Lappeenranta University of Technology
School of Engineering Science
Degree Program in Computer Science

Joona Hasu

Fundamentals of Shaders with Modern Game Engines

Examiners: Professor Jari Porras
Dr. Sc. (Tech) Jani Rönkkönen

Supervisors: Professor Jari Porras
Dr. Sc. (Tech) Jani Rönkkönen

ABSTRACT

Lappeenranta University of Technology
School of Engineering Science
Degree Program in Computer Science

Joona Hasu

Fundamentals of Shaders with Modern Game Engines

Master's Thesis

100 pages, 24 figures, 16 code snippets, 8 tables, 4 appendices

Examiners: Professor Jari Porras
Dr. Sc. (Tech) Jani Rönkkönen

Keywords: PixiJS, Defold, Unity, Shaders, Computer Graphics

This thesis took a broad look at fundamentals of shaders in modern game engines in order to provide information about their usability in three game engines, which are currently in use by Seepia Games. The first objective of this thesis was to find out how usable shaders are in Unity, Defold and PixiJS. This was achieved by comparisons, tests and implementations. The second objective was to make a deeper dive to test how usable shaders are in PixiJS. Examination and benchmarking were done by comparing filter and shader approaches against the original version. Third objective was to find out how usable visual tools are for shader creation using Unity. This was tested by creating shaders normally and using the Shader Graph tool and comparing the methods and results. The results of the thesis are the following: 1. Shaders are usable in all of the game engines of which Unity provides the best support. 2. Shaders are usable in PixiJS, but somewhat laborous and heavy use causes performance issues. 3. Visual tools are good for shader creation because they enable faster prototyping and lower the ceiling for learning shader development.

TIIVISTELMÄ

Lappeenrannan Teknillinen Yliopisto

School of Engineering Science

Tietotekniikan Koulutusohjelma

Joona Hasu

Fundamentals of Shaders with Modern Game Engines

Diplomityö

2018

100 sivua, 24 kuvaa, 16 koodin pätkää, 8 taulukkoa, 4 liitettä

Työn tarkastajat: Professori Jari Porras

TkT Jani Rönkkönen

Hakusanat: PixiJS, Defold, Unity, Varjostimet, Tietokonegrafiikka

Keywords: PixiJS, Defold, Unity, Shaders, Computer Graphics

Tämä opinnäytetyö käy läpi laaja-alaisesti varjostimien perusteet moderneissa pelimoottoreissa. Tarkoituksena oli saada informaatiota varjostimien käytettävyydestä kolmessa pelimoottorissa, joita Seepia Games käyttää tällä hetkellä. Tämän opinnäytetyön ensimmäinen tavoite oli selvittää, kuinka käyttökelpoisia varjostimet ovat Unity, Defold ja PixiJS pelimoottoreissa. Tämä saavutettiin vertailuilla, testeillä ja toteutuksilla. Toinen tavoite oli tehdä syventävää testausta kuinka käyttökelpoisia varjostimet ovat PixiJS-pelimoottorissa. Tarkastelu ja suorituskyvyn mittaaminen suoritettiin vertaamalla filter ja shader lähestymistapaa alkuperäiseen versioon. Kolmas tavoite oli selvittää kuinka käyttökelpoiset visuaaliset työkalut ovat varjostimien kehittämiseen käyttäen Unity-pelimoottoria. Tämä saavutettiin tekemällä varjostimia normaalisti sekä käyttämällä Shader Graph työkalua ja vertaamalla toteutustapoja ja tuloksia. Työn tuloksena selvisi: 1. Varjostimet ovat käytettävissä kaikissa pelimoottoreissa, joista Unity tarjoaa parhaan tuen. 2. Shaderit toimivat PixiJS:ssä, mutta ovat työläitä käyttää ja paljon käytettynä aiheuttavat suorituskyky ongelmia. 3. Visuaaliset työkalut osoittautuivat hyödyllisiksi, koska ne mahdollistivat nopean prototyyppauksen ja alensivat varjostimien opettelun kynnystä.

ACKNOWLEDGEMENTS

First and foremost, I would like to express my thanks to my thesis supervisors Professor Jari Porras and Dr. Jani Rönkkönen for their excellent guidance. I am also grateful for my family for their long-term support and encouragement during my many years of studies.

TABLE OF CONTENTS

LIST OF SYMBOLS AND ABBREVIATIONS

2D	Two-dimensional
3D	Three-dimensional
API	Application Programming Interface
AR	Augmented Reality
BRDF	Bidirectional Reflectance Distribution Function
CG	C for Graphics
CPU	Central Processing Unit
ESA	Entertainment Software Association
FOV	Field of View
GLSL	OpenGL Shading Language
GPGPU	General-purpose computing on graphics processing units
GPU	Graphics Processing Unit
HD RP	High Definition Render Pipeline
HLSL	High Level Shading Language
HTML5	Hypertext Markup Language 5
LW RP	Lightweight Render Pipeline
OpenCL	Open Computing Language
OpenGL	Open Graphics Library
OpenGL ES	OpenGL for Embedded Systems
PC	Personal Computer
PBR	Physically Based Rendering
PBS	Physically Based Shading
RAM	Random Access Memory
RGB	Red Green Blue
SH	Spherical Harmonics
SPIR-V	Standard Portable Intermediate Representation
SRP	Scriptable Render Pipeline
TCS	Tessellation Control Shader
TES	Tessellation Evaluation Shader
VR	Virtual Reality

VRAM	Video Random Access Memory
WebGL	Web Graphics Library

1 INTRODUCTION

Today's game industry is massive. It provides rich and engaging entertainment for players of all ages across all platforms. Entertainment Software Association's (ESA) 2017 report [1], shows how video games have evolved into a mass medium. More than 150 million Americans play video games and the biggest factor influencing decisions to purchase games was quality of graphics with 67%.

History of shaders starts somewhere around 1980s. In 1977, Star Wars Episode IV: A New Hope used small amounts of computer graphics, and it is one of the starting points in history of shaders. After the success of the movie George Lucas started Computer Division of Lucasfilm, which was tasked to develop digital editing and composition, hardware for two-dimensional (2D) image processing and for three-dimensional (3D) graphics rendering. Later in 1983 both the 2D and 3D groups of the Computer Division were divided into a new company called Pixar. This led to Pixar dropping hardware development and start working on general-purpose software solution called Photorealistic RenderMan. However, at the same time multiple individuals and groups develop different software and hardware solutions. For example, in 1985 Perlin published Image Synthesizer paper [2] and created surface shading functions with expressions and flow control. Later in the early 2000s, NVIDIA developed C for Graphics (cg), Microsoft made High Level Shading Language (HLSL) and OpenGL Architecture Review Board created OpenGL Shading Language (GLSL). [3]

Capabilities of today's hardware enables developers to create beautiful games using physically based rendering (PBR) and other techniques resulting in photorealistic game worlds and impressive effects. There are of course still limitations on what computers are capable of, therefore in some cases developers need to be creative to achieve great looking effects without sacrificing much performance. Mobile devices and even consoles are still lacking in processing power compared to desktop computers with the latest components. However, creative and innovative ways of using, for instance, shaders in combination with other visual effects can create great looking games for these devices. This is one of the inspirations behind this thesis.

Shaders define how object looks and how it reacts to the light. Usually light calculations are coded to the shader and there are many configurations and techniques of how they can be used. Nowadays shaders are also used to create special effects and even calculations using the parallel powers of Graphics Processing Units (GPUs). Today's game engines support modern shaders. Unity and Unreal Engine, for example, support HLSL shaders. Although Unity uses their own ShaderLab language, it supports HLSL shaders and even GLSL shader snippets. Unity also has Surface Shaders, which provide lightning, shadowing, light mapping and forward and deferred rendering automatically. They provide generated shader code that would otherwise be repetitive code, because the implementations for these features would be the same across different shaders. This thesis uses multiple game engines and shader languages. The chosen game engines for this thesis are the ones that are currently in use or used in the recent past by Seepia Games. These are PixiJS, Defold and Unity.

1.1 Goals and Delimitations

The goal of this thesis is to research about the fundamentals of shaders and how they can be used in different game engines. This thesis is comprised of three research questions. They are the following:

1. How are shaders used in the current game engines?
2. How usable are shaders in PixiJS?
3. How usable are visual scripting tools for shader development in Unity?

This thesis focuses on finding out how shaders are used in PixiJS, Defold and Unity, and examining how usable shaders are currently in PixiJS and what visual scripting tools in Unity provide in terms of shader creation. This thesis describes simple shader programs to provide tools for comparisons between the game engines. It does not describe shaders that are optimized for given task or provide information about best practises on shader development. Readers are expected to have basic knowledge about game engines and programming.

1.2 Structure of the Thesis

This thesis consists of ten chapters. Chapter two contains the scientific literature for this thesis. The third chapter takes a look at modern graphics APIs and game engines. The basics of rendering and graphics are investigated in chapter 4. The fifth chapter introduces the basics of shaders. In the sixth chapter the comparisons of shader usages between the selected game engines are described, and chapter seven contains the shader implementations done for projects and for the comparisons. In the chapter eight the results of shader usage in PixiJS is discussed along with discussion about visual tools usage in Unity. The ninth and tenth chapters contain the discussion, conclusion and the summary of this thesis.

2 LITERATURE REVIEW

Proper understanding of shaders requires wide range of knowledge, all the way from graphics pipelines to game engines and different lightning models that are used in game development. Good theoretical basis will also help in the creation of shaders that are actually usable in the real world.

There are many different game engines and they all have different ways of shader creation. However, there are also many similarities between them. Having a good grasp of GLSL shader basics translates well into other languages in the long run. For example, there are many tools that can convert shaders from language to another.

2.1 Literature Search

To achieve proper understanding about shaders a deep dive in to the theoretical side was needed. Additionally, investigation to what others have studied and researched was done to get good references for this thesis.

For the literature review, good keywords were selected. These were: Shaders, GLSL, HLSL, ShaderLab, Unity, Optimize Unity Shaders, Unity HLSL, Unity GLSL, Defold, Defold Shaders, PixiJS and PixiJS Shaders. Other keywords were also used. However, they were only to test what kind of resources could be found. Ultimately these were discarded as they gave similar results as the currently selected keywords or the results were unrelated to the topic.

The included sources contain knowledge about the basic theory about shaders using GLSL and HLSL, basics about Unity's ShaderLab language, rendering pipelines, different lightning techniques used in games and examples of different kinds of shaders. Most of the sources chosen to this thesis were found from the well-known computer science libraries such as IEEE Xplore, ACM DL and Science Direct. Google Scholar was used as the initial search tool for resources.

Other highly used resources in this thesis are Graphics Shaders: Theory and Practice (Second Edition) by Mike Bailiey and Steve Cunningham from CRC Press [3] and Unity 5.x Shaders and Effects Cookbook by Alan Zucconi and Kenneth Lammers from Pack Publishing [4]. Also, the documentation of PixiJS [5], Unity [6] and Defold [7] engines were valuable in writing this thesis. All in all, 4 conference and journal articles were selected as sources.

2.2 Data extraction

In their article, Humberto Andrade and Fernando Ramos [8] propose several alternatives to classical lightning techniques. Their proposed methods achieve higher performance with similar visual results when they, for instance, compared it to Phong lightning technique which is popular in modern games. The proposed specular lightning techniques produce diffuse-lit colours which are more accurate than the traditional lightning techniques. These methods could be useful in performance-critical applications. However, these techniques have a minor flaw; the light's reflection actually goes back to the light's origin, which is not how light in real world works, but it looks realistic enough in some use cases.

Background: Physically-Based Shading by Naty Hoffman [9] discusses the basics and the backgrounds of Physically Based Shading (PBS). The author explains the basics of physics and mathematics of shading and how light interacts with different kinds of matter. This provides great insights for this thesis, because it explains the fundamentals of PBS, which are many times glanced over or just mentioned and expected to be known by different shader tutorials and books. The article explains many basic concepts that are required to understand in shader programming: Bidirectional Reflectance Distribution Function (BRDF), different reflectance terms and how these mathematical models are implemented in movies and games. While all these are not usable in every scenario, for instance PBS is seldom used in 2D games, they are still valuable techniques to know and understand, especially now that many of the popular game engines support PBS.

Crafting Physically Motivated Shading Models for Game Development by Naty Hoffman [10] explains what practical advantages there are for physically based shading. He discusses how physically-based shading models allow developers to achieve photorealistic and cinematic looks easier. These models are also easier to troubleshoot. However, the used

game engine needs to have several basic features to physically-based models to work properly. For example, shading needs to happen in linear space with gamma-correct rendering. The chosen engine should also be able to support high dynamic range with lightning and shading without forgetting proper tone mapping support. Other problems that may arise, when dealing with physically-based models are mostly caused by poor selection of used techniques. In many cases games can get away with using alternative techniques that do not look as realistic, but usually players do not notice the difference.

Physically Based Shading Models in Film and Game Production: Practical Implementation at tri-Ace by Yoshiharu Gotanda [11] presents practical examples of physically shaded models using their own implementations as examples. Usually in different implementations artists can set material parameters incorrectly due to how human eye perceives real world light differently than their display devices represent them. Author's studio's artists had this problem when using ad hoc shading models as they were dealing with physically correct materials. He also introduces their customized Blinn-Phong and Ambient BRDF models and presents how these modifications improve their materials and help in their workflow. The customized Blinn-Phong can't handle reciprocity nor roughness, but by using approximation in their diffuse calculations they achieve good performance without having too much of a visual quality loss. Usually, simple ambient light implementations have constant ambient term, but using the Ambient BRDF model the shader regards the ambient lighting as area light. This provides clearer differences between materials which are covered by the ambient light. In combination with their custom application to create textures, these models provide better visual results without big performance reduction.

3 MODERN GRAPHICS

Computer graphics are made of many components. On high level the game engine has the information about game objects, which are usually made of models or sprites, and knows how they should be rendered. In order to do so, it needs a way to communicate with the graphics hardware. This is done via Graphics Application Programming Interfaces (Graphics APIs). Depending on the engine and the operating system there are few different APIs that can be used. This chapter provides high level overview of the selected engines and the currently used graphics APIs.

3.1 Graphics APIs

Graphics APIs are program libraries that are designed to render graphics to the display. They are mostly used in applications, simulations and games because they provide access to graphics processing unit's (GPU) hardware-accelerated rendering capabilities. Currently there are four different graphics APIs that are commonly used in games: OpenGL, Direct3D, Metal and Vulkan.

3.1.1 DirectX

Direct3D is a proprietary 3d graphics library developed by Microsoft. It is part of DirectX API Libraries and used in Xbox consoles and Windows operating systems. DirectX libraries also include, for example, Direct2D and DirectSound. In the recent years there has been a shift towards APIs that provide lower level access to the graphics hardware and the latest version of DirectX, which is DirectX 12, provides this. DirectX 11 is still considered to be a modern and low-level API and it is still used widely by games and game engines alike. Major downside for using DirectX 12 is that it is only supported currently by Windows 10 and Xbox One and is thus yet to see similar adoption to DirectX 11 by the gaming industry. Shaders used in DirectX APIs are written in the HLSL shading language. [12]

3.1.2 OpenGL

Open Graphics Library (OpenGL) [13, 14] is a cross platform 2D and 3D graphics API developed initially by Silicon Graphics and currently by Khronos Group. While Direct3D is locked to the Microsoft platforms, OpenGL has spread all the way from Linux to Android, macOS, iOS and Windows. However, OpenGL is being deprecated in the newest operating systems in the Apple ecosystem [15, 16]. By features OpenGL is comparable to DirectX 11 version of Direct3D (and by extension Direct2D). Current version of OpenGL is 4.6 and shaders written for OpenGL are created in the GLSL shading language.

3.1.3 Vulkan

Vulkan [17, 18] is Khronos Group's new cross platform graphics and compute API to compete with DirectX 12. It is based on AMD's Mantle API, which was donated to Khronos Group as basis for their new lower-level API. It is comparable to DirectX 12 in features and in the low-level access to the GPU. While it is cross platform API, it is not officially supported by macOS or iOS. However, MoltenVK [19], developed by Brenwill Workshop Ltd, is an implementation of Vulkan that aims to bring Vulkan support to macOS and iOS. It maps Vulkan to Apple's own Metal graphics API so that developers can use Vulkan and not rewrite their applications to use Metal. Shaders work slightly differently in Vulkan than in other graphic APIs. The shader code that is consumed is written in bytecode format called SPIR-V (Standard Portable Intermediate Representation) [20], which is also used in OpenCL (Open Computing Language). However, shader creators do not have to write bytecode because there are, for example, compilers that compile GLSL or HLSL [21] shaders into SPIR-V bytecode.

3.1.4 Metal

Apple has developed their own graphics API called Metal. It's latest version Metal 2 provides near-direct access to GPU in a similar way to DirectX 12 and Vulkan. It's also used in iOS and tvOS devices. Shaders for Metal are written in Metal Shading Language [22]

3.2 Game Engines

Depending on the use case, there are multitude of game engines to choose from. For developers seeking for an open source game engine, Godot and PixiJS are good examples. Godot provides many features including 2D and 3D rendering and visual scripting. On the other hand, PixiJS is excellent choice for 2D games and applications targeting Hypertext Markup Language 5 (HTML5) based platforms. Major commercial engines like Unity, Unreal Engine and CryEngine provide the latest and greatest technical features for the most use cases. Unity and Unreal Engine provide excellent multiplatform support and features while CryEngine platform support is focused on current consoles, Personal Computers (PC) and Virtual Reality (VR) devices. There are also many game engines that are developed and used only by game studios inhouse.

3.2.1 PixiJS

PixiJS [23, 5] is a JavaScript rendering library developed by Goodboy Digital Ltd. It is designed to be cross platform framework for websites, applications and HTML5 games. It provides support for Web Graphics Library (WebGL) and uses HTML5 canvas as a fallback if needed. It also provides solid 2D performance and small build size. With the rising popularity of instant games and playable ads and their focus on HTML5 and JavaScript technologies PixiJS becomes one of the more interesting alternatives for game development [24, 25]. While it is a rendering library, in this thesis it will be considered a game engine to keep things simpler.

3.2.2 Defold

Defold [26, 27, 28] is a cross platform game engine with a focus on performance. It was initially developed by Ragnar Svensson and Christian Murray in 2009, which was later bought by King Digital Entertainment Ltd in 2014. Their editor and tools support macOS, Windows and Linux, while their target platforms currently include iOS, Android, macOS, Windows and Linux. It supports both 2D and 3D rendering and physics using Box2D and Bullet physics engines. The scripting language used in Defold is Lua. It also provides fully

scriptable rendering pipeline with OpenGL for Embedded Systems (OpenGL ES) 2.0 shader support. While it supports both 2D and 3D rendering, most of their toolsets are developed 2D in mind.

3.2.3 Unity

Unity [29] is a game engine created by Unity Technologies to create 2D, 3D, VR and Augmented Reality (AR) games and applications. Unity Technologies [30] was founded in 2004 by Nicholas Francis, David Helgason and Joachim Ante.

Unity editor supports Windows and Mac and provides preliminary support for Linux. The editor supports 2D and 3D development with specific support and features for both. For example, for 2D physics Unity uses Box2D and for 3D it provides NVIDIA PhysX physics engine. It also has built-in support for UI development, cinematic cut-scenes and in-game sequences, particles, animation and audio mixing and mastering. Unity supports all the major graphics APIs like Vulkan, Metal and DirectX 12 and few VR specific ones like AMD LiquidVR [31]. The scripting language for Unity is C#. [32, 33]

4 BASICS OF RENDERING AND GRAPHICS

This chapter provides basic information on how rendering and graphics typically work in game engines. This includes information about coordinate systems used in game engines and how rendering pipeline draws pixels to the screen and what is view frustum and how it works.

4.1 Coordinate Systems

Usually 3d application uses either left-handed or right-handed Cartesian coordinate system. In both systems x- and y-axes are similar but in the case of the left-handed system positive z-axis is pointing away from the viewer, while in the right-handed system it is pointing towards the viewer. This is illustrated in the Figure 1.

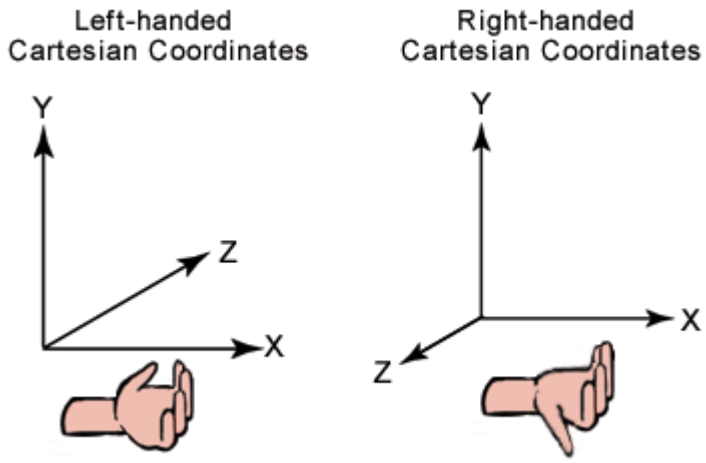

Figure 1. Left- and right-handed Cartesian coordinate systems [34]

Microsoft Direct3D uses left-handed system while OpenGL uses right-handed coordinate system [34, 35]. However, applications and engines may use their own set of coordinate systems independent of graphics APIs used. For example, 3D modelling software Blender and Autodesk 3ds Max use right-handed coordinate system, while game engines Unity and Unreal Engine use left-handed coordinate system [36].

4.1.1 Model Coordinates

Model Coordinate system or object space coordinates are the coordinates that the model itself has from the modelling software. Model coordinates represent the model-space geometry that vertex shader can pass to tessellation, geometry or fragment shaders, which are explained in detail in section 5. These coordinates are usually used when the objects geometry affects the wanted effect or that geometry itself is going to be changed somehow. Basically, this is a local coordinate system for that object. [3, pp. 143,162-163]

4.1.2 Eye Coordinates

In eye coordinate system (or camera space) the viewer is located at the origin (0,0,0). This makes it a natural system for creating the world as seen by the viewer. Eye coordinates are transformed from model coordinates. In this coordinate system the x and y are aligned to the display, while z is parallel to the viewing direction. [3, pp. 143,162-163]

4.1.3 World Coordinates

World coordinate system is used to build the complete scene. They are transformed from model coordinates and contains positions and rotations in that world. This system ties all the coordinate systems together. [3, p. 145]

4.2 Rendering Pipeline

Game scenes include many different objects. These objects are made of vertices and graphics primitives [37, pp. 1-2]. Vertices are the points in the object where lines meet, and graphics primitives indicate how these vertices are connected and thus how the shape is produced. OpenGL supports the following primitives: points, lines, line strips, line loops, triangles, triangle strips, triangle fans, quads, quad strips and polygons. On the other hand, Direct3D supports: point lists, line lists, line strips, triangle lists, triangle strips and triangle fans (Direct3D 9 only) [38]. Among these the triangle lists are the most commonly used. In triangle list one triangle represent three points. There are also some shader specific primitives. For example, only geometry shader supports primitives with adjacency

information [39, 3]. These are, for example, lines with adjacency and triangles with adjacency. Tessellation shader also has its own primitive called patch, which is a set of points and can represent many things [3]. Shaders are discussed in more detail in the chapter 5.

When a game or a program needs something to be drawn to the screen it sends commands to the rendering library. This is executed in the Central Processing Unit (CPU). Communication to the GPU happens via GPU drivers, which translates the function calls into calls that the GPU can understand. While the GPU processes these calls it stores information to the Video Random Access Memory (VRAM) where, for example, front and back image buffers and depth buffer are stored. These are important because front image buffer contains viewport's pixel data and back image buffer contains the scene renders. Viewport is the area in the display that holds the rendered image. When the rendering is done, buffer swap is performed. This means that the front and back buffers are swapped. Depth buffer on the other hand stores a value for every pixel in the image buffer that indicates the pixels depth in the image. This information can be used to hide pixels by their depth. Creation of a stencil buffer can also be done if the application requires it. Stencil buffer holds an integer mask for every pixel in the image buffer. It can be used to disable or enable pixels. Texture maps, which are the images applied to object's surface, are also kept in the VRAM. [37, pp. 2-4]

4.2.1 Vertex Transformations

Vertex transformations are calculations that are done to the geometrical data that is passed to the GPU. This is done so that the data can be drawn into a 2D viewport. Model coordinates can be transformed to camera space by model-view transformation, in which model's space coordinates are transformed to world space and then to camera space. After this projection transformation is done. This basically means that perspective is applied to the geometry. Vertices now exist in homogeneous clip space where graphics primitives are clipped to the visible region so that no primitives are rendered outside the viewport. They also have normalized device coordinates in range [-1, 1]. Lastly, they are transformed into window space by viewport transformation, where normalized coordinates are mapped to the viewport's pixel coordinates. In the window space z-coordinates are in the range [0, 1]. Texture coordinates are supplied by the application, which are then carried with the vertex

or GPU can generate them automatically. These coordinates are then used to interpolate colours from the texture map and rendered correctly with the graphics primitive. The final colour for each pixel is the combined colour of the data. [37, pp. 4-6]

4.2.2 Rasterization

Rasterization is the process of determining the pixels which are covered by the primitive and to interpolate the output variables of the vertex shader. These variables include depth, vertex colours, texture coordinates and pixel coordinates. All this combined is called a fragment and fragment shading (or pixel shading in Direct3D) is done at this point. In fragment shading the final colour and depth for each pixel is calculated. [37, pp. 6-7]

Before fragment shading the application can perform face culling, where polygonal graphics primitives that are facing away from the camera or facing the camera are removed. This is a good optimization because the culled primitives are unseen to the camera. Fragment operations are operations that are performed before the fragment is written to the image buffer. These are to test if the fragment should be discarded before final colour is blended to the image buffer. In pixel ownership test, the fragment is tested for being in the region of the viewport which is visible in the screen. If the fragment does not lie in the viewport it is discarded. In scissor test, scissor rectangle is specified and any fragments outside that rectangle are discarded. Next is the alpha test, in which the final colour of the fragment is calculated. This alpha is then compared to a constant alpha value which is given by the application which also gives the failing parameters for this test. If the fragment has passed the tests so far, it is time for stencil test where stencil buffer's value in the fragment's location is tested against a value which is specified by the application. If the test fails, the fragment can be discarded, or the application can have alternative actions that are taken in a similar way as the test would have passed. Last test is the depth test, where fragment's value in depth buffer is checked against the application specified value. [37, pp. 7-9]

4.3 View Frustum

The view frustum is the pyramid like volume of space which contains all that is visible in the screen. Basically, it is the representation of what camera sees from the scene and the apex of the pyramid is in the cameras position. View frustum is visualized in the Figure 2 below.

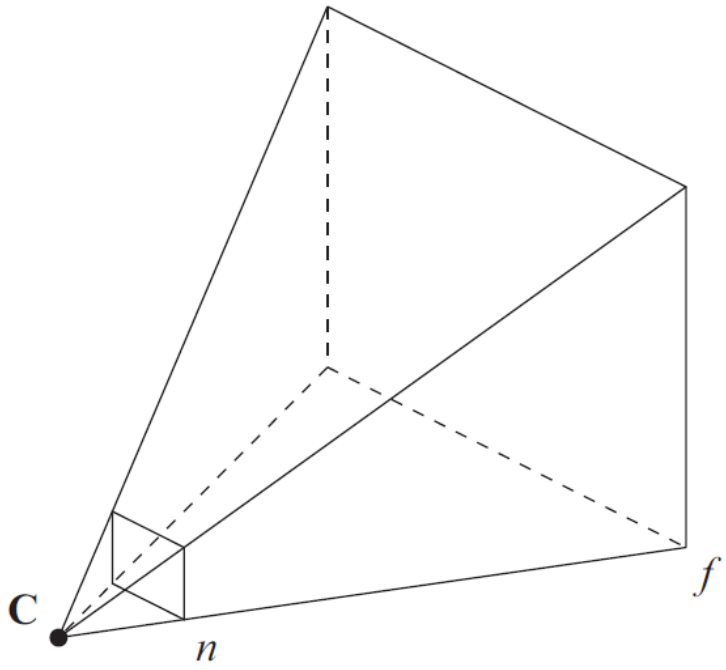

Figure 2. View frustum visualized [37].

In the figure the camera position is marked as C. Near frustum plane is at distance *n* and far frustum plane is at distance *f* from the camera. They represent the closes and farthest positions in which an object is visible. View frustum is bounded by six planes. Two of which are the near and far planes. Others are left, right, top and bottom frustum planes in the edges of the pyramid, which are between near and far clipping planes in the figure. View frustum is aligned to eye space coordinates where it is the origin and where x axis represents left and right, while y axis represents up and down. Z axis is the depth direction. [37, pp. 102-103]

4.3.1 Field of View

Horizontal Field of View (FOV) is the angle between left and right frustum planes, while vertical FOV is between top and bottom planes. Horizontal FOV affects the distance between camera and the projection plane. This is illustrated by the Figure 3 and trigonometrical relation

$$e = 1/\tan(\alpha/2)$$

where the distance is e and the horizontal FOV angle is α. On the other hand, vertical FOV is calculated using displays aspect ratio as illustrated in the Figure 3 and the it can be calculated using

$$\beta = 2\tan^{-1}(\alpha/e)$$

where β is the vertical FOV angle and α is the displays aspect ratio. Aspect ratio is calculated by dividing screen height with width. [37, pp. 103-106]

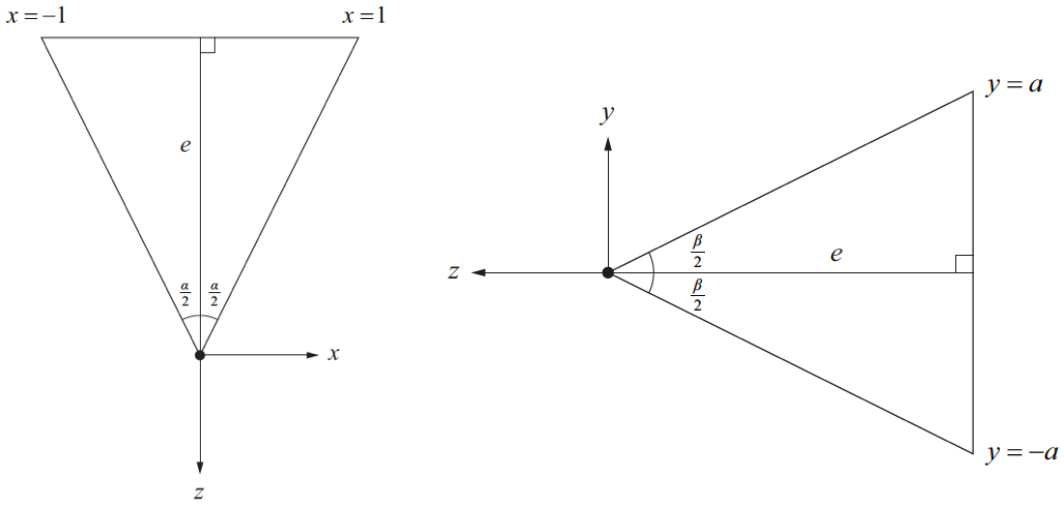

Figure 3. Horizontal and Vertical FOV: a) Horizontal b) Vertical [37]

5 SHADERS

Shaders are programs that run on the graphics pipeline. They provide necessary information for the graphics hardware so that they will be able to render each pixel correctly.

Shaders are a set of instructions which are executed all at once and to all the pixels in the screen. Basically, the shader code will work as a function that receives a position and returns a colour. Shader code will run extremely fast because it is run on the GPU rather than the CPU. While CPUs are powerful and might have multiple threads, they still do their work in series. On the other hand, GPUs work in parallel. They have lots of microprocessors and have special math functions that are accelerated via the hardware instead of being calculated in software by CPU. [40]

Modern graphics cards usually support four levels of parallelism: Device-Level Parallelism, Core-Level Parallelism, Thread-Level Parallelism and Data-level Parallelism. In Device-Level Parallelism multiple processors or multiple graphics cards can operate on the same system. Core-Level Parallelism means that processors have multiple cores which are capable of independent execution. In Thread-Level Parallelism each core can have multiple threads and Data-Level Parallelism means that there can be multiple instructions which can act on multiple data elements at once. [3, pp. 92-93]

Shaders are written in a shading language, such as GLSL or HLSL. These languages are designed to give developers more direct control of the graphics pipeline and to provide optimized support for matrix and vector data types and operations [41, 3, p. 94]. Before the introduction of the programmable shaders, fixed function pipeline was used. It provided developers with fixed set of operations for both vertices and fragments and had very limited number of features compared to current way of creating shaders [3, p. 1].

5.1 Vertex Shader

Vertex shader sets up the shader environment for vertex processing for tessellation and geometry shaders and also prepares for rasterization and fragment shader processing. It can

also do changes to the vertex coordinates. Vertex shader gets inputs that are going to the vertex processing stage of the graphics pipeline. This also includes the data that the application wants to send to shaders. Vertex shader outputs variables for the next stage of the pipeline.

Vertex shader can pass vertex coordinates to fragment shader using model-space geometry or eye-space geometry. For example, for tessellation shader, vertex shader can pass assembled primitives together with the data that controls the subdivision that tessellation shader should perform, while tessellation shaders output consists of collection of vertices for the new geometry. In the end vertex shaders mission is to pre-process vertices and manage attribute variables for the rest of the pipeline to use. One of variables it handles is lighting that depends on the vertices [3, pp. 139-141]

5.2 Tessellation Shader

Tessellation shaders interpolate geometry to create additional geometry that allows developers to perform adaptive subdivision, provide coarser models that the GPU can refine, apply detailed displacement maps without supplying equally detailed geometry, adapt visual quality using required level of detail, create smoother silhouettes and perform skinning more easily. Basically, tessellation in graphics is a process that can divide a surface into, for example, smoother mesh of triangles. Tessellation can increase the quality of the final image. [3, pp. 315-316]

Tessellation shaders have access to all the information in the graphics pipeline. Tessellation shader is able to choose tessellation parameters dynamically depending on the current information. In the shader pipeline tessellation shader comes after vertex shader. While vertex shader modifies vertices individually and has no reference to their primitives, tessellation shader amplifies a single primitive. [3, p. 316]

Tessellation shader consists of two different types: Tessellation Control Shader (TCS) and Tessellation Evaluation Shader (TES). TCS prepares the control points and controls how much tessellation should be done. Between these shaders there is a Tessellation Patch Generator (TPG) fixed-function stage, which is not programmable but takes in parameters.

It creates tessellated primitives using tessellation levels set by TCS and outputs their positions to TES in barycentric (u, v, w) coordinates. TES then reads both the outputs from TCS and TPG and calculates output coordinates (x, y, z) and if there are any attributes it interpolates them and finally applies any displacements. TCS is optional, but in this case some of the values otherwise outputted by TCS need to be set in the application. In DirectX TCS and TES are named Hull and Domain Shaders respectively and TPG is called Tessellator Stage [42]. [3, pp. 318-320]

5.3 Geometry Shader

Geometry shader has many uses. One of its main uses is that it can expand the basic model geometry to include different or even more primitives that was initially defined. It can generate all the familiar topology types and, for example, points, lines, triangles and quads. It can also use primitives with adjacent information [43, 3]. However, geometry shaders can only accept limited number of topologies. As an output geometry shader generates points, line strips or triangle strips and feeds them to the rest of the graphics pipeline. [3, pp. 291-293]

Usable variables for geometry shader are passed by a primitive assembly step once there are enough vertices for the current topology type. Basically, geometry shader will use vertex and tessellation shader products from geometry parts all the way to appearance parts. Ultimately geometry shader uses these as parts of the primitives it assembles. Vertices that the geometry shader generates are then assembled, rasterized and further processed in the remaining graphics pipeline. [3]

Both the geometry shader and tessellation shader are capable of creating and modifying geometry and gives control over level of detail. After tessellation shader the geometry is still the same, while geometry shader can handle geometry topology changes. There are also limitations for using both the tessellating shader and geometry shaders, because if the geometry shader requires adjacency, tessellation shaders can’t be used because it cannot emit any geometry with adjacency. [3, pp. 317-318]

5.4 Fragment Shader

Fragment shader produces final colour for each pixel from state variables and values interpolated across a polygon. Among other things fragment shader can compute the colour and light intensity of each fragment. Fragment shader can handle different kinds of built-in properties of vertices, but two of the most important ones are texture coordinates and pixel depth. They can also interpolate texture coordinates, and these can be sampled to help determine colours for each pixel. [3, pp. 157-158]

Fragment shader takes inputs from an application or the system, but the most important inputs are passed down variables from vertex, geometry or tessellation shaders. This data is interpolated across a graphics primitive and it should include enough information to set wanted colours for each pixel. [3, pp. 158-159]

Fragment shader outputs mainly pixel colour, but it can also produce depth value for each pixel. Produced pixel colour is ready to be processed by other pixel operations and be written into the framebuffer. The produced depth is useful for computing depths that are different from the usual interpolation of the vertex depths. [3, pp. 159-163]

5.5 Compute Shader

Compute shader is a shader stage outside the normal rendering pipeline which is used almost entirely for computing arbitrary information. And is thus quite different when compared to other shaders. They are efficient for massively parallel General-Purpose computing on Graphics Processing Units (GPGPU) algorithms. The purpose of GPGPU is to use GPU to perform computations that are traditionally done using CPU. Compute shaders can even be used to accelerate some parts of the game rendering. While other shaders have well defined inputs, compute shaders do not have user defined inputs or outputs at all, the only definition being the “space” of execution for the shader’s invocation. If there is a need for inputs for the shader it must fetch the data via texture access, arbitrary image load, shader storage blocks or via other forms of interface. Additionally, if the shader computes anything it must write the data to an image or shader storage block. [44, 45]

Compute shader operates in different space than the other shaders. This space is abstract. The shader has fixed size of work groups that it has to compute, which are defined by the user. Work groups are the smallest amount of compute that the user can execute. Compute shader executes these groups in random order, therefore processing order cannot be defined. The individual invocations in each work group will be executed in parallel. These work groups can communicate through a set of shared variables and special functions. However, invocations in different invocation groups cannot communicate with each other. [44]

6 SHADERS IN GAME ENGINES

While all game engines use shaders for rendering, the way they allow developers to leverage and customize them differ greatly. Engines usually provide built-in tools and concepts like materials to contain the needed information for render objects. However, some engines do not provide any such tools and leaves the shader development and usage to the developers.

6.1 PixiJS

Shaders in PixiJS are written in GLSL shading language using OpenGL ES 2.0 Shader Language syntax. That said, PixiJS reverts to canvas renderer when the device or browser does not support WebGL. This is significant because canvas renderer does not support shaders, and this needs to be taken into account by the developers when creating games and applications with support to legacy devices.

6.1.1 Current State of PixiJS Shaders

PixiJS supports filters, which work similarly as shaders, but are internally handled differently. Alternatively, shaders can be used with PixiJS, but it currently requires writing a rendering plugin to enable support for them. While shaders could be batched and possibly more optimized, filters do not support batching currently and thus will have negative effect on performance if used extensively. Batching in game engines means making a group of draw calls that are to be drawn together rather than one at a time [46, 47]. Draw calls are basically orders from the CPU to the GPU to draw something to the screen. There are many techniques used to create batching systems and usually these are game engine specific. Writing a proper rendering plugin for shaders with batching support would be possible, but it requires good knowledge about rendering pipelines and how things are done internally in PixiJS. It was left out of the scope of this thesis, but a simplified plugin using PixiJS examples was done to test the shader workflow.

Creating filters is relatively easy in PixiJS. In the simplest case, you are only required to write a fragment shader and pass it as a parameter to the created filter, which is then applied

to a display object where the effect is visible. While passing vertex shader as *null* to the filter makes the filter to use PixiJS's default vertex shader, developers can make their own vertex shaders if they need to. Variables can be passed to the filter via its uniforms object. However, there are few caveats to using filters that must be considered. In the current version (PixiJS V4) some of the features are created preferring the performance over the usability [48]. For example, if pixel coordinates or screen coordinates are required in the filter, they must be calculated from filterArea uniform, which is passed to the shader automatically. Basic example of filter usage can be seen in the Code Snippet 1.

```
1.  let fragmentShader = `
2.  precision mediump float;
3.  uniform vec4 uColour;
4.  void main(void)
5.  {
6.          gl_FragColor = uColour;
7.  }`.split('\n').reduce( (c, a) => c + a. trim() + '\n' );
8.
9.  let filter = new PIXI.Filter(null, fragmentShader);
10. filter.uniforms.uColour = {0: 255, 1: 255, 2: 255, 3: 255};
11.
12. let container = new PIXI.Container();
13. container.filterArea = app.screen;
14. container.filters = [filter];
15. app.stage.addChild(container);
```

Code Snippet 1. Applying filter to container

In the example, starting from the line 1, is the basic fragment shader, which only applies colour to the pixels. In the line 9 filter is created and in the line 14 it is applied to the container. Colour for the filter is passed via uniform object and thus to the filter in line 10. PixiJS also supports loading shader programs, for example, *.vert* and *.frag* files can be added to the application using PixiJS's default loader *(Pixi.Loader)*. Vertex shaders can be applied to the filter using the same method as the example's *fragmentShader* by passing it as the first argument when creating the filter.

While filters provide the basic shader functionalities, there are situations when more advanced features are required. One of the reasons could be, the need for extra textures to be used in the filter. In this case, creation of the custom rendering plugin is required to provide the support. PixiJS provides good plugin example via GitHub repository [49], which

can be used as a basis for the plugin. This example and createShaderPlugin [50] were used as a basis to test renderer plugin usage in PixiJS. Using this custom plugin shaders could be used instead of filters. This simplified slightly the writing of the fragment shaders. This will be examined more closely in the implementations in section 7.1.

6.1.2 Future Possibilities

Next major release of PixiJS is version 5 and it will add support for WebGL 2 and OpenGL ES 3.0 shaders. Additionally, this update will also make canvas renderer legacy code and WebGL 1 the fall-back renderer [51]. Furthermore, there might be some rework coming relating to shaders and their usage. This will hopefully make custom shader creation more developer friendly. At the time of writing this, the latest version of PixiJS 5 was alpha 3.

6.2 Defold

Shaders in Defold are written in GLSL shading language. However, to keep cross platform compatibility with desktop and mobile builds it is encouraged to write the shaders using OpenGL ES 2.0 specification compatible syntax. Additionally, Defold supports 3D models exported from modelling software. However, it only supports meshes that are stored as Collada files. [52]

6.3 Render Pipeline

Every object in Defold is drawn by the renderer, which is basically a render script that controls the rendering pipeline. The rendering pipeline interacts mainly with the OpenGL rendering API that Defold uses. Render scripts are written in Lua and must have *init*, *update* and *on_message* functions to work, but other than that they are completely customizable. Draw order in the render pipeline is controlled by creating render predicates, which control what should be drawn based on the material tags selected. [53]

6.4 Shaders and Materials

In Defold vertex attribute and uniform variables works like one would expect from GLSL shaders. These variables are passed to the shaders through materials. They are Defold's way of expressing how the objects should be rendered. It includes references to vertex and fragments shaders and the variables passed on to the shaders. Variables passed to the vertex and fragment shader are called vertex constants and fragment constants respectfully. Textures passed to the shaders are passed as samplers. They can be named, and their wrap and filtering settings can be modified per material. Materials also hold tags, which are information for the rendering pipeline to use. Example of editor's material view in the Figure 4 below. [54]

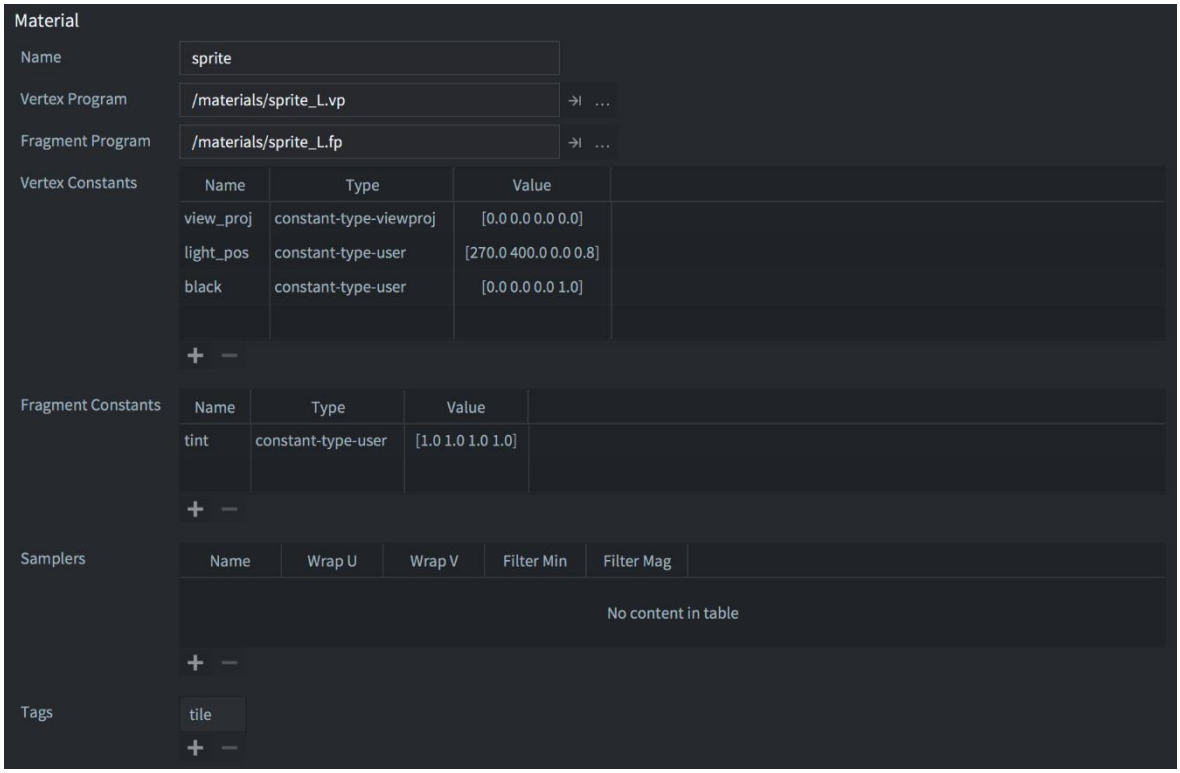

Figure 4. Screenshot of Defold editor's material view

6.5 Unity

Creating shaders for Unity can be done by writing or by using visual shader editing tools like built-in Shader Graph tool. There are also third-party tools to help with shader creation,

for instance, commercial Amplify Shader Editor provides visual shader editing tools with extensive feature list.

6.5.1 Render Pipelines

Unity's current render pipeline [55] is mostly a black box, which means that developers don't have much control over how it works. There are different rendering paths that can be chosen. These control how lighting is applied and which shader passes are used. The different paths are forward rendering, deferred rendering and legacy paths: deferred lighting and vertex lit. Forward rendering path [56] has base pass and additional passes that can be used. Basically, base pass renders object with one per-pixel directional light and up to four per-vertex lights and all Spherical Harmonics (SH) lights, which are computed approximations for lighting. Additional passes are rendered to affected object for each additional per-pixel light. Deferred rendering path [57] does not have a limit on how many lights can affect an object and all the lights are computed per-pixel making the deferred shading consistent and predictable. Then again, it does not support all the features that forward rendering does, for instance, anti-aliasing and semi-transparent objects are not fully supported.

6.5.1.1 Scriptable Render Pipeline

Scriptable render pipeline (SRP) is Unity's new way of providing developers a way of configuring rendering. On high level, scriptable rendering pipeline handles culling, rendering objects and post processing. This allows developers to take control of the rendering and making Unity's render pipeline less of a black box. [58, 55]

Unity also provides High Definition Render Pipeline (HD RP) [59] and Lightweight Pipeline (LW RP) [60]. HD RP is targeted towards PCs and latest generation consoles, while LW RP targets mobile platforms and VR. HD RP provides fully new render pipeline providing unified and coherent lighting and same feature sets for all the rendering paths and better debugging tools. LW RP is basically a subset of the built-in renderer with some optimizations for performance. For example, real-time global illumination is removed from

the pipeline and only single-pass forward rendering path is available. Customizing the pipelines can be done via the render pipeline assets or by customizing the C# code for the pipeline. These pipelines also enable developers the use of Shader Graph tool [61] for creating PBS shaders. Currently, both pipelines are considered preview features and are not ready for production.

6.5.2 Built-in Shaders

Unity provides sets of shaders for all the different rendering pipelines. LW RP and HD RP require the use of their own set of lit shaders and for this Unity provides tools to upgrade the materials and by extension the used shaders in them to the ones that the render pipeline supports.

6.5.3 Writing Shaders

Shaders in Unity are written in Cg/HLSL but they are wrapped with language called ShaderLab [62]. Unity supports Surface Shaders [63], fragment and vertex shaders. Surface Shaders generate much of the code that is usually cumbersome and repetitive to write by hand. Surface Shader compiler figures out what inputs and outputs are needed according to defined surface functions. These are then generated into vertex and fragment shaders. Otherwise shaders are written using CG/HLSL. If shader developers need the shader to be affected by lights and shadows, they are better off by writing them as Surface Shaders. Otherwise they could choose to make vertex and fragment shaders, which can also be affected by lighting, but this is much more complicated. There is still support for fixed function shaders, which are completely written in ShaderLab, but these are considered legacy by Unity. Fixed function shaders are converted into vertex and fragment shaders when they are imported. [64]

Even though most of the shaders in Unity are written in a declarative language ShaderLab, the actual shader code is written in CGPROGRAM or HLSLPROGRAM snippets. These in turn are written in regular CG/HLSL shading language. Every shader is made of sub-shaders and there must always be one. Unity goes through the list of sub-shaders when the shader is

loaded and tries to use first one that the hardware supports. In the case when there is no supported subs-shaders for the target hardware Unity tries to use a fall-back shader. Different shaders, like vertex, fragment and geometry shaders are defined using pragmas and written as functions within CGPROGRAM or HLSLPROGRAM snippets. Unity also supports manually writing GLSL shaders [65], but Unity recommends that this is only done for testing or for specific cases. Unity cross-compiles Cg/HLSL into optimized GLSL if targeted platform requires it. Shaders are saved in *.shader* files. [62]

Currently this way of writing shaders works best with the built-in rendering pipeline, but LW RP and especially HD RP have their own caveats to writing shaders. At least, at the time of writing this they were not specified or documented properly. However, Shader Graph tool works with these two pipelines and you can export the shader that is being generated to further optimize them by hand or to use as a basis for writing other shaders for these pipelines.

6.5.4 Shader Graph

Unity's Shader Graph tool provides developers visual node-based shader editor to make shaders without coding. It is designed to work with the new Scriptable Render Pipeline. Shaders created with it appear as normal shaders as the nodes used are based on Unity's standard shaders. Shader Graphs themselves are saved as *.ShaderGraph* files.

Shader Graph tool works by connecting nodes in a graph network to ultimately create the shader effect you want. Shader Graph window [66] contains all the nodes included in the shader and the tools to save the shader asset and to show the asset in project. Most important nodes are Blackboard, Master Preview and Master Node. Blackboard [67] contains all the properties, and their information, that are exposed to materials. Master Preview [68], like the name would suggest, is the shaders main preview window, which shows in real-time the

current shader using Unity's built-in meshes or custom meshes used in the project. Shader Graph window can be seen in action in the Figure 5. [61]

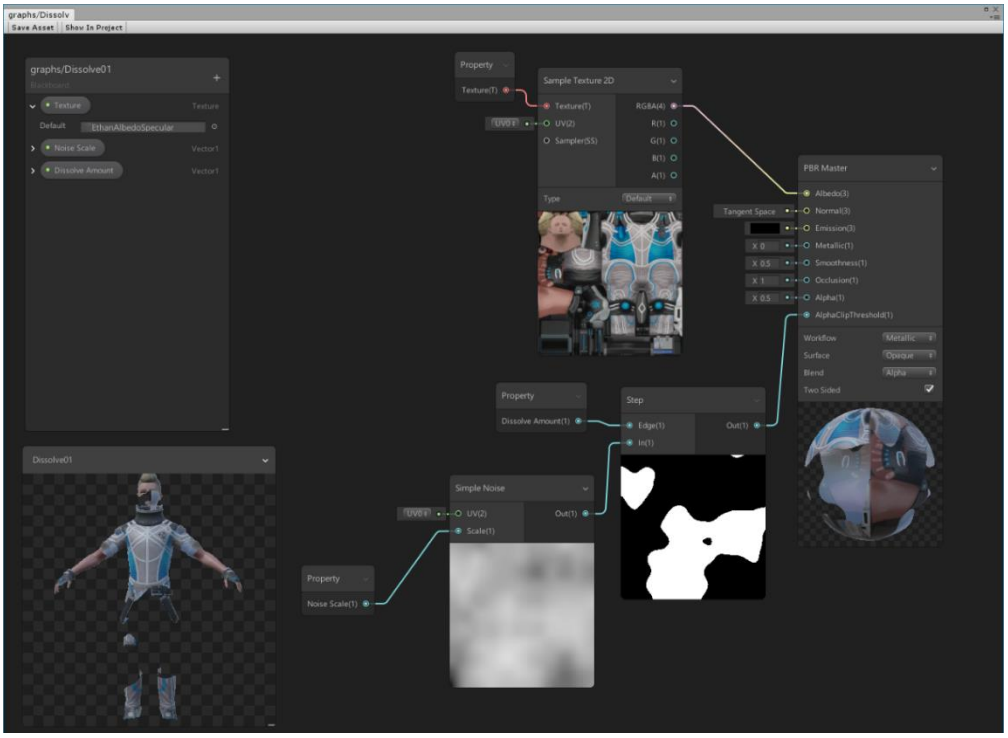

Figure 5. Creation of a dissolve shader using Shader Graph

6.5.4.1 Nodes

In Shader Graph, a node [69] is the basic building block. It defines the input and output ports if there are any. These nodes are then connected to each other and to Master Node with ports and edges to create the wanted effects. Master node [70] is the node that binds the graph together. Currently you can choose between PBR Master Node or Unlit Master Node. The difference between the two is that unlit won't be affected by lights, while PBR Master Node is designed for physically based rendering with full lighting support. Port [71] has a data type, which defines what kind of edge can connect to it. Edge [72] is the connection between two ports and there can be only one edge connection in to input port, however there can be multiple edges connected to an output.

6.5.4.2 Sub Graph

Sub Graphs [73] are graphs which can be used inside other Shader Graphs. They are usually made to perform repetitive tasks, so there is no need to create similar node structure multiple times in other shaders. Instead of having Master Node, Sub Graph has *SubGraphOutputs* node, which defines output ports for other Shader Graphs to use.

6.5.5 Applying Shaders to Objects

The created shaders are applied to objects by applying them to a material which is then applied to the object. Materials define how the objects are rendered by keeping references to all the required textures, tiling information, options and inputs that the shader needs. Materials are used by renderer components attached to the object to render the object correctly. Basically, the material specifies what shader it uses and that provides the requirements for the material. There can be many materials that use the same shader with different settings and there can be materials that used by many objects to share the shader. [74]

7 IMPLEMENTATIONS

For the implementations three were chosen from real-world projects and two different shader implementations were created as comparisons and one slightly more advanced shader was done using Unity to showcase tessellation. The real-world examples were shaders made for different projects as effects or as prototypes to test different ideas. The idea with comparison shaders was to create simple shaders to test the differences between the shader usages in the selected engines.

7.1 Shaders for Projects

Three implementations from Seepia Game's projects were chosen as examples. Each of them was done with a different engine to provide good coverage for real-world examples.

7.1.1 Darkness Shader in PixiJS

The idea for the effect wanted in the shader is to colour the underlying platform in black tint and to show light emanating from the players position. In the game the intensity for the platform tint would start from zero and as the player progresses it would end up completely hiding platform in darkness. However, the position where the player character currently is and the surrounding areas in the platforms should be visible, giving the illusion that the player is emitting light to the surrounding areas. This effect is illustrated below in the Figure 6 using the created test environment with four platforms using the created shader and mouse position representing the player position. In the example the mouse position is in the centre of the platforms to illustrate that the coordinates are respected in each of the platforms as they should.

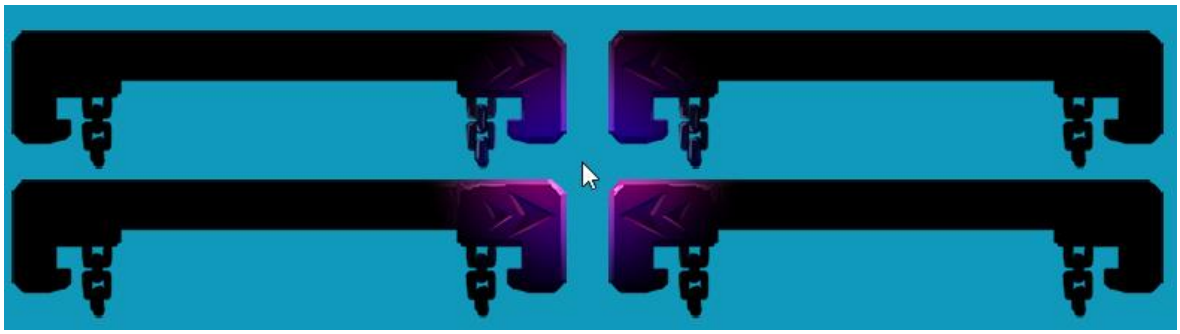

Figure 6. Darkness shader in the test environment.

In the end two versions of this shader were made. One that works as a custom filter and other that works as a shader by using the custom renderer plugin that was made as a test for it. In the Code Snippet 2 is the fragment shader code for the shader version.

```
1.  precision mediump float;
2.  uniform vec2 mouse;
3.  uniform vec2 resolution;
4.  uniform sampler2D uSampler;
5.
6.  uniform float uRadius;
7.  uniform float uSoftness;
8.
9.  uniform vec4 uDarknessColor;
10. uniform float uDarknessIntensity;
11.
12. varying vec2 vTextureCoord;
13.
14. void main()
15. {
16.         vec2 pixelPosition = vec2(gl_FragCoord.x, resolution.y - gl_FragCoord.y);
17.         vec4 textureColor = texture2D( uSampler, vTextureCoord );
18.         float lightDistance = distance(pixelPosition, mouse);
19.         float smoothedLight = smoothstep(0.0, 1.0, clamp(((lightDistance - uRadius)  /
            uSoftness), 0.0, uDarknessIntensity));
20.         gl_FragColor = mix(textureColor, vec4(uDarknessColor.xyz, textureColor.w),
            smoothedLight);
21. }
```

Code Snippet 2. Fragment shader code for the Darkness shader

In the shader version the position of the current pixel is calculated using *gl_FragCoord* and the passed resolution in the line 16. The GLSL built-in *gl_FragCoord* variable contains the window coordinates of the current fragment [75]. To get the correct y-coordinate *gl_FragCoord.y* should be subtracted from the passed resolutions y-coordinate, because the

default origin in GLSL is the bottom left corner while in PixiJS it is the top left corner. Texture colour is sampled from the texture with texture2D using *vTextureCoord* varying variable passed from the vertex shader in the line 17. In the next line the distance between current pixel and the mouse position is calculated using distance [76] function, which simply returns the distance between two points. Calculations for the light are done using *smoothstep* [77] function in the line 20, which performs Hermite interpolation between 0 and 1. It is used because it provides smooth transition between the tint colour and the lighter areas where the texture colour is visible. The function takes in two edges and the value which is being interpolated. In this case, the value is calculated by subtracting the wanted light radius (*uRadius*) from the light distance which is then divided by the softness factor (*uSoftness*). The softness factor can be changed to provide smoother or harder edges to the transition. This calculated value is clamped between 0 and the maximum wanted intensity for the tint colour (*uDarknessIntensity*). This makes it possible to control how dark the platforms can be. The final colour for the pixel is assigned using GLSL built-in *gl_FragColor* variable after it is linearly interpolated between texture colour and the tint colour using *mix* [78] function in line 21. Alpha for the tint comes from the texture colour so that the alpha of the sprite does not change. The weight for the interpolation comes from the calculated light value (line 20). Code wise differences between filter and shader versions is the screen coordinate calculation. Filter requires the usage of *filterArea* vector. It is automatically passed to the filter by PixiJS. Now the *pixelPosition* calculation changes to *vTextureCoord * filterArea.xy + filterArea.zw*. The *filterArea* uniform also needs to be added to the fragment shader.

7.1.2 Font Shader in Defold

At the time of writing this shader there was no magic bullet to create shadows to text nodes in Defold. However, there is support for setting up shadows in the editor itself, but no shader support to actually use them and show them in game. In one of the Seepia Games earlier projects there was a need for shadowed fonts to provide more polish and cleaner look to the game. Below in the Figure 7 is the resulting shader in action.

Figure 7. Font shader with shadow in action.

Defold provides few different built-in shaders to use for text. In this case the distance field-based shader is used. This allows the font to be rendered more crisply when rotated or scaled. It already has outline support, so it was easy to extend to support shadows. The shader code is defined in the Code Snippet 3.

```
1.  uniform mediump sampler2D texture;
2.  uniform mediump vec4 shadow_offset;
3.  varying lowp vec2 var_texcoord0;
4.  varying lowp vec4 var_face_color;
5.  varying lowp vec4 var_outline_color;
6.  varying lowp vec4 var_shadow_color;
7.  varying lowp vec4 var_sdf_params;

8.  void main()
9.  {
10.         lowp float distance = texture2D(texture, var_texcoord0).x;
11.         lowp float sdf_edge = var_sdf_params.x;
12.         lowp float sdf_outline = var_sdf_params.y;
13.         lowp float sdf_smoothing = var_sdf_params.z;
14.         lowp float alpha = smoothstep(sdf_edge - sdf_smoothing, sdf_edge + sdf_smoothing,
            distance);
15.         lowp vec4 tex = vec4(var_face_color.rgb, var_face_color.a * alpha);
16.         lowp float outline_distance = texture2D(texture, var_texcoord0).x;
17.         lowp float outline_alpha = smoothstep(sdf_outline - sdf_smoothing, sdf_outline +
            sdf_smoothing, outline_distance);
18.         lowp vec4 outline = vec4(var_outline_color.rgb, var_outline_color.a * outline_alpha);
19.         lowp float shadow_distance = texture2D(texture, var_texcoord0 - shadow_offset.xy).x;
20.         lowp float shadow_alpha = smoothstep(sdf_outline - sdf_smoothing, sdf_outline +
            sdf_smoothing, shadow_distance);
21.         lowp vec4 shadow = vec4(var_shadow_color.rgb, var_shadow_color.a *
            shadow_alpha);
22.         lowp vec4 result = mix(outline, tex, tex.a);
23.         result = mix(shadow, result, outline.a);
24.         gl_FragColor = result;
25. }
```

Code Snippet 3. Font shader's fragment code

The variable *var_shadow_color* in line 6 is added the fragment shader and to the vertex shader to get the needed values from the material. It is the only change made to the built-in vertex shader. Distances for font, outline and shadow are sampled from texture with texture

coordinates using *texture2D* function. Alpha for each pixel is then calculated for each font, outline and shadow using *smoothstep* function. This takes into account given smoothing and the calculated distance. In shadow's case, in line 20, the shadow offset is subtracted from the texture coordinates, when calculating the distance. Shadows position offset is given via the material in the Defold editor. The resulting alpha is then applied to a vector which contains the colour information that is later interpolated with each other to calculate the resulting colour to render. In line 22, the outline and the main colour (the face colour) of the font are interpolated by the main colour's alpha. This result is then interpolated with the shadow using the outline's alpha value. The resulting final colour is then assigned to the *gl_FragColor* variable.

7.1.3 X-ray Screen Shader in Unity

X-ray Screen shader was made to act as a screen for an x-ray machine in a 2D game. There are x-raid sprites and normal sprites for any given object and they are moved through the x-ray machine and normal sprites are shown before and after the they are passed the screen, but while they are below the screen the x-raid sprites are visible. This is achieved by having second camera in the scene which only has *Xray* layer as the culling mask thus only rendering that layer and culling others. The scenes main camera has all the other layers set as culling mask and thus does not render the *Xray* layer. The second camera renders into a render texture which is used by the X-ray Screen shader where simple noise, glitches and different colours are applied to the resulting image in x-ray screen sprite. This sprite is in the default layers and is being rendered by the main camera. Below in the Figure 8 is the X-ray Screen shader's ShaderGraph and the resulting image.

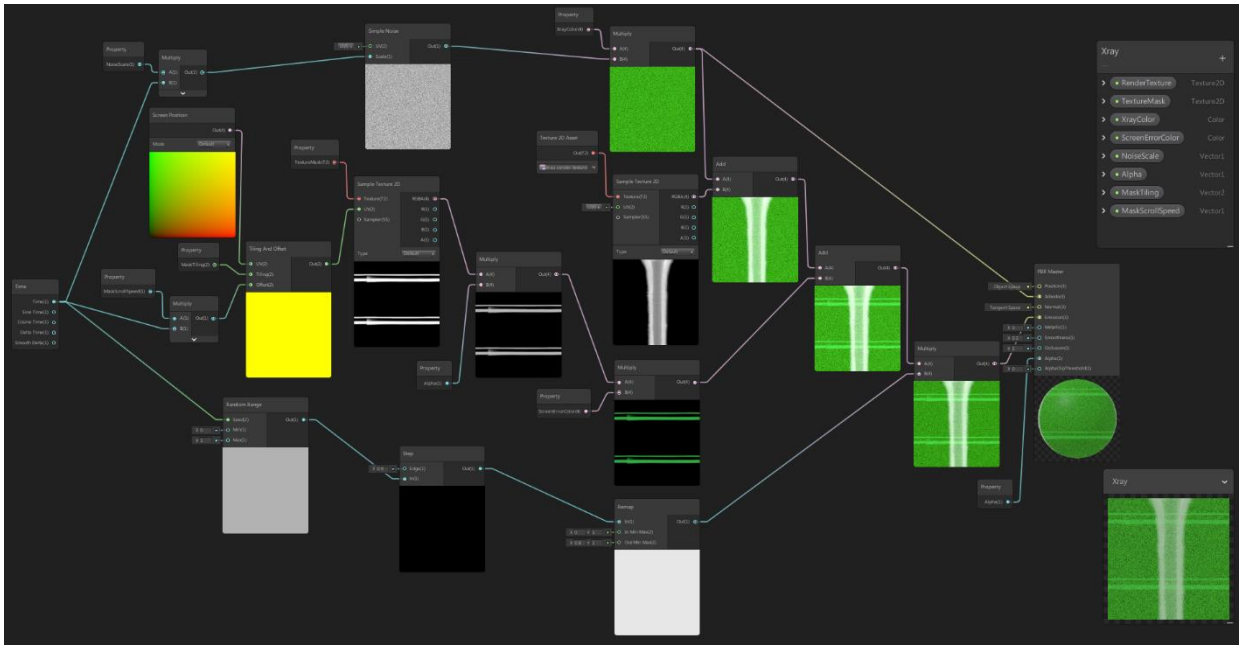

Figure 8. ShaderGraph of the Xray Screen shader.

The shader works by using Unity's built-in Time node to access the current time of the game to create random glitches, simple noise and to offset screen error texture uvs. Nodes responsible for creating the noise can be seen in the Figure 9 below.

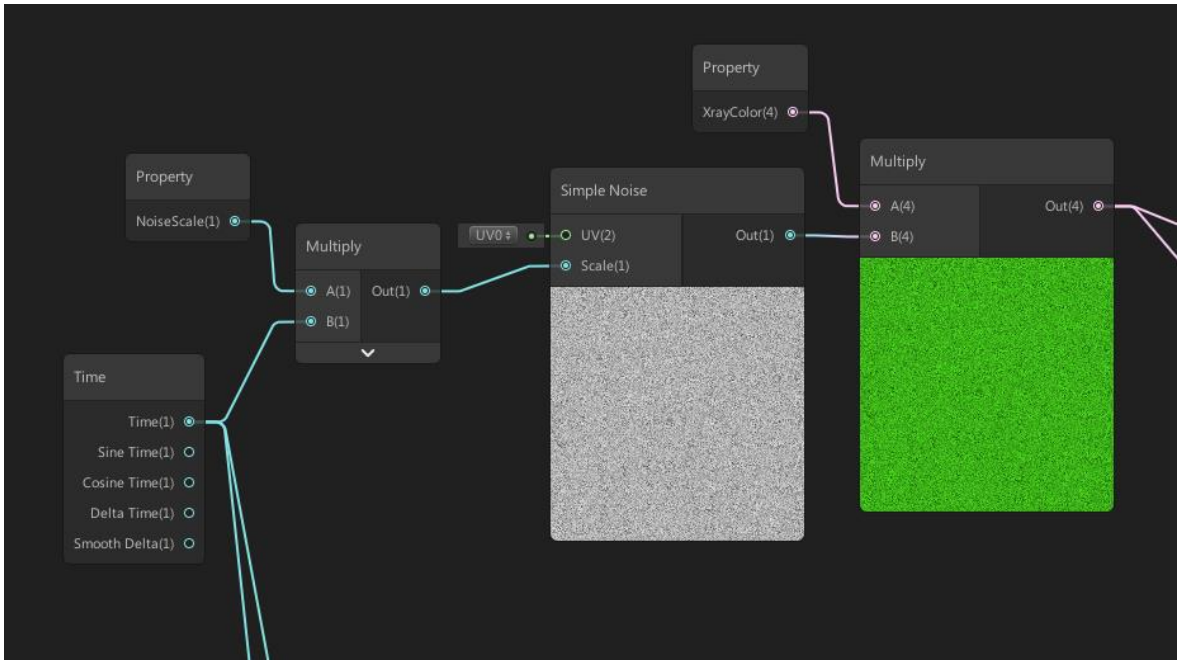

Figure 9. Nodes used to create noise

For creating noise *NoiseScale* (Vector1) property is exposed to the editor to add easy way of modifying the shader. Scale is multiplied by the game time to create changing noise using Simple Noise node. *XrayColor* (Color) property is also exposed to the editor to make it easy to change the colour of the noise by multiplying the noise with it. This is attached to the albedo port. Nodes for the flickering screen effect can be seen in the Figure 10 below.

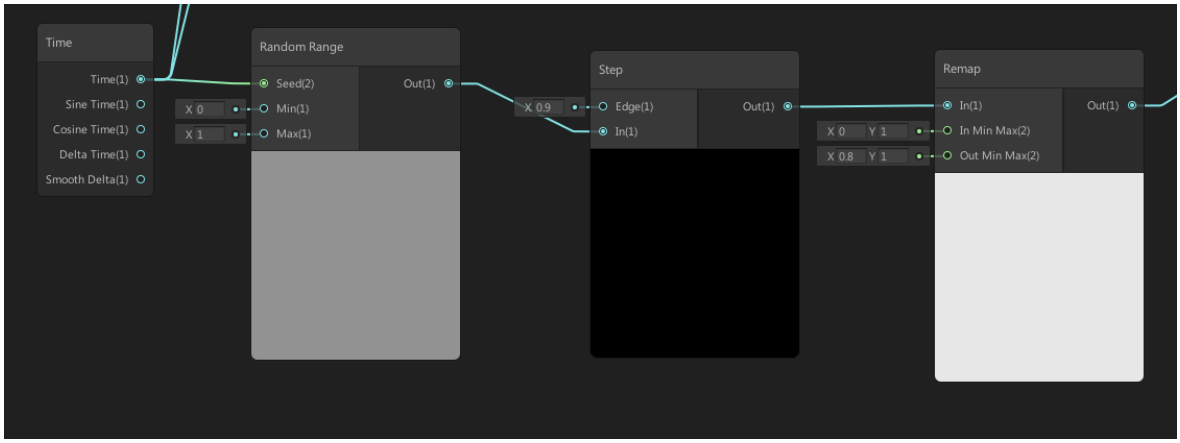

Figure 10. Flickering effect.

Screen flicker uses current time as a seed to Random Range node which return random values between 0 and 1. Step node then checks if the value is bigger than 0.9. It is the same as step function and it returns 1 if the input *In* is equal or greater than the input *Edge* and otherwise it returns 0. The resulting flicker intensity is controlled by Remap node, which in this case remaps the input values from 0 to 1 range to 0.8 to 1 range. This makes the flicker more muffled. In the Figure 11 we can see how the screen error effect is created.

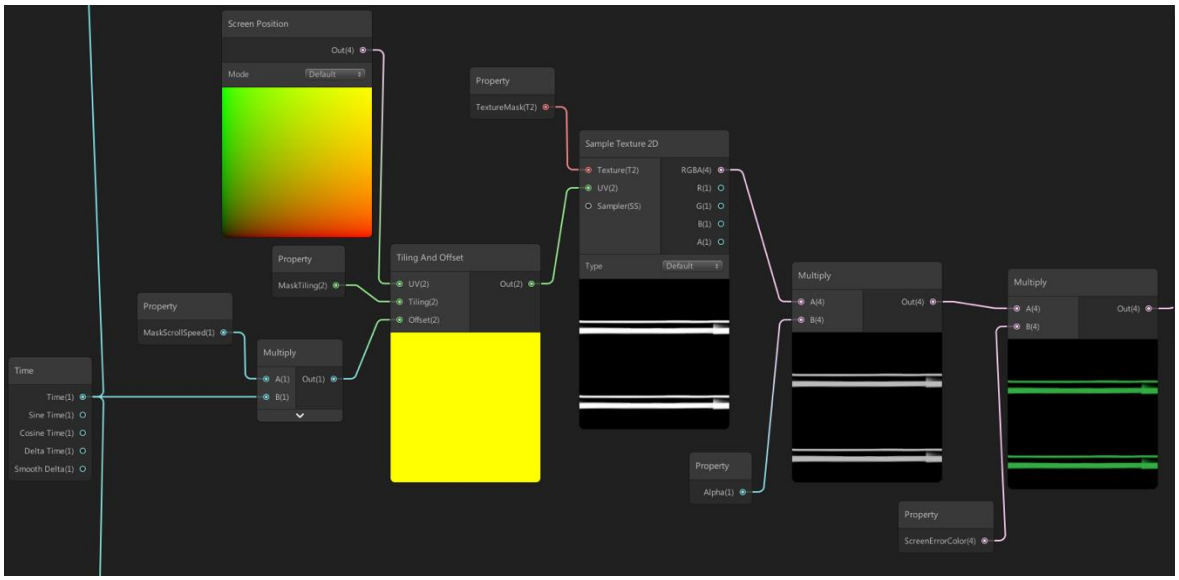

Figure 11. Screen error effect.

Screen error effect is basically a black and white texture which is being scrolled from the top of the screen to bottom of the screen. Time node is used to get the game time which is then multiplied by the exposed *MaskScrollingSpeed* property. This allows developers to control the speed of the effect's movement in the screen. The resulting value is used as an offset for tiling. Current meshes screen position is used as the initial uv and exposed property *MaskTiling* (Vector3) controls the tiling. The resulting values are used as the UV inputs in the error texture sampling. Next results are multiplied by shaders alpha and wanted colour to finish the look of the error effect. Lastly the resulting error effect and noise are added to the sampled render texture. This is then multiplied with glitch effect to achieve the wanted look. This is then attached to the emission port. Render texture nodes are shown in the Figure 12.

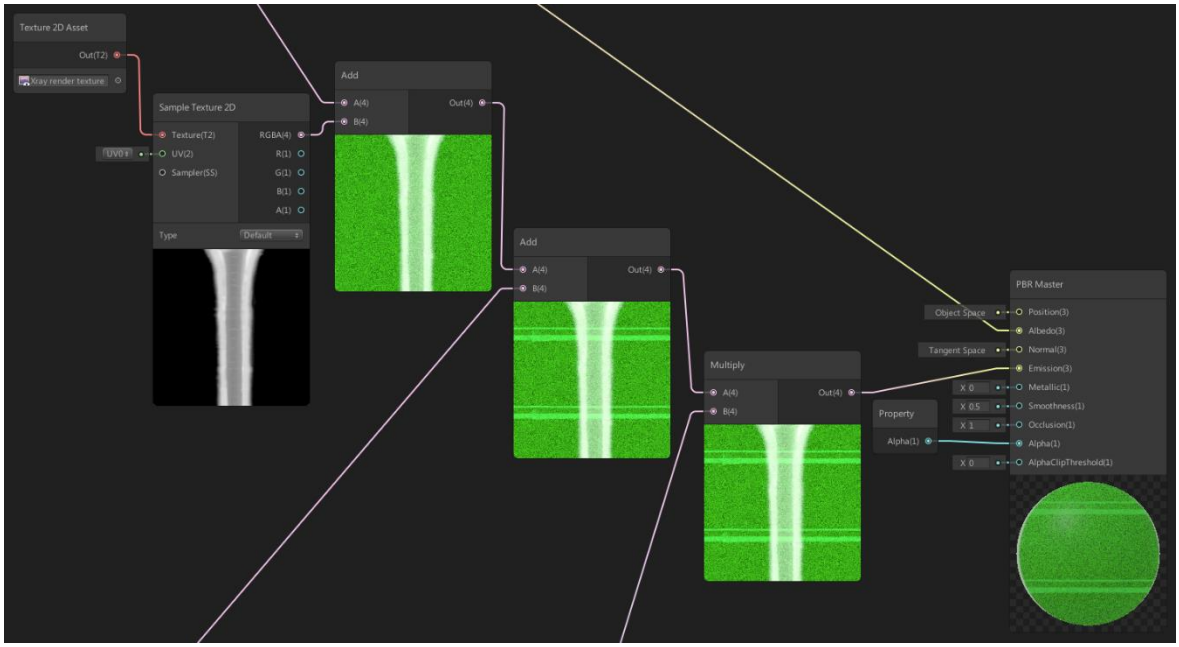

Figure 12. Render texture usage in the X-ray shader.

7.2 Implementations with Comparisons

While the game engines used in this thesis serve different purposes currently in Seepia Games projects, there is value knowing the differences between shader usages in these engines. It was decided to create two basic implementations using these engines to show the differences. Highlight effect and dissolve effects have many use cases and are relatively simple to implement and thus were chosen. Highlight shader was chosen to show how basic colour manipulation is achieved and dissolve effect was chosen to show how additional textures are used in these engines. In this case to show how noise map is used in the shaders.

7.2.1 Highlight Shader

Highlight shader is very commonly used to show players what object or character is chosen or just to highlight something important in the scene. Four different implementations were made: One using PixiJs, another with Defold and two more using Unity. These were made 2D games in mind, with the exception of the second Unity implementation, which provides highlight effect to 3D objects to show two different approaches for making shaders in Unity.

The 2D versions of the Highlight shader works by sampling the pixel's alpha and adding to it the offset pixel's alphas. This is used to check where the outside of the image is and where the outline should be created. Colour is applied to the outline and the normal colours and the outline is interpolated in the end to finish the effect. The 3D version works by using the model's normals to extrude the shape of the model outwards and applying the outline colour to it while rendering the model otherwise normally.

7.2.1.1 Pixi Implementation

Highlight shader works really well in the PixiJS using both the renderer plugin approach and the vanilla filter style. Resulting effect can be seen in the Figure 13.

Figure 13. PixiJS implementation of the Highlight shader.

In addition to the main texture, the fragment shader needs to know what the outline colour is, which is the vector4 uniform *uOutlineColor,* and outline size, which is the uniform float *uOutline.* For testing purposes this is enough, but if more precise control over the outline is needed, there could be uniforms for the horizonal and vertical offsets. These could be

calculated by the size of the sprite rather than have single value for both the offset and the outline size. Fragment code can be seen in the Code Snippet 4.

```
1.  precision mediump float;
2.  uniform sampler2D uSampler;
3.  uniform float uOutline;
4.  uniform vec4 uOutlineColor;
5.  varying vec2 vTextureCoord;
6.
7.  void main() {
8.      vec4 textureColor = texture2D( uSampler, vTextureCoord );
9.      float alpha = 4.0 * texture2D(uSampler, vTextureCoord).a;
10.     alpha += texture2D(uSampler, vTextureCoord + vec2(uOutline, 0.0)).a;
11.     alpha += texture2D(uSampler, vTextureCoord + vec2(-uOutline, 0.0)).a;
12.     alpha += texture2D(uSampler, vTextureCoord + vec2(0.0, uOutline)).a;
13.     alpha += texture2D(uSampler, vTextureCoord + vec2(0.0, -uOutline)).a;
14.     float outlineUsage = ceil(uOutline);
15.     vec4 outlinedTextureColor = textureColor * textureColor.a;
16.     vec4 coloredEdge = uOutlineColor * (1.0 - textureColor.a);
17.     outlinedTextureColor += coloredEdge;
18.     textureColor = mix(textureColor, outlinedTextureColor, outlineUsage);
19.     if(alpha <= 0.0)
20.     {
21.         discard;
22.     }
23.     gl_FragColor = textureColor;
24. }
```

Code Snippet 4. Highlight shader's fragment code in PixiJS.

Texture colour is sampled in the line 8 and after that the checking of the outline starts from the line 9. In line 14, the usage of the outline is checked. If the value is zero, there is no outline and if it is larger than zero then the outline is used. In the next line, is the alpha pre-multiplication. This is optional, but usually this gives better results, when filtering images or composing them [79]. Next the edge colour is multiplied to areas where alpha is zero. In line 18 the texture colour and outline colour are interpolated. In this case the outline is used when outline usage is 1. Lastly, areas where the alpha is zero are discarded. In PixiJS the usage and manipulation of shader uniforms is quite simple when the renderer plugin is set. In the Code Snippet 5 is the example usage of the shader.

```
1.      CreateShaderPlugin("ShaderTestPlugin", null, fragmentShader);
2.      let sprite = PIXI.Sprite.fromImage('assets/TestImage.png', "");
3.      sprite.anchor.set(0.5);
4.      sprite.position.set(400, 300);
5.      app.stage.addChild(sprite);
…
6.      sprite.pluginUniforms = {
7.              uOutlineColor: {0: 0, 1: 210, 2: 255, 3: 255},
8.              uOutline: 0.01
9.      };
10.     sprite.pluginName = "ShaderTestPlugin";
…
11.     sprite.pluginUniforms.uOutline = 0;
```

Code Snippet 5. Highlight shader usage in PixiJS.

In the example snippet the shader plugin is created, and the fragment shader is passed as a parameter to it in the line 1. The sprite is created from the test image and basic settings are set before it is added to the stage. What is important in relation to the shader is that the initial *pluginUniforms* are set. In lines 7 and 8, initial values for outline colour and outline size are set. After that, in line 10, the default sprite plugin for the test sprite is replaced with the rendering plugin, which has the fragment shader. These uniforms can be changed in runtime as shown in the line 10 where the outline is disabled.

7.2.1.2 Defold Implementation

Defold implementation is done using the same fragment shader as the PixiJS version and it also uses the default Defold vertex shader. However, the implementation differs in how the engine handles the shaders. Materials provide excellent way to setup and pass information to the shaders. Resulting effect can be seen in the Figure 14.

Figure 14. Defold implementation of the Highlight shader.

Because the used OpenGL ES version 2 is the same that is used in the PixiJS implementation, the same fragment shader could be used. However, the naming conventions are slightly different, but the shader functionality is the same. Main differences between the implementations is that the uniforms used in the Defold materials are vectors (vector4) and thus it is better to combine uniforms and pass more information in one than use multiple uniforms, which have information only in the x-channel and nothing in the y-, z- and w-channels. Fragment code can be seen in the Code Snippet 6.

```
1.  varying mediump vec2 var_texcoord0;
2.  uniform lowp sampler2D u_main_texture;
3.  uniform lowp vec4 u_outline_settings;
4.
5.  void main()
6.  {
7.      lowp vec4 texture_color = texture2D(u_main_texture, var_texcoord0);
8.      lowp float alpha = 4.0 * texture2D(u_main_texture, var_texcoord0).a;
9.      alpha += texture2D(u_main_texture, var_texcoord0 + vec2(u_outline_settings.w, 0.0)).a;
10.     alpha += texture2D(u_main_texture, var_texcoord0 + vec2(-u_outline_settings.w, 0.0)).a;
11.     alpha += texture2D(u_main_texture, var_texcoord0 + vec2(0.0, u_outline_settings.w)).a;
12.     alpha += texture2D(u_main_texture, var_texcoord0 + vec2(0.0, -u_outline_settings.w)).a;
13.     lowp float outline_usage = ceil(u_outline_settings.w);
14.     lowp vec4 outlined_texture_color = texture_color * texture_color.a;
15.     lowp vec4 colored_edge = vec4(u_outline_settings.xyz, 1.0) * (1.0 - texture_color.a);
16.     outlined_texture_color += colored_edge;
17.     texture_color = mix(texture_color, outlined_texture_color, outline_usage);
18.     if(alpha == 0.0)
19.     {
20.         discard;
21.     }
22.     gl_FragColor = texture_color;
23. }
```

Code Snippet 6. Defold Highlight shader fragment code.

Notable things in the example code is the aforementioned uniform usage. In this case, in line 3, is the *u_outline_settings*, which combines the PixiJS implementations colour and width uniforms into single uniform. The x-, y- and z-channels are used for the colour of the edge, while w-channel is reserved to the size. How this is setup in the material, can be seen in the Figure 15 below.

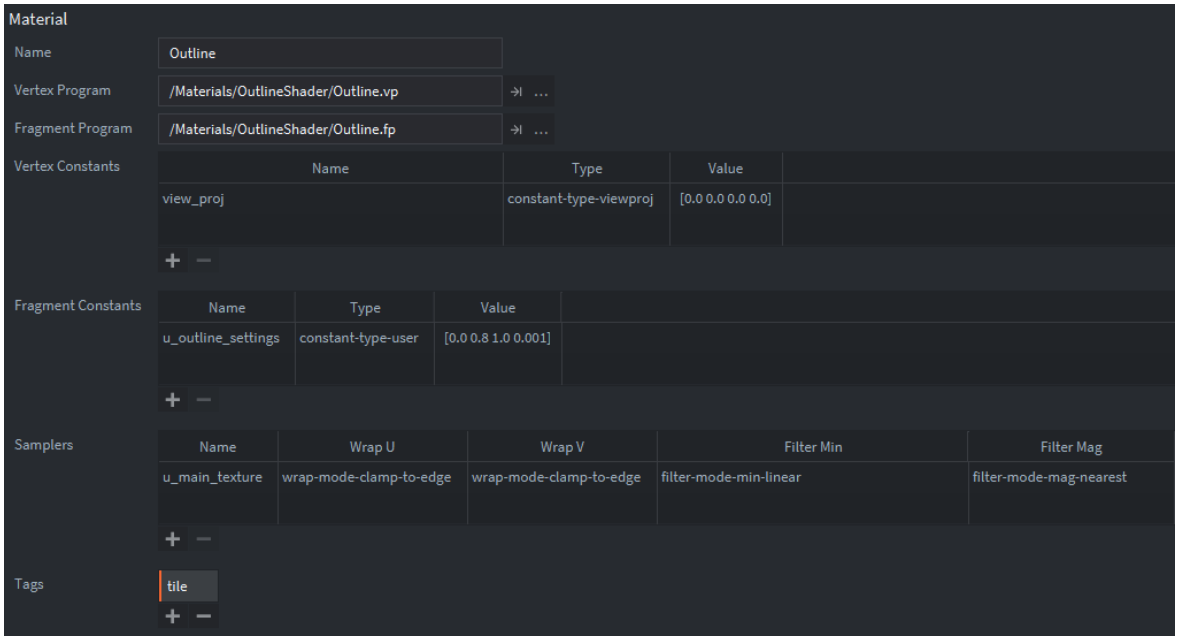

Figure 15. Defold Highlight shader material.

In Defold the manipulation of the uniform variables is extremely simple. When you have script attached to the game object, you can set wanted variable as can be seen in the Code Snippet 7. In this case the outline colour and size are set when the object has been initialized. As an example, there could be a function that is executed when the sprite has been clicked or tapped on, which then changes the w-channel's value from the initial zero to something greater and thus turning the outline on.

```
1.  function init(self)
2.      go.set("#OutlineSprite", "u_outline_settings", vmath.vector4(0.0, 0.8, 1.0, 0))
3.  end
```

Code Snippet 7. Highlight shader Defold usage.

7.2.1.3 Unity Implementation

The idea of the first Unity implementation is to replicate the same shader, which was used in the PixiJS and Defold implementations using Shader Graph. This worked out well, while the implementation is quite simple, the ShaderGraph looks more complicated at first glance. Resulting effect can be seen in the Figure 17 below and the ShaderGraph can be seen in the Figure 16.

Figure 17. Resulting effect in Unity.

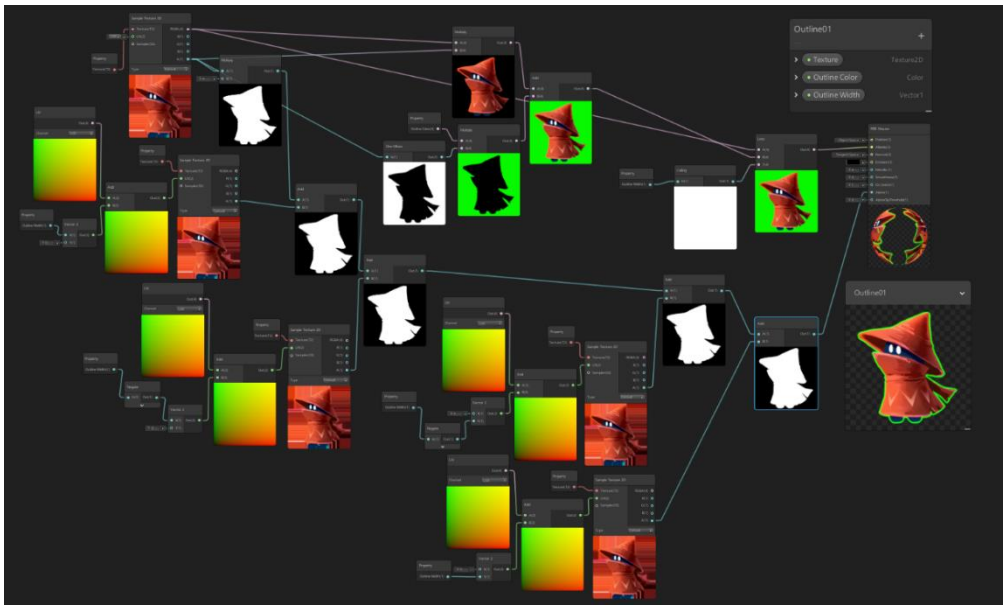

Figure 16. ShaderGraph of the Highlight shader.

The idea behind the alternative version is to provide example of similar effect for 3D models. It is also good opportunity to show how Unity's manually written shaders can be used. The resulting shader can be seen in the Figure 18.

Figure 18. 3D implementation of the Highlight shader in Unity.

The shader uses Unity's Surface Shader template and adds a vertex and a fragment shader programs to the SubShader as a pass. It has physically based lighting and shadows enabled. Code relating to the added pass can be seen in the Code Snippet 8 and the whole shader code can be viewed in Appendix 1.

```
1.      CGINCLUDE
2.      #include "UnityCG.cginc"
3.      struct appdata
4.      {
5.              float4 vertex : POSITION;
6.              float3 normal : NORMAL;
7.      };
8.      struct v2f
9.      {
10.             float4 position : POSITION;
11.     };
12.     float _OutlineWidth;
13.     float4 _OutlineColor;
14.     v2f vert(appdata v)
15.     {
16.             v2f o;
17.             o.position = UnityObjectToClipPos(v.vertex);
18.             float3 normal = mul((float3x3)UNITY_MATRIX_IT_MV, v.normal);
19.             normal.x *= UNITY_MATRIX_P[0][0];
20.             normal.y *= UNITY_MATRIX_P[1][1];
21.             o.position.xy += normal.xy * _OutlineWidth;
22.             return o;
23.     }
24.     ENDCG
…
25.     Pass
26.     {
27.             ZWrite Off
28.             CGPROGRAM
29.             #pragma vertex vert
30.             #pragma fragment frag
31.             half4 frag(v2f i) : COLOR
32.             {
33.                     return _OutlineColor;
34.             }
35.             ENDCG
36.     }
```

Code Snippet 8. 3D Highlight Shader code.

Vertex shader starts from the line 14. In the line 17 the object position is transformed from object space to camera's clip space in homogeneous coordinates and the object's normals are rotated to eye space in the line 18. Normals are then multiplied using projection matrix, so that the outline looks to the player to be constant size and always facing the camera. Last thing the vertex shader does is that after the outline width is applied to the calculated normals they added to the position's x- and y-coordinates in line 21. All that is left to be done is for the fragment shader to set the wanted colour to the pixels in the line 33. This pass is drawn first before the normal Surface Shader code and because the outline is extruded from the

base model the outline stays visible even when the model is drawn afterwards. Similar to the other outline implementations the outline colour and width can be controlled via scripts as seen in the Code Snippet 9.

```
1.  using UnityEngine;
2.
3.  public class OutlineTester : MonoBehaviour
4.  {
5.      [Header("Outline Settings")]
6.      public Color outlineColor;
7.      public float outlineWidth;
8.
9.      private Material _material;
10.
11.     private static readonly int outlineWidthID = Shader.PropertyToID("_OutlineWidth");
12.     private static readonly int outlineColorID = Shader.PropertyToID("_OutlineColor");
13.
14.     private void Start()
15.     {
16.         _material = GetComponent<Renderer>().material;
17.         if (_material != null)
18.         {
19.             _material.SetFloat(outlineWidthID, outlineWidth);
20.             _material.SetColor(outlineColorID, outlineColor);
21.         }
22.     }
23. }
```

Code Snippet 9. Unity Highlight shader usage.

The testing code is attached to the game object which has the outline shader and material applied. Starting from the line 6 are the exposed values for colour and width that can be edited via the editor. Material is fetched in the line 16 so that the values for the outline can be modified. Next, as an example, the outline width and colour are set using the values that the developer has chosen in the editor. Ultimately this is just to show how easily materials can be edited in Unity.

7.2.2 Dissolve Shader

Dissolve shader has few uses, for example, it can be used as an effect for objects that are getting destroyed. Main idea with the shader is to see how different engines can use multiple textures for different purposes. The shader uses main texture for the image itself and second texture for the noise map, which is used for the dissolve effect. The dissolve effect is done by sampling the noise map which basically contains values from zero to one. Usually dissolve shader has a controlling variable which is used to check whether a pixel is within the dissolve effect and discarded. In this case there is also a variable for the width of the coloured edge between sprite's normal pixels and the discarded ones. The line works like the dissolve, but the dissolve amount has the edge width added to it making the effect slightly larger. More detailed look to the effect can be seen in the implementations below. In the end all the implementations are quite similar in terms of the shader code and the main differences comes from the usages and the differences in the engine implementations.

7.2.2.1 Pixi Implementation

Dissolve effect required slight trickery in the PixiJS implementation. It uses the renderer plugin, which allows the use of the second texture. However, to use the noise texture it had to be created as a sprite and added to the stage. Position and the size of it did not matter and the width and height were set to zero making it invisible to players. For the shader to have access to the noise texture, the sprites texture needed to be added to the shader's uniforms. Now it could be sampled in the shader. Just by using it as a BaseTexture or Texture did not work. In the Figure 19 is the resulting effect.

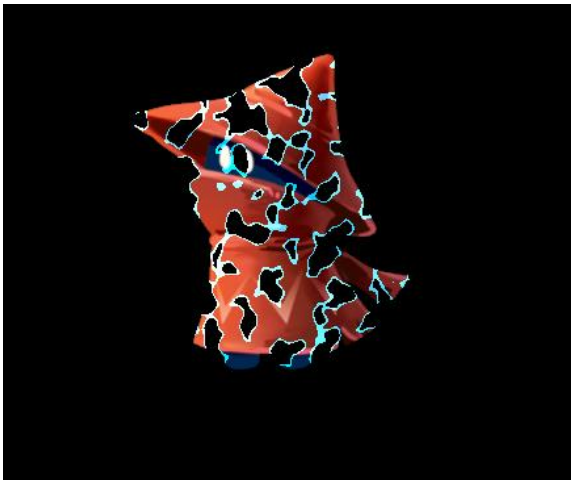

Figure 19. PixiJS implementation of the Dissolve shader.

To make things as simple as possible the noise texture and the test image are the same size. This makes the sampling easier, because the same texture coordinates can be used. In the Code Snippet 10 is the dissolve effects fragment shader.

```
1.  precision mediump float;
2.  uniform sampler2D uSampler;
3.  uniform sampler2D uNoiseTexture;
4.  uniform vec2 uDissolveSettings;
5.  uniform vec4 uEdgeColor;
6.  varying vec2 vTextureCoord;
7.
8.  void main() {
9.      vec4 texture = texture2D( uSampler, vTextureCoord );
10.     vec4 noise = texture2D( uNoiseTexture, vTextureCoord );
11.     float edgeSize = uDissolveSettings.x + uDissolveSettings.y;
12.     float dissolveUsage = ceil(uDissolveSettings.x);
13.     float edge = step(noise.r, edgeSize) * dissolveUsage;
14.     vec4 dissolvedTexture = texture - edge;
15.     vec4 coloredEdge = edge * uEdgeColor;
16.     dissolvedTexture = dissolvedTexture + coloredEdge;
17.     if(noise.r <= uDissolveSettings.x || texture.a <= uDissolveSettings.x)
18.     {
19.     discard;
20.     }
21.     gl_FragColor = dissolvedTexture;
22. }
```

Code Snippet 10. PixiJS Dissolve shaders fragment code.

In the line 12 the need for the dissolve effect is checked, because the shader tries to take in to account situations where there is no dissolve effect applied. This is done using ceil

function, which returns 1 in case there is any amount of dissolve effect to be shown and 0 in case there is not. In the line 13 the dissolve effect is checked using *step* function, where the red channel of the sampled noise texture is used as the edge for the function and the dissolve amount (*uDissolveSettings.x*) as the input. The idea is that the areas where noise is greater than the dissolve amount will return 1 and otherwise 0 and this is saved in the *edge* variable. In the line 14 the pixels that are in the dissolved areas are subtracted from the texture's pixels. After that the edge colour is applied to the pixels which are in the dissolved areas and the edge area. This is then added to the final colour information in the line 16. The pixels, which are in the dissolved area, and fully transparent pixels from the main texture are discarded to finish the effect using the if statement starting from line 17. Shader usage can be seen in the Code Snippet 11.

```
1.      let noiseSprite = PIXI.Sprite.fromImage('assets/SolidNoise.png', "");
2.      noiseSprite.width = 0;
3.      noiseSprite.height = 0;
4.      app.stage.addChild(noiseSprite);
…
5.      let sprite = PIXI.Sprite.fromImage('assets/TestImage.png', "");
6.      sprite.anchor.set(0.5);
7.      sprite.position.set(400, 300);
8.      app.stage.addChild(sprite);
…
9.      sprite.pluginUniforms = {
10.             uDissolveSettings: {0: 0.4, 1: 0.05},
11.             uEdgeColor: {0: 0, 1: 255, 2: 0, 3: 255},
12.             uNoiseTexture: noiseSprite.texture
13.     };
14.     sprite.pluginName = "ShaderTestPlugin";
…
15.     sprite.pluginUniforms.uDissolveSettings[0] = 0.6;
16.     sprite.pluginUniforms.uDissolveSettings[1] = 0.02;
17.     sprite.pluginUniforms.uEdgeColor = {0: 255, 1: 255, 2: 255, 3: 255};
```

Code Snippet 11. PixiJS Dissolve shaders usage.

Noise map is created as noise sprite in the line 1. The sprite, which has the shader applied is created on line 5. The actual shader usage can be seen in the line 9, where the uniforms for the shader are applied. From the line 15 onwards, one can see how the effect can be manipulated on runtime. All the applied uniforms can be changed and as an example the dissolve effects dissolve amount has been set to 0.6 and the line width dropped to 0.02 and the edge colour itself has been changed to white.

7.2.2.2 Defold Implementation

Dissolve effect's Defold implementation is similar to the outline shader implementation in that the fragment shaders are basically the same with PixiJS. Biggest differences come from how the image is used and how the noise texture is used. The easiest way to add extra texture using the built-in tools is to use quad in place of regular sprite. What this means in practical terms is that the scaling is different and quad model was required. Resulting effect can be seen in the Figure 20.

Figure 20. Defold implementation of the Dissolve shader.

In the Code Snippet 12 the fragment code can be seen. Main differences between PixiJS and Defold shaders are the uniform usage and naming.

```
1.  varying lowp vec2 var_texcoord0;
2.  uniform mediump sampler2D u_main_texture;
3.  uniform mediump sampler2D u_noise_texture;
4.  uniform lowp vec4 u_dissolve_settings;
5.  uniform lowp vec4 u_edge_color;
6.
7.  void main()
8.  {
9.          lowp vec4 texture = texture2D(u_main_texture, var_texcoord0);
10.         lowp vec4 noise = texture2D(u_noise_texture, var_texcoord0);
11.         lowp float edge_size = u_dissolve_settings.x + u_dissolve_settings.y;
12.         lowp float dissolve_usage= ceil(u_dissolve_settings.x);
13.         lowp float edge = step(noise.r, edge_size) * dissolve_usage;
14.         lowp vec4 dissolved_texture = texture - edge;
15.         lowp vec4 colored_edge = edge * u_edge_color;
16.         dissolved_texture = dissolved_texture + colored_edge;
17.         if(noise.r <= u_dissolve_settings.x || texture.a <= u_dissolve_settings.x)
18.         {
19.                 discard;
20.         }
21.         gl_FragColor = dissolved_texture;22.
22. }
```

Code Snippet 12. Defold Dissolve shader fragment code.

The material for the Dissolve shader can be seen in the Figure 21. Notable difference between normal sprite material and this is the addition of the second sampler texture for the noise.

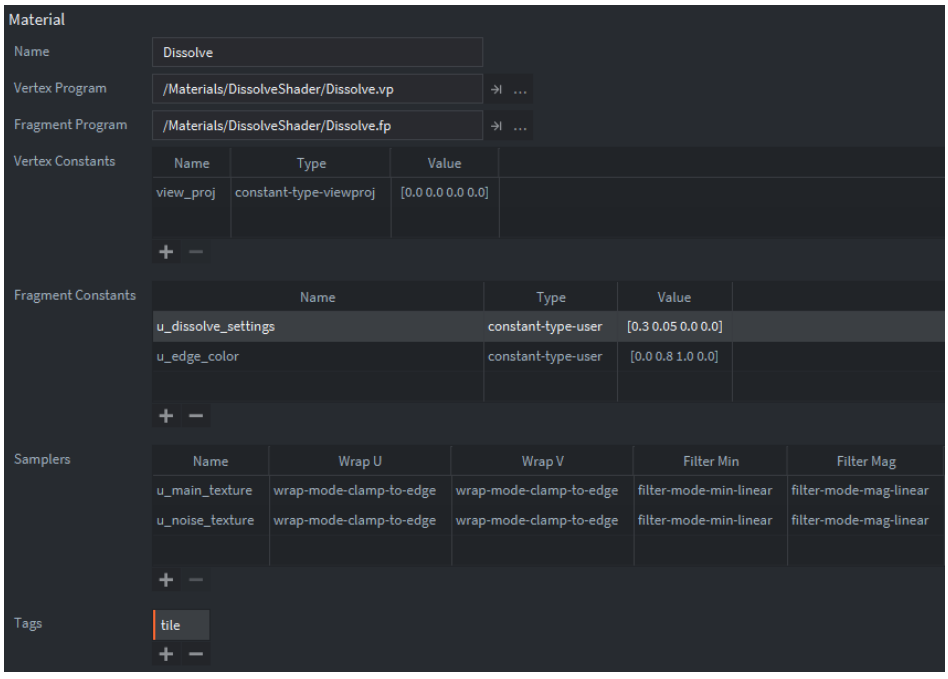

Figure 21. Defold Dissolve shader material settings.

Manipulating and using this during runtime is quite simple. In the Code Snippet 13 below the uniform *u_dissolve_settings* is changed for the object when the game object is initialized.

```
1.  function init(self)
2.      go.set("#DissolveObject", "u_dissolve_settings", vmath.vector4(0.6, 0.05, 0, 0))
3.  end
```

Code Snippet 13. Dissolve shader usage in Defold.

7.2.2.3 Unity Implementation

In Unity the dissolve effect is done using ShaderGraph. The resulting shader has effectively the same functionality as the other implementations. Resulting effect can be seen in action in the Figure 22.

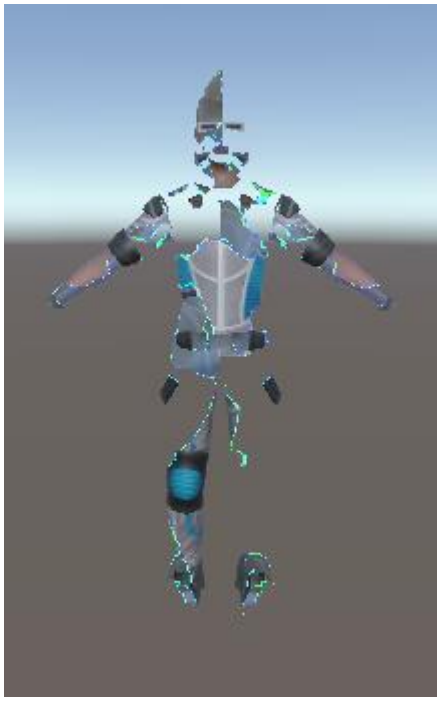

Figure 22. Unity implementation of the Dissolve shader.

Shader Graph provides built-in way to discard pixels by using alpha clip threshold. This is essentially the same as checking the pixel's alpha and discarding it if it is lower than the inputted value. The Shader Graph implementation can be seen in the Figure 23 below.

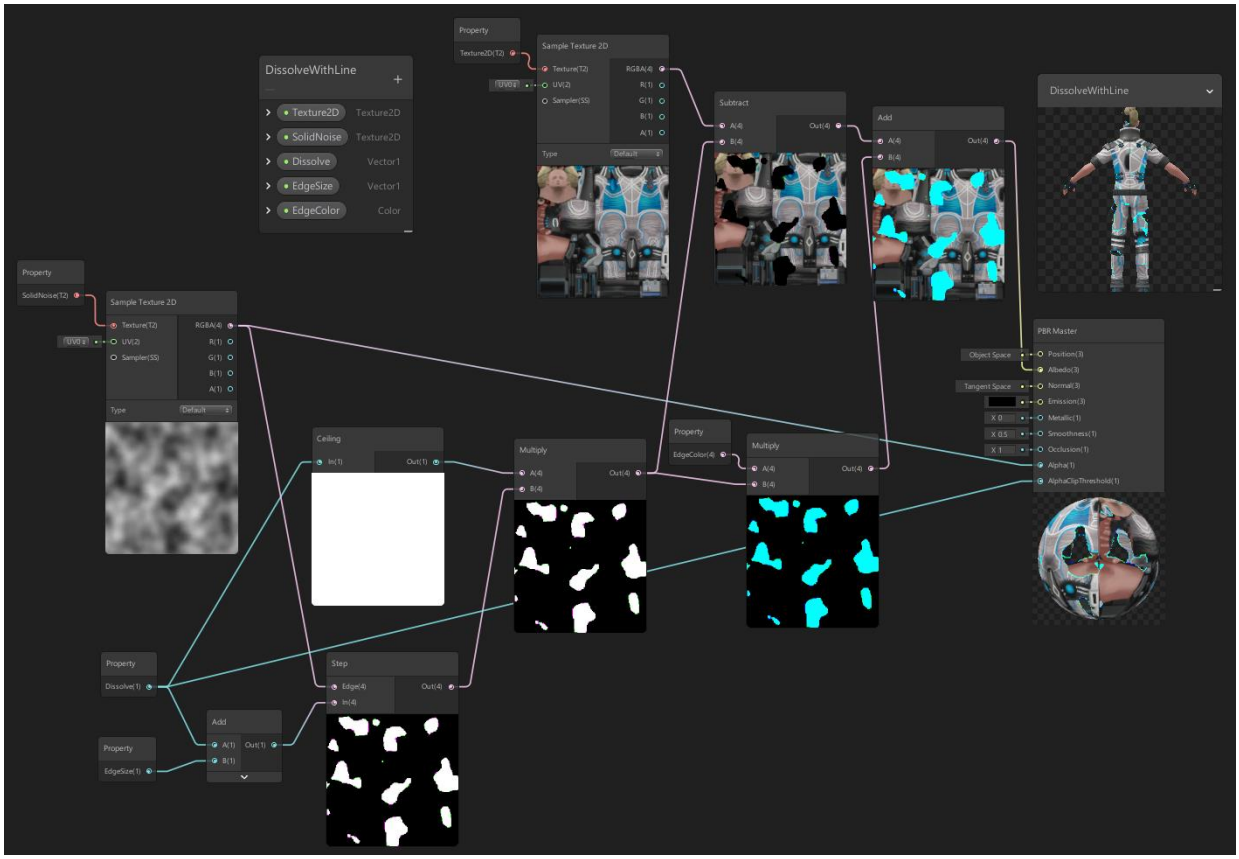

Figure 23. Shader Graph implementation of the Dissolve shader.

7.3 Advanced Implementation

Comparison implementations were chosen to be very basic in terms of complexity and the difficulty to implement using many engines without having to change the shader code too much. One of the reasons for this thesis was to try out and learn something new and exciting. One of the shader topics discussed in the theoretical part, in chapter 5.6, was tessellation shaders and it was chosen as the more advanced study topic. The shader is created using only Unity, because it has better support for it and provides excellent helper functions and examples [80].

7.3.1 Tessellation Shader in Unity

In the implementation snowy theme was selected due to all the possibilities this kind of shader provides. For example, this could bring more realism for character movement through snow, by leaving impressions to the snow from the steps. The resulting shader can be seen in the Figure 24.

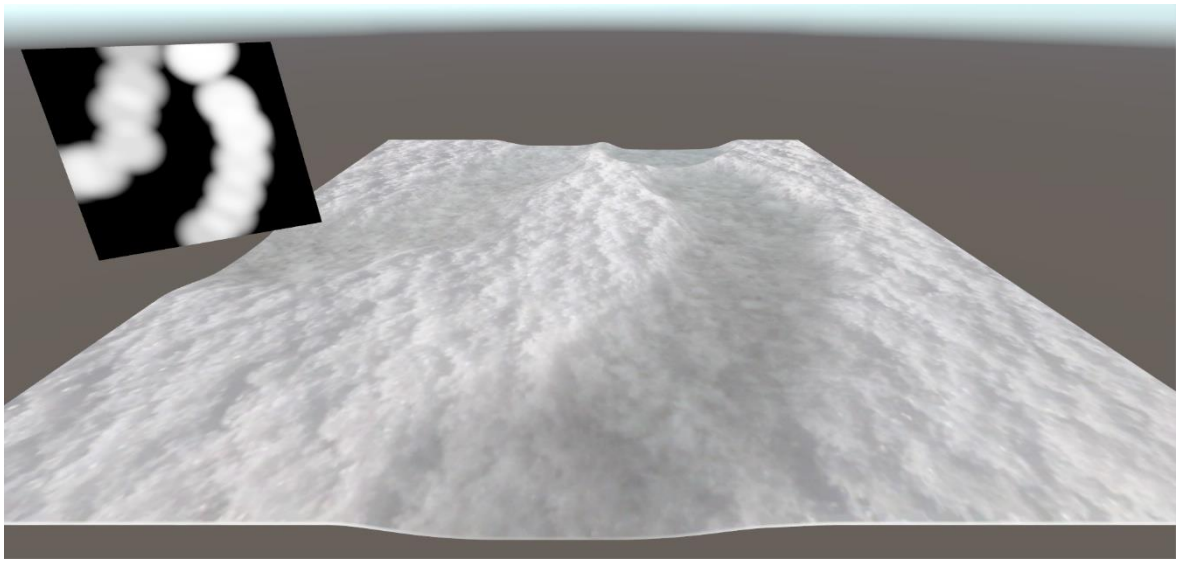

Figure 24. Tessellated snow using displacement texture.

In the test environment the shader is applied to a plane, which also has a *MouseDraw* script attached, see Appendix 1 for the source code. It records the user's left mouse clicks and drags to a render texture, where the actions are saved in the texture's red channel. The render

texture uses a simple displacement shader, which draws the actions to current mouse position using the settings provided by the *MouseDraw* script. The fragment shader code can be seen in the Code Snippet 14 below and the whole shader code in the Appendix 2.

```
1.  fixed4 frag (v2f i) : SV_Target
2.  {
3.      float red = tex2D(_MainTex, i.uv).r;
4.      half brushDistance = distance(i.uv, _BrushCoordinate.xy);
5.      half draw = smoothstep(0.0, 1.0, 1 - saturate((brushDistance - _BrushSize) /
        _BrushEdge)) * _BrushStrength;
6.      half finalColor = lerp(red, 1.0, draw);
7.      return finalColor;
8.  }
```

Code Snippet 14. Displacement shader for the render texture.

The fragment shader is quite similar to the one made for the darkness shader implementation in section 7.1.1. In the line 3 the current pixels red channel is sampled and next the distance between the mouse position is checked. Contrary to the darkness shader the value for the brush is inverted. Lastly the current red value and the new red value is interpolated and returned.

The tessellation shader then reads these values and applies the displacement to the plane. It is based on Unity's distance-based tessellation example [80] and the resulting shader code can be seen at Appendix 3. In the following code snippets, we can see the most important parts of the shader. In the Code Snippet 15 is the vertex code.

```
1.  void disp (inout appdata v)
2.  {
3.      float d = tex2Dlod(_DisplacementTexture, float4(v.texcoord.xy,0,0)).r *
        _Displacement;
4.      v.vertex.xyz -= v.normal * d;
5.      v.vertex.xyz += v.normal * _Displacement;
6.  }
```

Code Snippet 15. Tessellation shader's *disp* function.

The example shader calculates distance-based tessellation using Unity's built-in *UnityDistanceBasedTess* function, which determines the tessellation factors per vertex based on the distance from the camera. The displacement render texture's red channel is checked and used for the displacement of vertices. In the Unity's example the displacement was done only upwards. In this case, the data from the render texture is used to move vertices down, while brought up by the amount of the displacement (*_Displacement*), which is one of the exposed values to the developers. This essentially moves the collider of the mesh to the lowest point so that the object that creates these displacements in the first place is positioned correctly. Finally, in the Code Snippet 16 is the surf function of the shader.

```
1.  void surf (Input IN, inout SurfaceOutput o)
2.  {
3.      amount = tex2Dlod(_DisplacementTexture, float4(IN.uv_DisplacementTexture, 0,
        0)).r * _TextureMixPercentage;
4.      half4 mainTexture = tex2D (_MainTexture, IN.uv_MainTexture) * _MainColor;
5.      half4 secondaryTexture = tex2D (_SecondaryTexture, IN.uv_SecondaryTexture) *
        _SecondaryColor;
6.      half4 c = lerp(mainTexture, secondaryTexture, amount);
7.      o.Albedo = c.rgb;
8.      o.Specular = 0.2;
9.      o.Gloss = 1.0;
10. }
```

Code Snippet 16. Tessellation shader's surf function

The *surf* function is based on the Unity's tutorial with few exceptions. These being the removal of unpacking of normals, which created a weird look for the snow when it was used even with one texture, and the addition of second texture for the lower parts of the snow. Also, the *_TextureMixPercentage* float was exposed to provide a way to control how much the two textures were to be interpolated. When the amount for the interpolation is calculated using the displacement render texture data, it is multiplied by the percentage variable in the line 3. Next the main and secondary textures are sampled and tinted. In the line 6 the interpolation between the textures is done. Basically, this makes the lower parts of the snow use different texture than the upper parts and thus creating more realistically looking environment. The addition of the percentage variable provides more control for the developers.

8 RESULTS

The results for the PixiJS shader usability and visual tools usage testing in Unity are presented in this chapter. The shader usability is tested by benchmarking a real-world project and a test environment performance using built-in shader, filter and shader versions and by comparing the differences in the implementations. The presented results for the visual tools usage in Unity are based on observed differences between writing and visually designing shaders and by presenting the ideas that arose while using these tools.

8.1 Shaders and PixiJS

One of the main goals for this thesis was to find out how usable shaders are in PixiJS because it is the game engine that Seepia Games is currently heavily relying on. Using shaders in PixiJS has high potential, but it is something that is not necessarily required to create great looking games.

8.1.1 Performance

PixiJS filter and shader performance was tested using the Darkness Shader from Section 7.1.1. Both the filter and the shader versions were applied to the test environment and the actual game project. Applying the filter to the project was very easy as it only required adding the fragment shader code and apply its usage in place of the current implementation. On the other hand, implementing the shader required adding the plugin to the project and implementing its usage. Outside of the plugin the changes to the code base were quite minimal. In the original version of the project there are platforms to which the player tries to jump on and these use the PixiJS built-in *ColorMatrixFilter* filter. The resulting effect does not change after the initial creation and as a result this can be saved as a bitmap. This means that these platforms act like normal sprites and thus they can be automatically batched by PixiJS. The initial version of this implementation, which did not use the bitmap approach, had similar performance to the filter implementation of the Darkness shader. However, the darkness effect requires that the position of the player and the overall darkness is updated constantly and thus can’t be optimized in similar manner. The testing was done using a

desktop PC and several mobile devices. The main specifications of the PC are as follows: AMD FX-8350 CPU, 8GB of Random Access Memory (RAM) and NVIDIA GTX 1070 GPU. The mobile devices and their operating systems are the following: iPhone SE (iOS 12), iPhone 6S (iOS 11), Huawei HONOR7 PLK-L01 (Android 6.0) and Sony Xperia XA F3111 (Android 7.0). In all the devices Chrome browser was used for the testing. Safari was also tested on the iOS devices, but there was no major difference in terms of performance.

8.1.1.1 Chrome DevTools

Chrome DevTools [81] was used to make benchmarks to check what kind of workload different implementations create. This kind of profiling is crucial when optimizing the games performance and, in this case, it provided some insight to the CPU overhead. Chrome DevTools and other similar tools used to profile website and JavaScript performance provide massive amounts of information to developers. While this can be deep information, in this case, the information was kept to overall performance thus the summary values provided by the tool were used. It provides the following categories: Loading, Scripting, Rendering, Painting, Other and Idle. Loading events include, for example, parsing the HTML and sending and receiving request. In this case the benchmarks did not include any loading events, because benchmarking was started after the game was fully loaded. Scripting includes events all the way from animation frame events to mouse and key events and to the game code. Rendering events are related to style, layout and scroll updates. These include things like render tree updates and reserving textures for paint operations. Paint events are, for example, image decode and resize, composite layer handling and texture uploads. Other category includes everything that is not covered by the other categories, while idle means literally the time there was nothing to do. In this case changes in scripting, rendering and painting are the most interesting ones.

The benchmarks were done using 20 second segments and the summary values were observed. These benchmarks were done three times in the test environment and the averages can be seen in the Table 1 below, where the results are in milliseconds.

Table 1. Desktop benchmarks in the test environment using Chrome DevTools

Desktop Benchmark, Chrome DevTools, Test Environment					
Test Environment	Scripting	Rendering	Painting	Other	Idle
No effect	38,00	1,33	14,00	188,00	19776,50
Filter	159,67	55,33	215,67	268,00	19311,37
Shader	39,67	2,33	27,67	205,33	19725,37

These results paint a quite clear picture. When there is no effect applied to the platforms, the performance is clearly best. When all the platforms have the shader version applied, CPU usage is slightly higher in all the categories. In this case there is a clear impact to painting and it is usually quite costly in terms of performance [82]. However, when the filter version is applied, there is a massive performance hit across the board. This means that filter version is clearly causing the biggest CPU overhead. Similar benchmark was done to the actual project and the results can be seen in the Table 2 below.

Table 2. Desktop benchmarks in the project using Chrome DevTools.

Desktop Benchmark, Chrome DevTools, Project					
Checkpoint 0	Scripting	Rendering	Painting	Other	Idle
Original	827	379	29	200	18635,4
Filter	1250,67	28,00	47,33	124,00	18542,20
Shader	948,00	263,67	30,33	173,67	18587,10
Checkpoint 5	Scripting	Rendering	Painting	Other	Idle
Original	1302	17	20	119	18556,4
Filter	1299,67	12,00	32,00	111,00	18553,67
Shader	1321,00	32,33	34,67	135,33	18799,13
Checkpoint 50	Scripting	Rendering	Painting	Other	Idle
Original	885	294	48	235	18548,2
Filter	730,30	339,00	88,67	281,67	18569,80
Shader	501,00	281,33	288,00	290,00	18644,07
Checkpoint 122	Scripting	Rendering	Painting	Other	Idle
Original	483,00	376,00	263,33	330,67	19772,03
Filter	1381,67	27,33	52,33	128,33	18414,6
Shader	1230,00	68,00	26,67	146,67	19050,53

These results should be taken with a grain of salt. During the testing major fluctuations were observed in all test cases. However, the most notable result here is the performance of the original version, where the performance was most constant. In these tests there are quite many things moving in the screen and custom physics checks done to make any kind of clear interpretations. In the end, the negative effects of both filter and shader versions are clear when compared to original version, but when it comes to the difference between the two versions, the difference is much subtler.

8.1.1.2 Draw Calls and Rendering Related Commands

The CPU and GPU usages were also tested using SpectorJS [83] browser plugin. It provided us with the information about GPU related commands and draw calls. Tests were done to both the test environment and to the project version. To understand what is being benchmarked here, more detailed explanation about draw calls and related commands are in order. Draw calls are very resource intensive work for the CPU, where information about textures, states, shaders, rendering objects and buffers and other information that GPU needs

to draw resources to the screen is being prepared and encapsulated [47, 84]. Basically, a group of triangles using the same material and their texture and other properties are a draw call. Clear is a rendering command, which is used to clear images from the framebuffer. Both of these are backed by many different commands that the engines, or in this case PixiJS's, rendering pipeline needs to draw the images to the screen correctly. For example, there can be many different commands before a draw call is executed and all this combined can lead to CPU overhead and even bottlenecking when there are too many draw calls to handle. Draw calls, command counts and clears for the test environment can be seen in the Table 3 below.

Table 3. Test environment benchmark using SpectorJS.

SpectorJS Benchmark, Test Environment			
	Draw Calls	Commands	Clears
No Effect	1	7	1
Filter	8	140	5
Shader	4	44	1

When studying the information provided by the SpectorJS plugin, the differences between implementations are quite clear. In this optimal situation, the built-in version performs the best. It manages to batch the four platforms into one draw call and there is only one clear command, which happens at the start. The filter version generates two draw calls and one clear command per platform, while the shader version generates only one draw call per platform and only one clear at the start. There is a clear advantage for the shader version when compared to the filter one, but both have a clear negative effect in terms of performance when compared to having no custom effect applied.

This test showed how these implementations work in the best-case scenario, but when it comes to the real-world application their results started to get slightly muddier. Draw calls, command counts and clears for real world test case can be seen in the Table 4 below.

Table 4. Project benchmark using SpectorJS.

SpectorJS Benchmark, Project			
Checkpoint 0	Draw Calls	Commands	Clears
Original	8	59	1
Filter	25	298	8
Shader	18	162	1
Checkpoint 5	Draw Calls	Commands	Clears
Original	11	138	2
Filter	28	311	8
Shader	21	185	1
Checkpoint 50	Draw Calls	Commands	Clears
Original	9	78	2
Filter	20	220	6
Shader	15	132	1
Checkpoint 122	Draw Calls	Commands	Clears
Original	7	81	1
Filter	36	412	11
Shader	23	201	1

In these tests the original version performs the best due to properly working batching. Filter version increases everything quite much, while shader version stays somewhere in the middle. PixiJS tries to batch display objects that are next to each other in the hierarchy and using filters or shaders breaks this. Meaning that every time there is a display object using filter or shader in between the batchable display objects the batching is broken and these increase draw calls in the filter and shader cases even more. The increase is not only because these display objects can't be batched, but because the previous and the next objects in the hierarchy that could be in one batch are in two different batch and there is an object using filter or shader in the between adding more draw calls. Basically, this means that the original version can batch all the platforms, and other display objects, into fewer draw calls if there are no objects in between that would break batching. Similar increase can be seen in the commands, but the increase in clear calls in filter's case is due to the increased draw calls and how they are generated. In filter's case every time a display object is drawn to the screen the frame buffer is cleared and then the sprite is drawn to the framebuffer upside down and finally it is added to the canvas frame buffer.

8.1.1.3 Overall Mobile Performance

While the desktop benchmarks give some indication how filters and shaders perform using powerful enough hardware. Majority of players will be using mobile devices to play the game. Thus, it is important to know how they perform on higher and lower end mobile devices. Table 5 contains the average framerates from three different 30 second test runs. First benchmark was done using iPhone 6S, which had iOS 11 installed.

Table 5. Average FPS on iPhone 6S.

Mobile Benchmark, Average FPS, iPhone 6S (iOS 11)			
Checkpoint	Original	Filter	Shader
0	46,2	41,43	45,09
5	47,83	40,61	44,57
50	52,08	49,59	49,95
122	54,47	28,47	31,92

In these tests the performance was quite similar, until the test was done in the checkpoint 122. Original version stayed on top in all the tests here, while both the filter and shader versions had negative effects on performance, the difference was almost negligible in the first three checkpoints. In the first two benchmarked checkpoints, there are six platforms, which clearly shows in these results. In checkpoint 50, there are only four platforms, which explains the highest frame rates in filters and shaders case. When it comes to the checkpoint 122, the platform count is the highest, ten in total, which explains massive performance drop in filters and shaders case. Original version has the best performance in this case. This is mostly due to slight drop in background asset counts. This is also shown in the desktop benchmarks as the drop in scripting times and as one draw call drop in the previous test. Next up, in the Table 6, are the results using iPhone SE, which has the latest iOS 12 installed.

Table 6. Average FPS on iPhone SE.

Mobile benchmark, Average FPS, iPhone SE (iOS 12)			
Checkpoint	Original	Filter	Shader
0	55,43	55,99	57,61
5	58,95	52,3	57,56
50	59,58	58,14	59,85
122	58,44	42,78	50,35

These test results are somewhat surprising. The iPhone 6S and the iPhone SE have similar internal hardware, but SE has the newer operating system. Clearly the newer operating system has done major optimizations to JavaScript and WebGL performance. The results shown here, continue the same pattern as the first test, but with much higher average FPS. In checkpoints 5 and 122, where the filter version's draw calls are high, a clear performance drop can be seen. Although, the shader version also suffers clearly in the checkpoint 122, the actual fluidity of the gameplay was not affected nearly as much as using the filter version. However, there was still clear stuttering and slowdowns at times. Next in the Table 7, are the results for Huawei HONOR7, which has Android 6.0 installed.

Table 7. Average FPS on Huawei HONOR7.

Mobile benchmark, Average FPS, Huawei HONOR7 PLK-L01 (Android 6.0)			
Checkpoint	Original	Filter	Shader
0	17,36	4,49	10,38
5	13,39	5,22	13,91
50	13,49	6,43	15,04
122	32,06	4,73	17,86

Even though Honor 7 was closer to high end device when it was released, nowadays it represents the middle to low end of mobile devices. Here the similar trend continues, with few differences. Original version's checkpoint 0 is performing better than the next two checkpoints, while using filter and shader the performance is lower than the next two. This did not happen in the iPhone benchmarks. This might just be due to differences in hardware and the fact that checkpoint 0 makes great use of pixi-particles, which can also affect the performance. In this benchmark the shader version provides much better performance than

the filter version and sometimes even beating the original implementation. In the next Table 8, the results for Sony Xperia XA (F3111), which is using Android 7.0, are shown.

Table 8. Average FPS on Sone Xperia XA.

Mobile benchmark, Average FPS, Sony Xperia XA F3111 (Android 7.0)			
Checkpoint	Original	Filter	Shader
0	38,34	15,32	36,62
5	39,96	16,83	48,12
50	47,31	22,48	48,15
122	60,83	14,25	54,12

In this case the Sony seems to be performing much better than Huawei. The performance of the shader version is almost on par with the original. Shader and original versions trade blows consistently. However, the negative effect of the filter is very clear. The difference in performance between the Sony and the Huawei could be explained by their screen size and their Android versions. Huawei has a 1080p screen while the Sony only has the 720p screen. This means that Sony must handle much less pixels than the Huawei. Additionally, the different Android version could have similar performance improvements as could be observer in the iPhone benchmarks. Naturally, they have different hardware, which is also a factor.

8.1.2 Filters

Filters provide an easy way of creating shaders with relative ease. However, filters are limited by not being batched and missing some of the features, when compared to shaders but when creating simple effects, they work adequately. This is partly because of how optimized PixiJS otherwise is. It also provides many built-in filters and premade filters via their *pixi-filters* library [85]. These can be added to existing projects or their source code can be used as basis for custom filters.

8.1.3 Shaders

Shader creation in PixiJS is currently cumbersome process but can yield great visual benefits if used correctly. After careful examination of PixiJS forums, examples, and tutorials, a decent understanding about shader usage in PixiJS was achieved. While there were some excellent examples of the custom filters, the documentation about shaders, the differences between filters and shaders, or how to use them were quite minimal. Currently shader support is not built in to the core framework and needs a custom renderer plugin to work. PixiJS developers provide a good example for a renderer plugin which is hosted on GitHub [49]. It was used as basis for the plugin that was used in the testing. Simple visual effect can be achieved using both filters and shaders, but when advanced features are needed shaders and the accompanying plugin is essential. This also opens up potential for performance optimizations if needed or alternatively developers could wait for the next major release of PixiJS and see what changes that will bring shader wise. In the end using shaders (or filters) in PixiJS can be done relatively easily and they will provide improvement on visual effects but will impact performance negatively.

8.1.4 Quality of Testing

The results given above give a good starting point towards future decision making and possible optimization within Seepia Games projects. However, due to the nature of web centric development, the tools, frameworks and even languages change rapidly, and this makes the testing here only valid when using PixiJS V4 and the next major release is currently in alpha and the stable version should be coming in the near future. Other point to consider, is that when this thesis started knowledge about shaders, their optimizations and best practises, were practically zero. While this work has provided massive amounts of knowledge and practise around shaders, there are still many things to learn about good practises and optimizations and so on. For example, there could be some obvious optimizations missing from the shader implementation to the project. In the end, the testing provides good information about filter and shader usage in PixiJS.

8.2 Visual Tools for Creating Shaders Using Unity

Writing shaders can be scary to many developers, mainly because of the complexity and knowledge needed to write them. This has pushed the popularity of visual tools for, not just developers, but for example, designers to create effects they want without there being coders in the middle.

Shader creation in Unity has evolved in the recent years with the introduction of Surface Shaders and 3rd party visual scripting tools like Amplify Shader [86] and Shader Forge [87]. More recent changes to rendering pipelines and the addition of the Shader Graph have boosted this evolution.

This thesis has proven how invaluable visual tools are for beginners. When starting from zero and trying to look for shader tutorials and making first tests by writing them manually, the amount of knowledge required just to show a triangle or a sprite in the screen can be overwhelming, even when testing things out using ShaderToy or glslViewer, the feedback can be confusing. While testing things out initially using, for example, Unity and Shader Forge, the feedback is visual and clear. It shows the resulting effect in the previews, and this revitalized the motivation for this work. After the initial hurdle and with some knowledge, writing shaders manually was much easier when one understood what the expected result should be. Additionally, using visual tools speeds up prototyping of shaders immensely, because the result from the preview is almost instant.

In the end Shader Graph and other visual tools are not just for developers but for designers as well. The requirement for shader programming knowledge is still there, but it is not as essential anymore. However, this can easily lead to very badly optimized and poorly running shaders, which might look visually great. Thus, there is still a need for understanding how shaders work. There are also still cases, where manual coding is required, for example, to optimize parts of the generated code or just to remove unneeded boilerplate code. This is also the case when developing new graphic technologies or using technologies and techniques not supported by the tool. For example, at the time of writing the tessellation example and the second outline shader, Shader Graph did not support vertex manipulation nor tessellation.

9 DISCUSSION AND CONCLUSIONS

The purpose of this thesis was to compare how shaders work in the three game engines that are currently in use by Seepia Games. After researching about the theory of shaders, the makings of 3D graphics, ins and outs of game engines and finally implementing few different shaders, a clearer picture is started to form. Now answers to the three research questions could be given.

9.1 Comparisons Between Engines

In terms of usability of shaders Unity takes the crown, hands down, Defold coming as a clear second and PixiJS coming in last. While shaders are usable in all three, due to the requirements of the engines, there is a clear difference between how big of a role each engine gives to shaders.

Current version of Unity provides few different rendering pipelines and few ways of creating shaders, but the requirement for visually great looking games has evolved the shader creation in the engine greatly. The ease of creating shaders using Shader Graph is rewarding and the amount of documentation, tutorials and examples for writing shaders is excellent. There has been a slight stigma towards games created using Unity, that they have a certain Unity look, but moving towards PBS shaders and more modern rendering pipelines will be a positive factor going forwards.

Currently Defold provides excellent features for using shaders, but there are still few missing features. For example, even though the engine is mainly used for 2D games, there could be easier ways of using multiple textures in sprites and also supporting these in the editor itself. The usage of materials makes the shader usage very clear and the ease of manipulating the shaders via scripts is excellent.

PixiJS stands out in this comparison, not just by being heavily optimized and fast, but for whatever reason the usability of shaders has not been a high priority to the developers of the engine. The amount of research and trial and error used for making the filters and shaders

work, does not represent the results at all. After the research has been done, the usage of the shaders is relatively easy and makes a lot of sense, but the lack of documentation and examples, and of course the lack of prior knowledge, were a big factor when trying to get shaders to work. The results for the shader case, when compared to the filter case, are by far more satisfying even though they are not the expected silver bullet for great looking games in HTML5. In the end, both the filter and shader ways are usable within reason, but neither is recommended to be completely relied upon.

When deciding which engine is the best for the next project, the usability of the shaders is only one of the factors. Choosing of the engine should be done using common sense and project's purpose in mind. However, for projects that have need for great visuals and extensive use of shaders, Unity is the best choice out of these three.

9.2 PixiJS and Shaders

The second research question was how usable shaders are in PixiJS. Continuing from the last chapter, they are usable within reason. Shaders provide better results when lower end devices must be taken in to account.

The tested project was not created with shader or filter usage in mind and this shows in the results. Adding initial support to them in a project that has already grown quite large is relatively easy. While there are great many optimizations to be done to the project itself, getting proper filter or shader support requires some refactoring to achieve better results than seen in the testing. However, this implementation provided clear indications what works and what does not work when using filters or shaders. For example, when there are complex game objects, which include multiple sprites, adding custom filter or shader breaks the batching. This requires changes to the structuring for the hierarchy and refactoring to the code to provide better performance. However, researching how this could be done and possibly testing and implementing these kinds of changes, remains to be done in future works.

In the end, both ways of implementing the shaders are possible and functional in small scale. For more intensive use, the shaders offer better performance over the filters, but require lot

of additional work to implement. After making the initial implementations maintaining these features should be quite easy.

9.3 Unity and Shaders

Third research question was to find out how usable visual tools are for creating shaders. While the support for these tools is recent addition to the Unity engine itself, there has been 3rd party tools for many years and many of the competing game engines like Unreal Engine [88] and Godot [89] provide these tools out of the box. This makes perfect sense, because it provides more choices to the developers and even to the designers. It makes prototyping much easier and faster and not least of all it provides excellent bridge to shader development for beginners. In small projects this frees up resources, when developers can concentrate on creating features, designers can easily test different looks for scenes and objects using the tools.

Visual tools don't come without their drawbacks. While they provide excellent feature sets, in Unity's case not all shader features are currently supported. Automatic shader generation for each platform is a must have feature, but this might also generate undesired bloat and might require developers with proper knowledge to cut the bulk. The generated shaders might not be the most optimized pieces of code and thus might require optimization by hand by the developers. Optimization of generated shader, and optimization of Unity shaders in general is very interesting, but it remains one of the possible topics for future works.

10 SUMMARY

This master's thesis took a very broad look at the world of 3D graphics using shaders as the lens. There were three main objectives: first was to compare how shaders can be used in Unity, Defold and PixiJS; second was to find out how usable shaders are in PixiJS; and the last objective was to test and find out how usable visual tools are in creation of shaders in Unity.

For the first objective, research about 3D graphics and the three game engines was done and comparisons and clear findings was achieved. The second objective required deeper dive into how PixiJS game engine functioned, and an implementation of a renderer plugin was done to test filter and shader usage and their performance against original test project code base. Second objective was achieved. Third objective was achieved using the research done for other objectives and by a deeper dive into Unity. Implementations were done by writing shaders and by using Shader Graph to find out how usable Shader Graph tool is.

To support all the objectives three examples from previous and current projects were examined and additionally two comparative implementations of similar shaders was done using all the engines. Additionally, one more advanced shader was implemented because of the interests of the author.

There was a massive amount of knowledge gained from this research, which goes to show how much there is still to learn. The validity of the conclusions and results are quite subjective due to the broadness of the research and the requirement for deeper knowledge and understanding about the subject before being able to provide proper answers.

REFERENCES

[1] Entertainment Software Association, “2017 Essential Facts About the Computer and Video Industry,” Entertainment Software Association, 2017.

[2] K. Perlin, “An image synthesizer,” *ACM Siggraph Computer Graphics,* vol. 19, no. 3, pp. 287-296, 1985.

[3] M. Bailey and S. Cunningham, Graphics Shaders: Theory and Practice, 2nd Edition ed., Boca Raton: A K Peters/CRC Press, 2016.

[4] A. Zucconi and K. Lammers, Unity 5.x Shaders and Effects Cookbook, Birmingham: Packt Publishing Ltd..

[5] Goodboy Digital Ltd, “PixiJS — The HTML5 Creation Engine,” [Online]. Available: http://pixijs.download/release/docs/index.html. [Accessed 03 06 2018].

[6] Unity Technologies, “Unity User Manual (2018.2),” [Online]. Available: https://docs.unity3d.com/Manual/index.html. [Accessed 24 10 2018].

[7] Defold, “Welcome to Defold,” [Online]. Available: https://www.defold.com/manuals/introduction/. [Accessed 24 10 2018].

[8] Y. Kotsarenko, A. Humberto and F. Ramos, “Alternative Specular Approach for real-time rendering optimized for higher performance,” *Programación Matemática y Software,* vol. 4, no. 1, 2012.

[9] N. Hoffman, “Physically-Based Shading Models in Film and Game Production,” in *SIGGRAPH 2010*.

[10] N. Hoffman, “Crafting Physically Motivated Shading Models for Game Development,” in *SIGGRAPH 2010*.

[11] Y. Gotanda, “Physically Based Shading Models in Film and Game Production: Practical Implementation at tri-Ace,” in *SIGGRAPH 2010*.

[12] Microsoft, “Getting Started with DirectX Graphics,” [Online]. Available: https://msdn.microsoft.com/en-us/library/windows/desktop/hh309467(v=vs.85).aspx. [Accessed 01 06 2018].

[13] J. Peddie, “Who’s the Fairest of Them All?,” [Online]. Available: http://www.cgw.com/Publications/CGW/2012/Volume-35-Issue-4-June-July-2012/Who-s-the-Fairest-of-Them-All-.aspx. [Accessed 24 10 2018].

[14] Khronos Group, “OpenGL Overview,” [Online]. Available: https://www.opengl.org/about/. [Accessed 24 10 2018].

[15] Apple, “What’s New in iOS - iOS 12,” [Online]. Available: https://developer.apple.com/ios/whats-new/. [Accessed 22 07 2018].

[16] Apple, “What’s New in macOS - macOS 10.14,” [Online]. Available: https://developer.apple.com/macos/whats-new/. [Accessed 22 07 2018].

[17] Khronos Group, “Khronos Releases Vulkan 1.0 Specification,” [Online]. Available: https://www.khronos.org/news/press/khronos-releases-vulkan-1-0-specification. [Accessed 24 10 2018].

[18] AMD, “One of Mantle's Futures: Vulkan,” [Online]. Available: https://community.amd.com/community/gaming/blog/2015/05/12/one-of-mantles-futures-vulkan. [Accessed 24 10 2018].

[19] The Brenwill Workshop Ltd, “Want to run Vulkan on iOS and macOS?,” [Online]. Available: https://moltengl.com/moltenvk/. [Accessed 04 06 2018].

[20] Khronos Group, “The Industry Open Standard Intermediate Language for Parallel Compute and Graphics,” [Online]. Available: https://www.khronos.org/spir/. [Accessed 14 06 2018].

[21] Microsoft, “SPIR-V CodeGen,” [Online]. Available: https://github.com/Microsoft/DirectXShaderCompiler/wiki/SPIR-V-CodeGen. [Accessed 14 06 2018].

[22] Apple, “Metal 2,” [Online]. Available: https://developer.apple.com/metal/. [Accessed 01 06 2018].

[23] Goodboy Digital Ltd, “PixiJS V4 The HTML5 Creation Engine,” [Online]. Available: http://www.pixijs.com. [Accessed 03 06 2018].

[24] F. Lardinois, “Google is launching playable in-game ads,” Techcrunch, [Online]. Available: https://techcrunch.com/2018/03/15/google-is-launching-playable-in-game-ads/. [Accessed 03 06 2018].

[25] A. Blacker, “Facebook Messenger Instant Games Are gaining Traction, Are You Ready?,” Apptopia, [Online]. Available: http://blog.apptopia.com/facebook-messenger-instant-games-are-gaining-traction-are-you-ready. [Accessed 03 06 2018].

[26] Defold, “Frequently asked questions,” [Online]. Available: https://www.defold.com/faq/. [Accessed 03 06 2018].

[27] Defold, “Features,” [Online]. Available: https://www.defold.com/technology/. [Accessed 03 06 2018].

[28] Defold, “Shaders,” [Online]. Available: https://www.defold.com/manuals/shader/. [Accessed 03 06 2018].

[29] Unity Technologies, “Company Facts,” Unity Technologies, [Online]. Available: https://unity3d.com/public-relations. [Accessed 2 11 2017].

[30] Bloomberg, “Company Overview of Unity Technologies, Inc.,” [Online]. Available: https://www.bloomberg.com/research/stocks/private/snapshot.asp?privcapId=241908542. [Accessed 25 10 2018].

[31] Unity Technologies, [Online]. Available: https://unity3d.com/unity/features/graphics-rendering. [Accessed 26 10 2018].

[32] Unity Technologies, “Features,” Unity Technologies, [Online]. Available: https://unity3d.com/unity. [Accessed 2 11 2017].

[33] Unity Technologies, “Editor,” Unity Technologies, [Online]. Available: https://unity3d.com/unity/editor. [Accessed 2 11 2017].

[34] Microsoft, “3-D Coordinate Systems,” [Online]. Available: https://msdn.microsoft.com/en-us/library/windows/desktop/bb324490(v=vs.85).aspx. [Accessed 27 05 2018].

[35] Khronos Group, “Coordinate Transformations,” [Online]. Available: https://www.khronos.org/opengl/wiki/Coordinate_Transformations. [Accessed 29 07 2018].

[36] G. Portelli, “World Coordinate Systems in 3ds Max, Unity and Unreal Engine,” [Online]. Available: http://www.aclockworkberry.com/world-coordinate-systems-in-3ds-max-unity-and-unreal-engine/. [Accessed 29 07 2018].

[37] E. Lengyel, Mathematics for 3D Game Programming and Computer Graphics, 3rd ed., Boston: Cengage Learning, 2012.

[38] Microsoft, “Primitives,” [Online]. Available: https://msdn.microsoft.com/en-us/library/windows/desktop/bb147291(v=vs.85).aspx. [Accessed 29 05 2018].

[39] Microsoft, “Geometry Shader Stage,” [Online]. Available: https://docs.microsoft.com/en-us/windows/desktop/direct3d11/geometry-shader-stage. [Accessed 10 10 2018].

[40] P. G. Vivo and J. Lowe, “Getting Started,” [Online]. Available: https://thebookofshaders.com/01/. [Accessed 6 11 2017].

[41] Microsoft, “Variables,” [Online]. Available: https://docs.microsoft.com/en-us/windows/desktop/direct3dhlsl/dx-graphics-hlsl-variables. [Accessed 04 11 2018].

[42] Microsoft, “Tessellation Stages,” [Online]. Available: https://docs.microsoft.com/en-us/windows/desktop/direct3d11/direct3d-11-advanced-stages-tessellation#domain-shader-stage. [Accessed 11 10 2018].

[43] Microsoft, “Primitive Topologies,” Microsoft, [Online]. Available: https://msdn.microsoft.com/en-us/library/windows/desktop/bb205124(v=vs.85).aspx#Primitive_Adjacency. [Accessed 27 11 2017].

[44] Khronos Group, “Compute Shader,” [Online]. Available: https://www.khronos.org/opengl/wiki/Compute_Shader. [Accessed 10 11 2017].

[45] Unity Technologies, “Compute Shaders,” Unity Technologies, [Online]. Available: https://docs.unity3d.com/Manual/ComputeShaders.html. [Accessed 28 11 2017].

[46] “GPU multi-texture sprite batching!,” [Online]. Available: https://medium.com/goodboy-digital/gpu-multi-texture-sprite-batching-21c90ae8f89b. [Accessed 12 10 2018].

[47] Unity Technologies, “Draw call batching,” [Online]. Available: https://docs.unity3d.com/Manual/DrawCallBatching.html. [Accessed 1 10 2018].

[48] PixiJS, “v4 Creating Filters,” [Online]. Available: https://github.com/pixijs/pixi.js/wiki/v4-Creating-Filters. [Accessed 08 07 2018].

[49] PixiJS, “pixi-plugin-example,” [Online]. Available: https://github.com/pixijs/pixi-plugin-example. [Accessed 02 07 2018].

[50] TazOen, “Helper function to create shader plugins for pixi.js,” [Online]. Available: https://github.com/TazOen/createShaderPlugin. [Accessed 11 10 2018].

[51] “Pixi.JS Releases v5.0.0-alpha.2,” 02 07 2018. [Online]. Available: https://github.com/pixijs/pixi.js/releases/tag/v5.0.0-alpha.2.

[52] Defold, “Shaders,” Defold, [Online]. Available: https://www.defold.com/manuals/shader/. [Accessed 23 11 2017].

[53] Defold, “The Render Pipeline,” [Online]. Available: https://www.defold.com/manuals/render/. [Accessed 23 06 2018].

[54] Defold, “Materials,” [Online]. Available: https://www.defold.com/manuals/material/. [Accessed 23 06 2018].

[55] Unity Techologies, “Unity's Rendering Pipeline,” Unity Techologies, [Online]. Available: https://docs.unity3d.com/Manual/SL-RenderPipeline.html. [Accessed 02 08 2018].

[56] Unity Technologies, “Forward Rendering Path Details,” [Online]. Available: https://docs.unity3d.com/Manual/RenderTech-ForwardRendering.html. [Accessed 02 08 2018].

[57] Unity Technologies, “Deferred shading rendering path,” [Online]. Available: https://docs.unity3d.com/Manual/RenderTech-DeferredShading.html. [Accessed 02 08 2018].

[58] Unity Technologies, “Scriptable Render Pipeline Overview,” [Online]. Available: https://blogs.unity3d.com/2018/01/31/srp-overview/. [Accessed 5 3 2018].

[59] Unity Technologies, “The High Definition Render Pipeline: Focused on visual quality,” [Online]. Available: https://blogs.unity3d.com/2018/03/16/the-high-definition-render-pipeline-focused-on-visual-quality/. [Accessed 02 08 2018].

[60] Unity Technologies, “The Lightweight Render Pipeline: Optimizing Real Time Performance,” [Online]. Available: https://blogs.unity3d.com/2018/02/21/the-lightweight-render-pipeline-optimizing-real-time-performance/. [Accessed 02 08 2018].

[61] Unity Technologies, “Introduction to Shader Graph: Build your shaders with a visual editor,” [Online]. Available: https://blogs.unity3d.com/2018/02/27/introduction-to-shader-graph-build-your-shaders-with-a-visual-editor/. [Accessed 21 05 2018].

[62] U. Technologies, “ShaderLab Syntax,” Unity Technologies, [Online]. Available: https://docs.unity3d.com/Manual/SL-Shader.html.

[63] Unity Technologies, “Writing Surface Shaders,” [Online]. Available: https://docs.unity3d.com/Manual/SL-SurfaceShaders.html.

[64] Unity Technologies, “Writing Shaders,” Unity Technologies, [Online]. Available: https://docs.unity3d.com/Manual/ShadersOverview.html. [Accessed 2 11 2017].

[65] Unity Technologies, “GLSL Shader programs,” [Online]. Available: https://docs.unity3d.com/Manual/SL-GLSLShaderPrograms.html. [Accessed 1 12 2017].

[66] Unity Technologies, “Shader Graph Window,” [Online]. Available: https://github.com/Unity-Technologies/ShaderGraph/wiki/Shader-Graph-Window. [Accessed 23 05 2018].

[67] Unity Technologies, “Blackboard,” [Online]. Available: https://github.com/Unity-Technologies/ShaderGraph/wiki/Blackboard. [Accessed 23 05 2018].

[68] Unity Technologies, “Master Preview,” [Online]. Available: https://github.com/Unity-Technologies/ShaderGraph/wiki/Master-Preview. [Accessed 23 05 2018].

[69] Unity Technologies, “Node,” [Online]. Available: https://github.com/Unity-Technologies/ShaderGraph/wiki/Node. [Accessed 23 05 2018].

[70] Unity Technologies, “Master Node,” [Online]. Available: https://github.com/Unity-Technologies/ShaderGraph/wiki/Master-Node. [Accessed 23 05 2018].

[71] Unity Technologies, “Port,” [Online]. Available: https://github.com/Unity-Technologies/ShaderGraph/wiki/Port. [Accessed 23 05 2018].

[72] Unity Technologies, “Edge,” [Online]. Available: https://github.com/Unity-Technologies/ShaderGraph/wiki/Edge. [Accessed 23 05 2018].

[73] Unity Technologies, [Online]. Available: https://github.com/Unity-Technologies/ShaderGraph/wiki/Sub-graph. [Accessed 23 05 2018].

[74] Unity Technologies, “Creating and Using Materials,” [Online]. Available: https://docs.unity3d.com/Manual/Materials.html. [Accessed 03 08 2018].

[75] Khronos Group, “gl_FragCoord — contains the window-relative coordinates of the current fragment,” 04 09 2018. [Online]. Available: https://www.khronos.org/registry/OpenGL-Refpages/gl4/html/gl_FragCoord.xhtml.

[76] Khronos Group, “distance — calculate the distance between two points,” [Online]. Available: https://www.khronos.org/registry/OpenGL-Refpages/gl4/html/distance.xhtml. [Accessed 04 09 2018].

[77] Khronos Group, “smoothstep — perform Hermite interpolation between two values,” [Online]. Available: https://www.khronos.org/registry/OpenGL-Refpages/gl4/html/smoothstep.xhtml. [Accessed 04 09 2018].

[78] Khronos Group, “mix — linearly interpolate between two values,” [Online]. Available: https://www.khronos.org/registry/OpenGL-Refpages/gl4/html/mix.xhtml. [Accessed 04 09 2018].

[79] Microsoft Corporation, “Premultiplied alpha,” [Online]. Available: https://microsoft.github.io/Win2D/html/PremultipliedAlpha.htm. [Accessed 08 09 2018].

[80] Unity Technologies, “Surface Shaders with DX11 / OpenGL Core Tessellation,” [Online]. Available: https://docs.unity3d.com/Manual/SL-SurfaceShaderTessellation.html. [Accessed 14 09 2018].

[81] Google Developers, “Timeline Event Reference,” [Online]. Available: https://developers.google.com/web/tools/chrome-devtools/evaluate-performance/performance-reference. [Accessed 08 10 2018].

[82] Google Developers, “ Analyze Runtime Performance,” [Online]. Available: https://developers.google.com/web/tools/chrome-devtools/rendering-tools/. [Accessed 16 09 2018].

[83] Spector.JS, “Spector.JS,” [Online]. Available: https://spector.babylonjs.com/. [Accessed 16 09 2018].

[84] T. Jukić, “Draw calls in a nutshell,” [Online]. Available: https://medium.com/@toncijukic/draw-calls-in-a-nutshell-597330a85381. [Accessed 1 10 2018].

[85] PixiJS, “PixiJS Filters,” [Online]. Available: https://github.com/pixijs/pixi-filters. [Accessed 02 07 2018].

[86] Unity Technologies, “Introduction to the Amplify Shader Editor,” [Online]. Available: https://unity3d.com/learn/tutorials/topics/asset-store/introduction-amplify-shader-editor. [Accessed 16 09 2018].

[87] F. Holmér, “Shader Forge,” [Online]. Available: http://www.acegikmo.com/shaderforge/. [Accessed 16 09 2018].

[88] Epic Games, “Material Editor Reference,” [Online]. Available: https://docs.unrealengine.com/en-us/Engine/Rendering/Materials/Editor. [Accessed 17 09 2018].

[89] Godot, “Visual Shader Editor is back,” [Online]. Available: https://godotengine.org/article/visual-shader-editor-back. [Accessed 17 09 2018].

APPENDIX 1 UNITY OUTLINE SHADER

```
Shader "ShaderPlayground/OutlineManual"
{
    Properties {
            _Color ("Color", Color) = (1,1,1,1)
            _MainTex ("Albedo (RGB)", 2D) = "white" {}
            _Glossiness ("Smoothness", Range(0,1)) = 0.5
            _Metallic ("Metallic", Range(0,1)) = 0.0
            _OutlineColor("Outline color", Color) = (0, 0, 0, 1)
            _OutlineWidth("Outline width", Range(0.0, 5.0)) = 0
    }
    CGINCLUDE
    #include "UnityCG.cginc"
    struct appdata
    {
            float4 vertex : POSITION;
            float3 normal : NORMAL;
    };
    struct v2f
    {
            float4 position : POSITION;
    };
    float _OutlineWidth;
    float4 _OutlineColor;
    v2f vert(appdata v)
    {
            v2f o;
            o.position = UnityObjectToClipPos(v.vertex);
            float3 normal = mul((float3x3)UNITY_MATRIX_IT_MV, v.normal);
            normal.x *= UNITY_MATRIX_P[0][0];
            normal.y *= UNITY_MATRIX_P[1][1];
            o.position.xy += normal.xy * _OutlineWidth;
            return o;
    }
    ENDCG
```

(continues)

APPENDIX 1 (continues)

```
    SubShader {
        Tags { "RenderType"="Transparent" }
        LOD 200
        Pass
        {
            ZWrite Off
            CGPROGRAM
            #pragma vertex vert
            #pragma fragment frag
            half4 frag(v2f i) : COLOR
            {
                return _OutlineColor;
            }
            ENDCG
        }
        CGPROGRAM
        #pragma surface surf Standard fullforwardshadows
        #pragma target 3.0
        sampler2D _MainTex;
        struct Input {
            float2 uv_MainTex;
        };
        half _Glossiness;
        half _Metallic;
        fixed4 _Color;
        void surf (Input IN, inout SurfaceOutputStandard o) {
            fixed4 c = tex2D (_MainTex, IN.uv_MainTex) * _Color;
            o.Albedo = c.rgb;
            o.Metallic = _Metallic;
            o.Smoothness = _Glossiness;
            o.Alpha = c.a;
        }
        ENDCG
    }
    FallBack "Diffuse"
}
```

APPENDIX 2 MOUSEDRAW SCRIPT

```
using UnityEngine;
public class MouseDraw : MonoBehaviour
{
    [Header("Displacement Material")]
    public Material drawMaterial;
    public float distance = 15;
    [Header("Brush Settings")]
    public float brushSize = 0.05f;
    public float brushStrength = 0.3f;
    public float brushEdge = 0.01f;

    private Camera _mainCamera;
    private Material _snowMaterial;
    private RenderTexture _renderTexture;

    private RaycastHit _raycastHit;

    private static readonly int brushSizeID = Shader.PropertyToID("_BrushSize");
    private static readonly int brushStrengthID = Shader.PropertyToID("_BrushStrength");
    private static readonly int brushEdgeID = Shader.PropertyToID("_BrushEdge");

    private int _layerMask;
    void Start ()
    {
            _mainCamera = Camera.main;
            _layerMask = LayerMask.GetMask("Terrain");
            _snowMaterial = GetComponent<MeshRenderer>().material;
            _renderTexture = new RenderTexture(1024, 1024, 0, RenderTextureFormat.ARGBFloat);
            drawMaterial.mainTexture = _renderTexture;
            _snowMaterial.SetTexture("_DisplacementTexture", _renderTexture);

            UpdateBrushSettings();
    }
```

(continues)

APPENDIX 2 (continues)

```
    void Update ()
    {
            if (Input.GetKey(KeyCode.Mouse0))
            {
                    Ray ray = _mainCamera.ScreenPointToRay(Input.mousePosition);

                    Debug.DrawRay(ray.origin, (ray.direction * distance), Color.green);

                    if (Physics.Raycast(ray, out _raycastHit, distance, _layerMask))
                    {
                            drawMaterial.SetVector("_BrushCoordinate", new
                            Vector4(_raycastHit.textureCoord.x, _raycastHit.textureCoord.y, 0, 0));
                            RenderTexture temporary =  RenderTexture.GetTemporary(_renderTexture.width,
                            _renderTexture.height, 0, RenderTextureFormat.ARGBFloat);
                            Graphics.Blit(_renderTexture, temporary);
                            Graphics.Blit(temporary, _renderTexture, drawMaterial);
                            RenderTexture.ReleaseTemporary(temporary);
                    }
            }
    }

    public void UpdateBrushSettings()
    {
            drawMaterial.SetFloat(brushSizeID, brushSize);
            drawMaterial.SetFloat(brushStrengthID, brushStrength);
            drawMaterial.SetFloat(brushEdgeID, brushEdge);
    }
}
```

APPENDIX 3 DISPLACEMENT SHADER

```
Shader "ShaderPlayground/DisplacementShader"
{
    Properties
    {
        _MainTex ("Texture", 2D) = "white" {}
        _BrushCoordinate ("Brush Coordinate", Vector) = (0, 0, 0, 0)
        _BrushSize ("Brush Size", Range(0, 0.5)) = 0.01
        _BrushStrength ("Brush Strength", Range(0, 1)) = 0.5
        _BrushEdge ("Brush Edge", Range(0.0, 0.5)) = 0.01
    }
    SubShader
    {
        Lighting Off
        Blend One Zero
        Pass
        {
            CGPROGRAM
            #pragma vertex vert
            #pragma fragment frag

            #include "UnityCG.cginc"

            struct appdata
            {
                float4 vertex : POSITION;
                float2 uv : TEXCOORD0;
            };

            struct v2f
            {
                float2 uv : TEXCOORD0;
                float4 vertex : SV_POSITION;
            };
            sampler2D _MainTex;
```

(continues)

APPENDIX 3 (continues)

```
                fixed4 _BrushCoordinate;
                half _BrushSize;
                half _BrushStrength;
                half _BrushEdge;

                float4 _MainTex_ST;

                v2f vert (appdata v)
                {
                        v2f o;
                        o.vertex = UnityObjectToClipPos(v.vertex);
                        o.uv = TRANSFORM_TEX(v.uv, _MainTex);
                        return o;
                }
                fixed4 frag (v2f i) : SV_Target
                {
                        float red = tex2D(_MainTex, i.uv).r;
                        half brushDistance = distance(i.uv, _BrushCoordinate.xy);
                        half draw = smoothstep(0.0, 1.0, 1 - saturate((brushDistance - _BrushSize) /
                        _BrushEdge)) * _BrushStrength;
                        half finalColor = lerp(red, 1.0, draw);
                        return finalColor;
                }
                ENDCG
        }
    }
}
```

APPENDIX 4 TESSELLATION SHADER

```
Shader "Shadergraph/Tesselation01" {
    Properties {
        _TessellationAmount ("Tessellation", Range(1,32)) = 4
        _MainColor ("Snow Color", color) = (1,1,1,1)
        _SecondaryColor ("Ground Color", color) = (0,0,0,1)
        _MainTexture ("Main Texture", 2D) = "white" {}
        _SecondaryTexture ("Ground Texture", 2D) = "white" {}
        _DisplacementTexture ("Displacement Texture", 2D) = "white" {}
        _Displacement ("Displacement", Range(0, 1.0)) = 0.3
        _TextureMixPercentage ("Texture Mix Percentage", Range(0, 1.0)) = 1.0
    }
    SubShader {
        Tags { "RenderType"="Opaque" }
        LOD 300
        CGPROGRAM
        #pragma surface surf BlinnPhong addshadow fullforwardshadows vertex:disp tessellate:tessDistance nolightmap
         #pragma target 4.6
        #include "Tessellation.cginc"
        struct appdata {
            float4 vertex : POSITION;
            float3 normal : NORMAL;
            float2 texcoord : TEXCOORD0;
        };
        float _TessellationAmount;
        float4 tessDistance (appdata v0, appdata v1, appdata v2) {
            float minDist = 10.0;
            loat maxDist = 25.0;
            return UnityDistanceBasedTess(v0.vertex, v1.vertex, v2.vertex, minDist, maxDist, _TessellationAmount);
        }
        sampler2D _DisplacementTexture;
```

(continues)

APPENDIX 4 (continues)

```
            float _Displacement;
            void disp (inout appdata v)
            {
                  float d = tex2Dlod(_DisplacementTexture, float4(v.texcoord.xy,0,0)).r * _Displacement;
                  v.vertex.xyz -= v.normal * d;
                  v.vertex.xyz += v.normal * _Displacement;
            }
            struct Input {
                  float2 uv_MainTexture;
                  float2 uv_SecondaryTexture;
                  float2 uv_DisplacementTexture;
            };
            sampler2D _MainTexture;
            sampler2D _SecondaryTexture;
            fixed4 _MainColor;
            fixed4 _SecondaryColor;
            float _TextureMixPercentage;
            void surf (Input IN, inout SurfaceOutput o) {
                  half amount = tex2Dlod(_DisplacementTexture, float4(IN.uv_DisplacementTexture, 0, 0)).r *
                  _TextureMixPercentage;
                  half4 mainTexture = tex2D (_MainTexture, IN.uv_MainTexture) * _MainColor;
                  half4 secondaryTexture = tex2D (_SecondaryTexture, IN.uv_SecondaryTexture) *
                  _SecondaryColor;
                  half4 c = lerp(mainTexture, secondaryTexture, amount);
                  o.Albedo = c.rgb;
                  o.Specular = 0.2;
                  o.Gloss = 1.0;
            }
            ENDCG
      }
      FallBack "Diffuse"
}
```