

**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 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](#_Toc475439661)

[1.0 Introduction 6](#_Toc475439662)

[2.0 Background Information 7](#_Toc475439663)

[2.1 GPUs 7](#_Toc475439664)

[2.1.1 Hardware 7](#_Toc475439665)

[2.1.2 Manufacturers 8](#_Toc475439666)

[2.1.3 Software 8](#_Toc475439667)

[2.2 JavaScript 9](#_Toc475439668)

[2.2.1 Design 10](#_Toc475439669)

[2.2.2 Typed Arrays 10](#_Toc475439670)

[2.2.2 ArrayBuffer 10](#_Toc475439671)

[2.2.3 ECMAScript 2015 11](#_Toc475439672)

[2.2.4 Module Systems 11](#_Toc475439673)

[2.3 V8 11](#_Toc475439674)

[2.3.1 Design 11](#_Toc475439675)

[2.3.2 Concepts 11](#_Toc475439676)

[2.3.3 Usages 12](#_Toc475439677)

[3.0 Software Design 13](#_Toc475439678)

[3.1 Requirements 13](#_Toc475439679)

[3.2 Architecture 13](#_Toc475439680)

[3.3 Interface 13](#_Toc475439681)

[4.0 Development 14](#_Toc475439682)

[4.1 Embedding V8 14](#_Toc475439683)

[4.2 Module System 14](#_Toc475439684)

[4.3 Common Libraries 14](#_Toc475439685)

[4.4 GPU Bindings 14](#_Toc475439686)

[5.0 Testing 15](#_Toc475439687)

[5.1 OpenGL Demo’s 15](#_Toc475439688)

[5.1.1 3D Cubes 15](#_Toc475439689)

[5.1.2 3D Terrain 16](#_Toc475439690)

[5.1.3 3D Lighting 17](#_Toc475439691)

[5.2 OpenCL Demo’s 18](#_Toc475439692)

[5.2.1 Grayscale Demo 18](#_Toc475439693)

[5.2.2 Sobel Filter Demo 18](#_Toc475439694)

[5.2.3 Matrix Demo 18](#_Toc475439695)

[6.0 Future Work 19](#_Toc475439696)

[6.1 DirectX 19](#_Toc475439697)

[6.2 CUDA 19](#_Toc475439698)

[6.3 Applications 19](#_Toc475439699)

[7.0 Critical Appraisal 20](#_Toc475439700)

[8.0 Conclusion 21](#_Toc475439701)

# 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

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.

#### *2.2.4.1 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 the given module identifier does not lead to a valid module an error must be thrown with an acceptable message detailing why.

#### *2.2.4.2 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 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.

#### *2.1.4.3 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 it starts with “.”. Finally, relative identifiers are resolved relative to the call to require.

## 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

There are several concepts in V8 that one must understand if they are to use it as their own for JavaScript compilation.

#### 3.2.1.1 Isolate

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.

#### 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 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.

#### 3.2.1.3 Scopes

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.

#### 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 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 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

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 actually 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.



### 5.1.3 3D Lighting

Explain 3D lighting





## 5.2 OpenCL Demo’s

OpenCL bindings also key, three demos, source code in appendices

### 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 actually 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 looked into 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 look into 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. This enabled the possibility to add many additional features such as using JavaScript as the language you write the code in. However additional work could look at how this platform would compare with an all in one framework.

## 6.5 Alternative Languages

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.

## 6.5 User Interface

There is no current user interface in place for the platform and interaction it through command arguments provided to the program. Looking at a proper UI for the platform could prove a fantastic piece of usability research and would build on top of the ease of use factor that formed part of this project.

# 7.0 Critical Appraisal

# 8.0 Conclusion

References