

**BSc (Hons)** **Computer Games Technology**

**A JavaScript Runtime for Hardware Accelerated Applications**

**William Taylor**

**B00235610**

**22/04/2016**

**Supervisor:** **Paul Keir**

**Declaration**

This dissertation is submitted in partial fulfilment of the requirements for the degree of Computer Games Technology (Honours) in the University of the West of Scotland.

I declare that this dissertation embodies the results of my own work and that it has been composed by myself. Following normal academic conventions, I have made due acknowledgement to the work of others.

Name: WILLIAM TAYLOR

Signature:

Date:

**COMPUTING HONOURS PROJECT SPECIFICATION FORM**

**Project Title:** A JavaScript Runtime forHardware Accelerated Applications.

**Student:** William Taylor  **Banner ID:** B00235610

**Supervisor:** Paul Keir

**Moderator:** Mark Stansfield

**Outline of Project:**

The research is to develop a platform that allows GPU centric applications to be written in JavaScript. The platform’s goal is to provide compete bindings to industry standard GPU libraries (OpenCL & OpenGL) to allow developers to experiment and develop hardware accelerated applications in a dynamically typed and flexible language. The platform aims to expand the JavaScript ecosystem of runtimes and provide a workbench for those keen on the performance gains hardware acceleration can bring.

The research will highlight a number of key points. The first showing the speed of compilation and execution of JavaScript. The second showing how leveraging specialised hardware can accelerate traditional applications. Finally, the importance of accelerated programming and JavaScript to the technology sector.

**A Passable Project will:**

* Showcase a generalised GPU demonstration written in JavaScript.
* Will do an analysis of current GPU technologies in JavaScript and how they can be leveraged.

**A First Class Project will:**

* Develop and make available a platform that allows JavaScript developers to write generalised hardware accelerated applications.
* Showcase several generalised GPU demonstrations written in JavaScript.
* Do in depth research into future and current JavaScript technology which enables hardware acceleration.
* Research and demonstrate the advantages of JavaScript over other dynamic languages (e.g. Python) in developing hardware accelerated applications.

**Reading List:**

1. OpenGL Programming Guide: The Official Guide to Learning OpenGL, Versions 4.3
2. Programming 3D applications with HTML5 and WebGL
3. Heterogeneous computing with OpenCL
4. OpenCL Programming Guide

**Resources Required:**

Visual Studio, OpenCL & OpenGL enabled hardware, Chrome’s V8 JavaScript JIT compiler, Git + Github.

**Marking Scheme: Marks**

Introduction 10

Area Overview 15

Requirements and Design 10

Development 30

Project Demonstrations 20

Critical Self-Appraisal 5

Conclusions and Recommendations 10

**Signed:**

**Student Supervisor Moderator Year Leader**

**IMPORTANT: *By signing this form all signatories are confirming that any potential ethical issues have been considered and necessary actions undertaken and that Mark Stansfield (Module Coordinator) and Malcolm Crowe (Chair of School Ethics Committee) have been informed of any potential ethical issues relating to this proposed Hons Project.***

# Acknowledgements

**Paul Keir**

I would first of like to thank my supervisor Paul Keir for his advice and patience. Regular meetings held in both trimesters helped keep me on track and helped set out numerous ways in which to improve my work. His experience with GPUs and C++ also helped my project realise its full potential. Thanks to his assistance and advice, I could complete this project to a standard much better than it would have been had I done it alone and I could attain new skills in research and I am sure they will help me if I choose to do further study in the future or research based work in my professional career.

**Mark Stansfield**

I would also like to thank my moderator Mark Stansfield whose lectures helped set out the work required and he could answer any of my many tedious questions. He also helped me set out how my more technical project would alter the structure of certain coursework like my interim report and honours project and it’s thanks to this advice that I could submit both with a level of polish that I am sure the markers were happy with.

A JavaScript Runtime for Hardware Accelerated Applications

**William Taylor**

**April 22, 2016**

To investigate and explore how we can make hardware acceleration experimentation more accessible, a prototype runtime was developed on top of Google’s V8 JavaScript compiler and popular open source frameworks. In tandem, demonstrations were built using the runtime to investigate the advantages of our approach. Reasons for creating the platform are numerous, they include making the technology more accessible and aiding pre-existing efforts to find new ways of leveraging the hardware. The development of the runtime is discussed in detail as are revisions to the JavaScript standard that allow it to be used for general-purpose scripting. The runtime demonstrates that an all in one platform can streamline development and make general purpose computation on graphics hardware easier for both novices and experts.

**Keywords:** GPUs, GPGPU, Google, V8, JavaScript, Platform, Learning, Development, Experimentation

Table of Contents

[Acknowledgements 5](#_Toc476050794)

[1.0 Introduction 9](#_Toc476050795)

[2.0 Background Information 11](#_Toc476050796)

[2.1 GPUs 11](#_Toc476050797)

[2.1.1 Hardware 11](#_Toc476050798)

[2.1.2 Manufacturers 12](#_Toc476050799)

[2.1.3 Software 12](#_Toc476050800)

[2.2 JavaScript 13](#_Toc476050801)

[2.2.1 Design 14](#_Toc476050802)

[2.2.2 Typed Arrays 15](#_Toc476050803)

[2.2.2 ArrayBuffer 15](#_Toc476050804)

[2.2.3 ECMAScript 2015 16](#_Toc476050805)

[2.2.4 Module Systems 17](#_Toc476050806)

[2.3 V8 18](#_Toc476050807)

[2.3.1 Property Access 18](#_Toc476050808)

[2.3.2 Machine Code Generation 19](#_Toc476050809)

[2.3.3 Garbage Collection 19](#_Toc476050810)

[2.3.2 Key Concepts 19](#_Toc476050811)

[2.3.3 V8 Usages 20](#_Toc476050812)

[3.0 Software Design 21](#_Toc476050813)

[3.1 Requirements 21](#_Toc476050814)

[3.2 Architecture 21](#_Toc476050815)

[3.3 Interface 21](#_Toc476050816)

[4.0 Development 22](#_Toc476050817)

[4.1 Embedding V8 22](#_Toc476050818)

[4.2 Module System 22](#_Toc476050819)

[4.3 Common Libraries 22](#_Toc476050820)

[4.3.1 Console module 22](#_Toc476050821)

[4.3.2 Datetime module 22](#_Toc476050822)

[4.3.3 System module 23](#_Toc476050823)

[4.3.4 Http module 23](#_Toc476050824)

[4.3.5 File module 23](#_Toc476050825)

[4.3.5 Display module 24](#_Toc476050826)

[4.3.6 Maths module 24](#_Toc476050827)

[4.4 GPU Bindings 24](#_Toc476050828)

[4.4.1 GL module 24](#_Toc476050829)

[4.4.1 CL module 25](#_Toc476050830)

[5.0 Testing 26](#_Toc476050831)

[5.1 OpenGL Demo’s 26](#_Toc476050832)

[5.1.1 3D Cubes 26](#_Toc476050833)

[5.1.2 3D Terrain 27](#_Toc476050834)

[5.1.3 3D Lighting 28](#_Toc476050835)

[5.2 OpenCL Demo’s 29](#_Toc476050836)

[5.2.1 Grayscale Demo 29](#_Toc476050837)

[5.2.2 Sobel Filter Demo 29](#_Toc476050838)

[5.2.3 Matrix Demo 29](#_Toc476050839)

[6.0 Future Work 30](#_Toc476050840)

[6.1 DirectX 30](#_Toc476050841)

[6.2 CUDA 30](#_Toc476050842)

[6.3 Applications 31](#_Toc476050843)

[6.4 Framework vs Platform 31](#_Toc476050844)

[6.5 Alternative Languages 31](#_Toc476050845)

[6.5 User Interface 31](#_Toc476050846)

[7.0 Critical Appraisal 32](#_Toc476050847)

[8.0 Conclusion 33](#_Toc476050848)

# 1.0 Introduction

In recent history, there has been a seismic shift in technology. Processors have stopped getting faster at an exponential rate. Increasing the clock speed of processors has now been abandoned in favour of multicore processors (David Geer, 2005). Due to 3D and high resolution media increasing in popularity we now see Graphics Processing Units (GPUs) integrated into modern computers by default (Intelcom, 2016, Amdcom, 2016). Easily learning and experimenting with this new technology is of great importance if we are to see general purpose computing on graphics processing units (GPGPU) more widely adopted. This is the topic for this honours project where we will explore the possibility of an integrated platform for GPU technique experimentation and development. Specifically, we will look at a dedicated platform that leverages the popular scripting language JavaScript to provide a reliable and flexible tool to those learning how to leverage GPUs for the first time and to those who wish to develop their own GPU techniques in an easier manner. By doing so we hope to make this vital underutilized technology more accessible to new comers and to aid in the development of future optimizations that will unlock more compute power hidden in today’s computers.

There are currently several problems we have identified in experimenting with GPUs and writing GPU based applications that limit a programmer’s ability to get stuck into this exciting piece of technology. One is that at the time of writing this paper there is currently no easy to use integrated environment to experiment with various GPU APIs such as OpenCL and OpenGL. While one could argue that the Web provides an integrated environment through WebCL and WebGL which are web equivalents of OpenCL and OpenGL, I would strongly disagree for several reasons. The first being that due to the requirement of a browser being portable it is unable to provide support for GPU technologies designed for specific hardware such as CUDA or specific APIs locked into a single operating system such as DirectX. The other reason for disagreeing is because the browser has a security model that disables local access to the computer making the loading of data such as complex 3D geometric models overbearing and complicated. So, while the Web may provide a way to write GPU programs it is more for web developers to speed up their applications and not to provide a toolset to make GPU programming as easy as possible. Finally, the web itself is an open standard with not all browser conforming to that standard. Therefore, using the latest version of WebCL and WebGL may not always be an option. For instance, WebGL 2.0 only has experimental support in Chrome one of the world’s most popular browsers. So, using the web as a platform would be too limiting and too restrained for any enthusiast.

Another issue identified is the frustration of using the native bindings to the APIs from C++. As C++, doesn’t provide native support through the standard template library (STL) for images, models, input and windows it leads to a lot of extra work with additional libraries and APIs rather than letting you get on with your GPU technique development. The result is a lot of boilerplate before you get to writing what you will be experimenting with and that is the GPU programs themselves whether that is kernels in OpenCL or shaders in OpenGL. This issue is after you install various SDKs and tools to get access to these APIs, making it not only difficult when you start writing your program but difficult to get started in the first place. Additionally, most libraries focus on a sole area, for example GLM focuses solely on mathematics and GLUT on accessing windowing systems. There is no one tool that combines all under one package, all are widely different in terms of API design, coding style and support across various platforms.

The proposed solution is this project where we aim to build an all in one platform suitable for GPGPU experimentation, learning and prototyping. We will develop a JavaScript runtime which aims to provide a bulk of features out the box to reduce the learning curve required and provide native bindings to popular industry standard APIs that are suitable to both novices and experts. The platform should be easy to install and easy to use, skipping lengthy and numerous SDK installations in favour of a onetime install platform that provides everything required out of the box. The development of the platform and research should highlight several key points. The first showing the speed of compilation and execution of JavaScript and how it can be utilized as a generic scripting language for numerous environments. The second showing how leveraging specialised hardware which is more common than ever in today’s world can accelerate traditional applications. Finally, by showing the importance and relevance of both modern JavaScript as a general scripting language and accelerated programming for being the tool the programmers must leverage if we are to see more performant software.

# 2.0 Background Information

## 2.1 GPUs

The term Graphics Processing Unit (GPUs) was coined by NVidia when they released their graphics chip called the GeForce 256 (Nvidiacom, 2016). The origin of the modern GPU started in the 1970s where arcade manufactures to cut costs built systems with custom video chips to power the display. Today GPUs are an abundance, they are present in most computers including consoles, desktops, laptops, tablets, and mobile phones albeit in different forms. In the following section I will be providing a review of this key technology.

Research (Yang et al, 2008) took bread and butter computer vision algorithms and compared their performance when processed across a CPU and GPU. With a histogram, they saw a 44x speed up when computed on the GPU. When it came to edge detection they saw a 200x speed up. Additionally, research (Teodoro et al, 2009) found that optimising a histopathology application resulted in a speed factor increase of between 19x to 40x in their tests. In computationally expensive tasks we can see GPUs can provide unseen speed ups in expensive computations. We can also see how a workbench could be advantageous to experiment and test such optimisations.

### 2.1.1 Hardware

On a hardware level GPUs are distinct. A Central Processing Unit (CPU) typically consists of a couple of cores, the most common being dual core processors and quad core processors. This contrasts with GPUs which commonly have more than 100 cores making them great at processing parallel workloads (Nvidiacom, 2009). GPUs are also distinct when it comes to memory. A traditional CPU will have multiple layers of cache in which to store data it will process. The cache is traditionally very small whereas GPUs have dedicated memory which was designed specifically for the GPU and often has a higher bandwidth than traditional system RAM. The ability for GPUs to accelerate computation workloads has now expanded the hardware to not only be used for 3D rendering but also for scientific research, data analysis, financial modelling, image processing and gas exploration.

GPUs to become more mainstream have been shrunk and extruded into different form factors to suit the computers they would be integrated into. Dedicated graphics cards are found in high end desktops, laptops, and workstations. They are installed into these computers via an expansion slot and are often the most powerful and expensive cards as they do not need to meet harsh size restraints or power limits. Traditionally integrated graphics were chips installed on the motherboard, however in 2010 Intel integrated the graphics chip onto the CPU die setting the stage for modern integrated graphics (Intelcom, 2016). The result was better media performance by default for standard CPUs as there was an increased demand for CPUs to be capable of moderate graphics tasks such as HD media playback and light 3D rendering. Intel was not the only CPU manufacture to do this. AMD also pioneered the technology with their Accelerated Processing Unit (APU) technology in 2011 which was designed to provide better 3D and media performance in small form factor computers such as laptops and game consoles.

### 2.1.2 Manufacturers

Two major chip manufacturers AMD and NVidia dominate the dedicated graphics market. There is a consensus that NVidia today holds a majority share of the market, this is backed up both by Steam hardware reports (Steampoweredcom, 2016) and research undertaken at John Peddie Research. Although NVidia dominates the market AMD is still an influential player. The latest generation consoles, the Xbox One and PlayStation 4 are powered by AMD graphics cards. What’s more their Mantle API (Amdcom, 2016) was the starting point for the new API for both compute and graphics Vulkan (Khronosorg, 2016) which aims to supersede OpenGL and OpenCL entirely. AMD entered the graphics card market with the acquisition of ATI in 2006 and has been a keen player ever since. NVidia has its own accomplishments with its own compute API supported on its cards known as CUDA which is a direct competitor to OpenCL. NVidia cards are commonly the graphics card vendor of choice when it comes to laptops and general desktops as well.

Modern integrated graphics are now integrated onto the CPU die, making this technology completely dominated by the two major CPU manufacturers Intel and AMD. Intel added integrated graphics into their CPUs in 2010 with the launch of their Westmere microarchitecture. AMD arrived later with APUs based on their K10 architecture, that while not the first provided much better performance out of the box.

### 2.1.3 Software

Because GPUs are specialised hardware they have been traditionally accessed through industry approved API standards like OpenGL and OpenCL. Over the years, the number of APIs available have expanded as GPUs have evolved. The newest APIs include Vulcan (Khronosorg, 2016), Metal (Applecom, 2016) and CUDA (Nvidiacom, 2016). GPUs are traditionally used for parallel computation and advanced 3D rendering and in the following section I will be summarising the technology used most today to accomplish rendering and computation.

3D Rendering has been traditionally accomplished through either DirectX or OpenGL. DirectX is a set of Windows APIs for multimedia applications. DirectX’s key component is Direct3D which is a direct competitor to OpenGL and allows developers to write 3D applications. DirectX unlike OpenGL isn’t cross platform, you will only find it on Windows, one of its key faults. Another key differential is that DirectX isn’t backwards compatible unlike its competitor. OpenGL stands for Open Graphics Library; it is a cross platform API for 3D rendering. Unlike DirectX, OpenGL is only concerned with rendering and isn’t a set of APIs but rather one API for rendering only. OpenGL is backwards compatible and uniquely has multiple versions which has seen it expand onto other platforms. OpenGL ES brought the API to embedded systems, and WebGL brought hardware accelerated rendering to the Web. In any case as it is one of the few platform independent graphics APIs it is still used today, most notably on Linux and Mac.

The key element to DirectX and OpenGL are programs called shaders. To provide more control of the rendering both APIs have programmable sections the programmer can use to dictate how data is rendered on screen. In DirectX, such shaders are written in a language called HLSL or High Level Shading Language. In OpenGL, these shaders are written in a language called GLSL or OpenGL Shading Language. An example of a simple orthographic OpenGL shader can be found in Figure 1.



Figure 1: Typical 2D OpenGL shader

On the compute side of GPU APIs, we have CUDA and OpenCL. Apple originally proposed the Open Compute Library known as OpenCL to allow developers to take advantage of GPUs on their platform. They submitted it to the Kronos Group and it soon became an industry standard. Where OpenCL is different from CUDA is the range of devices it works on. OpenCL can run on any heterogeneous system and is not bound to a single operating system like DirectX or hardware manufacturer like CUDA. CUDA on the other hand will only run on NVidia hardware. Research (Karimi, K, 2016) found CUDA to perform better than OpenCL, however CUDAs inability to work across hardware from different manufacturers is certainly its biggest downfall yet it reserves its strength as the best performing API in the market. Like the above-mentioned rendering APIs CUDA and OpenCL have programmable elements called kernels where the programmer can dictate how data is transformed. In CUDA such kernels are written in CUDA C which is raw C/C++ with extensions allowing one to execute code on the GPU. In OpenCL kernels are written in OpenCL C which like CUDA mirrors the C/C++ language and adds extensions to fit the device it will run on. In Figure 2 you can see an OpenCL C kernel which performs a simple vector addition.



Figure 2: OpenCL kernel which performs a vector additional

## 2.2 JavaScript

We chose JavaScript as the language for the platform for various reasons. The first is its speed. JavaScript has benefited from a large amount of investment in compiler development with most browser vendors now opting for Just in Time (JIT) compilers over traditional interpreters for JavaScript execution. The result is a tenfold increase in JavaScript speed making the language more suitable for high performance applications. Second JavaScript is a very popular language, in their yearly survey StackOverflow found JavaScript to be the most popular technology (Stack overflow blog, 2016) by a large mile, so using it for the platform would be advantageous as the language is popular with many developers. Finally, JavaScript has had a new recent standard ECMAScript 2015 which has sought to remove previous issues and present JavaScript as a clear concise general purpose scripting language rather than a language for document object model manipulation in the browser.

### 2.2.1 Design

Severance, C., 2012. Javascript: Designing a language in 10 days. *Computer*, *45*(2), pp.7-8.

Richards, G., Lebresne, S., Burg, B. and Vitek, J., 2010, June. An analysis of the dynamic behavior of JavaScript programs. In *ACM Sigplan Notices* (Vol. 45, No. 6, pp. 1-12). ACM.

Javascript, A.P.I., 2007. The web is everywhere. *IEEE Communications Magazine*.

Ocariza, F., Bajaj, K., Pattabiraman, K. and Mesbah, A., 2013, October. An empirical study of client-side JavaScript bugs. In *Empirical Software Engineering and Measurement, 2013 ACM/IEEE International Symposium on* (pp. 55-64). IEEE.

Tilkov, S. and Vinoski, S., 2010. Node. js: Using JavaScript to build high-performance network programs. *IEEE Internet Computing*, *14*(6), pp.80-83.

### 2.2.2 Typed Arrays

Recent revisions of the JavaScript standard have added support for objects designed to make low level programming possible (Mozillaorg, 2016). I will summarise the most ground breaking set of objects known as TypedArray objects as it now allows JavaScript to work with binary data directly. Typed Arrays were added in the JavaScript standard ECMAScript 2015 as the language lacked any ability to work with low level data and the typed array specification was an answer to this issue. They allow JavaScript to have types that represent raw C data types such as char and float. I will now cover these objects.

### 2.2.2 ArrayBuffer

ArrayBuffer is the base type for every Typed Array object and it just represents a stream of binary data. Look at the Figure 3 we can take the struct person and represent it in memory in JavaScript with the following ArrayBuffer shown in Figure 4. At this point the JavaScript example and the C++ example have access to the same set of data and the same number of bytes in memory. This is an important step forward in JavaScript as it allows us to allocate and control bytes which was a concept absent from JavaScript till this point.



Figure 3: basic struct example



Figure 4: Figure 3 struct represented in modern JavaScript

Following the base type ArrayBuffer you can now also represent arrays of bytes with greater precision than before. JavaScript numbers are defined in the standard as 64-bit double precision numbers. This limits control but with TypedArrays you can now control a greater range of integral types. Consider the following C++ arrays shown in Figure 5.



Figure 5: C style arrays­­­­

Previously it was impossible to have variables in JavaScript that natively mimicked these due to JavaScript having one type for all types of numbers. But due to the addition of TypedArrays this is no longer the case as can be seen in Figure 6.



Figure 6: Figure 5 arrays represented in JavaScript

In short, the addition of these types to JavaScript better enables the language to interact with low level data structures and binary data. As such when building the platform using these objects has been prioritised as it stops the need to convert JavaScript data types to the data types found in C/C++.

### 2.2.3 ECMAScript 2015

JavaScript’s latest standard as mentioned above is known as ECMAScript 2015, but it was more than just an update, it was a complete overhaul of the language. The specification brought in new features to aid in solving problems found in older versions of the language. The standard brought about new declarations for variables the language. The two new keywords let and const aimed to solve the problem of variable scoping and reinitializing found in today’s JavaScript language.

// Code sample of let and const

The standard also brought about shortcuts to enable a more flexible language. Template literals were added to make string interpolation and construction not only easier but more readable as well.

// template literal example

Finally, a huge new feature they added new keywords for dealing with JavaScript’s complicated prototype based object model. It is no secret that the current system is baffling for new users. The new standard adds new keywords as syntactic sugar for the pre-existing prototype system.

// Code sample of class extends

### 2.2.4 Module Systems

The original JavaScript standard had no module system for the language. With web applications and JavaScript being used more two main module standards were devised to add this missing piece of the puzzle from the specification. These modules systems go by name of CommonJS and AMD (Asynchronous Module Definition). As part of the platform, once V8 was successfully embedded and the source code written in a file was parsed and executed the CommonJS standard was implemented to allow users of the platform to write code in a modular fashion and so built in libraries for common tasks could also be served through this module system.

#### 2.2.4.1 CommonJS

The CommonJS standard (Commonjsorg, 2016) specifies a contract for modules and how they should be handled. It is the system that Node uses for its module system, in fact node is what made it popualar. The specification lays down a set of requirements that if met result in a system that conforms to the standard. In the runtime, there should be a function called require which accepts a module identifier. The require function itself returns the exported contents of the foreign module. If, however the given module identifier does not lead to a valid module an error must be thrown with an acceptable message detailing why. In a module, which is normally a standalone JavaScript file there must be a variable called require which follows the above definition. There must also be a variable called exports which is an object that the module may add its API to as it executes. Finally, there must be a free variable module that is an object. This module object must have an id property and that module id value if passed to require should return itself. A module identifier is a string delimited by forward slashes. If a module id has no filename extension “.js” is added by default. The module identifier is relative if it starts with “.”. Finally, relative identifiers are resolved relative to the call to require.

#### 2.2.4.1 AMD

The other module system devised which is not used in our platform is the AMD system. AMD is found mostly on the client side of JavaScript programming, with the above mentioned CommonJS runtime used on the server side. To be AMD compliant the following rules must be met. In the runtime, there should be a single function define that is available as either a local or global variable. The first argument, id, specifies the id of the module being defined. This argument is optional, and if it not present, the module id should default to the id of the module that the loader was requesting for the given response script. When present, the module id must be an absolute id.

The second argument, dependencies, is an array of the dependencies that are required by the module that is being defined. The dependencies must be resolved prior to execution of the module factory function, and the resolved values should be passed as arguments to the factory function with argument positions corresponding to index in the dependencies array. The dependencies ids may be relative ids, and should be resolved relative the module being defined. This specification defines three special dependency names that have a distinct resolution. If the value of "require", "exports", or "module" appear in the dependency list, the argument should be resolved to the corresponding free variable as defined by the CommonJS modules specification. This argument is optional. If omitted, it should default to ["require", "exports", "module"]. However, if the factory function's arity (length property) is less than 3, than the loader may choose to only call the factory with the number of arguments corresponding to the function's arity or length.

The third argument, factory, is a function that should be executed to instantiate the module or an object. If the factory is a function it should only be executed once. If the factory argument is an object, that object should be assigned as the exported value of the module. If the factory function returns a value (an object, function, or any value that coerces to true), and then that value should be assigned as the exported value for the module.

## 2.3 V8

In 2008 Google set the benchmark for JavaScript compilers. They created a new JavaScript JIT compiler, V8 from the ground up to dramatically improve JavaScript execution speed. Browsers at the time used JavaScript interpreters instead. Internally they built a benchmark called V8 bench and measured performance increases overtime. As you can see in Figure 7 each subsequent revision of Chrome which in turn has a new version of V8 saw massive gains in JavaScript performance. This started the JavaScript compiler competition which saw all major JavaScript implementers drop their interpreters in favour of a JIT compiler in the hope that faster JavaScript would lead to a faster browser and better web experience. The key difference between an interpreter and a compiler is how the program is built and executed. Where an interpreter would typically execute one statement at a time, a compiler would translate the entire program into machine code ready to execute. V8 implements ECMAScript as specified in ECMA-262, 5th edition, commonly referred as ECMAScript 2015 and runs on Windows, Mac OS X, and Linux systems. V8 enables any C++ application to expose its own objects and functions to JavaScript code. It's up to the developer to decide on the objects and functions exposed to JavaScript. There are many applications that use V8 already including Adobe Flash, the Dashboard Widgets in Apple's Mac OS X and Yahoo Widgets. V8 has three central pillars embedded in its design to achieve fast JavaScript execution.

### 2.3.1 Property Access

JavaScript as mentioned before is a dynamic programming language. Objects are constructed dynamically on the fly without the need of type declarations as found in other popular languages like C++, C# and Java. This means any JavaScript compiler would need to deal with object property changes frequently. Most JavaScript engines use dictionary based data structures as storage for properties bound to a specific object, with a dynamic lookup being needed to resolve an objects property during runtime. This is a key feature of why accessing instance variables is slower in JavaScript compared to other languages. In these languages locating a variables location in memory is calculated by the fixed offset determined by the compiler due to a fixed layout for a memory structure. Access requires a simple memory load and store normally one instruction.

V8 to solve this abandons dynamic look ups in favour of a concept it refers to as hidden classes. Hidden classes are behind the scenes and are never exposed to the programmer. This idea is that class structures are generated at runtime whenever a property is changed and by doing so a dynamic lookup is not needed when accessing a property as we can have used the existing hidden class layout to determine were the property resides in memory. This enables fast access to properties on a given object without sacrificing the ability to dynamically add new properties to objects during runtime.

### 2.3.2 Machine Code Generation

Another key feature of V8’s performance is how it executes it. Interpreters were previously the norm however V8 directly compiles JavaScript source code into machine when first executed. There is no intermediate byte code as seen in Java, there code is directly compiled.

### 2.3.3 Garbage Collection

JavaScript relies on garbage collection for memory clean-up. Even though the delete keyword is found in JavaScript it is used for deleting properties from an object and not for deallocating memory as found in C++. V8 ensures a fast garbage collector by ensuring object allocation is fast, that garbage collection pauses are short and by ensuring there is no memory fragmentation. These features stop long pauses in a JavaScript application are no longer prevalent by processing only part of the object heap in most garbage collection cycles removing long pauses. V8 stores the object heap in two parts. The new space where objects are created and the old space where objects surviving a garbage collection cycle are promoted. When an object is promoted in a garbage collection cycle, V8 updates all pointers to the object which avoids falsely identifying objects as pointers which can result in memory leaks.

### 2.3.2 Key Concepts

There are several concepts in V8 that one must understand if they are to use it as their own for JavaScript compilation. An isolate in V8 is defined as a virtual machine (VM) instance with its own heap. The idea is that an application should be able to spin up multiple VM instances from within a single application. You create an isolate like so using the C++ V8 API. This is the first object we create in our runtime to launch V8 and prepare for JavaScript execution. In Figure 8 you can see this object being created.

Handles are pointers to objects exposed to JavaScript. All V8 objects are accessed using handles and are needed as JavaScript uses a garbage collector and objects cannot be released until all handles are released. Handles come in many different varieties the most common one being Local which is just a stack allocated handle to the value stored in V8. In Figure 9 you can see a handle being created.

Scopes are containers for a sequence of handles. They allow handles to be released on a function by function basis rather than by the primary scope. In Figure 10 all handles allocated in the current scope will be deleted when the HandleScope is deleted. Note to construct a HandleScope object you must pass the VM instance that the HandleScope will be run on. The GetCurrent function returns the current isolate.

A context is an execution environment that allows separate unrelated JavaScript code to run in a single instance of V8. Whenever you start up a V8 execution environment you must specify the context in which it runs. The contexts are used so you can have multiple JavaScript apps running at the same time, this is used to great effect in Chrome, where tabs have their own JavaScript context. Creating a context can be seen in Figure 11.

### 2.3.3 V8 Usages

Fast JavaScript execution did not go unnoticed. JavaScript can now be found in many environments other than the web.

You can now write server side applications in JavaScript with Node.js (Nodejs foundation, 2016) which uses V8. The argument made for the runtime was that mode server frameworks block due to multithreading issues and that by providing an asynchronous environment you could provide a fast server as it adopts a non-blocking model. Another key feature highlight was the ability to write end to end a web application in the same language which was a considerable advantage to full stack web developers. Today Node is massively popular and its package manager NPM is the most popular package manager in the world. But all this has been enabled by the fast JavaScript execution that V8 provides.

Finally, through open source projects such as Electron (Atomio, 2016) you can now write native desktop applications as well. Electron takes the Chromium browser and replaces its JavaScript runtime with Node.js. While this may seem strange what, it allows JavaScript written in the webpage to then access the file system which is normally blocked due to the browsers security model.

Because V8 is natively part of the open source browser Chromium, V8 has manifested itself into various browsers. These include Opera, Google Chrome, Vivaldi and host of other browsers that have be built for specific use cases from Chromium. This has greatly expanded V8’s usages but also includes a faster JavaScript compiler into various browsers and not just Google’s offering.

You can write full 3D games with the Unity game engine (Unity3dcom, 2016) which uses it as its scripting language. One think to note here is that the JavaScript found in Unity, referred to as Unity Script isn’t vanilla JavaScript as most people know it. It has a wave of additional features including no with keyword that brings all variables into scope (discussed later in this report) and a this keyword that refers to the current instance rather than caller of the method.

# 3.0 Software Design

## 3.1 Requirements

## 3.2 Architecture

## 3.3 Interface

# 4.0 Development

## 4.1 Embedding V8

The first step was to get V8 downloaded and linked inside our C++ application. That was surprisingly difficult as V8 is not a small source project. As it was such a big project it had a lot of custom build tools and technologies that were needed to build V8 from source. V8’s source can be built in multiple different ways, either with the GN meta build system or GYP meta build system. V8’s repository is also built on top of Google’s depot tools which must be installed as well and most of these technologies are poorly documented. After a large amount of time had been spent we managed to output V8 as a static library file which could now be linked to in a C++ application. Once there we followed the embedders guide which explains key concepts in V8.

## 4.2 Module System

Once V8 was successfully embedded and the source code written in a file was parsed and executed I looked at implementing a module system that would allow users to write modular code when using the runtime. I solved this by implementing the CommonJS standard which is used in the Node runtime as well. The CommonJS standard (Commonjsorg, 2016) specifies a contract for modules and how they should be handled.

## 4.3 Common Libraries

Once we had V8 embedded and a CommonJS module system implemented we wrote some basic libraries or common libraries for common tasks.

### 4.3.1 Console module

We provided a console module allowing users to write information to a console and read input from it as well. This is based on the console object found in most browsers for familiarity (Mozillaorg, 2016). A basic example of this module’s functionality can be seen in Figure 12.



Figure XX: Console API example

### 4.3.2 Datetime module

We also provided a date time module for managing time. These methods are based on the time browser specification so it’s familiar to web developers (W3org, 2016). We also added an additional pause method which mirrors the Win32 API Sleep function. Figure 13 shows off some of the functions found in this module.



Figure XX: Datetime API example

### 4.3.3 System module

To provide information on the system we provided a system module. While we don’t envisage this being part of an application we feel that a platform should provide useful information and this does that providing access to OS information, battery details, instruction sets and hardware information. Figure 14 shows the information available from this module.



Figure XX: System API example

### 4.3.4 Http module

JavaScript and JSON are prolific when it comes to services and data online. So, to provide access to content online, for instance JSON files we added a http module that allows the user to get content online which can then be streamed directly into an application. An example of this can be found in Figure 15.



Figure XX: Http API example

### 4.3.5 File module

Of course, a big feature needed for OpenCL and OpenGL is reading data off disk so we added a file system module which provides the ability to read text files, JSON files and images. Once read these objects can be passed directly to OpenCL and OpenGL for processing. Figure 16 gives a basic usage example.



Figure XX: Fs API example

### 4.3.5 Display module

A key component of any OpenGL demo is the ability to render your graphics to a window. The display module was built as the one stop shop to handle windows, message boxes, and basic components available on desktop operating systems. In Figure 17 you can see an example of how to open a window and enable an OpenGL context.



Figure XX: Display API example

### 4.3.6 Maths module

## 4.4 GPU Bindings

The core modules are the CL module and GL module which house the bindings to OpenCL and OpenGL. If you want to see these in action you can find them in the source code for the demos (see Appendix 2 and 3). The bindings written aim to mirror the APIs as much as possible by using concepts covered such as Typed Arrays for dealing with data buffers. If you look in Figure 18 you will see that by using the withkeyword which takes all data in an object and makes it available outside (see Figure 19 for a better example) we have API calls that match as if it was in C++ and this is by design to make sure the code written maps as directly as possible to people with previous experience.

### 4.4.1 GL module



Figure XX: CL/GL API example



Figure XX: With keyword example

### 4.4.1 CL module

# 5.0 Testing

To test and evaluate the applicability and viability of the developed platform several demonstrations were developed to help identify the benefits of the platform and to ensure that the platform developed meets the required goals. These demonstrations are relatively simply programs; however, they effectively demonstrate how simple concepts are easy to write and easy to get into.

## 5.1 OpenGL Demo’s

There are three OpenGL demonstrations that make up the demonstrations set. This includes a 3D Cubes demonstration which draw a randomised list of textured cubes in a scene and performs various translations and rotations on each primitive. Additionally, there is a 3D terrain demonstration which generates a random set of geometry from a given set of parameters and renders it on screen to view. Finally, there is also a lighting demonstration which is a simple per fragment lighting shader that correctly colours a 3D geometric primitive to simulate light in the scene.

### 5.1.1 3D Cubes

As alluded to in the previous section the following demonstration just renders a random number of cubes on screen and rotates them during execution. The output can be seen in Figure XX. This example makes good use of many features of the platform. First it utilizes good use of the file system module to not only load the texture that is bound to the cube in the fragment shader, but it also uses the file system to read the geometric data stored in the JSON file format from disk. The maths module that comes part of the system is also put to good use here as it is what enables the rotation and translations to take place as these are done through matrix calculations which come by default as part of the platform. In this demonstration, we can really see the added benefit of having a platform as it provides a lot of the functionality needed.

### 5.1.2 3D Terrain

The terrain demonstration is similar in various aspects. It reads data from a JSON file and renders it using OpenGL. However, the data read from disk isn’t the geometry to be rendered but rather the settings for generating the terrain. In Figure XX below you can see what this JSON file looks like. The seed value is the seed for the random number generator that is used to generate the random terrain heights. The grid object specifies the size of the grid and each individual slab.



When the demonstration is run, you will see the output seen in Figure XX. The terrain is generated by pushing geometry into a Typed Array which is then directly fed to a vertex array object. Once there it is drawn quite simply with a fragment shader that sets the output fragment to green. In the vertex shader we merely perform the matrix multiplications needed to draw the terrain in 3D and to set camera in a position which can survey a large section of the terrain. We turn off filling of the polygons to give a clearer view of the terrain generated.



### 5.1.3 3D Lighting

The final OpenGL demonstration created was a simple lighting example. In this example, we perform per fragment lighting on geometry in a scene with a single directional light facing up and to the right of the scene. The geometry in our case a cube is loaded in and has pre-calculated normals for the lighting calculations. Per fragment lighting is used as it is more accurate calculating a lighting value for each fragment rather than each vertex and interpolating across them which is what would have happened if the code in Figure XX was placed in the vertex shader.



To prove that the lighting is dynamic and not static the geometry is rotated on its Y axis showing how lighting is calculate per frame and not once and baked into the final image. The current lighting equation is the Phong Reflection Model without the specular highlight for simplicity. The output of the program can be seen below and even though each side is the same colour because the lighting alters the final colour of the surface we can see each side has a unique colour.

## 5.2 OpenCL Demo’s

To accompany the three OpenGL demonstration which help show the OpenGL bindings in full effect three simple OpenCL demonstrations were also devised to show the OpenCL bindings working in full effect as well. The first is a grayscale demonstration which takes an input image and write as grayscale version of it to disk using an OpenCL kernel. The second is a Sobel Filter demonstration which performs a edge detection test across the image again using an OpenCL kernel and finally we have a matrix demonstration that performs a matrix multiplication operations on a given matrix stored in JSON and writes the output to disk.

### 5.2.1 Grayscale Demo

Explain demo

### 5.2.2 Sobel Filter Demo

Explain demo

### 5.2.3 Matrix Demo

Explain demo

# 6.0 Future Work

There is a lot of future work that could be undertook to expand on the work explained here. This includes looking at different APIs that could be bound to the current platform. Looking at additional high level language could also be an option for instance Python or Lua. Different interfaces could be tried as well dropping the standard program as you see here and providing it with an interactive UI. All of this is discussed in detail in the below sections as there is plenty of scope for additional work and research that could greatly improve on the project that has been presented in this paper.

## 6.1 DirectX

DirectX could be an exciting addition in future work due to its API design. Unlike OpenGL and OpenCL DirectX has many object-oriented features. DirectX is a series of COM (Component Object Model) objects, this differs from OpenGL where all API calls are from standard functions. Another major difference is that DirectX doesn’t just cover 3D rendering. So, investigating how you could expose the wider DirectX API which also includes APIs for networking, media and input could prove a very interesting project. This would be because instead of wrapping standard C functions the code written would have to write the objects presented in the DirectX APIs. A sample of DirectX code can be found in Figure XX, where you can see that there would be a very different approach when embedding this API for use by JavaScript.



## 6.2 CUDA

CUDA could also be an exciting API to look at embedding into the platform. CUDA would be a difficult project as well CUDA programs are compiled with a custom compiler NVCC making it very different from OpenCL. What is more is that CUDA relies on annotations on C++ code and it would be interesting to see how this could be mimicked in JavaScript. This could involve the creation of a new set of objects that mimic CUDA concepts. For instance, a Kernel object that helps compile a CUDA style kernel for the user during runtime.



## 6.3 Applications

Looking at building full scale applications could also be an interesting topic in future work. As we have seen JavaScript is very usable in the small examples providing for testing the platform however we have not considered examples with larger amounts of code. Traditionally one of the issues often sited with JavaScript in real world use has been its lack of a type system which results in easy errors that are not caught at compile time but instead of caught at runtime. This could prove troublesome as JavaScript is not a type safe language which could make larger scale programs a lot more difficult to program. This future work could look at various extensions to the JavaScript ecosystem of languages to consider how to solve this problem including TypeScript and CoffeeScript which are compiled down to JavaScript but provide features such as multiline strings, classes which were not available in previous JavaScript versions and type safety to make simple errors are caught at compile time.



## 

## 6.4 Framework vs Platform

In this project a platform was built which would compile and run JavaScript code. This approach had several benefits not least allowing an integrated environment for the user to write the programs. However additional work could look at how this platform would compare with an all in one framework that seeks to expand on the default environment available i.e. C++ and see how it contrasts with the all in one platform approach laid out in this report. This would particularly yield an interesting comparison between adding to the pre-existing environment or creating one from scratch.

## 6.5 Alternative Languages

In this JavaScript was chosen as the platforms language due to its popularity, ease of use, ecosystem, and great compiler support. However, there are many additional high level languages that also meet these goals including Python and to a lesser extent Lua. Looking at how these languages could compare with JavaScript as a generic scripting language would also be an intriguing area of future work. Both have syntax and structures that aim to enhance productivity with day to day tasks. All mentioned languages have a wide range of compilers and interpreters to choose from. They are also all popular and considering each languages strength against each other in the field would be a very interesting project.

## 6.5 User Interface

There is no current user interface in place for the platform and interaction is through command arguments provided to the program. Looking at a proper UI for the platform could become a fantastic piece of usability research and would build on top of the ease of use factor that formed part of this project. Many all in one environments already provide this such as MATLAB which provides an all in one software solution for science and research and while the heart of the system is its libraries and scripting facility, the UI helps guide new users and expand on the platform to make it more effective through a useable approachable UI.

# 7.0 Critical Appraisal

This project had several large-scale goals. The first was a fulling working platform that did a, b, c, d. Several demonstrations that proved its effectiveness in real world use. In depth literature in the current state of GPUs on a desktop level and the APIs that are used by them. Finally, an in-depth investigation into the current state of JavaScript as a general scripting language and its compiler technology.

These goals were met. An initial prototype of the platform was made early in the research stage thanks to a lot of investment in development before the project started. This allowed a good amount of time to be spent on a review of technical areas that would form the basis for the project. Including an investigation into JavaScript and its latest standard, and GPUs from both a hardware and software perspective. The demonstrations were also made and highlight the advantages of the platform and helped demonstrate how simple GPU concepts can be if they are made as simple as they are, so there was a lot to celebrate in the project.

However, the amount of research undertaken could have been more expansive. While OpenGL and OpenCL were made part of the platform there are a myriad of GPU industry APIs that could have also been embedded but weren’t as the additional workload would have been unmanageable to finish in the given time. The platform is also extremely limited providing only the basic bindings to OpenGL and OpenCL which could have been taken further if more time could be spent on development. Finally, the last failure is that the result of the development project is only available on Windows and isn’t Available on other platforms.

Overall however I feel like the project was a great success. It culminated in a working version of the platform as well as several demonstrations that perfectly demonstrate the research and its value.

# 8.0 Conclusion

References

Amdcom. 2016. *Amdcom.* [Online]. [18 December 2016]. Available from: <http://www.amd.com/en-us/innovations/software-technologies/mantle>

Amdcom. 2016. *Amdcom.* [Online]. [5 December 2016]. Available from: <http://www.amd.com/en-gb/products/processors/desktop/a-series-apu>

Applecom. 2016. *Applecom.* [Online]. [18 December 2016]. Available from: <https://developer.apple.com/metal/>

Atomio. 2016. *Electron.* [Online]. [13 December 2016]. Available from: <http://electron.atom.io/>

Commonjsorg. 2016. *Commonjs.org.* [Online]. [5 December 2016]. Available from: <http://wiki.commonjs.org/wiki/Modules/1.1.1>

Ecma International. (2015). *ECMAScript Specification.* Available from: <http://www.ecma-international.org/ecma-262/6.0/>

David Geer, 2005. Chip makers turn to multicore processors. *Computer*, *38*(5), pp.11-13

Googlesourcecom. 2016. Googlesource.com. [Online]. [5 December 2016]. Available from: <https://chromium.googlesource.com/v8/v8.git>

Intelcom. 2016. *Intel® ARK (Product Specs).* [Online]. [18 December 2016]. Available from: <http://ark.intel.com/products/43546>

Intelcom. 2016. *Intel.* [Online]. [5 December 2016]. Available from: <http://www.intel.com/content/www/us/en/architecture-and-technology/visual-technology/graphics-overview.html>

Khronosorg. 2016. *Khronosorg.* [Online]. [18 December 2016]. Available from: <https://www.khronos.org/vulkan/>

Karimi, K., Dickson, N.G. and Hamze, F., 2010. A performance comparison of CUDA and OpenCL. *arXiv preprint arXiv:1005.2581*.

Mahalakshmi, M. and Sundararajan, M., 2013. Traditional SDLC Vs Scrum Methodology–A Comparative Study. *International Journal of Emerging Technology and Advanced Engineering*, *3*(6), pp.192-196.

Nodejs foundation. 2016. *Nodejsorg.* [Online]. [18 December 2016]. Available from: <https://nodejs.org/en/>

Mozillaorg. 2016. *Mozilla Developer Network.* [Online]. [6 December 2016]. Available from: <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Typed_arrays>

Mozillaorg. 2016. *Mozilla Developer Network.* [Online]. [5 December 2016]. Available from: <https://developer.mozilla.org/en-US/docs/Web/API/Console>

Nvidiacom. 2016. *Nvidiacom.* [Online]. [15 December 2016]. Available from: <http://www.nvidia.com/object/IO_20020111_5424.html>

Nvidiacom. 2009. *The Official NVIDIA Blog.* [Online]. [15 December 2016]. Available from: <https://blogs.nvidia.com/blog/2009/12/16/whats-the-difference-between-a-cpu-and-a-gpu/>

Nvidiacom. 2016. *Nvidiacom.* [Online]. [18 December 2016]. Available from: <http://www.nvidia.com/object/cuda_home_new.html>

Steampoweredcom. 2016. *Steampoweredcom.* [Online]. [18 December 2016]. Available from: <http://store.steampowered.com/hwsurvey/>

Shen, G., Gao, G.P., Li, S., Shum, H.Y. and Zhang, Y.Q., 2005. Accelerate video decoding with generic GPU. *IEEE Transactions on circuits and systems for video technology*, *15*(5), pp.685-693.

Stack overflow blog. 2016. *Stack Overflow Blog.* [Online]. [18 December 2016]. Available from: <https://stackoverflow.blog/2016/03/stack-overflow-developer-survey-results/>

Unity3dcom. 2016. *Unity.* [Online]. [18 December 2016]. Available from: <https://unity3d.com/>

W3org. 2016. *W3org.* [Online]. [5 December 2016]. Available from: <https://www.w3.org/TR/2011/WD-html5-20110525/timers.html>

Yang, Z., Zhu, Y. and Pu, Y., 2008, December. Parallel image processing based on CUDA. In *Computer Science and Software Engineering, 2008 International Conference on* (Vol. 3, pp. 198-201). IEEE.