Web based 3D Graphics using Dart

A thesis presented in partial fulfilment of the requirements

for the degree of

Doctor of Philosophy

in

Computer Science

at Massey University, Albany

New Zealand.

Timothy McMullen

2019

Abstract

The proportion of the population that has grown up with unlimited access to the internet and portable digital devices is ever increasing. Accompanying this growth are advances in web-based and mobile technologies that make platform independent applications more viable. Graphical applications, in particular, are popular with users but as of yet have remained relatively underdeveloped for platform independence due to their complex nature, and device requirements. This research combines web-based technologies to create a framework for developing scalable graphical environments while ensuring a suitable level of performance across all device types. The web programming language Dart provides a method for achieving execution across a range of devices with a single implementation. Working alongside Dart, WebGL manages the processing needs for the graphical elements, which are provided by content generative algorithms: the diamond square algorithm, Perlin noise, and the shallow water simulation. The content algorithms allow for some flexibility in the scale of the application, which is expanded upon by benchmarking device performance and the inclusion of the asset controller that manages what algorithm is used to generate content, and at what quality and size. This allows the application to achieve optimal performance on a range of devices from low-end mobile devices to high-end PCs. An input controller further supports platform independence by allowing for a range of input types and the addition of new input types as technology develops. The combination of these technologies and functionalities result in a framework that generates 3d scenes on any given device, and can alter automatically for optimal performance, or according to predefined developer metrics for emphasis on particular criteria. Input management functionality and web-based computing mean that as technology advances and new devices are developed and improved, applications do not need redevelopment, and compromises in features and functionality are only limited by device processing power and on an individual basis. This framework serves as an example of how a range of technolo-

gies and algorithms can be knitted together to design performant solutions for platform independent applications.

Acknowledgements

First and foremost I would like to thank Dr Daniel Playne for his endless patience, support and friendship during my PhD study and research. I would also like to thank Prof Ken Hawick for introducing me to computer graphics, and persuading me to embark on this journey.

I am grateful for the support of the Massey University Computer Science department, in particular to Chris Scogings, and all my fellow doctoral candidates.

Last but not least, I would like to thank my family. In particular, I am grateful for my parents John and Dianne for their support in this part of my life, as well as everywhere else. I am especially thankful of the help my fiance, Natalie.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The aim of this thesis is to research methods for creating web-based 3D platform independent graphical applications. Designing and creating a graphical application to run on multiple devices is challenging and time consuming. New devices are being released multiple times a year, and the boundaries of technology are constantly being pushed; tasks that were once complex and time consuming can now be done in a fraction of a second using just your phone[78]. We are now able to run complex simulations on a range of devices, yet we use them for basic tasks, such as checking emails or browsing the internet. Many of the more impressive applications are initially designed for one particular device, requiring the same application to be redeveloped multiple times to work with each device. Platform independence gives us the opportunity to create applications that can achieve a high level performance regardless of their technical specifications.

Currently, to have an application run across a range of devices requires it to be written in a language that is then interpreted by a given virtual machine[113]. Alternatively, an application can be written in a high level language, or using a framework that is then converted to various other supported languages, allowing for a given application to run across a range of devices. Each instance of code created would also need to be designed to interpret the graphical nature of the application, and the consequent effect on the

GPU.[114].

Previously mobile phones were used for calling and texting, personal computers (PCs) and laptops were used in the office, and for gaming, and consoles were hooked up to TVs and used for entertainment. Then consoles became smaller and portable, in the form of devices, such as Game Boys; TVs became smart, and phones were produced with touch screens. Now people are able to communicate with almost anyone almost anywhere in the world using a myriad of technology available on mobile devices, PCs, laptops, consoles, and even TVs. Websites are designed with mobile users in mind; tablets lurk in a grey area of being more portable than a computer but less powerful, and having more features than a mobile phone, but without being as compact; and there are endless forms of entertainment and distraction ready to be downloaded on a whim.

Despite all of this there is no consistency in the availability of programs, services, and applications across different device types. An iPhone user cannot be sure that an app they love will be be available to their Android-using friend. A Windows gamer is not guaranteed to be able to enjoy the same products as their friend who plays on a Mac. A long-time fan may not be able to continue playing when the next version of their favourite game comes out due to their device not meeting minimum system requirements.

Satisfying the goals of this thesis requires a new way of developing and deploying graphical platform independent applications. The objectives of this research are a framework that demonstrates the following,

- Performs at a real time interactive speed of a minimum of 60 fps (frames per second)
- Can run in a multithreaded implementation that generates a base environment faster than the same environment in a sequential implementation
- Can run on any device that has a web browser that is 3D graphics-capable
- Utilises web-based technologies to create a 3D graphical scene (without the use of

downloaded assets)

Technologies such as HTML and JavaScript can be launched across multiple devices using the internet to create graphical applications, and lead in the creation of various third party libraries. These libraries, although they simplify the process of using 2D or 3D graphics, tend to introduce inefficiencies into the application or system through needing to support a range of functions[104], some of which have become redundant. These libraries are also unable to avoid or lessen the need to transfer data to a user at runtime.

Using predefined simulations to create 3D graphics serves as an alternative to needing to download models or environments. This type of system requires a set of predefined algorithms to be used to create and run a given simulation. The simulation needs to be able to be adapted at runtime and change based on various settings. The advantage of using simulations like these is that a model is created, not downloaded, and can be adapted based on given system information. This in turn creates a more lightweight rendering process and allows for a more optimised graphical application.

This thesis explores various aspects of assorted technologies for use in a range of effects, as each one contributes differently to achieve the desired system. The idea of platform independence is a core feature of this thesis, as it is fundamental in allowing for an application to be run across a range of devices. To enable this, web technologies are heavily used, relying on newer feature sets such as HTML5, and WebGL[117] The use of simulations is central to improving performance, and becomes necessary to allow for an optimal design. As this research is also concerned with creating a graphical application various graphical API's are used along with different techniques to improve performance.

1.1 Platform Independence

The inspiration for platform independence comes from software that can perform regardless of the specific technology that is used to implement it [1]. This thesis is concerned with applying that method of developing to create a graphical application wherein the device being used can change without the need to redevelop any code. The first requirement is to create a code base that can run across a range of devices and optimise itself to allow for the ideal performance for the given device and user. There are several questions that must be asked and answered to better understand the requirements of achieving the goal. Firstly it needs to be understood what can be gained from a platform independent implementation. Then the most common ways of implementing a piece of platform independent code must be considered along with the pros and cons of developing platform independent applications.

Platform independence allows for a developer to create software that can be run across a range of systems and devices. The two main methods of achieving this are through the use of a virtual machine (VM), or using a programming language that is then compiled into a different program for each device architecture[5][23]. Both approaches have various general advantages and disadvantages over each other, which are extended further when compared against natively written applications designed for the given system[7].

When creating applications that run within a VM to be platform independent the typical process is the code that is passed into the VM is then interpreted into code for the CPU. This process creates a delay in the flow of data from the software through the system, and, because of this, most applications written in this manner become less efficient[7][56]. This can also occur as the code that is converted may not directly translate into something that the device's CPU can process, requiring it to be further translated in order to accommodate these situations[31].

The other option for producing code to run across a range of systems is by writing

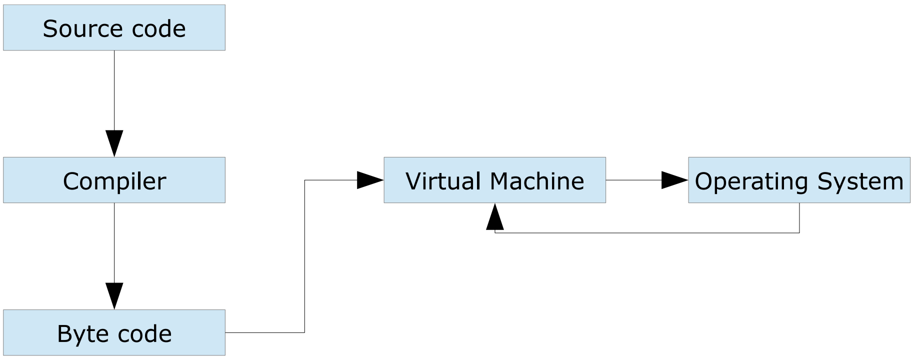


Figure 1.1: The diagram above show a simplified virtual machine, from code compliation, to platform execution.

an application in a framework such as QT [29] which is then converted and compiled into another programming language that a given device supports. The application written will then be complied into several different applications, each one designed for different hardware. The drawback of this is approach is that the interpretation completed by the original language may also have functionality which requires further translating to run on a device, much like what happens with the code running through a VM; the difference is that this translation happens before the application is run [109].

Developing using a platform independent approach allows for a developer to create one application that will run across a range of devices without the need to recreate or redevelop code. This reduces the time taken to develop for a range of devices, along with ensuring a set level of standards between each application created. As previously mentioned, to create an application for a range of devices is a time-intensive undertaking. To create an application that is platform independent requires the use of a language that is then interpreted by a VM, or is translated into various other languages. Within this thesis the focus is put on the use of web-based applications, written in Dart which has its code run within the Dart VM. The Dart VM is able to take code written in Dart to produce the desired effect. This approach is not limited to only Dart for web-based languages, as the likes of JavaScript work in the same manner [73].

The upside of written applications is that a developer only needs to be concerned with one language. The hardware that the VM is run upon, or the subsequent programs will be run on, can range greatly without the developer needing to be aware of any potential changes in hardware specifications. Abstracting the hardware from the development somewhat allows for the developer to focus more on the functionality of an application rather than what a given device is capable of. This will lead to a range of devices being able to run the same applications to similar effect.

One of the shortcomings of using a VM is that the processing time is somewhat longer than it would be if the application had been written in the device's native language [56]. Another downside of this method is that adding a level of abstraction to the process leads to an extra step needing to be taken, which, in turn, affects the performance of a given application. Because of this, an application run within a VM will struggle to perform when compared to that of a natively written application. Additionally, when using a VM, some features of the hardware architecture will not be supported, so the developer and application are unable to benefit from certain hardware acceleration[107]. An example of this would be SIMD in JavaScript, or until recently the use of accelerated graphical processing through the GPU in web applications.

Platform independence has its advantages and disadvantages; however, this research identifies that the goal of allowing for a range of devices to run a given application is vital. Furthermore, the ability to reconfigure an application at runtime is a core aspect of this thesis which also contributes to overcoming some of the downsides of using a VM, such as the one used by Dart.

1.2 Web Technologies

The advances in web technologies have rapidly accelerated since the release of the HTML5 standard[2]. HTML5 offers new functionality and range of features previously not supported natively in the web[98]. Through this new functionality it becomes easier to use web technology as a basis for platform independent applications, such as ones written in JavaScript. JavaScript is a scripting language written in 1995[21], it has advanced and evolved into a powerful tool for creating websites, and now is being used for a range of purposes such as simulations[61]. As the technologies used in a web browser have evolved so have their capabilities, leading to the constant evolution of tools and features that are offered and supported[21].

As web technology changes so does what it means to be web-based. What has thus far remained constant is that the application runs in a web browser. A web browser, such as Chrome, Firefox, or Internet Explorer, acts as a virtual machine, which executes the commands it has been passed that have been written in HTML, JavaScript or other supported languages[118]. With HTML5 these commands are no longer limited to just acting within the VM, but can now be used to create local web storage and the like, such as offline applications[41]. Applications can load data from this local storage saving time as there is no need to download the data again, or reprocess the data to a given state. The use of the GPU is also something recently supported though the work of WebGL. Allowing for the usage of the GPU lets a developer create 3D graphical applications that can run smoothly and take advantage of the graphical acceleration offered by the GPU[14].

The use of these new web technologies allow for the creation of the base of a platform independent application that has the ability to run across a range of systems - any which have a web browser. With the rise in the popularity of web-based applications has come a rapid advancement in the driving technologies used in this VM, such as the likes of just-in-time compilation (JIT)[64]. It is the use of advancements like this that give rise to the use of JavaScript and the web as a platform for a range of new applications, such as web-based simulations[117]. One new language that further improves upon the existing platform is Dart. Dart is a new web language that runs within its own VM in the Dartium browser.

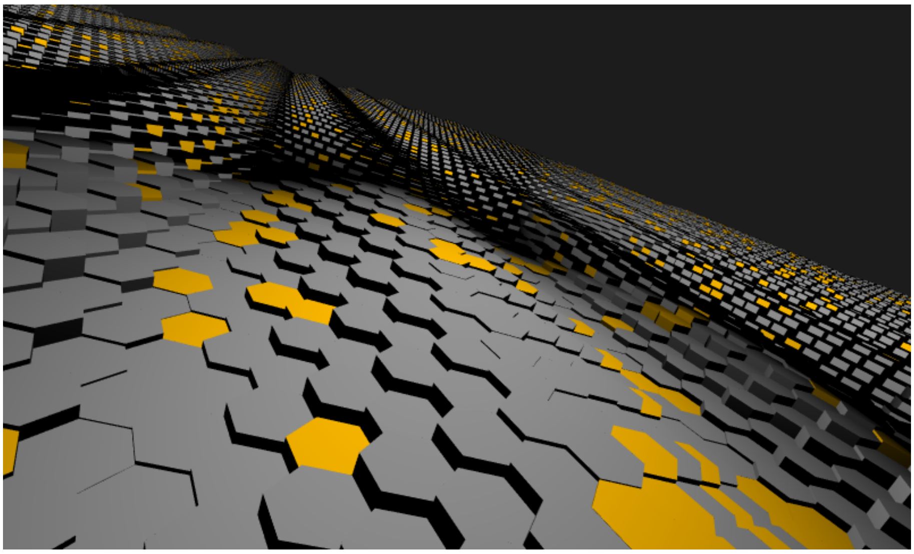


Figure 1.2: The image above is taken from a scene created using WebGL and JavaScript

Dart further extends the capabilities of the web, but is required to be run within its own VM, or be converted into JavaScript, which can then be run on other web browsers[122].

The constant evolution of web-based technologies provides a range of advantages that would otherwise be unavailable[116]. The most significant of these advantages is that it makes it almost completely unnecessary for the user to install additional software, as most modern operating systems come with a web browser preinstalled. As most common web browsers are connected to the internet this allows them to constantly update to a newer version, consequently updating the engine that the browesr uses, such as the V8 engine [34], in turn improving performance. With the updating of a given engine also comes the updating and implementing of functions, as well as support for additional languages. This improved performance enables a simulation or computationally complex problem to be computed using these web-based languages[117]. It is these advancements which make the use of web-based technologies well suited for the use and creation platform independence[116].

There are a few things to note when creating an application for the web, such as the fact that although some applications are able to be run offline, doing this frequently will result in a lack in some form of functionality[48]. Other applications will require a constant internet connection to be run so if that is not possible, then, of course, the application will not be able to run. As the use of VMs are vital in the usage of web-based applications, this means that an application will not run as efficiently as one written natively for that device[13].

Designing an application to be run with web technologies allows for it to be loaded and run across a range of devices. Each device is different and, because of this, the application needs to adapt accordingly; thus one of the requirements of this thesis becomes to create a graphical application that is able to dynamically change and adapt to suit the different performance requirements of devices. To aid in the attainment of this goal the use of computational simulations has become a core part of the overall system.

1.2.1 Dart

Dart is an example of a web language that suits platform independence due its having been designed for that very purpose. On the other hand, JavaScript is an example of a language that could be used to achieve platform independence due its progressive widespread use and support on most devices. The benefit of using Dart is that a developer can be sure that whichever device an application ends up on,it will be capable of performing at a high level. This is made possible by a number of design elements in Dart, such as the use of classes, and types, combined with features, such as SIMD, which produce highly efficient code that can be quickly interpreted by the compiler and executed in the VM.

1.2.2 JavaScript

This is in contrast to JavaScript, which is already in widespread use as a cross-platform solution, but has such an extensive range of ways to achieve the same result that a developer can very rarely be sure that their code is efficient. As a consequence, applications written initially in JavaScript perform slower than those written in Dart. This is particularly a concern in the context of graphical applications where, at this point in time, a developer needs to be sure that they are leveraging every possible source of processing power and providing as much speed as they can as graphical applications are computationally intensive, but a user may choose to run it on a low end device and expect it to run.

1.3 Computational Simulations

Computational simulations are used to produce various effects by following a set of rules to produce the desired results[75]. One of the most notable computational simulation is Conway's Game of Life by John Conway[35]. This simulation works by applying rules to a given lattice then updating and repeating the process. The result from Conway's Game of Life is that by following a simple set of rules a complex pattern can emerge[93]. With this knowledge it becomes possible to generate increasingly complex items using a given rule set.

As previously discussed, the constant improvement in web-based technology allows for the efficient use of some complex simulations[30]. Courtesy of this constant improvement it is now not uncommon to find cloth simulations and water simulations that can be run in a browser, although they are often limited in size as to allow for the simulation to complete in real time. Many of these simulations, however, can be run in a range of sizes to increase or decrease the amount of computational power required to efficiently run the simulation. The underlying ability to change a simulation at runtime leads to the creation and base of the system used within this thesis to test and configure the performance and adapt it to suitable levels. This ability to alter a simulation according to device requirements has provided the opportunity for the creation of the base of the system used within this thesis which makes it possible to test and configure performance and adapt it to suitable levels.

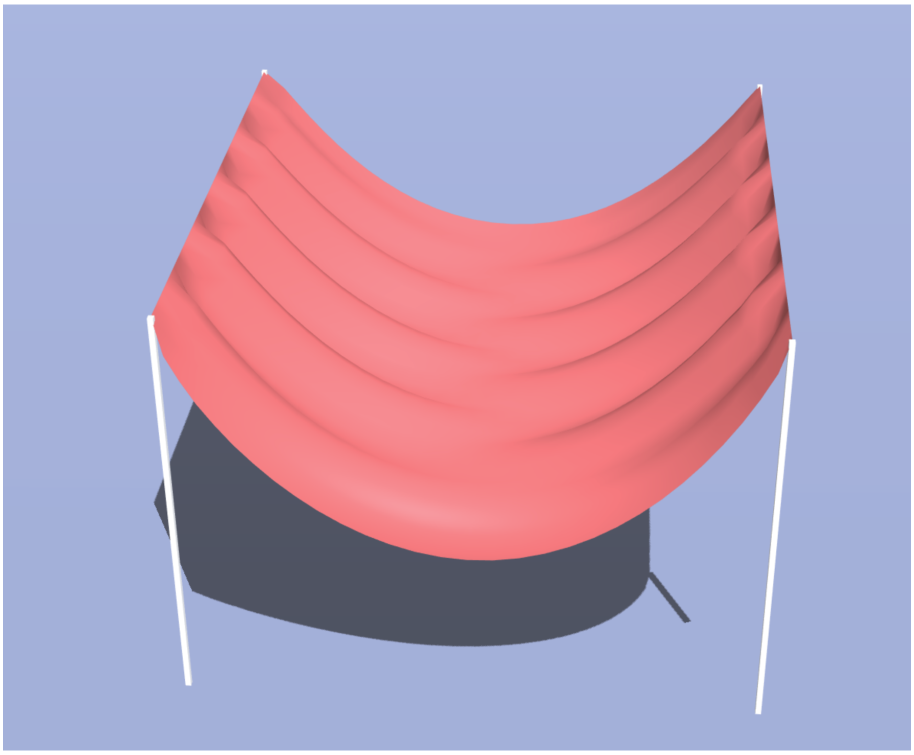


Figure 1.3: A basic cloth simulation running in WebGL.

A simulation can be defined by the rules it follows, ranging from how it interacts with a neighbouring cell or object through to how it interacts with a ray of light. It is by following these rules within a given system that the desired results are produced, whether that is the creation of a 3D scene, or a logic step in a weather system. As this thesis aims for optimisation of graphical platform independent applications, naturally the graphical aspect of simulations are a focal point of all undertakings. Furthermore, a requirement of these simulations is that they can be run in real time, and can be altered throughout the course of the application.

There is a significant, existing collection of simulations that are used to produce graphical effects, as well as corresponding techniques to improve the accuracy of these simulations[44][83]. Ray tracing is one such technique[60]. It is a process used within graphics as a way of simulating the way in which light will bounce around a 3D scene. The use of collision detection is another key feature that is used often within simulations, so as to determine how an object will act when hit with another or when two objects collide[47]. Each type of simulation can be used to produce a range of results based on the input and rule set provided.

There are numerous reasons why a developer might opt to use a simulation. One of their most obvious advantages is their flexibility. New results can be easily produced by actioning small changes[67]. When these changes affect the complexity of a simulation this also affects the speed of simulation, in turn dynamically changing the efficiency of a given system. Another desirable quality of simulations is that they can produce an asset or scene, such as landscape, without requiring a model to be loaded in from a given file thus improving speed as there is no need to load a file then read the data[70]. This provides another way in which simulations can improve the overall efficiency of a system, with more details in chapter 4. This is as opposed to using static models, which is more commonly used for detailed object where the quality, or another aspect of the model is important, and needs to remain constant.

Simulations are commonly implemented using random numbers. The use of random numbers is advantageous in aiding the production of new results; however, the reproducibility of random numbers can be difficult to control, which can manifest in undesired results[55]. This problem can be overcome with the use of a seed alongside a pseudorandom number generator. Introducing these techniques means more control can be brought to these kinds of simulations[55]. Overall the use of simulations is a popular method of generating desired information due their flexibility, customisability and efficiency[36]. The use of simulations allow the performance of tasks to be sped up and to be altered to in turn alter the data produced. It is this level of control over the efficiency of a system that makes simulations ideal for the creation of an algorithm that is able to change the performance requirements of a given system at runtime based on the device.

1.4 Procedural content generation

Procedural content generation (PCG) is the process of generating content with minimal input from a user [69]. The content created can vary largely based on implementation details, and can be used for creating objects with set properties, to creating entire landscapes, and worlds. This content can then be used within games, or other mediums to help automatically populate the world, giving designers a starting point, or a finished product. There are a number of methods employed when generating content, with each largely focused on a set content type. While these generative methods often share some properties, their usages are well defined.

Some of the main methods used for PCG, are search based, constructive generation, and fractal/noise based generation. Each of these methods are suited to different types of content. Search based algorithms are fairly diverse, in that they are capable of generating a variety of different content types. The down side of them though is that the content they generate is made up from existing objects, with various properties. This mean that the when generating content that the same property or element can be reused across different objects. The quality of the content generated though is based on an evaluating function which determines how appropriate a given element is for the content being created. [42]

Constructive generation is used largely when creating in closed environments, where boundaries can be defined. Constructive algorithms are mainly implemented in one of two ways, firstly adding rooms or areas within the confined space, then connecting these rooms together producing the final environment. The other popular method is taking the defined area, and removing sub areas via a given algorithm, producing the new layout or environment.

Fractal and noise based generative algorithms are often used when creating large landscapes and scenes. Noise based approaches such as Perlin Noise focus on generating random points which are then connected to produce a heightmap representative of the generated scenes landscape. Fractal approaches focus not on random numbers but on the recursive sub division of points based on a given algorithm to again produce a heightmap used to represent a scene. Both of these approaches, Noise and Fractals are discussed in more detail later in chapter 4.

1.4.1 Scalability

When considering the method of displaying assets for platform independence, scalability must also be taken into account to avoid creating an application that can perform on a high end device, but crashes on a lower end device. Currently, users do not have a choice of using the same software they have on a computer on a mobile phone by sacrificing display quality or user interactivity. Instead, that choice is made for them by either the application only being available on one, or similar device types, or the application made available on a range of devices but with inconsistent functions and features. In order to provide users with the highest possible display quality and user experience depending on their device, and the choice of sacrificing these in order to have the convenience of accessing the application on another device without losing any features developers have a few options depending on their preferred method of incorporating assets into a scene.

1.4.2 Models

Where a developer chooses to incorporate models to create a scene they would need to include a library of assets created at a range of set sizes and resolutions ready to be included within a scene depending on the device the application is being run on. The drawback of this approach is that the entire library of models may have to be downloaded regardless of the device type. Unfortunately, this does not support the aim of creating an application that is light on data usage.

1.4.3 Algorithms

An alternative approach to inserting assets into a scene is to generate them with algorithms and simulations. Generated content has the advantage of being able to be created at the appropriate size and resolution for the device the first time. This means that an application can be run on a computer with assets generated at the best possible quality as the device will allow, which could be considerable on a desktop PC that has been fitted with the most up to date technology; meanwhile the application will still have the same functionality if accessed on a mobile device and the same assets can be generated, without the user being stung with high data usage.

1.5 Level of detail

Level of detail (LOD) is becoming an increasingly popular method when rendering, to allow for an object to change how detailed it is based on various factors such as viewing distance. There are a number of issues faced when dealing with level of detail to minimise the visual effects of transitioning from one level of detail, to another. Most commonly is the effect of "popping" where an element of an object is hidden at one level of detail but suddenly appears as the level of detail changes[59]. Many of these level of detail approaches work based around starting with a high detailed model, then either manually or via an simplification algorithm reducing the quality of an object making it more efficient to render when finer details are unable to be viewed.

These simplification algorithms largely revolve around the removal of faces, via ether collapsing points, to join them together, thus reducing the number of faces with an object, or merging polygons which are close to coplanar[59]. There are a range of other methods for hiding, or changing the topology of a mesh, but they are largely outside of the scope of this thesis.

1.6 Performance

Continual advances in technology mean that cores are being produced to be smaller, and more powerful, which means significantly more processing power can be packed into hand-held devices, as well as larger devices such as desktop computers. This is advantageous as the more cores a device has, the greater performance it achieves from a concurrent implementation. Until somewhat recently, concurrent implementations were not a particularly suitable method for leveraging additional performance as devices that could benefit from this were limited. However, as technology progresses the range of devices that can, and the extent to which they can benefit from concurrency will make it an especially appealing solution for graphical platform independent applications.

1.6.1 Concurrency

In order to provide users with highly realistic graphical applications that are accessible on a range of devices developer must find the processing power to support them without knowing exactly what that processing power will be. Concurrency provides a method for the developer to leverage as much processing power as possible. This is particularly important in the context of graphical applications that require significant processing power, the demands of which only increase with the complexity of the simulation, and the level of detail desired.

1.6.2 GPUs

Another relatively recent enhancement to mobile device hardware includes the addition of GPUs, which have been a part of every iPhone released since 2007. For the purposes of a graphical application the processing power of a GPU is significantly superior to a CPU so it is an obvious option for providing additional computing resources[52]. At this stage, there is more than one way a developer might seek to solve this problem. A combination of

OpenGL and OpenGL ES can, allow an application to access the GPU; however, it would require the developer to add substantial support to allow this design to work for platform independence. Using multiple graphics libraries is another way a developer could access the GPU. While the setup of this does not have to be difficult, it does require vigilant, and ongoing maintenance to ensure that updates to the libraries do not adversely affect the range of devices on which the application is supported. Specifically tailored to use in web applications, WebGL offers a convenient solution due to its ability to plug into a wide range of APIs. It also has a wide range of support so a developer can be confident that their application will be able to access the GPU regardless of device type.

1.7 Publications

This research has lead to a number of publication in the area of web development, platform independent graphics, procedural content generation, and user interaction.

- Graphics on web platforms for complex systems modelling and simulation[70]

- Webgl for platform independent graphics [67]

- Procedural generation of terrain within highly customizable javascript graphics utilities for webgl[69]

- Meaningful Touch and Gestural Interactions with Simulations Interfacing via the Dart[68]

1.8 Previous work

Although significant research has been done in the fields of platform independence and of graphics, relatively little scientific exploration has been done regarding the application of graphics within platform independence. The focus within the field of graphical programming remains solely on the optimisation of graphics on a selected platform, without consideration towards how the application will run across multiple devices. On the other hand the field of platform independence focuses primarily on the goal of creating and optimising applications that are able to run across a range of systems. There are a few areas within these two fields that overlap, such as Flash, and Java 2D, but there is a noticeable lack in regards to 3D. This leads to somewhat of a void where the development of platform independent graphics is concerned.

When an application is designed to be platform independent it is run within a VM, or compiled into other languages. These methods allow for a range of systems to be supported, but leave out the graphical applications [53]. To date, a limited amount of development has been done in bringing OpenGL and other graphics libraries to the usage of VMs; languages that compile to run across a range of devices may have some graphical support. Because of the way in which both methods achieve platform independence it is evident that, due to the interpretation steps, some GPU features are not supported as well as they could be. Another method used to produce graphics for a range of platforms in the web was though the usage of Java3D on the client side, though this in turn had various flaws [4][114], and has since become deprecated.[90]

The benefit of many older methods was that applications were able to successfully leverage some of the GPU's processing power [53] while running across a range of systems within a VM. The advantages of these approaches is that no code needs to be rewritten, and basic support for graphical applications is offered. The downside of which is that because it is running within a VM, it becomes difficult to fully utilise the performance of a GPU[26]. Along with this difficulty comes the inefficiency produced by using a VM, or another language to convert code, as support for some features on a given platform will not exist[26].

More modern approaches to having a graphical application utilise game engines, in which an application is written once using the engine, such as Unreal, or Unity, then is compiled and deployed for a specific device. While the use of game engines is suitable for primarily games, it does not offer as fine tuned control over an application as one created natavily for the application. In turn a more native approach utilising a graphics library such as OpenGL, or DirectX limits the user to a set platform, but allows for greater control over the applications design, and performance optimisations. Alternatively libraries such as Three.js or BabylonJS allow for a graphical application to be written for the web, leveraging the power of WebGL, thought in turn still suffering from a similar effect as a game engine, where the library becomes deeply embedded into the created application.

With the advances made within web technologies, such as that of WebGL, it becomes possible to produce real time renderings across a range of platforms, through the use of various web browsers[70][117].

1.9 Aim of thesis

The aim of this research is to prove that using the latest technology, such as Dart and WebGL along with the use of simulations, it is possible to create graphical applications that will automatically configure themselves during runtime to produce optimal settings based on the device it is currently being run on, along with being able to be run across a range of devices.

This is an important contribution to computer science as many applications that are written to be platform independent lack performance especially if they are graphical. Furthermore few will offer a range of predetermined settings based on the device, let alone having an application configure itself to the device's specifications.

The research explores the use of simulations such as the diamond-square algorithm, used in terrain generation, along with the shallow water simulation and Perlin noise.

Chapter 2 explores the process of rendering on the web through advancements in technology, such as the rendering pipeline in OpenGL and WebGL, and the development of the web language, Dart. Chapter 3 describes the creation of a controller class that utilises various APIs offered in Dart to map inputs of a range of devices within a graphical application. The creation of an algorithm that is able to determine how best to render and create a scene throughout the runtime of the application is explored in Chapter 4. Chapter 5 covers the creation of the asset controller, which utilises the algorithms discussed in Chapter 4 to establish an ideal runtime for a given device. Chapter 6 assesses the performance of these algorithms across a range of devices and implementation types, and analyses the scenes generated according to developer metrics and device specifications. Chapter 7 evaluates how this research provides solutions to issues currently present in the software industry, and answers to frustrations experienced by consumers. Chapter 8 outlines the efficacy of this research, and suggests areas where future research can be undertaken.

,

Chapter 2

Rendering on the Web

The framework discussed in this project utilizes the 3D rendering pipeline, through WebGL and Dart, as the basis for creating platform independent graphical applications. WebGL is the subset of OpenGL that allows for an application written in a web language[50], such as Dart or JavaScript, to access the GPU to allow for accelerated graphics using a rendering pipeline. Dart is a new language designed to increase the performance of web applications, and improve the development process. To achieve this, Dart offers increased functionality compared to other web languages, while simultaneously offering improved performance through its virtual machine, and supported processor features such as SIMD[66].

2.1 3D Rendering Pipeline

The rendering pipeline is a process that takes data representing a model in 2D or 3D space and creates a 2D graphical representation to render on screen.

Currently, the best way to produce a rendering pipeline is through libraries and APIs like DirectX, or OpenGL. The rendering pipeline in this framework uses WebGL, which is a subset of OpenGL.

OpenGL has a range of versions, all of which are aimed at different hardware, thus producing a range in functionalities between each implementation. As of OpenGL 2.0

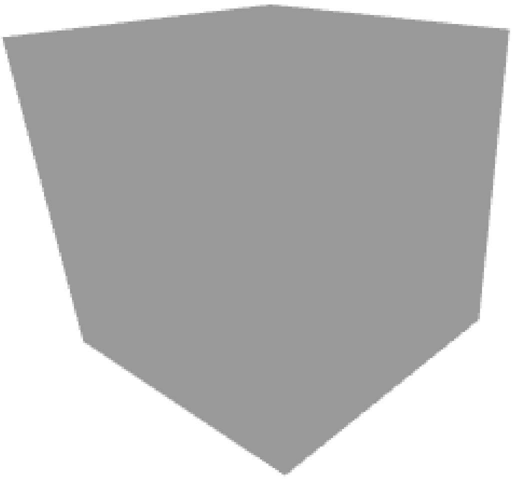


Figure 2.1: A basic grey cube being rendered using WebGL, a subset of OpenGL

development has focused on the use of a programmable pipeline (with OpenGL Shading Language (GLSL) shaders) over the fixed function pipeline[45]. This allows developers to have greater control over the rendering pipeline, and create more flexible products[91]. It is these rendering pipelines that process given data and produce the final rendering of an object.

The rendering pipeline begins by receiving the data that makes up a scene; this is usually in the form of vertices, indices, normals, and textures. The first step in the pipeline is to process this information to assemble the scene via per vertex operations. These vertex operations convert a point into the correct 3D space, allowing for correct lighting to be applied. This process is called primitive assembly. After primitive assembly is completed the primitives are rasterized; this is the process by which a 3D scene is converted into 2D. Subsequent to rasterization, depth layers are applied to ensure that objects closest to the screen are displayed, while objects further back are hidden. Finally, once a 2D frame has been created, textures and lighting are applied to each fragment within the frame[89]. Each process within the rendering pipeline is covered in more detail later in this chapter.

2.1.1 OpenGL

OpenGL comes in three major variations, OpenGL, OpenGL ES, and WebGL. Each of these are available in assorted versions which include support for different functionalities. OpenGL initially was designed to work with a fixed function rendering pipeline[108], but this has subsequently been moved to a more flexible programmable pipeline in OpenGL 2.0. OpenGL ES is a subset of OpenGL designed for use on embedded systems, whereas WebGL is designed for use within web based technologies. Due to the frequent improvements and advancements in OpenGL, newer versions are continually released to support the latest functionalities. However, these functionalities are often limited in scope in order support achieving design goals on the relevant platforms[88].

The fixed function pipeline initially offered by OpenGL allowed developers to harness the processing power of graphics hardware. This allowed for the acceleration of the various aspects which the user was required to provide, such as specific matrices, vertices and other configuration parameters, like how to handle the textures. The information provided by the user would then be passed through the fixed function pipeline to produce the appropriate rendering. The fixed function pipeline is the simpler approach to the rendering pipeline as it limits the user's ability to define custom shaders, instead applying user based configurations, thus providing limited functionality.

OpenGL 2.0 introduced shaders, which allowed for users to have greater control over how rendering occurs, thus creating the programmable pipeline. The programmable pipeline replaced the user's ability to configure different aspects of the rendering process with ability to create the rendering process through the use of programmable shaders. via the use of shaders. Initially, these shaders were limited to the vertex and fragment shaders which determine how an object is coloured and rendered. Recent versions of OpenGL include a more extensive range of shaders, such as the tessellation shaders, which allows for data produced by the vertex shader to be further subdivided to create a higher level of detail when rendering.

The first significant subset of OpenGL, OpenGL ES, was designed to work on embedded systems, such as smartphones. Initially, OpenGL ES 1.0 was based upon a fixed function pipeline, much like OpenGL 1.3, but with the inclusion of slight limitations on how it could be configured[102]. These limitations were put in place to allow for the appropriate use of a device's resources, particularly on low performance devices. One such example of these limitations is the removal of `GL_draw` , which allows the use of draw quads. OpenGL ES 2.0, on the other hand, is closer in functionality to OpenGL 2.0 as it incorporates an introduction to shaders. However, the fixed function pipeline was completely removed in OpenGL ES 2.0, unlike in OpenGL 2.0 where a developer was able to work using both the fixed function, and programmable pipelines.

WebGL is a more recent subset of OpenGL, based on OpenGL ES 2.0. The specifications of WebGL remain very close to those of OpenGL ES 2.0. Both subsets are able to achieve a closer interface with languages like JavaScript by utilising the programmable pipeline with a few limitations. These limitations primarily extend as far as data types, but also include the primitive types that can be rendered in OpenGL ES, such as lines and triangles. Through the use of WebGL developers are able to bring hardware accelerated graphics to the web without the necessity of installing third party plugins[62] though still require supported device drivers are installed.

OpenGL and its numerous subsets provide many options for carrying out work in 3D rendering. For the purpose of this framework WebGL offers the best solution as, like OpenGL ES 2.0, it has been designed to be as lightweight and streamlined as possible, but, unlike OpenGL ES 2.0, it is specific to web requirements. Thus WebGL is ideal for use in platform independent applications as much of the functionality of the rendering pipeline is available for the developer to utilise.

2.1.2 Rendering

The rendering pipeline is designed to receive a range of specific types of data, process the information, then render the scene accordingly. The various functions within the rendering pipeline can be customised via the use of shaders. Shaders process the data required to render an object, including where the vertices are and how they should be joined. When using the rendering pipeline within OpenGL it is necessary for the user to provide the appropriate instructions on how information should be passed for each draw call[110].

The first step in the rendering process is collating the necessary data for creating an object. An entire object is made up of basic shapes (primitives) which are comprised of simpler shapes called primitives. Primitives are made up of a series of vertices, which identify points in 3D space, represented as X, Y, and Z coordinates. The linking from one vertex to another is performed using indices, which contain information about how a set of vertices are joined together. Depending on how a user has instructed an object should be rendered, primitives can take on a number of shapes. Usually these shapes would take the form of quads, triangles, triangle strips, a single line or a point, along with a few other primitives[111]. It is via the development of these primitives that an object can be constructed, as seen in figures 2.2 and 2.3. This data is stored as a buffered array, enabling it to be passed via OpenGL to a shader program.

Once the basic form of an object is rasterised, textures and lighting can be applied to improve the quality of realism. Textures are applied to an object via the technique of mapping. This involves the rasterizer computing texture coordinates for each fragment, based on a primitives uv coordinates. This texture coordinates are then used to look up a texel value from the current texture map. In addition to textures, normals can be applied to an object to simulate the effects of lighting within a scene, and how light sources interact with objects. Normals are typically used in lighting calculations within the vertex shader. Where the user determines it is relevant, they can use normals to apply reflections

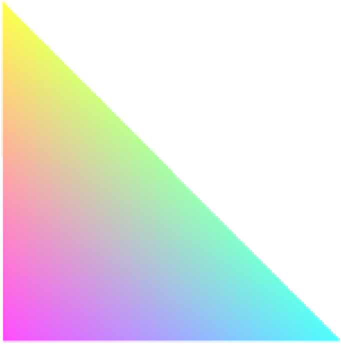


Figure 2.2: A triangle, created by joining 3 vertices together based on their corresponding indices.

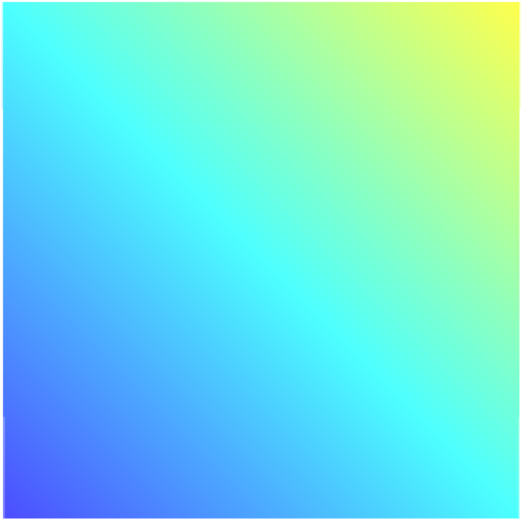


Figure 2.3: A simple square, created by joining two triangles together.

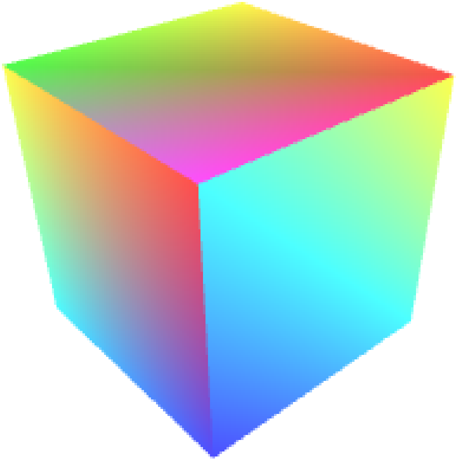


Figure 2.4: A cube produced by adding multiple triangles together.

and / or reflective qualities to the surfaces of an object. Normals can also be applied to change the brightness within a scene as light sources change and fluctuate, or as objects are manoeuvred.

Textures are linked to an object via the use of UV coordinates.They are applied to a model by linking each each set of UV coordinates with a corresponding fragment. This allows for a section of the image to be applied across a geometric primitive to apply the desired detail to the relevant area. Whereas textures are applied to fragments, normals are applied to vertex. When light hits an object normals are used to determine how light diffuses across its surfaces, and how much light is reflected back into the scene, thus dictating the extent to which an object is visible and how much ambient light there is within the environment. The angle at which light hits a surface will increase or decrease the spread of light as it moves across the scene.

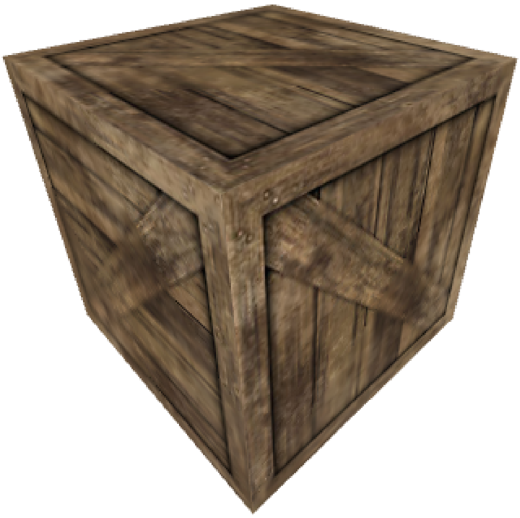


Figure 2.5: Applying a basic crate texture to a cube

Listing 2.1: Dart code for generating a surface normal for a triangle, made up of three points in 3D space.

```
Vector3 createNormals(){
   //The three points which make up the triangle
   Vector3 pointOne;
   Vector3 pointTwo;
   Vector3 pointThree;

   Vector3 U = pointTwo - pointOne;
   Vector3 V = pointThree - pointOne;

   Vector3 N = new Vector3.zero();

   N.x = ((U.y * V.z) - (U.z * V.y));
   N.y = ((U.z * V.x) - (U.x * V.z));
   N.z = ((U.x * V.y) - (U.y * V.x));
```

```
    return N;
}
```

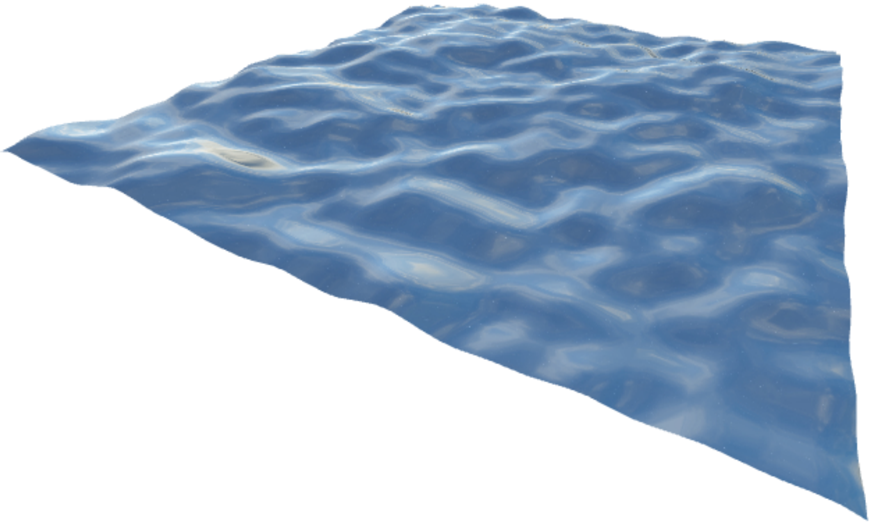


Figure 2.6: Light has been applied to this scene, based on generated normals using the code in Listing 2.1 where the normals are evaluated at a vertex, and interpolated to give a smooth surface normal

Object creation in OpenGL is adaptive and highly customisable due to the multiple fragment types and rendering methods that are available to the user. Incorporating textures and normals into an object's construction results in high quality renderings, in addition to improving the level of realism within a scene. Where the user is employing the method of a programmable pipeline, the utilisation of textures and normals is managed through the use of shaders.

2.1.3 Shaders

Shaders are programs that run alongside the rendering pipeline on the graphical processing unit (GPU)[96]. They replace various aspects of the fixed function pipeline providing more flexibility, and creating a programmable pipeline. There are two types of shaders utilised within WebGL; the vertex and fragment shaders. The vertex shader focuses on the positioning of the vertices of an object when rendering, whereas the fragment shader manages the colouring and textures of an object[94].

Shaders designed for OpenGL are written in OpenGL Shader Language (GLSL). Each version of OpenGL 2.0 onwards includes support for additional shader features, or incorporates new shaders altogether. OpenGL shaders are designed to complete one task which they are able to execute with great efficiency by making use of highly specific functionality. The most simple of shaders take in data in the form of attributes, such as `int, float, mat,` or `vec`. Applying basic operations to these attributes begins the customisable pipeline, including some predefined functionality. The results of these shaders are then placed in variables, such as `gl_Position` or `gl_Fragcolour`, to allow the data to continue through the pipeline, as seen in figure 2.2.

Listing 2.2: Basic vertex shader code example for vertex positioning using attributes and defined output, and for passing texture information to the fragment shader using the varying data type.

```
attribute vec3 vertexPosition;
attribute vec4 colour;
attribute vec2 uv;

uniform mat4 modelViewMatrix;
uniform mat4 projectionMatrix;

varying vec2 vUV;

void main(void) {
   vUV = uv;
   gl_Position = projectionMatrix * modelViewMatrix * vec4(vertexPosition, 1.0);
}
```

Listing 2.3: Basic Fragment shader code example for fragment texturing using varying data, passed from the vertex shader to produce the defined texture output.

```
uniform sampler2D uSampler; // Containing a bound texture


varying vec2 vUV;


void main(void) {
   gl_FragColor = texture2D(uSampler, vUV);
}
```

The fragment shader computes the lighting model and any other effects to calculate the colour of the object at the point it represents.

Of the two types of shaders used in WebGL, the vertex shader manages the process by which a point in 3D space is defined, and how that point is represented within the rendered scene. To define an object's location, a shader needs to interpret data passed from the user, and output that data to `gl_Position` as a `vec4`. The simplest process to create the final location is to take in the x, y, and z coordinates as a single `vec3` attribute, multiplying it by a model-view and a projection matrix. This calculation determines how to render the object within a scene, based on the location from where the user is viewing the scene, along with where the object is located relative to the area of scene. This location is then passed through the rendering pipeline to be used when rendering the object. Data can also be passed from the vertex shader to the fragment shader through the use of variables labelled as `varying`. One of the main uses of the varying label is in the passing of data for lighting, such as surface normals, or data relating to the direction and source of the lighting.

The fragment shader works via a similar method to the vertex shader, but, is also able to receive variables passed from the vertex shader varying variables, as well as being able to receive data through uniform variables. The fragment shader computes the lighting model and any other effects to calculate the colour of the object at the point it represents. By combining the data gathered from the vertex shader with what is passed from the main application, the fragment shader is able to interpret this into a colour to output to `gl_Fragcolour`. In the most straightforward scenario, in which there is only a single triangle to be coloured, and that primitive can be entirely the same shade, applying colour can be as simple as defining a single `vec4` (using RGB and alpha values) within the fragment shader and applying those same values to all fragments of which the object is comprised. As the level of detail a user wishes to create within a scene increases so do the various methods which the user may apply to create the desired effects. Such techniques include the use of surface normals, textures, and directional lighting data. Utilising these methods enables the user to create a dynamic colouring system that incorporates light, while shadows are typically achieved through a multi-pass render. This, in turn, makes an environment appear more realistic as the different techniques are applied and refined[97].

The creation of highly dynamic renderings within OpenGL, and, by extension, WebGL, are realised by the utilisation of the both the vertex and fragment shaders within the programmable pipeline. The ability for the user to have autonomy over aspects of the rendering pipeline is vital for applications such as WebGL, as this allows for developers to take advantage of a device's GPU to improve performance, while simultaneously increasing customisation options that are available while rendering.

2.1.4 Executing the rendering pipeline

Once the primitive is created in OpenGL, the shader must be compiled and initialised before the object can be rendered. The rendering process involves the linking of a shader program on the GPU to the CPU through the use of attributes or uniforms. To complete the rendering of an object a draw call must be set up and actioned for each primitive that is to be rendered. Additionally, any updated information must be passed to the shader.

The process of linking a shader to the GPU and CPU has many steps as shaders cannot be directly run on the GPU. They must first be compiled by OpenGL into a shader program that can then be passed to the GPU and deployed. Defined within a shader are attributes, uniforms, and varying data types. To allow for data to be passed to the shader a pointer is created. Various object types are utilised to allow for data to be passed to a shader, vertex buffer objects are used to store vertex attribute data, uniforms for values used by multiple objects, and textures for static data blocks.

Creating a shader program to be run on the GPU begins in OpenGL, where the type of shader must first be specified, such as, vertex, before the shader can be compiled. Once the shader type is defined the relevant source file is linked to the shader; this contains the code for the shader program that will be run on the GPU. After the shader has been linked to the source code it is compiled into a shader program. When OpenGL is required to render an object, the shader program is used to execute the draw call. In the case where there are multiple objects to render, but the same shader is being applied, all draw calls can be executed using the same, single shader program. Where the user wishes to apply more than one type of shader, it is recommended to group all objects pertaining to the same shader and render them all before switching shader programs and rendering all objects pertaining to that subsequent shader. The grouping of objects minimises the expense of changing between shaders, which is considered a costly procedure, and should be avoided where possible[28].

The advantage of using the GPU for rendering is that GPUs are designed with significantly more cores than standard CPUs. This means that shader programs, which are executed on the GPU, can run very efficiently by taking advantage of the numerous cores to render multiple fragments at the same time. The increased number of available cores is possible because of the limited interconnection between each processor on the GPU[72]. These additional cores, although not heavily interconnected, allow for the task of rendering each fragment to be split across multiple cores and threads, thus resulting in increased performance when rendering a scene compared to rendering a scene using a CPU[92].

There are three methods of passing data to a shader, the most common of which is attributes. Attributes are stored within a buffer, and linked to a shader program via a pointer; for each vertex new attributes are passed to the shader program using this linked buffer. On the other hand, uniforms are passed as a single draw, and are often used for world or position matrices. As uniforms remain constant for the rendering of each draw call, they do not change between each primitive. The advantage of this is the minimisation of amount of data to be transferred for each render call. The final method for passing data to a shader is through varying data types. Defining a variable as varying within both the fragment and vertex shaders allows for data to be passed from the vertex shader to the fragment shader. A common application of this process is generating normals using the vertex shader, then transferring this data, via the varying data type, to the fragment shader in order to apply lighting effects. This results in a fragment shader getting input from a varying as interpolated (assuming smooth shading case) values from the vertex shader.

Once the shaders are compiled, OpenGL can access the programmable pipeline in order to render a scene. The user must specify the location of the data that makes up an object, such as the vertices, and indices, along with uniforms, which contain model view positioning information. This data is then passed to the shader program via a draw call.

The draw call is a series of events that is separated into three main tasks. The first of these tasks is the setting up and defining of which shader programs will be used, as discussed in the previous section.

The second event is the defining of the uniforms. These normally contain matrices, which define how the scene is viewed; this data remains the same for all primitive that are rendered within the same draw call. Uniform data is passed by the simple method of using the OpenGL command, `uniformMatrix`. The specifics of this command will change slightly depending on what kind of data it has been passed. The pointer then links the uniform command to the relevant uniform, as defined in the shader program.

The final task of the draw call is to set up attributes, which contain all essential information relevant to the rendering of primitive. The setting up of attributes occurs once the uniforms have been defined and passed to the GPU. Attributes contain variables that change with each fragment they are passed. As a result, data contained within attributes is often stored as an array, which is then linked to an attribute pointer. This data is commonly used for storing information such as, vertex locations, normals, and texels; this information differs for each vertex.

The process of passing data from an attribute begins by first using the correct pointer to enable the attribute array. Once the attribute has been set up, the data is then bound to a particular buffer type, such as `ARRAY_BUFFER` or `ELEMENT_ARRAY_BUFFER`, to determine how the buffer is to be used.

The final step in setting up an attribute is to specify within OpenGL how each buffer is defined, such as, the number of elements for each fragment, the type of data, if the fragment is normalised, and the stride. This information might be specified in the case of a buffer containing mixed data, such as both textures and normals being stored within a single array. Once the attributes have been finalised, OpenGL can begin making draw calls.

The flow of the rendering pipeline can be seen in figure 2.7. It shows how data is moved through the rendering pipeling in order to produce the desired frame buffer. This process uses attributes as a method of passing data into the vertex shader, which is then used to

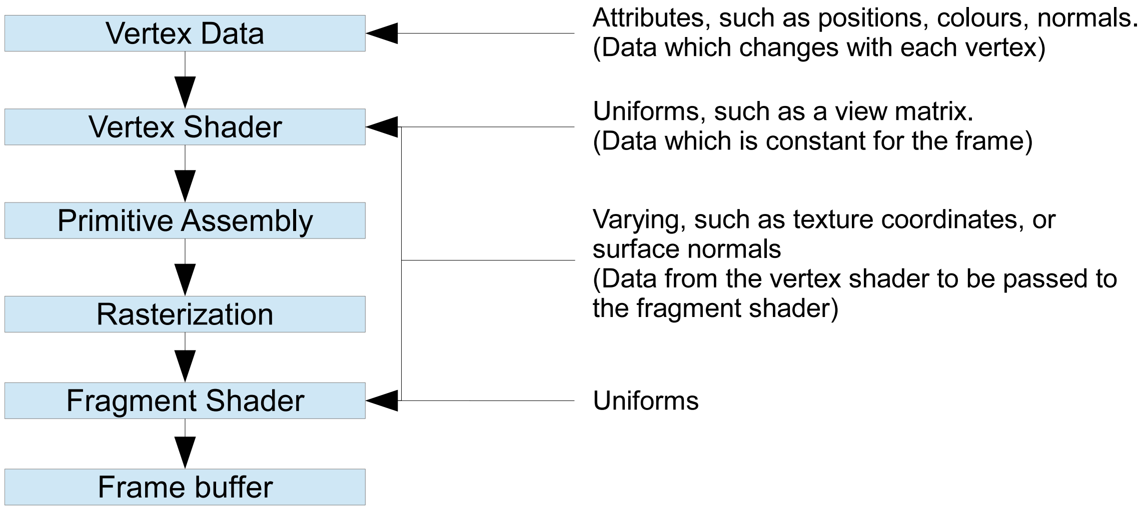

Vertex Data
Vertex Shader
Primitive Assembly
Rasterization
Fragment Shader
Frame buffer
Attributes, such as positions, colours, normals.
(Data which changes with each vertex)
Uniforms, such as a view matrix.
(Data which is constant for the frame)
Varying, such as texture coordinates, or surface normals
(Data from the vertex shader to be passed to the fragment shader)
Uniforms

Figure 2.7: The flow of data as it enters the rendering pipeline to produce the required frame.

produce a given vertex. This attribute data is unique for each vertex being worked on, unlike uniform data, which is designed to remain constant throughout a rendered primitive. Uniform data can be passed into both the vertex shader and fragment shader to be used when processing. As the data moves from the vertex shader, each primitive within the scene is then produced using the results of the vertex shader. The primitives are then projected from a 3D scene, to a 2D plane through the rasterization process. In order to efficiently communicate between the vertex shader and the fragment shader varying data types can be used. Varying data is passed though this pipeline, allowing for the vertex shader to influence the fragment shader. Once each fragment has been processed the final frame buffer is produced and is ready to be displayed.

There are two different function calls that a user may choose to employ to render a scene. Technically, the simplest function that could be used is `gl.DrawArrays`. When using `gl.DrawArrays` the user defines what kind of fragments, such as as quads, will be used to create the object. Following this, the user will define an offset in order to locate the first vertex that is to be rendered, then the user will specify the number of vertices to be rendered.

To achieve more efficient performance `gl.DrawElements` could be used. This function incorporates indices to allow for a single vertex to be referenced multiple times; this avoids defining the same vertex multiple times within a single buffer. Overall, this process contributes to minimising the amount of data that needs to be passed for each draw call. To use `gl.DrawElements` an indices buffer must be bound with the type `ELEMENT_ARRAY_BUFFER`, and the fragment, data type, indices array, and offset must be defined within the draw call. For a lightweight rendering, `gl.DrawArrays` would offer the best solution due to their simple design and modest amount of data[115]. Where the scene to be rendered is more complex, `gl.DrawElements` offer performance efficiency that is not available in `gl.DrawArrays`.

OpenGL offers a well defined rendering pipeline, that can be customised according to a developer's needs. OpenGL allows for the task of creating models by joining multiple primitives together to become highly optimised, and utilises the power of a device's GPU to achieve optimal performance. In addition to this efficient workflow, OpenGL allows for the user to incorporate customisable shader programs. Although these shader programs initially require a specialised set up, they offer the user an unparalleled avenue for customising the rendering process. OpenGL offers new processes with each implementation, as existing methods are developed or new ones added to improve OpenGL's capabilities. Each implementation is designed for a specific purpose, such as OpenGL ES is particular to mobile platforms. For the context of this framework, WebGL is the most appropriate implementation of OpenGL.

2.2 WebGL

WebGL is a graphics API based on OpenGL ES 2.0, which is, in turn, based on OpenGL 2.0, WebGL 2.0 is based on OpenGL ES 3.0. WebGL predominantly includes much of the same functionality that is offered by OpenGL, with only a few limitations[63]. These limitations are minor, and are incorporated to improve efficiency on low end devices[123]. Shaders are one feature that are common to all recent implementations of OpenGL; however they are a compulsory and integral part of WebGL which enable a web programming language, such as JavaScript or Dart, to access the programmable rendering pipeline. Within this framework WebGL is used for the rendering of simulations, and dynamically generated content. WebGL is designed to run within a web browser, such as Google Chrome, as this allows WebGL to potentially access a large amount of support by leveraging the virtual machine (VM) that is used by the browser. Utilising these web technologies also reduces the limitations on which devices an application could be run. This abstraction from the hardware makes it possible to create platform independent applications that are capable of a significant range of performance outputs.

2.2.1 WebGL for Platform Independence

WebGL lends itself to the creation of platform independent frameworks as it runs within a virtualised environment, with access the GPU for further acceleration. It is the performance increase offered by the GPU that has lead to the creation of many impressive graphical simulations that can be run across a range of devices[37].

One of the ways WebGL increases performance is by removing the necessity of installing third party plugins in order to access the additional rendering capability within a browser. This enables shaders to be deployed for processing on a device's GPU, which frees up resources on the CPU. Deploying shaders on the GPU further enhances the rendering pipeline's performance as shaders run outside of the VM's sandbox, whereas the rendering context of WebGL does not[46].

Another key performance improving method available in WebGL is the possibility of integration with web technologies, which offers the potential for an abstraction from hardware through a web-browser. It is abstraction from hardware that allows for a single code base to be run across multiple devices, thus producing various levels of performance[27].

Overall, WebGL offers an enhanced set of tools to achieve high performance rendering through a web-browser in a platform independent contexts. The performance advantages achieved by offloading work to the GPU further contributes to improved performance, and reduces the hampering effects of running within a VM.

2.2.2 WebGL Limitations

As mentioned above, one of the most significant advantages of WebGL its readiness to adapt between different platforms. A key reason WebGL is able to do this is due the various intentional limitations in place as a result of its design. These limitations are integrated to produce a more fluid user experience across devices, irrespective of a device's performance capability. The limitations range from which data types are included to what primitives are supported; there are also a reduced number of shaders available. All of these are limited in order to keep the WebGL 1.0 specifications as close as possible to those of OpenGL ES 2.0[63].

One of the most significant limitations of WebGL 1.0 is the constraints placed on data types; this is especially evident when using indices. Currently, support for indices within WebGL extends as far as unsigned shorts, which means the maximum number of vertices that are supported in a single draw call is 65,535[67]. This is due, in part, to the design of OpenGL ES 2.0, which shares in this limitation, but is also a result of continuing to maintain support for low end devices. In addition to restricted data types, there are only a few supported primitives. These include points, lines, triangles, and some similar variations of these. This means that in order to draw a square, two triangles must be rendered. While this increases the amount of data to be passed, it also reduces the amount of data required for each primitive, as well as reducing the complexity of the available primitives.

In keeping with the OpenGL ES 2.0 standard, WebGL supports the vertex and fragment shaders within a programmable pipeline; however, unlike OpenGL ES 2.0, offers no such support for the fixed function pipeline. This functionality was previously supported in OpenGL, in order to provide legacy support, but has been removed from WebGL and OpenGL ES 2.0 to provide a more streamlined pipeline. This means that a large number of processes, which were previously available in the fixed function pipeline, are no longer accessible. One such example of a functionality that has been excluded from the WebGL pipeline is the inbuilt rotation and translation of objects and cameras within a scene. This is an essential function of the pipeline as it allows the application to produce the correct matrices and shaders that are necessary to the rendering of a scene. However, this process, like most of the processes that are not supported in the fixed function pipeline, can readily be recreated and implemented by the developer.

Although WebGL does have some limitations most of these can be overcome through the use of various techniques and design decisions; such as only utilising graphics that are made up of triangles; and limiting the primitive count for each object. Additionally, objects can be split into multiple parts, and each part rendered individually in order to reduce the primitive count per draw call. When designing a WebGL application, if these limitations are taken into account, a highly optimized rendering pipeline can be created.

WebGL is supported in most major web browsers, on desktop, and mobile; this aids in its relevance and suitability for a platform independent framework. As mentioned previously, a significant advantage of using WebGL is that it removes the need to install a third party plugin for the browser as many older 3D web based applications did. Furthermore, specifications for WebGL 2.0 have been released, which look to improve various aspects of the current WebGL standard. Table 2.1 shows some of the main differences between each OpenGL implementations. It is worth noting that some of these limitations vary based on the hardware manufacturer as some devices may extend or limit functionality.

	OpenGL	OpenGL ES	WebGL
Target Platform	Desktop	Mobile devices	Web browsers
Current version	4.5	3.2	2.0
Indices limit	32-bit	32-bit	32-bit
Texture dimensions	any	power of two	power of two
Support 3D texture	Yes	Yes	Yes
Data precision	Double	Float	Float
Shader version	450	300 es	300 es

Table 2.1: The differences between each OpenGL implementation

WebGL is no exception to the constant evolution of technology, and the forthcoming release of WebGL 2.0 will certainly prove this. WebGL 2.0 will bring WebGL closer in line with OpenGL ES 3.0[50]. This will see the release of several features that have become commonplace in the world of 3D rendering techniques, and the updating and extension of existing features.

One of the new features that is included in WebGL 2.0 is 3D textures. Instead, textures will be able to be assigned X, Y, and Z values, which will then be able to be looked up from within a larger 3D image data set. This will greatly extend rendering and modelling possibilities when used in conjunction with 3D scans, such as those used in medical imaging[17].

Instancing will also be implemented in WebGL 2.0, that will allow for a greater increase in performance when rendering the same object multiple times. This is possible as only the position of said object will alter, not the physical properties; there is currently support for this within Chrome, and Firefox[80].

In addition to all of these improvements there will also be updated Vertex Buffer Objects (VBO) and Vertex Array Objects (VAO).These will allow for the bundling of buffer and array objects, thus streamlining and simplifying the processes associated with handling large quantities of data[50].

Regarding the improvement of additional features, one of the largest and most significant is the inclusion of support for 64 bit data types, that is, 64 bit integer, and unsigned 64 bit integer.

The significant number of improvements and extensions will not only contribute to superior performance, but also make WebGL more user-friendly and intuitive for developers by producing a more consistent experience between platforms. Consumers will also benefit from the possibilities generated from WebGL as the practical possibilities of applications created using WebGL can range from the entertainment sector to increasing the accessibility of medical information and technologies.

Where platform independence is the primary requirement or objective of graphical applications, WebGL is an ideal candidate. This is due to the numerous advantages as described above, as well as the ongoing support and updating of the WebGL standard. Moreover, WebGL is one of the first piece of technology that has been used in earnest to push the boundaries of web-based rendering through the integration of the GPU and use of shader programs. If compared to the classical rendering approach of OpenGL, which is older and consequently has a wider range of features and functionalities than WebGL at this stage in its history, WebGL does appear to have some limitations. However, these are easily negated as WebGL has adopted the most commonly used functionality of OpenGL ES

2.3 Dart

While there are many languages with built-in support for WebGL, this framework uses Dart. Dart is is a new programming language from Google, which is designed for use in web applications. Dart is similar to JavaScript in that it runs within its own VM, thus allowing for support across a range of devices. Dart was designed with a view of providing excellent efficiency; this not only includes application performance, but includes improved development experience through the addition of modern language features[122]. As Dart has been modified and adapted it has become possible to convert Dart into JavaScript in order to increase the range of supported devices[77]. Additionally, projects like Flutter- which aims to create cross platform applications with near native performance[33] utilise Dart as the core language for creating applications that are designed to run natively across iOS and Android devices[32].

2.3.1 Dart's Design

When developing for platform independence there are two major aspects to consider: the structure of the language; and the language's performance. Dart offers a range of improvements in the way of classes, isolates, and data types. In addition to this improved structure, Dart innately seeks to gain further performance increases by taking advantage of unique device features such as Single Instruction, Multiple Data (SIMD)[66].

Dart offers a range of new features when developing for the web, which were previously unavailable to developers. These new features, which are not currently supported in JavaScript, are predominantly represented by classes, the integrated development environment, and isolates[40]. The implementation of classes within Dart allows for newer programming methods to be implemented which were not possible with JavaScript. Whereas JavaScript has web workers, Dart has isolates to implement multithreading. Isolates allow tasks to be offloaded to other threads.

In Dart every object is an instance of a class. Classes in Dart offer a method of grouping functionalities that pertain to a particular type of object so that multiple instances of said object can be produced. Dart classes use Mixin-based inheritance to maximize code reuse by allowing for classes to be extended in multiple class hierarchies. Within Dart the use of classes allows for many optimisations to be completed in the VM in order to improve performance, whereas JavaScript requires customised, confined optimisations and creative coding to produce even a slight acceleration [34], such as those added via Googles Closure Compiler [38].

Dart supports a range of built in variable data types, which all manifest as first class objects. As all variables in Dart are objects a developer can choose to use predefined constructors, or to create custom implementations. Variables defined as a `var` can be treated as an integer or other supported type, until its type is confirmed; once the type is confirmed, it cannot be subsequently changed. This approach differs to that of JavaScript in which a variable can dynamically change types throughout runtime. Defining the data type of a variable (as is practiced in Dart) means that the developer knows what kind of data they will be working with, and what process can be applied to the variable. As data types within Dart must be defined arrays will only ever contain a single data type. For example, within an array of strings, each element will be a string. This differs from JavaScript in which a variable within an array could be any of a number of data types, such as object, integer, or string[73].

Concurrency is achieved in Dart via the use of isolates, in which each process within Dart runs in its own isolate [77]. Each isolate is allocated its own memory heap, in order to ensure that no memory is shared between processes, thus reducing the likelihood of race conditions [40]. Each isolate is run on a separate thread with its own event loop, therefore limiting its reliance on other isolates. As isolates run independently, it is essential that they are able to communicate. This is achieved by sending messages through ports, using send port and receive port[101]. The set up and running of isolates allow for web applications to be designed for, and to take advantage of, multithreading, in order to offer improved performance.

The design decisions of Dart, namely the use of first class classes, functions and defined data types, enables an IDE to perform static checking before an application is run. This helps minimise runtime errors caused by variables changing types, as can occur within in JavaScript. Dart's IDE also allows for the managing of third party libraries, which can be searched and downloaded through the IDE. Through the IDE these libraries can have version limits placed upon them, thus allowing for a range of versions to be used. This means that limits can be applied to stop a Dart package using a version of the library that might break the application. Dart's IDE also supports the dart2js function, which converts Dart code into JavaScript, hereby increasing the range of platforms on which an application can be supported.

Figure 2.8 shows the different processes required to run Dart code on various platforms. It demonstrates that Dart can not only run natively within the Dartium web browser, but can also be run in other web browsers using JavaScript. Additionally, this shows that the Dart VM can be used to create server-side applications, such as web servers, thus allowing for Dart to be the main language for both client and sever side applications[39]. To extend on Dart's capabilities and adapt for mobile devices, the Flutter framework allows for applications to be written in Dart, and be executed natively on both iOS and Android; Flutter and its functionality are covered in more detail later in this Chapter, and Chapter 3.

2.3.2 Development in Dart

Dart's support also allows for the implementation of the constant updates of Dart through the IDE. These updates improve upon Dart's existing code and to fix any bugs that may have been discovered. These updates not include fixes, but also new features, and optimised methods, which replace outdated ones. Some of the new features are created to assist developers in taking advantage of all the power offered by a modern CPU. New features also include ways to improve not only performance, but also the battery life of

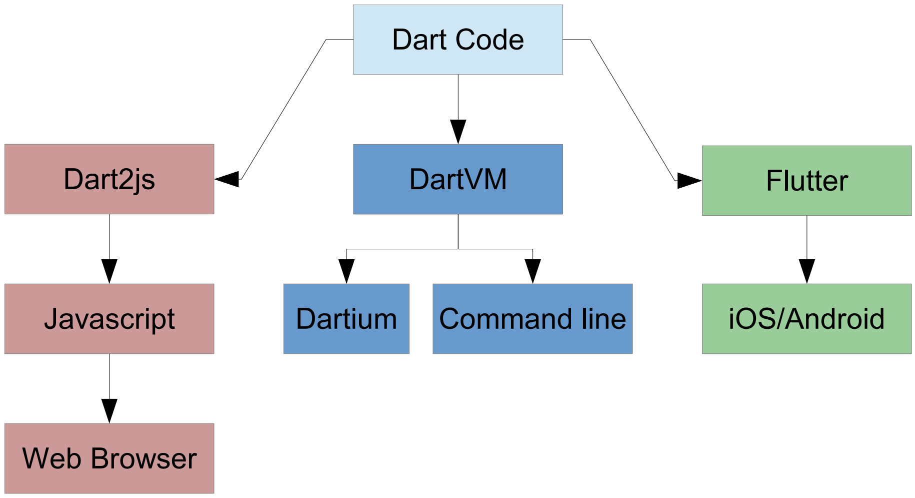


Figure 2.8: The execution of Dart code for a given platform, and the proccess required.

devices by using more efficient calls [65].

The original design goal for Dart has always been a focus on performance through rapid runtimes and fast compilers [24]. This focus is now realised in two ways: the use of the Dart VM; and the use of the dart2js compiler. The Dart VM, which Dart supports in order to improve performance, focuses on the addition and use of new features, such as the introduction of SIMD. SIMD allows for multiple pieces of data to be processed using a single instruction step, as opposed to processing multiple pieces of information the same way, multiple times[66]. This offers significant performance increases when dealing with graphical applications, especially as many of the calculations within graphics revolve around the use of 4x4 matrices.

Dart2js, on the other hand, focuses on optimisations that are to be produced in JavaScript code. Dart2js analyses the Dart application to make optimisations in a similar manner to how GCC optimizes C code. These optimisations work by moving code around and changing ordering [25] in addition to using minification, and compression. One final step that occurs in both Dart's VM and dart2js, is the use of tree shaking. Tree shaking is the process of removing unused code from an application to further reduce its size. Both the dart2js compiler, and the DartVM, provide Dart with two ways for offering a faster, and more effective runtime environment [120][122].

It is through both performance considerations, and language design that Dart has developed into an effective language, particularly in the context of modern applications. Dart offers a consistent development environment through the IDE and language structure, along with performance gains though the VM or dart2js compiler. Essentially, Dart utilises the newest technologies, and affords simplicity for the user when designing an application.

Listing 2.4: An example of Dart code, adding numbers to an array, then printing the array.

```
void main() {
List<int> x = new List<int>(10);


  for(int i = 0; i < 10; i++){
    x[i] = i;
  }
  print(x);
}
```

2.3.3 JavaScript

JavaScript remains an incredibly popular choice for programming on the web, despite its origins as a simple scripting language. It has become increasingly powerful through the implementation of improved virtual machines, such as Google Chrome's V8 engine; however it falls short of offering the many functionalities found in other modern programming languages[73]. Due to the ubiquity of JavaScript in web browsers it is supported across a range of devices. As JavaScript is still a relatively simple language it remains highly flexible in what it is able to achieve and how it works. However, this simplicity can also be disadvantageous as JavaScript was not designed to anticipate the many ways in which it is currently used[21].

JavaScript is supported on a range of devices, from smartphones to desktop computers. It runs within a VM in the browser, using an interpreter or just-in-time compiling, to create bytecode. These VMs are highly optimised in order to achieve a significant level of performance, using internal mechanisms to change data types in order to make operations less costly. Dart is able to leverage power from these VM's, particularly the Chrome V8 engine, by using the dart2js compiler. The JavaScript code produced by Dart is already highly optimized (as discussed previously) in addition to being designed to take advantage of a JavaScript engine's optimization.

One compelling advantage of JavaScript is its flexibility; this flexibility is limited to the VM within which JavaScript runs. This can limit what data it can access locally, or externally from another website, along with other various limitations. Also due to JavaScript's flexibility it is often difficult to constrain the behaviour of an object within the system [105]. Furthermore, this flexibility causes JavaScript to have few protections in regards to security, as it can allow for cross-site scripting [16]. This means that a developer needs to find a safe level of dependency when using third party libraries, or loading scripts from other sources.

JavaScript's flexibility imposes further limitations due to its small standard library, and its originally lightweight design. This can cause unique, and undesirable behaviour at runtime of which a developer needs to be aware when coding. This has prompted Dart, and many other languages, to be innately designed to circumvent a range of these issues that occur in JavaScript, such as those that are a result of all variables being defined as globals, unless declared as local[6]. Additionally, functions in JavaScript are not unique; meaning a single function can be defined multiple times, wherein the most recent definition is the one that is used. This means that the developer needs to carry out yet another check when using a third party library to ensure that there are no duplications in function names.

JavaScript excels at lightweight web based applications that require limited performance. Where more performance is required, or newer technologies are involved, Dart offers the portability of JavaScript through the use of dart2js, but also make high performance possible through VM optimisations, made available by the Dart VM.

Listing 2.5: The JavaScript code produced by running dart2js on the sample code in listing 2.4.

```
var H = map();

main: function() {
     var x, i, line;
     x = new Array(10);
     for (i = 0; i < 10; ++i)
        x[i] = i;
     line = H.S(x);
     H.printString(line);
   }
```

2.3.4 Flutter

Flutter is an initiative by Google to create native applications for both iOS and Android that are written in Dart. This supports the possibility of running Dart across multiple platforms with a native implementation to allow for high performance applications written using a single code base. Furthermore, applications written in Dart using Flutter can be updated over the web as opposed to all updates being accessed through the app store[32], thought Apple does apply some restrictions on downloading executable or interpreted code. Allowing for Dart to run natively on a mobile device means that applications can fully utilise a device's performance.

2.4 Summary

Dart is soundly constructed to support the creation of high performance web applications through its design. Dart offers improved performance over that which is provided by JavaScript, while still being able to make use of devices that do support Dart through the use of dart2js. Dart2js helps Dart to maintain a balance between performance and portability. This portability, along with frameworks like Flutter, make Dart an ideal language for achieving platform independence without compromising on performance.

The combination of WebGL running within a Dart application allows for a fast and efficient rendering application based on the web. Although WebGL does have some limitations, these can be mitigated through design; furthermore, many of these limitations become obsolete with the release of WebGL 2.0 standard. Dart allows for applications that run quickly and at high level of performance to be created for the web, with enhanced structure and design for an improved runtime and development experience. It is the bridging of these two pieces of technology that allows for the creation of a framework that has the potential to make platform independent graphical applications a genuine possibility.

Chapter 3

Utilising Dart for Platform Independence

Platform independence is an essential aspect of this framework. Dart is used to achieve this through several methods, the most significant of these are: Dart's VM; and dart2js, which converts Dart to JavaScript code. To a lesser extent, Flutter is also important for achieving platform independence. The Dart VM can be run directly on desktop, and through Flutter on mobile devices[33]. Devices upon which Dart is not supported or available can be provided for with the dart2js function. This converts Dart code to JavaScript, which is supported on most modern web browsers, therefore making Dart applications accessible to those that are unable to support the Dart VM.

Whereas Dart has built in support for multiple devices, it is up to the developer to create the appropriate functionality for interpreting user interactions. This framework utilises the creation of a controller class to assist the developer in fulfilling this requirement. Predominantly, the information collated by the controller class is passed to the camera class, which manages user movement within a scene. However, the controller class could also be utilised to create other interactions or user commands, such as changing the resolution of an asset, or performing an action against another object or asset. The setup

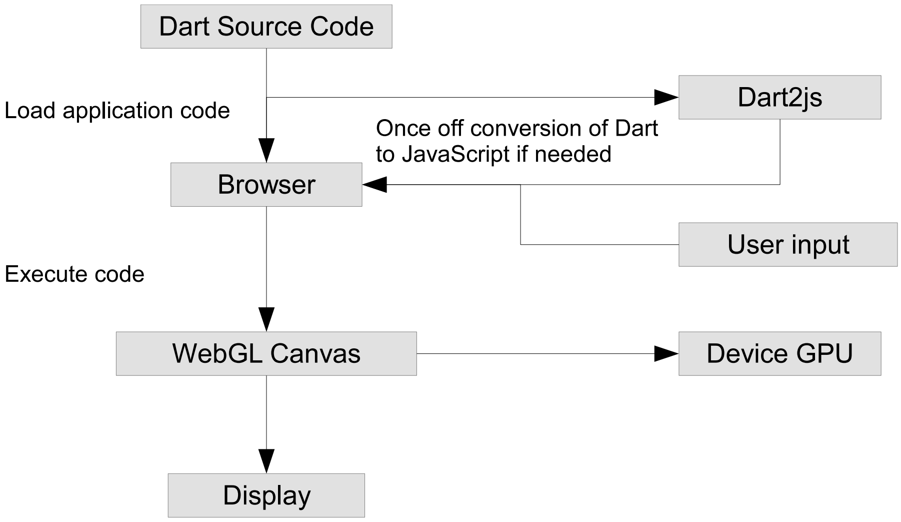


Figure 3.1: The life cycle of a Dart application as it renders a scene in a web browser.

of the controller class, how it manages various input types, and how those interactions translate into actions will be discussed later in this chapter.

The use of platform independent technology raises a point of difference in between that of itself and device independent. Within this work, an emphasis is place on the ability to utilise as much of a devices capabilities as possible rather than just being able to run across a range of devices.

3.0.1 Dart's Virtual Machine

The Dart VM is currently used primarily for running server side applications, along with applications designed to run within the Chromium browser. An advantage of writing code in Dart is that the same language can be used for both client and server side development. To use the Dart VM on a server a developer simply needs to install the Dart SDK, then they will be able to run a Dart application. Additionally, an application written in Dart can simply run within the Chromium browser without the need to install peripheral add-ons, as Dart's VM is built into the Chromium browser.

The Dart VM works directly with the Dart language, thus skipping the intermediary step of producing bytecode, as is often performed with other languages. This is unlike

Java, which is converted into Java bytecode to be run on the Java Virtual Machine (JVM). Within this framework the two keys differences between these approaches are: what the VM is sent; and how the code can be optimised.

Using the JVM allows for other languages to be compiled into Java bytecode, to then be run on the JVM[106].

Dart's VM, on the other hand, is closer in design to that of JavaScript's, wherein the language itself is interpreted by the VM [31]. This has the advantage of not allowing flawed or dangerous bytecode to be parsed into the runtime environment. Where the VM needs to run code from other languages, all that is required is for the language be converted into Dart; similar to Closure converting into Java bytecode[8]. Additionally, the Dart VM implements just-in-time compiling, whereas bytecode based VMs interpret exactly and in a linear order. The predominant performance issue of using a language based VM, as opposed to a bytecode based VM is that the code base is slightly larger as source code. However, using a language based VM does allow for increased flexibility when being interpreted, which is a highly valuable attribute in the context of web-based applications.

Listing 3.1: An example of Dart coding using SIMD to add two sets of numbers together

```
void main(){
  var a = new Float32x4(1.0, 2.0, 3.0, 4.0);
  var b = new Float32x4(5.0, 6.0, 7.0, 8.0);
  var sum = a + b;
}
```

3.0.2 Dart2JS

The Dart package includes a process called dart2js. This is what is used when converting Dart to JavaScript[3]; it has been designed to allow for Dart code to be run across a range of web browsers. This functionality has become somewhat of a main focus of the Dart project, and, in turn, has not only become increasingly stable, but its performance is also able to exceed that of code natively written in JavaScript on the V8 engine, when used in Chrome[120]. This performance ability is due to dart2js having been created by the same developers who created the V8 engine that is used by JavaScript, thus allowing for the converted code to benefit at least currently, from targeted optimisation. Dart's VM also introduces support for SIMD operations in the web; which is used in a small capacity within this framework, but is worthy of future investgation. Part of the dart2js optimization involves tree shaking; this is a process by which the application will try to remove any unused code from the source, resulting in smaller file sizes. This is especially important within in the context of web applications, as the larger a file, the longer it will take to load and process.

Listing 3.2: The JavaScript code produced by running dart2js on the sample code in listing 3.1.

```
NativeFloat32x4: {
  static: {
    NativeFloat32x4$: function(x, y, z, w) {
      var t1, t2, t3, t4;
      t1 = $.$get$NativeFloat32x4__list();
      t1[0] = x;
      t2 = t1[0];
      t1[0] = y;
      t3 = t1[0];
      t1[0] = z;
      t4 = t1[0];
      t1[0] = w;
      t1 = new H.NativeFloat32x4(t2, t3, t4, t1[0]);
```

```
        t1.NativeFloat32x4$4(x, y, z, w);

        return t1;

      }

   }

}


main: function() {

      H.NativeFloat32x4$(1, 2, 3, 4).$add(0, H.NativeFloat32x4$(5, 6, 7, 8));

}
```

3.0.3 Flutter

There may be some occasions at which native performance on a mobile device is preferred or required. Although, dart2js can allow a Dart application to run on mobile devices, it cannot provide native performance. However, Flutter has been developed to allow code written in Dart to natively be run across both iOS and Android[33].

Part of the Flutter implementation process is to create and install an application, that has beenwritten in Dart, as opposed to the other methods of implementation that require the application to be accessed via a web browser. Having a native application installed on a device offers performance improvements along with allowing for access to native functionalities that might not be available through a web browser's VM.

Flutter applications are written using the Flutter SDK, utilising much of Dart's existing functionality. Developing a Flutter application follows much the same process as developing for iOS with Xcode, or for Android with Android Studio in that the application is developed within Android Studio, which is able to utilise the required SDK.

Applications created using Flutter also have the advantage of allowing for an application to be updated over internet, rather than requiring updates to be pushed through an app store. This means that by using Flutter an application is able to change dynamically like a website, but still offer native performance.

If an iOS or Android device has a web browser, the Dart application may still be accessed this way. However, the application will not benefit from the same performance availability as a native application created via Flutter could benefit from.

3.0.4 Supported devices

As Dart is designed to run across a range of devices this framework must have the ability to dynamically adapt according to any given device's performance specifications, and without regard to device's operating system. In terms of performance capability, devices have traditionally been divided into three groups: smartphones, tablets, and PCs.

Smartphones are constantly pushing the boundaries of performance, and of what can be completed on portable devices. Using ARM CPUs, smartphones can benefit from superior power efficiency in lieu of high performance. The performance on smartphones also ranges greatly between individual devices; some top end phones contain a multicore ARM, while others only support a single core. These disparities in performance ability not only between smartphones, but between all devices, make it necessary for platform independent applications, which are designed to run across a large of range of devices, to be able to scale based on a device, and to be able to run at an optimal setting at runtime.

The next available increase in terms of performance comes in the form of tablets. Tablets range in CPU types; some support an ARM based CPU, while other use x86 architecture. Generally, tablets have larger screens than smartphones, although there are some smartphone screens that come close to tablet size. Many tablets have also begun supporting additional peripherals, which allow for a larger range of input methods, meaning that users are no longer limited to just a touch screen. With these types of specifications we can start to expect an increased level of performance output from high-end devices, in addition to some having dedicated GPUs[15]. Additionally, as technology advances, this level of performance will become available to smartphones, although smartphone screen size is unlikely to consistently become as large as tablet screens.

PCs are the most customizable device type, and as such can range greatly in performance and capabilities. Most modern PCs support a multicore CPU, and utilise the x86 architecture; however, despite these commonalities, there is a large discrepancy in device performance. This performance variation can be caused by a number of factors, predominantly a device having one the following: an integrated GPU; an external GPU through the PCIE slot; or, no GPU, therefore using the CPU for rendering. Additionally, other supported accelerators on a PC, like a GPU, may or may not be available, thus a framework designed to run across multiple devices must be able to scale as required. Furthermore, PCs also offer the largest range of input peripherals, such as the Leap Motion, gamepads, and other input controllers. Consequently, this means there is a need to accommodate how a framework controls the large range of input methods by which a device may be interacted with.

Device type	Processor	Input
Smartphone	ARM CPU	Touch, Accelerometer, Orientation sensor
Tablet	ARM CPU, x86	Touch, Accelerometer, Orientation sensor, Keyboard
PC	Integrated GPU, External GPU, CPU only	Leap Motion, Keyboard, Mouse, Touch, Gamepad, etc

Table 3.1: Example of different inputs for various devices

In order to get the best results from an application when it is designed to run across mobile devices and PCs, it needs to have the ability to dynamically scale up or down in order to best match the given device's capabilities. This dynamic scaling is developed from creating algorithms that support a range of different levels of detail. This enables the application to take advantage of a device's available resources in order to produce the highest possible performance.

3.0.5 Performance

Within the context of platform independence and this framework, device performance should dictate the requirements and scalability of an application. Within this framework, this is divided into two main considerations: devices with, and without GPUs; and, what CPU is being used. The presence or absence of a GPU will create different requirements when a scene is being rendered, whereas the type of CPU that is being used will influence the rate at which data can be generated. Regardless of the GPU or CPU type, this framework is designed to run on any device; however the difference in quality of output between devices may be minimal or significant, depending on how the device has been able to manage the dynamic creation of content. The use of Dart combined with WebGL does produce a small degree of abstraction that means the developer does not need to account for the exact processing details of the hardware. However, significant hardware features, such as the use of a touch screen, or absence of a GPU must still be considered in order to achieve a suitable level of performance of an application that has been designed to be platform independent[121].

Where possible, if the GPU has been tasked with rendering the scene, should the scene be too complex, or the level of detail too high for the capability of the particular GPU, the framerate will drop. On the other hand, if level of detail is set too low, the quality of the scene produced is impaired. This cause and effect relationship makes it necessary to balance out what the minimum amount of data is necessary in order to achieve a base scene, while also giving additional information in order to improve the look and feel of what is being rendered, but still maintaining a solid framerate. Additionally, if a device does not have a dedicated GPU the rendering is performed on the CPU, which can further throttle a device's performance as the CPU is no longer occupied only with generating data, but also the rendering of data[81]. Combining WebGL and Dart allows the developer to have specific control over how a scene is rendered within a web browser with regard to considerations such as the level of detail to be taken into account when displaying a scene, while still allowing for an application to remain platform independent.

With respect to CPUs, using languages like Dart allows developers to extract themselves from low-level considerations, such as setting instructions for different CPU architectures without the need to recompile code. This is one of the key motivators for platform independence, and Dart is able to cater to this desire absolutely as it negates the need for developers to devote significant resources to accounting for different CPU architectures. This is due to code written in Dart being processed by the Dart VM, or the JavaScript VM. The VM has the ability to produce highly optimized code for a range of devices[24], which includes a range of specific instruction sets, such as ARM's RISC (Reduced Instruction Set Computing) or Intel's CISC (Complex Instruction Set Computing). Both of these types of CPUs have different advantages to the other, which is why they are used in different devices. x86 is primarily used in PCs and servers, although it is being introduced for use in tablets. Currently, ARM is predominantly used in smartphones and tablets, but also in other devices in which power efficiency is prime[49].

In order for an application to successfully be platform independent it must be able to run across multiple devices, and, at runtime, adjust to whatever resources are available in terms of raw processing power provided by the CPU, and GPU, if applicable. Dart manages the application's response to CPU capability, through its VM so that the developer does not need to account for this; whereas the combination of Dart and WebGL handles the specific processing aspects of the GPU, or lack thereof, so that the developer need only consider more significant hardware concerns.

3.1 User Input Management

To take full advantage of the platform independence provided by Dart, regardless of platform, the developer must still consider the manner in which a user interacts with an application. As the range of possible inputs for a device is ever expanding, so are the manners in which they are used to interact with a device, such as the Leap Motion introducing gestures as a possible way in which a user can interact with a PC[124]. This means that applications must be able to accept and interpret a range of different input types, then correlate them into a single action that the application should perform. For example, the input from a gamepad should manifest in a similar action to a touch screen event. This framework manages different inputs through the use of a controller class, which is discussed later in this chapter.

For most mobile devices, the main inputs will derive from touch screen and device rotation on smartphones and tablets. PC interaction is most likely to come from mice and keyboards; however, in order to keep pace with the latest technologies, applications that seek to truly be platform independent should be able to receive input from PCs that incorporate a range of input devices, such as touch screens and Leap Motion.

In general, many interactions can be interpreted similarly, such as inputs between touch screens and mice - a mouse double-click and touch screen double tap can be dictated by the developer to result in the same action. Other similar actions between these two devices would include: touch screen swiping, and mouse cursor movement. Other actions, such as: a double finger tap; holding down a finger; and multiple finger inputs can be defined by the developer to be interpreted by the application as similar actions to other inputs from other devices such as mouse right-clicks.

Advances in technology now mean that gestures are a vital part of reading in and understanding the user input. They tell how the user is interacting with the system - interpreting this information is a main function of the controller class. Gestures such as pinch-to-zoom have become common features on many applications on touch devices [100]. In the interests of creating user-friendly applications it is preferable to include these existing gestures to keep consistency within the system and to reduce the learning curve new users will be faced with when learning how to interact with a given environment. There is an ever-increasing variety of inputs and their associated gestures available across a range of systems. Such examples include the rotation and orientation sensors on smartphones, and the hand positioning information gathered from a Leap Motion[82]. These inputs, although they are different from sensors, can be treated similarly when dealing with gestures. This is possible as through the Leap Motion rotation and orientation information of the hand above the sensor can be gathered.

There are some scenarios wherein it may be more suitable to ignore the inputs given. For example, for some applications it may be desirable to ignore general hand movement from a Leap Motion, but to convert gestures to meaningful actions.

3.1.1 Controller Class

In order to account for the large variations in possible inputs, this framework has implemented the controller class - a modular approach to manage converting device inputs into meaningful interactions. A controller class is essential for creating applications that take full advantage of the platform independent possibilities made available by the use of Dart. The specific inputs that were considered for this framework are: Leap Motion, keyboard, mouse, touch screen, and gamepad. The controller class takes all available inputs and determines which of these will be most beneficial for use within the application, as defined

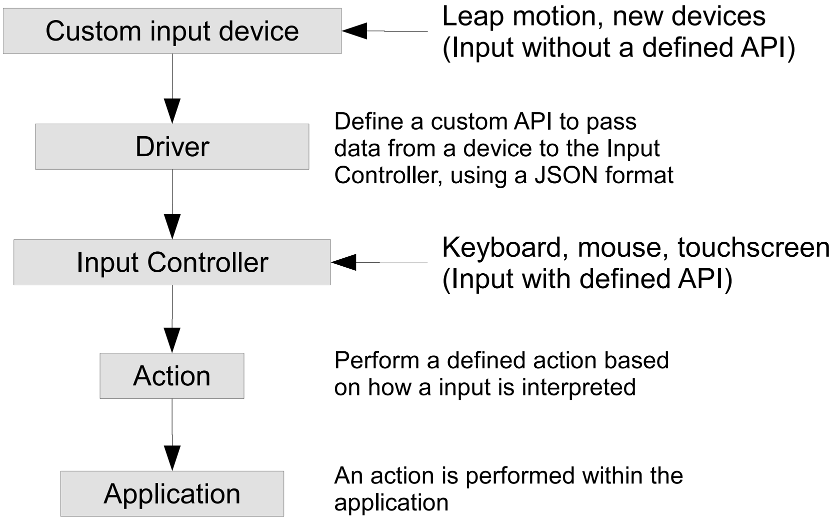

Custom input device
Leap motion, new devices
(Input without a defined API)
Driver
Define a custom API to pass
data from a device to the Input
Controller, using a JSON format
Input Controller
Keyboard, mouse, touchscreen
(Input with defined API)
Action
Perform a defined action based
on how a input is interpreted
Application
An action is performed within the
application

Figure 3.2: The flow of input data within an application via the controller class within this framework

by the developer. Some inputs, which are not natively supported by Dart, can be added through the use of a local host web server[68].

The controller class checks if a type of input is supported, and consequently also contributes to determining what kind of device is being used, in addition to the information provided by the users browser. For example, the lack of an accelerometer would contribute to the identification of a PC. Based on the discovered inputs, the controller class will set up a range of device listeners that will be called to interpret information passed from the device. Received data is converted from its various forms, such as hand placement or orientation, into the appropriate forms to be passed to, and used by, the rest of the application. The controller class is designed to allow for users to remove some inputs; for example, to ignore the information from a Leap Motion, but to utilise the information from a keyboard instead. Mapping key inputs is necessary in converting functions, then calling updates with the new values. An additional feature of the controller class is gesture recognition, that is, how a hand waving in the air can be used to move around a simulated world[68]. Each of these build upon one another to comprise the controller class.

The controller class is able to explicitly state how the application is to interact with, and respond to the environment, based on the given device's inputs; it does this by checking what inputs are currently available. This is ascertained by performing basic checks with a given API, such as confirming if an input is supported through the touch API. Subsequently, the controller class will locate the available inputs and create the relevant device listeners for a range of inputs to be interpreted, based on the device's inputs. A key feature of creating the controller class is that it allows for a maximum amount of user input options as it also allows for additional inputs to be parsed through it. Once an event listener has been created and added, any event called with the given information will be passed to the controller class for processing.

The controller class simplifies and removes the need to create multiple functions for

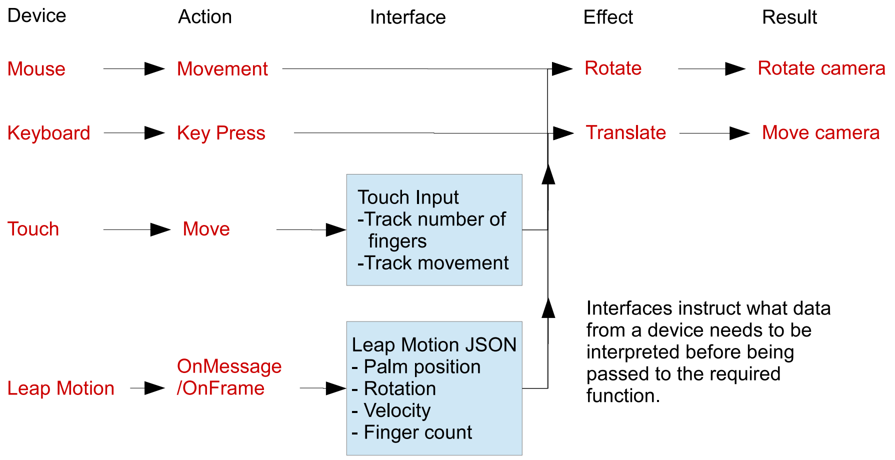


Figure 3.3: An example of how the controller class will process a range of inputs, but will produce the same effect.

multiple inputs. It does this by mapping various inputs that share similar functionality, such as the touch events and mouse events both being used to control the scene in the same way, so that the controller class will interpret both of these inputs in the same manner meaning that each output will result in the same action. This, therefore, removes the need to rewrite an additional function that is almost identical. The class distinguishes between alternative inputs that are essentially performing the same event or action, then creates the output accordingly. Figure 3.3 shows how various inputs can be mapped to manipulate camera positioning.

The user input is controlled by various event listeners, which are set up during the initialisation. These event listeners will call a function within the controller class whenever an input is registered with them. It is this input that needs to be processed and converted into usable data. The conversion process is performed by the controller class, which will then process the information and pass it along to the camera class or simulations as required. A number of measures have been included to ensure that the information passed is correct, and that the system is not overloaded with multiple inputs sensors. Such measures include: the limiting of inputs within the system; how frequently data is gathered from input devices; and combining inputs to create a single input for processing.

Event listeners are a tool used to trigger events when certain actions occur, such as a key press or a rotation on a phone. These event listeners are the way in which most inputs are read into the system to be processed; they are set up in various ways in Dart for different input methods. The most common of these is the implementation of a certain event class to handle different input types, such as the touch event, which handles touch input. Although more uncommon, but just as effective, is the implementation that is used in the Leap Motion wherein data is passed through a local host web server via a JSON file. This technique was also used for gamepads, until the newer gamepad API was released[22]. In this instance the advantage of using a local host web server is the ability to pass user input data through custom event listeners to be processed.

As a range of input methods are possible on some devices this can lead to an influx of information that is not relevant to the user's interactions. To overcome this the available inputs must first be identified, then those sorted as to, which are the most relevant as defined by the controller class. Based on these inputs, only certain event listeners will be set up. Advantageously, this method limits the inputs to only those that are relevant to the device. For example, on a smartphone a mouse event would not be used, rather a touch screen event would be used instead. This process allows for only relevant information to be passed to the controller class, therefore limiting the amount of input that requires processing, and reducing the passing of redundant information. Consequently, this also reduces the processing power required to run the application.

3.1.2 Custom inputs

Custom inputs exist as a way to increase the amount of inputs available extending from the ones that are natively supported within a browser's VM. These custom inputs are implemented in a range of ways in order to pass data to the application. An example of this would be the Leap Motion requiring the use of a local host which passes data from the local host to the client within Dart. Previously this was also how data from a gamepad was passed and interacted with; however this has recently become a standard within browsers so now the gamepad API exists. The act of creating custom inputs can result from a combination of various methods and processes. One of these processes must relate to the setting up of one of these custom inputs, which requires the use of device drivers that pass data to the local host. This local host must be a server that is able to create and pass JSON files to the client. Additionally, the local host assists in the processing of data that is not natively accessible from within the browser's VM. As technology advances, current custom inputs will become standard, like the gamepad; however new technologies will also be created meaning workarounds will always be required for utilising custom inputs.

A current example of a new technology that does not yet have a standard is the Leap Motion. It is an input device designed to read in information about a hand placement in 3D space above the sensor. Within this framework the input is taken not from a native event listener in Dart, but via the use of a third party driver and library. The driver acts by creating a locally hosted web server on a device that the Leap Motion Dart library can connect to. This locally hosted web server passes information to the Dart application through a JSON[86] file, that contains a range of information about a hand's location above the sensor, such as its orientation and rotation along with the number of fingers that are currently extended[79]; additional information can also be accessed if necessary.

In the situation of the Leap Motion the driver installed, which interacts with the physical Leap Motion, also creates a locally hosted server. This server is hosted on different ports as opposed to port 8080, which is used for web traffic. As the Leap Motion drivers can only be installed on a PC this means that devices such as smartphones are unable to use this input method. The creation of a web server allows for the data from the Leap Motion to be transferred via local host to the client. This functionality can also be extended to allow for another device to connect to this web server, then interpret the movement information from the Leap Motion that is stored as a JSON file. This JSON file stores a range of information about the hand's position in 3D space that can then be accessed easily on the client.

Within the wider context of custom event listeners, there are numerous advantages of storing data as a JSON files. Most significant of these is that using a web server to host this JSON data allows for the data to be passed easily through to the application without the need to create a custom API when using newer devices that do not yet have an existing, native API in browsers. As the web server acts as a local host, this means that the data transfer between local host and the application is minimal.

Using JSON files also allows for data to be passed as a JavaScript object. This in turn means that the data passed is easily accessible as, once parsed using Dart's native JSON parser, the data can be accessed as if it was part of a object. Furthermore, this means that functions can be written into the JSON file if necessary. All of these attributes contribute to an uncomplicated flow of data, which is then interpreted by the client.

Once information is collated into the JSON file it is accessed via a custom library and a custom event listener is implemented. For the purpose of the Leap Motion, the Leap Motion event listener is a class created by the Leap Motion library. This library has the ability to return information in 3D vectors, lists, or single integers depending on what information is being requested. It is this information that the controller class requires in order to allow for interactions within the scene.

The Leap Motion library allows relevant information to be passed to the controller class. This information is used to show interactions within the scene, such as movement of the camera along the X, Y and Z axes, or rotation of the camera. This information is passed through the Leap Motion controller as various matrices. This can then be interpreted by the controller class as an instruction to either rotate or translate the camera within the scene. Accessing the information about the number of fingers that are visible can be used to specify the direction of the translation or rotation. For example, if one finger is held up the translation is performed upon the X axis; two fingers result in a translation the Y axis, or three fingers results in the translation occurring on the Z axis. It is by combining these different inputs that the precision of the inputs can be increased, and the way in which the developer wishes them to be interpreted.

The relevance of multiple inputs for platform independent applications will differ according to the design, developer intent, and client use. Within this framework, much of the design reflects the intention of creating graphical simulations. For this reason, the controller class is principally involved with gathering data related to how a user may wish to move about, and interact with a simulated environment. To process this information, this framework implements a camera class.

3.1.3 Camera Class

The camera class is designed to control how the scene is viewed. This is managed by the use of view matrices, the projection matrix, and the model matrix, which interface with the controller class allowing the camera class to determine how a 3D scene is displayed. These matrices are updated via the controller class, which is based on user input. Within Dart, the camera class can take advantage of SIMD within matrices, which are vital for determining how a scene is viewed.

Each object within an environment is managed by its own view matrix, which is concerned with the rotation of the object in relation to the user. The data from each view matrix is collated by the model matrix so that there is continuity between how each object is being viewed in relation to the entire environment. The projection matrix then takes information about the shape of the camera, and the positioning and area of the viewing plane to collate all the data from the viewing and world matrices into a single image.

Whenever a user changes the way in which they are interacting with, or viewing the scene the corresponding information is gathered and interpreted by the controller class. This input is then converted and passed through various functions to the camera class. These functions can include translation and rotation across a range of axes, but can also be used to change other aspects of the camera class.

For camera movement across the X, Y and Z axes SIMD is a powerful tool that allows for rapid processing, particularly with 4x4 matrices, which are used within the camera class. To take full advantage of these this framework uses the vector math library written by John McCutchan[66]. This library allows for the creation of matrices that are natively accelerated using SIMD, as well as offering a range of functionalities that aid in the integration of the camera class to further optimise the creation and viewing of a scene.

As the camera class is interfaced and managed by the controller class the user is able to control how the scene is viewed, while taking advantage of the acceleration offered by SIMD as a result of an application being written in Dart. The camera class is only one example of how the controller class can be combined with innate features of Dart, such as SIMD, to provide rapid performance within platform independent applications.

3.2 Summary

Dart offers a comprehensive array of methods by which a developer can achieve platform independence. The Dart VM caters to applications that are more server based, whereas the dart2js function enables applications to run on any browser that supports JavaScript. Additionally, Flutter allows applications to run natively on iOS and Android devices.

To more comprehensively achieve platform independence a developer needs to account for the wide range of potential inputs across all platforms. This can be accomplished through the inclusion of a controller class. Where applicable, the controller class can also be grown to include custom input for emerging technologies that do not yet have standards. Within this framework, the controller class is predominantly occupied with the management of the camera class. However the function of the controller class can be customised to best suit the needs of whatever type of application may be developed. Where the controller and camera classes are created to provide comprehensive platform independence, they are further supported by Dart through Dart's in built functions such as SIMD.

Dart offers developers a robust and powerful base upon which to build applications that seek to be platform independent. Where a developer is required to undertake additional work and improvements, inherent features within Dart will assist in creating applications that can take advantage of available devices resources to achieve high performance across all platforms.

Chapter 4

Procedural Content Generation

The use of Dart, and the inclusion of the controller class means that it is possible for an application to be run on, and utilise the inputs from many devices; however, without considering how the application will handle the performance capabilities of any given device the application may crash or not even run at all if it the device does not have sufficient hardware specifications to launch or maintain the application. Alternatively, the application may not be able to take advantage of increased performance availability offered by high powered devices. To cater to the variation in performance offerings from a wide range of devices, and the potential change in performance output during runtime, this framework incorporates three different content generation algorithms for the creation of landscapes.

Landscape generation is the simplest method of procedurally generating content and demonstrating, and testing performance across multiple devices. Terrain not only provides a base for larger, more complex scenes, but also lends itself to clearly responding to and showing the effects of scaling. A scene may need to be scaled when examined up close or from a distance, as well as scaling to suit performance availability.

4.1 Procedural Content Generation

This framework has been designed to procedurally create an environment that can be both generated, and maintained at a high level of quality for a given device. Accordingly, the environment is adaptive, and has the flexibility to dynamically scale up or down in quality. This dynamic scaling is based around developer defined metrics, as opposed to requiring users to manually adjust settings. This scaling changes the runtime and creation cost of the various assets that are being produced and rendered. The resulting assets can then be scaled at runtime based on the ongoing performance costs in order to create an ideal runtime environment.

Designing the algorithms that can create an asset with sufficient detail, but can also be scaled to suit a range of devices poses certain difficulties as an object must have instructions on how to create itself along with information specifying the performance costs involved. The algorithms used within this framework are able to access and meaningfully interpret a device's specifications in terms of functionality and performance ability. This means that the algorithms can use metrics to measure performance. There are two primary metrics: the one-off cost of setting up a simulation; and the ongoing cost of maintaining and updating said simulations. These metrics are essential to creating a dynamic and tunable base that can be used when designing and implementing dynamically generated content.

The advantage of dynamic content generation is that content created can scale across multiple devices. This means the quality of an asset created is limited only to a device's hardware. Additionally, as device performance increases so does the quality of the content created. By utilising this framework, an application can be scaled to run across a range of devices. Dynamic content generation also means that there is no need to create multiple iterations of the same asset at different resolutions.

Dynamic content generation lends itself to platform independence as creating content that can scale across multiple devices means that a single application can adapt and produce an ideal runtime experience for a user regardless of the device type. This is further improved by being implemented into a framework that utilises web technologies, such as Dart and WebGL, which are both designed to run across a range of platforms.

The quality of the generated content is only limited to a device's hardware, meaning that as a device's performance increases so does the quality of generated assets. Tying the content's quality to a device's performance means that the quality of assets improves consistently with the device. This means that when a user upgrades a component of a device, or the device itself, the application will automatically improve without the need for user input.

Any application that has been designed to utilise this framework can adapt to the device's performance. This means that trade offs can be made in order to achieve a given goal. An example of such a goal could be having a design preference towards high quality rendering, in which case interactivity within the scene could be limited, along with the updating of simulations. On the other hand if a low-end device was to run an application using this framework, the asset quality could be limited in order to achieve a constant level of interactivity, along with utilising a less computationally expensive simulation. Allowing these trade offs between different components of the system to occur means that the content can be generated not only to suit a device, but also to respond to the goals and intentions of an application's design.

This approach of utilising dynamically generated content also removes the need to create each asset at multiple resolutions. This means that in lieu of storing multiple premade resources, they are, instead, dynamically generated. This is key when dealing with web technologies, in which loading in large resources from remote servers can become time consuming and costly[76]. Additionally, using dynamically generated content means that the limit on the quality of the assets that can be produced is directly related to the resource that is available.

Content is procedurally generated using various algorithms or simulations to create data that makes up an object. As scalability is a key attribute in this framework, the algorithms are designed to scale to the limitations of the hardware. In this framework the algorithms chosen are designed to create a scalable environment that can adapt throughout runtime. This is achieved through the use of the diamond-square algorithm, the shallow water simulation, and the use of Perlin Noise. Using these three techniques in content generation means that there is more than one method by which the application can manage the runtime requirements on a given device. Alternative content creating algorithms were also researched, but were not used within the final framework, these include constructive generation, agent based landscape creation for terrain generation.

Designing an algorithm or simulation that can scale indefinitely is a core component to the goal of dynamic content generation. This means that when designing or implementing an algorithm or simulation, the variables used to describe the magnitude of the asset can be changed. Once the content of a simulation or algorithm changes these changes need to propagate throughout the system, in order to ensure continuity throughout a scene. This can be achieved through the use of functions designed to assist in the reconfiguring of content, as described later in this chapter.

By generating scalable content a developer can have an application run across multiple devices with varying performance, without the need to create multiple implementations of the same asset. This means that an application will perform as designed across a range of devices, without the need to redevelop code or content. Furthermore, designing a framework for generating content in this manner means that content is generated to the level that the device's hardware supports.

The diamond-square algorithm is designed for heightmap manipulation on a given mesh, allowing for the creation of a base terrain[74]. This heightmap can be translated into a series of vertices and indices to be rendered. The diamond-square algorithm incurs a once off cost when rendering, with minimal maintenance cost, with the exception of dynamically adjusting the size once created.

Based on a device's hardware, this framework implements one of two methods for generating bodies of water that are designed to interact with terrain; these are the shallow water simulation, and Perlin Noise. The shallow water simulation, designed as a simplified implementation of the Navier-Stokes equation, creates a fluid simulation[99]. Simulating water this way has a small initial generation cost to create the initial body of water, but has a much larger ongoing cost while simulating the movement of water throughout the scene. Perlin Noise can be implemented as an alternative to the shallow water simulation where available resource is limited. Using Perlin Noise simplifies the representation of water, as it does not simulate fluid; instead a simple wave-like, undulating surface is created, and updated[95]. The shallow water simulation offers a high level of accuracy in representing bodies of water and the way in which they interact with terrain, whereas Perlin Noise offers a lightweight alternative in regards to both generation and ongoing performance requirements.

Constructive generation is the process of taking a defined area, splitting into various sub zones, populating the area, then reconnecting each area. This is a popular method when generating rooms, or a dungeon as is often used in video games [42]. One of the more popular constructive generation algorithms involves taking the area, and producing a Binary space partitioning (BSP) tree. This is an idea method as for each area it will be split into two new areas. These new areas become children elements in the BSP tree, which are then split in to, until a minimum area is achieved. These small areas are then populated based on a defined tile set, filling in each area. These areas can then be connected to each other via hallways or doors based on the applications purpose, or theme.

While constructive generation can be beneficial when requirements such as size, and area are defined it is not as well suited for environments which are rapidly changing, as this can lead to a complete rebuild of the underlying BSP whenever an attribute such as area size is changed. As the framework is designed to adapt rapidly to these changes constructive generation techniques and methods such as these have not been implemented.

Agent based landscape on the other hand take the approach of taking a defined area, and having numerous agents run various constructive algorithms to enhance the landscape[42]. This approach gives a user greater control over what makes up a landscape or area, but at the additional cost and complexity of maintaining numerous agents. The approach of agent based landscapes is ideal for when a designer wants to generate a landscape, and be able to tweak the parameters. These parameter tweaks allow for a range of unique landscapes to be rapidly generated, and iterated upon. Agent based landscapes can generate a wide variety of terrain, given the use of few agents, it still falls short in regards of scaling, which is a key element to this framework.

4.1.1 Level of detail

In addition to generative algorithms used to create content, it is also important to take note of the various methods which can be used to achieve a desired level of detail for a given area of terrain. The need for using level of detail techniques is minimised in this framework, as all assets are scaled up to the desired level of detail rather than taking a high detailed model and scaling down, simplifying the mesh in the process. Common methods of utilising level of detail techniques are discrete, and continuous level of detail. While the algorithms used to simplify the mesh include edge collapse, catmull-clark, and loop subdivision, in addition to others[59]].

Discrete level of detail is the method in which geometry of a mesh is created numerous times, at various levels of detail. The geometry is then updated with a new set of precomputed geometry as required. Discrete level of detail works well when dealing with complex objects in which defined features are wished to be kept. This is because when creating the various sets of geometry, a designer can insure that the geometry maintains set characteristics. Insuring unique elements of a mesh of maintained also helps minimise the effects of popping, in which new features of a mesh can unexpectedly appear. Though this approach does have the down side of needing the store numerous copies of the same model to switch between as needed.

Continuous level of detail differs from discrete in that the mesh updates itself dynamically based on factors such as view distance, or rendering performance capabilities. This benefits the application as it does not need to store multiple copies of each mesh, but does mean that throughout runtime, a mesh needs to dynamically update itself. This redefining of detail to a mesh throughout runtime, starts with a high quality mesh, then slowly reduces its detail. This means that whenever the quality needs to increase, the mesh needs to be reduced from a higher level of detail[59].

There are numerous ways of simplifying detailed meshes, all largely based around reducing the amount of geometry displayed, while maintaining the objects shape, and structure. Simple algorithms such as edge collapse can achieve various levels of success, and are simple to implement. More detail algorithms such as polygon merging can affect more of a mesh, but at a greater performance cost.

Edge collapse follows the pattern of selecting an edge within a mesh, and removing given edges. The removal of an edge, will cause the two vertices ether side of the edge to merge into a single point. This edge removal may cause a triangle to become a two lines stacked on top of each other so steps must be taken to account for this, and ensure only a single edge remains.

Polygon merging is the process of taking a complex shape containing a range of polygons and merging two or more together, in order to reduce the complexity of a mesh. The major requirement for this, is that the polygons are coplanar. The new resulting polygon then needs to be triangulated to produce an object ready to be rendered.

Dynamically generating content allows for assets to be created to suit an ideal level of quality, as dictated by a device's hardware. In turn, this allows for a framework based around dynamic generation to implement trade offs when developing content in regards to different aspects of performance. To achieve this outcome the algorithms must be able to maintain continuity when dynamically scaling in size. These generative algorithms (the diamond-square algorithm, shallow water simulation, and Perlin Noise) lend themselves to the creation of a framework designed for platform independent applications.

4.2 Algorithms

Within this framework the content generation requirements have been split into two categories in order to better understand how they affect the performance of a system. These categories are: low ongoing cost; and high ongoing cost. Content generation that fits into the low ongoing cost category is rarely updated once it has been created; within this framework, the diamond-square algorithm is categorised as low ongoing cost. The other type of algorithms - shallow water simulation, and Perlin Noise - have a high ongoing cost, and are in a state of constant motion in terms of updates.

4.2.1 Diamond-Square

The diamond-square algorithm is used for generating a simple heightmap, which forms the basis of the environment. The process of generating a heightmap using the diamond-square algorithm is split into two core stages: the diamond step; and, the square step. To execute the first iteration of the square step the algorithm begins by setting the height values of the four corners of the map - this creates a square mesh, the size of which is predefined by the developer or user. Following the square step, the diamond step sets the height of the each of the four points between those which were set by the square step. Within this framework this second set of points could be thought of as identifying North, South, East, and West. Once the first square step is initialised a loop is used to perform the diamond step, followed by the square step until the algorithm has completely subdivided the mesh to produce the desired heightmap.

Within this framework these steps have been developed upon to allow the diamond-square algorithm to dynamically scale in order to accommodate adjustments of resolution throughout the runtime of the application.

The first step in creating the heightmap using the diamond-square algorithm is to initialise the mesh. Once the mesh is initialised the diamond and square steps are looped through the grid until the heightmap is completed. The initialisation involves creating a 2D grid with predefined corner values. These values influence, and ultimately define the height of the completed map, as the diamond-square algorithm utilises height variables to maximise the level of height variation possible when assigning height values to newly generated points within the map. As the diamond-square loop continues to refine the mesh the height values of new points also drop as the height values are assigned by taking the average between the two closest values.

Within the context of this framework landscapes are generated using random numbers; if, for example, the corner values were set to 0.0, and water level was defined as 0.0, the expected resulting landscape would have similar amounts of ground and water area, once the heightmap had been completed and water added. Following the initialisation of the mesh the diamond step sets the height of the center point to be the average of the height of the four corners. A small fluctuation is added based on the height value with the addition or subtraction of random noise.

Listing 4.1: The diamond-square algorithm.

```
//The defined size of the diamond square mesh
```

```
int tileResolution = 129;

for (int sideLength = tileResolution - 1; sideLength >= 2; sideLength =
    sideLength ~/ 2, height /= 2) {
  int halfSide = sideLength ~/ 2;
  //Squre Step
  for (int x = 0; x < tileResolution - 1; x += sideLength) {
    for (int y = 0; y < tileResolution - 1; y += sideLength) {
      double avg = heightMap[x][y]
      + heightMap[x + sideLength][y]
      + heightMap[x][y + sideLength]
      + heightMap[x + sideLength][y + sideLength];
      avg /= 4.0;
      double offset = (-height) + rng.nextDouble() * (height - (-height));
      heightMap[x + halfSide][y + halfSide] = avg + offset;
    }
  }
  //Diamond Step
  for (int x = 0; x < tileResolution; x += halfSide) {
    for (int y = (x + halfSide) % sideLength; y < tileResolution; y +=
        sideLength) {

      double avg = heightMap[(x - halfSide + tileResolution) % tileResolution][y]
      + heightMap[(x + halfSide) % tileResolution][y]
      + heightMap[x][(y + halfSide) % tileResolution]
      + heightMap[x][(y - halfSide + tileResolution) % tileResolution];
      avg /= 4.0;

      double offset = (-height) + rng.nextDouble() * (height - (-height));
      heightMap[x][y] = avg + offset;
    }
  }
}
```

Next, the square step runs through the grid using the centre point (created by the previous diamond step) and corner points to generate values that split the mesh into four smaller squares. Once this step is completed, the diamond step runs again, finding the centre points of the four smaller squares. Then the square step is run again, splitting the mesh into even smaller squares. This loop continues, each time in finer detail than the last loop, slowly refining the mesh until each point has been assigned a value and the heightmap is complete.

The ability to dynamically scale created content based on the runtime requirements of an application is essential to this framework; two different functions were created to achieve dynamic scalability of the generated heightmap. One function upscales the heightmap to enable more detail to be added to the model; the other function downscales the heightmap to remove detail, and reduce data costs.

Upscaling the heightmap is divided into three core components. Firstly, the correct data must be located in order to run the process that creates new points. Next, the new, bigger (therefore more subdividable) mesh needs to be created. Finally, the data sourced and processed from the first mesh must be placed into the newer mesh, which has been designed to replace the current one.

When upscaling the heightmap locating the required data points is a simple matter of using the existing points as a base. The more challenging aspect of upscaling a heightmap is creating the necessary space to allow for the generation of new data points between the existing data points. This is done by creating a new mesh of twice the original size, and assigning the previous values to the new mesh, with an empty space between each point. Once the appropriate spacing has been achieved in the new mesh, a single parse from the diamond-square algorithm populates the empty data points within the mesh. The result is a heightmap that contains all the characteristics of the previous map, but with more defined and detailed features. The code for this can be seen in listing 4.1. Once the heightmap has been updated the various buffers and attributes within the rendering engine are swapped out accordingly in order to create a smooth transition between resolutions.

When compared to upscaling the heightmap, downscaling is simpler as there is no requirement for the creation of the necessary space to accommodate the generation of new data points. Although downscaling does require the creation of a new mesh (in this case, a smaller mesh with corner values that create an area less than the existing mesh) the data is simply copied to the new mesh, skipping the necessary data points so as to reduce the detail and data requirements of the map. As with the upscaled map, once the new heightmap is set up the buffers are swapped within the rendering engine as appropriate for the new level of detail.

The diamond-square algorithm procedurally generates content that can be dynamically scaled throughout runtime. Generating the heightmap incurs a single, one-off cost during initialisation, but benefits from minimal rendering cost. The introduction of the scaling functions can introduce an ongoing maintenance cost as the area is dynamically increases or decreases in quality. However, the cost of this scaling is minimised as much as possible, and is unlikely to be a frequently occurring cost.

The diamond-square algorithm is a well-defined, reliable process for producing heightmaps, and, for the purposes of the this framework, the procedural generation of heightmaps provides a base upon which further simulations can be applied.

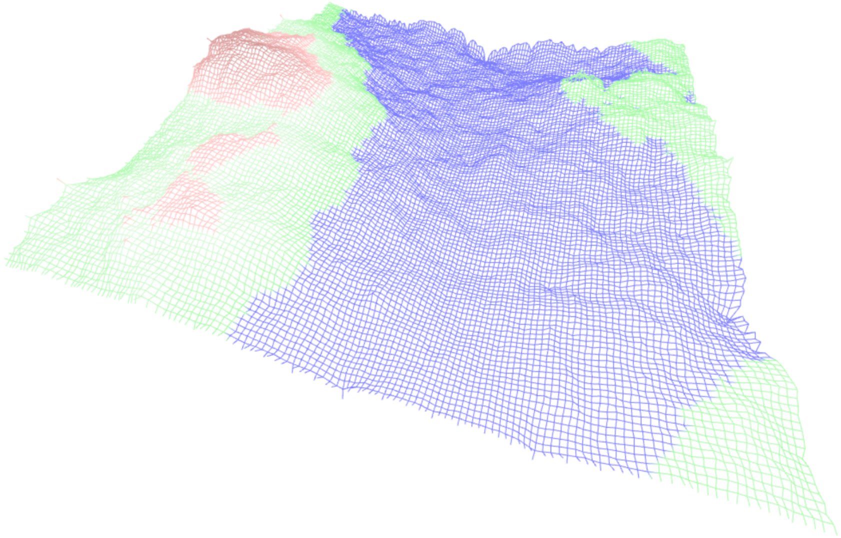


Figure 4.1: A basic 3D visulisation of the heightmap produced by the diamond-square algorithm

4.3 Water Algorithms

Within this framework the use of water simulations provides an opportunity to diversify the performance demands placed upon a system. As simulations incur an ongoing performance cost, this allows for an application to balance runtime performance requirements against producing the most visually pleasing rendering while generating content. This framework investigates two different implementations: the shallow water simulation and Perlin Noise. Each of these methods can be created to become dynamic and are therefore able to automatically improve the quality of a rendered scene.

4.3.1 Shallow Water

The shallow water simulation is used to accurately create a small, simple body of water. This simulation requires certain information for various aspects for the process, such as the area and boundaries of a given body of water. Once the boundaries and height coordinates for the water have been set, the user introduces the effect of kinetic energy (by suddenly increasing or decreasing the height of the water at a given point) into the system. The simulation then tracks the flow of water via velocity along the X and Y axes, and via changes in the height of a cell. Using a staggered grid, wherein the velocities are applied as an average across the cell, and the water height and ground values are stored central to the cell location, the information gathered from the tracking process is then used to adjust the height of a given water cell according to the formula (Navier-Stokes equation), thus creating waves and ripples in the water[85].

Within this framework the shallow water simulation is comprised of two main processes: initialisation, and updating. The first step in the initialisation process is to identify the points at which water should be located. This data is found using the contour tracing algorithm. The implementation for the initialisation is then based on the heightmap that has been created by the diamond-square algorithm, which allows for more accurate boundary conditions, thus allowing for improved interactions within the generated scene. Following this, there are various stages in the updating process that need to be managed. The updating cycle contains two principal steps: calculating of the advection of the simulated fluid; and the rendering of the water. The constant maintaining, and updating of the state of the water simulation further contributes to the runtime costs, which must be managed by the system in order to ensure the application is running optimally.

Initialisation of the shallow water simulation relies on information taken from the contour tracing in order to correctly locate where bodies of water are within a scene. As this framework focuses on optimisation, where the size of a given body of water is too small to warrant the cost of simulation, the shallow water simulation will not be run. Once the contour tracing algorithm has identified where bodies of water should be located, and the bodies of water that are too small to simulate have been excluded, the boundaries of each body of water are defined in order to ensure the water reflects correctly once it has been simulated.

Within the generated terrain the outlines that are found by the contour tracing algorithm at a given height can be assumed to the boundaries within which water should be located, as these outlines would naturally form the edges of where the water would meet the land. This data is then used to create the arrays that the simulation will use to store and access information.

Using contour tracing provides the opportunity to improve the performance of the shallow water simulation as it ensure that only the data that is visible will be processed. Alternatives to contour tracing would involve the processing of the area of the entire heightmap, as opposed to the multiple, but much smaller areas produced by contour tracing. Such methods would increase demands on processing power, therefore reducing optimisation opportunities.

After the contour tracing algorithm has located bodies of water, boundaries are introduced to said bodies, which allow the water to interact with the rest of the simulated environment in a rudimentary manner. The boundaries are used to set a size for each body of water, and to determine where water can flow within a scene. The boundaries for each body of water are stored separately, but also contain information relevant to the entire mesh, rather than just a single body of water. This information is utilised when scaling of content is required.

Once a body of water has been created and found to be of a size large enough to be worth simulating the movement of the fluid from cell to cell (advection) within the lattice must be calculated, then this calculation must be applied and the relevant cells updated accordingly.

Advection is indicated by change in height, and change velocity in the X and Y directions within a single cell. As cells are located within a staggered grid, the advection calculation is determined by the edge values along the cells within the staggered grid; this continues to update progressively throughout the body of water. The resulting advection

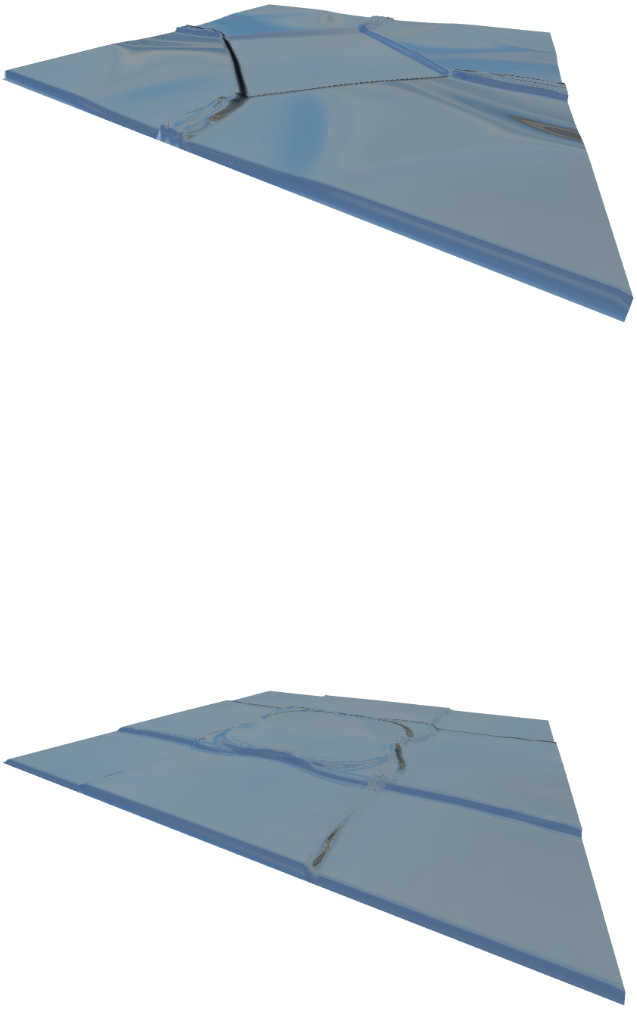


Figure 4.2: The changes of the shallow water simulation as water moves within the container

is then used to apply the relevant updates to each cell, thus creating the visual effect of movement in the water. This entire process occurs at a limited time each second, thus ensuring a seemingly constant flow of water, rather than an unnatural lag or acceleration.

While the shallow water algorithm is determining advection, it is also checking that no boundaries are interacted with. When boundaries are interacted with, the algorithm will ensure the appropriate change in velocity occurs to accurately reflect the effect of the water with that particular boundary. Fundamentally, all boundaries are reflective boundaries of one kind or another. However; where the effect of an absorbing boundary is desirable, the boundary should be set above 0, but below 1 as what is appropriate for the level of absorption desired. A boundary with an absorbing value would be appropriate for simulating an edge such as a shoreline, whereas as a boundary with a reflective value would be better suited for a more sudden edge, like a swimming pool.

The decision of which boundary to use will determine how the water will interact when it hits an obstacle, such as a wall. The simpler and more efficient of the two boundaries is the reflecting boundary. The reflecting boundary works by simply setting the velocity across the edge at a given location on the lattice to 0, this means that when updating the velocity no outside force is applied to the existing value. On the other hand, the absorbing boundary modifies the value of the velocity by reducing the velocity by a set percentage. This is an increased performance costs compared to the reflective boundary as it involves executing additional operations.

Once advection is calculated and any boundaries discovered the updating of the lattice can begin. The first step is updating the height of each cell; this is based on the advection result along the Z axis combined with the change in velocity at that location. Once the height has been updated the new velocity can be computed. This is found by comparing the new height of the cell with its previous height, then accounting for the amount of time within the timestep.

The shallow water simulation offers a solution for achieving accurate water flow due to the Navier-Stokes equation's centrality to the algorithm. However, the shallow water simulation incorporates a simplified version of the Navier-Stokes equation, meaning that the simulation is reasonably computationally efficient for the level of realism that it produces[9]. Nevertheless, simulating large systems, or simulating on lower-end, or mobile devices is likely to be too computationally expensive to run at optimal levels. In such cases, the alternative solution is to seek to recreate the effect of water, rather than attempt to simulate it.

4.3.2 Perlin Noise

Perlin Noise is used to represent the visual effects of movement within water. It is a lightweight algorithm, which makes it an ideal alternative to the shallow water simulation when devices with lower performance capabilities are being used, or where large systems are being rendered.

Perlin Noise works by producing wave-like effects across a mesh for a given number of permutations within a single cycle. This cycle is repeated once all permutations have been completed, thus producing a repeating pattern. In the simplest implementation, the waves produced by Perlin Noise are very evident, but these can be refined by being built upon using octaves, which increase levels of detail across a mesh[19].

Just as the contour tracing algorithm is used to delineate bodies of water prior to running the shallow water simulation, it also does this when Perlin Noise is being used. However, as Perlin Noise does not actually simulate water, instead creating waves and noise which create the appearance of the flow of water, Perlin Noise does not interact with boundaries further than determining the edges of the mesh. This is unlike the shallow water simulation, which uses boundaries to calculate where water should be reflected or absorbed. This means that Perlin Noise has a much lower performance cost as there is an entire set of calculations that do not need to performed within each timestep.

Once the bodies of water have been located Perlin Noise utilises randomly generated vectors for each location on a mesh in order to create convincing variety. When computing a new height for a location on the mesh, these vectors are averaged around the current point, to produce a new vector. The dot product of the new vector and current point vector is then computed to produce the finalised height of the resulting point on the mesh. This process is then repeated across the entire mesh to produce the final resulting heightmap. The resulting mesh has a wave like pattern which can be made more detailed through the use of octaves.

Octaves can be applied to points on the mesh in order to produce finer detail in the final mesh. This process adds additional complexity to the system as each point requires multiple passes to produce the final point. Octaves can be tuned in order to change the amount of detail that is to be added, along with how many octaves are to be applied.

The updating of Perlin Noise is much simpler than that of the shallow water simulation, as no boundaries need to be checked, and no change in velocities calculated. Instead when updating Perlin Noise the height value defined is updated and passed into the Perlin Noise algorithm, which in turn changes the resulting vector and the shape of the mesh. This updating repeats with the height value slowly increasing until the height hits a set threshold and is reset, creating a loop within the Perlin Noise algorithm.

When scaling with Perlin Noise there are two factors to consider: scaling of the resolution; and scaling of the octaves. Scaling the resolution changes the quality of the mesh, as it will either increase or decrease the number of data points being rendered accordingly, but it keeps the overall shape of the rendered mesh. Similarly, adjusting the octaves will either increase or decrease the finer detail within the mesh, producing a smoother or rougher surface respectively; the more octaves produced the more detailed the mesh. However, it is important to note that increasing the number of octaves also increases the computational time required to produce the desired mesh. Both the scaling of resolution and the number of octaves have different effects, and can applied irrespectively of each other. However, to achieve the best result, the resolution should be scaled first, followed by the relevant changes in the number of octaves. Perlin Noise offers a lightweight alternative to the shallow water simulation, as well as providing multiple methods for altering the resulting mesh in terms of quality and performance impact.

The algorithms utilised within this framework allow for optimal compromise between the level of detail of an environment and the computational expense of creating an environment at said level of detail. The diamond-square algorithm has a low ongoing cost whereas the shallow water simulation and Perlin Noise have high ongoing costs. Between the latter algorithms, the shallow water simulation is better suited for higher performing devices where a more accurate level of detail is desirable. Whereas Perlin Noise provides varying levels of detail at lower accuracy, but with a decreased performance cost. The combination of all of these algorithms allows a user to experience an environment regardless of device capabilities, but also allows customisation options for directing the device's resources to a particular aspect of the system.

4.4 Implementation

For the purpose of this framework the generated assets have been split into two core classes; the land class, and the water class. The land class, as its name suggests, is responsible for the implementation of the diamond-square algorithm, which is used to create the mesh representing the contour of the land. The land class also manages the functions required for scaling a generated tile. The water class is slightly more complex as it offers support for two different water implementations - the shallow water simulation, and Perlin Noise.

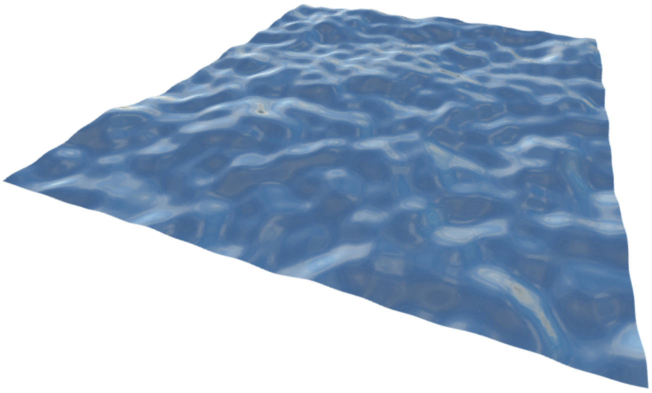


Figure 4.3: Perlin Noise producing basic movement in a scene, to represent water

4.4.1 Land generation

The land class contains four main aspects. These are: the diamond-square algorithm; connecting tiles; scaling; and, contour tracing. The diamond-square algorithm is used to generate the heightmap that creates the base of the scene. Next, the tiles within the grid generated by the diamond-square algorithm must be interconnected to ensure there is a seamless flow between tiles when moving from one to the next. The scaling of the generated heightmap is also handled by the land class, adding further complexity to the class as connecting tiles may have different scaling factors applied to them. The final generative role of the land class is using contour tracing to produce a map wherein various conditions have been met to signify that water should be spawned at that location. Once the heightmap has been generated by the land class it can be processed for rendering using WebGL.

The diamond-square algorithm is implemented as part of the land class allowing it to be generated concurrently as an isolate, or sequentially as part of the land class. The diamond-square algorithm as described previously uses midpoint displacement as a means of creating a heightmap, used for rendering. To implement the diamond-square algorithm first a square mesh must be defined, then the diamond-square algorithm can begin stepping through the mesh to create the desired heightmap.

The heightmap is created during the land class initialisation. This is achieved through the use of a loop which is designed to move through a mesh, subdividing the mesh with each iteration. The first instance of loop covers the entire mesh, identifying the four furthest corners of the mesh, and the centremost point. Thereafter, the loop is constrained to a smaller area based on the halfway points between existing points, thus allowing the mesh to further be refined. Each time the mesh is refined, the variation of height between every point is reduced. This restricts the possibility of extreme height differences within a small areas of the mesh.

Within the loop is a defined variable called `sideLength`; this is used to determine how the diamond and square steps go through the mesh to produce the required heightmap. With each subsequent iteration of the main loop this `sideLength` value is halved. Consequently, with each iteration of the main loop, the diamond and square steps will be performed additional times. The effect of this is that the diamond and square steps will work though the mesh in large blocks which are slowly reduced in size along with the `sideLength` variable.

The diamond step goes through the mesh using X and Y variables to define the location of the mesh which is currently being worked on. The X and Y variables are slowly increased based on the `sideLength`. This can cause some data to be skipped, but as the mesh is slowly refined this data is filled in. The square step is performed after the completion of the diamond step, and is used to assign height values to points in the mesh between existing data points. The square step works in a similar fashion as the diamond step using the `sideLength` variable to search the mesh. The square step focuses on the creation of

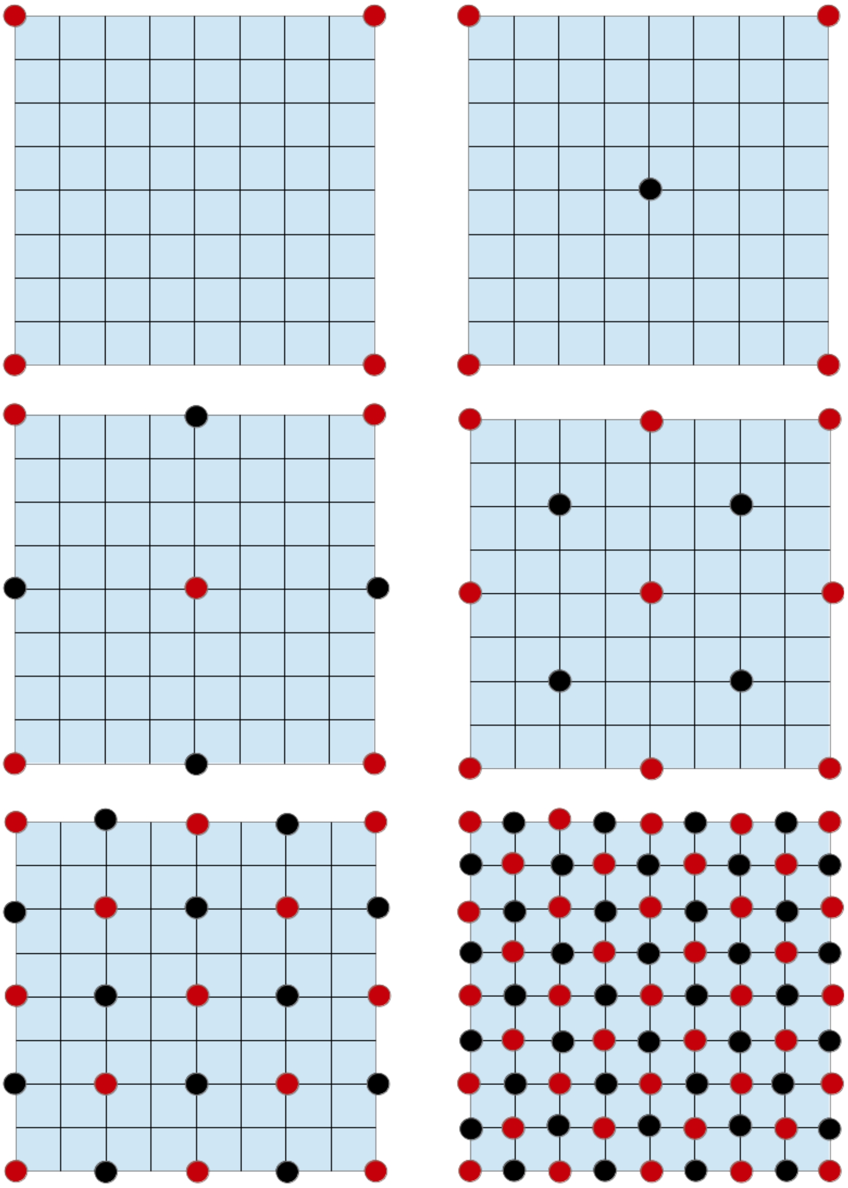


Figure 4.4: The diamond-square algorithm subdividing a mesh, using the diamond step (black) and square step (red).

the corner points within a mesh, which are then used for the base of the diamond step.

Each point visited by the loop is assigned a height value. This value is based on the type of step being performed, that is, either the diamond step, or the square step. In the diamond step this height value is defined by the values of the four corners that are nearest to the point that is being worked on. The average value of the four corners is taken and a random number assigned to the resulting average thus creating the new height value for that particular point. The new value is then placed in the mesh to then be utilised in future steps. This process is repeated across the mesh until the X and Y variables being worked on go outside the range of the mesh, at which point the square step is performed. In the square step the height value assigned to a point is based on the points on the mesh which are a `sideLength` above, below, to the left, and to the right of the point which is being worked on.

The refinement of the mesh is possible as each step bases newly generated data off the previous step, and the area in which each step is performed is also updated based on the previous step.

Once the diamond-square algorithm has created the base heightmap, the tiles within this grid are interlinked to produce a seamless transition when moving between the tiles. When implementing a generated heightmap within the system there are two areas of consideration which must be accounted for. Firstly, the edges of a newly generated tile must align with the existing tiles. Secondly, when a tile within the heightmap is scaled to increase or decrease in quality the connection between the edges of tiles must remain intact.

When it comes to linking tiles of different resolutions it is necessary to set a limitation on how much variance is allowed between two tiles. Limiting the variance between tiles controls how much the resolution can change from one tile to the next thus producing a high quality scene while still taking opportunities to optimise. In this framework the variance limit is set to a multiple of two. This means a tile with a resolution of 64 can be connected to 32 resolution tile on one side, and 128 resolution tile on another side. However, due to the variance limit, a 128 resolution tile cannot connect to a 32 resolution tile. Setting the variance to a multiple of two simplifies the process of connecting tiles within the application. This is supported by the dynamic content controller which evaluates the most appropriate resolution for a tile based on the existing tiles. That is, the dynamic content controller will not generate a 32 resolution tile adjacent to a 128 resolution tile.

A separate algorithm was created to allow for simple terrain stitching, allowing the generation of a tile next to one of a lower or higher resolution. This algorithm first stores a value of where the neighbouring tiles are; that is, the tiles that are above, below, to the left and to the right, or null if a tile will not have a neighbouring tile on one side. Then, during the square step of the diamond-square algorithm, a check is performed to ascertain if the tile currently being worked on is from an edge, and, if so, whether or not it has another tile connected to it. Once a connection has been established the edge value is determined by taking the current resolution of the tile and dividing it by that of the connected tile; the result will be either 2, 1 or 0.5. Once the edge value is found it is used as the multiplier for the square step to use to find the corresponding X or Y value of the new tile relative to the neighbouring tile. This algorithm allows the edges of tiles with different resolutions to be used as if the resolutions were the same.

The terrain stitching algorithm described is a minimal implementation of a much larger and more complex area of research [58]. Other more complex methods perform terrain stitching by dynamically creating new triangles to fill in potential holes, based on the viability of the holes to the user at runtime [59]

The second area of consideration in the linking of tiles is the change in quality of a generated terrain that is caused by scaling up or down as this can lead to a change in performance when creating or rendering the terrain. These changes can be completed

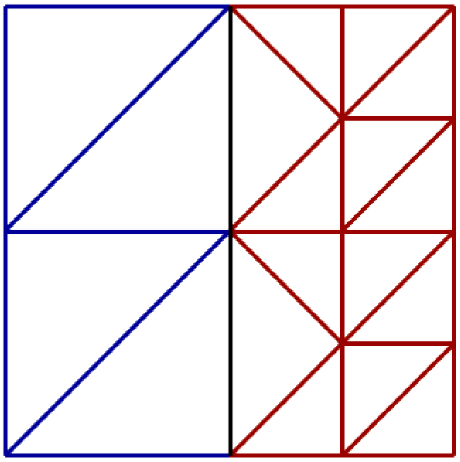


Figure 4.5: The linking of tiles of different resolutions, via terrain stitching

through various functions within the land class. Increasing the resolution of a tile is done by using a single parse of the diamond and square steps of the diamond-square algorithm. The effect produced is similar to as if the tile was originally created at the higher resolution. On the other hand, reduction in resolution is achieved by removing every second value from the list, and every second list from the lattice. Downscaling retains the original form of the terrain, and reduces the size to approximately a quarter of the original size each time an asset is downscaled. The ability to downscale a given tile allows for the configuration of an application to greatly reduce the computational expenses of a generated scene, without significant negative impact on the visual aspect of the scene [71].

An important aspect of optimisation within this framework is the ability to dynamically adapt the detail of a mesh in response to change in various performance factors. A function that upscales a given tile was created to support the framework in providing optimisation in these situations. The purpose of this function is to upscale the detail of a given tile within the grid. The process performed to achieve this is comprised of three parts: locating the correct data for processing and creating a new mesh; placing the processed data into a new tile in order to replace the old one; and reconnecting any edges that may no longer match the neighbouring tiles.

Locating the required data points is a simple task as each point will become the base for the subsequent data point as the diamond-square algorithm will use this data to create the new data points within the tile. As the the diamond-square algorithm will create new data points, it is necessary to first create space between each data point within the regular memory structure to allow for the newly data generated. Once the spacing has been completed a single parse from the diamond-square algorithm populates the empty data points within the mesh. This produces a tile that maintains all the former characteristics of the previous tile, but has more defined features.

Before the tile can be placed within the grid the edges that link two tiles together must be checked to ensure they will connect without any gaps or incongruities. This task is completed by checking for neighbouring tiles, then reading the resolution of the mesh of each neighbouring tile. Once the resolution of each neighbouring tile is found the edge data can be passed to the new tile, thus enabling any necessary adjustments to the vertices of a given edge to occur to ensure the edges of the new tile will match the edges of the neighbouring tiles.

Overall, upscaling by replacing a tile within the grid is a simple task as each tile is stored as its own class. This means each tile can maintain every aspect of itself and from within this class the process to create the rendering is repeated, simply removing the old tile from the grid. The new tile can then be placed within the grid, thus replacing the previous tile. The only complication to this straightforward process is ensuring the connecting of neighbouring edges of the tiles remain intact. However, this is managed by the process mentioned in the previous paragraph.

Where upscaling is mostly a simple process, downscaling is even less complicated as data is being removed as opposed to being added. Downscaling begins in a similar way to upscaling. Initially, a new mesh of a smaller size is created into which the remaining data will be deposited. The relevant data is culled by the deletion of every other data point in a list, and every second list within the mesh structure. This data is then passed into the mesh before the previous mesh is deleted. Once the new mesh is populated the edges must be checked. However, as detail is being removed the neighbouring tiles may need to recompute their edges to maintain the connection and ensure there is no breaking along the seam. This is performed in a similar method to checking the edges when upscaling a tile. Once the resolution of the tile has been changed the `create object` function is called. This function manages the visual aspect of assets and scenes to ensure a smooth flow between tiles within a grid. Without affecting the data produced by the diamond-square algorithm this function rearranges vertices and indices around the edges of tiles to ensure they connect correctly. The mechanics of this function are discussed in further detail later in this chapter.

4.4.2 Contour tracing

As previously stated the land class acts as a base for the water class to allow water simulations to interact with the generated heightmaps in a meaningful way. In order to achieve this contour tracing is used to produce boundaries. The use of boundaries have two advantages: firstly, they allow for interaction with objects; and, secondly, they reduce the amount of data that needs to be processed, as parts of a heightmap will never have water, thus reducing the area in which the water simulation is required to run.

Contour tracing is a well documented and studied process, with a range of applications in computer vision [125]. Contour tracing works by finding a starting point on a lattice, then tracing around the edges to locate a blob within the lattice[12].This allows for set areas within the heightmap to be separated into defined areas based on set criteria; in this case a minimum height is used to define the height at which water is located. Each area has a defined state within a mesh, meaning that interactions with the mesh can easily be simulated. Though this method does require special consideration for identifying islands within an enviroment, this is done by checking if a defined area exists wholly within another defined area.

The algorithm for contour tracing first finds a starting point in a lattice that fits the defined criteria, then faces left and tries to move forward. If the algorithm is unable to move forward, it will then rotate to the right, and try to move forward again. This process repeats until the algorithm can move forward. The algorithm ends once it has returned to the starting location with an additional step to avoid potential branching[12]. Once the contour tracing has finished, the resulting section of the mesh is used to produce a blob. Once a blob is found, and the information about it configured, it is then used as a point at which the water class can be implemented. This is done by initializing the lattice on which the water is based to be of the size found by the contour tracing. The boundaries are also defined by the shape of the body of water, and passed to the water class.

Boundaries allow for interaction to occur between the generated content, in the case of this framework specifically the heightmap generated by the diamond-square algorithm, and one of the water simulations. Boundaries are stored in a 2D mesh that is accessed each time the water class updates. These updates take the boundaries into consideration when updating the water's state to allow for reflection to occur within the generated environment.

The creation of the blobs through contour tracing allows for different areas of a lattice to be labelled as suitable for a water simulation to run in. These identified areas are then grouped within each instance of their corresponding land class. This group is then iterated through to find the size of each blob, and to determine if a found area is suited to host a water simulation, based on its size. Once a blob has been found to be a suitable size, an instance of the water class is passed this blob and uses it for generating its initial state.

Having multiple instances of the water class lends itself to a concurrent solution. Additionally, this method helps improve performance, as, instead of running a water simulation

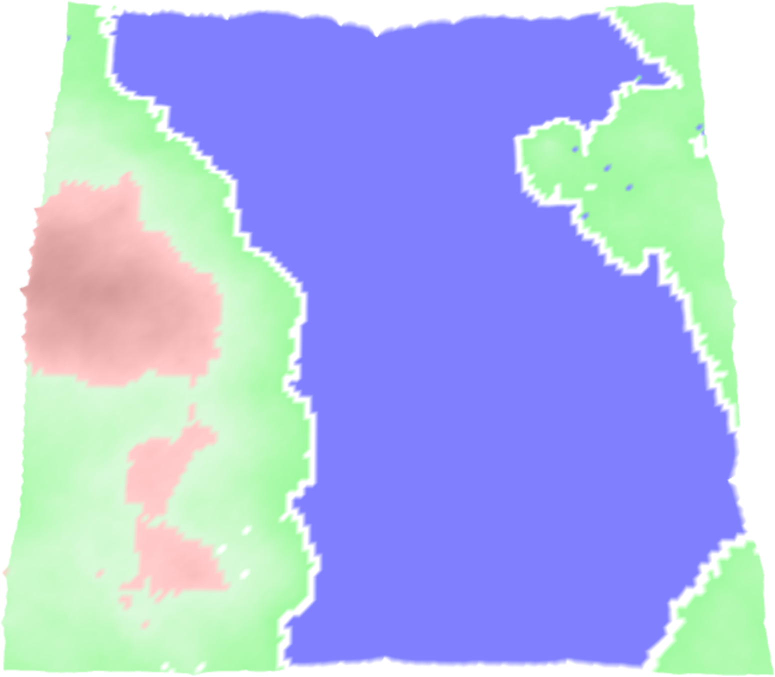


Figure 4.6: Edges produced via contour tracing on a basic terrain generated by the diamond-square algorithm. (Edges are white)

across the entire heightmap, it will only process data within a found blob, effectively reducing the amount of data that needs to be processed. Furthermore, small areas in which a water simulation would have little effect are not simulated at all, creating another aspect by which the application can be optimised.

Some newer graphical frameworks allow for efficiently rendering, and generation of terrain and water, through the use of geometry, and tessellation shaders [84]. Unfortunately these are not supported within WebGL. As this framework aims to support a range of devices through the use of WebGL, this functionality needs to be achieved through the use of CPU bound algorithms.

4.4.3 Rendering

The rendering of the land class is split into three main parts: the generation of indices, then vertices; the use of shaders; and, the draw calls that render the resulting meshes. The indices and vertices created are based on the heightmap produced via the diamond-square algorithm, so when the diamond square is updated part of that process is to also update the generated indices and vertices. The use of shaders, as discussed in Chapter 2, are vital for rendering using WebGL, and are used within the land class to render the scene. The final stage in rendering is the render call itself, which requires special setup with the attribute and uniform pointers, along with ensuring the correct rendering method is applied. Though more modern graphical frameworks do allow for additional methods of producing and rendering land and water, these are not compatible with what is offered in WebGL.

The generation of indices and vertices is based on the generated heightmap's resolution in order to ensure that the rendered scene accurately portrays the generated data. As the heightmap is created on a square mesh, only simple indices that do not create any complex shapes need to be produced. Additionally, the corresponding vertices are also easily generated with each point on the X and Y axes following a linear increase in size, while the Z value, used for height, is defined by the value from the heightmap.

As the mesh is laid out on an X and Y based grid with the height assigned to a Z value, the creation of the indices is as follows. Firstly, a nested for loop is created that loops through the two dimensional array that is used to store the heightmap. For each point on the map the loop creates six values that are used to create two triangles that make up the rendered mesh. For each point on the mesh two triangles are created using the six generated points; these triangles are used to form a square which makes up part of the mesh. The first triangle takes in the current point's X and Y values as the starting point, after which it will take in the point if X is increased by one, and then if Y was increased by one, thus creating the first triangle. The first triangle is used to represent the top and left sides of the rendered square in the mesh, whereas a second triangle will need to be created to render the bottom and right sides of the square. This second triangle is made up using the starting points of X+1 and Y+1, which are the opposite ends of the square to the original X and Y. To create the second triangle the points X+1 and Y, are joined with X and Y+1 to create the last set of triangle indices that are used to create that portion of the mesh. This action of creating the indices repeats for each point on the mesh, up to n-1, as the final edges of the mesh are produced by the immediately preceding iteration.

Each index created is assigned a value that corresponds with a vertex that is made up of three points, the X, Y and Z values, representing a point's location in 3D space. The indices created above link these vertices together for WebGL to use in the shader program to render an object. The creation of the vertices follows a similar pattern to that of the indices, utilising the nested for loop. In this situation the vertices generated can simply be assigned a value based on the current X and Y values from the loop, then altered by the desired resolution. Additionally, the Z value is based on the heightmap previously created by the Diamond-Square algorithm. Each of these vertices are referenced multiple times by the indices, reducing the amount of vertex data that needs to be stored, as a single point in 3D space is only created once, but can be referenced multiple times through the indices.

When generating the indices and vertices for the edges of the heightmap it is important to manage the connecting of edges as described earlier. In order to achieve this, the land class checks the values and resolution of the connecting classes. If the tile is required to scale the edge values this is achieved by editing the method used to produce the indices where X or Y equal 0 or X and Y equal n-1. When creating these indices the pattern changes from generating two triangles for each square to generating three triangles for two squares. This means that the edge vertices will link together correctly, and that the flow can carry on to the higher quality tile smoothly, without adding complications, as can be seen in 4.5. These indices are treated the same as all other indices, and follow the same rendering pipeline after creation.

These generated indices and vertices are then piped into the corresponding index and vertex buffers in WebGL. These buffers are passed to the shader and are used when rendering the scene.

Once the indices and vertices of the generated heightmap have been created and stored as buffers, these buffers can then be accessed and utilised by the shader program to produce the desired fragments. As discussed in Chapter 2, this data is passed to the shader as attributes or uniforms. The fragment is produced using the three indices along with the corresponding vertices. Additionally, in this framework the rendered object is coloured based on a fragment's height, adjusting the RGB and alpha values as needed.

Rendering the heightmap in the land class is completed by having the core application call the render command for each instance of the land class. This function call is accompanied by the view, world and projection matrices, which are used to determine how a scene is to be viewed, and correctly order the tiles within the grid, based on their position. Within the render function WebGL is assigned the compiled shader program to use, and the associated attributes and uniforms are linked. WebGL then enables the vertex buffer, and binds that vertex data. Once bound, the vertex buffer is linked through the `vertexAttribPointer` creating an association between the stored vertex data and WebGL. Additionally, the `vertexAttribPointer` tells WebGL how that data is sorted, and the data type that is used. To keep this framework simple each buffer only contains a single type, avoiding passing data that corresponds to different variables.

Once the buffers have been linked, the final task is to call `DrawElements`; this uses the indices to render the vertices stored in the vertex array. `DrawElements` also needs to know the method to use when linking the indices, as they can be drawn as different primitives. In this framework triangles are used primarily, but this can be changed depending on the style required. The number of indices in the array to render is also required; this is often set to the length of the index array. The act of splitting up the generated content means that WebGL can render a large landscape without being affected by the index limitation.

4.5 Framework implementation

The algorithms utilised in this framework contribute to the design of an application that can create and dynamically scale content to best suit both an application's requirements, and a device's performance capabilities. The algorithms detailed previously have each beenchosen for their scalability. Furthermore, the implementations of each algorithm have also been designed to be scalable. Scalability within this framework is made more accessible and computationally affordable by the introduction of the optional concurrent solution offered in Dart. As discussed previously (chapter 2), concurrency in Dart is achieved by the use of isolates which allow for a single aspect of an application to be split from the core of the application into a new thread that runs in parallel, before returning to the main thread.

Within the landscape, the tile based structure of the mesh helps control both the creating and rendering of generated content. These tiles are given X and Y coordinates corresponding to their location in 3D space, along with a desired quality for the content created for a given tile. Each instance of a given algorithm can be launched as an isolate from a tile, thus introducing concurrency into the system, if required. The result of using a tiled system for controlling the flow of the framework is that it allows for each aspect to be worked on individually, without dramatically affecting the performance of the overall system. Additionally, designing the framework to utilise tiles creates a modular system allowing different aspects to easily be changed or new systems integrated.

As the tile base encourages a certain level of modularity this also means that the framework needs to be protected from concurrency concerns that may occur when content is being generated. This ensures that the content generated can be correctly integrated into a system, without negatively impacting the scene's quality. An example of a key process that needs to be robust is the connection between neighbouring grids in order to ensure a certain level of continuity between each tile. This is so the edges between tiles match, and do not produce tears within generated content.

Another function of the tiles within the grid-based structure is setting up the rendering for each aspect of the content that is controlled by the tile. This means that in this implementation, a given tile must have the relevant data for the vertices, indices, and normals for the heightmap produced by the diamond-square algorithm, and the chosen water simulation. Advantageously, this also aids in overcoming the vertex limitations imposed by WebGL, as the tiles split the scene into a grid. The splitting of the grid reduces the number of vertices that have to be called in a single draw call, instead spreading the data across multiple tiles, and thus multiple draw calls.

This framework can utilise both concurrent and sequential solutions via the tile based system, with the sequential implementation predominantly not requiring the use of isolates. However, due to the computational requirements of some aspects of the content generation, some algorithms need to be tuned to allow the framework to take advantage of isolates even in the sequential implementation.

4.5.1 Isolates

There are three aspects to the creation and use of isolates. Firstly, is the spawning of isolates, which requires a somewhat specialized Dart file, along with the data required by the isolate for initialisation. Secondly, is the opening and setting up of the send and receive ports, which is a relatively simple, yet very important task as the ports control how data enters an isolate, as well as how data is sent between isolates. The third and final process is message passing between isolates, which is based on how the send and receive ports have been set up[40].

Isolates are created by linking an isolate file to the core application, then calling the isolate library to spawn the individual isolates. Each isolate contains a unique memory heap, and therefore cannot share memory with other isolates. Instead, communication between isolates is achieved through message passing using a send and receive port, which is defined when an isolate is created. This means that the part of the application used to spawn an isolate must also check that the isolate has correctly been initialized before it starts sending messages. Although the creation and initialisation of an isolate does take time, the core code can be easily integrated into a class structure to be run sequentially, thus negating the time cost.

The send and receive ports are both defined when the isolate is created; the creation process also involves the initialisation of the receive port. Once the receive port has been initialised, the main application sends a message to the newly created isolate instructing on how to initialise the send port. When the isolate's send port has been initialized, it sends a message back to the main application confirming that initialisation of both send and receive ports is complete.

Within this framework callbacks are utilised by the receive ports for interpreting the message passed to an isolate. The callback reads in the message, splits the content according to the instructions in the first piece of data, then calls the corresponding function. Depending on the message the isolate will call one of several functions then pass the required data (as passed to the isolate by the message) to that function. As messages passed between isolates can contain any data type, this framework stores the data as an array to simplify the splitting of the message. The message passing structure for within and between isolates within this framework is designed to support the generative algorithm that is used for content generation. Multiple arrays can be contained in a single message meaning that if an isolate has been set up accordingly, the isolate can interpret all of the instructions within the message, thus minimising the passing of messages between isolates.

After a message has been received, and the data sent to the required function, the processed data is sent back, in the form of a message, to the isolate that sent the original message. The data is sent back in a similar manner to which it was received to further ensure continuity between all the isolates within a single application. Despite the message effectively being a reply to a previous message it is treated as an entirely new message by the receiving isolate. This restarts the cycle of receiving and sending data between isolates. This method of passing large data within a single message between isolates does introduce an increased performance cost compared to languages that use considerable numbers of smaller pointers.

The use of isolates contributes to minimising the potential for runtime errors caused by race conditions. Isolates avoid creating race conditions as no data is shared between isolates. Isolates can achieve concurrency as when each isolate is spawned it is created on a new thread. This means that on a multicore device numerous components of this framework can be worked on concurrently. This multithreaded approach can improve performance in situations where the data is repeatedly processed as the ongoing performance improvement offsets the time required to spawn and initialise an isolate. This provides another method in which this framework can provide an optimised solution for applications written in Dart.

Figure 4.7 shows this flow of data through an isolate, as the data is generated on the new thread, then passed back to the main thread, in the case of this framework as WebGL buffers. Once recived the main thread will procced to use those buffers to render until the receive port receives an updated set of buffers. This update command can only be issued after the first set of data has been received, as to avoid trying to update non-existant data. Many of the technologies used within this framework also benefit from the increased performance opportunities offered through the use of isolates. While isolates do incur initial performance costs this is, at the very least, offset, if not over compensated for

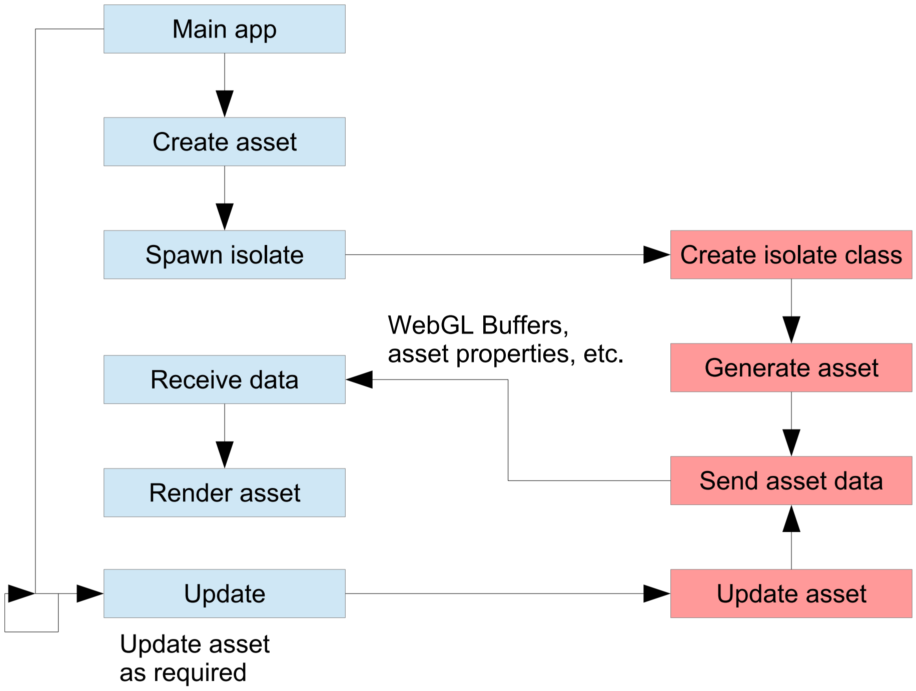


Figure 4.7: The flow of data between two threads using isolates. The main thread is blue; the secondary thread is red.

through ongoing performance improvements.

4.5.2 Water generation

The implementation of generated water uses two different algorithms: the shallow water simulation; and, Perlin Noise. Each of these offer a unique way of representing water, with the largest differences being in performance cost, and the level of detail each one offers. Within this framework both are created and used within the same water class, which defines what water simulation should be run. The water class then allows for the implementation of the shallow water simulation or Perlin Noise to determine whether sequentially or concurrently is the best way to achieve the desired output.

A separate instance of the water class is created and initialised for each blob found by the land class. This blob corresponds to the area where water is to be generated and rendered. Each instance of the water class can be run sequentially as a new instance of the water class, or be run by spawning a water isolate, which allows for the framework to offload some work to other threads.

To implement the shallow water simulation a range of arrays are created and initiated to be the size of the blob for which they are created. The boundaries are defined within an array as booleans. The update cycle is then called, which applies the advection to the system before updating the velocity of a cell in the lattice. To ensure a constant movement within the system an alteration is added that will, in turn, alter the height of the cell at a random location to create a wave; this can then propagate throughout the system. This is completed by finding a location within the body of water and adding water to that location by increasing the height value, while removing water from the surrounding cells equal to the amount added.

The boundaries are defined by the contour tracing performed. That data is then checked to ascertain the size of the area that the list needs to cover. This area measures the maximum and minimum X and Y values. This range is then used to define the size of the lattice. The boundaries are then copied to cover their location on the lattice, allowing for smaller sections of a large landscape to be taken out of the system with each one being worked on independently of one another.

The advection is implemented using an upwind method, based on a staggered grid. This produces the updated advection for the height by taking the change in height, and multiplying that by the velocity in the X and Y directions. The resulting value is then multiplied by the given timestep, and subtracted by the original height. The advection for the velocity works in a similar way where the change in velocity in one direction is then multiplied by the current velocity in the other; or the change in the X velocity is then multiplied by Y. Once again this value is then multiplied by the assigned timestep.

Working out the new height value and velocities in the X and Y axes starts with the updating of the height. This is determined by taking the change in velocity and multiplying that by the advected height value, thus producing the resulting change in height as to how it was affected by the velocity; that is, causing the water to rise or fall. The velocity on the X and Y axes work in a similar way, where the change in height will increase or decrease the velocity; for example, if the height has dropped suddenly then the water at that cell would drop, that in turn would increase the velocity in the X and Y directions of the cell, unless it was at a boundary. in which case the velocity at the boundary would remain set to 0, and the velocity reflected back into the blob, like water hitting then rebounding off a wall.

Boundaries add an additional check to each stage of the update as the cell's neighbours must be checked to ensure that the value being used is not from a boundary, or, if it is, then the default height value, or zero must be used as the velocity. As the flow of water will eventually level off this will in turn produce a static body of water, in which case using the shallow water simulation would become pointless. To ensure constant movement within the system an additional process is added at a set interval. This process will then find a random location within the body of water and add a small increase in height to that location and reduce the same amount from another location. This way the amount of water in the system remains constant, and a constant movement of water is present. If water was added without the same amount removed, the water would continue to flow, but a constant increase of water would occur, causing an area to become covered with water where it otherwise should not be.

The Perlin Noise implementation is slightly more straightforward than the shallow water simulation, as effects such as reflection from boundaries do not need to be accounted for. The Perlin Noise implementation is made up of three main parts. The first of these is the Perlin calculator class, which handles the calculations involved in producing Perlin Noise. Following this is the update step, which handles how data is changed across the lattices. The final part of the Perlin Noise implementation is scaling, how it is used within the system, and how it can be used to control the system.

Perlin Noise is implemented in a class structure that can be taken advantage of within an isolate. This implementation is similar to the shallow water simulation, the design of which aids in allowing the water class to easily utilise either method for generating water. The water class is initialized with a mesh based on the blobs from a generated heightmap, and the level of detail required. This mesh is used as a guide for the size of the Perlin Noise, while the level of detail required is used to dictate the octaves and resolution of the created system.

When the Perlin water class is created, it can use the Perlin Noise calculator library designed for this framework. This calculator contains all the functions required to utilise Perlin Noise, and to produce the required data for a mesh to be generated. The Perlin calculator encompasses two mains roles within the system: storing the permutations required by Perlin Noise; and containing the functions required to utilise Perlin Noise within a system.

Pseudo-random vectors are used in Perlin Noise in order to create a random mesh. These vectors are stored in a list as permutations. Having these vectors pregenerated means that the Perlin Noise will maintain the same shape each time, and will save in processing resources as this data does not need to be regenerated each time the application is run. This is a randomly sorted array containing all numbers between 0 and 255. These numbers are also repeated and stored in a new list. This means that as the list is progressed through, Perlin Noise will naturally repeat itself.

The next part of the Perlin calculator library is that all functions required for the Perlin Noise are stored within one location. Using this library means that all that is required when wanting to use Perlin Noise is to call the functions `PerlinNoise` or `PerlinOctaveNoise` with the latter offering greater detail. `PerlinOctaveNoise` uses the `PerlinNoise` function to generate the required noise, but adds to the generated noise for each octave required, with the octaves having a reduced effect each time.

The Perlin Noise implementation takes in the X, Y, and Z coordinates of a point in 3D space, and returns a resulting value, which, within this framework, is used for the new height value. Once the `PerlinNoise` function receives the required values, the values are clamped periodically between 0-255, in order to avoid overflow within the system. A fade curve is then produced for each passed in value; this is designed to smooth the final output. Once the fade curve has been produced, the hash coordinates are generated using the supplied X, Y, and Z values. These hash values are used to obtain the corner points of the supplied point, which are used to obtain the final vector. This hash value is passed to the `grad` function, which is designed to calculate the dot product between a pseudo-random vector and the passed in vector. This is applied from all corner points to the original vector. The result of the `grad` functions are then interpolated between each other to produce an averaged vector, which is smoothed using the values generated by the `fade` function called earlier. This final fade value is then used as the new height value for a given location on the lattice.

In order to update the state of Perlin Noise, a change in the vector passed must occur. Within this implementation of Perlin Noise this change is introduced through the adjustment in the height variable passed into the Perlin calculator for each point on the lattice. The adjustment of the height within the lattice changes the initial vector, thus leading to a change in the vector returned. In order to ensure that when Perlin Noise repeats it does so smoothly, the change variable used is reset to 0 once it reaches 256; this results in limiting the maximum height assigned via Perlin Noise.

This implementation of Perlin Noise utilises two different methods of achieving scalability. One of these is changing the resolution, which affects the quality of the mesh generated. Alternatively, octaves allow for finer detail to be added to the mesh, without affecting the resolution. Changes in both resolution and octaves are used within Perlin

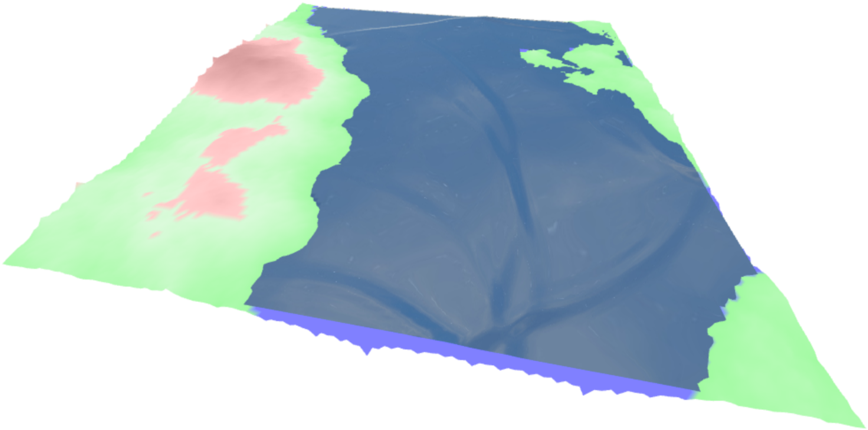


Figure 4.8: The shallow water simulation, based on defined contours, interacting with edges as boundaries causing waves to ripple when interacting with terrain edges.

Noise via the `PerlinOctaveNoise` function.

In order to scale the resolution of a mesh, the points passed to the `PerlinOctaveNoise` function are scaled to be between 0 and 1. Scaling all points to between 0 and 1 means that when the resolution changes the points' requests remain constant. This means there is only a change in the number fragments that make up the mesh generated. Using this method allows for a lattice to be generated, and have the resolution changed dynamically, but still maintain the original form, regardless of the resolution required.

Scaling using octaves utilises a repeat in Perlin Noise to create scalability through calling a single function multiple times, with each iteration resulting in a finer detail. This is achieved through the use of a loop that adds the effect of each iteration of the Perlin Noise together, where each iteration has either an increasing or decreasing effect on the system. This loop will continue based on the number of octaves required, with the effect being determined by a persistence variable. The use of octaves does not affect the number of fragments generated, but instead increases the detail producing finer quality fragments in areas where smaller details are easily noticed.

4.6 Summary

The structure of Dart allows for a very controlled flow of information through an application by way of classes, and futures. While class structures are not new they have been improved upon in the web with the introduction of Dart [122]. Futures are also a new aspect to web programming that can be taken advantage of with Dart. These two features allow for more complex processing of information while data is being loaded, or worked upon. With Dart various techniques are easily applied to the system to help control how the scene is rendered and created. These range from the way an object is created, to the order it is created in, and what it is based on. These techniques can easily be added to the system through various constructs.

The simulations used to generated a scene within this application are based on the use of each aspect being controlled by one class, which may be based off multiple instances of the same class, or the creation of a new class, leading to the next step within the system. By using classes and instances of classes, class constructors can be implemented and used to aid in the creation of a class and simulation. As the creation of some simulations is based on the work of another, such as the water being based on the landscape, a form of linking is required to ensure the correct data is used. In this situation this is handled by the contour tracing algorithm, which the land class uses to then create the water simulation. This then creates a new instance of the water class, which is handled independently from the original land class.

These simulations are enhanced by level of detail methods, but with a focus on generating content at the desired quality, and scaling up. Rather than more traditional approaches in which a high quality model is scaled down. Additionally the simulations used in this framework, are largely in part examples used to show what is possible in this framework.While other methods were explored, Diamond Square, Perlin noise and Shallow Water were found to be sufficient for this framework and research.

Chapter 5

Dynamic Content Controller

The dynamic content controller is used within this framework to monitor and direct how generative content is produced and maintained, in relation to performance requirements. Frameworks that have been designed for platform independence can be improved through the use of this dynamic content controller as it allows for an application to scale based on a given device's performance capabilities. The dynamic content controller is implemented as the base of the designed framework, managing when and how content is created, along with how it is modified and/or removed during runtime. It controls the performance requirements of rendering a scene by modifying generated assets during initialisation or runtime. The flexibility afforded by the inclusion of the dynamic content controller provides developers with another tool by which they can achieve platform independence. runtime

5.1 Maintaining Optimised System State

Initially, the dynamic content controller (asset controller) measures the time taken for the first tile within the grid to be rendered. It uses these measurements to estimate the cost of creating an ideal base system state. If the estimated cost means that content can be generated and updated at around 60 fps the base system will be created. However, if it will take less time, the asset quality will be improved and/or the number of tiles increased before the base system is created. Conversely, if the estimated cost is too high, either the asset quality will be reduced, or the number of tiles decreased.

The asset controller continues to track each generated asset within a 3D scene, modifying the asset as needed to ensure the best performance for a given device. This is possible as the asset controller is continuously monitoring the performance impact of generating and updating assets. The asset controller can modify performance output through the use of generative algorithms, which are designed to be scalable.

As this research is concerned with the optimisation of platform independent graphical applications through the use of generative content, the asset controller must focus on the visual impact of each asset generated, in order to minimise the effect on the rendered scene. This can be achieved by ensuring a smooth flow within a scene, which, in turn, is achieved by making the quality of neighbouring assets the same, then changing the quality of the assets as they move closer, or come more into focus. Conversely, when the user moves away from an object the quality is reduced in order to free up resources on a device. This method means that as the user navigates around a generated scene the assets change in quality accordingly to if the user moves towards or away from them. This leads enables the device's performance to be distributed with preferential focus on where the user is, as this has the larger impact on perceived scene quality.

As the generative algorithms (Diamond-Square algorithm, Shallow water simulation, Perlin Noise) focus on the procedural creation of assets, the created assets are used as the base of the graphical application comprising the landscape and other objects which make up environmental details. Consequently, these created assets are used mainly for simpler objects and often lack finer detail when compared to custom-made models. This means that while the controller can modify larger generated assets, the smaller and more finely detailed objects remain outside of the controller's scope.

To allow the user control over prioritising between the quality of a generated object, and the number of objects generated, this framework implements metrics to allow a compromise to emerge to ensure a minimum level of performance. These metrics allow for an application to focus on the generation of a large number of assets at lower quality, or fewer assets, at a higher quality. For example, an application might require a large area to be created, but does not require a high level of quality on assets generated. In this situation the metric applied would increase the number of tiles drawn, with a reduction in quality in order to achieve a balanced level of performance. Alternatively, an application might require a high level of detail, but only a few assets generated. In which case the metric would ensure the generation of assets at a higher quality, while producing them in smaller quantities. These metrics are tools made available to the developer for producing the desired outcome of a rendering across devices while still preserving performance requirements.

In addition to introducing metrics to manage the rendering quality, and quantity of generated objects, there is another trade off available. This is between an application's overall rendering performance and the application's level of responsiveness. This means that the framework can be used to support an application that requires limited user input, or user input can be delayed, in order to improve performance. On the other hand, where an application may need to be extremely responsive the rendering properties can be reduced. This builds upon the previously mentioned metrics in order to achieve a greater level of control over a system.

The utilisation of the asset controller allows for a very significant reduction in development time for the number of devices that an application may be run on. This is due to the nature of the procedurally generated assets being easily reproducible, as well as being scaled to fit a set device's specifications, with the assets being scaled across devices. This means that a developer only needs to have created a scaling algorithm once to produce

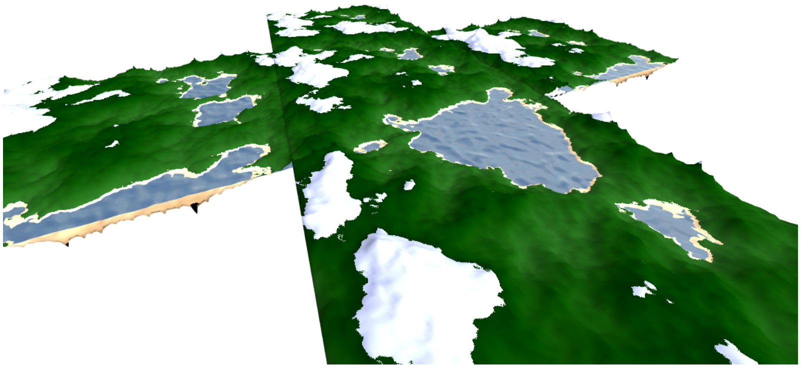


Figure 5.1: This image shows a scene based on setting the developer metrics to favour high quality over number of tiles.

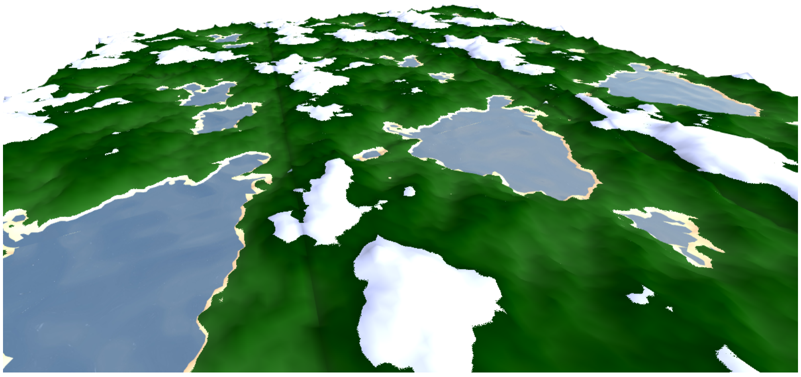


Figure 5.2: This scene is lower in resolution than the scene depicted above, but is compromised of more tiles. This reflect developer metrics that favour coverage over resolution.

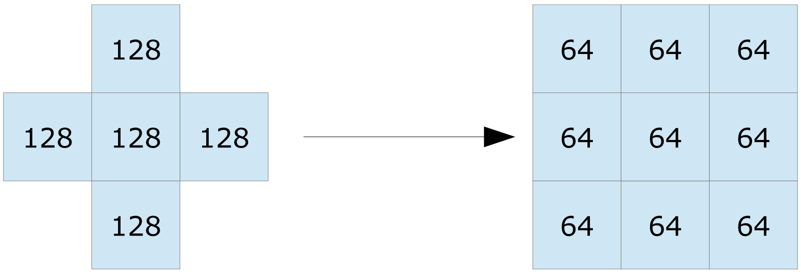


Figure 5.3: This diagram shows the change in tiles resolution within the scene in 5.2.

content across a range of devices, rather than reproducing the same content multiple times at different qualities.

The asset controller utilises a base system state that is then built upon through the generation of various assets. The system state can be manipulated via the defined metrics in order to achieve the desired balance between system settings and simulation quality. The state can be altered further based on a device's performance, and the application's requirements. These changes are implemented through each instance of an asset's class, which are stored as part of the asset controller. The scaling of assets, and the manner in which the modifications are applied, are also managed by the asset controller. This ensures there is a smooth transition as the assets change in quality, thus minimising impact on the user's experience.

This asset controller is responsible for the creation of the base system state. This base state is determined by some simple performance tests as the application is initialized. The result of these tests give a rough indication of the device's performance outputs, which are, in turn, used to determine the best initial state of an application. These initial tests are based on metrics assigned by the developer, in order to mimic the desired runtime results. These performance tests account for the time required to create an asset's initial state, then extend upon this data to determine how many update cycles need to occur in order to achieve an optimal frame rate (usually 60 frames per second). Once the tests have been completed the controller can produce an estimated ideal runtime environment.

This environment is constantly monitored and altered by the asset controller, in order to achieve the desired result based on the defined metrics.

Much of the asset controller's functionality is based on the defined metrics. This allows the controller to manage and prioritise the manipulation of a scene when a user interacts with the system. These metrics allow the controller to change the generated assets' quality, or alter the number of assets created according to the user's specifications. Such changes affect the runtime performance of the application, which, in turn, means that the changes executed by the asset controller can influence how smoothly an application runs. In addition to the metrics that manage the quality of assets, a variable is included that prioritises dedicating performance power specifically to the rendering functions of the application over other tasks within the application. A situation when these metrics might be utilised would be to prioritise the level of realism in bodies of water. In this case the asset controller would use the shallow water simulation to render the water, instead of Perlin noise.

Each asset is created and maintained by an instance of that asset's class. This class determines how to create, scale or remove each instance of itself from the scene. The asset controller utilises each class's built in functionality to adjust the generated content. This means that the asset controller only needs to send function calls to an asset's class if needed, and allows for an individual class to have control over the assets. This minimises the required input from the asset controller, thus reducing overhead from the application. Furthermore, the location of each asset in 3D space is stored within the asset controller, which simplifies the scaling of assets based on proximity to the user as the asset controller has quick access to the coordinates of each asset. Additionally, the storing of 3D coordinates in the asset controller when generating new assets simplifies the linking of neighbouring tiles together as each instance of a given object's locations are already known.

As the asset controller alters the application, it also tracks the effects of these changes in order to ensure that the subsequent changes in performance requirement were appropriate, and that the current runtime performance is adequate. This is ascertained by checking the new timings of assets during updating cycles. If it is found that the new timings are too high or too low, the assets are adjusted accordingly to bring the performance closer to what is deemed desirable.

When the asset controller is updating an object it calls the instance of that object's class to replace the object by generating a new one, initially using the data from the previous object. This means that the updated object will remain mostly the same as the previous object when scaling, gaining or losing only minor details. Additionally, when going through an update cycle, the asset controller will produce new objects at the furthermost distance from the user (as determined by metrics and runtime statistics); the objects will then be scaled up as the user approaches. This process allows for the objects to be created at a lower quality, thus minimising the performance impact on the application.

5.2 Implementation

The asset controller is made up of three key parts; the benchmarking, which occurs when the application is started; the creation of each asset based on the previous benchmarks, which may include the creation of isolates; and the process for updating and maintaining an asset. The assets themselves can be created as an implementation of the asset's class and run on the main thread, or can be introduced to the system using an isolate implementation, which allows for multithreading to be used, thus providing an opportunity for improving performance.

5.2.1 Benchmarks

Benchmarks are used throughout the framework in order to tune performance requirements of an application. The benchmarks are first run when an application is started up to determine a base system state for assets to be created at. The benchmarks are then periodically run throughout the application's runtime in order to fine tune performance to best suit the current requirements.

The first benchmark, which is run when the application starts, is based on timing how long the application takes to generate a single tile at the maximum resolution. If the time taken to generate the tile exceeds the ideal (60 fps [51]) then the tile resolution is reduced until the tile can be generated in the ideal time. An additional benchmark is run on the creation of a single isolate to determine the performance offering of a multithreaded approach on the system. The interpretation of the results are used to create the application's initial configuration.

The isolate benchmark returns two key values: the time taken to create the isolate's VM; and the time taken to pass data between the main application and isolate. Both values are of equal importance as the time taken to create and initialise an isolate is predominantly performed on the application's main thread, meaning that creating new assets within an application could impact how smoothly the visual component of the application looks and runs. The second aspect of the second benchmark in regards to the time taken to pass data becomes increasingly significant the larger the scene is that is being produced, as currently all render calls in WebGL are required to be performed from the main thread, further increasing the thread's workload.

Throughout the application's life different events might occur on a device that impact its performance. This means that periodically the framework must self-regulate priorities within the system to better suit the application's requirements at runtime. These requirements are once again based on the idea of maintaining a frame rate of approximately 60 fps, in addition to adhering to the defined metrics when managing the creation or manipulation of a generated asset. In this situation an ongoing process occurs that monitors the updating of various assets, and compares this to the distance from the user's view, and checks the results against the metrics as to whether the asset needs to be altered in one way or another. This results in the framework checking the impact and relevance of each asset throughout runtime, and adjusting them as required.

Listing 5.1: Dart code for Initialising the asset controller, using the results of the benchmarks to determine how to set up the scene.

```
InitialiseAssetController(){
  Bool usingIsolates = false;

  let seqResults = this.beginBenchmark(false);
  let conResults = this.beginBenchmark(true);

  if (seqResults.init > conResults.init || seqResults.update >
      conResults.update) {
     usingIsolates = true;
  }

  let DS = new DiamondSquare(location, details, usingIsolates);
  let water = new WaterSim(DS, usingIsolates);

}

// Use just the algorithms to generate data, ignoring rendering.
// Using the algorithms default values, but still setting true or false to the
    usage of isolates
```

```
beginBenchmark(concurrent) {
   let st = window.performance.now();
   createHeightMap(concurrent);
   createPerlinNoise(concurrent);

   let t1 = window.performance.now();
   updatePerlinNoise(concurrent);

   let t2 = window.performance.now();
   // Return a new object
   return {
      init: t1 - st,
      update: t2 - t1
   }
}
```

5.2.2 Asset creation - Isolates

Isolates introduce a system for concurrency within Dart making available the option to offload tasks that require processing to new threads within a new VM. This framework can take advantage of isolates as required to improve performance and split up tasks from various algorithms, such as the dividing of the mesh in the Diamond-Square algorithm. It is worth noting that WebGL requires all draw calls to come from a single thread, meaning that the processing performed within an isolate does need to be returned to the main thread in order to be rendered[54].

The requirements of creating assets using isolates vary slightly from the typical method of creating assets as the asset class needs to be modified in order to send and receive data between isolates as required. This means that within an asset's class, data from the benchmarks must be used to determine how much of the work, if any, should be processed using isolates.

Where sequential implementations occur within the same VM, each new isolate used in concurrent implementations runs within its own new VM. This requires send and receive messages to be passed between the isolates. The design of this framework enables these messages to include data relating to scaling an asset up or down, or changing an asset's state in order to better suit the scene's requirements.

Where the asset class determines that performance will be improved by using concurrency the asset class will launch a new isolate containing the algorithm that requires processing. This is the same algorithm that would typically be used, and the same function will also be called. The key difference is that the results from the typical processing method will be returning sequentially, whereas the concurrent implementation will return the data using send and receive methods, as shown in figures 5.5, 5.4.

Once the data has been processed and the asset class has retrieved it the rendering pipeline can be updated to represent the changes within the scene. The advantage of using isolates is that the framework can utilise multi-core processors. This means that multiple tiles worth of data can be split up and worked on concurrently, thus improving the system's overall performance[43]. For smaller systems, the initial time taken to create an isolate may result in it being quicker to produce some aspects sequentially. However, it may still be more efficient for more complex tasks to be processed using isolates.

5.2.3 Creation

The asset controller can utilize the results from the benchmarks to create a base system state for assets to be generated. The controller works by interpreting the benchmark results and based on the device's performance, and the developer metrics, producing the appropriate base system for an application. This base system state is designed to determine

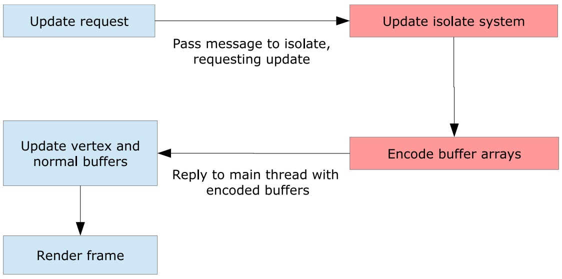

Update request
Pass message to isolate, requesting update
Update isolate system
Encode buffer arrays
Reply to main thread with encoded buffers
Update vertex and normal buffers
Render frame

Figure 5.4: The steps taken to update an asset concurrently.

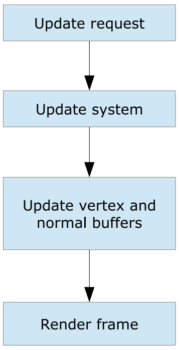


Figure 5.5: The steps taken to update an asset sequentially.

whether implementing isolates will be a viable option, in addition to producing default values to use when creating the various assets.

The base state is designed to allow an application using this framework to run at an ideal level based on the device, and the developer's desires (as expressed through the metrics). The aim is normally to achieve a screen refresh rate of 60 fps, while also maintaining a smooth flow as the scene changes, and assets being changed with it. This can be achieved through a number of ways, such as switching the method of producing water, or changing the size of the overall system that is to be generated and displayed. These benchmarks are also used to determine the possible areas wherein it may be advantageous to utilise isolates within Dart.

When creating this base system state the asset controller will also determine how each tile is to be displayed. This will include information about the tile, such as the resolution of the landscape to be generated, or what kind of water is to be rendered. Depending on the influence of the developer metrics, the framework will focus on generating high quality tiles in more central locations, as these will be more noticeable to the user, while gradually lowering the quality of the tiles as the user moves away from the central location. On the other hand, in some situations the metrics will require a larger area to be produced with minimal regard for the quality. For example, in a flight simulator it is ideal to be able to view a large distance around the user; however, as the user is so far away from the landscape, they would not expect a high level of detail.

The asset controller also evaluates where a performance increase may be offered by isolates when creating the base system state. In some cases, where an asset is created and not often modified, a sequential implementation in which the asset is created and maintained within a class may offer the best performance. On the other hand, assets that are in a constant state of change may benefit from Dart's isolates, as the initial cost of creating the isolate would quickly be outweighed by performance improvements offered by the isolates.

The resulting state the scene is initialized to may not be perfect, as the benchmarks that were run may not have been completely accurate. To compensate for this the framework's asset controller is run throughout the life cycle of the application in order to alter generated assets to maintain the desired level of performance within the scene. In addition to this, when assets are created or destroyed the asset controller can alter the runtime performance in order to improve performance.

5.2.4 Update Cycle

Now that the scene has been created with assets being generated to an initial state, it is ready to be rendered and interacted with. As the scene continues to exist, the various assets generated may need to be updated to reflect changes made by the user, such as moving around the scene, thus changing how an object is viewed. The resulting changes must be managed in order to control how the asset is updated to ensure minimal effect on the user's interaction within the rendered environment. Firstly, the order in which the tiles within the scene (and, by extension, their respective assets) should be modified is determined. This ensures the changes occur smoothly and minimises the noticeable effects of the change on the assets. Next, the asset controller is constantly updating to ensure it is receiving the most up to date data on what is occurring within the environment. Finally, once it has been determined that assets must be changed, and what that change is, the asset is updated.

5.2.5 Asset configuration

A key concern of updating assets is minimising how noticeable the change will be to the user. To reduce the obviousness of updating an asset the process is split into three main parts. Firstly, the asset controller prioritises updating the assets that are closest to the user, starting with the furthest away tile as per the developer metrics. Then, during the updating process, the asset controller reviews how much of a change in performance occurred. Based on the performance changes within the system, further adjustment to the assets, and the respective tiles on the grid, are made. Once the assets have been adjusted the system as a whole will then add or remove new tiles.

To minimise the user noticing the visual impact of changing assets, the assets are modified in order of proximity to the user, starting with those that are the furthest away, but still close enough to have been rendered. This method ensures that the smallest, or least detailed assets are modified first, reducing the obviousness of future changes, as well as lessening performance impact on the scene.

This framework applies a developer metric to determine the highest quality possible for the user's range of view. This metric ensure that the tiles within immediate proximity to the user are maintained at the set levels of detail. This means that as the user moves through the scene, upcoming assets will have already been scaled up to the set level of detail, whereas the furthermost assets in the opposite direction will have been scaled down in order to maintain balanced performance. Performance is further preserved by the asset controller modifying existing assets, rather than creating new assets, which would have increased performance requirements as based on the asset.

In order to preserve the desired level of performance, all changes that are the result of the asset controller updating a scene must be monitored. The monitoring occurs over a defined period of time, after which, changes in performance are recorded, and resulting changes to asset generation applied, if necessary. An example of a performance change significant enough to require subsequent change in the asset generation process would be if the frame rate increases or decreases by more than 15% the quality of generated assets would likewise be increased or decreased to return the frame rate back to the desired rate. However, if the frame rate fluctuation falls within the allowed variation, no changes to asset generation are required. Allowing for a certain amount of variation from the desired level of performance means that the asset controller is not constantly altering the scene on occasions where the resulting changes would have minimal impact. Once any necessary adjustments have been made to the asset controller, the monitoring process is repeated.

Another way in which updates to the system can affect performance is that throughout the life cycle of the application it may have been necessary to add or remove tiles in response to the application's performance requirements, or the user's movement. The creation of a new tile, and its respective assets is one of the larger generative tasks. This is due to the performance requirements in potentially creating new isolate, along with the new classes that need to be created and initialised. The performance impact of creating a new tile is minimised as, after the initial landscape has been generated, all newly created tiles are added to the outer edges of the grid. This allows for all newly created tiles to be generated at the lowest scale, thus minimising their impact on the scene when generated. On the other hand, the removal of a tile is a much simpler, less expensive task in terms of performance as the assets are being destroyed, not generated. Furthermore, the removal of tiles frees up additional device resources, which can also positively impact performance capabilities.

The addition of the asset controller within this framework manages the updating of assets within a scene with three key methods. Firstly, the asset controller manages the generating and updating of content proactively so that assets are altered before the changes are required and the changes are not too obvious to the user. This is achieved through updating assets from the outer edges of the user's view inward. Secondly, while the asset controller is updating the scene, ongoing performance changes are also being monitored, with the appropriate changes in asset generation being applied to maintain balanced performance output. Finally, the asset controller will add or remove tiles in order to balance performance, or in response to user movement, if required. These tasks, and any changes made by the asset controller subsequent to required changes in performance, are based on the metrics that are predefined by the developer.

5.3 Developer Metrics

As mentioned above the developer metrics are vital for ensuring that the asset controller accurately maintains the balance between different aspects of the system, such as the resolution of tiles to how the water is simulated. In this framework the balance is defined as a compromise between the quality of a rendered tile, and quantity of tiles rendered. The metrics within this framework have been designed to be easily modified to allow for other metrics options to be implemented as they become relevant.

The metrics are defined by the developer when an application is created using this framework, with each metric assigned a decimal number between 0-1. The assigned value contributes to the asset controller determining when a new tile should be produced, or if an existing should be upgraded instead. The asset controller also utilises the results from ongoing performance benchmarks to determine how best to make changes to a scene. For example, figure x shows a metric bias towards improving existing tiles over the production of new tiles. However, as the necessity for producing new tiles increases, this will eventually become more important within the asset controller algorithm, thus a new tile will be produced. The metrics aspect of this framework allows the developer to weight the importance of how the system can best be displayed against their original design goals. In turn, this means that a scene can be viewed and explored as the developer intended, but within in the bounds of the framework to ensure the best performance output possible on a given device.

As the complexity for controlling a scene grows, so does the need to further how the metrics can be applied. This means that the framework can be modified in order to accommodate a metric that balances different parts of the system. An example of this would be if a developer desired high quality simulations for improved realism within in a scene at the expense of reducing the overall size of the scene, or another aspect such as resolution of a given tile. In the current implementation of the framework the asset controller's main ability to balance different aspects of a scene is through the updating of generated assets, which allows for the processing requirements of an application to be rebalanced as required.

5.3.1 Updating of asset's

When the asset controller calls for a generated asset to be updated there are four events that occur. Firstly, the asset itself will scale up or down in quality as required. This is this followed by a checking of the edges of the tile or asset in order to ensure consistency within the scene. Next, the WebGL buffers used for rendering (vertex, normals, indices) are updated to reflect the new changes in the data set. Finally, the previous data is removed from the application, freeing up device resources.

When updating an asset, new data, which represents the asset, is created. These updates consist of generating new data points based on the existing asset, thus producing a higher or lower quality asset as deemed necessary by the asset controller. As previously discussed in Chapter 4, these updates slightly change the asset but are designed to allow for the object to keep its overall shape. This means that the asset within a tile, such as the landscape itself, must maintain its connections with other tiles within the grid. The framework can achieve this via two distinct methods. The first, and simplest, method is to ensure that when an asset is updated, every edge of each tile is assigned the same value, and that these values cannot be altered. This produces a single base level for all edges of each tile, and is the cheapest solution in terms of performance. The other method is, after a tile has changed, to rejoin all the edges around itself in order to match the surrounding tiles and produce a smooth link between tiles. This is achieved by the issuing new values to each edge of the tile. This method does increase performance requirements; however it also ensures that there is a continuous flow of the landscapes between tiles, without producing noticeable changes within the scene.

Once the asset has been updated the resulting changes need to be reflected within the rendered scene. This process is very similar to that of updating the water asset when the water moves through the scene. However, when changing static assets indices must also be updated. This is unlike the updating of water as the size of the body water does not change, meaning only the height values of some vertices need updating. Static assets, on the other hand, will change in resolution, thus requiring the indices to be updated. Thus the process to update an asset that has changed resolutions requires a slightly larger overhaul, in order to recreate the indices. New arrays containing the updated data points are produced, which are then passed to the WebGL buffers. These arrays contain the new indices, vertices, and normals that are required when rendering the scene. The arrays are then bound to their respective WebGL buffers, replacing the older set of information that was used for rendering the asset. The changes that have been made can then be immediately reflected in the next draw call.

Once the asset has been updated, and the new WebGL buffers created, the next priority is to remove any unnecessary data related to the asset, or that may be leftover from performing modifications to the asset. Additionally, it is necessary to remove links between older data and data which is still in use, as this will allow Dart's garbage collector to free up data.

5.3.2 Creation of new asset's

As the user moves throughout the scene, different assets will adjust in quality to ensure that the user is presented with assets rendered at the highest quality that the device can support. Inevitably, the user will move though the scene to a point where new tiles must be generated in order to ensure that there is further content for the user. The creation and integration of new tiles within the grid and scene at runtime within this framework can be managed by the asset controller in order to achieve smooth content generation at minimal performance cost. The creation of a new tile within this framework is implemented in three stages. Firstly, the location of where the tile is to be generated is identified. Then, the base assets are produced and passed to the scene. The final step is to pass these new classes to the asset controller so it can scale and adjust each asset within the scene as required.

The identification of where to create a new tile is based on the device's performance capabilities, and the developer metrics. These indicate to the asset controller when and where new tiles need to be produced. These locations are places on the grid that will be coming into a defined range of the user's view, therefore there must be content available for display within this range. As a new range of tiles come into the user's range of view and are consequently produced, a range in the opposite direction will fall outside the required area and will consequently be removed from the scene. Once the location at which a tile must be created is found, the framework will start generating the base asset, in this case the land class, using the diamond square algorithm. Land will be created at a minimum resolution, so as to reduce the effect on the performance of the system. Once the tiles have been created they must be connected to the edges of existing tiles to ensure a smooth transition between tiles.

Once the base of the tile has been created, further assets can be generated and added to the scene. The first asset, water, is added as an instance of the water class. Firstly, a contour trace is performance on the landscape to ascertain where water will be placed (if at all), after which the water simulation will be run. This water simulation will be performed with either Perlin noise, or the shallow water simulation, depending on the quality required for the scene.

Once each of these classes have been created and initialized, they can be passed to the asset controller. The asset controller then places these assets within the grid, where it can manage how the assets operate. This means that the asset controller has access to the data that indicates the location of each asset within the grid, thereby allowing the asset controller to also access data on what assets are located within a given tile. Storing assets within a tile allows the asset controller to issue function calls to a single asset, or to multiple assets located within an entire tile. This allows for large scaling calls to be applied to all assets within a given tile, or for smaller scaling calls to adjust a single asset within the tile, such as changes that might be applied during the maintenance cycle of a water algorithm.

The functionality of being able to add tiles throughout runtime is a significant advantage within the asset controller, and to the framework as a whole, as it allows a generated

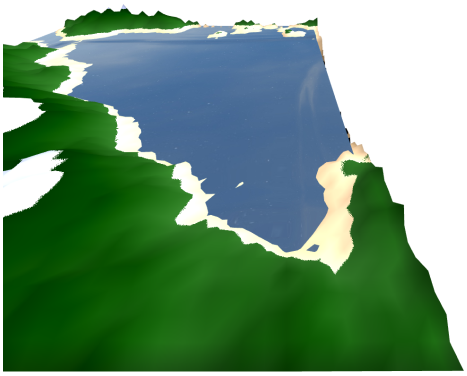


Figure 5.6: Single Shallow Water tile, where the water is unable to move past the tile, so reflects.

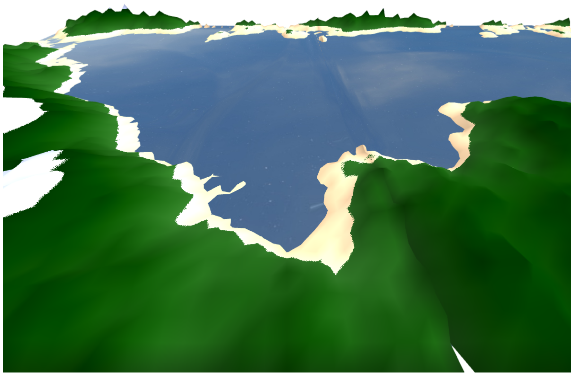


Figure 5.7: Two Shallow Water tiles, where the water is able to flow between tiles

scene to continue adding generated content through the application's lifecycle.

5.4 Summary

The asset controller implemented within this framework offers the developer an optimised, streamlined approach to the process of determining how to create and maintain generated assets. This is done through the implementation of various metrics to produce a bias that influences how the scene is to be produced and maintained. The asset controller can use these metrics, in addition to various benchmarks, to create a foundation upon which a generated scene can be based. The use of these metrics are inherent to ensuring that the desired user experience is available at any given time. They also provide important cues on how the asset controller should respond to any changes in performance availability on any given device at any point during runtime. The asset controller can respond to required changes in performance availability in a number of ways, such as: scaling assets up or down; increasing or decreasing the size of the grid; increasing or decreasing the quantity of assets generated or improving or reducing the resolution of generated assets.

Another way in which the asset controller can provide improved generation of assets is that it allows the framework to take advantage of multithreading available through the use of isolates in Dart. Although low performance devices will not provide much opportunity for the asset controller to utilise isolates, multithreading means that performance requirements do not compromise the opportunity for higher end devices to be able to run higher quality simulations. Where possible, the asset controller can use isolates to provide an efficient means of processing a large amount of data. Examples of when this framework could utilise the processing power of isolates include: when a significant number of assets needed to be generated; when assets needed to be generated at a high resolution; or when complex algorithms, such as the shallow water simulation, are being used.

The use of the asset controller means that a developer can be sure that an application will be able to adapt and scale throughout its lifecycle.

Chapter 6

Results

By their very nature graphical applications are complex. They require significant processing, which presents a barrier for the widespread production of web-based graphical applications. This thesis has devised a solution to this problem in the form of a framework built using Dart. Dart and the developed framework were tested to determine the effectiveness of the framework in a realised implementation. The results show that Dart offers high performance for complex graphical applications, and that the design of the framework adapts well to suit a range of platforms.

The speed of the framework can be gauged in two ways; when compared to a native implementation, and when compared to an implementation in another web language, that is, JavaScript.

For the comparison to a native language, that is, a language not run within a VM, a copy of Perlin Noise was written in C. Typically, a native implementation would be expected to perform faster than an implementation run in a VM. However, in this case, both application performed at similar speeds due to them sharing the same bottleneck in the updating of the OpenGL/WebGL buffers, and render calls[119]. The point of this test is not to suggest that Dart is as fast as C, but to show how in some cases, they are subject to similiar bottlenecks.

To test the performance of Dart, and to ensure an accurate comparison to JavaScript multiple benchmarks were created and run. The benchmarks that were selected for the comparison measured the updating sequences of Perlin Noise, and the shallow water simulation. Testing the updating sequence is representative of performance of the initialisation sequence, whilst also providing an opportunity to meaningfully test concurrent and sequential implementations. As well as indicating that Dart performs faster than JavaScript, these benchmarks indicate at what system state a concurrent implementation offers the best performance.

The adaptability of the framework was tested by running it across a range of devices with varying developer metrics. The successfulness of the framework achieving platform independence was judged by the level of accuracy with which the framework produced and maintained a scene in relation to previously defined developer metrics on each device.

6.1 Benchmarks

For the development of the language performance benchmarks the updating sequence of Perlin Noise, and of the shallow water simulation were chosen to test on. The updating sequence of an application allows for more variation in performance to occur unlike the set up cycle, which uses simple, and single once-off algorithms, such as the Diamond-Square algorithm, that complete rapidly irrespective of device specifications, and the complexity of the algorithm. This is clearly visible when running the Diamond-Square algorithm, as even when tested at the largest scale, of 10 tiles, with a resolution of 128x128, it completed almost 8 times faster sequentially.

To ensure continuity between the framework code, the JavaScript code was generated from the Dart code using the Dart2js process. Incidentally, the Dart2js process automatically optimises JavaScript code, which is capable of outperforming human written JavaScript [24].

Device	CPU	GPU	RAM
PC	i7 - 6800k 3.2 GHz	Nvidia 1080gt	32 GB DDR4
MacBook Pro 2013	i7 - 4558U	Nvidia 750m	16 GB DDR3
iPhone 7	Quad-core 2.34 GHz	PowerVR Series 7XT	2 GB LPDDR4
Samsung Galaxy S7	Octa-core (4x 2.3GHz, 4x 1.6GHz)	Mali-T880 MP12	4 GB LPDDR4

Table 6.1: Specifications of devices used for benchmarks

To incorporate a range of devices the framework was tested on a Windows PC, MacBook Pro, iPhone 7, and a Samsung Galaxy S7; the specifications of each device can be seen in 6.1.

As this research identified concurrency as a possible solution for providing better performance the benchmarks were performed in both concurrent, and sequential implementations.

To run the various benchmarks the first step was to create a fair testing system for all devices, and implementations. As debugging web applications on some devices such as Android and iPhone can become challenging, and affect device performance [11], the information regarding each set of benchmarks was stored on the device. After each benchmark was run the data was uploaded to a local web server running a basic REST server to retrieve the benchmarked results. This eliminated the need to open any developer tools while the benchmarks were running, which would have affected timings and skewed results.

Part of ensuring a fair test was keeping the resolution of the tiles, and the number of tiles generated the same for each series of benchmarks. The developer metrics allow the framework to create a system at whichever size it detects will best suit device performance. To override this function for the purposes of benchmarking the system settings were predefined before running the tests. The given algorithm was run to update the system 100 times as quickly as possible, then the system setting changed to the next increment of number of tiles, or resolution. Once all benchmarks had been run for Perlin Noise, the process was repeated using the shallow water simulation.

Once a set of tests were completed the results were sent to the node.js server in the form of a simple text file containing all necessary test data, such as size, resolution and timing. The timing information used within the benchmark was generated by taking the time difference between the call to update the asset, and the resulting update being returned. This did not take into account the time taken to initialize the scene, or to update the asset on screen.

The testing of the algorithms in a concurrent implementation was a similar process to that of testing sequentially, utilising a web server to retrieve the data, and cycling through the various system states. The primary difference is how the timing information was generated. Testing a concurrent implementation requires a check to be performed to ensure all threads have completed; this is not required when testing a sequential implementation as a task is only started once the previous task has completed. As isolates and web workers utilise a message passing approach to concurrency a callback is triggered when a thread returns data. When this callback is fired, a check was performed to check the status of all other threads. Once all threads had completed only then was the timer stopped. This ensured that all the required updating had been completed, and that the timer was not stopped before then.

Once the benchmarks had been completed the resulting data was interpreted to determine the performance of Dart against JavaScript, and the performance of a concurrent solution compared to that of a sequential implementation. This demonstrates if, and when Dart offers a performance advantage; likewise when a concurrent solution is better suited for a given situation, than a sequential implementation. The results are displayed as weighted averages, as to minimise the amount of data shown while still reflecting the overall trend.

6.1.1 Dart vs JavaScript

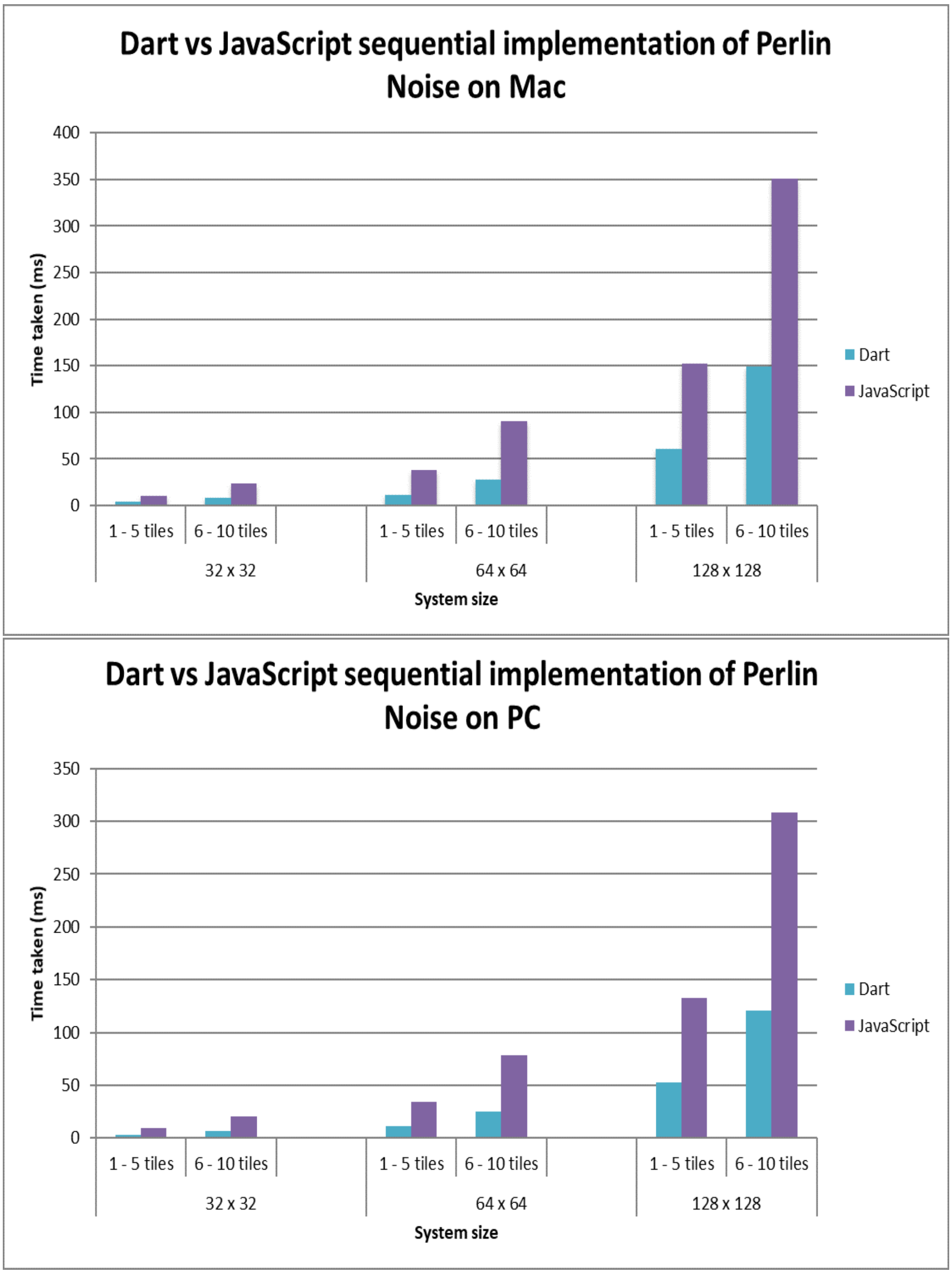
Dart and JavaScript were compared against each other both sequentially and concurrently in order to gain an understanding of the advantages of each language based on the algorithm and implementation. As Dart does not have a native VM for iOS or Android this comparison was made exclusively between Dart and JavaScript on a Window PC, and MacBook Pro. The results of these benchmarks can be seen in figures 6.4, 6.3, 6.1, 6.2. Even though Dart is not natively supported on iOS or Android, code that has been written in Dart and converted to JavaScript using the Dart2js function still performs faster than original JavaScript code [24].

In the comparison of Dart and JavaScript, Dart outperformed JavaScript in the updating of both Perlin Noise, and the shallow water simulation. This performance improvement is most evident on the Macbook Pro in the shallow water update, which completed in Dart almost three times faster than in JavaScript. The higher performance of Dart is also observable on PC, particularly when the system settings include a higher tile count. The most obvious example of this in the Perlin Noise benchmarks where Dart completes up to 2.5 times faster than JavaScript for generating 10 tiles at 128 x 128.

These results indicate the clear performance advantage offered by Dart when compared to JavaScript performing the same task. These results are regardless of whether implemented concurrently or sequentially.

6.1.2 Sequential vs Concurrent

Once the language performance had been established, the benchmarks were analysed to identify the performance advantages of a concurrent implementation over a sequential implementation. The benchmarks were run on the updating of Perlin Noise, and of the shallow water simulation in both concurrent and sequential implementations, across all devices using Dart where possible, otherwise utilising Dart2js to produce JavaScript code


Dart vs JavaScript sequential implementation of Perlin Noise on Mac

Dart vs JavaScript sequential implementation of Perlin Noise on PC

Figure 6.1: Comparison of updating time for sequential Dart and JavaScript implementations of Perlin Noise

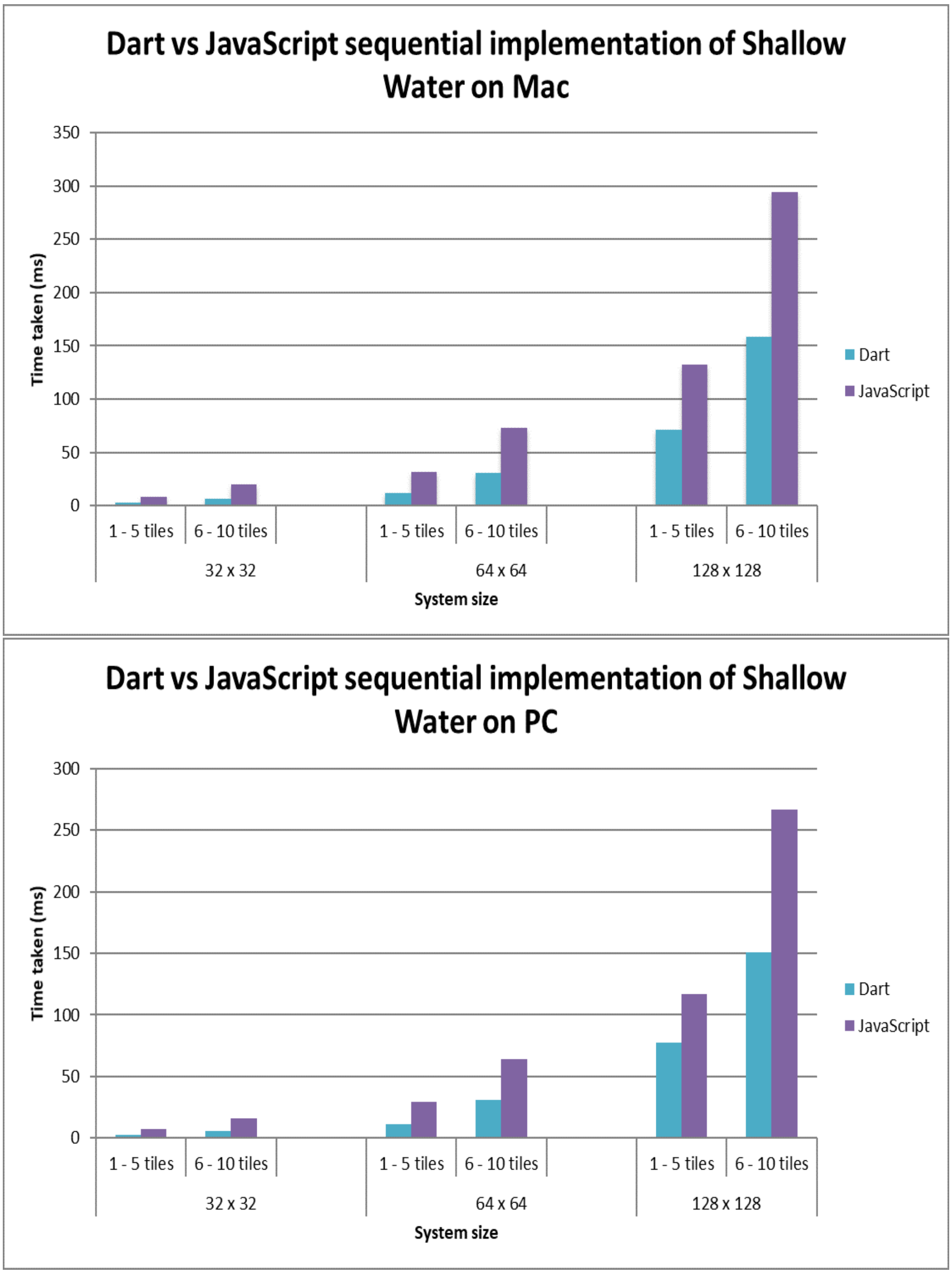


Figure 6.2: Comparison of updating time for sequential Dart and JavaScript implementations of Shallow Water simulation

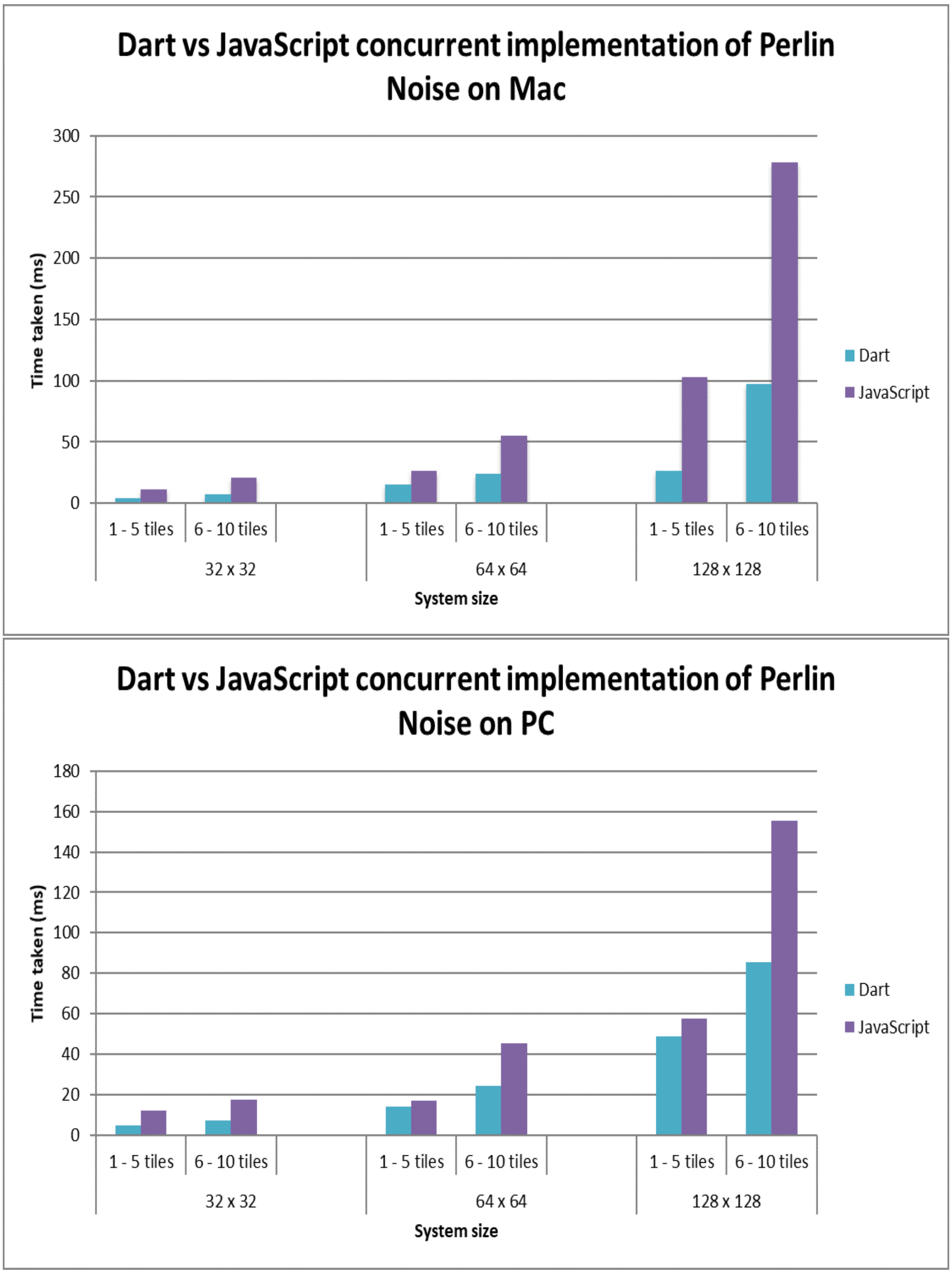


Figure 6.3: Comparison of updating time for concurrent Dart and JavaScript implementations of Perlin Noise

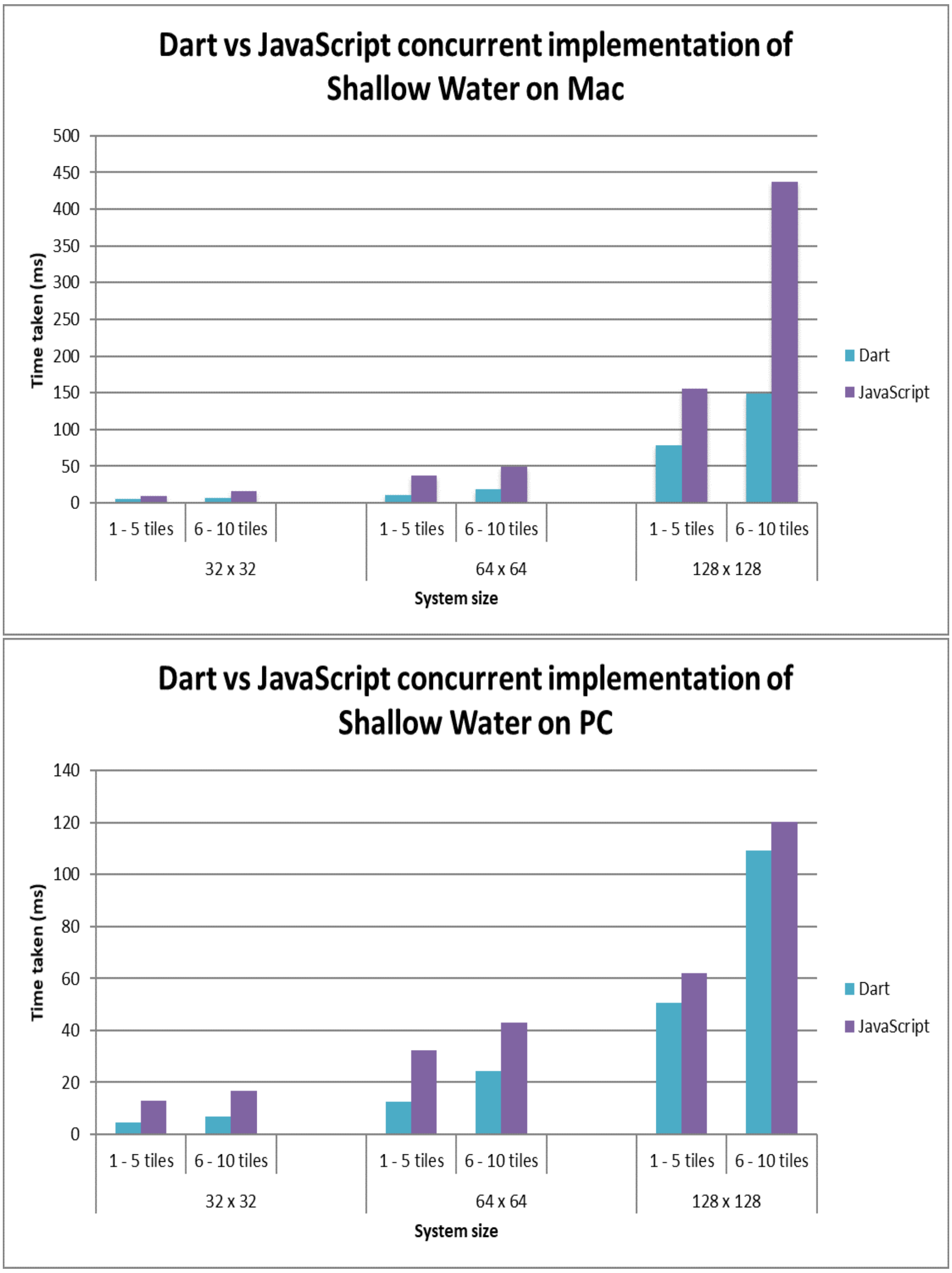
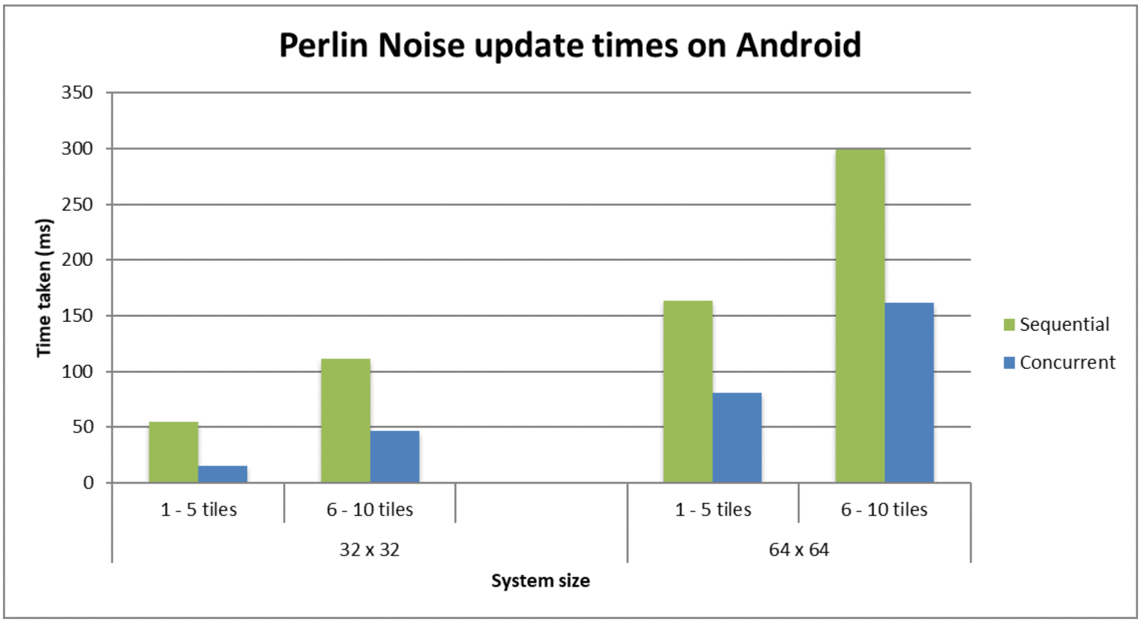
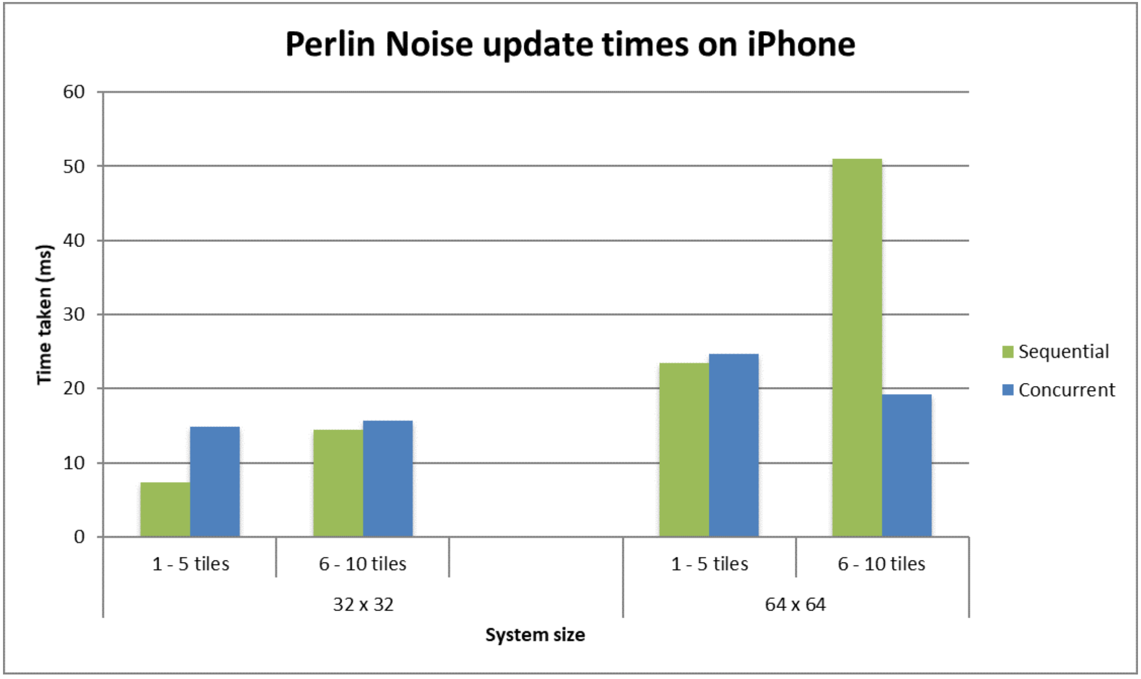


Figure 6.4: Comparison of updating time for concurrent Dart and JavaScript implementations of Shallow Water

for each benchmark. When testing on both iOS and Android the benchmarks were only run on system settings of up to a resolution of 64 x 64. This limitation is due to two factors: firstly, the screen size of handheld devices makes the noticeable difference between asset quality at resolutions of 64 x 64 and 128 x 128 negligible; secondly, the devices did not have sufficient hardware specifications to complete the scenes in a timely manner - this was particularly noticeable on Android, which became somewhat unresponsive at times when the system was set to 128 x 128.

The benchmarks comparing concurrent and sequential implementations highlight two findings. The first is that with all system states, and with both the Perlin Noise and shallow water simulation the Android phone experiences significant performance benefits from a concurrent implementation. This can be explained by Android OS allowing multiple background processes to occur at any given time[57]. Such processes are run on the main core, which would impact the performance of a sequential implementation. Therefore, where the process is complex a concurrent solution provides a much more effective data transfer system. It is worth noting that although the Android device does seem to have a higher specifications than the iPhone, its performance was found to be worse, this could be due to a number of issues, such as the acitvation of low power CPU cores being activated, slowing down the overall system. On all other devices, a concurrent implementation of Perlin Noise only began to offer superior performance once the system state was more complex than 5 tiles at a resolution of 64 x 64. Likewise, the concurrent implementations of the shallow water simulation began to outperform a sequential implementation at 6 - 10 tiles at a resolution of 64 x 64, with the exception of 1 - 5 tiles at 128 x 128 on the MacBook Pro, which performed better in a sequential implementation.


Perlin Noise update times on Android
Time taken (ms)
350
300
250
200
150
100
50
0
1 - 5 tiles
6 - 10 tiles
32 x 32
1 - 5 tiles
6 - 10 tiles
64 x 64
System size
Sequential
Concurrent


Perlin Noise update times on iPhone
Time taken (ms)
60
50
40
30
20
10
0
1 - 5 tiles
6 - 10 tiles
32 x 32
1 - 5 tiles
6 - 10 tiles
64 x 64
System size
Sequential
Concurrent

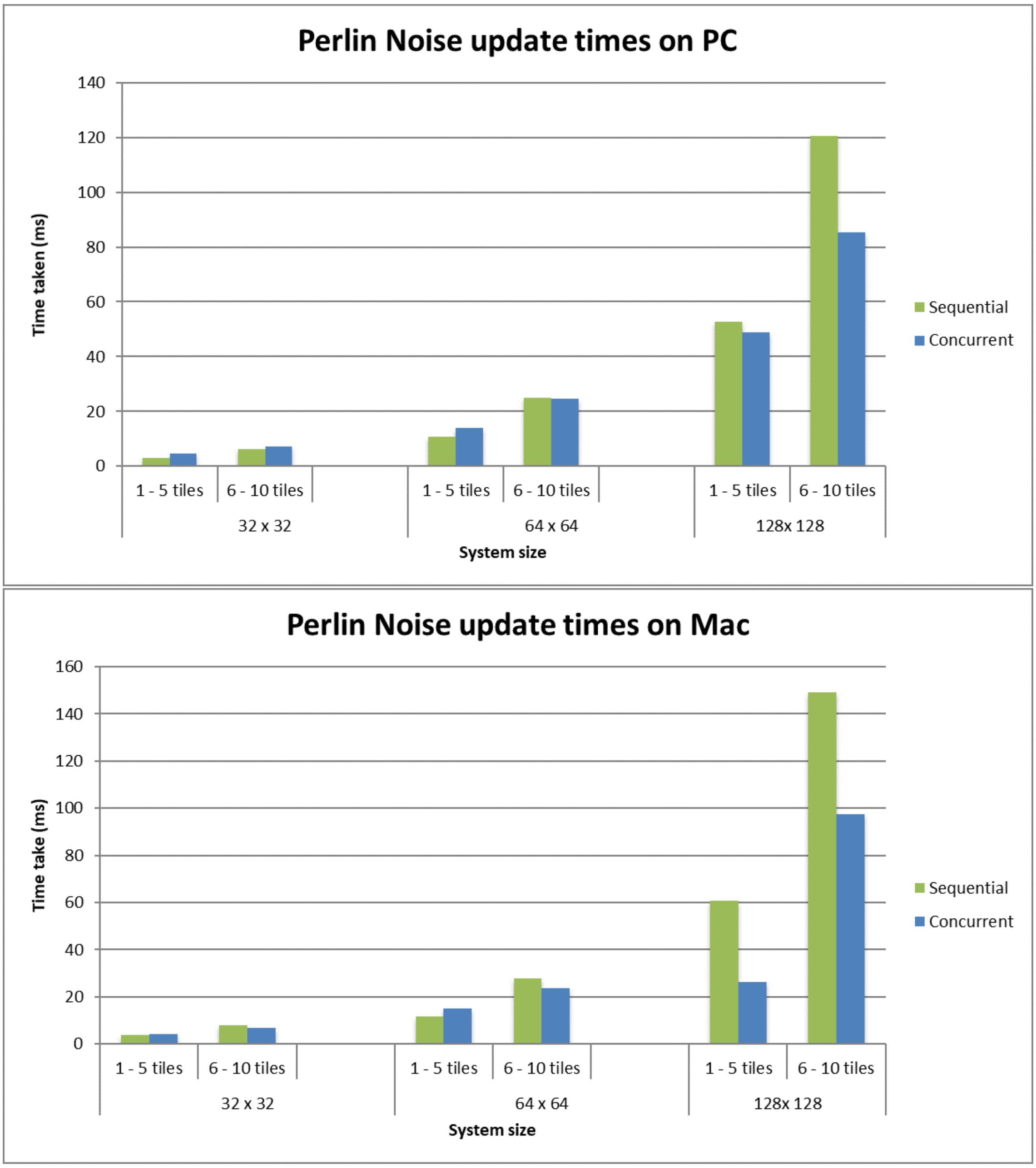
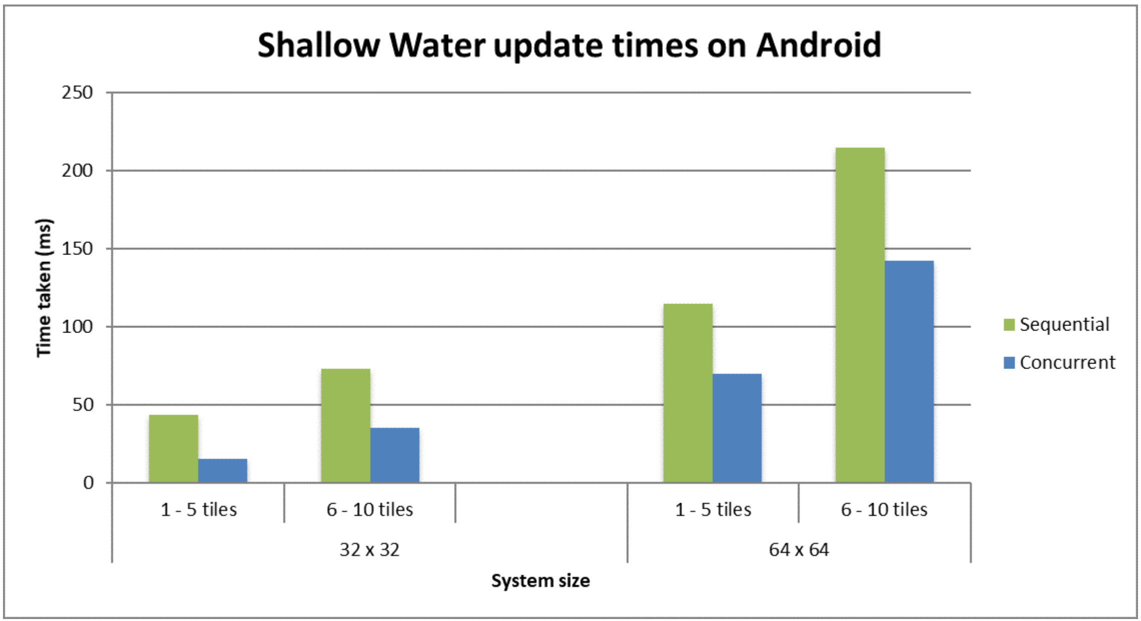
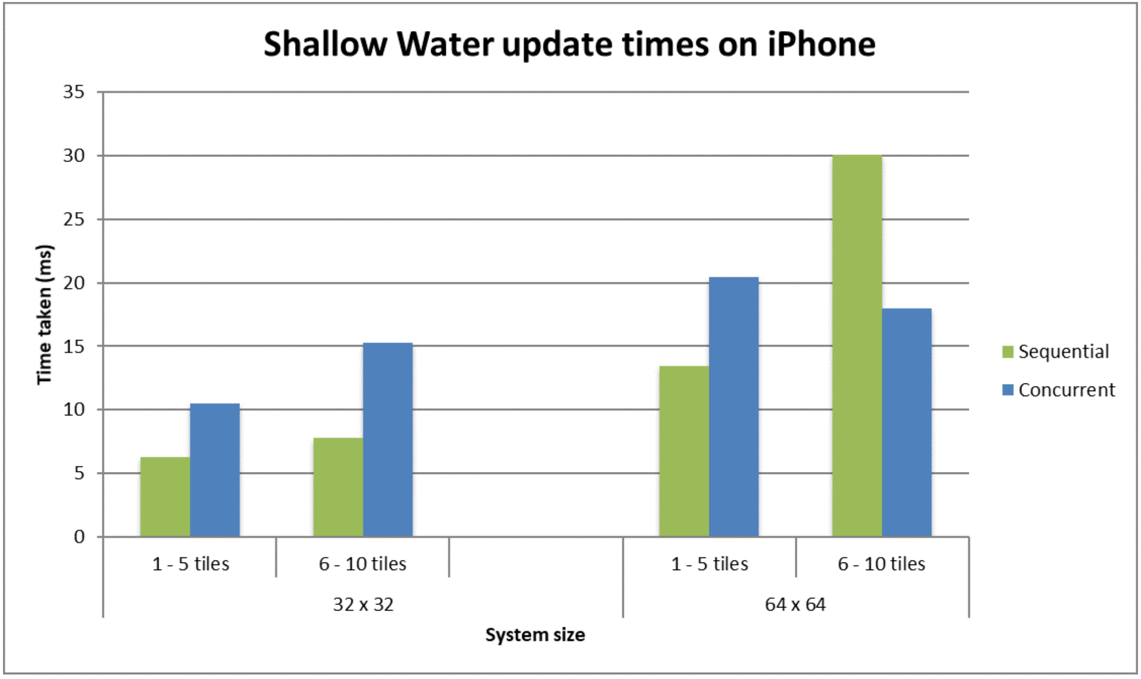


Figure 6.5: Comparison of updating time for concurrent and sequential Perlin Noise, across a range of devices


Shallow Water update times on Android
Time taken (ms)
250
200
150
100
50
0
1 - 5 tiles
6 - 10 tiles
1 - 5 tiles
6 - 10 tiles
32 x 32
64 x 64
System size
Sequential
Concurrent


Shallow Water update times on iPhone
Time taken (ms)
35
30
25
20
15
10
5
0
1 - 5 tiles
6 - 10 tiles
1 - 5 tiles
6 - 10 tiles
32 x 32
64 x 64
System size
Sequential
Concurrent

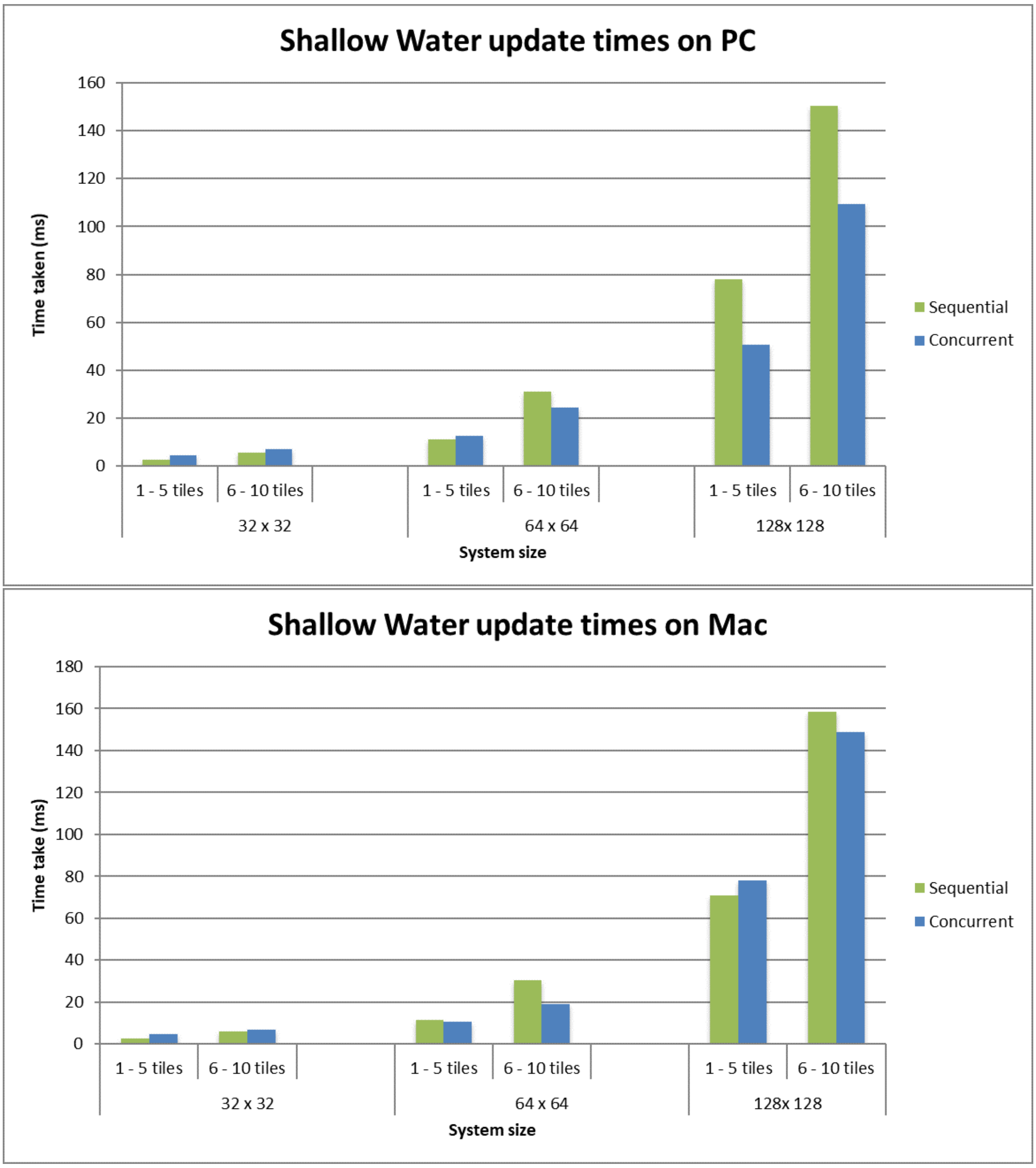


Figure 6.6: Comparison of updating time for concurrent and sequential Shallow Water simulation, across a range of devices

The comparison of implementation types show that Android benefits from concurrency at all system states for the updating of Perlin Noise and the shallow water simulation, whereas other devices begin to experience performance benefits after the system grows to a size of six tiles or greater, at a resolution of 64 x 64 or higher.

These benchmarks show clearly that Dart outperforms JavaScript in all tested situations, regardless of concurrent or sequential implementation. In addition to this, it can be seen that a sequential implementation is useful for smaller tasks, or for simpler system states; however, concurrency offers the best performance for more complex simulated tasks.

6.2 Device performance

In order to demonstrate how the framework performs on each device the framework was run with the following emphasis through the use of the developer metrics. The first metric defined was the use of a high quality asset, this sets the framework to create fewer assets at a higher resolution rather than generating a larger number of assets. The next metric tested had a bias towards increasing the area covered, meaning that the focus was on producing as many tiles as possible, without much regard as to the quality of the generated assets. The final set of metrics tested was to cover an even split between the quality of the tiles created, and the number of tiles generated. This produces a scene of reasonable quality, while also generating more tiles.

Testing the framework's performance on each device had different requirements to the tests run for benchmarking. Whereas benchmarking was concerned only with the timings of updating the water algorithms, therefore the framework only generated water, testing for device performance requires the entire landscape to be generated. This means that, via the contour tracing algorithm that is discussed in Chapter 4, only portions of the entire landscape were set to be water. This consequent reduction in the total water area in turn reduces the average processing requirements per tile. This meant that devices were able to generate larger landscapes than benchmarks alone would suggest.

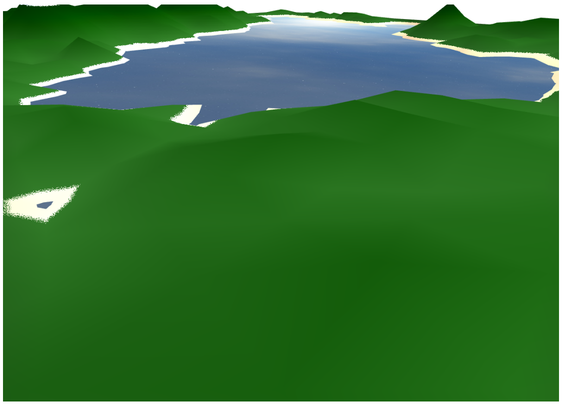


Figure 6.7: Single generated tile at a resolution of 32 x 32

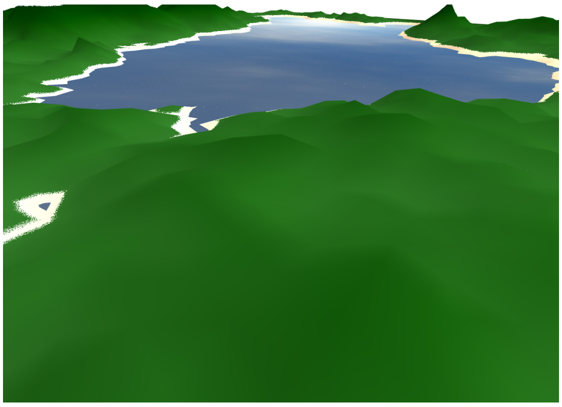


Figure 6.8: Single generated tile at a resolution of 64 x 64

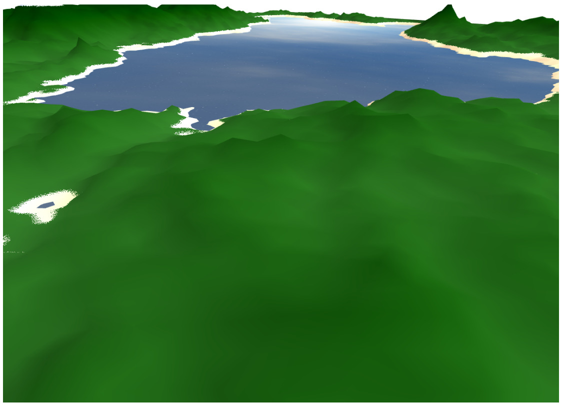


Figure 6.9: Single generated tile at a resolution of 128 x 128

6.2.1 PC

The first set of developer metrics set the framework to create fewer tiles, but at a high resolution. This also changes the preferred method of water generation from Perlin Noise to the shallow water simulation. These metrics resulted in a scene consisting of five tiles, utilising shallow water. The central tile was generated at a resolution of 128 x 128, while the outer four were created at 64 x 64.

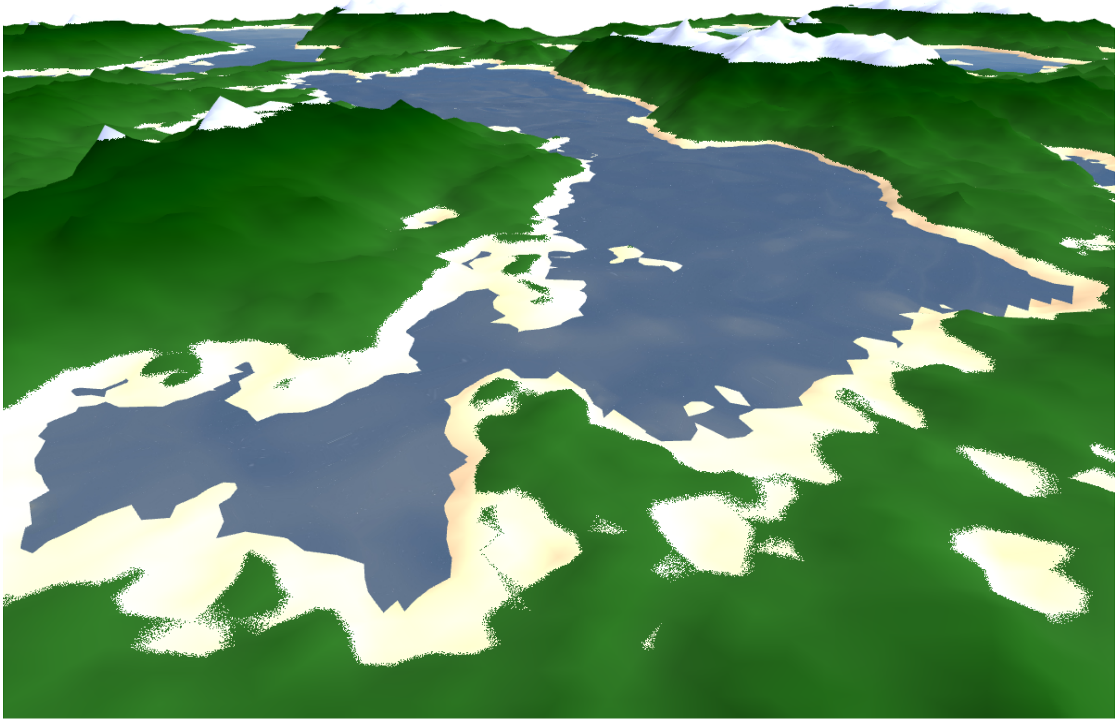


Figure 6.10: Generated scene when developer metrics on a PC define a preference towards high quality tile generation.

When set to generate a larger number of tiles with no regard to quality, the water generation utilised Perlin Noise. The total scene size was fifteen tiles, with the five central tiles set to a resolution of 64 x 64, and the outer 10 set to 32 x 32.

The final set of developer metrics allowed for a higher resolution of tile to be created, but utilised Perlin Noise for the water generation, which allowed for a larger number of tiles to be generated. The use of these developer metrics resulted in a scene with a total size of seven tiles; the centre tile was produced at a resolution of 128 x 128, and the remaining six outermost tiles were generated at a resolution of 64 x 64.

6.2.2 Macbook

The MacBook Pro had similar results to the PC, outperforming it in some areas. When tested with the developer metrics set to create a high quality scene, the result was seven tiles generated utilising the shallow water simulation; that is two more tiles than what the PC produced with the same metrics. As with the PC, the centre tile was generated at a resolution of 128 x 128, and the outer tiles at a resolution of 64 x 64.

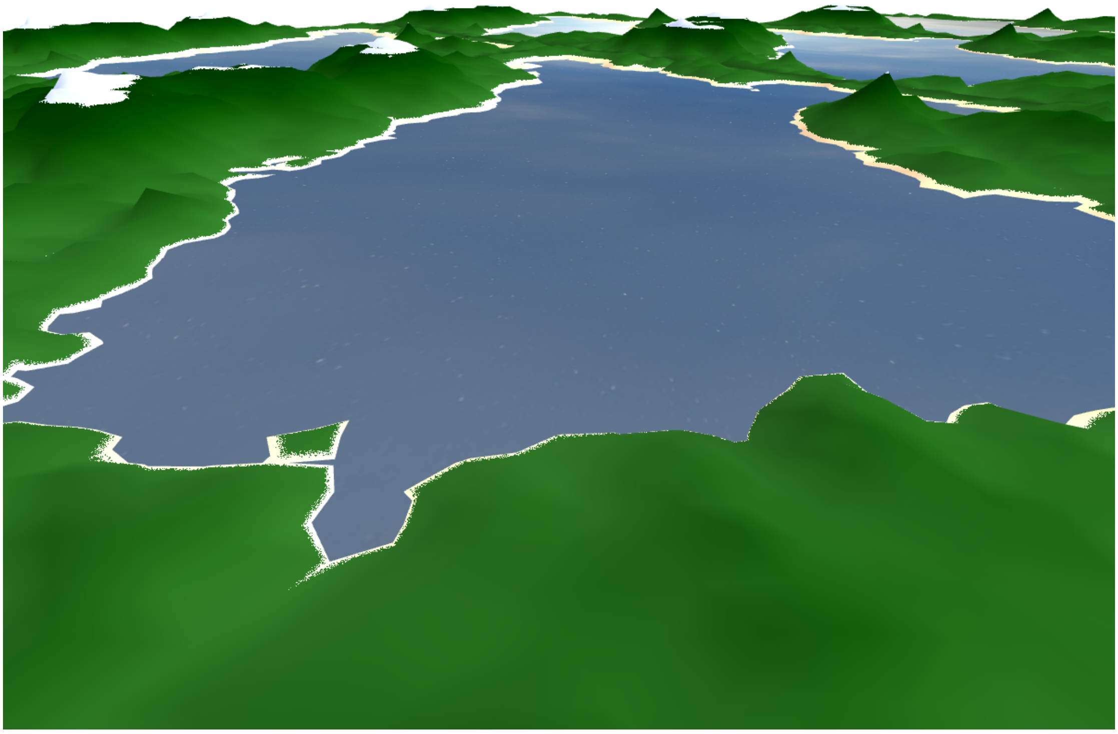


Figure 6.11: Generated scene when developer metrics on a Mac Book Pro define an even level of quality, and number of tiles generated.

When the developer metrics defined the generation of a larger area with reduced emphasis on the quality of tiles generated the result was a total of 14 tiles - one tile less than the PC using the same metric. Like on PC these tiles used Perlin Noise to display water, and created the centre four tiles at a resolution of 64 x 64, with the outer ten tiles being generated at a resolution of 32 x 32.

When the developer metrics were set to display an equal balance between area covered, and quality of assets generated a total of nine tiles were created. Similarly to the PC, these tiles utilised Perlin Noise; however, all tiles were created at a resolution of 64 x 64.

6.2.3 Android

Despite experiencing the greatest performance advantage from utilising a concurrent implementation Android was the slowest performing device overall. For both mobile devices the resolution was limited to a maximum of 64 x 64; however, when the developer metrics were set to high quality, the Samsung Galaxy S7 was only able to generate a single tile with a resolution of 64 x 64 using the shallow water simulation.

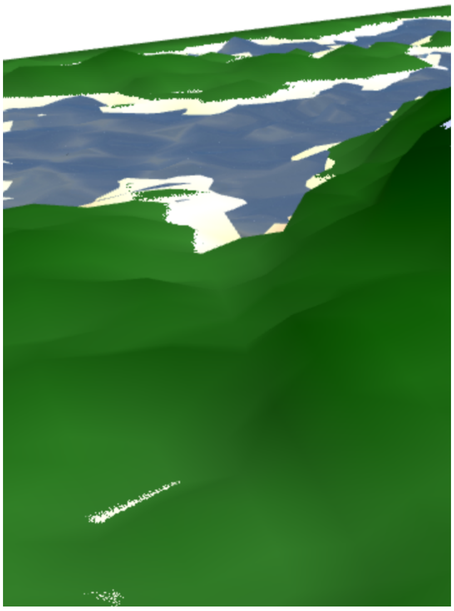


Figure 6.12: Generated scene when developer metrics on an Android define a preference towards high number of tiles generated.

When the developer metrics were defined to generate multiple tiles at a lower resolution the result was a total of five tiles created. These tiles utilised Perlin Noise to create water, and were set to the lowest resolution of 32 x 32.

The final developer metric of creating a scene with even emphasis on the quality of tiles produced and the number of tiles generated resulted in a total of four tiles created. These tiles again used Perlin Noise, with the centre tile having a resolution of 64 x 64, and the surrounding three tiles at a resolution of 32 x 32.

6.2.4 iPhone

The iPhone proved much more capable than Android at using high quality assets regardless of the developer metrics. As with Android, the iPhone was limited to a maximum resolution of 64 x 64; however, for the high quality metric it was able to generate a scene of six tiles at a resolution of 64 x 64 using the shallow water simulation.

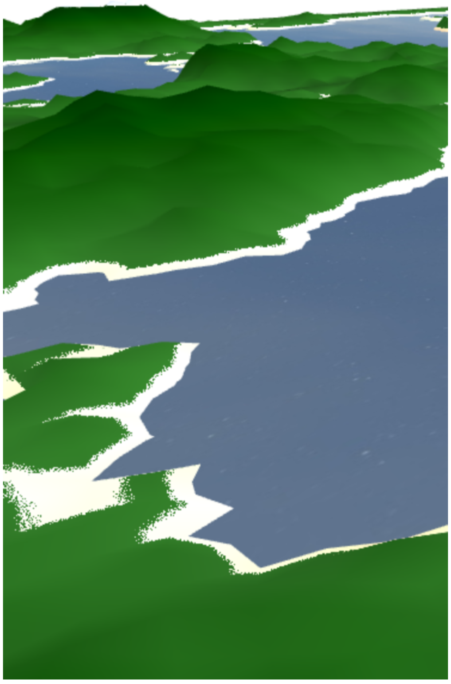


Figure 6.13: Generated scene when developer metrics on a iPhone define an preference towards high quality tile generation

When the developer metrics were set to generate a large number of tiles the iPhone was capable of generating 12 tiles at 32 x 32, using Perlin Noise - that is more than double the number of tiles Android produced with the same metrics.

Once the developer metrics were set to an even level of detail and number of tiles generated, the resulting scene contained 10 tiles. Two of these tiles had a resolution of 64 x 64, while the other eight were set to 32 x 32. In this case the water generation method was the shallow water simulation. The results of this developer metric contrast sharply to the Android results using the same metrics, which produced less tiles and used the Perlin Noise method for generating water.

6.3 Conclusion

The results presented in this chapter demonstrate that Dart outperforms JavaScript, even when the JavaScript code has been optimised using the Dart2js function. These results also suggest that as the complexity of an application increases so does the viability of using a concurrent implementation, although a sequential implementation is more suitable for simple generation tasks. The results also prove how a framework can be created to scale in response to the performance capability of a range of devices.

Chapter 7

Conclusions

This research sought to provide solutions for the development of 3D graphical applications that can be accessed on a range of devices without compromising the set of features and functions available across different device types. This research identified the need to make considered decisions about the development language, method of asset generation, scalability of the application, and maximising access to processing power. Individual solutions to these concerns have been identified and combined to form an arrangement of technologies, functions, and processes for creating platform independent 3D graphical applications.

The objectives of this research (laid out out in chapter 1) have all been questioned, with various results found. Real time rendering on a range of devices though web technologies has been achieved, though Dart and WebGL. The use of threads to accelerate simulation proccessing was also found to benefit a range of devices, at higher levels of compute requirements. Additionally by using these simulations, environments can be generated without the need of downloading assets at various levels of details

Choosing to develop with a web language has numerous traits which help to achieve platform independence in a holistic way. As web languages run in a VM this gets around compatibility issues that would other arise from using languages that are natively supported on some devices but not others, thereby requiring code to be redeveloped for widespread platform independence. Ideally, the web language chosen will also have methods for achieving concurrency in order to provide further processing power should it be required for complex renderings.

Dart provides all the necessary functionality; however, as it is a new language, support for it can be uncertain, and it is not guaranteed to continue long term, or remain used for the same purposes as it is now. In contrast, JavaScript is a versatile language, but at this stage does not provide as many opportunities for optimising code as is desirable for achieving high performing platform independent applications. The existence of Dart suggests that there is at the very least an awareness that this kind of language is in demand, so it is not unrealistic to hope that more languages will be developed to cater specifically to achieving platform independence[112].

Methods of asset generation are extremely varied and to some extent come down to a developer's personal preference. To include 3D graphics in a platform independent application the size of the assets must be variable so that they can scale according to device processing capability. Using generative algorithms to create content allows for assets to be rendered at the highest possible quality that a device is able to achieve. From a consumer-friendly perspective, generated content also caters to users who have limited data available to them and would not respond well to an application that had large files frequently downloading. Alternatively, a library of models at a range of sizes could be created for inclusion on a range of devices, although this would incur a higher rate of data usage.

Language choice and asset generation method are the fundamental aspects of a platform independent graphical application. The next most important components are those that reinforce the range of devices upon which an application can be run. The ability to scale content has been identified as an essential process. It is all very well to have methods of implementing content at various resolutions and sizes; however, the application must have a way to recognise what the appropriate resolutions and sizes for content are. This framework has used the controller class to achieve this, but this could be replaced by any process that benchmarks the application at runtime to ascertain the performance output of a device, then uses that data to alter the system state to provide optimal asset quality and user experience. There should also be a function that allows the developer to dictate a bias towards various asset qualities, such as number of assets rendered. This framework uses developer metrics to place emphasis on asset resolution, and number of assets, but these could have any criteria as determined by the developer.

Graphical applications require significant processing power so it is important that a developer of platform independent graphical applications take advantage of any source of processing power available to them. Ideally, this will come in the form of accessing the GPU of a device as these are becoming increasingly common and powerful in a range of devices. WebGL currently provides the most user-friendly solution for developers; however, there are other options available in the form of graphical libraries, OpenGL, DirectX and OpenGL ES.

7.1 Implications for platform independence

This research has identified that the various CPU architectures present in a range of devices, and the variety in languages that are natively supported on a range of devices are problematic for creating applications that can run on more than a few selected device types. The solution to this is to use web languages as they run in a web browser's VM, and web browsers are present in all modern devices.

The development of modern web languages is still very much in a growth phase. When compared to native applications, even simple web applications are only just beginning to equal their complexity. By comparison, graphical web-based applications are very much still in their infancy for two key reasons.

Firstly, the purpose of making an application to be web-based would be so that it is not device dependent. However, until recently, devices that are web-capable have not also been significantly graphics-capable. For example, mobile phones have been able to connect to the internet since the late 1990's[10], yet modern graphical applications, such as games, have only become prevalent since the mid 2000's. This means that conversation about achieving platform independence via the web has only been tangible since the 1990's, and platform independence specifically in a graphical context since the mid 2000's.

Secondly, even though platform independence may have been thought about within the industry, the means to develop it has not been available until recently. This is indicated by the release of WebGL in 2011, which allows web applications to access the GPU. Likewise, Dart was only released in July 2014. The significance of the release of Dart is that until it was available JavaScript was really the only choice for graphics on the web. However, as JavaScript was not designed for this purpose it has areas of inefficiencies that are not present in Dart, which has been designed with platform independence in mind.

The release of all this technology indicates that the industry is keen to explore the possibilities of platform independence via the web, it just needs development.

7.2 Implications for web languages

The most popular web languages in use today, such as PHP and JavaScript, have only existed since the mid 1990's.

Since the mid 1990's developers have used web languages to create increasingly engaging websites, which has lead to the popularity of web applications. Currently, web applications are very much centered around storing and accessing data, social media and communication, and organisation and administration for both personal and business interests.

JavaScript is still the most popular choice for the development of web applications, despite this being somewhat more complex endeavour than what the language was originally intended for. Furthermore, the use of JavaScript has to be heavily supplemented by frameworks such as React, and Angular. Now that advances in technology mean that most devices are capable of running interactive web pages with quality graphical components it is time for the development of web languages that are designed with such uses in mind.

With particular uses in mind web languages can be created with more structure to make the formalities of programming less of an issue for a developer. This can be achieved through utilising classes, types, interfaces, and a defined structure. This process of evolution can be seen in other programming languages such as C deriving C++, or C# wherein additional features were added, and the language adapted to better suit the emerging needs of a developer.

There is scope to create web languages that allow for optimised code so that they perform faster. This can be achieved through minimising the impact of the virtual machine, allowing for the application to be run as close to natively as possible, or allowing for access to specialised CPU instruction sets. Or the access to native threads, without the overhead produced by web workers or isolates.

7.3 Future work

7.3.1 Generative algorithms

To increase the level of complexity and realism of a scene the algorithms used to generate assets could be extended to include finer environmental detail, such as vegetation, and the ability to place specific assets and have the environment intelligently generate around it.

Modelling of vegetation could be added in many ways. For lower quality options, different textures could be applied directly to the land mesh to represent grass, but could become as complex as rendering individual blades of grass that move in the wind. Larger assets such as trees could likewise be represented by wrapping textures around tree-shaped frames, through to having individually unique assets generated by algorithms that can generate trunks, roots, and leaves. This research only went so far as to investigate how to generate content, so further research would allow for the manual placement of assets and for the system to respond accordingly when generating the remaining environment. For example, a user might place a tree at a location, and it would be undesirable for the system to generate water at that same location, thereby submerging the tree.

7.3.2 Implementation types

This research confirmed that for more complex graphical applications better performance is achieved via a concurrent implementation. However, concurrent implementations incur a processing time overhead as a result of initialising web workers (in JavaScript) and isolates (in Dart). This can negatively impact performance for the simpler components of an application, such as the generating of a mesh using the Diamond-Square algorithm.

Further research would allow the application to complete simple processes in a sequential implementation, thereafter the application could switching to a concurrent implementation to render more complex aspects, such as generated assets using algorithms such as the shallow water simulation.

7.3.3 Other languages

As web languages become more complex and capable there are likely to be alternatives worth investigating for the purpose of creating platform independent graphical applications. At this stage web languages stand to be improved by the inclusion of more feature sets that increase the performance of web applications. As the development of high performance graphical web applications is relatively recent the criteria for desirable feature sets is still being determined.

However, it is already evident that some useful feature sets would be those that provide lower level access to the GPU in order to improve performance[18]. Likewise, having a more structured language that allows code to be built in a more optimised fashion would prove useful.

Additionally, a compiled web language would minimise runtime errors, thereby ensuring a higher degree of stability of the application, and making the development process more efficient[20].

7.3.4 Developer metrics

Depending on the intended use of an application the developer metrics may need to vary considerably. This could be expanded through utilising metrics to define preferred content generation algorithms or to alter the properties, such as colour and size, of assets in response to other aspects of the scene. Additionally these metrics could be extended to allow for predefined asset placement, so that while a scene may change an underlying structure is maintained.

7.3.5 VR support

To remain truly platform independent applications will need to have support for virtual reality (VR). Support for VR has three components. The first is setting the target frame rate of the application to 90 fps[87]. Ideally, this frame rate would be toggled on and off per the device type as a frame rate that high would be unnecessary for use on any other device. Secondly, to support VR the application would need the functionality to render to two different canvases. This would need to be managed by the application so that it could detect when a VR device was present in order to render a second frame. Finally, the user inputs of a VR device would need to be mapped and added to the input controller.

Likewise, similar additions and alterations would need to be made for any new device type significantly different to existing technology.

7.4 Conclusion

There are many methods a developer may choose to combine in order to achieve platform independence, and there are new technologies being developed and released everyday that support the pursuit of widespread platform independence.

This research has proven that given even the limited and still new technology that is currently available platform independent graphical applications are viable. This particular framework relates to the simulation of environments; however, there are many ways of using graphical platform independence that have not even been imagined yet.

Advances in hardware development make it economical for the widespread manufacturing of mobile phones that are as powerful as a low end computers - it is time for software to catch up, and platform independence is the way to do it.

Bibliography

[1] J. Almeida, M. van Sinderen, L. Pires, and D. Quartel, “A systematic approach to platform-independent design based on the service concept,” in *Enterprise Distributed Object Computing Conference, 2003. Proceedings. Seventh IEEE International*, Sept 2003, pp. 112–123.

[2] G. Anthes, “HTML5 leads a web revolution,” *Communications of the ACM*, vol. 55, no. 7, pp. 16–17, jul 2012.

[3] M. Belchin and P. Juberias, *Web Programming with Dart.* Berkeley, CA: Apress, 2015, ch. Using Pub and dart2js to Compile Applications to JavaScript, pp. 75–86.

[4] M. Bender, R. Klein, A. Disch, and A. Ebert, “A functional framework for web-based information visualization systems,” *Visualization and Computer Graphics, IEEE Transactions on*, vol. 6, no. 1, pp. 8–23, 2000.

[5] G. Benguria, X. Larrucea, B. Elvesæter, T. Neple, A. Beardsmore, and M. Friess, “A platform independent model for service oriented architectures,” in *Enterprise Interoperability: New Challenges and Approaches.* Springer, London, 2007, pp. 23–32.

[6] G. Bierman, M. Abadi, and M. Torgersen, “Understanding TypeScript,” in *ECOOP 2014 – Object-Oriented Programming. Lecture Notes in Computer Science*, R. Jones, Ed., vol. 8586. Berlin, Heidelberg: Springer, 2014, pp. 257–281.

[7] W. Binder, J. Hulaas, P. Moret, and A. Villazón, "Platform-independent profiling in a virtual execution environment," *Software: Practice and Experience*, vol. 39, no. 1, pp. 47–79, 2009.

[8] M. Bolin, *Closure: The Definitive Guide*, 1st ed. O'Reilly Media, Inc., 2010.

[9] D. Bresch and B. Desjardins, "On the construction of approximate solutions for the 2D viscous shallow water model and for compressible Navier–Stokes models," *Journal de Mathématiques Pures et Appliquées*, vol. 86, no. 4, pp. 362 – 368, 2006.

[10] P. Budmar. (2012) Why japanese smartphones never went global. PC World. [Online]. Available: https://www.pcworld.idg.com.au/article/430254/why_japanese_smartphones_never_went_global/

[11] T. Buschtöns. (2012) Debugging javascript on Android and iOS. [Online]. Available: https://eclipsesource.com/blogs/2012/08/14/debugging-javascript-on-android-and-ios/

[12] F. Chang and C.-J. Chen, "A component-labeling algorithm using contour tracing technique," in *2013 12th International Conference on Document Analysis and Recognition*, vol. 2. IEEE Computer Society, 2003, pp. 741–741.

[13] A. Charland and B. Leroux, "Mobile application development: Web vs. native," *Commun. ACM*, vol. 54, no. 5, pp. 49–53, may 2011.

[14] B. Chen and Z. Xu, "A framework for browser-based multiplayer online games using webgl and websocket," in *Multimedia Technology (ICMT), 2011 International Conference on*, July 2011, pp. 471–474.

[15] K.-T. Cheng and Y.-C. Wang, "Using mobile GPU for general-purpose computing–a case study of face recognition on smartphones," in *VLSI Design, Automation and Test (VLSI-DAT), 2011 International Symposium on.* IEEE, 2011, pp. 1–4.

[16] R. Chugh, J. A. Meister, R. Jhala, and S. Lerner, “Staged information flow for javascript,” *SIGPLAN Not.*, vol. 44, no. 6, pp. 50–62, jun 2009.

[17] J. Congote, A. Segura, L. Kabongo, A. Moreno, J. Posada, and O. Ruiz, “Interactive visualization of volumetric data with webgl in real-time,” in *Proceedings of the 16th International Conference on 3D Web Technology.* ACM, 2011, pp. 137–146.

[18] A. Corporation. (2018) WebGPU demos. [Online]. Available: https://webkit.org/demos/webgpu/

[19] C. Crassin, F. Neyret, S. Lefebvre, and E. Eisemann, “Gigavoxels: Ray-guided streaming for efficient and detailed voxel rendering,” in *Proceedings of the 2009 symposium on Interactive 3D graphics and games.* ACM, 2009, pp. 15–22.

[20] A. Crichton. (2018) Javascript to rust and back again: A wasm-bindgen tale. [Online]. Available: https://hacks.mozilla.org/2018/04/javascript-to-rust-and-back-again-a-wasm-bindgen-tale/

[21] D. Crockford, *JavaScript: The Good Parts*, 1st ed. O’Reilly Media, Inc., 2008.

[22] P. Cullen. (2013) Joystick to JSON/HTTP in processing. [Online]. Available: http://mindmeat.blogspot.co.nz/2013/06/joystick-to-jsonhttp-in-processing.html

[23] C. Daly, J. Horgan, J. Power, and J. Waldron, “Platform independent dynamic java virtual machine analysis: The java grande forum benchmark suite,” in *Proceedings of the 2001 Joint ACM-ISCOPE Conference on Java Grande*, ser. JGI ’01. New York, NY, USA: ACM, 2001, pp. 106–115.

[24] Dart. (2015, March) Dart VM and dart2js performance. [Online]. Available: https://www.dartlang.org/performance/

[25] ——. (2015, April) Frequently asked questions. https://www.dartlang.org/support/faq.html.

[26] M. Dowty and J. Sugerman, "GPU virtualization on VMware's hosted I/O architecture," *ACM SIGOPS Operating Systems Review*, vol. 43, no. 3, pp. 73–82, 2009.

[27] Emscripten. (2018) Opengl support in emscripten. [Online]. Available: https://kripken.github.io/emscripten-site/docs/porting/multimedia_and_graphics/OpenGL-support.html

[28] C. Everitt and J. McDonald, "Beyond porting: How modern OpenGL can radically reduce driver overhead," Steam Dev Days, 2014.

[29] A. Ezust and P. Ezust, *An Introduction to Design Patterns in C++ with Qt 4*, 1st ed. Prentice Hall, 2006.

[30] P. A. Fishwick, "Web-based simulation: Some personal observations," in *Proceedings of the 28th Conference on Winter Simulation*, ser. WSC '96. Washington, DC, USA: IEEE Computer Society, 1996, pp. 772–779.

[31] B. N. Florian Loitsch. (2011, November) Why not a bytecode VM? [Online]. Available: https://www.dartlang.org/articles/why-not-bytecode/

[32] Flutter. (2016) Flutter.io. [Online]. Available: https://flutter.io/

[33] ——. (2018) Technical overview. [Online]. Available: https://flutter.io/technical-overview/

[34] A. Gal, B. Eich, M. Shaver, D. Anderson, D. Mandelin, M. R. Haghighat, B. Kaplan, G. Hoare, B. Zbarsky, J. Orendorff, J. Ruderman, E. W. Smith, R. Reitmaier, M. Bebenita, M. Chang, and M. Franz, "Trace-based just-in-time type specialization for dynamic languages," *SIGPLAN Not.*, vol. 44, no. 6, pp. 465–478, jun 2009.

[35] M. Gardner, "Mathematical games: The fantastic combinations of John Conway's new solitaire game life," *Scientific American*, vol. 223, no. 4, pp. 120–123, 1970.

[36] A. A. Giunta, S. F. Wojtkiewicz, M. S. Eldred *et al.*, “Overview of modern design of experiments methods for computational simulations,” in *Proceedings of the 41st AIAA aerospace sciences meeting and exhibit, AIAA-2003-0649*, 2003.

[37] Google. Chrome experiments. [Online]. Available: https://experiments.withgoogle.com/collection/chrome

[38] ——. What is the closure compiler. [Online]. Available: https://developers.google.com/closure/compiler/

[39] ——. (2018) Write HTTP clients & servers. [Online]. Available: https://www.dartlang.org/tutorials/dart-vm/httpserver

[40] C. Grobmeier. (2011, Nov) Dart isolates. Online. [Online]. Available: https://www.grobmeier.de/dart-isolates-08112011.html

[41] J. Harjono, G. Ng, D. Kong, and J. Lo, “Building smarter web applications with HTML5,” in *Proceedings of the 2010 Conference of the Center for Advanced Studies on Collaborative Research*, ser. CASCON ’10. Riverton, NJ, USA: IBM Corp., 2010, pp. 402–403.

[42] M. Hendrikx, S. Meijer, J. Van Der Velden, and A. Iosup, “Procedural content generation for games: A survey,” *ACM Trans. Multimedia Comput. Commun. Appl.*, vol. 9, no. 1, pp. 1:1–1:22, Feb. 2013. [Online]. Available: http://doi.acm.org/10.1145/2422956.2422957

[43] S. Herhut, R. L. Hudson, T. Shpeisman, and J. Sreeram, “Parallel programming for the web.” in *HotPar*, 2012.

[44] F. S. Hill, *Computer Graphics: Using OpenGL*, 3rd ed. Pearson, 2007.

[45] F. S. Hill and S. M. Kelley, *Computer Graphics: Using OpenGL*, 2nd ed. Prentice Hall, 2000.

[46] C.-F. Hollemeersch, B. Pieters, A. Demeulemeester, P. Lambert, and R. Van de Walle, "Real-time visualizations of gigapixel texture data sets using HTML5," in *International Conference on Multimedia Modeling.* Springer, 2012, pp. 621–623.

[47] P. M. Hubbard, "Collision detection for interactive graphics applications," *Visualization and Computer Graphics, IEEE Transactions on*, vol. 1, no. 3, pp. 218–230, 1995.

[48] R. Ijtihadie, Y. Chisaki, T. Usagawa, H. Cahyo, and A. Affandi, "Offline web application and quiz synchronization for e-learning activity for mobile browser," in *TENCON 2010 - 2010 IEEE Region 10 Conference*, Nov 2010, pp. 2402–2405.

[49] D. Jacquet, F. Hasbani, P. Flatresse, R. Wilson, F. Arnaud, G. Cesana, T. Di Gilio, C. Lecocq, T. Roy, A. Chhabra *et al.*, "A 3 GHz dual core processor ARM cortex TM-A9 in 28 nm UTBB FD-SOI CMOS with ultra-wide voltage range and energy efficiency optimization," *IEEE Journal of Solid-State Circuits*, vol. 49, no. 4, pp. 812–826, 2014.

[50] KHRONOS Group. (2015, Febrauary) WebGL 2 Specification. [Online]. Available: https://www.khronos.org/registry/webgl/specs/latest/2.0/

[51] M. Kim, S. Ki, Y. Seo, J. Park, and C. Jhon, "Dynamic rendering quality scaling based on resolution changes," *IEICE TRANSACTIONS on Information and Systems*, vol. 98, no. 12, pp. 2353–2357, 2015.

[52] J. Kruger and R. Westermann, "Acceleration techniques for GPU-based volume rendering," in *Proceedings of the 14th IEEE Visualization 2003 (VIS'03).* IEEE Computer Society, 2003, p. 38.

[53] H. A. Lagar-Cavilla, N. Tolia, M. Satyanarayanan, and E. de Lara, "VMM-independent graphics acceleration," in *Proceedings of the 3rd International Con-*

ference on Virtual Execution Environments, ser. VEE '07. New York, NY, USA: ACM, 2007, pp. 33–43.

[54] G. Lavoué, L. Chevalier, and F. Dupont, "Streaming compressed 3D data on the web using JavaScript and WebGL," in *Proceedings of the 18th international conference on 3D web technology.* ACM, 2013, pp. 19–27.

[55] P. L'Ecuyer, "Random numbers for simulation," *Commun. ACM*, vol. 33, no. 10, pp. 85–97, oct 1990.

[56] C.-M. Lin, J.-H. Lin, C.-R. Dow, and C.-M. Wen, "Benchmark Dalvik and native code for Android system," in *Innovations in Bio-inspired Computing and Applications (IBICA), 2011 Second International Conference on*, Dec 2011, pp. 320–323.

[57] Y. Liu, C. Xu, and S.-C. Cheung, "Characterizing and detecting performance bugs for smartphone applications," in *Proceedings of the 36th International Conference on Software Engineering.* ACM, 2014, pp. 1013–1024.

[58] Y. Livny, Z. Kogan, and J. El-Sana, "Seamless patches for GPU-based terrain rendering," *The Visual Computer*, vol. 25, pp. 197–208, 03 2009.

[59] D. Luebke, M. Reddy, J. D. Cohen, A. Varshney, B. Watson, and R. Huebner, *Level of Detail for 3D Graphics.* San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2003.

[60] J. D. MacDonald and K. S. Booth, "Heuristics for ray tracing using space subdivision," *The Visual Computer*, vol. 6, no. 3, pp. 153–166, 1990.

[61] J. Mailen Kootsey, D. Siriphongs, and G. McAuley, "Building interactive simulations in a web page design program," in *Engineering in Medicine and Biology Society, 2004. IEMBS '04. 26th Annual International Conference of the IEEE*, vol. 2, Sept 2004, pp. 5166–5168.

[62] C. Marion and J. Jomier, "Real-time collaborative scientific WebGL visualization with WebSocket," in *Proceedings of the 17th international conference on 3D web technology*. ACM, 2012, pp. 47–50.

[63] C. Marrin, "WebGL specification," *Khronos WebGL Working Group*, 2011.

[64] J. K. Martinsen, H. Grahn, and A. Isberg, "A comparative evaluation of JavaScript execution behavior," in *Web Engineering*. Springer, 2011, pp. 399–402.

[65] J. McCutchan. (2013, March) Dart VM uses more CPU features for faster performance. Dartlang. [Online]. Available: http://news.dartlang.org/2013/03/dart-vm-uses-more-cpu-features-for.html

[66] ——. (2013) Using simd in dart. [Online]. Available: https://v1-dartlang-org.firebaseapp.com/articles/dart-vm/simd

[67] T. McMullen and K. Hawick, "Procedural generation of terrain within highly customizable JavaScript graphics utilities for WebGL," in *Proc. 10th Int. Conf. on Modeling, Simulation and Visualization Methods (MSV'13)*, 2013.

[68] ——, "Meaningful touch and gestural interactions with simulations interfacing via the dart programming language," in *Proceedings of the International Conference on Modeling, Simulation and Visualization Methods (MSV'14)*, 2014, p. 1.

[69] ——, "Procedural generation of landscapes for interactive environments using the dart programming language," 2014.

[70] T. McMullen, K. Hawick, V. Preez, and B. Pearce, "Graphics on web platforms for complex systems modelling and simulation," in *Proc. International Conference on Computer Graphics and Virtual Reality (CGVR'12)*, 2012, pp. 83–89.

[71] S. Melax, "A simple, fast, and effective polygon reduction algorithm," *Game Developer*, vol. 11, pp. 44–49, 1998.

[72] P. Micikevicius, "3D finite difference computation on GPUs using CUDA," in *Proceedings of 2nd workshop on general purpose processing on graphics processing units.* ACM, 2009, pp. 79–84.

[73] T. Mikkonen and A. Taivalsaari, "Using javascript as a real programming language," Mountain View, CA, USA, Tech. Rep., 2007.

[74] G. S. P. Miller, "The definition and rendering of terrain maps," in *Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques*, ser. SIGGRAPH '86. New York, NY, USA: ACM, 1986, pp. 39–48.

[75] R. Milner, *An algebraic definition of simulation between programs.* Citeseer, 1971.

[76] J. C. Mogul, F. Douglis, A. Feldmann, and B. Krishnamurthy, "Potential benefits of delta encoding and data compression for http," in *ACM SIGCOMM Computer Communication Review*, vol. 27, no. 4. ACM, 1997, pp. 181–194.

[77] S. Mohanty, S. R. Dey *et al.*, "Dart evolved for web-a comparative study with javascript," in *IJCA Proceedings on International Conference on Emergent Trends in Computing and Communication (ETCC-2014)*, no. 1. Foundation of Computer Science (FCS), 2014, pp. 73–77.

[78] G. E. Moore, "Cramming more components onto integrated circuits," *Proceedings of the IEEE*, vol. 86, no. 1, pp. 82–85, Jan 1998.

[79] L. Motion. (2018) Api overview. [Online]. Available: https://developer.leapmotion.com/documentation/csharp/devguide/Leap_Overview.html

[80] Mozilla Corporation. (2018) The WebGL API: 2D and 3D graphics for the web. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/WebGL_API

[81] J.-H. Nah, Y.-S. Kang, K.-J. Lee, S.-J. Lee, T.-D. Han, and S.-B. Yang, "MobiRT: an implementation of OpenGL ES-based CPU-GPU hybrid ray tracer for mobile devices," in *ACM SIGGRAPH ASIA 2010 Sketches.* ACM, 2010, p. 50.

[82] K. Nasim and Y. J. Kim, "Physics-based interactive virtual grasping," in *Proceedings of HCI Korea.* Hanbit Media, Inc., 2016, pp. 114–120.

[83] H. Ng and R. Grimsdale, "Computer graphics techniques for modeling cloth," *Computer Graphics and Applications, IEEE*, vol. 16, no. 5, pp. 28–41, Sep 1996.

[84] M. Nieβner, B. Keinert, M. Fisher, M. Stamminger, C. Loop, and H. Schäfer, "Real-time rendering techniques with hardware tessellation," *Comput. Graph. Forum*, vol. 35, no. 1, pp. 113–137, Feb. 2016. [Online]. Available: https://doi.org/10.1111/cgf.12714

[85] P. H. Nils Thuerey, *Shallow Water Equations.*

[86] V. Norgren. (2014) leap_motion 2.2.0. [Online]. Available: https://pub.dartlang.org/packages/leap_motion

[87] Oculus. Guidelines for vr performance optimization. [Online]. Available: https://developer.oculus.com/documentation/pcsdk/latest/concepts/dg-performance-guidelines/

[88] OpenGL Wiki, "Fixed function pipeline OpenGL wiki," 2015, [Online; accessed 6-May-2018]. [Online]. Available: http://www.khronos.org/opengl/wiki_opengl/index.php?title=Fixed_Function_Pipeline

[89] ——, "Fragment OpenGL Wiki," 2017, [Online; accessed 6-May-2018]. [Online]. Available: http://www.khronos.org/opengl/wiki_opengl/index.php?title=Fragment

[90] Oracle. (2018) Java client roadmap update. [Online]. Available: https://www.oracle.com/technetwork/java/javase/javaclientroadmapupdate2018mar-4414431.pdf

[91] J. Owens, "GPU architecture overview," in *ACM SIGGRAPH 2007 courses.* ACM, 2007, p. 2.

[92] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips, "GPU computing," *Proceedings of the IEEE*, vol. 96, no. 5, pp. 879–899, May 2008.

[93] N. H. Packard and S. Wolfram, "Two-dimensional cellular automata," *Journal of Statistical Physics*, vol. 38, no. 5-6, pp. 901–946, 1985.

[94] T. Parisi, *WebGL: up and running.* O'Reilly Media, Inc., 2012.

[95] K. Perlin, *GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics*, 1st ed. Addison-Wesley, 2004, ch. Implementing improved Perlin noise, pp. 409–416.

[96] M. Pharr and R. Fernando, Eds., *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*, 1st ed. Addison-Wesley Professional, 2005.

[97] M. Pharr, W. Jakob, and G. Humphreys, *Physically based rendering: From theory to implementation.* Morgan Kaufmann, 2016.

[98] M. Pilgrim, *HTML5: up and running.* O'Reilly Media, Inc., 2010.

[99] D. Playne, K. Hawick, and M. Johnson, "Simulating and benchmarking the shallow-water fluid dynamical equations on multiple graphical processing units," *Parallel and Distributed Computing 2014*, p. 29, 2014.

[100] V. Preez, B. Pearce, K. Hawick, and T. McMullen, "Human-computer interaction on touch screen tablets for highly interactive computational simulations," in *Proc. International Conference on Human-Computer Interaction*, 2012, pp. 258–265.

[101] A. Prokopec and M. Odersky, "Isolates, channels, and event streams for composable distributed programming," in *2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)*. ACM, 2015, pp. 171–182.

[102] K. Pulli, "New APIs for mobile graphics," in *Multimedia on Mobile Devices II*, vol. 6074. International Society for Optics and Photonics, 2006, p. 607401.

[103] A. Rauschmayer. (2012) Javascript myth: Javascript needs a standard bytecode. [Online]. Available: http://www.2ality.com/2012/01/bytecode-myth.html

[104] N. Rego and D. Koes, "3Dmol.js: molecular visualization with WebGL," *Bioinformatics*, vol. 31, no. 8, pp. 1322–1324, 2015.

[105] G. Richards, S. Lebresne, B. Burg, and J. Vitek, "An analysis of the dynamic behavior of javascript programs," *SIGPLAN Not.*, vol. 45, no. 6, pp. 1–12, jun 2010.

[106] S. Samuel and S. Bocutiu, *Programming Kotlin.* Packt Publishing, 2017.

[107] M. Schoeberl, S. Korsholm, C. Thalinger, and A. Ravn, "Hardware objects for java," in *Object Oriented Real-Time Distributed Computing (ISORC), 2008 11th IEEE International Symposium on*, May 2008, pp. 445–452.

[108] M. Segal and K. Akeley, "The opengl graphics system: A specification (version 1.1)," 1999.

[109] Y. Shi, K. Casey, M. A. Ertl, and D. Gregg, "Virtual machine showdown: Stack versus registers," *ACM Trans. Archit. Code Optim.*, vol. 4, no. 4, pp. 2:1–2:36, jan 2008.

[110] D. Shreiner, G. Sellers, J. M. Kessenich, and B. M. Licea-Kane, *OpenGL programming guide: The Official guide to learning OpenGL, version 4.3.* Addison-Wesley, 2013.

[111] D. Shreiner and The Khronos OpenGL ARB Working Group, *OpenGL programming guide: the official guide to learning OpenGL, versions 3.0 and 3.1.* Pearson Education, 2009.

[112] K. Simpson, *You Don't Know JS: ES6 & Beyond.* O'Reilly Media, Inc., 2015.

[113] J. E. Smith and R. Nair, "The architecture of virtual machines," *Computer*, vol. 38, no. 5, pp. 32–38, May 2005.

[114] K. Sowizral, K. Rushforth, and H. Sowizral, *The Java 3D API Specification*, 1st ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1997.

[115] J. Spitzer, "OpenGL performance tuning," in *NVIDIA Corporation, GameDevelopers Conference*, 2003.

[116] M. Stal, "Web services: Beyond component-based computing," *Commun. ACM*, vol. 45, no. 10, pp. 71–76, oct 2002.

[117] A. Taivalsaari, T. Mikkonen, M. Anttonen, and A. Salminen, "The death of binary software: End user software moves to the web," in *Creating, Connecting and Collaborating through Computing (C5), 2011 Ninth International Conference on.* IEEE, 2011, pp. 17–23.

[118] A. Taivalsaari, T. Mikkonen, D. Ingalls, and K. Palacz. (2008) Web browser as an application platform:the lively kernel experience.

[119] G. Tavares, "WebGL techniques and performance," in *Google I/O*, 2011.

[120] The Computer Language Benchmarks Game. (2017) Dart programs versus node.js. [Online]. Available: https://benchmarksgame-team.pages.debian.net/benchmarksgame/compare/dart-node.html

[121] D. Thevenin and J. Coutaz, "Plasticity of user interfaces: Framework and research agenda." in *Interact*, vol. 99, 1999, pp. 110–117.

[122] K. Walrath and S. Ladd, *Dart: Up and Running.* O'Reilly Media, Inc., 2012.

[123] WebGL Public Wiki, "WebGL and OpenGL Differences WebGL Public Wiki," 2014, [Online; accessed 6-May-2018]. [Online]. Available: http://www.khronos.org/webgl/wiki_1_15/index.php?title=WebGL_and_OpenGL_Differences

[124] F. Weichert, D. Bachmann, B. Rudak, and D. Fisseler, "Analysis of the accuracy and robustness of the leap motion controller," *Sensors*, vol. 13, no. 5, pp. 6380–6393, 2013.

[125] C. R. Wren, A. Azarbayejani, T. Darrell, and A. P. Pentland, "Pfinder: Real-time tracking of the human body," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 19, no. 7, pp. 780–785, 1997.