UNIVERSITY OF THE WEST *of* SCOTLAND

UWS

BSc (Hons) Computer Games Technology

A JavaScript Runtime for Hardware Accelerated Applications

William Taylor

B00235610

24/03/2017

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: 24/03/2017

Form to accompany Dissertation

To be completed in full

Surname Taylor
First Name William   **Initials** WT
Borrower ID Number B00235610
Course Code COMPSCI
Course Description BSc (Hons) Computer Games Technology
Project Supervisor Paul Keir
Dissertation Title A JavaScript Runtime for Hardware Accelerated Applications
Session   2016/2017

Please ensure that a copy of this form is bound with your dissertation before submission

COMPUTING HONOURS PROJECT SPECIFICATION FORM

Project Title: A JavaScript Runtime for Hardware 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. Also, I attained 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 content like my interim report and final honours project report and it is 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

March 24th, 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, V8, JavaScript, Learning, Development, Experimentation*

Table of Contents

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 (Intel.com, 2016, Amd.com, 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 browsers 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 support in Chrome, Firefox and Opera with all other browsers not supporting the standard. 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++, does not 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 (OpenGL Mathematics Library) focuses solely on mathematics and GLUT (OpenGL Utility Toolkit) 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 (Nvidia.com, 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. Research (Yang et al, 2008) took bread and butter computer vision algorithms and compared their performance when processed across a Central Processing Unit (CPU) and a 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. 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 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 (Nvidia.com, 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 (Intel.com, 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 (Steampowered.com, 2016) and research undertaken at Jon Peddie Research an organisation that tracks GPU shipments. 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 is more their Mantle API (Amd.com, 2016) was the starting point for the new API for both compute and graphics Vulkan (Khronos.org, 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. Unique to AMD's integrated solution was the ability to have dual graphics (Amd.com, 2017) where the integrated GPU would work in tandem with a AMD GPU fitted via an expansion slot.

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 (Khronos.org, 2016), Metal (Apple.com, 2016) and CUDA (Nvidia.com, 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 is not cross platform, you will only find it on Windows, one of its key faults. Another key differential is that DirectX is not 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 is not 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.

```glsl
layout(location = 0) in vec3 positions;
layout(location = 1) in vec3 uvs;

uniform mat4 projection;
uniform mat4 model;
out vec2 uv_cords;

void main() {
    vec2 position2D = (projection * model * vec4(positions, 1.0)).xy;
    gl_Position = vec4(position2D, 0.0, 1.0);
    uv_cords = uvs.xy;
}
```

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 CUDA's 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.

```c
__kernel void vec_add(__global float* a, __global float* b, __global float* result)
{
    int gid = get_global_id(0);
    result[gid] = a[gid] + b[gid];
}
```

Figure 2: OpenCL kernel which performs a vector additional

2.1.4 Limitations

Even though the parallel compute advantage of GPUs is second to none there are still various limitations of GPUs that have been identified in multiple research papers (Vuduc, R et all, 2010) (Richardson A, Gray, A, 2008). To exploit the power of GPUs a high level of parallelism must be inherent in the application. Where this is not the case developers must adapt sequential code to run in parallel which is often a difficult task. Without this the application that leverages the GPU will not get a speed up as the workload must be parallel to perform better than a traditional CPU.

In the scientific domain, the utilisation of GPUs can lead to inaccuracies. Double floating point precision and error correction is not always supported by default in standard APIs. In fact, by forcing correct floating point calculations or double precision you incur a performance penalty (Nickolls, J. and Dally, W.J., 2010). This can add additional difficulties to domain areas where accuracy is paramount. This is an error which is baked into the hardware as double precision was not considered necessary until GPUs were adapted for more general use.

Compatibility is an outstanding issue that do plague GPUs today as well. Certain APIs only work on select hardware and should a user run the application that utilises an API that is not supported on their hardware it renders the application useless. However, this issue is being address through APIs like NVidia's Thrust (Thrust.github.io, 2017) and OpenCL. Here the APIs can utilize other parallel programming mechanisms if a GPU is not available. Thrust will utilize open source thread libraries like OpenMP (Open Multi-Processing library) and OpenCL code does not need to run on a GPU and can run on a CPU as a backup.

GPUs are also commonly criticised for large power consumption. In fact, they drain power so much that companies like NVidia created Optimus (Nvidia.co.uk, 2017) a GPU switching technology to enable the GPU to be switched on and off when needed to stop a dedicated card in a mobile device sapping all battery. While this criticism does not really matter much in the mind of your average developer it is a valid criticism for hardware manufacturers who are trying to market general purpose use of their processors when their power limit disables them from being put in low power devices.

Programmers also struggle with GPU utilization due to its constrained environment. Typical programming language staples such as conditional branch logic must be abandoned in scenarios where high performance is the key objective. Latency has also been a limiting factor of GPUs. As the device, does not have raw access to memory available to the CPU, any data to be processed must be directly transferred to the GPUs own memory and back again once it has been processed. This latency bottleneck limits GPUs to only work on larger work sets as the time of transferring memory cuts out any gain in increased computation performance.

While these limitations render a GPU impractical or harder to use in certain areas it does not stop them being incredibly useful. These topics were covered to ensure that both the advantages and disadvantages were covered before proceeding with the platform.

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

JavaScript was famously designed in 10 days in 1995 by Brendan Eich when Netscape hired him to create a new scripting language for their browser (Severance C, 2012). JavaScript's design was aimed to be a lightweight interpreted language to complement Java. But at the same time, it would need to appeal to non-professionals in the software world. Like other languages at the time JavaScript took the basic syntax from the C language with simpler semantics and dynamic memory allocations. As JavaScript was designed for web pages the complexity of concurrency and memory was left behind to present the language as a lightweight tool compared to other languages at the time. JavaScript as we know it today was brought about through the languages standardisation in the late 1990s. Through the ECMAScript standard, the core language specification has been refined and improved upon to fully realise the goals set out when Brendan Eich first started the project.

The key trait of the language is its dynamic nature. It does not have a strong type system and objects can be created on the fly with no need for pre-existing type definitions. These types can then be modified at runtime by adding additional properties which can be anything from numbers to arrays to new methods. Even though these utilities are not always used in a program (Richards G et al, 2010) nevertheless they are key to the design of the language as they are still used in some areas if not used extensively. This design yields maximum flexibility for the language. JavaScript currently resides on 94.4% of the top ten million websites on the web today (W3techs.com, 2017).

JavaScript's importance has been in large part to the expansion of the web. With the explosion in use of mobile computers accessing web content has been made easier. Most operating systems have a browser by default as do mobile operating systems like iOS and

Android. The result was an explosion in the number of users of the web and as the web has grown the key technologies that power it has grown as well. This is a key factor in JavaScript's success today.

However, its design is its own worst enemy. Multiple fundamental problems with the language are what lead to application issues. Its lack of a type system while great for writing dynamic code results in common issues across most web applications. Research undertaken (Ocariza F et al, 2013) found that one of the most common problems faults in JavaScript today is incorrect method parameters. This is of course enabled because any type can be passed to a function via a parameter and if you pass say a number to a function that was expecting an array there is no compilation step that will detect this. However, these issues can be solved through a various number of ways including utilization of Microsoft Typescript (Typescriptlang.org, 2017) and Facebook's Flow (Flowtype, 2017) which add optional type annotations to catch these errors.

But of course, as the language has become more popular we have seen it pushed into areas it was never designed to be in. Research undertook (Tilkov, S. and Vinoski, S., 2010) shows the design of the language has advantages even to domain areas such as high performance network programs. Here the event driven model of JavaScript rather than a multithreaded feature which most languages boast contains several advantages. Not least that utilizing the language is easier, that code written is easier to understand and in the case of Node the code written is more scalable. Here in this project we are aiming to see our own advantages for our own use case for JavaScript in our environment.

2.2.2 Typed Arrays

Recent revisions of the JavaScript standard have added support for objects designed to make low level programming possible (Mozilla.org, 2016). I will summarise the most ground breaking set of objects known as Typed Array 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.

2.2.3 ArrayBuffer

ArrayBuffer is the base type for every Typed Array object and it just represents a stream of binary data. Looking 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.

```
struct person {
    double height;
    char name[256];
    int age;
};
person p;
```

Figure 3: basic struct example

```javascript
// Using ES2015 new class keyword
class Person extends ArrayBuffer {
    constructor() {
        super(268); // 268 bytes

        this.height = new Float64Array(this, 0, 1);
        this.name = new Int8Array(this, 4, 256);
        this.age = new Int32Array(this, 260, 1);
    }
}


let p = new Person();
```

Figure 4: Figure 3 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 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. 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 Typed Arrays you can now control a greater range of integral types. Consider the following C++ arrays shown in Figure 5.

```cpp
unsigned char characters[10];
unsigned int positive[100];

double ratios[100];
```

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 Typed Arrays this is no longer the case as can be seen in Figure 6.

```javascript
let characters = new Uint8Array(10);
let positive = new Uint32Array(10);
let ratios = new Float64Array(100);
```

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++.

2.2.4 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 in 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 as shown in Figure 7.

```javascript
for(var i = 0; true; i++)
    break;

for(let j = 0; true; j++)
    break;

i++; // valid operation, i is a global variable
j++; // not valid operation j not in scope

const y = 0.0;
y = 0.0; // error can't reassign const y
```

Figure 7: let and const keyword example

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. Figure 8 shows off this feature.

```javascript
let x = 100.0;
let y = 100.0;

console.log(`${x} : ${y}`) // > 100 : 100
```

Figure 8: string template example

They also 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. This stops users having to mess with an objects internal prototype property and express object oriented ideas more effectively.

```javascript
class Number extends Object {
    constructor() {
        this.value = 0.0;
    }
}
```

Figure 9: Class inheritance example

2.2.5 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 (Commonjs.org, 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 popular. 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*.

2.2.4.2 AMD

The other module system devised which is not used in our platform is the AMD system (Requirejs.org, 2017). 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. There should be a single function *define* that is available as either a local or global variable. The first argument to the function is the id for module being defined. This argument if not present should default to the id of the module that the loader was requesting for the given response script. This identifier must be unique to comply with the standard.

The next argument, *dependencies*, is an array of the dependencies that are necessary for the module being defined. These dependencies must be taken care of prior to the invoke of the module factory function, and the values specified in this array should be passed as arguments to the factory function with argument positions corresponding to index in the dependencies array. The dependencies identifiers may be relative, and should be resolved relative the module that is 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 and if left out should default to *["require", "exports", "module"].*

The final argument, factory, is a regular function or object that should be executed to create the module. If the factory argument is a function it should be executed once only, if instead it is an object then it should be assigned as the exported value of the module. If the factory function returns a value (any value that coerces to true) then that value should be assigned as the exports for the module being defined. By following these rules, you fulfil the requirements set out in the AMD module specification.

2.3 V8

In 2008 Google set the benchmark for JavaScript compilers. They created a new JavaScript JIT (Just in time) 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 10 each subsequent revision of Chrome which in turn has a new version of V8 saw massive gains in JavaScript performance.
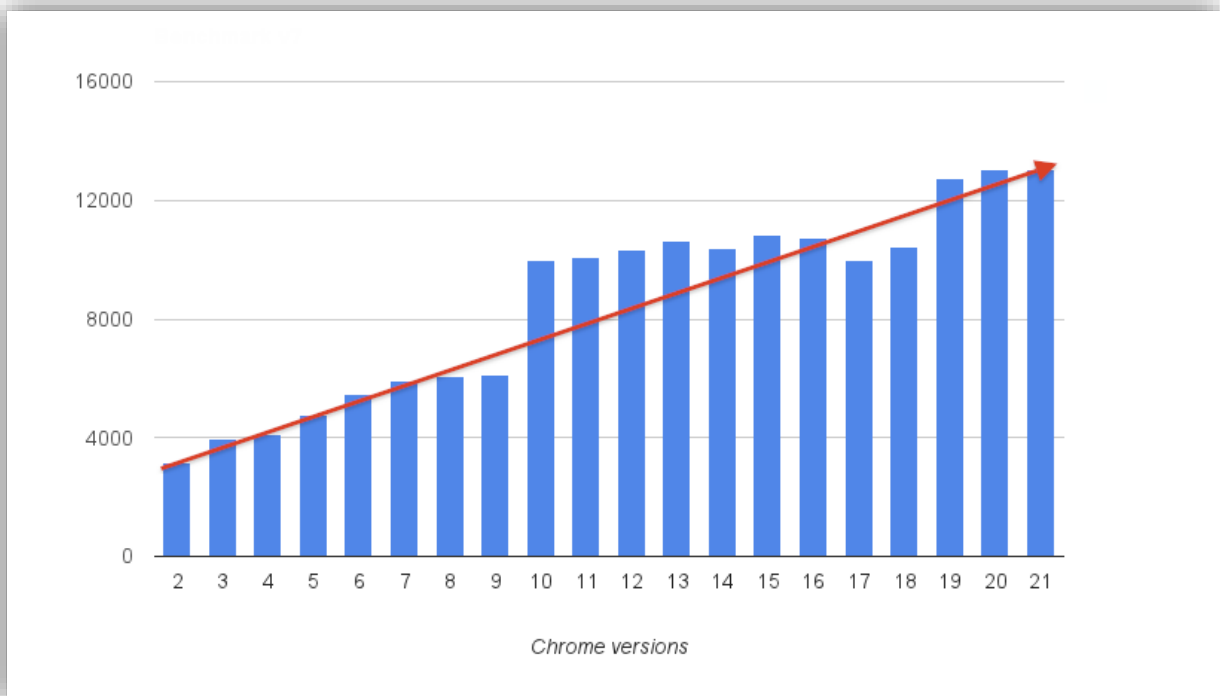
Figure 10: V8 Bench score against each Chrome version

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 meaning the compiler is cross platform as well. V8 enables any C++ application to expose its own objects and functions to JavaScript code. It is 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 offset determined by the compiler due to a fixed layout for a memory structure. Access requires a simple memory load and store.

V8 to solve this abandons dynamic look ups in favour of a concept it refers to as hidden classes (Richards G et al, 2010). 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 V8 can use 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, the code is directly compiled for maximum speed and execution.

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 make sure long pauses in a JavaScript application are no longer prevalent by processing only part of the object heap in most garbage collection cycles. 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.

2.3.4 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 VM (Virtual Machine) 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 11 you can see this object being created.

```
void example()
{
    v8::Isolate::CreateParams createParams; // Options
    v8::Isolate* isolate = v8::Isolate::New(createParams);

    isolate->LowMemoryNotification();       // hint to call garbage collection
    isolate->Dispose();                     // cleanup memory allocated
}
```

Figure 11: Isolate usage in C++

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 12 you can see a handle being created.

```
void example()
{
    v8::Isolate* isolate = v8::Isolate::GetCurrent();
    v8::Local<v8::String> stringValue = v8::String::NewFromUtf8(isolate, "Value");
    v8::String::Utf8Value(stringValue);

    auto characters = value.operator*();       // Dereference to get characters
    auto len = stringValue->Length();    // Length of the string
}
```

Figure 12: Handles usage in C++

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 13 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.

```cpp
void example()
{
    // All Handles declared in this function are deleted when this is destroyed.
    v8::HandleScope scope(v8::Isolate::GetCurrent());
}   // Cleaned up once scope is deallocated from the stack
```

Figure 13: Scopes usage in C++

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 14.

```cpp
void example(v8::Isolate* isolate)
{
    // Create the global obejct for the context
    auto global = v8::ObjectTemplate::New(isolate);
    // Create our JavaScript context which can execute code
    auto context = v8::Context::New(isolate, nullptr, global);
}
```

Figure 14: Context usage in C++

2.3.5 JavaScript & V8 Usages

Fast JavaScript execution did not go unnoticed. JavaScript can now be found in many environments other than the web which has proven the capability, adaptability and embeddability of the language.

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 most server frameworks block due to multithreading blocking 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 an 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 one of the most popular package managers in the world. All of this has been made possible by the fast JavaScript execution that V8 provides.

Through open source projects such as Electron (Atom.io, 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 is JavaScript written in the webpage to access resources normally blocked due to the browsers security model. Native modules can then be installed through NPM to gain access to common system components. Electron itself powers some popular text editors including Atom and Visual Studio Code.

Because V8 is natively part of the open source browser Chromium, V8 has manifested itself into various browsers which used chromium as the starting point for their browser implementation. These include Opera, Google Chrome, Vivaldi, and host of other less well known browsers that have been 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.

Finally, you can also write full 3D games with the Unity game engine (Unity3d.com, 2016) which uses it as its scripting language. One thing to note here is that the JavaScript found in Unity, referred to as UnityScript is not 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 the caller of the method.

3.0 Software Design

3.1 Requirements

When developing the platform, some goals were set out to clarify what was going to be achieved and what had to be implemented. The ability to compile and execute JavaScript was paramount. This is because the platform itself is meant to provide a high-level language to work with the platform and without this key component we cannot build the platform on top of it.

An additional requirement was a set of common modules that would be the platforms internal modules that housed reusable code to make programming GPUs easier. These modules should do the heavy lifting required for the platform including math operations, file system operations and bindings to desktop level OpenGL and OpenCL calls. The software must be fast and easy to use. High performance libraries and tools should be used and where possible optimizations should take place to ensure that the platform is fast. The ease of use factor should be accomplished by making the platform a onetime install instead of a sequence of tools to download and install. Another factor is that the software should be accessed and used as a command line tool for easy use.

3.2 Architecture

Figure 15 explains the architecture for the platform. The platform is an executable written in C++ which embeds V8 and runs the JavaScript written by the user. In their script the user can import and use utilities the platform provides to write their application, this may include OpenCL bindings or core libraries such as the maths module. The C++ platform utilizes various libraries to simplify the tasks it must perform including SDL2, GLEW, FreeImage and Poco.
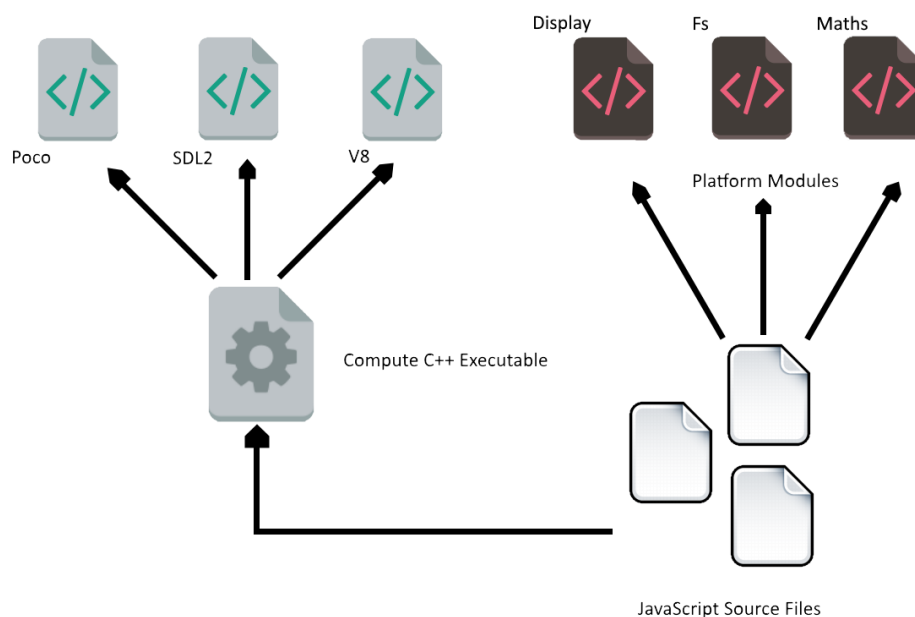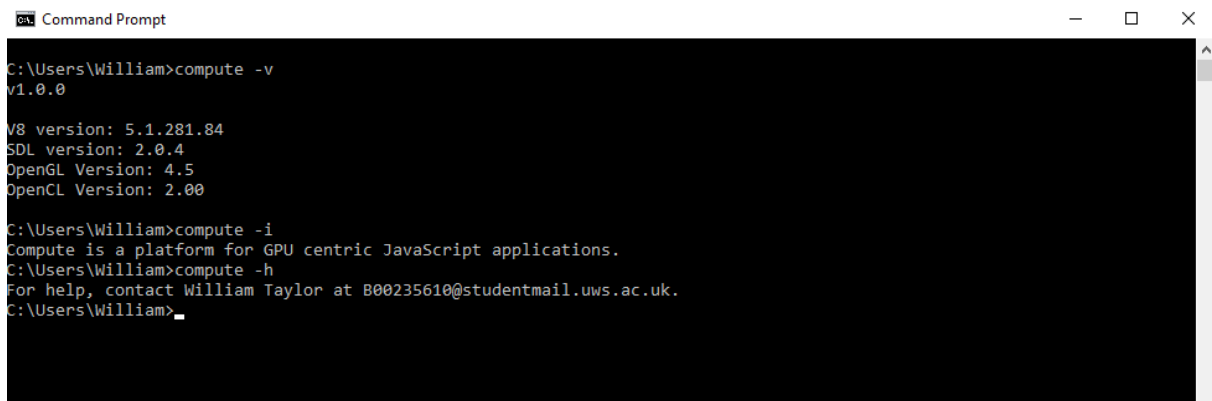

Figure 15: Architecture diagram

3.3 Interface

The interface will be a simple command line utility that programmers are familiar with. The advantage of this is that if a UI was to be built in the future it can be built on top of this command line tool. The application will be compiled and built under the name *compute* and upon install it would be available anywhere on the system once it has been added to the *PATH* environment variable on Windows. Providing a filename to the program will execute the JavaScript found in the file. Omission of the file extension will assume *.js* by default. Failure to pass any arguments to the program will cause the platform to enter a read evaluate print loop commonly referred to as REPL. Additional arguments are added such as *–h* for help information and *–i* for information on the platform. The final argument not discussed is the version argument *–v* which will print the version of the platform, the version of libraries it was built with and the versions of OpenCL and OpenGL available to the programmer. The REPL mode can be seen in Figure 17 and the program arguments can be seen in Figure 16.
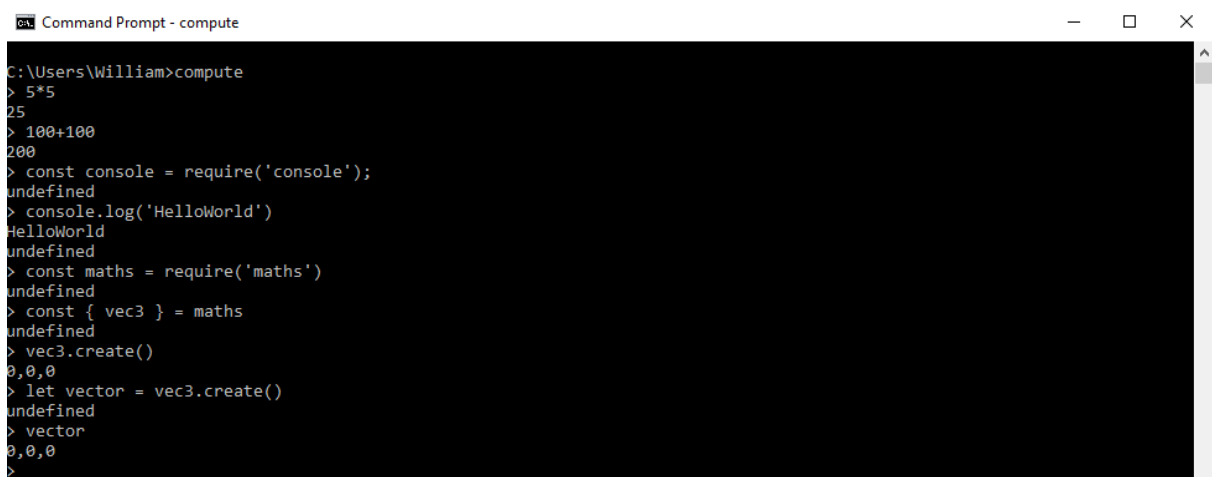

Figure 16: Platform arguments

Figure 17: REPL mode using the platform

4.0 Development

The first step in the development of the platform was to set up an online repository to host the source code for the project (GitHub, 2017). This was done so issues and progress could be easily tracked and any potential errors could be easily reverted. With the repository setup, we could now start looking at writing the platform.

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.

4.2 Module System

With V8 successfully embedded and a lone JavaScript source file being successfully parsed and executed we looked at implementing a module system that would allow users to write modular code when using the runtime. We solved this by implementing the CommonJS standard which is used in the Node runtime as well. This would allow users to load JavaScript modules themselves dynamically during runtime. It also provided a way for the platform to serve up its own modules that aid in GPGPU experimentation.

4.3 Common Libraries

With V8 and CommonJS implemented the next step was to write the core modules that would provide access to functionalities such as console output, the file system and OpenCL and OpenGL bindings.

4.3.1 Console module

To start 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 (Mozilla.org, 2016). A basic example of this module's functionality can be seen in Figure 18.

```
const console = require('console');

console.error('Error Log');
console.warn('Warn Log');
console.log('Log Log');
```

Figure 18: 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 is familiar to web developers (W3org, 2016). We also added an additional pause method which mirrors the Win32 API Sleep function. Figure 19 shows off some of the functions found in this module available for users use.

```javascript
const datetime = require('datetime');
const console = require('console');

const { setTimeout, setInterval, pause } = datetime;

setInterval(() => console.log('1 second has passed'), 1000);
setTimeout(() => console.log('5 second has passed'), 5000);

pause(100);
```

Figure 19: Datetime API example

4.3.3 System module

To provide information on the system we provided a system module. While we do not 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 20 shows the information available from this module.

```javascript
const console = require('console');
const system = require('system');

const toString = str => JSON.stringify(str, null, 4)

console.log(toString(system.instructions));
console.log(toString(system.os));
console.log(toString(system.battery()));
console.log(toString(system.hardware));
```

Figure 20: 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 lightweight 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 21.

```
const console = require('console');
const http = require('http');

const address = 'localhost';
const page = '/';
const port = 3000;
const body = {};

http.post(address, page, port, body, (res, err) => {
    console.log(res, err)
});

http.get(address, page, port, (res, err) => {
    console.log(res, err)
});
```

Figure 21: 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 22 gives a basic usage example showing how a JSON can be read and outputted directly to the console.

```
const console = require('console');
const fs = require('fs');

let image = fs.readImage('image.jpg');
let json = fs.readJson('demo.json');

console.log(`${image.width} : ${image.height}`);
console.log(JSON.stringify(json));

fs.freeImage(image);
```

Figure 22: Fs API example

4.3.6 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, message boxes, and basic components available on desktop operating systems. In Figure 23 you can see an example of how to open a window and enable an OpenGL context.

```javascript
const { openMessage, openWindow } = require('display');
const { setTimeout } = require('datetime')
const console = require('console');

let title = console.read(), body = console.read();

openMessage(title, body, () => {
    openWindow({}, window => {
        window.setTitle('HelloWorld');
        window.show();
        window.enableOpenGL();
        window.onFrame(() => {
            window.swapBuffers();
        });
    });
});
```

Figure 23: Display API example

4.3.7 Maths module

Another key component in OpenGL are vector and matrix mathematics. Again, this is not something that is in the standard template library in C++. To solve this in the platform we have added a maths module that provides vector and matrix mathematic functions. The module code for this has been adapted from the open source project gl-matrix (Glmatrix.net, 2017) which is a maths library designed for WebGL applications that is just raw JavaScript so there were no issues adding it to the platform we are creating. An example can be seen in Figure 24.

```javascript
const { mat4, vec4, vec3 } = require('maths');
const console = require('console');
const matrix = mat4.create(), vector = vec4.create();

mat4.translate(matrix, matrix, vec3.fromValues(5.0, 0.0, 0.0));
mat4.scale(matrix, matrix, vec3.fromValues(2.0, 2.0, 2.0));

vec4.add(vector, vector, vec4.fromValues(1.0, 1.0, 0.0, 0.0));
vec4.sub(vector, vector, vec4.fromValues(5.0, 0.0, 2.0, 0.0));

console.log(`Matrix: ${matrix}`);
console.log(`Vector: ${vector}, Length: ${vec4.length(vector)}`);
```

Figure 24: Maths API example

4.3.8 GL/CL Modules

The core modules that enable access to the GPU whether integrated or dedicated are the CL module and GL module which house the bindings to OpenCL and OpenGL. 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 25, you will see that by using the *with* keyword which takes all data in an object and makes it available outside (see Figure 26 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.

```
const console = require('console');
const gl = require('gl');
const cl = require('cl');

with(cl && gl) {
    console.log(glGetError() == GL_NO_ERROR);
}
```

Figure 25: GL API example

```
const point = {
    x: 100,
    y: 200
};

with (point) {
    console.log('X: ' + x); // x, y are now available
    console.log('Y: ' + Y);
}
```

*Figure 26: With keyword example*

# 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 using the platform allows people to take simple concepts and apply them easily.

## 5.1 OpenGL Demos

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 27. 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 a JSON file from disk. The maths module that comes as 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 in the vertex shader. In this demonstration, we can see the added benefit of having a platform as it provides a lot of the functionality needed.
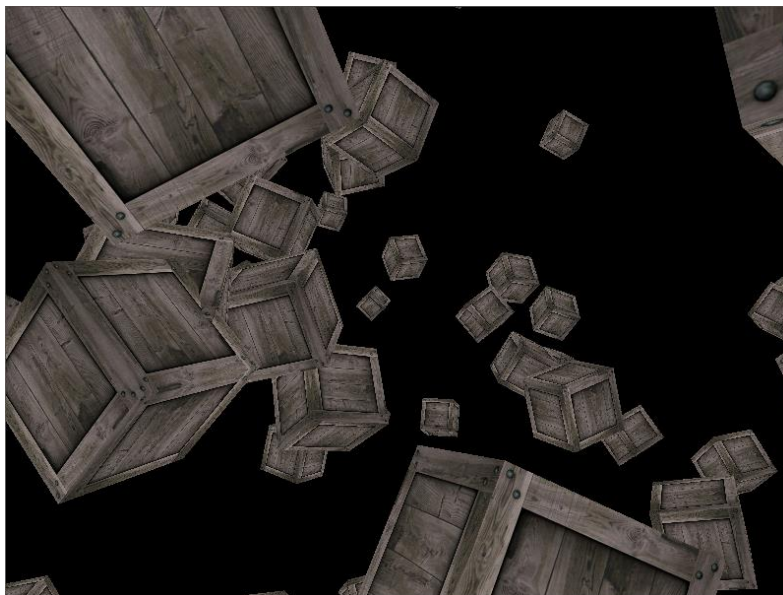


*Figure 27: 3D cubes output*

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 28 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.

```json
{
    "terrain": {
        "grid": {
            "x": 1, "y": 1, "w": 100, "h": 100
        },
        "seed": 50
    }
}
```

Figure 28: Input settings JSON

When the demonstration is run, you will see the output seen in Figure 29. The terrain is generated by pushing geometry into a Typed Array which is then directly fed to a vertex buffer 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 the 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.
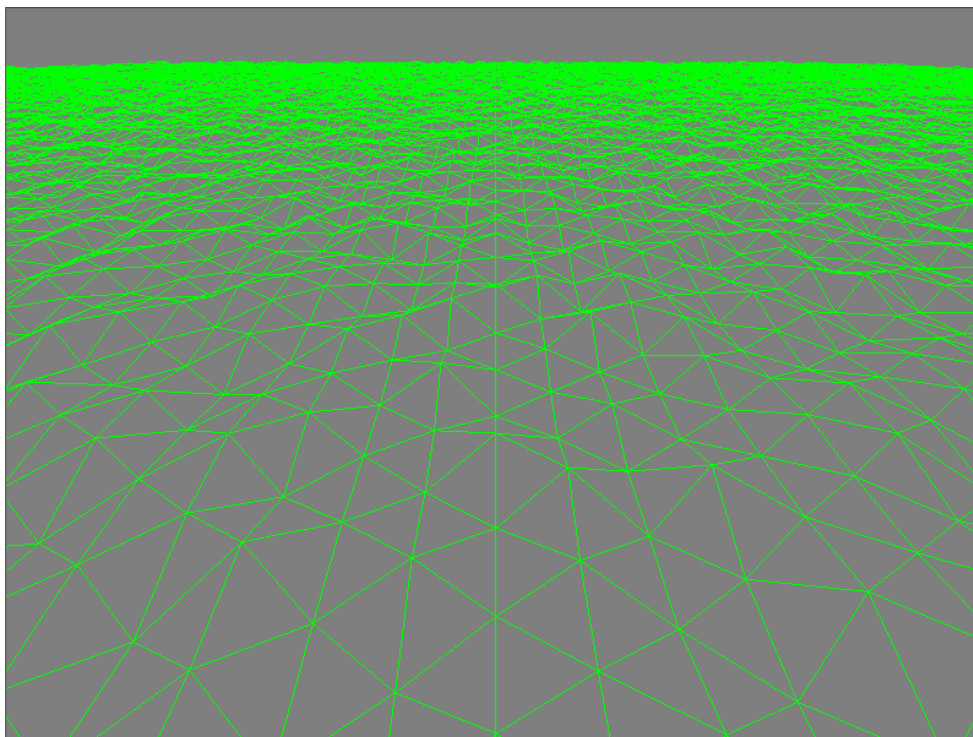
Figure 29: Terrain demo output

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 for the cube is loaded in and has pre-calculated normals for the lighting calculations. Per fragment lighting is used as it is a more accurate method for calculating a lighting value for geometry, at least when compared to per vertex lighting which calculates the lighting value for each vertex and interpolates across the surface leading to inaccuracies. To prove that the lighting is dynamic and not static the geometry is rotated on its Y axis showing how lighting is calculated per frame and not 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 in Figure 31. Figure 30 shows the fragment shader used for the lighting demonstration.

```glsl
#version 330
out vec4 colour;
in vec3 normal;

struct DirectionalLight
{
    vec3 colour, direction;
    float ambient;
};

void main()
{
  DirectionalLight sun;
  sun.direction = normalize(vec3(-5.0, -1.0, 0.0));
  sun.colour = vec3(1.0, 0.0, 0.0);
  sun.ambient = 0.25;

  float diffuse = max(0.0, dot(normalize(normal), -sun.direction));
  colour = vec4(1.0, 0.0, 0.0, 1.0) * vec4(sun.colour * (sun.ambient + diffuse), 1.0);
}
```

Figure 30: Lighting Fragment Shader
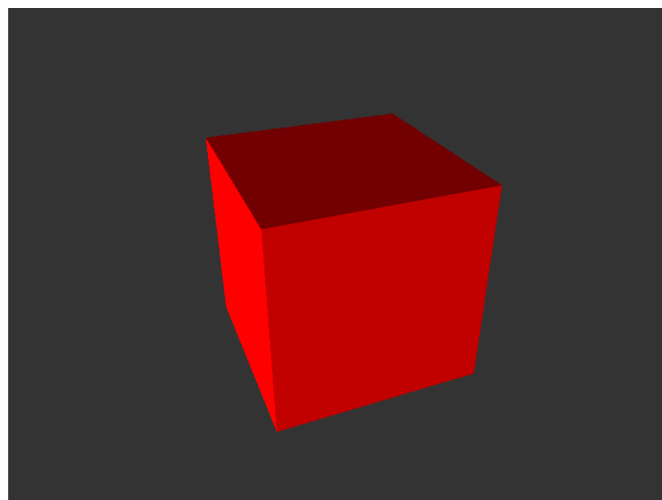
Figure 31: Lighting demonstration output

5.2 OpenCL Demos

To accompany the three OpenGL demonstrations 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 writes a grayscale version of it to disk using an OpenCL kernel. The second is a Sobel Filter demonstration which performs an edge detection test across the image again using an OpenCL kernel and finally we have a matrix demonstration that performs a matrix multiplication operation on a given matrix stored in JSON and writes the output to disk.

5.2.1 Grayscale Demo

So, the first demonstration as shown below is a simple program that reads an input image from disk, copies that memory into an OpenCL image object then executes the kernel shown in Figure 32. The mentioned kernel just takes all components in that image and calculates the sum, then divides by three, the three representing the number of components in the image, then sets all image components to that value resulting in a grayscale image (See Figure 33). This new image is then transferred from OpenCL to regular memory and then written to disk as a PNG. This program was made remarkably simple thanks to the fs module which can load and write PNG's, BMP's, and JPG's.

```
const sampler_t sampler = CLK_ADDRESS_CLAMP_TO_EDGE | CLK_FILTER_NEAREST;

__kernel void grayscale(__read_only image2d_t src, __write_only image2d_t dest)
{
  int2 pixel = (int2)(get_global_id(0), get_global_id(1));
  float4 colour = read_imagef(src, sampler, pixel);
  float total = 0.0f;
  total += colour.x;
  total += colour.y;
  total += colour.z;
  total /= 3.0f;
  total = clamp(total, 0.0f, 1.0f);
  write_imagef(dest, pixel, (float4)(total, total, total, 1.0));
}
```

Figure 32: The grayscale kernel

Figure 33: Left is the input image on the right the output

5.2.2 Sobel Filter Demo

The second demonstration as stated before is a simple Sobel Filter. It operates like the previous demonstration does. It loads an image from disk and passes it to OpenCL. However, the kernel instead of performing a sum of all components instead calculates the magnitude of change of the pixel's value resulting in edge detection. The output can be seen on the right-hand side in Figure 34 which shows the edges identified from the source image which is on the left.
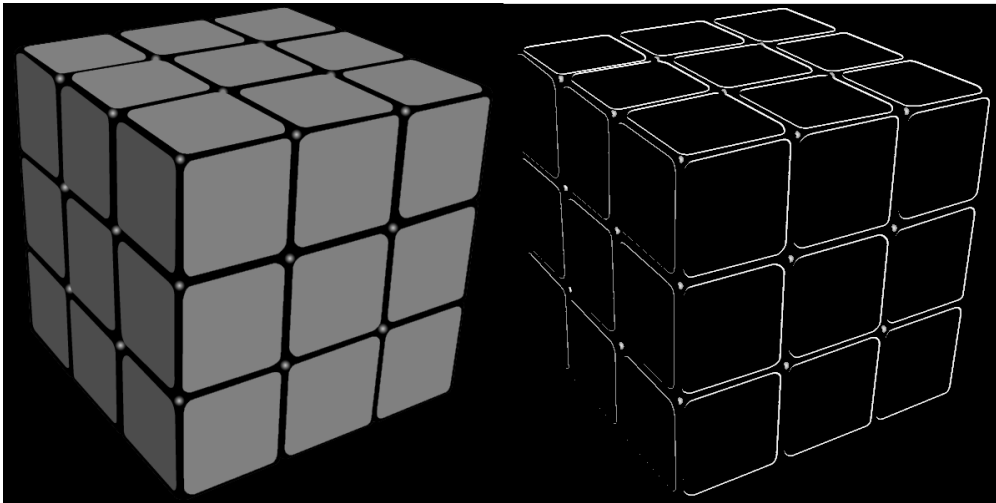
Figure 34: Left is the input image on the right the output

5.2.3 Matrix Demo

The final demo is a matrix multiplication demonstration. In Figure 35 you can see the output of the program. But how does this work? Well in Figure 36 you can see the input JSON data for the program. The first field is the multiply field which is the matrix multiplication value to be used when calculating the output. The second field is the matrices field which is an array of all the matrices we want to calculate an output for. The program loads this JSON and passes it into Typed Arrays which are then put into OpenCL buffers. Once there we execute are simple matrix multiplication kernel for each matrix specified in this list and then store the output in a new list. Once all outputs have been calculated we then write this into the previously mentioned output JSON file which can be seen in Figure 35.

```json
{
    "matrices": [
      {
        "mat4": [
          2, 0, 0, 0,
          0, 2, 0, 0,
          0, 0, 2, 0,
          0, 0, 0, 2
        ]
      }
    ]
}
```

Figure 35: Matrix output JSON

```
{
    "multiply": {
        "mat4": [
            2, 0, 0, 0,
            0, 2, 0, 0,
            0, 0, 2, 0,
            0, 0, 0, 2
        ]
    },
    "matrices": [
        {
            "mat4": [
                1, 0, 0, 0,
                0, 1, 0, 0,
                0, 0, 1, 0,
                0, 0, 0, 1
            ]
        }
    ]
}
```

Figure 36: Input matrix JSON

5.3 Demo Results

These demonstrations have proven that the platform serves its purpose. These programs would have been much more complex if they were written in C++. That can be proved by the underlying code that sits inside the platform. These six demonstrations were varied and by looking at resources online we can see with the platform they are far easier to write. This is exactly what the platform was meant to show. These demonstrations are simple in theory and simple to write. The platform has achieved this goal.

6.0 Future Work

There is a lot of future work that could be undertaken 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 languages could also be an option. Different interfaces could be tried as well dropping the CLI and providing it with an interactive GUI UI. All of this is discussed in detail as there is plenty of scope for additional work and research that could greatly improve on the project that has been presented in this report.

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 does not 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 37, where you can see that there would be a very different approach when embedding this API for use by JavaScript.

```
IDirect3D9* d3d = Direct3DCreate9(D3D_SDK_VERSION);
D3DPRESENT_PARAMETERS d3dpp;
memset(&d3dpp, 0, sizeof(d3dpp));
d3dpp.Windowed = FALSE;
d3dpp.SwapEffect = D3DSWAPEFFECT_DISCARD;
d3dpp.hDeviceWindow = hWnd;
d3dpp.BackBufferFormat = D3DFMT_X8R8G8B8;
d3dpp.BackBufferWidth = 800;
d3dpp.BackBufferHeight = 600;

d3d->CreateDevice(D3DADAPTER_DEFAULT D3DDEVTYPE_HAL, hWnd, 32, &d3dpp, &d3ddev);
```

Figure 37: DirectX device initialisation example

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 (see Figure 38) 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.

```
// __constant__ Makes the data available on the GPU
__constant__ int numbers[5] = { 1, 2, 3, 4, 5 };
```

Figure 38: CUDA annotation example

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 provided 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 are 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 using Typescript which is compiled down to JavaScript but provides features such as type safety to make sure simple errors are caught at compile time. What is more, is that with Typescript you could write your own API definition files that tell the Typescript compiler what are valid errors when using the platform. An example of the type safety that Typescript provides can be seen in Figure 39.

```typescript
let bool: boolean = false;
let word: string = "HelloWorld";
let num: number = 100

// Type annotations for both parameter and return type
function timesTen(value: number) : number { return value * 10; }

timesTen(num);   // Fine
timesTen(word);  // Compile error
timesTen(bool);  // Compile error
```

Figure 39: Typescript optional type annotations demonstrated

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 and no lengthy C++ compilation stage. 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. This work could build on top of the C++ bindings that OpenCL already offers to simplify use and provide an API that is more native to C++'s language design.

```cpp
std::vector<cl::Device> devices;

cl::Platform p = cl::Platform::getDefault();
p.getDevices(CL_DEVICE_TYPE_GPU, &devices);
```

Figure 40: OpenCL C++ Wrappers example

6.5 Alternative Languages

In this project, 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.

6.6 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 (Mathworks.com, 2017) 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 expands 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 provided a layer of useful utilities for GPU programmers and the ability to write GPU experiments in a dynamic way. Secondly, several demonstrations that not only proved its effectiveness in real world use, but the advantages of the idea and concept. Thirdly, an in-depth literature review into the current state of GPUs on a desktop level and the APIs that are used by them. Finally, an in depth look at JavaScript as it exists today through the recent ECMAScript standard and the design and implementation details of V8 arguably the world's most popular JavaScript compiler.

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 was done to ensure the feasibility of the project as there were serious doubts at the start as to whether high performance rendering applications could be accomplished on a desktop level with JavaScript. This initial development allowed a good amount of time to be spent on a review of technical areas that would form the basis for the project. This included an investigation into JavaScript and its latest standard, GPUs from both a hardware and software perspective and V8 the central pillar of the project that enables all of this. Several demonstrations were also made, six in total with three being OpenGL based and the other three being OpenCL based. Not only were they good to highlight the platform in action but they 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 as could have the technical implementation. More research could have been undertaken to explore why knowledge of GPGPU is lacking across the common developer and what could be done to address this. 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 were not 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 is not available on other platforms.

The project itself exposed the inner workings of JavaScript and this will be a very valuable skill as JavaScript gets popular day after day and knowing it a level that most people do not will certainly be an advantage. The ability to get hands on experience with V8 was also welcome. As web is so important to today's world it is great to look under the hood and understand the optimizations that V8 enables to make JavaScript as fast as it is today. So, if nothing else these learning points have certainly made a fantastic addition to my personal development.

Overall, 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. Additionally, the research and development are unique. Citing relevant references were difficult to find in relation to V8 and JavaScript. There clearly is not a lot of research in this area with regards to the advantages and design of both and this should hopefully be a good starting point for that. This project can be built upon in future in many ways and this gives a good starting point for future research that looks at crafting bespoke platforms or environments for certain areas.

8.0 Conclusion

This project culminated in a number points that can be made about the technologies used in this project and the overarching topic that is general purpose graphics programming. Introducing high level concepts to enable GPU access has proven to be of great success. Modular components for operations such as vector and matrix mathematics and Typed Arrays for the storage of vertex data greatly simplified example applications and led to a better declaration of intent which made the program more readable. Additionally, modular components for image loading and Typed Arrays for image buffers again led to a better declaration of intent and cut down the boilerplate required to access the hardware. By doing so it is clear that we made GPGPU less intimidating and more approachable. It is no wonder then that recent versions of the OpenCL standard build off this idea of reusable classes for interaction with its API over regular C calls. However, taken further to encompass not only the API itself but the operations required to load, translate and pass data to this API we can make an incredibly exciting technology as approachable as a hello world program in the simplest of programming languages.

Increased approachability did not just lead to an easier time getting on with this technology but it leads to a much better advantage. With the code being easier to write it becomes easier to prove the applicability of the technology. GPGPU no longer becomes a specialist area as regular programmers can easy as one two three write parallel code to speed up existing sequential code. An increase in popularity would lead to faster software, increased use cases and more investment in the very technology that can speed up applications in a way without the need to upgrade hardware.

Remarkably JavaScript has been widely successful with its integration into the platform. It has quite successfully replaced what would have been C++ code with ease. It has also proved to be a very versatile language with the standard introducing objects like Typed Arrays allow the language to be used in a wide range of scenarios outside what it was previously envisaged for. It is very impressive to see how flexible JavaScript has been when integrating the language with the OpenCL and OpenGL APIs. It makes one wonder of the direction of travel for GPGPU. We have seen first-hand the advantages of making access to this hardware easier through higher level languages and how by presenting an all in one package how simple a demo could be to make in such little time. It certainly makes things easier and allows more time to focus on what you want to do.

JavaScript's speed and ability to be embedded into different areas is in large part thanks to Google's V8 JavaScript compiler. Its focus on speed and optimizations has led to many breakthroughs allowing the dynamic nature that makes JavaScript special possible without loss of performance. V8's hidden classes optimisation enables unseen property access speeds but does not stop property mutation. As does its garbage collection strategy which aims for small clean-up phases rather than long pauses at random intervals in the applications lifetime.

Utilizing GPUs especially the ones found already in many computers today will be of great importance if we are to capitalize on these fast processors that remain dormant for most applications. This project lays at an approach that would allow the common developer to experiment and get familiar with the hardware before transferring any kernels or shaders written into the application they will be working on. We found that the platform makes experimentation and learning easier than ever before and would aid in accelerating adoption of GPGPU techniques.

References

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

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

Amd.com. 2017. Radeon™ Dual Graphics Cards [Accessed 12 Mar. 2017]. Available from:
http://www.amd.com/en-us/innovations/software-technologies/dual-graphics

Apple.com. 2016. Apple.com. [Online]. [18 December 2016]. Available from:
https://developer.apple.com/metal/

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

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

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

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

Flowtype. (2017). A static type checker for JavaScript. [online] Available at:
https://flowtype.org/ [Accessed 12 Mar. 2017].

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

Glmatrix.net. (2017). glMatrix. [online] Available at: http://glmatrix.net/ [Accessed 28 Feb.
2017].

Github Repository. [online] Available at: https://github.com/wt-student-
projects/computing-honours-project [Accessed 10 Mar. 2017].

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

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

Khronos.org. 2016. Khronos.org. [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.

Mathworks.com. (2017). MATLAB - MathWorks. [online] Available at: https://www.mathworks.com/products/matlab.html [Accessed 12 Mar. 2017].

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

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

Nickolls, J. and Dally, W.J., 2010. The GPU computing era. IEEE micro, 30(2).

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

Nvidia.com. 2016. Nvidia.com. [Online]. [15 December 2016]. Available from: http://www.nvidia.com/object/IO_20020111_5424.html

Nvidia.com. 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/

Nvidia.co.uk. (2017). NVIDIA Optimus Technology | NVIDIA UK. [online] Available at: http://www.nvidia.co.uk/object/optimus_technology_uk.html [Accessed 12 Mar. 2017].

Nvidia.com. 2016. Nvidia.com. [Online]. [18 December 2016]. Available from: http://www.nvidia.com/object/cuda_home_new.html

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.

Richardson, A. and Gray, A., 2008. Utilisation of the GPU architecture for HPC. EPCC, The University of Edinburgh.

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.

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

Steampowered.com. 2016. Steampowered.com. [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/

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

Thrust.github.io. (2017). Thrust - Parallel Algorithms Library. [online] Available at: https://thrust.github.io/ [Accessed 12 Mar. 2017].

Typescriptlang.org. (2017). TypeScript - JavaScript that scales. [online] Available at: https://www.typescriptlang.org/ [Accessed 12 Mar. 2017].

Unity3d.com. 2016. Unity. [Online]. [18 December 2016]. Available from: https://unity3d.com/

Vuduc, R., Chandramowlishwaran, A., Choi, J., Guney, M. and Shringarpure, A., 2010, June. On the limits of GPU acceleration. In Proceedings of the 2nd USENIX conference on Hot topics in parallelism (Vol. 13). USENIX Association.

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

W3techs.com. (2017). Usage Statistics of JavaScript for Websites, February 2017. [online] Available at: https://w3techs.com/technologies/details/cp-javascript/all/all [Accessed 28 Feb. 2017].

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.

Appendix 1 – OpenCL Grayscale Demo Source Code

```javascript
const cl = require('cl')
const fs = require('fs');

function acquireRuntime() {
  const platforms = [], devices = [];
  cl.clGetPlatformIDs(0, null, platforms);
  cl.clGetPlatformIDs(platforms.length, platforms, null);
  cl.clGetDeviceIDs(platforms[0], cl.CL_DEVICE_TYPE_ALL, 0, null, devices);
  cl.clGetDeviceIDs(platforms[0], cl.CL_DEVICE_TYPE_ALL, devices.length, devices, null);
  return { platform: platforms[0], device: devices[0] };
}

with (cl) {
  const image = fs.readImage("image.jpg"), kernelSrc = fs.read('./kernel.cl').contents;
  const { platform, device } = acquireRuntime();
  const props = [CL_CONTEXT_PLATFORM, platform, 0];
  const context = clCreateContextFromType(props, CL_DEVICE_TYPE_ALL, null, null, null);
  const commandQueue = clCreateCommandQueue(context, device, 0, null);
  const program = clCreateProgramWithSource(context, 1, kernelSrc, null, null);
  const error = {}, format = {
    image_channel_order: CL_RGBA,
    image_channel_data_type: CL_UNORM_INT8
  };

  const w = image.width, h = image.height;
  const globalWork = Uint32Array.from([1920, 1200]);
  const localWork = Uint32Array.from([16, 16]);
  const region = Uint32Array.from([w, h, 1]);
  const origin = Uint32Array.from([0, 0, 0]);
  const output = new ArrayBuffer(w * h * 4);

  clBuildProgram(program, 0, null, null, null, null);

  const kernel = clCreateKernel(program, "grayscale", null);
  const buffers = [
    clCreateImage2D(context, (1 << 2)|(1 << 5), format, w, h, 0, image.data, error),
    clCreateImage2D(context, (1 << 0), format, w, h, 0, null, error)
  ];

  clSetKernelArg(kernel, 0, Uint32Array.BYTES_PER_ELEMENT, buffers[0]);
  clSetKernelArg(kernel, 1, Uint32Array.BYTES_PER_ELEMENT, buffers[1]);
  clEnqueueNDRangeKernel(commandQueue, kernel, 2, null, globalWork, localWork, 0, null, null);
  clEnqueueReadImage(commandQueue, buffers[1], CL_TRUE, origin, region, 0, 0, output, 0, null, null);
  clReleaseCommandQueue(commandQueue);
  clReleaseKernel(kernel);
  clReleaseContext(context);
  clReleaseProgram(program);

  fs.writeImage("grayscale.png", output, w, h);
  fs.freeImage(image);
}
```

Appendix 2 – OpenCL Matrix Demo Source Code

```javascript
const maths = require('maths'), cl = require('cl'), fs = require('fs');
const matrixData = fs.readJson('in-mat4.json'), out = [];
const { platform, device } = acquireRuntime();

function acquireRuntime() {
    const platforms = [], devices = [];
    cl.clGetPlatformIDs(0, null, platforms);
    cl.clGetPlatformIDs(platforms.length, platforms, null);
    cl.clGetDeviceIDs(platforms[0], cl.CL_DEVICE_TYPE_ALL, 0, null, devices);
    cl.clGetDeviceIDs(platforms[0], cl.CL_DEVICE_TYPE_ALL, devices.length, devices, null);
    return { platform: platforms[0], device: devices[0] };
}

with (cl) {
    const properties = [CL_CONTEXT_PLATFORM, platform, 0];
    const context = clCreateContextFromType(properties, CL_DEVICE_TYPE_ALL, null, null, null);
    const commandQueue = clCreateCommandQueue(context, device, 0, null);
    const program = clCreateProgramWithSource(context, 1, fs.read('matrix.cl').contents, null, null);
    const err = clBuildProgram(program, 0, null, null, null, null);
    const kernel = clCreateKernel(program, 'mul', null);
    const param = CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR;

    matrixData.matrices.forEach(matrix => {
        const multiply = Float32Array.from(matrixData.multiply.mat4);
        const input = Float32Array.from(matrix.mat4);
        const output = new Float32Array(input.length);
        const aBuffer = clCreateBuffer(context, param, input.byteLength, input, null);
        const bBuffer = clCreateBuffer(context, param, multiply.byteLength, multiply, null);
        const cBuffer = clCreateBuffer(context, CL_MEM_WRITE_ONLY, output.byteLength, null, null);

        clSetKernelArg(kernel, 0, Uint32Array.BYTES_PER_ELEMENT, aBuffer);
        clSetKernelArg(kernel, 1, Uint32Array.BYTES_PER_ELEMENT, bBuffer);
        clSetKernelArg(kernel, 2, Uint32Array.BYTES_PER_ELEMENT, cBuffer);
        clEnqueueNDRangeKernel(commandQueue, kernel, 2, null, Uint32Array.from([4, 4]), null, 0, null, null);
        clEnqueueReadBuffer(commandQueue, cBuffer, CL_TRUE, 0, output.byteLength, output, 0, null, null);

        clReleaseMemObject(aBuffer);
        clReleaseMemObject(bBuffer);
        clReleaseMemObject(cBuffer);

        out.push({ "mat4": Array.from(output) });
    });

    clReleaseKernel(kernel);
    clReleaseProgram(program);
    clReleaseCommandQueue(commandQueue);
    clReleaseContext(context);
}

fs.writeJson('out-mat4.json', { "matrices": out });
```

Appendix 3 – OpenCL Sobel Demo Source Code

```javascript
const cl = require('cl'), fs = require('fs');

function acquireRuntime() {
  const platforms = [], devices = [];
  cl.clGetPlatformIDs(0, null, platforms);
  cl.clGetPlatformIDs(platforms.length, platforms, null);
  cl.clGetDeviceIDs(platforms[0], cl.CL_DEVICE_TYPE_ALL, 0, null, devices);
  cl.clGetDeviceIDs(platforms[0], cl.CL_DEVICE_TYPE_ALL, devices.length, devices, null);
  return { platform: platforms[0], device: devices[0] };
}

const { platform, device } = acquireRuntime();
const source = fs.read('./kernel.cl');

with (cl) {
  const properties = [CL_CONTEXT_PLATFORM, platform, 0];
  const context = clCreateContextFromType(properties, CL_DEVICE_TYPE_ALL, null, null, null);
  const cmdQueue = clCreateCommandQueue(context, device, 0, null);
  const program = clCreateProgramWithSource(context, 1, source.contents, null, null);
  const err = clBuildProgram(program, 0, null, null, null, null);
  const kernel = clCreateKernel(program, 'sobel', null);
  const img = fs.readImage('image.png');
  const format = {}, error = {};
  format.image_channel_order = CL_RGBA;
  format.image_channel_data_type = CL_UNORM_INT8;

  const output = new ArrayBuffer(img.width * img.height * 4);
  const region = Uint32Array.from([img.width, img.height, 1]);
  const global = Uint32Array.from([600, 600]);
  const local = Uint32Array.from([1, 1]);
  const origin = Uint32Array.from([0, 0, 0]);
  const imageMemory = [
    clCreateImage2D(context, (1 << 2)|(1 << 5), format, img.width, img.height, 0, img.data, error),
    clCreateImage2D(context, CL_MEM_READ_WRITE, format, img.width, img.height, 0, null, error)
  ];

  clSetKernelArg(kernel, 0, Uint32Array.BYTES_PER_ELEMENT, imageMemory[0]);
  clSetKernelArg(kernel, 1, Uint32Array.BYTES_PER_ELEMENT, imageMemory[1]);
  clEnqueueNDRangeKernel(cmdQueue, kernel, 2, null, global, local, 0, null, null);
  clEnqueueReadImage(cmdQueue, imageMemory[1], CL_TRUE, origin, region, 0, 0, output, 0, null, null);

  clReleaseKernel(kernel);
  clReleaseCommandQueue(cmdQueue);
  clReleaseContext(context);
  clReleaseProgram(program);

  fs.writeImage('output.png', output, img.width, img.height);
  fs.freeImage(img);
}
```

Appendix 4 – OpenGL Cubes Demo Source Code

```javascript
const { openWindow } = require('display');
const maths = require('maths');
const gl = require('gl');
const fs = require('fs');

const shaders = { vs: fs.read('./vert.glsl'), fs: fs.read('./frag.glsl') };
const geometry = fs.readJson('./cube.json');
const texture = fs.readImage('./crate.jpg');

const translations = [], rotations = [];
const projection = maths.mat4.create();
const model = maths.mat4.create();
const view = maths.mat4.create();

function createShader(shaderType, shaderSource) {
    with (gl) {
        const shader = glCreateShader(shaderType);
        glShaderSource(shader, 1, shaderSource, 0);
        glCompileShader(shader);
        return shader;
    }
}

function createShaders() {
    with (gl) {
        const program = glCreateProgram();
        const vs = createShader(GL_VERTEX_SHADER, shaders.vs.contents);
        const fs = createShader(GL_FRAGMENT_SHADER, shaders.fs.contents);
        glAttachShader(program, vs);
        glAttachShader(program, fs);
        glLinkProgram(program);
        glUseProgram(program);
        return program;
    }
}

function createBuffer(data, index, count) {
    with (gl) {
        const vertexBuffer = new Uint32Array(1);
        glGenBuffers(1, vertexBuffer);
        glBindBuffer(GL_ARRAY_BUFFER, vertexBuffer[0]);
        glBufferData(GL_ARRAY_BUFFER, data.byteLength, data, GL_STATIC_DRAW);
        glEnableVertexAttribArray(index);
        glVertexAttribPointer(index, count, GL_FLOAT, GL_FALSE, 0, 0);
        return vertexBuffer;
    }
}

function createTexture(texture) {
    with (gl) {
        const textureID = new Uint32Array(1);
        glGenTextures(1, textureID);
        glBindTexture(GL_TEXTURE_2D, textureID[0]);
        glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, texture.width, texture.height, 0, GL_RGBA, GL_UNSIGNED_BYTE, texture);
        glTexParameter(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
        glTexParameter(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
        return textureID;
    }
}
```

```javascript
function onRender(program, rotation) {
    const projectionLocation = gl.glGetUniformLocation(program, "projection");
    const modelLocation = gl.glGetUniformLocation(program, "model");
    const viewLocation = gl.glGetUniformLocation(program, "view");

    for (let i = 0; i < translations.length; i++) {
        const { x, y, z } = translations[i];
        const scaleValue = 0.3 + x / 50.0;
        const scaleVector = maths.vec3.fromValues(scaleValue, scaleValue, scaleValue);
        const rotationVector = maths.vec3.fromValues(rotations[i].x, rotations[i].y, rotations[i].z);

        maths.mat4.identity(model);
        maths.mat4.translate(model, model, maths.vec3.fromValues(x, y, z));
        maths.mat4.scale(model, model, scaleVector);
        maths.mat4.rotate(model, model, maths.glMatrix.toRadian(rotation), rotationVector);

        gl.glUniformMatrix4(projectionLocation, 1, gl.GL_FALSE, Float32Array.from(projection));
        gl.glUniformMatrix4(modelLocation, 1, gl.GL_FALSE, Float32Array.from(model));
        gl.glUniformMatrix4(viewLocation, 1, gl.GL_FALSE, Float32Array.from(view));
        gl.glDrawArrays(gl.GL_TRIANGLES, 0, 36);
    }
}

function onLoad() {
    const vao = new Uint32Array(1);
    gl.glGenVertexArray(1, vao);
    gl.glBindVertexArray(vao[0]);
    gl.glEnable(gl.GL_DEPTH_TEST);
    gl.glEnable(gl.GL_TEXTURE_2D);

    createTexture(texture);
    createBuffer(Float32Array.from(geometry.cube), 0, 3);
    createBuffer(Float32Array.from(geometry.colour), 1, 3);
    createBuffer(Float32Array.from(geometry.uvs), 2, 2);

    maths.mat4.perspective(projection, maths.glMatrix.toRadian(45.0), 4.0 / 3.0, 0.1, 100.0);
    maths.mat4.lookAt(view, maths.vec3.fromValues(4, 3, -3),
        maths.vec3.fromValues(0, 0, 0),
        maths.vec3.fromValues(0, 1, 0)
    );

    for (let i = 0; i < 100; ++i) {
        translations.push({ x: Math.random() * 10 - 5, y: Math.random() * 10 - 5, z: Math.random() * 10 - 5 });
        rotations.push({ x: Math.random(), y: Math.random(), z: Math.random() });
    }
}

openWindow({ x: 450, y: 250, w: 800, h: 600 }, window => {
    window.onClose(() => fs.freeImage(texture));
    window.setTitle('Cubes Example');
    window.show();
    window.enableOpenGL();

    onLoad();

    let program = createShaders();
    let rotation = 1.0;
    window.onFrame(() => {
        gl.glClear(gl.GL_COLOR_BUFFER_BIT | gl.GL_DEPTH_BUFFER_BIT);
        gl.glClearColor(0.0, 0.0, 0.0, 0.0);

        onRender(program, rotation);
        rotation += 0.5;

        window.swapBuffers();
    });
});
```

Appendix 5 – OpenGL Lighting Demo Source Code

```javascript
const { openWindow } = require('display');
const maths = require('maths');
const gl = require('gl');
const fs = require('fs');

const geometry = fs.readJson('./cube.json');
const shaders = {
    vs: fs.read('lighting.vs.glsl'),
    fs: fs.read('lighting.fs.glsl')
};

function createShader(shaderType, shaderSource) {
    with (gl) {
        const shader = glCreateShader(shaderType);
        glShaderSource(shader, 1, shaderSource, 0);
        glCompileShader(shader);
        return shader;
    }
}

function configureDisplay(callback) {
    openWindow({ x: 450, y: 250, w: 800, h: 600 }, window => {
        window.setTitle('Basic Lighting');
        window.show();
        window.enableOpenGL();

        callback(window);
    });
}

function createProgram() {
    with (gl) {
        const vs = createShader(GL_VERTEX_SHADER, shaders.vs.contents);
        const fs = createShader(GL_FRAGMENT_SHADER, shaders.fs.contents);

        const program = glCreateProgram();
        glAttachShader(program, vs);
        glAttachShader(program, fs);
        glLinkProgram(program);
        glUseProgram(program);
        return program;
    }
}

function createBuffer(data, index, count) {
    with (gl) {
        const vertexBuffer = new Uint32Array(1);
        glGenBuffers(1, vertexBuffer);
        glBindBuffer(GL_ARRAY_BUFFER, vertexBuffer[0]);
        glBufferData(GL_ARRAY_BUFFER, data.byteLength, data, GL_STATIC_DRAW);
        glEnableVertexAttribArray(index);
        glVertexAttribPointer(index, count, GL_FLOAT, GL_FALSE, 0, 0);
        return vertexBuffer;
    }
}
```

```javascript
configureDisplay(window => {
    let projection = maths.mat4.create(), view = maths.mat4.create();
    let program = createProgram(), vao = new Uint32Array(1);
    let rotation = 0;

    with (gl) {
        glGenVertexArray(1, vao);
        glBindVertexArray(vao[0]);
        glEnable(GL_DEPTH_TEST);

        createBuffer(Float32Array.from(geometry.cube), 0, 3);
        createBuffer(Float32Array.from(geometry.normals), 1, 3);

        maths.mat4.perspective(projection, maths.glMatrix.toRadian(45.0), 4.0 / 3.0, 0.1, 100.0);
        maths.mat4.lookAt(view,
            maths.vec3.fromValues(4, 3, -3),
            maths.vec3.fromValues(0, 0, 0),
            maths.vec3.fromValues(0, 1, 0)
        );

        window.onFrame(() => {
            glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
            glClearColor(0.2, 0.2, 0.2, 0.2);

            const projectionLocation = glGetUniformLocation(program, "projection");
            const modelLocation = glGetUniformLocation(program, "model");
            const viewLocation = glGetUniformLocation(program, "view");
            const model = maths.mat4.create();

            maths.mat4.rotate(model, model, maths.glMatrix.toRadian(rotation), maths.vec3.fromValues(0.0, -1.0, 0.0));

            glUniformMatrix4(projectionLocation, 1, GL_FALSE, Float32Array.from(projection));
            glUniformMatrix4(modelLocation, 1, GL_FALSE, Float32Array.from(model));
            glUniformMatrix4(viewLocation, 1, GL_FALSE, Float32Array.from(view));
            glDrawArrays(GL_TRIANGLES, 0, geometry.cube.length / 3);

            window.swapBuffers();
            rotation += 0.25;
        });
    }
});
```

Appendix 6 – OpenGL Terrain Demo Source Code

```javascript
const { openWindow } = require('display');
const maths = require('maths');
const gl = require('gl');
const fs = require('fs');

const terrain = fs.readJson('./terrain.json');
const shaders = {
    vs: fs.read('./terrain.vs.glsl'),
    fs: fs.read('./terrain.fs.glsl')
};

function createShader(shaderType, shaderSource) {
    with (gl) {
        const shader = glCreateShader(shaderType);
        glShaderSource(shader, 1, shaderSource, 0);
        glCompileShader(shader);
        return shader;
    }
}

Math.seed = seed => {
    return () => {
        if (seed == 0) return 0;
        seed = Math.sin(seed) * 10000;
        return (seed - Math.floor(seed)) / 2.0
    };
};

const makeTerrain = desc => {
    const incX = desc.grid.x, incY = desc.grid.y;
    const random = Math.seed(desc.seed);
    const vertices = [], heights = {};

    for (let y = 0; y < desc.grid.h; y += desc.grid.y) {
        for (let x = 0; x < desc.grid.w; x += desc.grid.x) {
            heights[x + incX] = heights[x + incX] || {};
            heights[x] = heights[x] || {};
            heights[x + incX][y + incY] = random();
            heights[x + incX][y] = random();
            heights[x][y + incY] = random();
            heights[x][y] = random();
        }
    }

    for (let y = 0; y < desc.grid.h; y += desc.grid.y) {
        for (let x = 0; x < desc.grid.w; x += desc.grid.x) {
            vertices.push(x, heights[x][y], y);
            vertices.push(x + incX, heights[x + incX][y], y);
            vertices.push(x + incX, heights[x + incX][y + incY], y + incY);
            vertices.push(x + incX, heights[x + incX][y + incY], y + incY);
            vertices.push(x, heights[x][y + incY], y + incY);
            vertices.push(x, heights[x][y], y);
        }
    }

    return vertices;
}
```

```javascript
function createProgram() {
    with (gl) {
        const program = glCreateProgram();
        glAttachShader(program, createShader(GL_VERTEX_SHADER, shaders.vs.contents));
        glAttachShader(program, createShader(GL_FRAGMENT_SHADER, shaders.fs.contents));
        glLinkProgram(program);
        glUseProgram(program);
        return program;
    }
}

openWindow({ x: 450, y: 250, w: 800, h: 600 }, window => {
    window.setTitle('Terrain Generation Example');
    window.show();
    window.enableOpenGL();

    with (gl) {
        const vertexArrayObject = new Uint32Array(1), vertexBuffer = new Uint32Array(1);
        const terrainData = makeTerrain(terrain.terrain);
        const vertexData = Float32Array.from(terrainData);
        const program = createProgram();

        glGenVertexArray(1, vertexArrayObject);
        glBindVertexArray(vertexArrayObject[0]);
        glEnable(GL_DEPTH_TEST);
        glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
        glGenBuffers(1, vertexBuffer);
        glBindBuffer(GL_ARRAY_BUFFER, vertexBuffer[0]);
        glBufferData(GL_ARRAY_BUFFER, vertexData.byteLength, vertexData, GL_STATIC_DRAW);
        glEnableVertexAttribArray(0);
        glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, 0);

        const projection = maths.mat4.create();
        const model = maths.mat4.create();
        const view = maths.mat4.create();

        maths.mat4.perspective(projection, maths.glMatrix.toRadian(45.0), 4.0 / 3.0, 0.1, 100.0);
        maths.mat4.lookAt(view,
            maths.vec3.fromValues(0, 4.5, 0),
            maths.vec3.fromValues(8, 0, 8),
            maths.vec3.fromValues(0, 1, 0)
        );

        window.onFrame(() => {
            const projectionLocation = glGetUniformLocation(program, "projection");
            const modelLocation = glGetUniformLocation(program, "model");
            const viewLocation = glGetUniformLocation(program, "view");

            glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
            glClearColor(0.5, 0.5, 0.5, 0.5);
            glUniformMatrix4(projectionLocation, 1, GL_FALSE, Float32Array.from(projection));
            glUniformMatrix4(modelLocation, 1, GL_FALSE, Float32Array.from(model));
            glUniformMatrix4(viewLocation, 1, GL_FALSE, Float32Array.from(view));
            glDrawArrays(GL_TRIANGLES, 0, vertexData.length / 3);

            window.swapBuffers();
        });
    }
});
```