

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

**A JavaScript Runtime for Hardware Accelerated Applications**

**William Taylor**

**B00235610**

**22/04/2016**

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

# 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 was able to complete this project to a standard much better than it would have been had I done it alone and I was able to 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 who’s lectures helped set out the work required and he was able to 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 was able to 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 2](#_Toc474088224)

[1.0 Introduction 5](#_Toc474088225)

[2.0 Background Information 6](#_Toc474088226)

[2.1 GPUs 6](#_Toc474088227)

[2.1.1 Hardware 6](#_Toc474088228)

[2.1.2 Manufacturers 7](#_Toc474088229)

[2.1.3 Software 7](#_Toc474088230)

[2.2 JavaScript 9](#_Toc474088231)

[2.2.1 Design 9](#_Toc474088232)

[2.2.2 Typed Arrays 9](#_Toc474088233)

[2.2.2 ArrayBuffer 9](#_Toc474088234)

[2.2.3 ECMAScript 2015 10](#_Toc474088235)

[2.2.4 Module Systems 10](#_Toc474088236)

[2.3 V8 10](#_Toc474088237)

[2.3.1 Design 11](#_Toc474088238)

[2.3.2 Concepts 11](#_Toc474088239)

[2.3.3 Usages 11](#_Toc474088240)

[3.0 Software Design 11](#_Toc474088241)

[3.1 Requirements 12](#_Toc474088242)

[3.2 Architecture 12](#_Toc474088243)

[3.3 Interface 12](#_Toc474088244)

[4.0 Development 12](#_Toc474088245)

[4.1 Embedding V8 13](#_Toc474088246)

[4.2 Module System 13](#_Toc474088247)

[4.3 Common Libraries 13](#_Toc474088248)

[4.4 GPU Bindings 13](#_Toc474088249)

[5.0 Testing 14](#_Toc474088250)

[5.1 OpenGL Demo’s 14](#_Toc474088251)

[5.1.1 3D Cubes 14](#_Toc474088252)

[5.1.2 3D Terrain 14](#_Toc474088253)

[5.1.3 3D Lighting 14](#_Toc474088254)

[5.2 OpenCL Demo’s 14](#_Toc474088255)

[5.2.1 Grayscale Demo 14](#_Toc474088256)

[5.2.2 Sobel Filter Demo 14](#_Toc474088257)

[5.2.3 Matrix Demo 14](#_Toc474088258)

[6.0 Future Work 15](#_Toc474088259)

[7.0 Critical Appraisal 16](#_Toc474088260)

[8.0 Conclusion 17](#_Toc474088261)

# 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.

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.

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.

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

### 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

### 2.2.4 Module Systems

## 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.

### 2.3.1 Design

### 2.3.2 Concepts

### 2.3.3 Usages

Fast JavaScript execution did not go unnoticed. JavaScript can now be found in many environments other than the Web and in the programs written above. You can now write server side applications in JavaScript with Node.js (Nodejs foundation, 2016) which uses V8. You can write full 3D games with the Unity game engine (Unity3dcom, 2016) 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 and because it has been used so successfully in other runtimes.

# 3.0 Software Design

## 3.1 Requirements

## 3.2 Architecture

## 3.3 Interface

# 4.0 Development

## 4.1 Embedding V8

## 4.2 Module System

## 4.3 Common Libraries

## 4.4 GPU Bindings

# 5.0 Testing

## 5.1 OpenGL Demo’s

### 5.1.1 3D Cubes

### 5.1.2 3D Terrain

### 5.1.3 3D Lighting

## 5.2 OpenCL Demo’s

### 5.2.1 Grayscale Demo

### 5.2.2 Sobel Filter Demo

### 5.2.3 Matrix Demo

# 6.0 Future Work

# 7.0 Critical Appraisal

# 8.0 Conclusion

References

Appendix 1 – 3D Demo Source Code

Appendix 2– Grayscale Demo Source Code

Appendix 3 – Finance Demo Source Code