2.2 Project Aims and Objectives 6](#_Toc513641914)

[2.2.1 Hypothesis 7](#_Toc513641915)

[3 Literature Review 8](#_Toc513641916)

[3.1 JavaScript – The Language of the Web 8](#_Toc513641917)

[3.2 Native Plugins – Better performance and capabilities at the cost of security 9](#_Toc513641918)

[3.3 ASM.js – Safer and Faster 10](#_Toc513641919)

[3.4 WebAssembly 12](#_Toc513641920)

[3.5 Previous Study Analysis 16](#_Toc513641921)

[3.5.1 Benchmark 1 16](#_Toc513641922)

[3.5.2 Benchmark 2 17](#_Toc513641923)

[3.5.3 Benchmark 3 - Compiled size 19](#_Toc513641924)

[3.5.4 Benchmark 4 – WebAssembly call overhead 20](#_Toc513641925)

[3.6 Conclusion 21](#_Toc513641926)

[4 Methodology 22](#_Toc513641927)

[4.1 Introduction 22](#_Toc513641928)

[4.2 Development 22](#_Toc513641929)

[4.3 Testing 23](#_Toc513641930)

[4.4 Creating a Fair Test 23](#_Toc513641931)

[4.4.1 Configuration 1: 24](#_Toc513641932)

[4.4.2 Configuration 2: 24](#_Toc513641933)

[4.4.3 Configuration 3: 24](#_Toc513641934)

[4.5 Test configurations 25](#_Toc513641935)

[4.6 Conclusion 25](#_Toc513641936)

[5 Findings and Analysis 27](#_Toc513641937)

[5.1 Introduction 27](#_Toc513641938)

[5.2 Unreal Engine Benchmarks Results 27](#_Toc513641939)

[5.2.1 Introduction 27](#_Toc513641940)

[5.2.2 Multi-threading and SIMD 28](#_Toc513641941)

[5.2.3 Emscripten Optimization Flags 31](#_Toc513641942)

[5.2.4 Browser Performance Comparison 35](#_Toc513641943)

[5.2.5 WebAssembly vs ASM.js Performance 38](#_Toc513641944)

[5.2.6 Compilation Size and Times 44](#_Toc513641945)

[5.2.7 Development Issues 50](#_Toc513641946)

[5.3 Conclusion 53](#_Toc513641947)

[6 Conclusion & Recommendations 55](#_Toc513641948)

[6.1 Conclusion 55](#_Toc513641949)

[6.1.1 Conclusion 1 55](#_Toc513641950)

[6.1.2 Conclusion 2 55](#_Toc513641951)

[6.2 Aims and Objectives 55](#_Toc513641952)

[6.3 Recommendations 56](#_Toc513641953)

[6.3.1 Research Recommendations 56](#_Toc513641954)

[6.3.2 Developer Recommendations 57](#_Toc513641955)

[6.3.3 Future Work 57](#_Toc513641956)

[7 Bibliography 59](#_Toc513641957)

[8 Figures 64](#_Toc513641958)

[9 Appendices 66](#_Toc513641959)

[9.1 General Links 66](#_Toc513641960)

[9.2 Project 1 Screenshots 66](#_Toc513641961)

[9.3 Project 2 Screenshots 68](#_Toc513641962)

[9.4 Framerate recording script 69](#_Toc513641963)

[9.5 Data Sheets 69](#_Toc513641964)

# Introduction

### Project Rationale

The web is a constantly growing platform and the technologies behind it are always evolving. The primary programming language used on the web, JavaScript, has evolved from basic scripting to an advanced language capable of creating many amazing applications. However, the issue seen by many since its inception has been its performance. The primary issue with JavaScript is the lack of compilation due to it being an interpreted language (Clark, 2017). Interpreted languages tend to have significant overhead, compiling instructions in real time. Many languages like Java compile into a portable byte code. JavaScript is instead transferred in an inefficient text format increasing translation times.

Over time JavaScript and its supporting technologies have evolved to greatly increase performance, mainly with the introduction of JIT compilers. It is often however still insufficient for more advanced applications such as games. Because of the poor performance (as well as a lack of features available using JavaScript), native plugins such as Flash and ActiveX were introduced, in an attempt to offer better performance and more advanced features within a web browser. Running native code in a web browser caused a variety of compatibility issues and security became a great concern. Native plugins have since largely been abandoned. With the constant demand for performance, ASM.js was a poorly designed replacement. ASM.js is a highly optimized subset of JavaScript, however, is not treated as optimizable by all browsers and is still JavaScript and comes with many of its flaws.

These issues outlined are where WebAssembly is intended to come in. A highly portable, bytecode supposedly capable of much increased performance compared to JavaScript and other alternatives. The following research is focusing on whether WebAssembly is truly the solution that it is often claimed to be as well as research into surrounding areas in improving web page performance.

## Project Aims and Objectives

The aim of this study is to prove or disprove the claims of WebAssembly offering significant performance improvements by the WebAssembly team and its supporters as well as investigate other potential solutions in optimizing the performance of a web page or application.

The following objectives have been produced to try and achieve this aim;

1. Conduct a literature review. Primarily into different targets used to execute code on the web, with a particular focus on performance. This will help to understand the current state of running advanced applications on the web.
   1. This objective was successfully achieved and aided in narrowing down the development and testing process as some targets could be removed entirely.
2. Develop projects that can be exported to a number of the different platforms identified in the Literature Review and contains a way of recording the performance of the project.
   1. This objective was only partially achieved. Due to limitations/issues with Unreal Engine, recording of data was limited and had to be manually recorded. If these issues are resolved in the future, then reconducting these tests in the same manner with an improved recording technique may produce different results.
3. Export the project into numerous web-based configurations.
   1. This was successfully achieved, with a large variety of different configurations that varied based on export platform, Emscripten settings and additional features such as SIMD and multi-threading.
4. Record performance of all versions of the project numerous times across different browsers and hardware to produce reasonable results.
   1. This objective was successfully achieved except in a few cases. Performance in a few of the project versions could not be recorded.
5. Analyse the results from objective IV to determine what the best solution might be for future projects and existing projects and identify any patterns and determine reasons for those patterns and results.
   1. This was successfully achieved, and a number of different conclusions were discussed in the analysis. Some recommendations can come from the results which are discussed in section 6.3.

### Hypothesis

The primary hypothesis is;

*WebAssembly can provide a great improvement in performance for advanced workloads on the web.*

This hypothesis will be tested through performance tests that contain a specific heavy workload that run on the discussed platforms such as WebAssembly and JavaScript.

The second hypothesis is;

*WebAssembly produces significantly more compact files than JavaScript/ASM.js*

This hypothesis is going to be tested measuring the outputted file sizes from the benchmarks used to test the primary hypothesis.

# Literature Review

This section is a review of the current and upcoming technologies used to power the web. Specifically looking into the performance aspects of each. It will focus greatly on JavaScript, how it went from a basic and very slow interpreted language into something much more optimal with new browser processing methods. We will also focus on how to improve performance further, moving away from standard/vanilla JavaScript into performance optimized platforms such as ASM.js and WebAssembly (Wasm/WASM) and how these may be effectively utilized on the web.

## JavaScript – The Language of the Web

The origins of JavaScript may be considered very different to how the language is utilized today. Announced in 1995 by NetScape (Netscape Communications Corporation, Sun Microsystems Inc., 1995), the language was designed to be complimentary to HTML, the primary mark-up language of the web. JavaScript started as an interpreted language, with supported browsers using only a simple interpreter (Clark, 2017). Because of poor performance, its applications were often limited to small tasks.

With increased demand for larger applications running inside of a browser, improvements needed to be made to the execution performance of JavaScript. The biggest step to improving performance was the introduction of Just-In-Time (JIT) compilers. The first JIT compiler for JavaScript was Mozilla’s TraceMonkey (Dalziel, 2013), which was first made publicly available in Firefox 3.5 in 2009 (Shankland, 2009) and followed quickly by the introduction of Google’s Chrome V8 engine, a JavaScript engine tuned for performance (Dalziel, 2013). There is no denial that the introduction of JIT compilers changed the usability of JavaScript on the web, with Firefox’s TraceMonkey offering 20-40x the performance of the previous version of Firefox (Paul, 2008).

The aim of a JIT compiler is to produce machine code, or at minimum an intermediate code as early as possible to avoid interpreting each instruction. The V8 JIT compiler compiles the entire page into machine code on page load (Laurens, 2013). Other implementations of a JIT compiler may compile to an intermediate language maintaining portability across different processor architectures while still offering a huge performance benefit over simple interpretation. This method would not be feasible with JavaScript, firstly because JavaScript in text form is already the language of the web and due to the generated intermediate code varying between browsers because of different compiler techniques.

The result of a JIT compiler may be that execution time is greatly reduced, however can result in a larger overhead on the first load of a page when the JavaScript is compiled. This initial optimization pass is performed by a basic compiler, designed to keep overhead short. To improve performance further, most JIT compilers included a secondary level of optimization. One of the better examples of this is Microsoft’s ChakraCore JavaScript engine. It runs a basic and advanced compiler on an additional thread. This architecture works alongside a profiler on the main processing thread (Zhu, 2017) to optimize specific sections of code.

The profiler watches and stores information on code execution. In cases where functions or pieces of code are either performing poorly or used regularly, the profiler may spawn a background thread to re-compile the selected code with a much deeper level of optimization. This is an important part of JavaScript performance that and emphasis will be put into the differences between basic compiled JavaScript code and ‘warm code’ during testing, code recompiled after regular use (Shan, 2017).

I hope to look deeper into the different methods implemented in the various browsers and how ‘warm’ and ‘cold’ JavaScript code can differ between them. If developers can effectively code to take advantage of this feature in JIT compilers, then is there actually any need for other technologies such as WebAssembly to even exist?

## Native Plugins – Better performance and capabilities at the cost of security

Due to early versions of JavaScript lacking several features and browsers still using interpreters, native plugin support was implemented in browsers to allow access to more OS features and more performant code. One of the first available native plugins was ActiveX, announced in 1996 by Microsoft (Microsoft Corp, 1996). It launched with the intention of providing access to a wider range of features than those currently available through the web including graphics, secure transactions and video playback (Microsoft Corp, 1996). The ActiveX architecture is based on the Component Object Model (COM) which are used extensively through Windows (Carnegie Mellon University, 2000).

The extensive list of available features made it a popular choice for larger applications, particularly compared to JavaScript. However, ActiveX, like other native plugins had one major flaw, security. Being native binary code, ActiveX and other native plugins were not designed to run in a “sandbox” (Carnegie Mellon University, 2000), a mechanism designed to monitor untrusted application and limit potentially harmful system calls (Goldberg, 1996). Security is therefore in the hands of the user to trust the application author.

Because of these security issues, native plugins such as ActiveX and Flash based on NPAPI (NetScape Plugin Application Programming Interface) have been almost entirely phased out in the latest versions of major browsers. ActiveX is still supported in Internet Explorer; however, its capabilities are heavily restricted. In Internet Explorer 10, Microsoft introduced the Enhanced Protected Mode, restricting execution only to trusted applications inside of an AppContainer that creates a sandbox and restricts application access to system calls (Microsoft, n.d.).

Given that these plugins are given permission to run at the same permission level as the browser, executing instructions directly on the CPU (Bhatia, 2014). It is no surprise that performance should be high. The implementations give plugins un-restricted access to the machine and can be left unsupervised. However, the increase in performance did not make up for the major security flaws in this model.

Due to almost all native plugin support being removed from major browsers it would be an incredibly poor choice to develop using any of these native plugins. So, there is not enough reason to include them during my performance testing. However, they are still an important step in the history of advanced applications on the web and paved the way for better options.

## ASM.js – Safer and Faster

ASM.js is considered the next step in creating high performance code ready to run on the web. Unlike native plugins, ASM.js has the advantage of still being JavaScript, and so it is capable of running in any modern web browser. First appearing in 2013 in the Firefox browser, its primary purpose was to provide performance much closer to native (Wagner, 2013), though it will never reach native execution performance due to the additional steps required. By 2013, JavaScript’s capabilities had increased dramatically. It had become capable of far more advanced tasks making native plugins largely redundant.

Before the release of ASM.js, HTML5 was in active development by the World Wide Web Consortium (W3C, 2012). With it came a slew of new features and JavaScript based APIs such as a 2D drawing canvas, Web Workers for multi-threading and the ability for web apps to be stored and run offline (Walker, 2011). These are just a few of the biggest contributions to HTML5’s success. The new APIs allowed functionality far more advanced than was previously possible almost entirely removing any need for native plugins such as Flash. HTML5 is a major contributor to the death of native plugins and since the HTML5 standards have been implemented across all major browsers, native plugin developers such as Adobe have confirmed plans to end development on native plugins such as Flash in the near future (Warren, 2017). Flash was previously a popular choice for browser games, but the HTML5 Canvas is a direct replacement for it.

ASM.js is often defined as a small subset of JavaScript that is heavily optimized which can be used as a compiler target for lower level languages like C/C++ (Zakai & Nyman, 2013). Through continuous optimizations of the small subset since it is the Mozilla team achieved performance only 50% slower than native by 2015 (Zakai & Wagner, 2015). An impressive feat considering ASM.js is still JavaScript, an interpreted language and is an example of how far JavaScript development has come since it first appeared in 1995 not just in features but in performance too.

With ASM.js still being JavaScript, it maintains a higher level of compatibility with browsers and web pages. No additional setup is required and high performant ASM.js scripts can replace vanilla JavaScript ones. Since ASM.js is a compiler target, frameworks and even game engines such as Unity and Unreal Engine can be ported directly to JavaScript and used directly in plain JavaScript with great performance benefits, Emscripten’s zlib and fasta benchmarks receiving 13% and 24% improvements respectively (Zakai & Wagner, 2015).

ASM.js is currently the most compatible option when it comes to getting good performing code on the web, almost all major browsers support the highly optimized subset, and those that do not offer improved performance remain compatible, due to it still being JavaScript. Like an assembly language, ASM.js has a very small number of ‘instructions’ each of which can be almost directly translated into a single machine instruction without large amounts of additional work, compared to traditional JavaScript. This small number of instructions is limited to only math operations on numeric data, again this bears great resemblance to a traditional assembly language.

Because ASM.js is still JavaScript, it cannot technically make use of statically typed data, and so some tricks are required to force variables into data types. For example, in Figure 1 below, bitwise OR with 0 or ‘|0’ is used to specify a variable of integer type while ‘|0.0’ can force type to be floating point. Instead of using traditional arrays, ASM.js makes use of a virtual heap, which may be viewed as a typed array, while still being just a heap, which can offer considerable performance benefits over a traditional JavaScript array, which is not necessarily a contiguous block of memory like an array in C/C++ (Rauschmayer, 2014), that can support any kind of object within a single array. The heap array can be viewed in a number of different ways, reading as little as int8 values or as large as float64 (or double) values (Herman, et al., 2014). The result of this is highly optimized and code can be converted directly into assembly, bypassing the JavaScript interpreter or JIT compiler.



Figure 1 - Sample ASM.js (Herman, et al., 2014)

Due to its superior browser compatibility compared to WebAssembly at this stage, it is important to include it during testing as it may come out as the superior choice at this time factoring in both performance and compatibility. ASM.js should be guaranteed to be compatible for as long as JavaScript is used on the web, which is no doubt going to continue as the language of the web. WebAssembly however, still has the potential to fail and the project cancelled/abandoned in the future as it is not relied upon to the same level.

## WebAssembly

The primary focus of this study is to find the suitability of WebAssembly for day-to-day web development and whether claims of much improved performance are true and if they are make up for possible additional development time or difficulties over developing just for JavaScript. This study will explore its usability in more advanced applications such as 3D games and whether its performance results offer a considerable enough improvement to warrant using it.

WebAssembly is the newest development in the world of web performance. It was released in only 2017 (WebAssembly, 2017) and is still in early stages of support. WebAssembly has several important design goals that aim to achieve what its predecessors such as ASM.js could not. To achieve the design goals of being fast, safe and portable, several key requirements have been specified (WebAssembly Community Group, 2018):

* Near native performance by using features available on all current hardware.
* Safe code executed in a memory-safe sandboxed environment.
* A set of well-defined behaviours.
* Portable between any modern processor architectures and hardware.
* Can be compiled from any language, allowing better access to the web for almost any developer.
* Not restricted to specifically the web, capable of running in almost any VM.
* An open platform that can integrate easily with any environment.

The design goals are very demanding but many of these have already been met with the Minimum Viable Product (MVP). The attempt to keep the platform open and independent of any architecture or programming language could lead to it becoming a compilation target for more than just web pages.

Games, applications and even development libraries written in almost any language could in future be compiled into WebAssembly and have almost guaranteed compatibility on almost any platform from a low-end smartphone to a high-end desktop either through the web or dedicated application. The closest attempts to this have been through isomorphic (can be executed on either server or client) JavaScript through projects such as the Meteor framework (Brehm, 2013). The author feels that these implementations are rather restricted by the limited performance of JavaScript. WebAssembly being compiled into a binary format rather than interpreted from a human readable format has the potential to make cross-platform compatibility a reality for more demanding applications.

While WebAssembly has reached its MVP, it still has many important features that may affect performance and ease of use considerably in the future. These new features would also aid in fully reaching the initial design goals of WebAssembly.

The feature that is likely the most crucial to reaching maximum performance on the web are multi-threading, which would allow WebAssembly modules to execute in parallel on the multi-core processors inside most PCs and phones today. To preserve portability, threading is most likely to be implemented using POSIX threads often referred to as pthreads (Bastien, 2017). Pthreads are the standardized C language threads which should therefore already be heavily supported on most systems. Its portability means a relatively simple API with only a few simple routines (Barney, 2017). With multi-threaded processors having become standard, there is no denial that designing a program to run across multiple threads can have a great performance benefit. It is disappointing that this will not be testable with WebAssembly. Some multi-threading is available through JavaScript using Web Workers, but their implementation may be considered very limited with only basic control and communication between threads (Danford, 2018). It could still be beneficial to test the performance benefits of Web Workers in vanilla JavaScript and ASM.js to see what potential improvements in a web environment may be possible when threading is introduced into WebAssembly.

SIMD (Single Instruction Multiple Data) is another important planned feature and another method of parallel processing. However, SIMD allows parallelizing within a single core instead of multiple (Landwerth, 2014). SIMD allows a single calculation to be run on multiple pieces of data within a single instruction. This could be particularly useful in games where for example, you may wish to rotate all objects within the game world by the same amount or move many objects a certain distance at the same time. Testing completed by Microsoft found a 400-500% performance improvement using AVX (Advanced Vector Extensions), the most modern form of SIMD found in high-end AMD and Intel processors. In its greatest form can process a 512-bit register, which can contain sixteen 32-bit integers or single precision floating point or eight 64-bit integers or double precision floating point values in a single assembly instruction (Reinders, 2013). Other processors may instead use AVX-256 with half the maximum throughput of AVX-512.

Other planned features for WebAssembly are Exception Handling, Garbage Collection, Bulk Memory Operations, Web Content Security Policy and ECMAScript Module Integration. These features are mostly for improved ease of use. Exception handling and Garbage collection could lead to support WebAssembly becoming a compilation target for higher level, garbage collected languages such as C# or Java. Most other planned features are quality of life improvements such as improved integration with JavaScript and better security. Other proposed features include larger memory access being the 32-bit 4GB limit, improved OS functionality and further improvements to existing functionality. But any development should not rely on these proposed features being supported in WebAssembly any time soon. (WebAssembly, 2017).

Unlike ASM.js, which was initially developed internally at Mozilla, and native plugins that have been individually developed by a number of different companies such as Google, Adobe and Macromedia, WebAssembly is a cross-browser project with support from all major browser vendors (Wagner, 2017). This is a great positive, as future compatibility should be an important consideration for any new development project and the backing of all major vendors means it is likely to receive good support across the major browsers.

The fundamental difference between WebAssembly and JavaScript is in how they are executed. While JavaScript is primarily a high-level interpreted language (though now used primarily through a JIT compiler), WebAssembly is low-level, retaining cross-platform portability (Google GmbH; Microsoft Inc. USA; Mozilla Inc. USA; Apple Inc, USA, 2017). WebAssembly is designed to be a compiler target for popular languages like C/C++ which can then perform heavy optimizations and does not need hand-holding whilst executing like an interpreted language, removing a great amount of overhead from execution (Google GmbH; Microsoft Inc. USA; Mozilla Inc. USA; Apple Inc, USA, 2017).

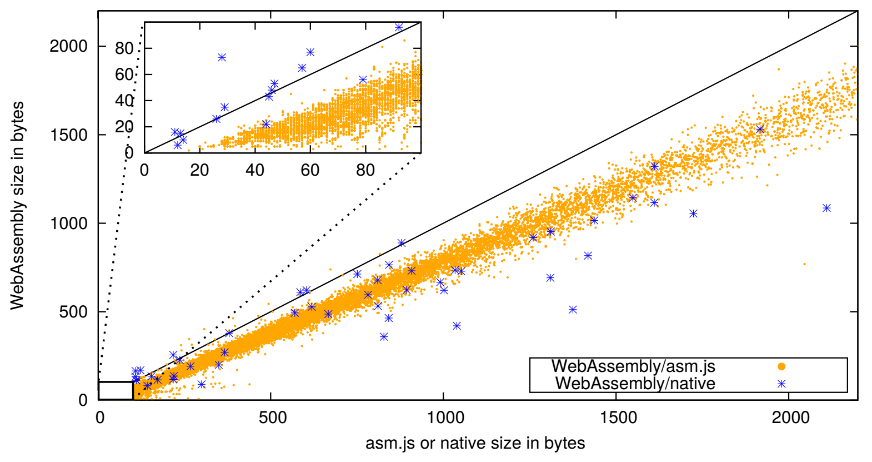
With average script size of a page continuously increasing, average size almost quadrupling from 113kb of JavaScript to 408kb of JavaScript between 2010 and 2016 (Google Chrome Developers, 2016), it is important to factor in the overall size of the web site you are sending to the user. This is an area where WebAssembly can improve over even minified JavaScript/ASM.js. As can be seen in Figure 2, identical code compiled to both WebAssembly and ASM.js is in most cases notably smaller. Testing from (Google GmbH; Microsoft Inc. USA; Mozilla Inc. USA; Apple Inc, USA, 2017) found WebAssembly to be on average 62.5% the size of an identical ASM.js module.

Figure - Size comparison between compiled ASM.js and WebAssembly from; (Google GmbH; Microsoft Inc. USA; Mozilla Inc. USA; Apple Inc, USA, 2017)

There is a lot of differences between WebAssembly and either ASM.js or vanilla JavaScript, and it is important to attempt to test all of them. The most important part to test is the raw execution performance between the three. The second most important part to test is the file sizes of the produced code. One important thing to note here is that all JavaScript code should go through the process of minification.

The process of minification is to reduce code file sizes and is generally exclusive to JavaScript due to it being commonly transferred over a network. Simple minification is the removal of unnecessary characters such as whitespace, comments and long variable/function names. More advanced methods of minifying may include more advanced source mapping, however most improvements come from removing unnecessary characters. UglifyJS2 reduced the size of popular JavaScript library, JQuery by 65% (Bazon, 2012). Gzip, a standardized compressed file format is also usually applied on top of this and the same test reduced file size by about two-thirds on top of minification.

The process of minification can relate quite heavily to obfuscation, a technique common to platform independent bytecode compiled languages such as Java designed to make de-compilation more difficult and harder to read by humans, while maintaining the same instructions (Low, 2005).

As a result of its aim to be compact, compiled WebAssembly code is difficult to read for any human, the output is essentially a set of instructions like a traditional assembly language. To improve debugging of WebAssembly, it may also be compiled to WebAssembly Text (Wast). It is made up of symbolic expressions (S-expressions) whose original purpose was to represent complex data structures, created by the Network Working Group (Rivest, 1997). Rather than just representing a data structure, The WebAssembly team’s interpretation of S-Expressions represents a tree of nodes containing mostly instructions (Wagner, 2018). To keep transferred file sizes small, Wast can either be created from the original program source using Emscripten (Zakai, 2018) or generated in the browser and in future will allow for basic debugging of WebAssembly such as breakpoints and code stepping (Delendik, 2018).

Other less major factors are development time, something which would be difficult to measure in my testing and a consideration for future developments on the respective platforms. JavaScript and WebAssembly are in active development while ASM.js is almost inevitably going to be replaced by WebAssembly soon.

## Previous Study Analysis

The biggest reason for focusing on WebAssembly is due to it being a relatively new technology and performance of it in many cases is not well known. The previous studies of its performance to often leave several unanswered questions and many factors ignored that should be required to make it a scientifically accurate test.

In this section, we will analyse some of the best and worst available studies and learn from their mistakes to ensure that the testing methods used provide the fairest results possible. Previous studies of sufficient quality may also be used to draw comparisons between results.

Below are four varying benchmarks comparing different components/technologies of the web. The first, goes in to detail about performance of WebAssembly compared to ASM.js in a number of high quality tests. The second goes in to detail about several issues found in JavaScript today not necessarily related to the underlying performance of JavaScript itself. It is important to remember that the technology may not always be to blame. The third discusses the claims made by the WebAssembly team of it being considerably smaller when compared to ASM.js, from the few comparisons available, it seems that this may well be the case. The final is about the biggest highlighted potential downside of WebAssembly, the call overhead. Should the call overhead be non-negligible, it could greatly change the way in which WebAssembly applications are developed, trying to reduce the number of calls.

### Benchmark 1

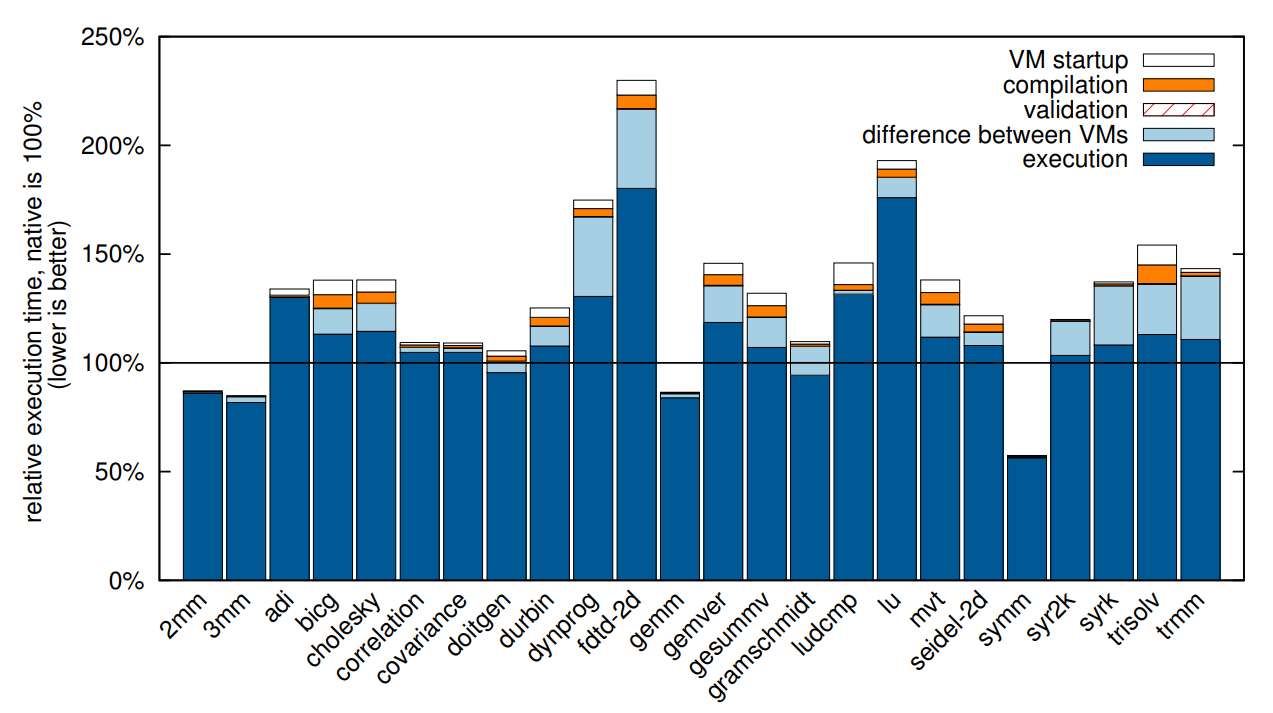
The first set of benchmarks are those found in (Google GmbH; Microsoft Inc. USA; Mozilla Inc. USA; Apple Inc, USA, 2017). These benchmarks have been approved by all of the major browser vendors and benchmark a wide variety of different aspects of the development cycle as well as performance. Some of these are not entirely relevant, such as improving compilation times as they are not relevant to improving front-end performance and longer compile times can result in deeper optimization and therefore a faster experience for the user.

Figure - PolyBenchC performance in WebAssembly relative to native speeds (Google GmbH; Microsoft Inc. USA; Mozilla Inc. USA; Apple Inc, USA, 2017)

The benchmarks taken here are compared to both ASM.js and native performance. As can be seen in Figure 3, WebAssembly runs very close to native in most of the tests in PolyBenchC, a C based benchmarking suite focusing heavily on maths and physics calculations. Over ASM.js, they found an improvement in performance of more than 33% (Google GmbH; Microsoft Inc. USA; Mozilla Inc. USA; Apple Inc, USA, 2017). Also recorded is the total file size of the compiled benchmark suite on each of the three compiler targets. WebAssembly beats out ASM.js being almost 40% smaller and even averages a 15% reduction over native x86-64 code.

The authors here have clearly put in a lot of effort into providing a large quantity of well detailed benchmarks. Each benchmark was run an impressive 15 times and compilers used for each format are specified. The benchmarks do however, fail to clearly specify the results on any browser other than Firefox and Chrome for desktop and hardware tested is limited to a single high-end desktop. A greater variety of browsers should have been used and across hardware configurations closer to the average user such as a phone or laptop.

While the benchmarks do provide some interesting results, most mean very little to an average developer. The PolyBenchC suite is made up entirely of numerical computations and it can be difficult to judge how much of an improvement WebAssembly could provide once other non-mathematical code is included into the program.

The overall impressive testing and results from this made me shift my own research priorities from creating a similar, purely mathematical benchmark suite to something that may be more relatable to an average developer targeting the web for a demanding application such as a game.

### Benchmark 2

Whilst the data provided by (Selakovic & Pradel, 2015) is not comparing differences between two different technologies, it does instead look at other important issues usually caused by the developer instead of the development stack used. It is also not limited to purely front-end execution performance that is focused on here.

The aim of this study was to find and resolve major performance issues amongst several existing JavaScript libraries and frameworks such as jQuery and Angular.js. Both of which are deployed across a huge number of existing websites. W3Techs found that the jQuery library is used in almost three-quarters of all websites (Q-Success, 2018). Because of the huge popularity of these libraries, any possible optimizations could have a far greater overall benefit to users than a small number of new websites adopting WebAssembly or ASM.js so the improvements are worth mentioning as they contribute towards my target of improving front-end performance.

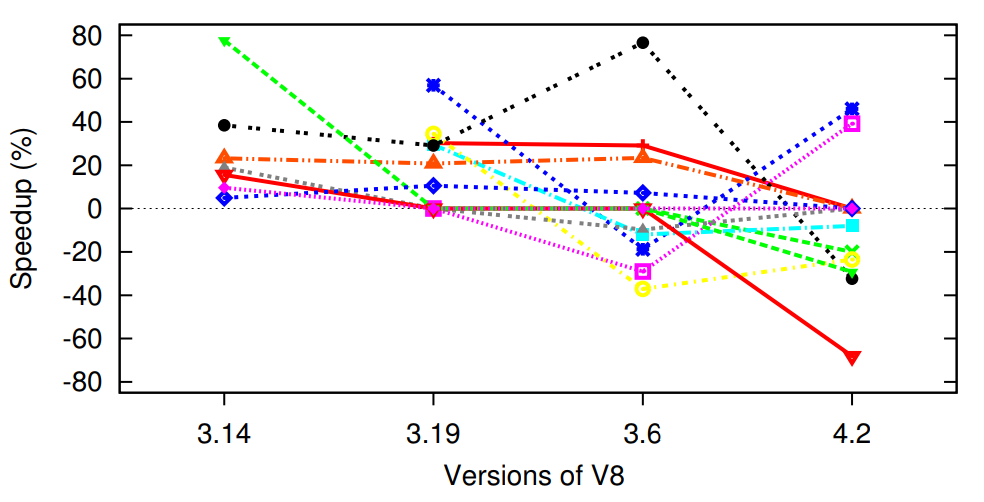
The testing of this paper focuses heavily on testing changes across different JavaScript engines. Its testing highlights that some ‘optimizations’ may provide performance improvements on some engines but can result in a degradation of performance on other engines. The results show that less than half of the optimizations tested provided a performance improvement across Google’s V8 engine and Mozilla’s SpiderMonkey. These results varied further across different versions of each engine, often drastically. This could mean that the improvements from specific optimizations could be completely reversed in an updated engine. In Figure 4 we can see that some of the optimizations deteriorate and, in some cases, reduce performance over the original code in newer engine releases. In the tests conducted, this appears to be an issue more specific to Chrome and the V8 engine, SpiderMonkey and Firefox tend to be more consistent across new versions.

Figure - Various optimization improvements over different versions of the V8 engine (Selakovic & Pradel, 2015)

Inefficiencies in the original tested libraries were caused by a variety of different issues. The most common root cause was down to inefficient API usage. It was particularly common for the String and Reflection APIs to be used ineffectively. This commonly occurred where a single function could be used to achieve the same result as two separate calls. Other common issues occurred with inefficient iteration, unnecessary computation of the same operation, over complicating simple tasks among many other issues which rarely occurred in testing.

While a small number of optimizations (3.52%) caused a decrease in performance and many had no measurable difference in performance (26.41%), a small number were able to achieve a significant improvement in execution times. 5.63% of optimizations reduced execution time by more than 10% and another 5.81% caused a reduction of more than 90%.

Data gathered from this research highlights an issue common to development across all languages and technology stacks. Often the inefficient use of the language tools can have a bigger effect than the performance of the language itself. The common issues found here would almost definitely have similar effects on any other language. Moving to a “faster” language, one which may be compiled rather than interpreted would only reduce execution time by so much when code has been poorly designed.

The research here is well formed and even considers inaccuracies and threats to validity within its own data, some of which could influence the results provided. For example, it is noted that only the V8 and SpiderMonkey JavaScript engines are tested, and that performance may vary further on engines such as Chakra, the engine powering Internet Explorer/Edge browsers. However, the research fails to point out a few important details that could have a great effect on the data gathered:

* There is no mention of hardware used for testing. This is something that could cause results to vary greatly. Optimizations may reduce the number of CPU cycles at the cost of memory and vice versa. So, some optimizations could have a negative effect on a mobile phone for example or even a greater positive effect.
* Neither ASM.js or WebAssembly have been referenced here. Some interesting research could have been conducted on whether introducing either of those could have improved the tested libraries further than just the JavaScript optimizations applied.
* The results found still show no real sign of what performance improvements are possible with real world data and has no mention of the kinds of data used in the tests. The changes could have a negative effect on larger or smaller test data compared with the original code.

As the intention of this study is to try and support decisions to use different technologies for new and possibly even existing projects. Even if moving to WebAssembly over JavaScript provided a large gain in performance, it is important to consider that some time refactoring using the current technology stack may provide a greater improvement in a shorter time period/at a lower cost.

### Benchmark 3 - Compiled size

One claim often made by the WebAssembly team or supporters of the project is the reduced file size of WebAssembly in comparison to an ASM.js module or JavaScript equivalent, even after minification and compression, but it is a claim rarely backed up by evidence, despite being easy to test. Unlike tests based on processing speed that require a great deal of effort to compare accurately, comparing the size of the results is simple. There is only one real factor to be critical of and that is the compilation mode of Emscripten. Otherwise results here are straightforward. Tested by Alon Zakai, the popular physics framework Box2D was reduced from 55.3KB in ASM.js to 43.1KB compiled to WebAssembly, a reduction of approximately 22% (Zakai, 2016).

Zakai, who founded the Emscripten project, also investigates some interesting methods to decrease the size of a compiled WebAssembly module using the Emscripten toolchain. Between Emscripten 1.37.22 and 1.37.29 compile size of the tested program found an 80% reduction and the use of the Closure Compiler finds a 50% reduction in JavaScript glue code total size (Zakai, 2018).

In addition to that, Zakai investigates the downsides of using C libraries and functions such as printf. The result of using these libraries increases compile size due to the entire library and any dependencies also being compiled. Instead, “EM\_ASM” can be used to call simple JavaScript such as “console.log()”. Replacing “printf” with “console.log()” reduced the WebAssembly module from 45KB to just 206 bytes and the JavaScript reduced by 90% (Zakai, 2018).

While initial signs are positive, there are few real-world examples of WebAssembly really resulting in a smaller compile size. Given the use of a binary format instead of a text format, it is expected for WebAssembly to produce smaller files. File sizes will be noted across all the tests produced in JavaScript, ASM.js and WebAssembly. File compression methods for JavaScript such as minification will be considered as well as testing the -Os Emscripten flag to produce size-optimized WebAssembly modules.

### Benchmark 4 – WebAssembly call overhead

The biggest claimed downside to WebAssembly currently is the overhead for context switching between JavaScript and WebAssembly when calling a WebAssembly function within JavaScript or vice-versa. Bebenita conducted a simple test consisting of two versions of a loop. One which would execute a small task with a loop inside the WebAssembly, the other would loop inside the JavaScript code and call a WebAssembly function for each iteration. The results are extremely significant, with the latter taking more than three times longer than the equivalent pure JavaScript. The former resulting in 30x better performance over the pure JavaScript version (Bebenita, 2017). Ben Smith, a developer of WebAssembly also notes that moving between WebAssembly and JavaScript will have some overhead (Smith, 2017).

Because of these results and the claims made by member of the WebAssembly team, it would have helped to have tested this area of potential performance downside. However, this is not possible to specifically test through Unreal Engine.

## Conclusion

While it might be reasonable to simply suggest WebAssembly is going to be a great success and replace ASM.js due to it being the newer technology and designed specifically for the purpose of high performance portable code, it still has a long way to go. Today, the compatibility of WebAssembly is far from perfect. While the four major browsers (Chrome, Firefox, Edge and Safari) do technically support it, the actual support can vary greatly. Mozilla appears to be far ahead of the other browsers, implementing new WebAssembly related features into Firefox before others get these features. Firefox recently released a new streaming and tiering compiler for WebAssembly (Clark, 2018) allowing WebAssembly code to be compiled as it is downloading, instead of compiling after file has fully downloaded. Outside of the four major browsers, support for WebAssembly is hit or miss and older browsers such as Internet Explorer will never receive support for the new standard. For many, ASM.js may still be the best option for a balance between performance and compatibility across all browsers.

Unlike the others, there are no reasons to recommend using a native plugin such as Flash or Silverlight, these are greatly outdated and often proven to be insecure. Modern browsers are also phasing out the use of these plugins and heavily restricting their use. As for performance, while the difference between native plugins and ASM.js or WebAssembly is relatively unknown, it is safe to assume that even if performance was greater, that it would not be worth the short-term gain if support is inevitably going to be dropped.

From the comparisons that are available, it does seem that WebAssembly is likely to be better for performance, because of its more efficient format, which is closer to an assembly language, but it does appear that it is not without some potential drawbacks. For that reason, it would have been beneficial to have included a test specific to this potential issue. However, it has been excluded due to the development platform and because of the likelihood that this latency will become irrelevant over time as the platform matures.

Other factors relating to performance should be relatively easily to find. Particularly compiled file sizes. As there will be numerous pieces to the overall development, the size of each can be noted. Little else can be done here to prove size difference, except for the Emscripten optimization flags which will certainly be considered and can even be modified in Unreal Engine which will be used heavily to test the graphics and maths performance of WebAssembly in a broader view.

# Methodology

## Introduction

This section discusses the methods to be used when conducting the research and generating data. The primary aim of the study is to discover if WebAssembly is an effective alternative to previous web technologies such as JavaScript and ASM.js. This is particularly focused on some uses that have not been focused on previously such as 3D gaming.

The research is made up two minor development components, each created using the Unreal Engine. The first focusing primarily on a high GPU workload. The test will consist of a high quality rendered area with effects such as lighting and reflections. The second focuses more on creating a heavy CPU workload. This will be created through forcing a high number of object collisions, something which is heavily reliant on the CPU with a great deal of mathematical operations.

## Development

In order to produce scientifically accurate data, some things must be taken into consideration during the development of the UE (Unreal Engine) based solution. First, since no publicly available version of UE can compile into both ASM.js and WebAssembly, Unreal Engine must be compiled from source to support this feature.

WebAssembly on UE is in active development and so choosing a single appropriate version can be tricky. The version used to compare both ASM.js and WebAssembly together will be UE 4.17. However, in this release WebAssembly is considered experimental and so to provide the results of a potentially more optimized WebAssembly export, the newly released UE 4.19 will also be tested against.

Rather than attempting to create my own demo projects, one sample project will be taken and modified them to meet the requirements. The freely available Content Examples pack contains several demonstrations specifically stressing a single component of the game engine, such as physics. Only two projects have been included and they focus specifically on the individual areas. The two main demos will be a graphically demanding demonstration, which will contain numerous graphical features such as particle effects, high poly-count models and large textures. The other will be a physics-based demo, the best way of testing raw maths performance in a game engine.

To reduce anomalies of the original benchmark, all user control will be removed as this has the potential to increase the variation between each run of the test, potentially making the posted findings invalid. Instead camera will be programmatically controlled to move in the same direction for every test.

## Testing

To record data, framerate and frame time must be recorded per frame. This data can then be viewed in several ways. Averages and exported data can be viewed in several ways and the winner should be easily visible from this. Thus, it will be relatively easy to draw a conclusion and recommendations for developers wishing to create UE games and exporting to the web.

It may be a little more difficult to recommend developers not primarily targeting the web to do so. If performance is within a close margin to native, then it may be recommendable to either focus primarily on exporting games to a web platform, or in addition to releasing native versions. WebAssembly and even ASM.js can potentially run within a standalone interpreter and so could appear as a native client and provide support without additional work for many different platforms such as Linux which can often prove difficult to develop for, due to its great number of varying distros. The nature of WebAssembly would remove many of the concerns of development across multiple platforms. The only downside would be the whatever performance difference is found from these tests.

Output code size will also be a factor for consideration here. It will be easy to compare the outputted file sizes when exporting to ASM.js and WebAssembly. File size is rather important for internet technology and can greatly affect loading times if a file is of suboptimal size, particularly over a slower internet connection or mobile connection.

There are several potentially detrimental factors to consider when testing the performance of the application. In terms of hardware, the performance can often vary greatly for different benchmarks and so effects of certain instructions can have a larger effect than others. To avoid this potentially happening, a variety of different hardware configurations will be used and tested.

Alongside the hardware, software can also have a huge effect on end results. Other applications running in the background can often have a great effect if they are attempting to execute. To avoid this as much as possible, a variety of software configurations will be used, including multiple browsers and multiple operating systems. This is with each test being run directly after a reboot and only the minimal required applications running in the background.

## Creating a Fair Test

As stated previously, several factors must be considered to provide the most valid results possible. First and foremost is to test against a wide range of hardware and software configurations. For that reason, the benchmarks will be tested against these 3 configurations:

### Configuration 1:

|  |  |  |
| --- | --- | --- |
| Name: PC | | |
| Operating System | Windows 10 Pro (64-bit) | [*microsoft.com*](https://www.microsoft.com/en-gb/windows/features) |
| OS Build | Version 1703 (OS Build 15063.540) |  |
| Processor (CPU) | AMD Ryzen 7 1700X @3.7GHz | [*amd.com*](https://www.amd.com/en/products/cpu/amd-ryzen-7-1700x) |
| RAM | G.Skill 16GB DDR4 @3000MHz | [*gskill.com*](https://www.gskill.com/en/product/f4-3000c15d-16gtzr) |
| Storage | Samsung 960 Pro 512GB | [*samsung.com*](http://www.samsung.com/uk/memory-storage/960-pro-nvme-m-2-ssd/MZ-V6P512BW/) |
| Graphics Processor (GPU) | XFX AMD Radeon R9 390 | [*xfxforce.com*](http://xfxforce.com/en-us/products/amd-radeon-r9-300-series/amd-radeon-r9-390-double-dissipation-black-edition-r9-390p-8db6) |
| Display | Acer 2560x1440 @144Hz | [*acer.com*](https://www.acer.com/ac/en/US/content/model/UM.HG0AA.001) |
| Chrome | Chromium 65.0.3325.181 (64-bit) | [google.com](https://www.google.com/chrome/) |
| Firefox | Quantum 59.0.2 (64-bit) | [mozilla.org](https://www.mozilla.org/en-GB/firefox/new/) |

### Configuration 2:

|  |  |  |
| --- | --- | --- |
| Name: Laptop | | |
| Operating System | Ubuntu 17.10 (64-bit) | [*ubuntu.com*](https://www.ubuntu.com/desktop/1710) |
| Processor (CPU) | Intel i7-4710MQ @2.5GHz | [*ark.intel.com*](https://ark.intel.com/products/78931/Intel-Core-i7-4710MQ-Processor-6M-Cache-up-to-3_50-GHz) |
| RAM | Unknown 16GB LPDDR3 @1600MHz |  |
| Storage | Western Digital 750GB Caviar Black HDD | [*wdc.com*](https://www.wdc.com/products/internal-storage/wd-black-mobile.html) |
| Graphics Processor (GPU) | Nvidia GTX 850M | [*geforce.com*](https://www.geforce.com/hardware/notebook-gpus/geforce-gtx-850m) |
| Display | Unknown 1920x1080 @60Hz |  |
| Chrome | Chromium 65.0.3325.181 (64-bit) | [google.com](https://www.google.com/chrome/) |
| Firefox | Quantum for Ubuntu 59.0.2 (64-bit) | [mozilla.org](https://www.mozilla.org/en-GB/firefox/new/) |

### Configuration 3:

|  |  |  |
| --- | --- | --- |
| Name: Phone | | |
| Model Name | OnePlus One A0001 | [oneplus.com](https://www.oneplus.com/) |
| Operating System | Android 7.1.2 Nougat | [*android.com*](https://www.android.com/versions/nougat-7-0/) |
| OS Build | N2G47O |  |
| Processor (CPU) | Qualcomm® Snapdragon™ 801 @2.5GHz | [*qualcomm.com*](https://www.qualcomm.com/products/snapdragon/processors/801) |
| RAM | Unknown 3GB LPDDR3 @1866MHz |  |
| Storage | Unknown 64GB eMMC 5.0 |  |
| Graphics Processor (GPU) | Adreno 330 | [*qualcomm.com*](https://www.qualcomm.com/news/onq/2013/01/11/inside-snapdragon-800-series-processors-new-adreno-330-gpu) |
| Display | Unknown 1920x1080 @60Hz |  |
| Chrome | Chrome Beta 66.0.3359.106 | [google.com](https://www.google.com/chrome/) |
| Firefox | Quantum for Android 59.0.2 (64-bit) | [mozilla.org](https://www.mozilla.org/en-GB/firefox/new/) |

The hardware being tested should provide a broad range of results from high end desktops down to a mobile phone with each varying greatly in different areas of performance. On each hardware configuration, the test will be executed numerous times. Each device will be tested using Google Chrome and Mozilla Firefox. These browsers are supported across all devices used and claim to support the WebAssembly standard.

Per each test, a reboot will be performed beforehand to so that there is no competition between the benchmark and other processes for computer resources. Other services provided by the Operating Systems used are not easily controllable and the average user would likely leave these background services anyway.

## Test configurations

There are many different versions that have been mentioned in previous sections of this research and many areas that need testing. The following is a full list of each version being tested which apply to both projects:

|  |  |  |  |
| --- | --- | --- | --- |
| **Build Target** | **Unreal Engine** | **Emscripten Flag** | **Other** |
| ASM.js | 4.17 | -O2 |  |
| ASM.js | 4.17 | -O3 |  |
| ASM.js | 4.17 | -Oz |  |
| ASM.js | 4.17 | -O3 | Multi-threading |
| ASM.js | 4.17 | -O3 | SIMD |
| WebAssembly | 4.17 | -O2 |  |
| WebAssembly | 4.17 | -O3 |  |
| WebAssembly | 4.17 | -Oz |  |
| WebAssembly | 4.19 | -O2 |  |
| WebAssembly | 4.19 | -O3 |  |
| WebAssembly | 4.19 | -Oz |  |

## Conclusion

The aim of this study is to test performance of the new WebAssembly compilation target on the web compared to other web and non-web solutions. Little testing has been done with WebAssembly to see how viable it is for day-to-day use, particularly for demanding games. To do this, 3 small projects will be created to test both gaming and non-gaming related applications.

The first aimed at testing WebAssembly’s performance when it comes to graphics. Conducting a graphics related test has many benefits. Firstly, the benchmark will be quite varied in what it tests. It will sufficiently test aspects such as memory reading (e.g. loading textures) as well as varied mathematical calculations through lighting, reflections etc. Using Unreal Engine should allow for high-end graphics to stress both the graphics card and CPU. We can compare the performance of the various WebAssembly versions to those compiled to ASM.js and natively compiled.

The second test will also be built using Unreal Engine, this one removing much of the stress placed on the GPU and instead further stressing the CPU with advanced physics simulation and collisions. Data for both this and the previous solution will be collected through regular framerate logs during simulation that can be recorded.

# Findings and Analysis

## Introduction

## Unreal Engine Benchmarks Results

### Introduction

This section contains the results for the two Unreal Engine based benchmarks. Containing various sections, the primary focus is comparing WebAssembly to ASM.js and various performance aspects of that. There are also a number of other contributing factors to achieving the best performance on the web. For that reason, a number of other web components have been tested and compared.

Multi-threading and SIMD performance has been tested to try and gauge what potential performance improvements it could bring when support for these planned features is added to WebAssembly in the future. This expected performance can of course only be estimated, there are too many changing variables meaning neither can provide similar performance improvements. However, because of the significant differences in performance which are discussed later, we can draw a confident conclusion from the results.

Another key component can be the browser used and results can vary greatly between them. Ideally, more browsers would have been tested, including all of the latest browsers from the W3C members (Chrome, Firefox, Edge and Safari) as well as those not part of the consortium. However, limited hardware availability and operating systems restricted the use of these. For that reason, the browsers tested were limited to Firefox and Chrome which were compatible with all devices used.

The final two pieces of data analysis are based on compilation. The first being the compilation from source code to the middle language (ASM.js or WebAssembly) and the second compiling from ASM.js or WebAssembly into executable machine code. Emscripten, the most popular toolchain for compiling C/C++ into code executable on the web and used by the Unreal Engine has a number of different optimization flags that can affect performance and file sizes. The compilation times for source code to the specified web languages were not tested. However, the outputted file sizes are compared and where possible the compilation times from ASM.js/WebAssembly to machine code is measured and compared.

Finally, some issues arose during the development and testing process and these are discussed with potential reasoning as to why these tests failed and what might be required to resolve these in the future. The biggest issues that arose were during testing on the Phone configuration. None of the tests on this device completed. References to tests on this device are not included in any section except the discussion of issues.

Results used in the graphs here are averages of multiple test runs. Most tests are run five times, three times on the first boot with a standard page refresh. Between the fourth and fifth tests the machine tested was rebooted and all browser history and cache cleared to ensure accurate fresh boot performance. Any tests that had significant performance decreases across the first three runs compared to others were only run three times as their results were already conclusive and there was nothing more to gain from testing these specific benchmarks again. All of the results were not even close to other results and those only tested three times had usually taken considerably longer to load and complete the test, the additional eight runs would have caused a significant increase in testing time.

### Multi-threading and SIMD

Unreal Engine version 4.17 offers ‘experimental’ support for both Multi-threading and SIMD (Single Instruction Multiple Data). While it is unknown how exactly Unreal Engine utilizes either of these features, it is expected that at best it would offer substantial performance improvements and at worst would offer little to no benefit. By offloading a task like world simulation to another thread, the main thread should be able to complete other tasks during that process and result in an iteration of the game loop being completed quicker. With SIMD you would expect a lot of repeated calculations to be completed in a fraction of the time. Multiple values can be loaded into the large registers of a modern processor and reduce the instructions required for a large block of data.

However, the results of these tests were completely unexpected. Tests run with multi-threading or SIMD enabled resulted in significantly worse performance. Figure 5 shows how significant this decrease in performance is and raises a number of questions as to how performance is so significantly decreased.

The results from this test, in figure 5, show a decrease in performance with multi-threading and a much larger decrease when SIMD is enabled. To ensure these results were not an anomaly, each test was repeated a total of 12 times. 3 times for each of the hardware/browser combinations.

Figure - Framerate comparison of ASM.js with and without Multithreading (MT) and SIMD

In this first test, the multi-threading test achieves just 57% of the total frames rendered against standard ASM.js. While the difference is significant, it does appear to significantly decrease over time during each run, reaching 82% during the last second of the test and a low of under 20% at the beginning of the test and this decrease in difference can be seen in Figure 6 below. If the test were to continue over a longer period of time then it could potentially perform better than the standard ASM.js. The multi-threading tests also failed to complete on Chrome on either device, so results are Firefox only.

Figure - Results of the Multithreading test as a percentage of ASM.js

On the other hand, SIMD produces entirely different results to the multi-threading test here. It averages just 18% of the total frames and does not appear to improve over time unlike the multi-threading tests. The framerate consistently hovers around 20 frames per second (fps) while the ASM.js test is consistently around 100 frames per second. The latter also potentially being limited by the 60 fps lock and 144 fps lock on the Laptop and PC devices respectively.

Figure - Comparison of Multi-threading (MT) and SIMD against ASM.js in the second Unreal Engine test

The more CPU intensive nature of the second Unreal Engine based test highlighted an even greater performance disparity between a standard ASM.js build and those built with either Multithreading or SIMD enabled.

The SIMD benchmark consistently struggled and achieved an average of just 4% of the number of frames rendered with no signs of improving towards the end of the test. Multithreading achieved marginally better results, still only rendering 7% on average, but performance increasing 300% between start and end of the test from 4% to 12%, seen in Figure 8.

While both Multithreading and SIMD support were considered experimental in Unreal Engine, the results were a surprise. Both features are designed to increase performance but the results of these two tests have shown to heavily decrease performance. The reason for this comes down to a poor implementation of these features as well as some limitations that come with JavaScript threading. Multithreading in JavaScript is only available through Web Workers which are limited in their functionality and their performance compared to the commonly used ‘pthreads’ are relatively unknown. Latency and thread creation costs may result in this effect and creation of threads could explain the gradual increase in performance as a sufficient number of threads are created.

Figure - Increasing framerate of SIMD against the standard ASM.js test

Poor SIMD performance in ASM.js may be down to its support being removed from active development (Ecma International, Technical Committee 39 , 2017), though it is incredibly poor performance is still unexpected, and this alone would not explain it. The only other explanation is a poor use of SIMD in Unreal Engine when it is compiled for HTML and the web. This issue has not been noticed by the Unreal Engine team or its community, but the issue arises consistently across different devices, tests and browsers, so the data is conclusive.

### Emscripten Optimization Flags

During testing the Emscripten Optimization Flags were varied to see the effect that Emscripten Optimization flags might have on the performance of a WebAssembly or ASM.js module. There are four available flags: -O0, -O2, -O3 and -Oz/-Os. The first three increasing in optimization for execution speed and the last prioritising compile size potentially at the expense of performance. The equivalent Unreal Engine settings can be found in the source code extract in Figure 9.

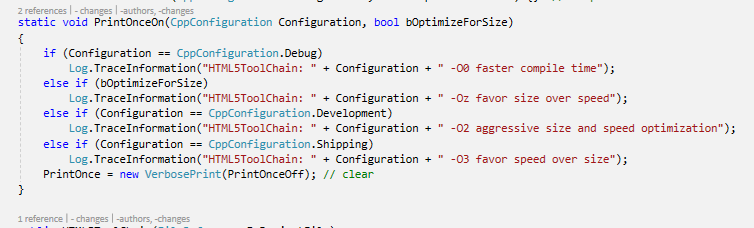
Access to the -O0 or “Debug” compile option was unavailable in the second test. This is an issue that could be recreated with other projects, and therefore must be an issue with Unreal Engine, however no previous instance of this occurring has been found. And due to server configuration issues, -Oz could not be tested due to Unreal Engine also compressing the file when this flag is enabled while the server could not handle files compressed by gzip. Because of this, testing was limited to the -O2 and -O3 flags, or Development and Shipping Unreal Engine configurations.

Figure - Section of UE4 source code displaying Unreal Engine build options alongside equivalent Emscripten flags

#### Unreal Project 1

The difference between the two flags in the first benchmark compiled to ASM.js, in Figure 10, is the reverse of what is expected. The -O3 flag actually causes a small decrease in performance of 1.44%. This is excluding the first data point which had a 28% increase in performance and therefore did not accurately reflect the overall performance of the test. The full data suggests that this test is reaching the 60 fps and 144 fps limits of the two devices.

Figure 11 - Unreal Project 1 - Comparing Emscripten flags in WebAssembly

Figure 10 - Unreal Project 1 - Comparing Emscripten flags in ASM.js

In this project, WebAssembly offers a small improvement of ~2.5% when moving from the -O2 to the -O3 optimization flag. Because this test is more GPU focused, improvements in ASM.js or WebAssembly are going to be small in nature as most time between frames will be spent drawing the frame, a process relatively unaffected by these flags. These tests also regularly hit the maximum framerates of the devices and therefore differences might be smaller than expected.

#### Unreal Project 2

The second Unreal Engine project, which is significantly more CPU intensive shows a large performance increase between the two flags, closer to what was originally expected. The 20% performance improvement on average is much closer to what was expected and makes the choice clear when deciding which flag to use, given the almost negligible difference between the flags in the first project.

Figure 13 - Unreal Project 2 - Comparing Emscripten flags in WebAssembly

Figure 12 - Unreal Project 2 - Comparing Emscripten flags in ASM.js

Like the ASM.js tests, WebAssembly shows a significant 14.5% average increase in framerate between the optimization flags. As mentioned previously, this is the result that was expected when comparing the two flags and the first project test were not sufficient in testing the performance difference here.

#### Conclusion

The first test project failed to show any significant difference either way. However, the second project showed significant increases in ASM.js and WebAssembly of 20% and 14.5% respectively. The results of the second project are enough to prove that the Emscripten flags do indeed provide a significant improvement. It is expected the -Oz flag would perform somewhere between -O2 and -O3 performance but further testing beyond the scope of this research is required to confirm whether this is true.

### Browser Performance Comparison

A major contributing factor to performance and compatibility on the web comes from the browser used. For that reason, it is important to test how different browsers react to the various versions tested. Due to device limitation and browser compatibility, testing was limited to the Firefox and Chrome browsers, which offer versions available on all test devices.

Figure 14 - Unreal Project 1 - ASM.js on Firefox and Chrome

Figure 15 - Unreal Project 2 - ASM.js on Firefox and Chrome

Displayed in Figures 14 and 15 are the average performances of the all the ASM.js builds for the two Unreal Engine projects. In the first test, differences are minimal, Chrome having a 2% advantage over Firefox in this test. In the second test, Firefox averages significantly higher than Chrome particularly at the start and averaging 51% increased performance overall. Once again, a significantly increased difference in first test. An expected result due to the nature of the first test pressuring the GPU rather than the CPU.

In both tests, Chrome improves its performance when working with WebAssembly. For the first test, it doubles the gap between itself and Firefox to 4%, though still a relatively small increase. And in the second, an increase to 54% higher performance from 51% in ASM.js.

Figure 17 - Unreal Project 2 - WebAssembly on Firefox and Chrome

Figure 16 - Unreal Project 1 - WebAssembly on Firefox and Chrome

WebAssembly gives a greater advantage to the Chrome browser and when it comes to the CPU heavy test, it performs significantly better than Firefox. These results do line up with the findings of Mashable where Chrome came out on top in two out of three of its performance tests (Chin, 2017). However, testing completed by Digital Trends finds Firefox and Chrome almost equal, each winning beating out the other in two of the four tests conducted (Coppock, 2018). The variation in performance between the browsers is expected due to how new WebAssembly is. The code in each browser will be different implementations and neither of which will be highly optimized and so anomalies and large differences are expected.

### WebAssembly vs ASM.js Performance

The primary aim of this investigation was to find the usability of WebAssembly and prove or disprove the claims made by the WebAssembly development team of greatly improved execution performance over JavaScript on the web and other advantages such as reduced file size. Using Unreal Engine to create these tests allowed for more rapid development and makes it much easier to form valid tests that can stress both the CPU and GPU with a varied workload.

The first data point has been removed from all but Figure 18 for the first test. It varies greatly compared to the rest of the test, particularly with ASM.js. This is an example of the browsers generating “warm code”, code reoptimized when repeatedly executed. Naturally in a game, a large amount of code is repeatedly executed each frame. We can see that warm code is rapidly generated and the ASM.js versions of the project reach a fully optimized state within 2 seconds of the test start.

In all later graphs, the best performing version of ASM.js and WebAssembly is used. This results in one test using ASM.js -O2 instead of -O3 due to it performing marginally better. We want to compare the strongest from either side.

Figure 18 - Unreal Project 1 - WebAssembly vs ASM.js overview

#### Overall Performance Comparison of WebAssembly and ASM.js

Figure 19 - Unreal Project 1 - Best WebAssembly vs ASM.js

Figure 19 provides an interesting and unexpected result with WebAssembly performing marginally worse than ASM.js consistently throughout the test. ASM.js beats out WebAssembly by about 1.2% and in a few cases dips below WebAssembly in framerate. Without understanding the implementations of the code, it is difficult to say where the small increase has come from. This difference comes from a maturity in the WebGL interface for JavaScript and the latency and switching between WebAssembly and JavaScript with not a lot of code being executed within WebAssembly. Unlike the second test. Where the increased amount of processing within WebAssembly allows it to gain a greater advantage to ASM.js, given few context switches are required.

Figure 20 - Unreal Project 2 - Best WebAssembly vs ASM.js

Because WebAssembly is still at an early stage, further optimizations and tweaks can be expected to the point where a future browser and Emscripten updates along with advances in the WebAssembly standards and specification should beat ASM.js for this test.

As with previous comparisons, the second project has returned more divisive results that are more in line with expectations. WebAssembly almost consistently beats out ASM.js results by a reasonable margin. With an 11.3% increase in performance, the victor in this test is clear. Unlike the first test, this test does appear to provide performance more in line with previous research results which generally give WebAssembly a small but noticeable jump in performance.

This test displays a similar pattern to previous tests where ASM.js particularly struggles at the start of the test. This is typical behaviour for JavaScript and ASM.js if it has not been optimally compiled. The rapid increase within the first two seconds of the test is down to the browser(s) recompiling the JavaScript to generate “warm code” for many functions or sections of code that are repeated regularly. Since this is a test with a large number of repeated collisions and collision resolutions, only with different parameters, those specific functions will be selected for optimization by the JIT compiler of the browser because of the regular repetition. This is a characteristic likely to be seen across the core of the engine. Much of the code in this area will be repeatedly executed at least once per frame. The collisions will however take priority, as the test will cause thousands of collisions every second.

#### Browser Specific Comparison

Results between the two browsers tested can vary greatly. Previously tested was the performance of each specific technology on either browser, but not comparing ASM.js and WebAssembly on each browser. If each browser consistently favours a particular technology, then it is possible to load a different version based on the users’ browser.

Figure 21 - Unreal Engine Project 1 - Firefox ASM.js vs WebAssembly

Figure 22 - Unreal Engine Project 2 - Chrome ASM.js vs WebAssembly

A few patterns emerge when we break the first test into browser specific performances and not only compare them to one another but to the overall averages. In Firefox we see ASM.js go against expectations and produce better results than WebAssembly, even beating out the overall average of the test, while WebAssembly performs below the average and even further below ASM.js.

It is a surprising result, since it goes against what WebAssembly has been created for. However, ASM.js is the more mature technology and Mozilla (developer of Firefox) are the original developers behind ASM.js and pushed it more than other browser vendors, being one of the few to specifically optimize for the code subset.

In Chrome, results are more in line with the expected results. WebAssembly outperforms ASM.js consistently and performs above the overall average. Chrome was never a major supporter of the ASM.js project, while it has been a major supporter of the WebAssembly initiative and Google (developer of Chrome) is a member of the WebAssembly specification team (though Mozilla is also). While Chrome performs better in this case, this may not be true for other projects. Both Firefox and Chrome are known to be highly optimized and small variations in individual tests are to be expected.

Figure 23 - Unreal Engine Project 1 - Firefox ASM.js vs WebAssembly

Figure 24 - Unreal Engine Project 2 - Chrome ASM.js vs WebAssembly

The broken-down results for the second project are very interesting and the beginning of the tests invalidates previous analysis. In the first four seconds of the test, performance of WebAssembly and ASM.js on the two browsers are almost the reverse of one another.

At the beginning of the test, ASM.js performs significantly better than WebAssembly on Firefox and at times achieves twice the performance of WebAssembly on the browser with an overall 64% improvement in the first four seconds. Towards the end of the test, performance returns to an expected state with framerates on both tests stabilising and WebAssembly achieving a reasonable 13% performance increase over ASM.js.

In the first four seconds of the test, the difference between the two technologies is large, but more in the expected direction. For the first four seconds, WebAssembly outperforms ASM.js by an impressive 116%, more than double the ASM.js framerate. During this period, ASM.js suffered from a combination of unoptimized code from the JIT compiler and the JIT compiler spending more time producing more optimized code. Similar to Firefox, results normalize for the later part of the test and performs 7.4% better than ASM.js for the second part of the test.

After specific browser analysis, it is difficult to rely on the data from the first four seconds because of how much it varies between the two browsers. WebAssembly on Firefox in particular, produces results far from what is expected, but an average of the two browsers appears to produce normal results. It is difficult to say why WebAssembly performs so badly at the beginning of the test but is an average of ten results on two different devices and so is consistently bad for that duration. Attempts were made to contact the Firefox development team to find out the reasoning behind the performance issues and no official response was received. However, a similar bug has been reported in Firefox. The user states that WebAssembly performance stutters in another Unreal Engine based demo on Firefox. The bug is currently being tracked, however there is currently no expected date for a fix and is listed alongside a number of other WebAssembly related bugs, which were not encountered during the testing conducted here.

An official response to the request associated with this research or the listed Bugzilla bug may be received at a later date and these can be tracked through the following links:

* StackOverflow issue: <https://stackoverflow.com/questions/50163320/firefox-webassembly-specific-performance-issue-in-unreal-engine-4>
* Bugzilla issue: <https://bugzilla.mozilla.org/show_bug.cgi?id=1341133>

### Compilation Size and Times

A big issue on the web is page load times. Many modern websites have been found to be bloated and take longer to load than they really should. With ASM.js and WebAssembly, two components are crucial to the loading time of the page. The first is the download size of the page. The effect this has generally depends on the internet speed of a user and can disproportionately affect users in rural areas or those using 3G/4G mobile connections. Compilation times on the other hand is not affected by the users’ internet speed, but more by the hardware used. Compilation times can be considerably worse on a mobile or low-end PC/laptop device due to the considerably weaker hardware in mobile phones.

A large range of user devices can mean a dilemma for developers. For a product release, either the -Oz or -O3 flag would be used. The decision is difficult, as either way a set of users may be negatively affected.

#### Compilation Sizes

Figure 25 - Comparison of produced file sizes for each version of Unreal Project 1 tested

The overview in Figure 25 provides some general patterns across everything tested. The -O2 Emscripten produced the largest files. -O3 produces slightly reduced file sizes but consistently falls well short of the small file sizes produced by the -Oz + gzip combination. Outside of the module side, it is interesting to note an improvement in Unreal Engine between 4.17 and 4.19 where file sizes are consistently smaller, even using identical build parameters.

Figure 26 - Comparison of produced file sizes for each version of Unreal project 2 tested

A problem that arose here is gzip. As noted previously, it made the compiled project incompatible with the web server being used. Unreal Engine has no method of disabling gzip when using -Oz Emscripten flag. In previous investigations, it was found that gzip reduced the size of the jQuery JavaScript library by 65%. Using this value, it can be estimated that the -Oz compiled ASM.js module would be around 53,000KB in size. From that, the -Oz flag can be estimated to reduce the file size by approximately 50% to just 26KB in size. Assuming gzipping has a similar benefit in the WebAssembly UE4.19 version, its approximate size alone based on the -O3 version would be approximately 16,000KB. It may then be estimated that the Emscripten optimization flag saves an extra 37% in file size down to 11,570KB. It is likely that the process of gzipping is less effective on WebAssembly, primarily due to its binary format. A binary format should naturally be more compact and efficient than a text representation.

The compilation of the second project produces almost identical results for the module. Only the overall size is noticeably different due to a greatly reduced number of textures and 3D models included. The changes between the project are relatively small in code with the main part of the compiled component being the engine. This is the reason behind the changes being very small.

Comparing ASM.js against WebAssembly we can see significant improvements in both of the projects across the board. WebAssembly offers significant reductions, particularly in the versions unoptimized for size. Both -O2 and -O3 result in a reduction of around 70%. When optimizing for size and going through the process of gzipping, the module size decreases by a great deal for both ASM.js and WebAssembly. ASM.js module size is reduced by 83% from the -O3 mode. The difference between WebAssembly and ASM.js is noticeably smaller than the previous versions. The 55% reduction may be smaller than the other versions but, reducing file sizes by more than half is a significant improvement if it can be repeated on regular websites, that are not running a large 3D project.

Figure 27(b) - ASM.js vs Wasm in Unreal project 2

Figure 27(a) - ASM.js vs Wasm in Unreal project 1

#### Compilation/Loading Times

Reducing load times should be a big focus for any web developer. A report by DoubleClick, a Google subsidiary found 53% of mobile users leave the website if the page fails to load within the first three seconds (Doubleclick, 2016) and 25% increased ad views for fast loading pages. This research does not consider any loading indication which can extend the time a user is willing to wait. However, research at Carnegie Mellon University discovered that progress bars can have a negative effect on loading times less than 5 seconds (Missig & Dickison, n.d.). For that reason, it is important to consider how quick each version loads and consider the appropriate feedback given to the user during loading.

When looking into load times, there are some important factors that must be considered due to the great effect they can have on the results. Both a first load and a warm reload must be tested. A first load is taken on a fresh boot with all browser cache and data cleared. First load was tested three times, resetting the machine and browser cache between each. A warm reload was also tested twice after the first test to find the effects of caching on the browsers.

Issues arose when testing this value for the second project. In most cases, the compilation time would be displayed for a short period of time before the project loads, whilst loading textures and performing other tasks outside of compilation. However, the second project does not contain any textures and so no time was left between end of compilation and final project loading to be able to record the compilation times. This occurred on all WebAssembly versions built with Unreal Engine 4.19 (and very rarely on 4.17). These findings suggest that changes have been made to the loading process between the two versions which are undocumented, resulting in the compile time not being shown to the user.

Figure 28 - Compile times for the major tested versions

For the first load we can see significant reductions in compilation time. The slower development build of ASM.js took almost 18 seconds to compile on average while the equivalent version in WebAssembly reduced the compilation time by just under 75%. The shipping version of the project increased this to more than 75% on average for the first load. While there are other tasks that are completed during loading that were not recorded. The final version of the WebAssembly took only 3.1 seconds to load and falls just shy of the 3 second limit where 53% of users choose to leave the page. This drop off in visitors can vary based on whether alternative sites are available and whether the experience is considered by a user to be worth the wait.

When breaking down the results into the separate browsers we can see that while ASM.js is relatively consistent on first load across both, WebAssembly produces wildly different results. WebAssembly in both cases is compiled in under 1 second on Firefox, about 25x faster than the ASM.js equivalent. However, compilation in Chrome takes significantly longer. It does still offer a considerable improvement over ASM.js and offers a reduction of more than half with both build types.

Figure 29 - Compile times by browser and build version

Reloading the page without clearing caches first reveals only one interesting pattern seen in Figure 30. Consecutive load times of ASM.js in Firefox see a significant reduction after the first load. Code caching is being used to cut compilation times significantly here.

Figure 30 - Consecutive compile times after quick refresh

Elsewhere, results remain similar. ASM.js on Chrome and WebAssembly on Firefox produce almost identical results on each run. WebAssembly on Chrome shows a minor increase of 100-200ms on each run. Separating results into specified devices in Figure 31 below shows that the trend is primarily seen on the PC device. However, the second and third run both show an increase over the initial run on the Laptop device too. A trend that does point towards increasing load times per run, a potential concern if the trend were to continue upwards.

Figure 31 - WebAssembly Chrome consecutive load times by device

### Development Issues

During development and testing, a few issues arose that affected the quality of results and the amount of testing that had been planned. This section goes into some detail about these issues and what could be done to resolve these issues in future testing.

#### Compressed Versions

As discussed previously, it was not possible to test the compressed versions of any project. This was due to the remote server being used to host the web pages, supplied by [freehosting.com](https://www.freehosting.com/), being unable to serve compressed files. The failure to load was unexpected and came late in the testing phase and had not been previously tested to ensure compatibility. It was an option to use a locally hosted server on the machine to ensure compatibility, however would have compromised the results of the compressed versions. Performance would likely have been affected by a server running alongside the client. Because of this, no results were recorded for any compressed versions, but using the definition of both Emscripten and Epic Games, it is safe to assume the size optimized Emscripten flag would result in performance between the -O2 and -O3 flags.

#### Multi-threading in Chrome

Because of the console error displayed in Figure 32, the multithreading versions of both tests failed to load in Chrome. Since the multi-threading versions failed to show any promising results in Firefox, it is not expected that Chrome would produce results much stronger than those on Firefox and it is not an issue worth pursuing further.

Figure 32 - Console Error from Google Chrome and multi-threading test version

Failed to load: Uncaught (in promise) RangeError: Invalid atomic access index

at store (<anonymous>)

at \_\_\_pthread\_mutex\_trylock\_owner (\*\*:28)

at \_\_\_pthread\_mutex\_trylock (\*\*:7)

at \_\_\_pthread\_mutex\_timedlock (\*\*:28)

at \_\_\_pthread\_mutex\_lock (\*\*:7)

at \_\_ZN5physx6shdfnd9MutexImpl4lockEv (\*\*:11)

at \_\_ZNK5physx6shdfnd6MutexTINS0\_19ReflectionAllocatorINS0\_9MutexImplEEEE4lockEv (\*\*:6)

at \_\_ZN5physx6shdfnd6MutexTINS0\_19ReflectionAllocatorINS0\_9MutexImplEEEE10ScopedLockC2ERS5\_ (\*\*:7)

at \_\_ZN5physx13GuMeshFactory18addFactoryListenerERNS\_21GuMeshFactoryListenerE (\*\*:17)

at \_\_ZN5physx9NpPhysics14createInstanceEjRNS\_12PxFoundationERKNS\_17PxTolerancesScaleEbPNS\_6pvdsdk5PsPvdE (\*\*:27)

*[\*\* : blob:http://kieranwarren.com/aa13345a-7727-40a3-87c8-7126c066fdfb]*

In the console error in Figure 32, the URL and current GUID have been removed to simplify reading of the error. This error is most certainly caused by threading and attempting to access an incorrect index within the underlying threading code (which uses the pthreads discussed previously). A resolution within the project is most likely not possible as the error is almost definitely a browser incompatibility, since Firefox has no issues loading the project, albeit with poor performance.

#### Device 3 (Phone)

Originally, it was planned to have 3 test devices. With the third being a mobile phone to try and create a good variety of different devices. However, no test was successfully completed, with only one configuration partially loading. Here we discuss further the reasons behind the tests failing in Chrome and Firefox. The issues differed greatly between the Chrome and Firefox, therefore they are discussed separately.

##### Chrome

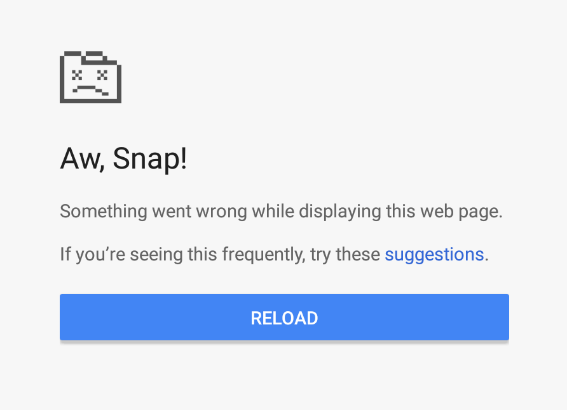
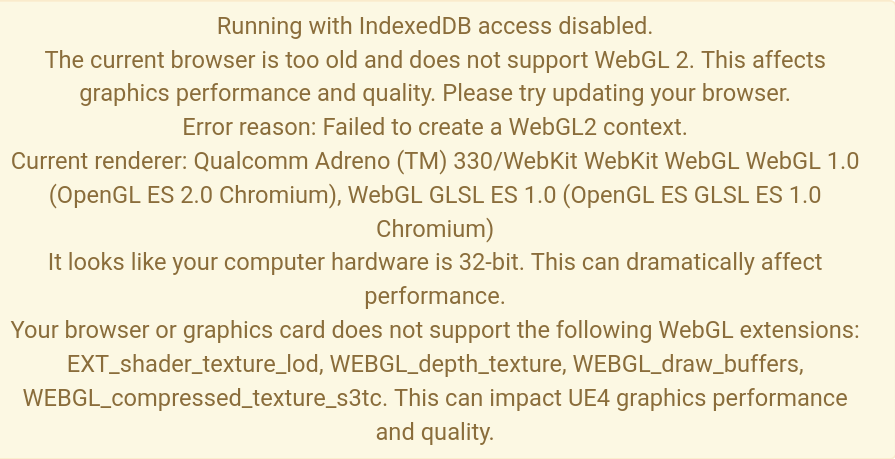
Neither the ASM.js or WebAssembly versions of either project successfully loaded. The WebAssembly version continued to load for more than 15 minutes, so it was assumed to have failed. The ASM.js failed to even complete downloading and crashed the page during the download as seen in Figure 33. During loading, both versions displayed the warning messages seen in Figure 34, which shed some light into why it may not have loaded.

Figure 34 - Warnings displayed on Chrome for both ASM.js and WebAssembly

Figure 33 - Chrome message when loading ASM.js project

Figure 34 in particular suggests that compatibility issues are the likely reason behind it failing to load or taking an unreasonably long time to. Firstly, the device does not support the newer WebGL2 standard, of which all projects were built on. Secondly, the 32-bit CPU appears to be a reason for slow loading. 64-bit CPUs are standard in most newer phones. The phone tested was released in 2014, before 64-bit mobile CPUs were popularized.

##### Firefox

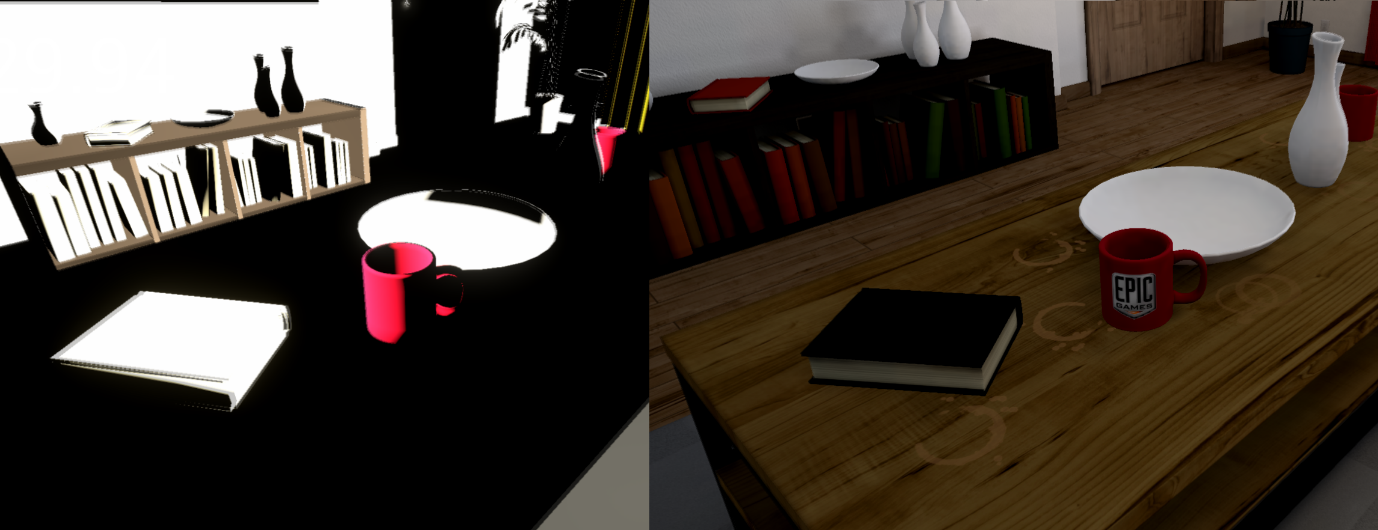
WebAssembly in Firefox produced some promising results with the first project successfully loading in reasonable time. It did however load with graphical issues with textures missing and light incorrectly rendered, an example of which is displayed in Figure 35. Because of these issues, it would not pass a simple usability test and performance statistics were impossible to record.

Figure 35 - Incorrectly rendered on phone (left), correctly rendered (right)

Other tests were not so successful in Firefox. With ASM.js returning the error in Figure 36(a). The error appears to contradict itself. It suggests that it attempts to load WebAssembly and that ASM.js is unavailable. A confusing error since no other hardware/browser combination returns the same error when loading ASM.js projects. When loading the second WebAssembly project, the rather generic error in figure 36(b) is displayed. This error does not really give any hint as to why the project is incompatible.



Figure 36(b) - WebAssembly project 2 Firefox (mobile) error

Figure 36(a) - ASM.js Firefox (mobile) error

##### Conclusion

It was rather frustrating to find that the phone device failed to successfully run any of the projects. The device may be considered outdated by some and is not a device that is very reflective of the average user. It is unknown whether the project successfully loads on other devices, but there is definitely a better chance of it running on a more recent device. Particularly one with a 64-bit CPU and supports WebGL 2. This could not be tested due to lack of accessibility to modern devices.

#### Recording Performance Results

During the development phase, a few issues arose that prevented results from being as detailed as originally intended. The original plan was to record the framerate at each individual frame. This in itself was not the issue. The issue arose when attempting to record and export this data from the project when exported to the web.

On first attempt, a bug was discovered in the Unreal Engine. The entire list of framerates was placed into a textbox with the intention of copying to the clipboard and exporting to the appropriate place. However, copying from a text box in either HTML5 version is currently unsupported.

Other attempts were made to export directly to a file either with the built-in save game functionality within Unreal Engine or writing to a text file. Once again, neither of these methods worked as expected when exporting to HTML5. Both require access to write to a file, which is not an available permission within the sandboxes used in modern browsers. Some browsers such as Chrome have implementations of file I/O used specifically with native plugins. However, these have now been almost entirely deprecated because of security concerns (Chrome Developers, Unknown). File I/O without specific user permission is something unlikely to return because of these security concerns

Due to these failures, recording data had to be reduced to a method with reduced data and a more manual method of recording. Framerate recordings were limited to the average of the previous second and recorded manually from the screen. This method still produced reasonable results, except at the beginning of tests where it would have allowed for more detailed analysis of the “warm code” effect on the ASM.js projects.

## Conclusion

While some tests were not completed due to technical issues, the data that was gathered is of good quality. Using that data, a number of conclusions can be drawn. Some areas provided results along with what would have been expected while others provided surprising results.

WebAssembly, the primary focus of the research, was found to offer a good improvement over ASM.js when it came to the intensive CPU focused work from the second project. The second project found on average an 11.3% improvement when moving to WebAssembly, a conclusive result that can be recommended upon. However, it fails to solve a number of existing issues with the web outside of performance. WebAssembly is still in its early stages and as a result many minor or more niche browsers do not yet support the standard.

WebAssembly also removes multi-threading and SIMD, but the results show that, at least in the case of an Unreal Engine based project, this is insignificant and not recommended in its current state. It is likely that particularly SIMD.js should not be used anywhere in its particular state. There is also little chance of it being improved in the future since it has been effectively abandoned and will be replaced by a WebAssembly based SIMD implementation. In theory, it should offer improved performance, but the reality may vary when it is eventually implemented.

WebAssembly has also proven itself not just in execution performance but also in compiled size and compilation times. Results from these tests found WebAssembly to be even smaller compared to ASM.js than previous investigation might have suggested with a 55% reduction, compared to the 22% reduction in Box2D and 40% in PolyBenchC.

# Conclusion & Recommendations

This study’s conclusion consists of three sections. In this first section, the hypothesis of the research will be discussed and concluded upon through the previously analysed data. A decision will be made as to whether it was proven, disproven or still inconclusive. Final conclusions relating to that. The second section will be a discussion of the original aims and objectives and whether these have been achieved in the project and how successful they were.

Finally, a discussion on recommendations. This recommendation section will be split into two sections. The first will be more academic recommendations, relating to the research itself and how it could be built upon in future studies. The second will be recommendations for developers that may be looking to start a new project reliant on WebAssembly/ASM.js or converting an existing project to either platform.

## Conclusion

### Conclusion 1

The first conclusion that has been produced from this study is;

*Using WebAssembly over ASM.js in its current state results in a significant performance improvement in CPU heavy workloads.*

The findings from the study support this statement. In the CPU heavy workload, an 11.3% performance improvement is achieved. This improvement can be viewed in Figure 20 and WebAssembly better than ASM.js through almost the entirety of the test. Further investigation did highlight that improvements can vary based on a number of different factors including the browser used and the method used for compilation.

### Conclusion 2

The second conclusion that has been produced from this study is;

*WebAssembly in every case produces considerably smaller compiled files than ASM.js directly benefiting end users.*

This statement is supported by the previous research conducted on WebAssembly module sizes in addition to the findings of this research. Previous research found size reductions ranging from 22-40%. The results from this investigation fared even better, and in all cases reducing file sizes by more than 50%.

## Aims and Objectives

The aim of this study was to prove the effectiveness of WebAssembly as a replacement to JavaScript/ASM.js and to find the best choices during the development phase for optimal application performance. These aims have been successfully achieved with WebAssembly coming out on top and comparisons between Emscripten optimization flags, enabling/disabling multi-threading and SIMD helping to find the optimal solution with the multi-threading and SIMD investigation providing valuable data which may remain valid when those features are added to the WebAssembly specification in future.

There were multiple objectives outlined for this research project and now the success or failure of each will be discussed, with recommendations for any objectives that were not completed.

1. Conduct a literature review. Primarily into different targets used to execute code on the web, with a particular focus on performance. This will help to understand the current state of running advanced applications on the web.
   1. This objective was successfully achieved and aided in narrowing down the development and testing process as some targets could be removed entirely.
2. Develop projects that can be exported to a number of the different platforms identified in the Literature Review and contains a way of recording the performance of the project.
   1. This objective was only partially achieved. Due to limitations/issues with Unreal Engine, recording of data was limited and had to be manually recorded. If these issues are resolved in the future, then reconducting these tests in the same manner with an improved recording technique may produce different results.
3. Export the project into numerous web-based configurations.
   1. This was successfully achieved, with a large variety of different configurations that varied based on export platform, Emscripten settings and additional features such as SIMD and multi-threading.
4. Record performance of all versions of the project numerous times across different browsers and hardware to produce reasonable results.
   1. This objective was successfully achieved except in a few cases. Performance in a few of the project versions could not be recorded.
5. Analyse the results from objective IV to determine what the best solution might be for future projects and existing projects and identify any patterns and determine reasons for those patterns and results.
   1. This was successfully achieved, and a number of different conclusions were discussed in the analysis. Some recommendations can come from the results which are discussed in section 6.3.

## Recommendations

Recommendations for this project have been divided into sections. The first is recommendations for future work following on from this research. The second section is specifically aimed at developers looking for conclusions on which technologies are specifically best to use at this moment in time.

### Research Recommendations

The research carried out provides very conclusive results which in many ways may be difficult to build upon. If looking at Unreal Engine (or games) specifically, then after testing GPU and CPU specifically, it would make sense to balance and make use of both demanding graphics and an intensive CPU workload in a single test, in order to better simulate a real game. Had a test like this been created prior to the knowledge of the full effect of WebAssembly on both CPU and GPU then the results would likely be between the two tests. It would vary slightly depending on whether the workload shifts more towards CPU or GPU.

WebAssembly is still in its early stages and so this research may become outdated quickly dependent on changes to the WebAssembly specification and the optimizations that the browser vendors may introduce specific to WebAssembly. Planned features such as SIMD and multi-threading have been considered and an attempt has been made to predict what difference they may make. However, the research showed that, at least in an Unreal Engine project, that these features may not bring any performance improvements. They could instead reduce performance similar to the effect seen with these features in ASM.js throughout section 5.

### Developer Recommendations

This research was largely based around clearing up confusion and expectations about the various web technologies to aid in decision making for new projects.

Given WebAssembly is still in early development, it is difficult to recommend for all kinds of projects. If the developer is aiming for performance critical applications and willing to sacrifice maximum browser and device compatibility, then WebAssembly might be an appropriate choice. It would still be wise for the developer to use a fall-back which may be based on ASM.js or WebAssembly. This does highlight one issue with Unreal Engine where ASM.js has been entirely replaced with WebAssembly in 4.19, meaning there may be difficulty in creating the fall-back version. Depending on the development platform, if web is not critical, then a natively compiled version may make more sense.

For any application not designed to be performance critical but instead maximum browser and device compatibility critical then WebAssembly is not a sensible choice. ASM.js or vanilla JavaScript would make much more sense in these cases since every modern web browser is expected to support JavaScript.

### Future Work

Due to some of the issues during the testing phase. There is some potential for some future work, primarily within the mobile space. Any researcher wishing to continue this work should ensure access to the latest mobile devices. This was an unexpected issue that arose in the final stage of testing and sourcing a replacement mobile device was no longer possible. The first portion of the future testing should be focused on creating a basic project that runs consistently on mobile through multiple browsers. However, in terms of features, the second project does not get much simpler. It contains no additional models or textures. This means that very little outside of the engine itself requires loading. This is where the issues were found in this study, so this must be resolved first. However, there may be further issues on mobile even if loading succeeds but recommendations cannot be made on unknown issues.

Once a minimal project has been achieved that runs consistently on mobile, then it can be expanded upon to increase variety in the test. Should one want to continue testing Unreal Engine’s usability on the web rather than performance specifically, then one might wish to include additional features to test their usability and whether issues occur using them through the web. For example, during testing, particle effects were experimented with but in no case did they successfully load on any device due to incompatibilities.

Alternatively, if one wishes to move away from Unreal Engine specific testing, then they may want to create a series of specific tests on some of the areas highlighted in the literature review such as JavaScript “warm code” and the claimed overhead of WebAssembly.

Future research would have the benefit of newer versions of browsers as well as some of the expected WebAssembly features. Therefore, it may be beneficial to see how WebAssembly has progressed between now and the time of research using the existing projects created for the purpose of this research. In this case, it would be important to consider the hardware used. CPU and GPU benchmark databases such as PassMark (https://www.cpubenchmark.net/) can be used to estimate the gain in performance between the hardware used here and those used in future. That change in performance can then be deducted from any differences between the results here and those taken in the future.

# Bibliography

Barney, B., 2017. *POSIX Threads Programming.* [Online]   
Available at: https://computing.llnl.gov/tutorials/pthreads/

Bastien, J., 2017. *Threads.* [Online]   
Available at: https://github.com/WebAssembly/design/issues/1073

Bazon, M., 2012. *Should you switch to UglifyJS2?.* [Online]   
Available at: http://lisperator.net/blog/should-you-switch-to-uglifyjs2/

Bebenita, M., 2017. *WebAssembly is “30X” Faster than JavaScript.* [Online]   
Available at: https://medium.com/@mbebenita/webassembly-is-30x-faster-than-javascript-c71ea54d2f96  
[Accessed 28 March 2018].

Bhatia, S., 2014. *Browser Performance and Security After NPAPI.* [Online]   
Available at: http://www.cs.tufts.edu/comp/116/archive/fall2014/sbhatia.pdf  
[Accessed 9 May 2018].

Brehm, S., 2013. *Isomorphic JavaScript: The Future of Web Apps.* [Online]   
Available at: https://medium.com/airbnb-engineering/isomorphic-javascript-the-future-of-web-apps-10882b7a2ebc

Carnegie Mellon University, 2000. *Results of the Security in ActiveX Workshop.* [Online]   
Available at: https://resources.sei.cmu.edu/asset\_files/WhitePaper/2000\_019\_001\_52716.pdf

Chin, M., 2017. *Firefox Quantum vs. Google Chrome: Which browser is faster?.* [Online]   
Available at: https://mashable.com/2017/11/15/google-chrome-vs-firefox-quantum/#W8.XDyF\_yaqw  
[Accessed 23 April 2018].

Chrome Developers, Unknown. *File I/O.* [Online]   
Available at: https://developer.chrome.com/native-client/devguide/coding/file-io  
[Accessed 4 May 2018].

Clark, L., 2017. *A crash course in just-in-time (JIT) compilers.* [Online]   
Available at: https://hacks.mozilla.org/2017/02/a-crash-course-in-just-in-time-jit-compilers/

Clark, L., 2018. *Making WebAssembly even faster: Firefox’s new streaming and tiering compiler.* [Online]   
Available at: https://hacks.mozilla.org/2018/01/making-webassembly-even-faster-firefoxs-new-streaming-and-tiering-compiler/  
[Accessed 17 March 2018].

Coppock, M., 2018. *The best web browsers.* [Online]   
Available at: https://www.digitaltrends.com/computing/best-browser-internet-explorer-vs-chrome-vs-firefox-vs-safari-vs-edge/2/  
[Accessed 23 April 2017].

Dalziel, J., 2013. *The race for speed part 2: How JavaScript compilers work.* [Online]   
Available at: http://creativejs.com/2013/06/the-race-for-speed-part-2-how-javascript-compilers-work/index.html

Danford, J., 2018. *Using Web Workers.* [Online]   
Available at: https://developer.mozilla.org/en-US/docs/Web/API/Web\_Workers\_API/Using\_web\_workers

Delendik, Y., 2018. *Using the WebAssembly JavaScript API.* [Online]   
Available at: https://developer.mozilla.org/en-US/docs/WebAssembly/Using\_the\_JavaScript\_API

Doubleclick, 2016. *The Need for Mobile Speed.* [Online]   
Available at: https://www.doubleclickbygoogle.com/articles/mobile-speed-matters/  
[Accessed 27 April 2018].

Ecma International, Technical Committee 39 , 2017. *SIMD.js.* [Online]   
Available at: https://github.com/tc39/ecmascript\_simd  
[Accessed 22 April 2018].

Goldberg, I., 1996. *A Secure Environment for Untrusted Helper Applications (Confining the Wily Hacker).* [Online]   
Available at: https://www.usenix.org/legacy/publications/library/proceedings/sec96/full\_papers/goldberg/goldberg.pdf

Google Chrome Developers, 2016. *Planning for Performance: PRPL (Chrome Dev Summit 2016).* s.l.:s.n.

Google GmbH; Microsoft Inc. USA; Mozilla Inc. USA; Apple Inc, USA, 2017. *Bringing the Web up to Speed with WebAssembly.* [Online]   
Available at: https://dl.acm.org/citation.cfm?id=3062363

Herman, D., Wagner, L. & Zakai, A., 2014. *asm.js Working Draft - 18 August 2014.* [Online]   
Available at: http://asmjs.org/spec/latest/#standard-library

Landwerth, I., 2014. *The JIT finally proposed. JIT and SIMD are getting married..* [Online]   
Available at: https://blogs.msdn.microsoft.com/dotnet/2014/04/07/the-jit-finally-proposed-jit-and-simd-are-getting-married/

Laurens, T., 2013. *How the V8 engine works?.* [Online]   
Available at: http://thibaultlaurens.github.io/javascript/2013/04/29/how-the-v8-engine-works/

Low, D., 2005. *Protecting Java Code Via Code Obfuscation.* [Online]   
Available at: https://web.archive.org/web/20050308231232/http://www.cs.arizona.edu/~collberg/Research/Students/DouglasLow/obfuscation.html

Microsoft Corp, 1996. *Microsoft Announces ActiveX Technologies.* [Online]   
Available at: https://news.microsoft.com/1996/03/12/microsoft-announces-activex-technologies/

Microsoft, n.d. *Supporting enhanced protected mode (EPM).* [Online]   
Available at: https://msdn.microsoft.com/en-us/library/dn519894(v=vs.85).aspx

Missig, J. & Dickison, D., n.d. *Time perception with a concurrent variable-speed progress bar.* [Online]   
Available at: https://missig.org/julian/projects/jmissig%20Progress%20Bar%20Paper.pdf  
[Accessed 5 May 2018].

Netscape Communications Corporation, Sun Microsystems Inc., 1995. *Netscape and Sun Announce JavaScript, the Open, Cross-platform Object Scripting Language for Enterprise Networks and the Internet.* [Online]   
Available at: https://web.archive.org/web/20070916144913/http://wp.netscape.com/newsref/pr/newsrelease67.html

Paul, R., 2008. *Firefox to get massive JavaScript performance boost.* [Online]   
Available at: https://arstechnica.com/information-technology/2008/08/firefox-to-get-massive-javascript-performance-boost/

Q-Success, 2018. *Usage statistics and market share of jQuery for websites.* [Online]   
Available at: https://w3techs.com/technologies/details/js-jquery/all/all

Rauschmayer, A., 2014. *Speaking JavaScript.* 1st ed. March: O'Reilly.

Reinders, J., 2013. *Intel® AVX-512 Instructions.* [Online]   
Available at: https://software.intel.com/en-us/blogs/2013/avx-512-instructions

Rivest, R., 1997. *S-Expressions.* [Online]   
Available at: http://people.csail.mit.edu/rivest/Sexp.txt

Selakovic, M. & Pradel, M., 2015. *Performance Issues and Optimizations in JavaScript: An Empirical Study.* Darmstadt: s.n.

Shankland, S., 2009. *Mozilla releases Firefox 3.5 beta.* [Online]   
Available at: https://www.cnet.com/news/mozilla-releases-firefox-3-5-beta/

Shan, P., 2017. *Is JavaScript really interpreted or compiled language.* [Online]   
Available at: http://voidcanvas.com/is-javascript-really-interpreted-or-compiled-language/

Smith, B., 2017. *GitHub WebAssembly.* [Online]   
Available at: https://github.com/WebAssembly/bulk-memory-operations/issues/3#issuecomment-328233952  
[Accessed 6 April 2018].

W3C, 2012. *A vocabulary and associated APIs for HTML and XHTML.* [Online]   
Available at: https://www.w3.org/TR/2012/CR-html5-20121217/

Wagner, L., 2013. *Asm.js in Firefox Nightly.* [Online]   
Available at: https://blog.mozilla.org/luke/2013/03/21/asm-js-in-firefox-nightly/

Wagner, L., 2017. *WebAssembly consensus and end of Browser Preview.* [Online]   
Available at: https://lists.w3.org/Archives/Public/public-webassembly/2017Feb/0002.html

Wagner, L., 2018. *Understanding WebAssembly text format.* [Online]   
Available at: https://developer.mozilla.org/en-US/docs/WebAssembly/Understanding\_the\_text\_format

Walker, G., 2011. *HTML5 fundamentals, Part 3, The power of HTML5 APIs.* [Online]   
Available at: https://www.ibm.com/developerworks/web/library/wa-html5fundamentals3/index.html

Warren, T., 2017. *Adobe will finally kill Flash in 2020.* [Online]   
Available at: https://www.theverge.com/2017/7/25/16026236/adobe-flash-end-of-support-2020

WebAssembly Community Group, 2018. *WebAssembly Specification Introduction.* [Online]   
Available at: https://webassembly.github.io/spec/core/intro/introduction.html

WebAssembly, 2017. *Features to add after the MVP.* [Online]   
Available at: http://webassembly.org/docs/future-features/

WebAssembly, 2017. *Roadmap.* [Online]   
Available at: http://webassembly.org/roadmap/

Zakai, A., 2016. *A WebAssembly Toolchain Story.* [Online]   
Available at: http://kripken.github.io/talks/emwasm.html#/9  
[Accessed 28 March 2017].

Zakai, A., 2018. *Shrinking WebAssembly and JavaScript code sizes in Emscripten.* [Online]   
Available at: https://hacks.mozilla.org/2018/01/shrinking-webassembly-and-javascript-code-sizes-in-emscripten/  
[Accessed 28 March 2018].

Zakai, A., 2018. *WebAssembly (Emscripten).* [Online]   
Available at: https://github.com/kripken/emscripten/wiki/WebAssembly

Zakai, A. & Nyman, R., 2013. *Gap between asm.js and native performance gets even narrower with float32 optimizations.* [Online]   
Available at: https://hacks.mozilla.org/2013/12/gap-between-asm-js-and-native-performance-gets-even-narrower-with-float32-optimizations/

Zakai, A. & Wagner, L., 2015. *asm.js Speedups Everywhere.* [Online]   
Available at: https://hacks.mozilla.org/2015/03/asm-speedups-everywhere/

Zhu, L., 2017. *Architecture Overview.* [Online]   
Available at: https://github.com/Microsoft/ChakraCore/wiki/Architecture-Overview

# Figures

**Figure 1;** Sample ASM.js (Herman, et al., 2014)

**Figure 2;** Size comparison between compiled ASM.js and WebAssembly from; (Google GmbH; Microsoft Inc. USA; Mozilla Inc. USA; Apple Inc, USA, 2017)

**Figure 3;** PolyBenchC performance in WebAssembly relative to native speeds (Google GmbH; Microsoft Inc. USA; Mozilla Inc. USA; Apple Inc, USA, 2017)

**Figure 4;** Various optimization improvements over different versions of the V8 engine (Selakovic & Pradel, 2015)

**Figure 5;** Framerate comparison of ASM.js with and without Multithreading (MT) and SIMD

**Figure 6;** Results of the Multithreading test as a percentage of ASM.js

**Figure 7;** Comparison of Multi-threading (MT) and SIMD against ASM.js in the second Unreal Engine test

**Figure 8;** Increasing framerate of SIMD against the standard ASM.js test

**Figure 9;** Section of UE4 source code displaying Unreal Engine build options alongside equivalent Emscripten flags; Location - Engine\Source\Programs\UnrealBuildTool\HTML5\HTML5ToolChain.cs

**Figure 10;** Unreal Project 1 - Comparing Emscripten flags in ASM.js

**Figure 11;** Unreal Project 1 - Comparing Emscripten flags in WebAssembly

**Figure 12;** Unreal Project 2 - Comparing Emscripten flags in ASM.js

**Figure 13;** Unreal Project 2 - Comparing Emscripten flags in WebAssembly

**Figure 14;** Unreal Project 1 - ASM.js on Firefox and Chrome

**Figure 15;** Unreal Project 2 - ASM.js on Firefox and Chrome

**Figure 16;** Unreal Project 1 - WebAssembly on Firefox and Chrome

**Figure 17;** Unreal Project 2 - WebAssembly on Firefox and Chrome

**Figure 18;** Unreal Project 1 - WebAssembly vs ASM.js overview

**Figure 19;** Unreal Project 1 - Best WebAssembly vs ASM.js

**Figure 20;** Unreal Project 2 - Best WebAssembly vs ASM.js

**Figure 21;** Unreal Engine Project 1 - Firefox ASM.js vs WebAssembly

**Figure 22;** Unreal Engine Project 2 - Chrome ASM.js vs WebAssembly

**Figure 23;** Unreal Engine Project 1 - Firefox ASM.js vs WebAssembly

**Figure 24;** Unreal Engine Project 2 - Chrome ASM.js vs WebAssembly

**Figure 25;** Comparison of produced file sizes for each version of Unreal Project 1 tested

**Figure 26;** Comparison of produced file sizes for each version of Unreal project 2 tested

**Figure 27(a);** ASM.js vs Wasm in Unreal project 1

**Figure 27(b);** ASM.js vs Wasm in Unreal project 2

**Figure 28;** Compile times for the major tested versions

**Figure 29;** Compile times by browser and build version

**Figure 30;** Consecutive compile times after quick refresh

**Figure 31;** WebAssembly Chrome consecutive load times by device

**Figure 32;** Console Error from Google Chrome and multi-threading test version

**Figure 33;** Chrome message when loading ASM.js project

**Figure 34;** Warnings displayed on Chrome for both ASM.js and WebAssembly

**Figure 35;** Incorrectly rendered on phone (left), correctly rendered (right)

**Figure 36(a);** ASM.js Firefox error

**Figure 36(b);** WebAssembly project 2 Firefox error

# Appendices

## General Links

Data sheet: <http://kieranwarren.com/PerformanceResults.xlsx>

Project 1 builds: <http://kieranwarren.com/ue1/>

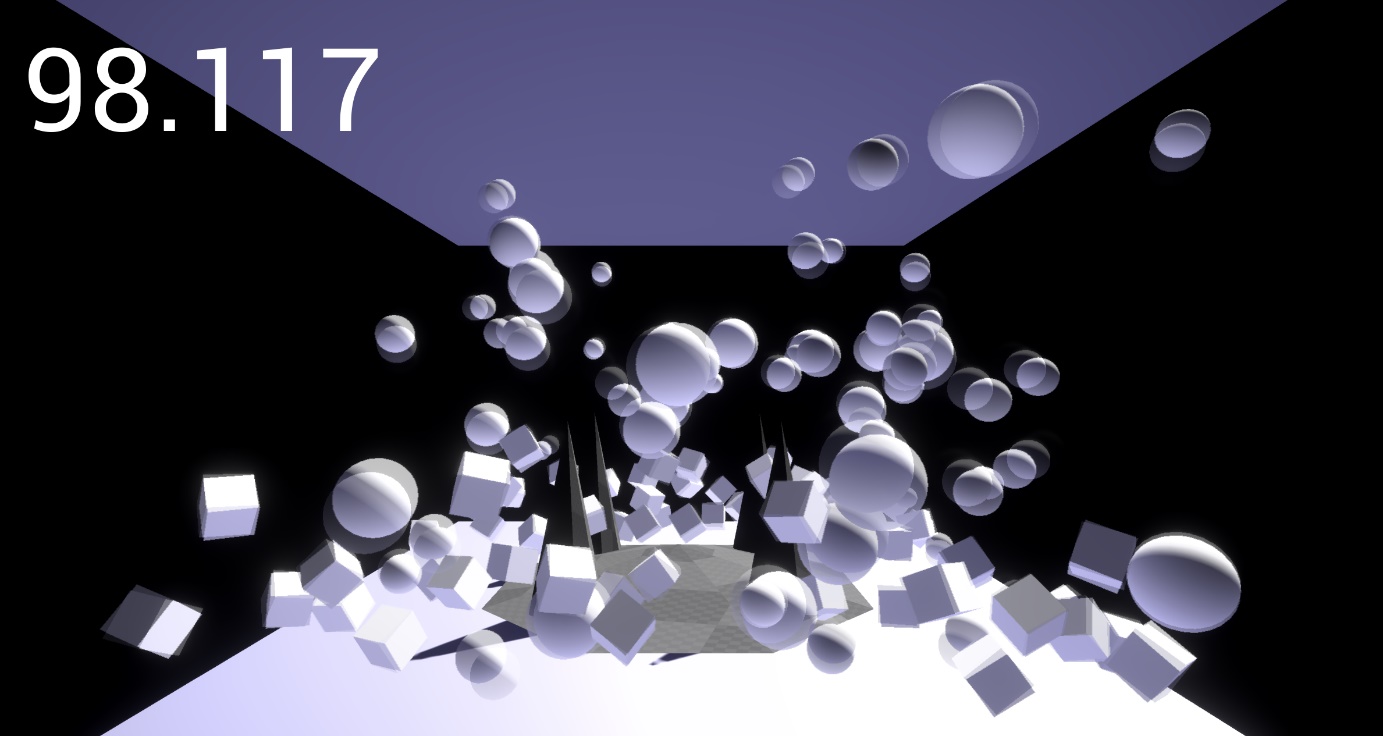
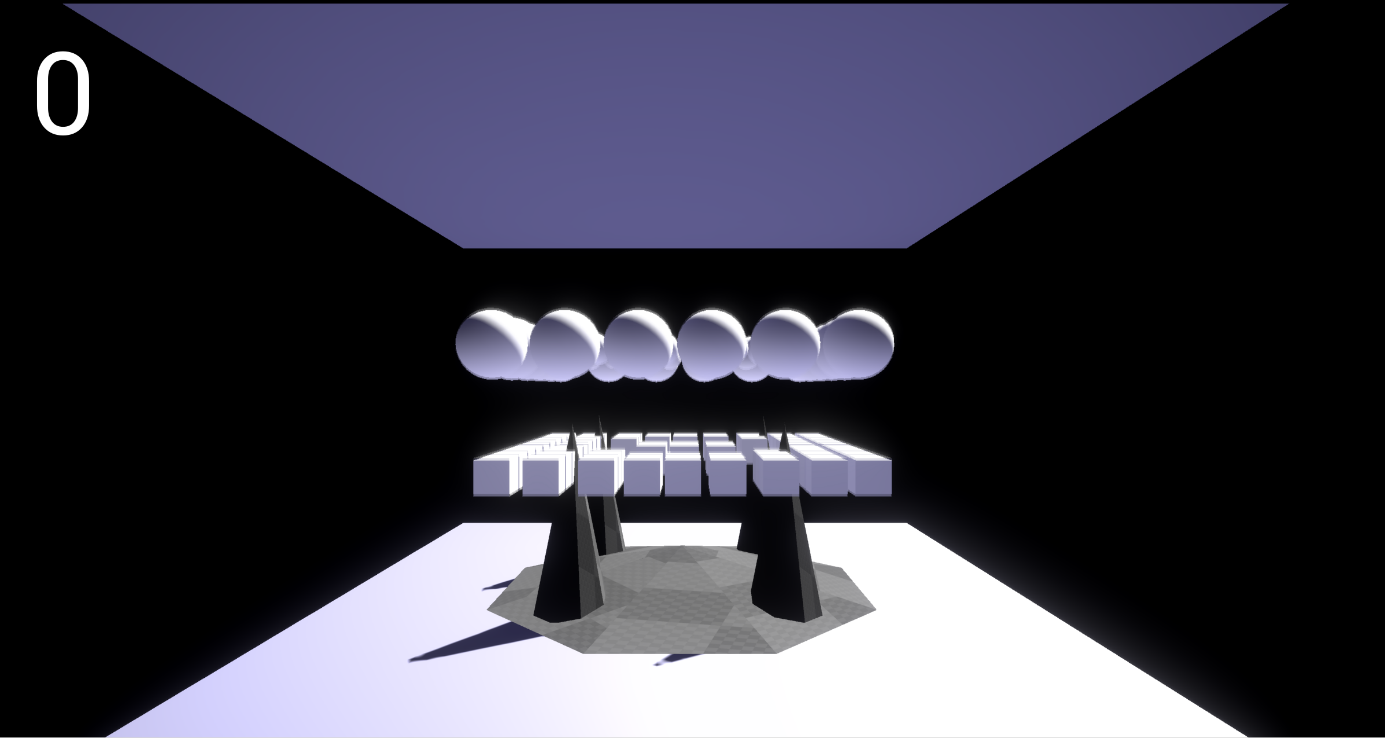
Project 2 builds: <http://kieranwarren.com/ue2/>

## Project 1 Screenshots

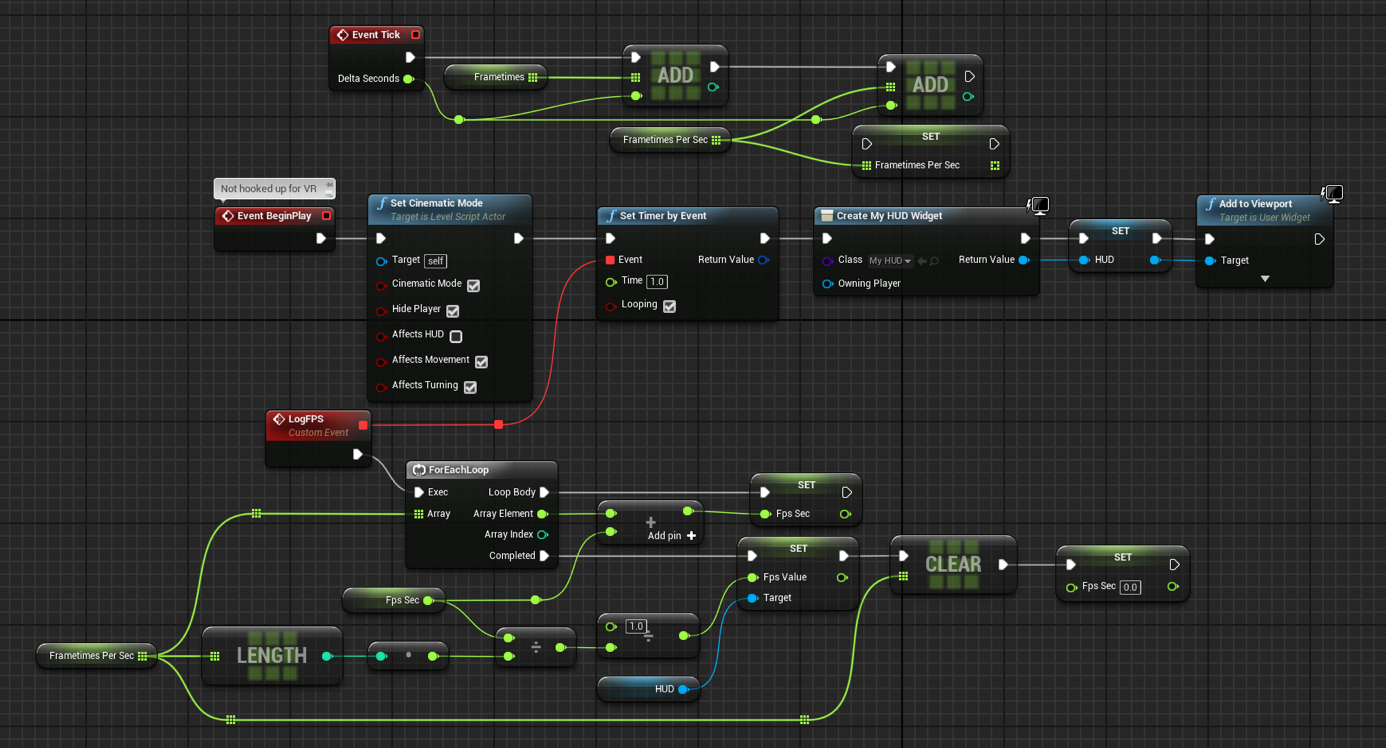




## Project 2 Screenshots



## Framerate recording script

Above is the script used to record performance in both projects. This Unreal Engine Blueprint is placed into the Level Blueprint and it will automatically log the framerate to the screen once per second.

## Data Sheets

Included alongside this document is the spreadsheet which includes all recorded data. This document is unformatted and so some tables and data may not be labelled in full detail.