

**Computer Games Technology**

A JavaScript Runtime for Hardware Accelerated Applications

**Computing Honours Project (COMP10034) Interim Report**

William Taylor

**B00235610**

13/12/2016

**Supervisor: Paul Keir**

Contents

[1.0 Introduction 4](#_Toc469583678)

[1.1 The Topic 4](#_Toc469583679)

[1.2 The Problem 4](#_Toc469583680)

[1.3 The Project 4](#_Toc469583681)

[2.0 Technical Review 5](#_Toc469583682)

[2.1 GPUs 5](#_Toc469583683)

[2.1.1 Hardware 5](#_Toc469583684)

[2.1.2 Integrated vs Dedicated 5](#_Toc469583685)

[2.1.2 NVidia & ATI 5](#_Toc469583686)

[2.1.3 Intel & AMD 5](#_Toc469583687)

[2.1.5 Software 5](#_Toc469583688)

[2.1.6 OpenGL & DirectX 5](#_Toc469583689)

[2.1.7 CUDA & OpenCL 6](#_Toc469583690)

[2.2 JavaScript 6](#_Toc469583691)

[2.2.1 Typed Array Architecture 6](#_Toc469583692)

[2.2.2 ArrayBuffer 6](#_Toc469583693)

[2.2.3 Uint8Array, Uint32Array, Float32Array, Float64Array 7](#_Toc469583694)

[2.3 V8 JavaScript Compiler 7](#_Toc469583695)

[2.3.1 Interpreter vs Compiler 7](#_Toc469583696)

[2.3.2 ECMAScript 2015 8](#_Toc469583697)

[3.0 Current Progress & Plan for Completion 8](#_Toc469583698)

[3.1 Development Technique 8](#_Toc469583699)

[3.1.1 Agile Software Development Methodology 9](#_Toc469583700)

[3.1.2 Scrum Method 9](#_Toc469583701)

[3.1.3 Fortnightly Sprints 9](#_Toc469583702)

[3.1.4 Gantt Chart Schedule 9](#_Toc469583703)

[3.2 Prototype JavaScript Runtime 9](#_Toc469583704)

[3.2.1 V8 10](#_Toc469583705)

[3.2.2 Modules 11](#_Toc469583706)

[3.2.3 Libraries 11](#_Toc469583707)

[3.3 Current Demonstrations 14](#_Toc469583708)

[3.3.1 Image Manipulator Demo 14](#_Toc469583709)

[3.3.2 3D Textured Cube Demo 15](#_Toc469583710)

[3.4 Plans for Completion 15](#_Toc469583711)

[3.4.1 Complete Bindings 15](#_Toc469583712)

[3.4.2 Professional Demonstrations 16](#_Toc469583713)

[3.4.3 Presentation 16](#_Toc469583714)

[3.4.4 Final Honours Project Report 16](#_Toc469583715)

[4.0 Concluding Remarks 17](#_Toc469583716)

[4.1 Excellent Progress 17](#_Toc469583717)

[4.2 Presentations & Demonstrations 17](#_Toc469583718)

[4.3 Possible Future Problems 17](#_Toc469583719)

[4.4 Conclusion 17](#_Toc469583720)

[References 18](#_Toc469583721)

[Appendix 1 – Project Specification 19](#_Toc469583722)

[Appendix 2 – OpenGL Demo Source Code 21](#_Toc469583723)

[Appendix 3 – OpenCL Demo Source Code 23](#_Toc469583724)

[Appendix 4 – Supervisor Meetings 25](#_Toc469583725)

# 1.0 Introduction

## 1.1 The Topic

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. Due to 3D and high resolution media increasing in popularity we now see Graphics Processing Unit (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 GPU parallel programming more widely adopted.

## 1.2 The Problem

Developing and learning how to utilise this technology is a problem however. There is no easy to use integrated environment to experiment with various GPU APIs such as OpenCL and OpenGL. What’s more using these APIs tends to be in low level C++ leading to a lot of boilerplate which needs to be written before you actually start writing GPU based programs.

## 1.3 The Project

In this project we aim to build an all in one platform suitable for GPU programming experimentation, learning and prototyping. We will provide a JavaScript runtime which aims to provide a bulk of features out the box and native bindings to popular industry standard APIs. Finally, we aim to ensure that the techniques developed using the platform should be easily integrated into real world applications. 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.

# 2.0 Technical Review

## 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 systems 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. Note I will be looking at GPUs

### 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 which have 2 cores and quad core processors which have 4. 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 will typically be faster than normal 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.

### 2.1.2 Integrated vs Dedicated

GPUs in order to become more mainstream has had to been shrunk and extruded into different form factors to suite 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 expansive cards as they do not need to meet harsh size restraints or power limits. Dedicated cards are made available from two major manufacturers AMD and NVidia. Another type of GPU is Integrated graphics. This is where the GPU is integrated onto the CPU die allowing increased media application performance without the need of an expensive large GPU card. The first modern integrated GPU was built by AMD with their Accelerated Processing Unit (APU) chip which was designed to provide better 3D and media performance in small form factor computers such as laptops and ultra-books.

### 2.1.2 NVidia & ATI

Explain dedicated graphics cards made available by these manufacturers

### 2.1.3 Intel & AMD

Explain integrated graphics cards made available by these manufacturers

### 2.1.5 Software

How you access these cards

### 2.1.6 OpenGL & DirectX

GPUs have been traditionally accessed through industry APIs with most under the umbrella of the Kronos Group a non-profit organisation with multiple world leading companies advising and signing off on industry standards. OpenCL and OpenGL are two of their more well-known standards. Although there are other very popular APIs such as CUDA and DirectX that are not maintained by the Kronos Group they are for specific hardware and software. DirectX for example is only accessed through the Windows operating system.

OpenCL stands for open compute language. It is an API designed to allow programmers to process data in parallel across multicore devices such as GPUs. OpenCL is best used for computer vision and image processing where traditional algorithms can be paralysed to great effect. OpenCL also has a Web variant called WebCL which provides much of the same functionality to Web based applications. OpenGL on the other hand stands for open graphics library which aims to provide a pipeline with programmable elements. It’s used for high detailed, high resolution 3D rendering but can also be used for advanced 2D work and general graphics applications.

The section people learn and experiment the most with are shaders in OpenGL and kernels in OpenCL. These are custom built programs built to take data provided by the program and produce an output either on screen or to an output buffer. The platform we build will be providing a high level environment to mutate and prepare data sent to these programs, but will let the user write the custom shader and kernel programs which can then be imported into any application.

### 2.1.7 CUDA & OpenCL

Before proceeding it is best to summarise why GPU programming is an important subject to address. In recent years GPU’s have become more mainstream. If you for instance look at the latest 6th generation intel processors, they all ship with integrated GPU chips on the die (Intelcom, 2016). As such most computers which ship with an Intel processor now have a high performing GPU chip waiting to be utilized.

Much research has investigated the advantage of taking traditional algorithms and seeing the speeds up available. Research by Yang et all (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 by Teodoro et all (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.

## 2.2 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 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. JavaScript is also asynchronous meaning that the complexity of multicore programming will not be found here, instead a simple and easy to understand event driven style leads to a less complex environment. 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 Typed Array Architecture

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. They provide the ability to interact with raw binary data. They provide the perfect way to work with low level data structures and provide much more control over the underlying data.

### 2.2.2 ArrayBuffer

ArrayBuffer is the core type for every Typed Array object and it just represents a stream of binary data. Look at the following example we can take the struct person and represent it in memory in JavaScript with the following ArrayBuffer.



Figure 1: basic struct example



Figure 2: Figure 1 struct represented in modern JavaScript

At this point the JavaScript example and the C++ example have access to the same set of data and the same amount 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.

### 2.2.3 Uint8Array, Uint32Array, Float32Array, Float64Array

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. Look at the following C++ arrays.



Figure 3: 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.



Figure 4: Figure 3 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++.

## 2.3 V8 JavaScript Compiler

### 2.3.1 Interpreter vs Compiler

In 2008 Google back set the benchmark for JavaScript compilers. They created a new JavaScript JIT compiler, V8 from the ground up to dramatically improve JavaScript execution speed. Internally they built a benchmark called V8 bench and measured performance increases overtime. As you can see in figure 5 each subsequent revision of Chrome which in turn has a new version 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.

### 2.3.2 ECMAScript 2015

V8 implements ECMAScript as specified in ECMA-262, 5th edition, and runs on Windows (XP or newer), Mac OS X, and Linux systems that use IA-32, x64, or ARM processors. V8 enables any C++ application to expose its own objects and functions to JavaScript code. It's up to the programmer to decide on the objects and functions exposed to JavaScript. There are many applications that use V8 including Adobe Flash and the Dashboard Widgets in Apple's Mac OS X and Yahoo Widgets. V8’s key trait is its high level of performance.



Figure 5: JavaScript benchmark scores with each release of the Chrome web browser.

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 which uses V8. You can write full 3D games with the Unity game engine which uses it as its scripting language. Finally, through open source projects such as Electron (Atomio, 2016) you can now write native desktop applications as well. We built our platform on top of the V8 compiler to ensure that the platform is fast and efficient and provides access to the latest JavaScript standard.

# 3.0 Current Progress & Plan for Completion

## 3.1 Development Technique

When developing the runtime, we followed an agile style when it came to development. In the following section I will be breaking down why I chose this style and how it affected the project’s development.

### 3.1.1 Agile Software Development Methodology

Agile software development itself is a large umbrella term for a wide range of practices used to enforce the goals set out in their Agile Manifesto. The manifesto itself comprises of four values.

1. Individuals and interactions over processes and tools
2. Working software over comprehensive documentation
3. Customer collaboration over contact negotiation
4. Responding to change over following a plan

### 3.1.2 Scrum Method

There are many methods that adopt an agile approach, I chose scrum as its widely adopted and one I personally enjoy. Research has also been taken in this area to identify the advantages of scrum. Mahalakshmi and Sundararajan (2013) identified various advantages to using scrum including;

* Increased control of the project schedule
* Increased software quality
* Flexible enough when adapting to changes
* Work proceeds and completes more logically

### 3.1.3 Fortnightly Sprints

For each task I managed a fortnightly sprint to achieve my work. This has worked wonders in the first part of the development and I will continue to use this for the final section of the project. When it came to assigning tasks into weekly sprints I would take one task from a Gantt chart, figure 6 and break it down into multiple sub tasks which could be completed on a weekly or fortnightly basis.

### 3.1.4 Gantt Chart Schedule

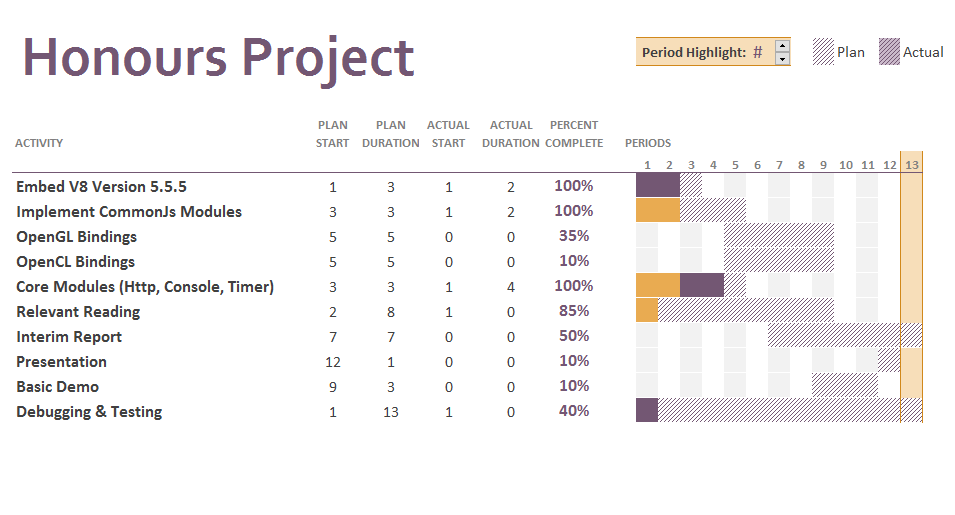
To better manage the workload I created a Gantt chart to manage tasks and set out the schedule for the project. Note you can also find in the appendices copies all of the meetings I had with my supervisor which also helped coordinate progress and work load.

Figure 6: Gantt chart with all major developments set out.

## 3.2 Prototype JavaScript Runtime

In trimester one we managed to complete a prototype version of our JavaScript runtime. In the following section I will be covering the steps needed to build the runtime, a quick overview of key concepts related to the technology used and how I made it a standalone application.

### 3.2.1 V8

The first step was to get V8, Google’s JavaScript JIT compiler 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 has a lot of custom build tools and technologies that were also needed to build V8 from source. After a large amount of time had been spent we had 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.

#### 3.2.1.1 Isolate

An isolate in V8 is defined as a 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.



Figure 7: Sample function which creates a new VM instance

#### 3.2.1.2 Handles

Handles are pointers to objects exposed to JavaScript. All V8 objects are accessed using handles and are needed as JavaScript uses a garbage collection 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.



Figure 8: Sample function which creates a handle to a Number variable available in JavaScript

#### 3.2.1.3 Scopes

Scopes are containers for a sequence of handles. They make handle management easy and deal with handle deletion in one shot rather than you having to delete the handle yourself individually.



Figure 9: Example of creating a handle scope

#### 3.2.1.4 Context

A context is an execution environment that allows separate, unrelated, JavaScript code to run in a single instance of V8. Whenever you start up V8 execution environment you have must specify the context in which it runs. The reason contexts are used is so that 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.



Figure 10: Example of creating a context and global object template

### 3.2.2 Modules

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.

#### 3.2.2.1 Require

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 for any reason the given module identifier does not lead to a valid module an error must be thrown with an accepted message detailing why.

#### 3.2.2.2 Module Context

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 its 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.

#### 3.2.2.3 Module Identifiers

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 the first time is “.” or “..”. Finally, relative identifiers are resolved relative to the call to require.

### 3.2.3 Libraries

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

#### 3.2.3.1 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).



Figure 11: Console API example

#### 3.2.3.2 Datetime module

We also provided a date time module for managing time. These methods are based on the time browser specification so its familiar to web developers (W3org, 2016). We also added an additional pause method which mirrors the Win32 API Sleep function.



Figure 11: Datetime API example

#### 3.2.3.3 System module

To provide information on the system we provided a system module which allows the user to check system conditions. 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 12: System API example

#### 3.2.3.4 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.



Figure 13: Http API example

#### 3.2.3.5 Fs module

Of course a big feature of 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 object can be passed directly to OpenCL and OpenGL for processing.



Figure 14: Fs API example

#### 3.2.3.6 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 and message boxes, basic components available on desktop operating systems. You can see an example of how to open a window and enable an OpenGL context.



Figure 15: Display API example

#### 3.2.3.7 CL module & GL module

The core modules are the CL module and GL module which house the bindings to OpenGL and OpenCL. If you want to see these in action you can find the demo code in the appendices. The bindings found 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 14 you will see that by using the **with** keyword which takes all data in an object and makes it available outside (see figure 15 for a better example) we have a 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.



Figure 14: CL/GL API example



Figure 15: With keyword example

## 3.3 Current Demonstrations

In addition to our draft runtime we also have crafted two demos which shows us the bindings to both OpenCL and OpenGL from JavaScript working in full motion. I will quickly cover them and show example outputs and the code for them can be found in the appendices.

### 3.3.1 Image Manipulator Demo

Prepared in the first trimester is this image manipulator demonstration. This demonstration loads a local image then opens an OpenCL context and executes and OpenCL kernel that outputs a result. The first kernel executed performs a grayscale operation on the image and writes it to an output buffer. The second kernel performs a simple blur across a given image and writes this to an output buffer. Upon completion these output buffers are written to disk. Below you can find a sample output of the program. This demonstration is written entirely with the prototype runtime and certainly shows we are making fantastic progress.

Figure 17: Original image followed by the grayscale output and blurred output

### 3.3.2 3D Textured Cube Demo

Also prepared this trimester is this 3D Textured Cube demo which takes geometric data written in JSON and renders it on screen in 3D using OpenGL. The demo itself uses Vertex Array Objects (VAO) and Vertex Buffer Objects (VBO) to load and store vertex data on the GPU. It utilizes custom shaders, texture objects allowing us to bind textures to the geometry points specified and good use of 3D matrices to perform translations, rotations and required camera work to view the object in 3D.

­­ 

­­ Figure 18: 3D rotating textured cube rendered using OpenGL

## 3.4 Plans for Completion

In this final parts of this report will be covering the action points I will be focusing on in the incoming trimester to finish the progress made. Great progressed has been achieved and we need to make sure it carries through to the next trimester.

### 3.4.1 Complete Bindings

Most importantly I will be looking to add a majority of the bindings required to OpenGL and OpenCL to build more advanced demonstrations to showcase the project. This may result in additional issues however because both API’s have sections we have not touched upon which could prove difficult to expose to JavaScript without careful consideration. However, as we have seen full OpenGL ES 2.0 has been made available to JavaScript through WebGL beforehand we don’t expect major issues here. So the chances of a bottleneck here are small.

### 3.4.2 Professional Demonstrations

Once we have fully built out the bindings to OpenGL and OpenCL we will be looking to more advanced demos that prove the platform serves a purpose. Ideally we will be looking at traditional methods that could be help demonstrate the speeds up possible and how using the runtime can help prototype this work.

### 3.4.3 Presentation

The presentation is due in the near future and work will need to be done on this. Luckily due to a good level of work being completed already I have no shortage of things I have to demonstrate. I plan to do a proper presentation on the work done providing ample opportunity to show the runtime in action through developed examples and demonstrations.

### 3.4.4 Final Honours Project Report

Finally, the most important piece that needs to be completed is the final honours project report which is the bulk of the marks for the project. I plan to make an early start on this to ensure this isn’t rushed in the last weeks of trimester 2.

# 4.0 Concluding Remarks

## 4.1 Excellent Progress

Progress has been extremely good in my view. Not only do we have a working version of the runtime which is capable of compiling modern JavaScript and providing industry standard bindings go GPU APIs but working demonstrations as well. Most implementation work for the honours project is handled in trimester two and to make such progress will certainly help me in the incoming trimester.

## 4.2 Presentations & Demonstrations

Shortly we will begin work on the presentation for this project. Thankfully due to a lot of progress being made I will have a good selection of items to showcase from including two demos. I will be aiming to use these demos to highlight the progress made and to make one wonder of the possible direction and advances that can be made in future.

## 4.3 Possible Future Problems

We could have problems in the future as we seek to provide more bindings to industry APIs. As these APIs have been built up over years with many new additions and extensions it may prove difficult to provide a complete set of bindings that are bug free due to the amount of functions available in each API. Nevertheless, we will seek to provide proper, tested, working calls to the most popular methods in these APIs so we can build the demonstrations required to show that prototyping applications using this platform is advantageous and reliable.

## 4.4 Conclusion

In conclusion a lot of work has been completed and we are well on track to producing a well-produced piece of research. The following months will bring forward much advancement but all were made possible thanks to the addition of early prototype work and building on this will be the key to a good result at the end of the module.

# References

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/

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>

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

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.

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.

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.

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

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

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

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

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

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

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

# Appendix 1 – Project Specification

**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.***

# Appendix 2 – OpenGL Demo Source Code





# Appendix 3 – OpenCL Demo Source Code





# Appendix 4 – Supervisor Meetings

To be attached at a later date