SAMUEL STYLES

Student # 135743 / samuel.styles93@gmail.com

BACHELOR OF SCIENCE

Games Programming

3D Fur generation

Rendering realistic fur in Unity using shell texturing.

MODULE 6GST0XF101.2 - Bachelor’s thesis

Farhan E., Siorak N.

Word Count: 11’059

19.07.2024

SAE INSTITUTE GENEVA

Bachelor’s thesis - MODULE 6GST0XF101.2

I, Samuel Styles, certify that I have personally written this Bachelor thesis.

I also certify that I have not resorted to plagiarism and have conscientiously and clearly mentioned all elements borrowed from others.

Styles

Geneva, 19 July 2024

Foreword

After multiple years of studies in commercial-related fields, I wanted to better understand the world and the way it worked; keep my mind active; accumulate knowledge and stay physically active. For those reasons I decided to change paths and go towards mechanical engineering.

During my studies as a mechatronics apprentice, I found exactly what I had wished for, I discovered so much about the world: physics, mechanics, electricity, electronics, thermodynamics and so much more. At that point in my life, my entire being, up until then only reproducing thoughtless tasks, was finally fully stimulated. But as every dream comes to an end, the actual daily work was only stimulating on rare occasions. Therefore, I decided to continue training and accumulate more knowledge by starting a federal diploma of higher education in mechatronics engineering. Regrettably, due to productivity requirements, tight deadlines, and finally physical exhaustion, I ended up injuring my back and having to stop the work and studies which I was so passionate about. At that moment I felt as if my life had been driven straight into a brick wall, but it gave me time to really think about what my principal motivation was.

Before entering the SAE, I started work experience as a game programmer at *DamaDamaGames*, a game development company in Lausanne. At that moment I really discovered my love for programming and my passion for visual effects and shader. So, I joined the SAE to have an academic training and understanding of what game programming really was. In my second year of studies, I worked on two projects that orientated me towards graphics programming: creating and optimising a handmade CPU Rasterizer from scratch and creating an OpenGL rendering engine. Arriving in my third year, my objective was clear, I wanted to work on a subject related to 3D fur rendering in a Real-time environment because it is visually pleasing but also technically interesting in terms of optimisation possibilities.

Acknowledgment

I would like to express my deepest gratitude to Elias Farhan, Head of Game Programming, for his invaluable guidance, technical expertise, and support throughout the development of my major project. His mentorship has been invaluable in shaping my skills and understanding in the field.

I am also grateful to Nicolas Siorak, my Bachelor’s teacher, for his insightful feedback and assistance during the writing process. His constructive criticism and encouragement have been extremely useful in refining my academic work.

I extend my heartfelt thanks to the staff of SAE Institute in Geneva for their warmth, hospitality, and accessibility. Their dedication to fostering a conducive learning environment has greatly enriched my educational experience.

Furthermore, I am immensely grateful to my fellow students, Fabian Huber and Johanna Palminha, for their support and collaboration during challenging times and to Julia Styles for her dedication and time towards the proofreading of this thesis. Their contributions have significantly enhanced my journey through this programme.

Lastly, I would like to acknowledge the collective efforts of all individuals who have contributed to my academic and personal growth.

Abstract

This study addresses visual issues with shell texturing in real-time environments, exploring improvements using HLSL shaders in Unity. Through AAA game examples, fur rendering techniques, and expert interviews, the research validated the project's relevance. Quantitative tests highlighted GPU frame time and memory usage as crucial optimization factors. The findings confirm real-time fur rendering feasibility and underscore the importance of context-specific shader design. Future research could explore other game engines, custom rendering engine development, and alternative methods, providing pathways for further academic inquiry.

Key words: Shell-Texturing, Video game, Fur rendering, Unity, URP, HLSL, Shaders, Graphics programming.

Table of Content

1. Introduction 1
2. State of the Art 5
    2.1 Definitions 5
    2.2 Current Fur rendering techniques 10
    2.3 Shell Texturing in the gaming industry 16
    2.4 Unity Shell Texturing Project 20
    2.5 Conclusion 23
3. Qualitative Analysis 24
    3.1 Interviews 24
    3.2 Project Analysis 25
    3.3 Conclusion 29
4. Test Protocol 30
    4.1 Definition of the used metrics 30
    4.2 Measurement Tools 33
    4.3 Conclusion 38
5. Practical Project 39
    5.1 Development environment 39
    5.2 Limitations 41
    5.3 Projects procedure 42
    5.4 Technical implementation 47
    5.5 Conclusion 58
6. Quantitative Analysis 59
    6.1 Measurement conditions 60
    6.2 Project Measurements 60
    6.3 Project Analysis 71
    6.4 Conclusion 73
7. Conclusion and further research 74
8. References 76
    8.1 Articles 76
    8.2 Bibliography 77

8.3 Filmography 77
8.4 Webography 78
9. Figures 85
10. Appendices 88
Appendix A: Interview of David Sena 88
Appendix B: Interview of ChatGPT 89
Appendix C: Shell_Original.shader code 91
Appendix D: Shell.shader code 93
Appendix E: SimpleShell.cs code 96
Appendix F: SimpleCameraController.cs code 99
Appendix G: HLSLGeom.shader code 100
Appendix H: Fur.hlsl code 102
Appendix I: Common.hlsl code 105
Appendix J: HLSLComplex.shader code 106
Appendix K: FurComplex.hlsl code 108
Appendix L: HLSLGeom_Controller.cs script 111
Appendix M: HLSLComplex_Controller.cs script 112
Appendix N: FurSceneManager.cs script 113
Appendix O: SO_SceneData.cs script 115
Appendix P: DataExtractor.cs script 115
Appendix Q: Extracted Data spreadsheet 116

Glossary

2D:

A two-dimensional space is a mathematical space with two dimensions, meaning points have two degrees of freedom: their locations can be locally described with two coordinates, or they can move in two independent directions.

3D:

A 3D environment is a three-dimensional representation of geometric data that resembles the real world. It is simply a digital setting or background created with computer graphics software. It can be an indoor or outdoor location with realistic or non-realistic objects that give you visual sensations.

AAA:

In the video game industry, AAA (Triple-A) is an informal classification used to classify video games produced and distributed by a mid-sized or major publisher, which typically have higher development and marketing budgets than other tiers of games.

ACES tonemapping:

Academy Color Encoding System (ACES) tonemapping can (unsurprisingly) be described as filmic. This means that it applies a sigmoid-style curve in a logarithmic space to produce the tone mapping. Further, it performs the operations mostly independently on the color channels rather than operating strictly on luminance like many other operators. This characteristic makes the operator naturally desaturate when it reaches the limits of the present adaptation level. Highlights gently roll toward white, and shadows gently roll toward black.

AI:

Artificial intelligence (AI) is the simulation of human intelligence processes by machines, especially computer systems. Specific applications of AI include expert systems, natural language processing, speech recognition and machine vision.

API:

Application Programming Interface, the part of a technical solution exposed for use by a user application.

Bloom:

Bloom (sometimes referred to as light bloom or glow) is a computer graphics effect used in video games, demos, and high-dynamic-range rendering (HDRR) to reproduce an imaging artifact of real-world cameras. The effect produces fringes (or feathers) of light extending from the borders of bright areas in an image, contributing to the illusion of an extremely bright light overwhelming the camera or eye capturing the scene. It became widely used in video games after an article on the technique was published by the authors of Tron 2.0 in 2004.

C++:

C++ is a cross-platform language that can be used to create high-performance applications. C++ was developed by Bjarne Stroustrup, as an extension to the C language. C++ gives programmers a high level of control over system resources and memory.

C#:

C# (C Sharp) is a general-purpose high-level programming language supporting multiple paradigms. C# encompasses static typing, strong typing, lexically scoped, imperative, declarative, functional, generic, object-oriented (class-based), and component-oriented programming disciplines.

CG:

Cg (short for C for Graphics) and High-Level Shader Language (HLSL) are two names given to a high-level shading language developed by Nvidia and Microsoft for programming shaders. Cg/HLSL is based on the C programming language and although they share the same core syntax, some features of C were modified, and new data types were added to make Cg/HLSL more suitable for programming graphics processing units.

CPU:

A central processing unit (CPU) is a hardware component that's the core computational unit in a server. Servers and other smart devices convert data into digital signals and perform mathematical operations on them.

Direct X:

DirectX is a suite of multimedia technologies required by many Windows games. If your PC does not have the right version of DirectX installed (the product box should tell you which one you need), your game might not work properly.

Geometry Shader:

The Geometry Shader (GS) stage processes entire primitives: triangles, lines, and points, along with their adjacent vertices. It is useful for algorithms including Point Sprite Expansion, Dynamic Particle Systems, and Shadow Volume Generation. It supports geometry amplification and de-amplification.

GPU:

Graphics Processing Unit, a specialised electronic circuit dedicated to accelerating visual processing algorithms.

Global illumination:

Global illumination (GI), or indirect illumination, is a group of algorithms used in 3D computer graphics that are meant to add more realistic lighting to 3D scenes. Such algorithms take into account not only the light that comes directly from a light source (direct illumination), but also subsequent cases in which light rays from the same source are reflected by other surfaces in the scene, whether reflective or not (indirect illumination).

GLSL:

OpenGL Shading Language (GLSL) is a high-level shading language with a syntax based on the C programming language. It was created by the OpenGL ARB (OpenGL Architecture Review Board) to give developers more direct control of the graphics pipeline without having to use ARB assembly language or hardware-specific languages.

HDRP:

The High-Definition Render Pipeline (HDRP) is a high-fidelity Scriptable Render Pipeline built by Unity to target modern (Compute Shader compatible) platforms.

HLSL:

The High-Level Shader Language (HLSL) is a proprietary shading language developed by Microsoft for the Direct3D 9 API to augment the shader assembly language and went on to become the required shading language for the unified shader model of Direct3D 10 and higher.

IDE:

An integrated development environment (IDE) is a software application that helps programmers develop software code efficiently. It increases developer productivity by combining capabilities such as software editing, building, testing, and packaging in an easy-to-use application.

LOD:

In computer graphics, level of detail (LOD) refers to the complexity of a 3D model representation. LOD can be decreased as the model moves away from the viewer or according to other metrics such as object importance, viewpoint-relative speed or position.

Pass:

A Pass is the fundamental element of a Shader object. It contains instructions for setting the state of the GPU, and the shader programmes that run on the GPU. Simple Shader objects might contain only a single Pass, but more complex shaders can contain multiple Passes.

Physically based rendering (PBR):

Physically based rendering (PBR) is a computer graphics approach that aims at rendering images in a way that models the lights and surfaces with optics in the real world. It is often referred to as "Physically Based Lighting" or "Physically Based Shading".

Post-processing:

The term post-processing (or postproc for short) is used in the video and film industry for quality-improvement image processing (specifically digital image processing) methods used in video playback devices, such as stand-alone DVD-Video players; video playing software; and transcoding software. It is also commonly used in real-time 3D rendering (such as in video games) to add additional effects.

Prefab:

Unity’s Prefab system is used to create, configure, and store a GameObject complete with all its components, property values, and child GameObjects as a reusable Asset.

Real-Time:

A real-time system has been described as one which controls an environment by receiving data, processing them, and returning the results sufficiently quickly.

RenderDoc:

RenderDoc is a free and open-source frame debugger that can be used to analyse single frames generated by other software programmes such as games.

Render Pipeline:

The computer graphics pipeline, also known as the rendering pipeline or graphics pipeline, is a framework within computer graphics that outlines the procedures necessary for transforming a three-dimensional scene into a two-dimensional representation on a screen.

Rim light:

A rim light is placed behind a subject that exposes the outline or rim of the subject with light. This lighting highlights the contours of a subject and creates a dramatic and mysterious effect.

Script:

A script is a sequence of instructions that can be executed by a computer or programming language. A script is a common type of computer programme, its defining characteristic being that it does not have to be compiled in advance of being run. It is interpreted and executed in real time.

SRP:

Unity’s Scriptable Render Pipeline (SRP) is a feature used to control rendering via C# scripts. SRP is the technology that underpins the Universal Render Pipeline (URP) and the High Definition Render Pipeline (HDRP).

Shader:

In computer graphics, a shader is a computer programme that calculates the appropriate levels of light, darkness, and colour during the rendering of a 3D scene—a process known as shading. Shaders have evolved to perform a variety of specialised functions in computer graphics special effects and video post-processing, as well as general-purpose computing on graphics processing units.

Shader Graph:

Shaders can be built visually with Shader Graph Instead of writing code, nodes can be created and connected in a graph framework. Shader Graph gives instant feedback that reflects changes, and it is simple enough for users who are new to shader creation.

Shell texturing:

Shell texturing involves creating mesh layers, then using one or more alpha textures to create the effect of volume. It looks quite good when used for rendering fur and can look good for grass as well.

SubShader:

SubShaders are useful for defining different GPU settings and shader programmes for different hardware, render pipelines, and runtime settings. Some Shader objects contain only a single SubShader; others contain multiple SubShaders to support a range of different configurations.

Subsurface scattering:

Subsurface scattering (SSS), also known as subsurface light transport (SSLT),[1] is a mechanism of light transport in which light that penetrates the surface of a translucent object is scattered by interacting with the material and exits the surface potentially at a different point.

Tracy:

Tracy is a real time, nanosecond resolution frame profiler that can be used for remote or embedded telemetry of your application. It can profile CPU (C++, Lua), GPU (OpenGL, Vulkan) and memory. It can also display locks held by threads and their interactions with each other.

UI:

User interface (UI) design is the process designers use to build easy-to-use and pleasurable interfaces in software or computerised devices.

Unity:

Unity is a cross-platform game engine developed by Unity Technologies […]. It is particularly popular for iOS and Android mobile game development, is considered easy to use for beginner developers, and is popular for indie game development.

Unreal Engine:

Unreal Engine (UE) is a series of 3D computer graphics game engines developed by Epic Games. Unreal Engine is written in C++ and features a high degree of portability, supporting a wide range of desktop, mobile, console, and virtual reality platforms.

URP:

The Universal Render Pipeline (URP) is a prebuilt Scriptable Render Pipeline, made by Unity. URP provides artist-friendly workflows that let you quickly and easily create optimised graphics across a range of platforms, from mobile to high-end consoles and PCs.

UV:

UV mapping is the 3D modelling process of projecting a 3D model's surface to a 2D image for texture mapping. The letters "U" and "V" denote the axes of the 2D texture because "X", "Y", and "Z" are already used to denote the axes of the 3D object in model space, while "W" (in addition to XYZ) is used in calculating quaternion rotations, a common operation in computer graphics.

Vignetting:

In photography and optics, vignetting is a reduction of an image's brightness or saturation toward the periphery compared to the image centre. The word vignette, from the same root as vine, originally referred to a decorative border in a book.

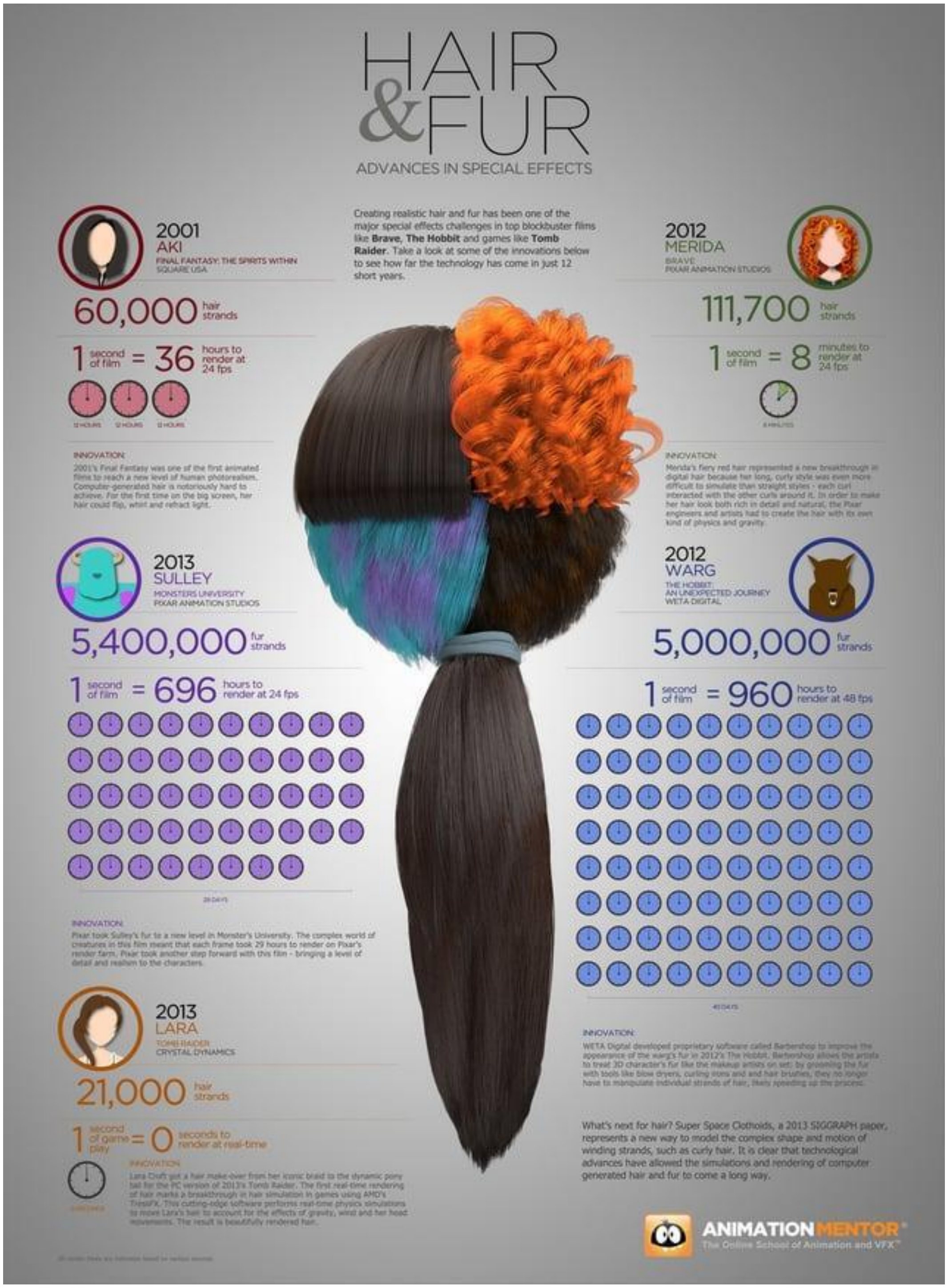
VSync:

Short for vertical sync, VSync is the graphics technology responsible for synchronizing the frame rate of a game to the refresh rate of a monitor. This synchronization delivers smooth, uninterrupted gameplay for graphics-heavy 3D games.

1. Introduction

Dark Souls III the “fastest selling title in the history of Bandai Namco Entertainment America” (Porter, 2017) was released with graphical issues regarding fur. Four years later, *Genshin Impact* “reported to have the highest revenue ever for a game in its first year” (Tyler, 2021) is released with the same problems when rendering grass. This observation is the reason that motivates this research around what is commonly known as Shell-Texturing.

With the constant improvement of hardware, the last decade has seen major innovations in the way graphics are rendered. Many technologies are currently used and developed throughout diverse domains that require the use of graphical illustrations. From the gaming industry that must render an image in a few milliseconds or the visual effects and film industry that will tend to expose the most perfect image possible, all require their own specific hardware, software, and tools. Regarding the generation of fur, it is necessary to differentiate the practices related to each domain. In the film industry, where images are pre-calculated and the experience is already scripted, there is no need to meet real-time requirements, thus enabling the possibility of pushing the boundaries in terms of visual rendering by directly sculpting or modelling hair or fur with a high level of detail. The necessary adjustments, such as physical interactions, can be calculated during long periods of time without impacting the way it is perceived by the viewer.



Creating realistic hair and fur has been one of the major special effects challenges in top blockbuster films like **Brave**, **The Hobbit** and games like **Tomb Raider**. Take a look at some of the innovations below to see how far the technology has come in just 12 short years.

INNOVATION:
2001's Final Fantasy was one of the first animated films to reach a new level of human photorealism. Computer-generated hair is notoriously hard to achieve. For the first time on the big screen, her hair could flip, whirl and refract light.

INNOVATION:
Merida's fiery red hair represented a new breakthrough in digital hair because her long, curly style was even more difficult to simulate than straight styles - each curl interacted with the other curls around it. In order to make her hair look both rich in detail and natural, the Pixar engineers and artists had to create the hair with its own kind of physics and gravity.

INNOVATION:
Pixar took Sulley's fur to a new level in Monster's University. The complex world of creatures in this film meant that each frame took 29 hours to render on Pixar's render farm. Pixar took another step forward with this film - bringing a level of detail and realism to the characters.

INNOVATION:
WETA Digital developed proprietary software called Barbershop to improve the appearance of the warg's fur in 2012's The Hobbit. Barbershop allows the artists to treat 3D character's fur like the makeup artists on set: by grooming the fur with tools like blow dryers, curling irons and and hair brushes, they no longer have to manipulate individual strands of hair, likely speeding up the process.

INNOVATION:
Lara Croft got a hair make-over from her iconic braid to the dynamic pony tail for the PC version of 2013's Tomb Raider. The first real-time rendering of hair marks a breakthrough in hair simulation in games using AMD's TressFX. This cutting-edge software performs real-time physics simulations to move Lara's hair to account for the effects of gravity, wind and her head movements. The result is beautifully rendered hair.

What's next for hair? Super Space Clothoids, a 2013 SIGGRAPH paper, represents a new way to model the complex shape and motion of winding strands, such as curly hair. It is clear that technological advances have allowed the simulations and rendering of computer generated hair and fur to come a long way.

Figure 1 - Hair and fur Render Time on characters. (Edgardlop, 2013)

On the other hand, the gaming industry requires the image to change dynamically according to events such as the user’s input.

That significant difference adds a layer of complexity when it comes to fur rendering. The outcome is that, in a real-time environment, rendering hair or fur is either created with a complex mesh but is intended to be static in game, meaning that there are no physical interactions with the fur or hair, or with simple planes or guides, where visual fidelity is discarded beforehand but physical interactions are possible. In both scenarios there is a trade-off between visual quality and fidelity of physical behaviour. This trade-off is precisely what will be measured and the main reason for the research that will be pursued in this thesis.

The primary goal is to develop a Unity project using the URP render pipeline to implement Shell-Texturing through HLSL shaders. Following this initial step, the same shader will be employed across different scenes of varying complexity. Subsequently, an executable will be generated, exported, and tested on multiple platforms to collect data for comparative analysis. The overarching goal is to enhance the shader project by leveraging the gathered data, thus refining it, and presenting an optimised setup that carefully balances trade-offs for optimal performance and visual fidelity.

The plan will start with the state of the art, providing a comprehensive definition and precise explanations of the key terms involved. Following this, a paragraph will delve into various examples of use of the shell-texturing technique within the AAA game industry.

To conclude, a brief introduction to the project and its procedure will be done. Once the foundational overview and historical context are established, the project will be described in detail including the metrics, software, hardware, procedure, and technical implementation. Lastly an examination of the project's results will be conducted leading to the final chapter of this thesis, the paths to further research.

The initial approach was to create three projects: a Unity projects using the URP pipeline to mimic *Genshin Impact*’s style; an Unreal Engine project to approach the rendering style of *Dark Souls III*; a DirectX12 rendering engine to implement necessary techniques by hand and have better control and understanding of the entire pipeline. This approach proved to be too challenging and had to be scaled back to fit into the time available. The Unreal Engine project was therefore abandoned since it was more oriented towards technical art than graphics programming skills. Regarding the DirectX12 project, after beginning with tutorials and a simple triangle rendering project, the amount of time required was evaluated and projected to be overly time-consuming. Thus, making it impossible to fulfil in an acceptable manner. For these reasons, the project focuses on the Unity version and its seamless exportation to multiple platforms. Delving deeply into techniques within Unity's Universal Render Pipeline (URP), will facilitate a comprehensive grasp and practical application of graphics programming principles.

2. State of the Art

2.1 Definitions

2.1.1 Fur in video games

Fur in the context of video games refers to the simulated representation of animal pelage, providing a lifelike and visually compelling aspect to virtual creatures. It involves the depiction of dense hair-like structures on the surface of characters or objects, enhancing their realism and adding depth to the gaming experience. Fur rendering techniques aim to capture the intricate details of fur, such as its length, colour, and overall appearance, contributing to the aesthetics of the virtual world (Kajiya & von Herzen, 1984).

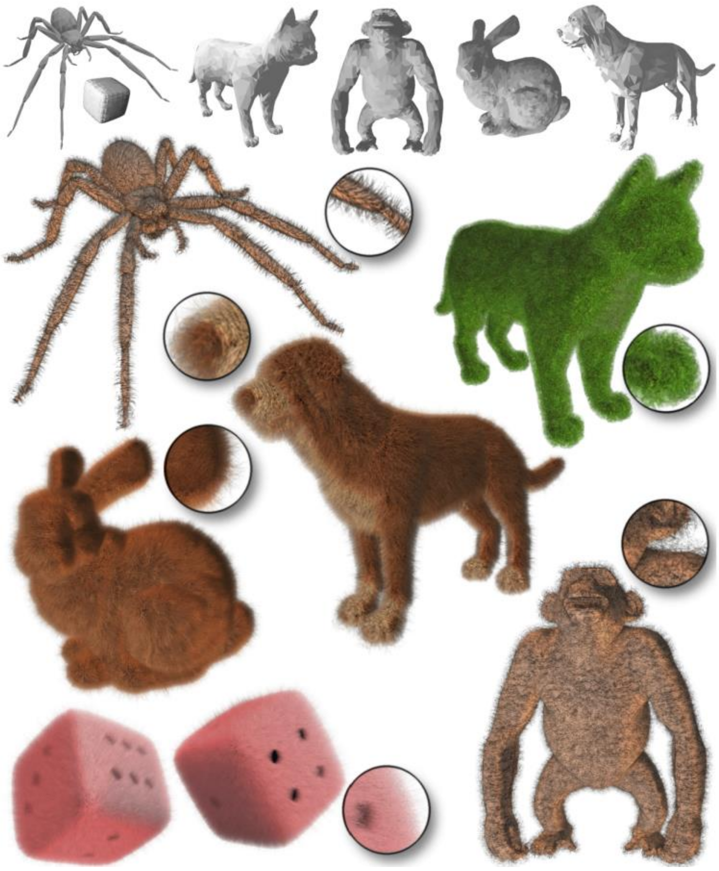


Figure 2 - Fur generation example from Real-Time Fur over Arbitrary Surfaces. (Lengyel et al., 2001)

When referring to fur, it is important to exclude aspects related to human hair or hairstyles, grooming, and similar elements. While these aspects are essential in various digital representations, the intention here is to dive deeply into the technical and artistic challenges associated with replicating the unique characteristics of animal fur.

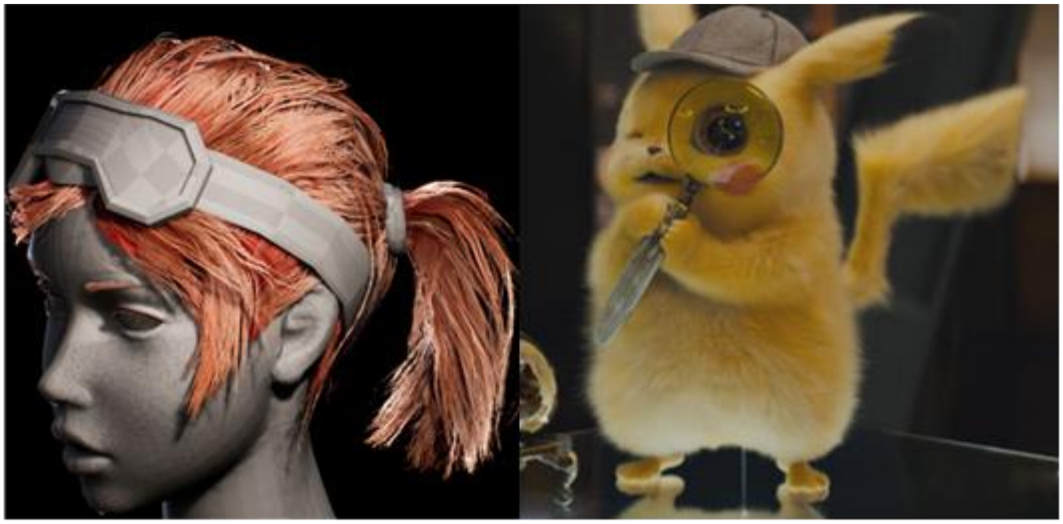


Figure 3 - 80.lv student hair for games from Greg Mourino. (80.lv, 2018) & Detective Pikachu film caption. (Machkovech, 2019)

By setting aside discussions on human hair, the exploration can be focused on the intricacies of rendering fur in a gaming environment. This exclusion allows for a more concentrated examination of the complexities involved in simulating animal pelage, providing a clear understanding of the specialised field of fur rendering in video game development.

2.1.2 Realistic rendering

Rendering is the process of generating an image from a model by simulating how light interacts with surfaces. It involves calculations to determine the colour, shadow, texture, and visual attributes of the objects in a scene. Rendering is essential in computer graphics for creating realistic or stylised visuals in various applications like animation, gaming, and virtual reality (Shirley et al, 2021).

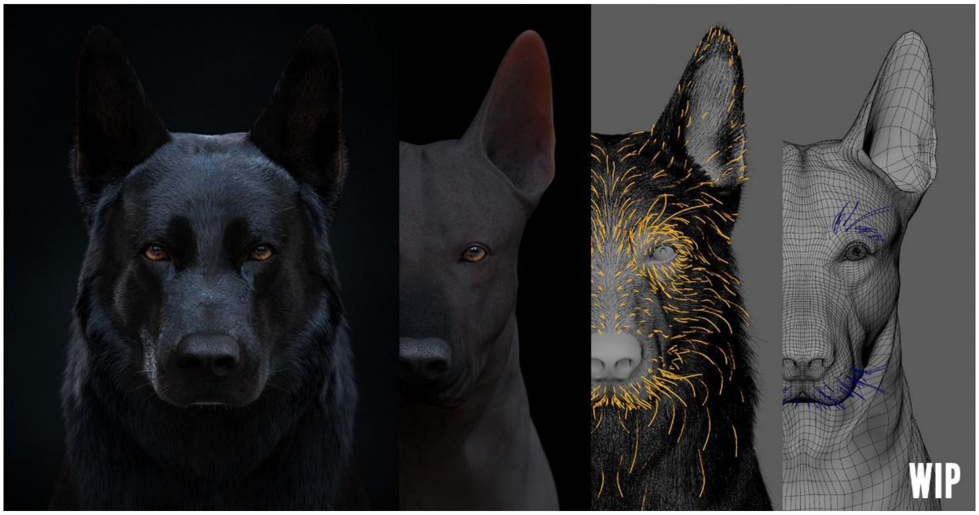
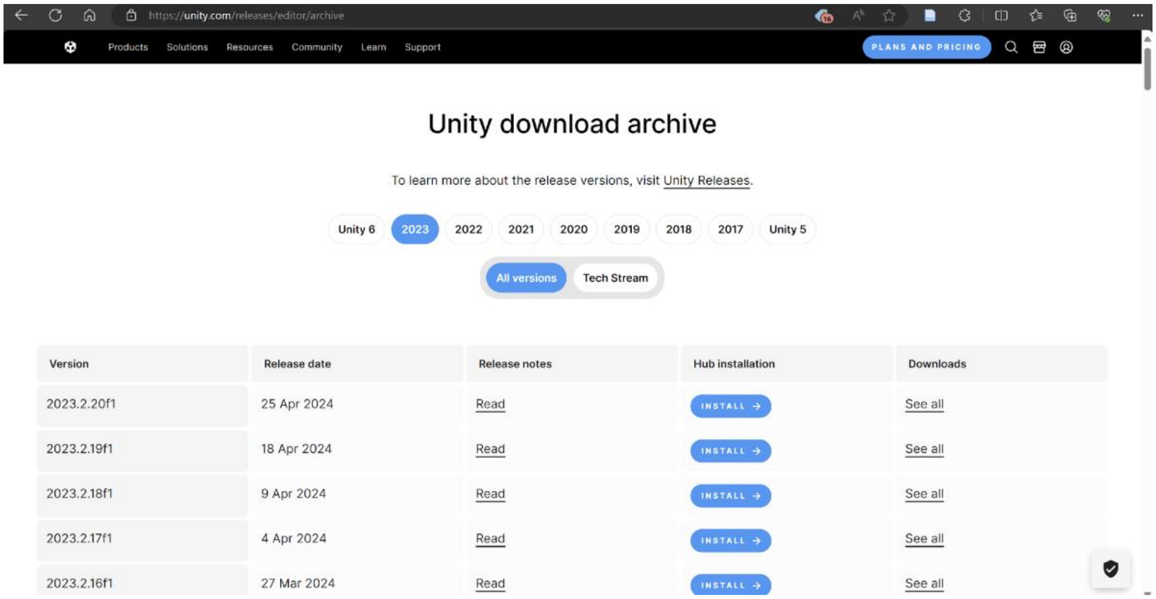


Figure 4 - Realistic Dog Portrait: Experimenting with Real-Time Fur. (80.lv, 2021)

When referring to realistic rendering, the focus lies on accurately simulating how light behaves in the real world to create lifelike images. This involves complex calculations to model light interactions such as reflection, refraction, and absorption on surfaces. By meticulously calculating these phenomena, realistic rendering aims to produce visuals that closely resemble how objects appear in the physical environment, adding depth, detail, and authenticity to computer-generated imagery (Adobe, n.d.; Chaos, n.d.; Bluebird International, n.d.).

2.1.3 Unity

Unity is a powerful and widely used game engine that allows creators to build interactive experiences for various platforms. It provides a user-friendly suite of tools for designing, prototyping, and deploying games and interactive content. Since 2015, Unity has developed versions of its engine, each with its own set of tools and improvements. These versions are archived and available on their website ("Download archive”, n.d.-b).



Unity download archive

To learn more about the release versions, visit Unity Releases.

Version	Release date	Release notes	Hub installation	Downloads
2023.2.20f1	25 Apr 2024	Read	INSTALL →	See all
2023.2.19f1	18 Apr 2024	Read	INSTALL →	See all
2023.2.18f1	9 Apr 2024	Read	INSTALL →	See all
2023.2.17f1	4 Apr 2024	Read	INSTALL →	See all
2023.2.16f1	27 Mar 2024	Read	INSTALL →	See all

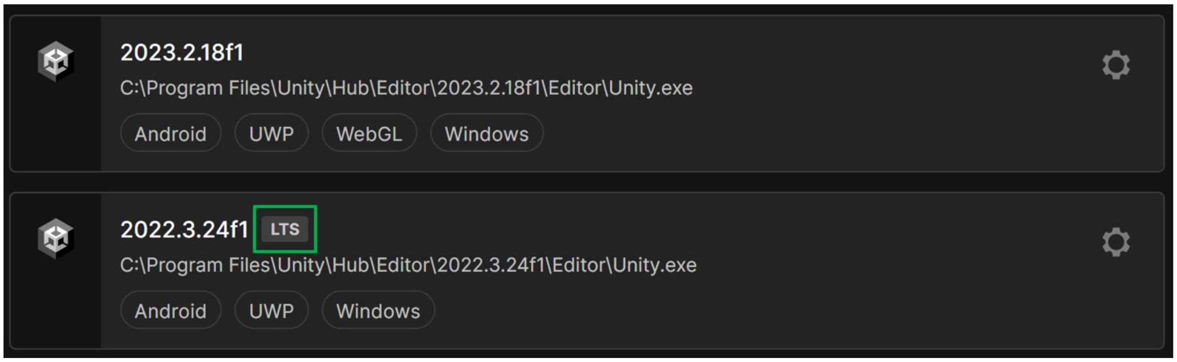
Figure 5 - Unity download archive. (Unity Technologies, n.d.-b)

Since Unity is continuously improving (Figure 5), these versions are categorised in two ways:

- Tech Stream: considered as unsafe:

> *“It is for creators who value getting earlier access to new features to prepare for future projects. These versions are primarily recommended for the preproduction, discovery, and prototyping phases of development, but they can be used to get ready for the next LTS by enabling earlier feature adoption”* (“LTS vs Tech Stream: Choose the right Unity release for you.”, n.d.-d).

- Long-Term Support: considered safe: “It is the release for creators who value maximum stability and support for their next project” (“LTS vs Tech Stream: Choose the right Unity release for you.”, n.d.-d).



2023.2.18f1
C:\Program Files\Unity\Hub\Editor\2023.2.18f1\Editor\Unity.exe
Android UWP WebGL Windows

2022.3.24f1 LTS
C:\Program Files\Unity\Hub\Editor\2022.3.24f1\Editor\Unity.exe
Android UWP Windows

Figure 6 - Unity Hub 3.7.0 Installs screenshot, (Styles, 2024)

2.1.3.1 Unity’s graphics pipelines and their differences:

Regarding the graphics pipelines, there are three main setups:

- BRP: “This pipeline offers broad compatibility and ease of use, suitable for most projects. It provides a balance between visual quality and performance, with features like dynamic lighting, shadows, and post-processing effects.” (“Using the Built-in Render Pipeline”, n.d.-g).
- URP: “Formerly known as the Lightweight Render Pipeline (LWRP), URP is optimised for performance on a wide range of platforms, including mobile devices and low-end hardware. It emphasises efficiency and scalability while still supporting many modern rendering features.” (“Universal Render Pipeline overview”, n.d.-e).

- HDRP:

  “HDRP is designed for projects requiring high visual fidelity, such as AAA games or advanced architectural visualization. It relies on cutting-edge rendering techniques like physically based rendering, real-time global illumination, subsurface scattering, and high-quality post-processing effects to achieve stunning visuals.” ("Create high-quality graphics and stunning visuals”, n.d.-a).

Each pipeline offers different trade-offs in terms of visual quality, performance, and supported features. Choosing the correct pipeline to use is crucial according to the desired project.

2.2 Current Fur rendering techniques

2.2.1 Shell-Texturing

The Shell-Texturing technique focuses on creating multiple shells around the object allowing for efficient and realistic rendering of fur without the need to individually model each hair strand. This method plays a crucial role in achieving convincing fur effects in video games, enriching the visual narrative, and pushing the boundaries of graphical realism (Döllner, Hinkenjann, & Wiemker, 2006).

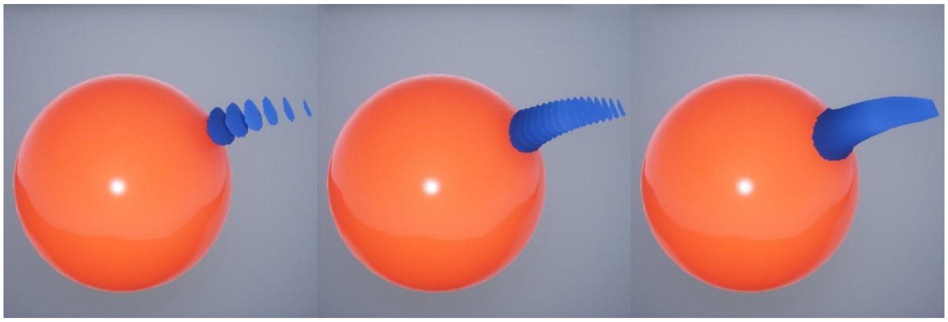


Figure 7 – A shell-based fur strand example. (GiM, n.d.)

This method stands out for its remarkable efficiency in rendering fur, as it cleverly constructs a shell around the object. This approach eliminates the need to model each individual hair strand, making it a highly optimised solution for various applications, especially in the context of video games (Döllner, Hinkenjann, & Wiemker, 2006).

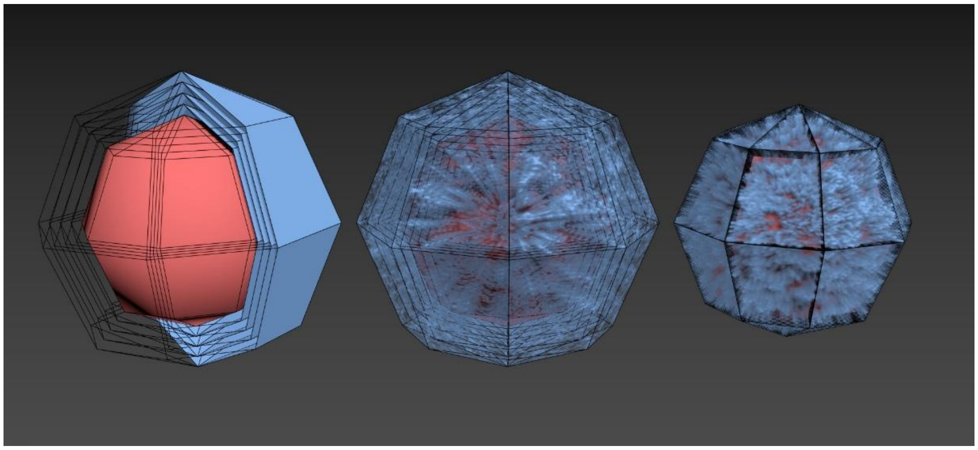


Figure 8 - Visualisation of shells on low-poly mesh. (GiM, n.d.)

However, it is essential to acknowledge that while Shell-Texturing excels in terms of optimisation, it does come with certain visual trade-offs that are related to the camera's perspective. Depending on the viewing angle and distance, some imperfections may become apparent, impacting the overall visual quality. Despite these considerations, the Shell-Texturing technique remains a powerful tool in terms of graphical rendering, offering a compelling balance between performance and aesthetics. (XBDEV.net, n.d.; NVIDIA, n.d.; Kajiya & Lischinski, 1993; "Fur shader", n.d.-c; Lengyel et al., 2001).

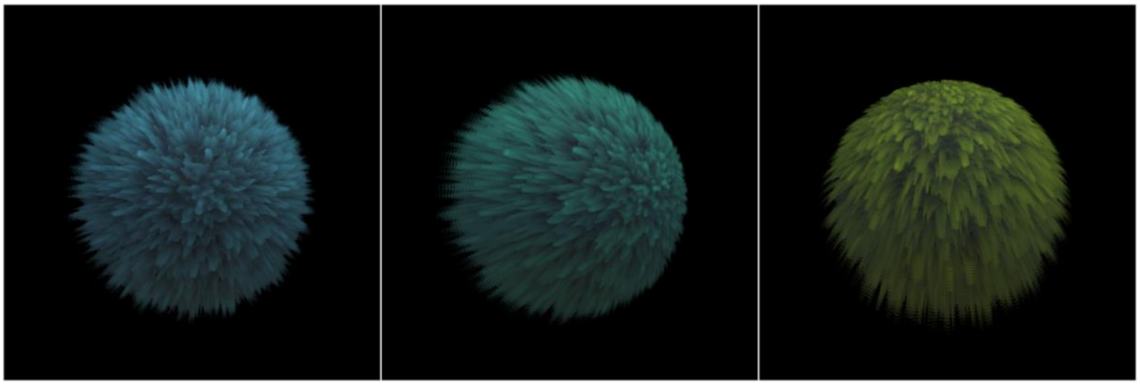


Figure 9 - Example of coloured Shell-based fur. (Afanasev, 2018)

2.2.2 Fin

The fin technique consists in generating extra geometry inside a triangle, composition of vertices, and extruding what is commonly referred to as a fin, to later apply a 2D texture map on that extruded part.

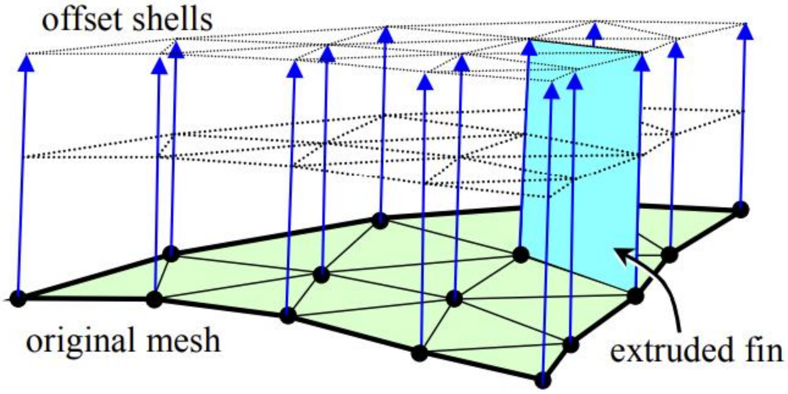
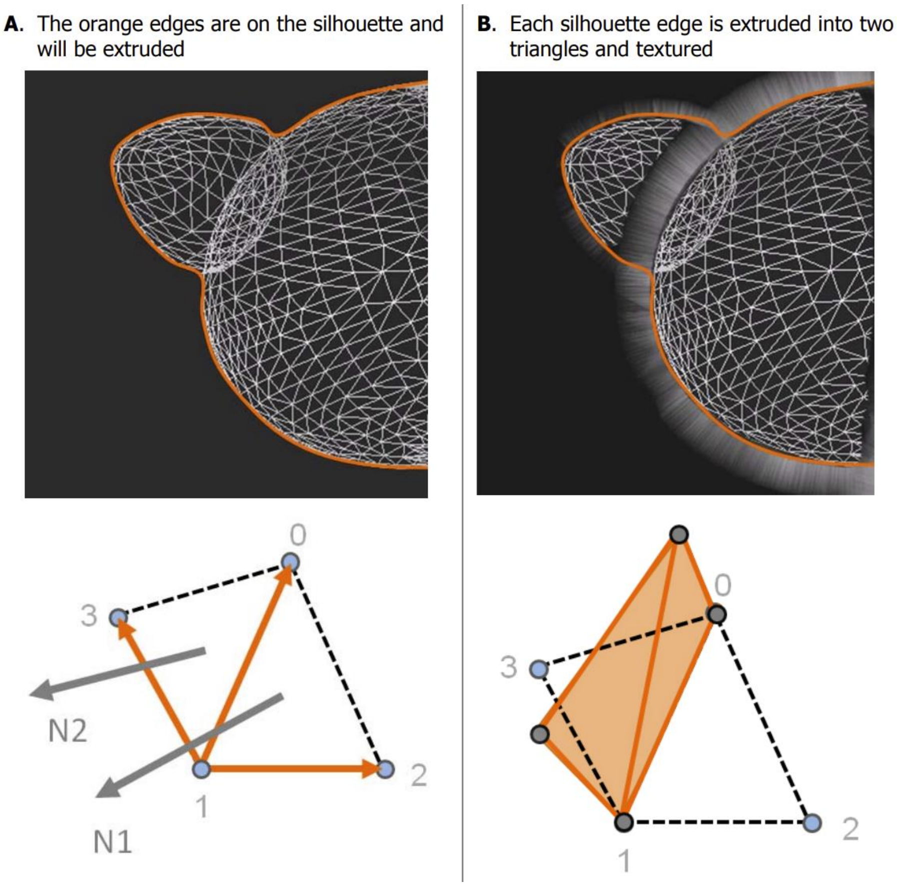


Figure 10 - Illustration of fin extrusion. (Lengyel et al., 2001)

As explained by Lengyel & al in 2001 in the paper *Real-Time Fur over Arbitrary Surfaces*:

“We place “fins” normal to the surface and render these using conventional 2D texture maps sampled from the volume texture in the direction of hair growth. The method generates convincing imagery of fur at interactive rates for models of moderate complexity. Further-more, the scheme allows real-time modification of viewing and lighting conditions, as well as local control over hair color, length, and direction, this technique already allowed its users to have a lot of control over the result.” (Lengyel et al., 2001)

Nvidia later published a white paper about this technique, improving it thanks to graphics programming modernization and DirectX10 allowing the use of geometry shaders as mentioned in Nvidia's white paper: "Fins are rendered by creating new geometry per frame along the silhouette edges of a mesh. Before DirectX 10 this required adding degenerate triangles to the mesh at every edge but now fins can be generated efficiently using the Geometry Shader." (Tariq, 2007)



A. The orange edges are on the silhouette and will be extruded

B. Each silhouette edge is extruded into two triangles and textured

Figure 11 - Fin generation example and illustration. (Tariq, 2007)

2.2.3 Polygon

The polygon method consists in manually positioning planes or polygons to later project 2D textures on them to obtain hair or fur.

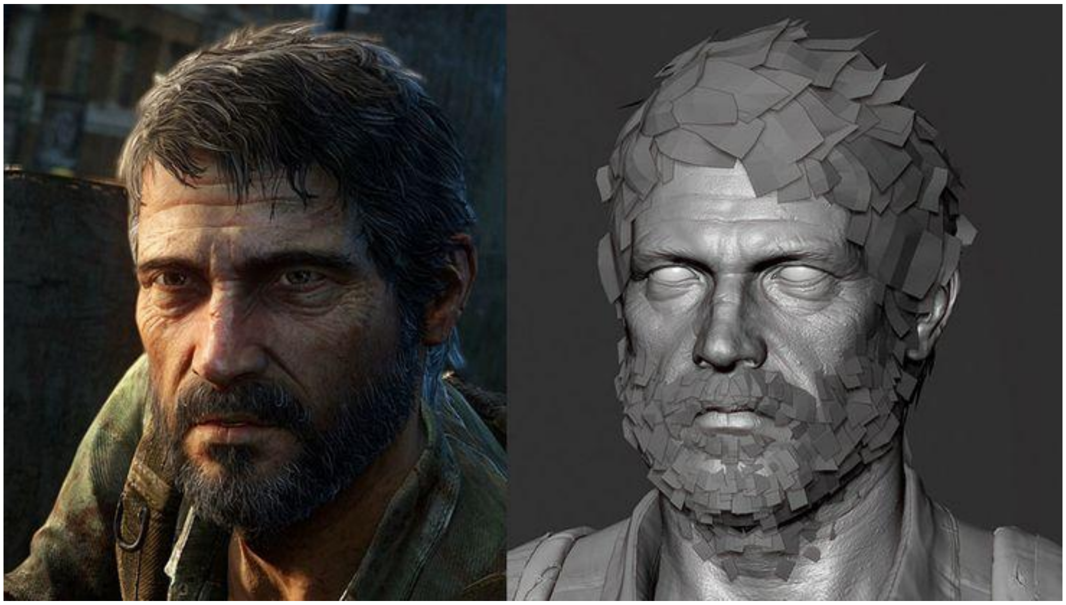


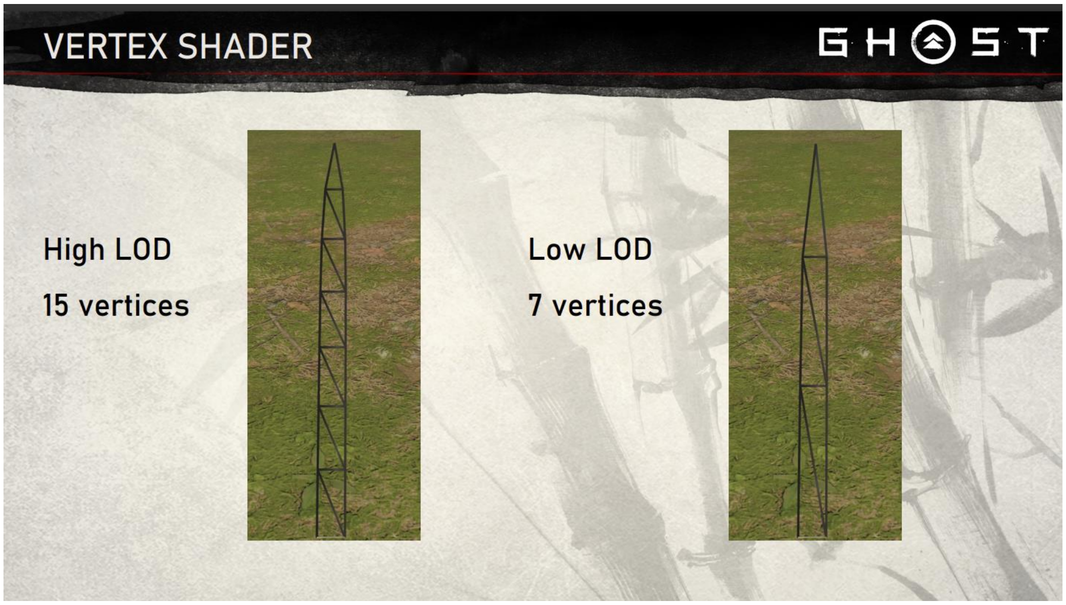
Figure 12 - Polygonal hair example from The Last of Us. (Imgur, n.d.)

On the image above 2D planes (Figure 12, right), a combination of few vertices, were manually placed on the character’s head so that each hair strand could be coloured afterwards (Figure 12, left)

This technique is used in video games due to its very efficient results. The full process is explained on the YouTube channel CG Cookie - Unity Training (CG Cookie – Unity Training, n.d.).

2.2.4 Geometry

One of the common usages of dynamic geometry creation in real-time is for grass generation in games. Although grass does not have the same physical properties as hair or fur, it probably needs to be at least as optimal since it often covers considerably more space in games than fur (White, 2008).



VERTEX SHADER

High LOD

15 vertices

Low LOD

7 vertices

Figure 13 - Vertex shader with adaptive mesh according to LOD. (GDCVault, n.d.)

Procedural Grass in *'Ghost of Tsushima'* is a conference where the developers cleverly generated grass with relatively simple geometry. The grass adapts its geometry according to the distance of the camera. On the left side of the image above (Figure 13), the mesh is subdivided into 15 vertices for more detail when the camera is close. On the right side, the mesh is rendered with fewer vertices since the distance of the camera does not require as many details to seem visually accurate (GDC, 2022).

2.3 Shell Texturing in the gaming industry

2.3.1 Shadow of Colossus

The first example of the use of a shell texturing system in a AAA game was *Shadow of Colossus*, “a 2005 action-adventure game developed by Japan Studio and Team Ico, and published by Sony Computer Entertainment for the PlayStation 2.” (“Shadow of the Colossus”, 2024.-a).

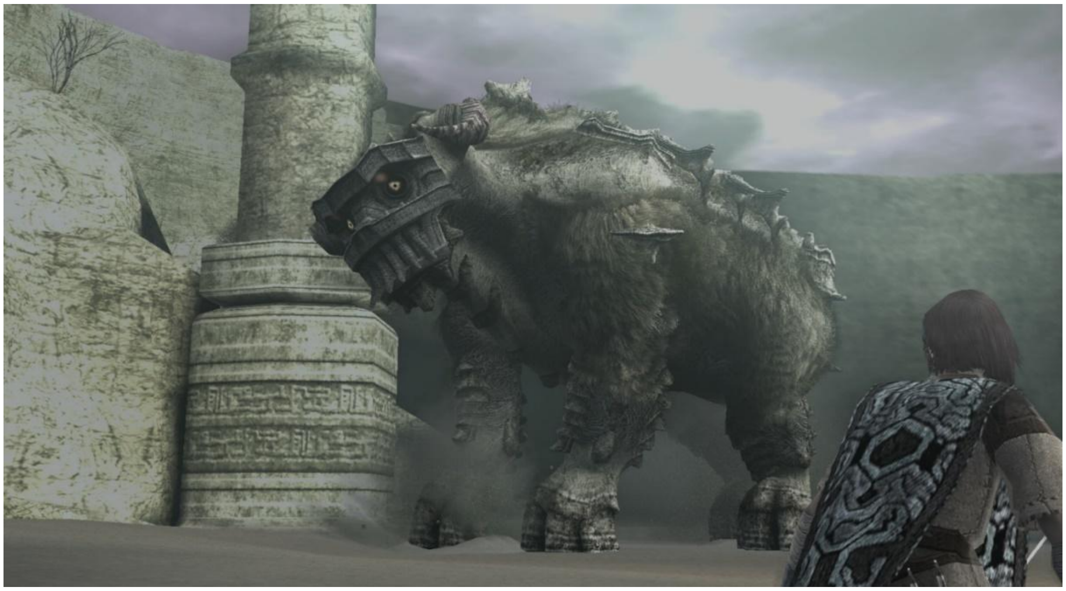


Figure 14 - Shadow of Colossus in game caption of the player looking at a Colossus. (Froyok, n.d.)

In *Shadow of Colossus*, the Shell-Texturing was used for some of the colossuses’ fur. Léna Piquet , known under the pseudonym Froyok, did a breakdown that covers aspects going from the number of layers to the direction of the fur according to the mesh. Piquet’s blog also gives a translation of an interview, originally in Japanese, with the team that created the game. (Froyok, n.d.)

2.3.2 Viva Piñata

“*Viva Piñata* was released in 2006 and created by Xbox Game Studios and Rare” (“Viva Piñata”, 2023).

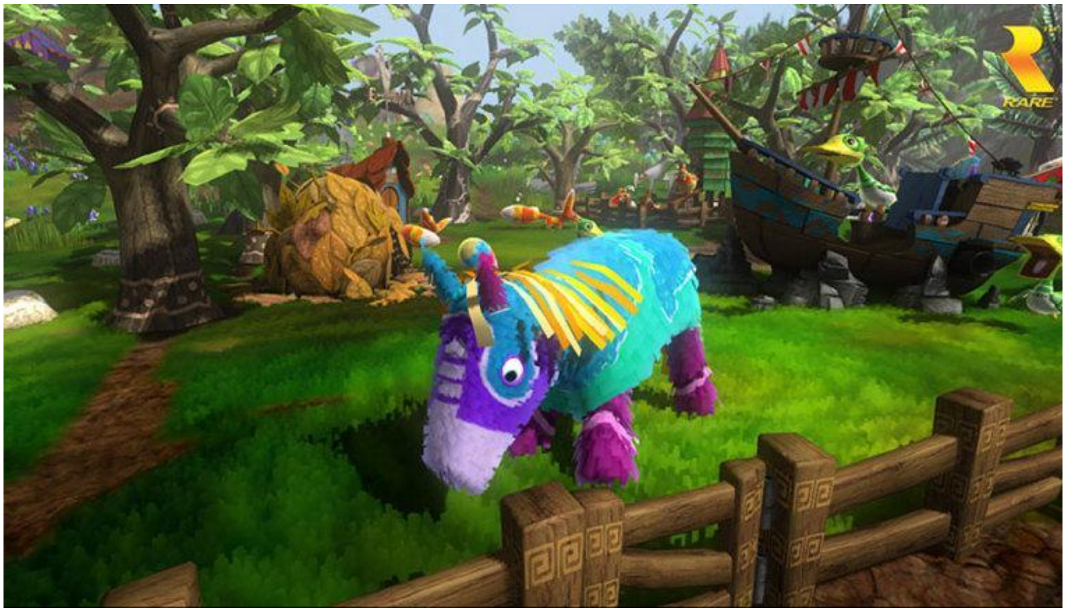


Figure 15 - Viva Piñata in Game Caption. (NBC News, 2006)

In *Viva Piñata*, shell texturing was used to simulate the paper-like fur and the environment’s grass. Unfortunately, no information about the engine and technology used for its creation was found.

2.3.3 Dark Souls 3

Dark Souls III, produced by FromSoftware and published by Bandai Namco Entertainment in 2016 (“Dark Souls III”, 2020), is a reference to demonstrate that shell texturing is used in a game with realistic graphics.

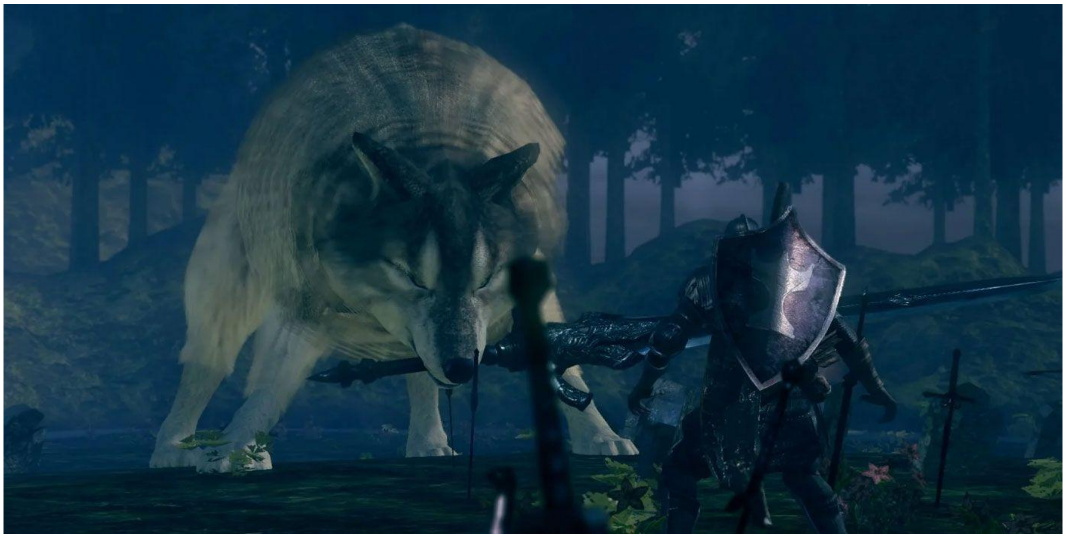


Figure 16 - Dark Souls III Wolf boss (Sif). (Creswell, 2021)

According to a souls modding community forum, the engine used by FromSoftware was called Dantelion and it is also proprietary. The only information gathered on its development is that it relied on third party software such as the Havok engine for physics, Fmod for audio. Any information directly related to the fur rendering does not seem available (Souls modding, n.d.).

2.3.4 Genshin Impact

The most relevant game to point out for this project is undoubtedly *Genshin Impact* since it is a more recent example of a game where Shell texturing is used ("Genshin Impact", 2024.-d).

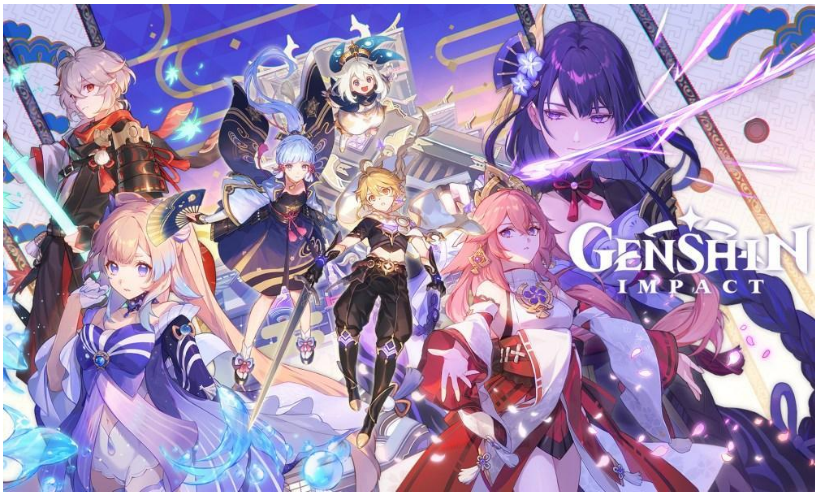


Figure 17 - Genshin impact illustration. (For The Win, 2022)

This game was created by miHoYo and published in 2020. This reference is an example of the use of shell texturing to render grass and fur in a stylised graphical environment.

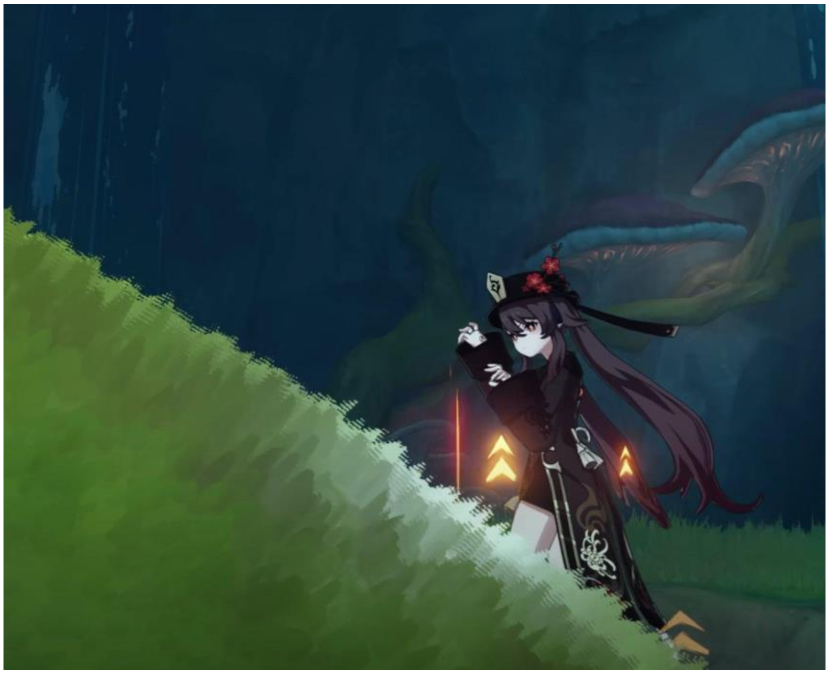


Figure 18 - In-game caption of Genshin Impact. (X (formerly Twitter), 2024)

The use of Unity as the game engine in *Genshin Impact* ([ANSWERED], n.d.) presents a significant advantage for this Bachelor’s project. Serving as a valuable point of reference, the game offers a tangible benchmark for comparison and enhancement throughout our project's development. This not only facilitates iterative improvements but also sets the minimum thresholds and requirements aimed to be surpassed.

2.4 Unity Shell Texturing Project

2.4.1 Objectives and limitations

The objective of this project is to focus primarily on the understanding of what is shell texturing and how it is made in a real-time environment. The understanding of how to code in HLSL will be necessary and since Unity has its own methods, an adaptation to that understanding will have to be made to be able to code inside the Unity editor.

The final goal of the project is to determine how to resolve the shell rendering problems when fur strands are perpendicular to the direction the camera is facing and to understand in what configurations this resolution is not feasible.

As a Bachelor’s degree student, it is essential to acknowledge the limitations related to an academic setting. Primarily, the time available to develop the project is restricted by an academic calendar, allowing limited time for its creation. Furthermore, despite rigorous and dedicated study, the level of expertise necessary for such a project would probably require years of specialised experience and exposure to industry-standard practices and technologies.

The project's ambition is calibrated to reflect a learning exercise and a demonstration of potential rather than a product ready for professional application. This perspective ensures that the project remains a demonstrative tool, designed to expose understanding and skills, rather than meeting commercial standards. It is important to acknowledge that many researchers and professional graphics programmers have already treated this subject and are still working towards its improvement.

2.4.2 Unity projects

The Unity projects will focus mainly on the replication of the shell texturing method seen in *Genshin impact* and if possible, its improvement. It will be decomposed in six steps, each adding a layer of improvement either related to the technical or visual outcome.

2.4.2.1 Initial approach

The first approach will be a naïve attempt to produce, without prior knowledge, a technique combining simple shaders and C# scripts to produce the shell texturing. The objective of this first approach is to understand how to replicate shell texturing ideally without having to focus on the intricacies of HLSL. Any specific algorithms related to graphical fidelity such as light simulation, reflection, and shadowing will be ignored at this point since the focus is on understanding the theory and not producing a high-quality project.

2.4.2.2 HLSL attempt

Once that first step is accomplished, the project will be improved by exclusively using shader language to generate the necessary shells for producing fur. The use of a geometry shader is intended to remove these shells. Eliminating the C# scripts will align the project more closely with professional graphics programming practices. During the process, the objective will be to improve the code and test its limitations to have the most visually pleasing outcome without neglecting its optimality. At this point, the project will ideally contain some graphically accurate lighting and shading models.

2.4.2.3 Shell and Fin

Once the shell-texturing shader is done, the implementation of a fin shader will produce extracted vertices over the entire surface of any object on which textures of fur strands will be projected. Since both shaders are exactly opposite (Lengyel et al., 2001), the idea is to create a merged version to either render shells or fins according to circumstances such as the distance and angle of the camera. This adaptation will allow it to seamlessly transition from shell to fin and avoid edge cases where the viewer obviously sees flaws.

2.4.2.4 Scene variations

With the different fur generating shaders created, tests will be done throughout multiple scenes of defined complexities to be able to obtain data such as frame rate and memory usage. These scenes will contain objects with shapes of different levels of complexity to evaluate specific edge cases and varying size environments to push the boundaries of the hardware used and obtain measurements that will later be compared to those extracted from different devices.

2.4.2.5 Hardware implementations

Once that comparison is done on a personal computer, the project will be ported on hardware setups such as the Nintendo Switch and Android mobiles so that an analysis of frame rate and visual outcome can be made. The comparison made on different hardware will first demonstrate the possibility of using the created shaders on lower capacity machines, but it will also deliver insight on possible improvements or limitations.

2.4.2.6 Other Improvements and edge cases

Lastly, improvements will be attempted regarding lighting, shadowing and physical behaviour by implementing algorithms found in different papers.

2.4.3 Anticipated protocol

Up to this point, the shell texturing, fin, geometry and polygon techniques have been used and developed over time to render hair, fur or grass. These five methods have the same common point of interest being rendered in real-time.

Regarding the test protocol, it is expected to have at least three scenes that will mimic *Genshin Impact*’s shell texturing technique. These three scenes will be declined with each developed method, the naïve shell rendering, the HLSL geometry technique, and the shell and fin technique for a total of nine scenes.

Each scene will contain objects with different levels of complexity, the first scene will be rendering a simple plane with four vertices, the second will contain the same plane and multiple basic Unity spheres and the last will contain multiple complex objects.

The extraction of data such as frame rate and memory used will be done for each scene. To extract this data multiple tools will be used. The first tools used will be Unity's set of tools such as the Statistics window, the Frame debugger, the Profiler, and the Render Debugger. Secondly, the use of RenderDoc or Nvidia Nsight Graphics will be used to analyse frames.

2.5 Conclusion

Throughout this state of the art, the definition of the important composites of the problematic was laid out, exposing the most common ways to render fur and hair, reviewing four games that have been using shell texturing for several years and finally exposing the objectives and components of the anticipated project.

3. Qualitative Analysis

3.1 Interviews

With the projects and anticipated protocols defined, it was important to reach out to professionals and experts in the graphics programming field to see what their thoughts about the approach were, or if they had any feedback or tips about the methodology. The questions varied according to each specialist and all of them can be found in the appendixes (See Appendix A &B). Unfortunately, only one specialist answered the given questions.

The most important information gathered was first the mention of the *Real time rendering 4th edition* that seemed to be relevant to the project and secondly the advice regarding the project itself. Resuming the advice received can be summarised in two key points: knowing and keeping in mind the platform on which the project is being developed and the purpose for which the shaders are being created (Sena, e-mail, 14 May 2024) (see Appendix A).

Due to a lack of answers from professionals and given the wide use of AI nowadays the interview was attempted using ChatGPT and the answers gave interesting insight about the process of fur creation. One of the common pieces of advice from ChatGPT and David Sena was the importance of context. The optimisation strategies and techniques will differ according to the use and platform for which the project is developed. The second converging element is the way the process should be approached (ChatGPT, Interview, 2024) (see Appendix B).The mention of Unreal Engine and any other suggested tools was discarded since the project aims at hand coded reproduction and improvement of fur shaders in a stylised fashion.

3.2 Project Analysis

Two projects have been selected for analysis mainly because they were created with Unity but also because they both have shell texturing and use the URP rendering pipeline.

The analysis criteria are first and foremost the visual quality and time to render a frame, then data related to the memory usage, the number of vertices and lastly, if possible, how optimal they are on lower capacity machines such as Nintendo Switch or mobile phones.

3.2.1 Genshin Impact

Genshin Impact is the first project selected for analysis, first because it brought up the question of the utility of shell texturing given the flaws of that method but also because it was exported to lower capacity machines like mobile phones as seen in the minimal hardware requirements given by Danielson and Yonezawa (2024) on the website ScreenRant.

Originally the intention was to use software such as RenderDoc of Nvidia Nsight graphics to capture a frame from the game and be able to compare it to the project. Unfortunately, due to an anti-cheat system (Hoyoverse.com, 2024) it was impossible to use such a tool on this game. One possibility would have been to use a cracked version of the game or use an anti-hack bypass such as the EasyPeasy-Bypass given by the github user gmh5225 (Gmh, n.d.). However, none of these seemed to be an ethical choice. For that reason and after multiple trials using RenderDoc and Nvidia Nsight Graphics, the decision to stop trying and do a purely visual analysis of it had to be made.

Regarding the visual quality, the method used to render grass is shell texturing, recognizable due to the discontinuity of layers (Figure 19, Yellow rectangle).

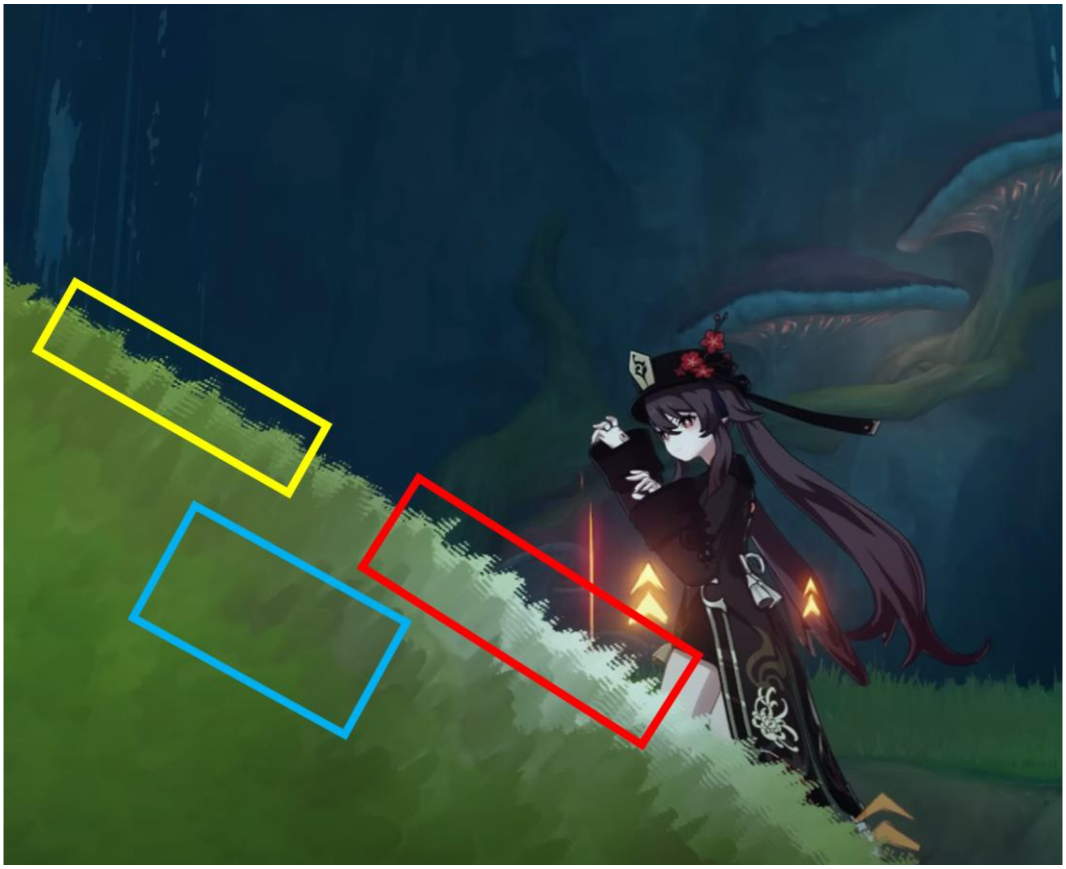


Figure 19 - Genshin impact caption with coloured rectangle highlighting specific elements. (X (formerly Twitter), 2024)

The colour seems to be randomly distributed between two values of green (Figure 19, Blue rectangle). The most interesting fact about Genshin’s Impact grass shader is the reaction to light sources (Figure 19, Red rectangle).

3.2.2. Hecomi – UnityFurURP

Thanks to Hecomi a Unity Fur project was developed with the URP pipeline (Hecomi, 2024). In this project, available on github, multiple shaders are shown exposing three methods of fur rendering: shell, fin, and geometry (Figure 20) (Hecomi, n.d.).

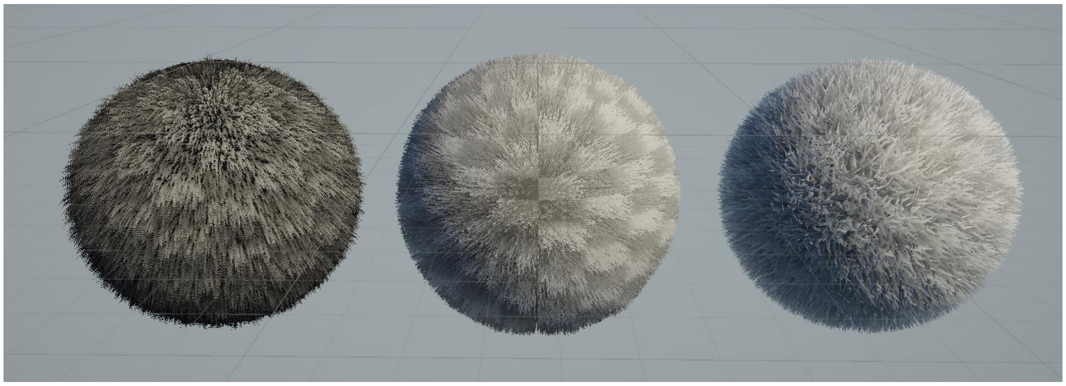


Figure 20 - Caption in Unity editor of the cloned github project from Hecomi showing shell, fin, and geometry shader from left to right. (Styles, 2024)

Since the project is based primarily on shell texturing, it is useful to compare it with a version developed by an experienced graphics programmer (Hecomi, 2024).

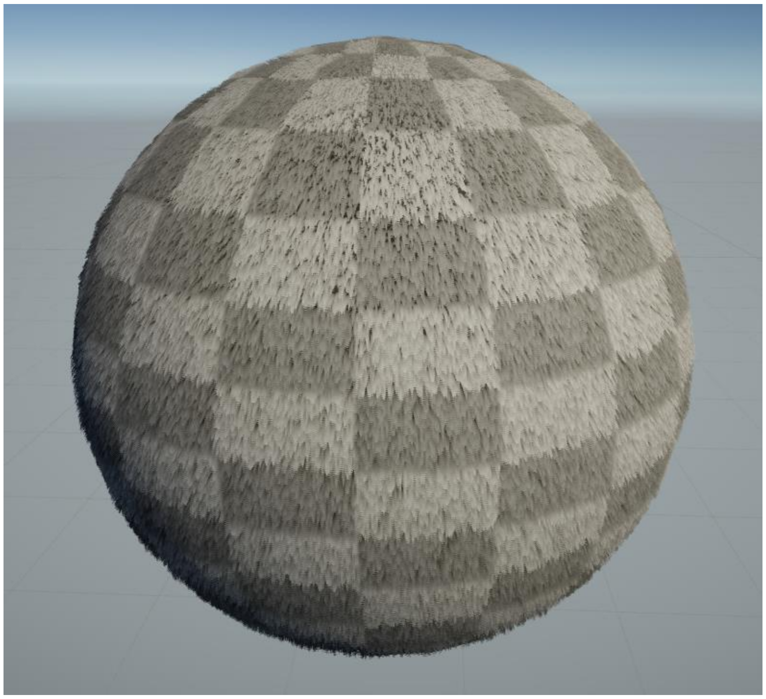
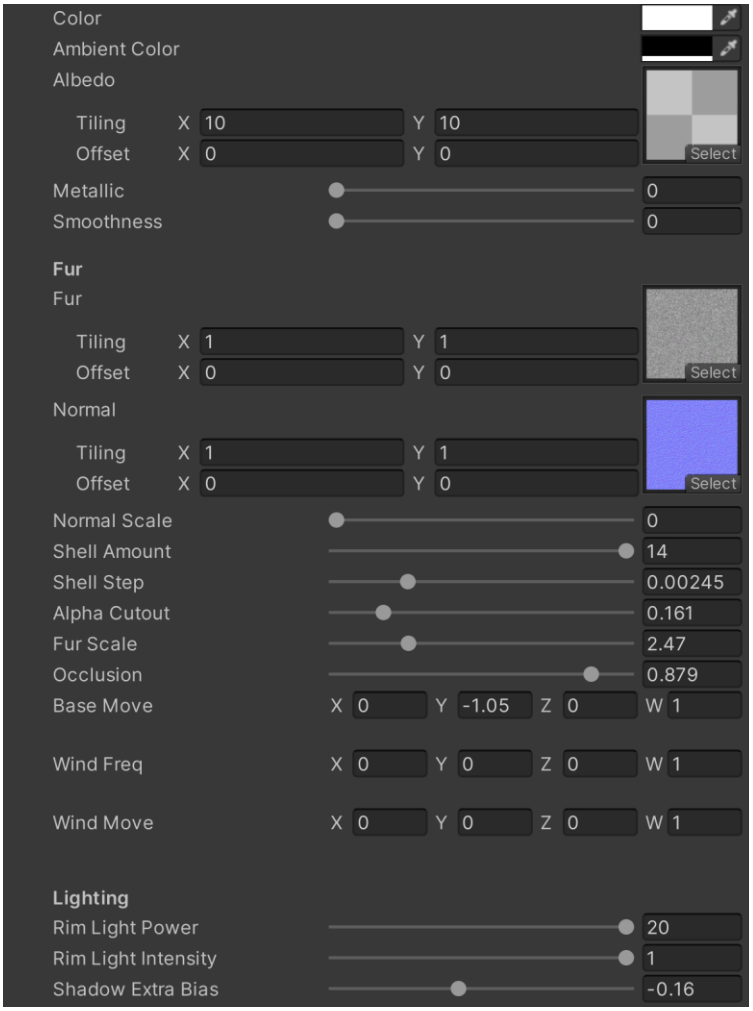


Figure 21 - Caption in Unity editor of the cloned github project from Hecomi showing shell. (Styles, 2024)

The shell scene contains a sphere with 515 vertices, a directional light, and a post processing volume adding ACES tonemapping, bloom and vignetting. The sphere with shell textured fur has the same problems identified in *Genshin Impact* (Figure 21).

The shader itself enables the material to have multiple exposed parameters like colour, number and length of shells (Figure 22).



Color
Ambient Color
Albedo
Tiling X 10 Y 10
Offset X 0 Y 0
Metallic 0
Smoothness 0

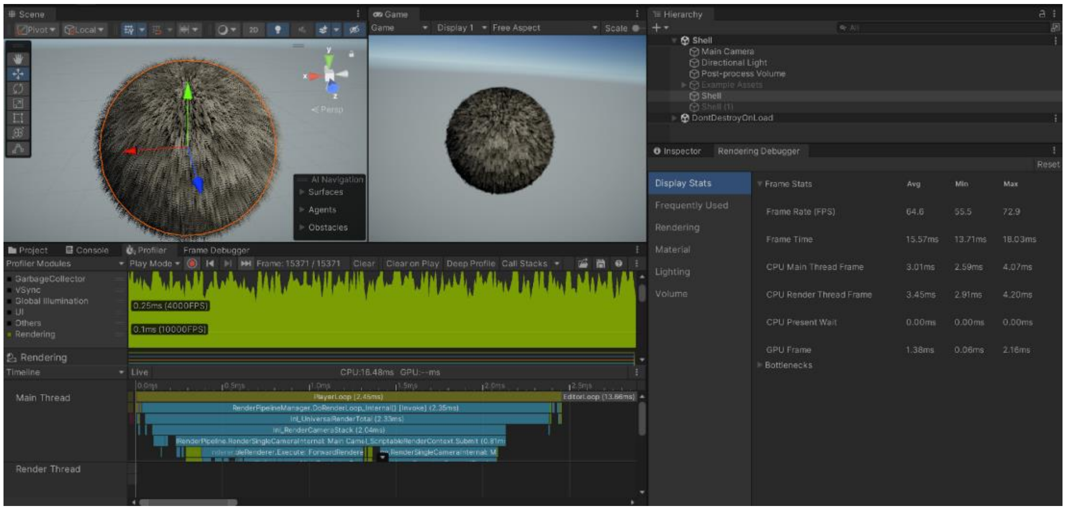
Fur
Fur
Tiling X 1 Y 1
Offset X 0 Y 0
Normal
Tiling X 1 Y 1
Offset X 0 Y 0
Normal Scale 0
Shell Amount 14
Shell Step 0.00245
Alpha Cutout 0.161
Fur Scale 2.47
Occlusion 0.879
Base Move X 0 Y -1.05 Z 0 W 1
Wind Freq X 0 Y 0 Z 0 W 1
Wind Move X 0 Y 0 Z 0 W 1

Lighting
Rim Light Power 20
Rim Light Intensity 1
Shadow Extra Bias -0.16

Figure 22 - Caption in Unity editor of the cloned github project from Hecomi showing shell material parameters . (Styles, 2024)

The parameters exposed give the user the possibility of changing the way the fur is rendered including specific parameters like wind and rim lighting (Figure 22).

Regarding the code itself, a geometry shader was used to extrude layers but the code being sliced in 6 HLSL files makes it hard to understand for a beginner in graphics programming. The Unity Fur project will still be useful as a reference. On the statistical side, data related to the rendering process can be extracted thanks to Unity's profiler tool.



Frame Stats	Avg	Min	Max
Frame Rate (FPS)	64.6	55.5	72.9
Frame Time	15.57ms	13.71ms	18.03ms
CPU Main Thread Frame	3.01ms	2.59ms	4.07ms
CPU Render Thread Frame	3.45ms	2.91ms	4.20ms
CPU Present Wait	0.00ms	0.00ms	0.00ms
GPU Frame	1.38ms	0.06ms	2.16ms

Figure 23 - Caption in Unity editor of the project from Hecomi showing measures taken. (Styles, 2024)

The entire render pipeline takes 2,35ms per frame, varying between 0,06 and 2,16ms for a GPU frame containing a total of 7160 vertices with 1.33 GB of memory used. The GPU frame takes on average 1.38ms (Figure 23). These values, taken from Hecomi's shell scene, will be very useful for our project when it comes to analysing efficiency and visual appeal.

3.3 Conclusion

Throughout this qualitative analysis, information from a senior graphics engineer and an AI tool was obtained. The analysis of two projects that brought a point of comparison in terms of efficiency and visual appeal was done.

4. Test Protocol

4.1 Definition of the used metrics

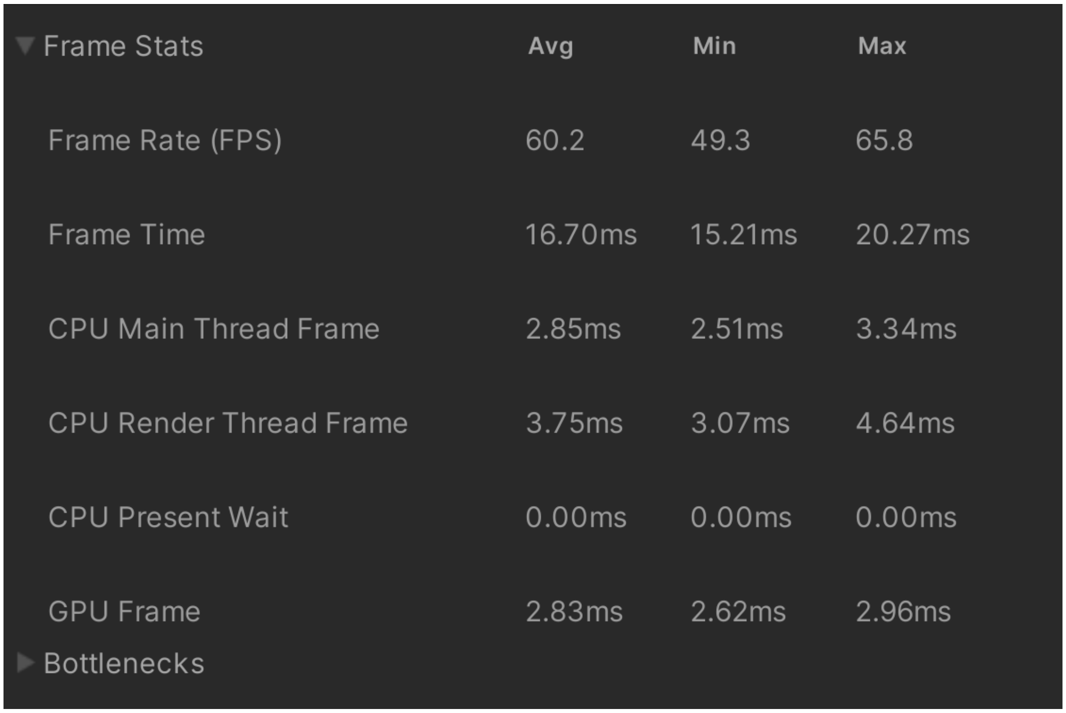
4.1.1 Frame time

The frame time is an essential metric used to define if our project is efficient or not. According to the interview done with David Sena:

“At the end of the day, performance constraints from the hardware platform that you're targeting are what truly matters for a product. An amazing technique that takes too long to execute is not useful because it can't be used.” (Sena, e-mail, 14 May 2024) (see Appendix A)

It is understandable that, related to his experience, visual quality is important but not as much as fluidity of frame rate.

In this project the measure will not be done in frames per second but in seconds per frame or how much time it takes for our renderer to finish its work and display the result of the shader for one frame. The time measurement will be done globally and specifically, respectively accounting for all passes and only the pass rendering the shells itself.



▼ Frame Stats	Avg	Min	Max
Frame Rate (FPS)	60.2	49.3	65.8
Frame Time	16.70ms	15.21ms	20.27ms
CPU Main Thread Frame	2.85ms	2.51ms	3.34ms
CPU Render Thread Frame	3.75ms	3.07ms	4.64ms
CPU Present Wait	0.00ms	0.00ms	0.00ms
GPU Frame	2.83ms	2.62ms	2.96ms
▶ Bottlenecks			

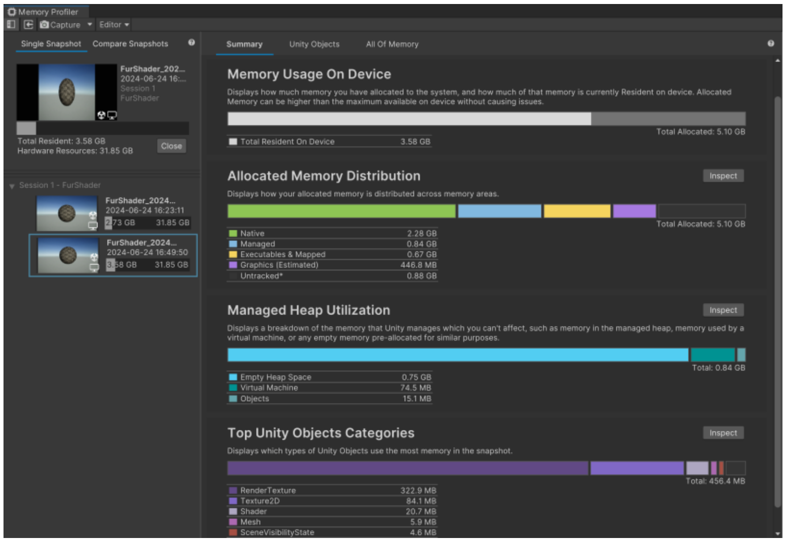
Figure 24 - Capture of the Render Debugger from Unity. (Styles, 2024)

Regarding the requirements, the human perception threshold is stated to be at a minimum of 24 frames per second and a maximum 60 although this maximum limit is currently still being debated. This means that a frame must take up to 0.041666 seconds or 41.66666ms maximum (“ Flicker fusion threshold”, 2024.-b). The objective is to not go over this time per entire frame and stay under the 2,16ms per GPU frame obtained from Hecomi’s project analysis in section 3.2.2.

4.1.2 Memory usage

The memory usage is the second point to measure. Since the improvement of technology described by Moore’s law (“Moore’s law”, 2024.-c) does not apply to memory, it is even more important to be able to measure it and limit its usage.

Thanks to Unity it is possible to obtain the values for the Managed heap “The used heap size and total heap size that managed code uses” and the Graphics “The estimated amount of memory the driver uses on Textures, render targets, Shaders, and Mesh data.” Normally one would try to obtain a VRAM usage value, but Unity has not yet been able to determine that value since it is platform specific thus rendering the measurement very complex ("How to use Unity's memory profiling tools", n.d.).



FurShader_202...
2024-06-24 16:...
Session 1
FurShader

Total Resident: 3.58 GB
Hardware Resources: 31.85 GB

Session 1 - FurShader

FurShader_2024...
2024-06-24 16:23:11
2.73 GB 31.85 GB

FurShader_2024...
2024-06-24 16:49:50
3.58 GB 31.85 GB

Memory Usage On Device

Displays how much memory you have allocated to the system, and how much of that memory is currently Resident on device. Allocated Memory can be higher than the maximum available on device without causing issues.

Total Allocated: 5.10 GB

Total Resident On Device	3.58 GB

Allocated Memory Distribution

Displays how your allocated memory is distributed across memory areas.

Total Allocated: 5.10 GB

Native	2.28 GB
Managed	0.84 GB
Executables & Mapped	0.67 GB
Graphics (Estimated)	446.8 MB
Untracked*	0.88 GB

Managed Heap Utilization

Displays a breakdown of the memory that Unity manages which you can't affect, such as memory in the managed heap, memory used by a virtual machine, or any empty memory pre-allocated for similar purposes.

Total: 0.84 GB

Empty Heap Space	0.75 GB
Virtual Machine	74.5 MB
Objects	15.1 MB

Top Unity Objects Categories

Displays which types of Unity Objects use the most memory in the snapshot.

Total: 456.4 MB

RenderTexture	322.9 MB
Texture2D	84.1 MB
Shader	20.7 MB
Mesh	5.9 MB
SceneVisibilityState	4.6 MB

Figure 25 - Unity editor caption of the Memory Profiler tool indicating the difference memory allocations. (Styles, 2024)

With that in mind the principal value indicating the performance of the shaders is the graphics (estimated) allocated memory.

The expectation is to stay in the same order of magnitude as the measures on Hecomi's shell shader, meaning 446.8 MB for the graphics estimated total. Any excess of memory usage will indicate a problem in the procedure and shader itself.

4.1.3 Vertices

Since our project intakes a certain number of vertices to increase them and render the fur on multiple extruded layers, one of the measurements of interest is the number of vertices resulting from the shader's action.

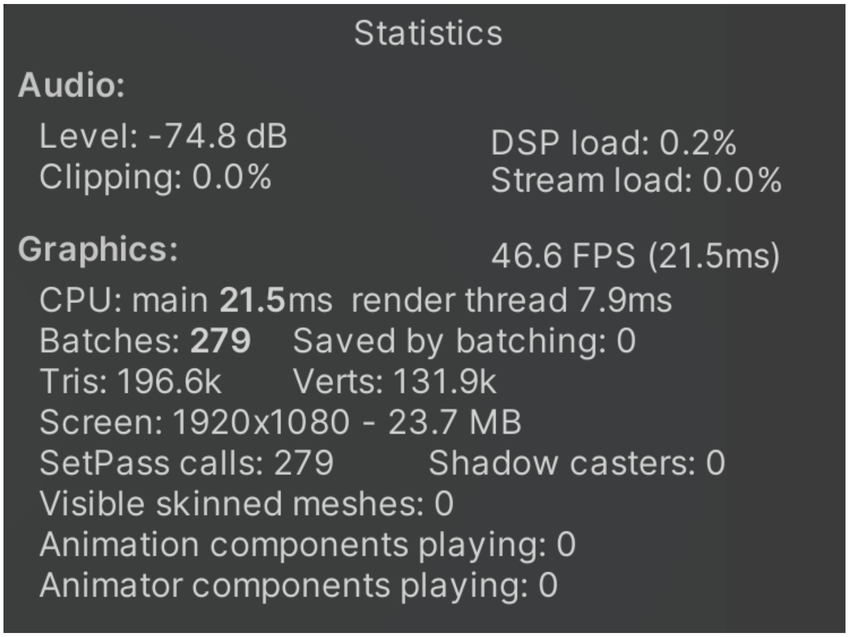
The values expected must be sensibly close to those obtained on Hecomi's shader, in this case 14 times the original amount (7160 output /515 input = 13.90291). Due to a lack of experience, values exceeding that amount will be tolerated if they do not surpass twice of that amount.

4.2 Measurement Tools

Unity, being a very complete game engine, will give us access to an entire set of tools that will be specifically useful for measuring the values previously exposed. This chapter will expose all the tools used throughout the project.

4.2.1 Unity Statistics:

The Unity statistics tool is a window exposing the most common values used for performance optimisation and analysis.



```
Statistics
Audio:
  Level: -74.8 dB                DSP load: 0.2%
  Clipping: 0.0%                 Stream load: 0.0%

Graphics:                        46.6 FPS (21.5ms)
  CPU: main 21.5ms  render thread 7.9ms
  Batches: 279    Saved by batching: 0
  Tris: 196.6k    Verts: 131.9k
  Screen: 1920x1080 - 23.7 MB
  SetPass calls: 279        Shadow casters: 0
  Visible skinned meshes: 0
  Animation components playing: 0
  Animator components playing: 0
```

Figure 26 - Unity editor Statistics tool. (Styles, 2024)

The values of interest are mainly:

- Render thread: time taken by the rendering thread to do its work for a frame

- Tris: the number of triangles rendered

- Verts: the number of vertices rendered

4.2.2 Unity Profiler & Timeline

The Unity profiler is a very complete tool used to analyse a lot of elements such as CPU usage, rendering, memory, audio, video, and many others. It also gives a timeline in which the different the different threads can be analysed in detail.

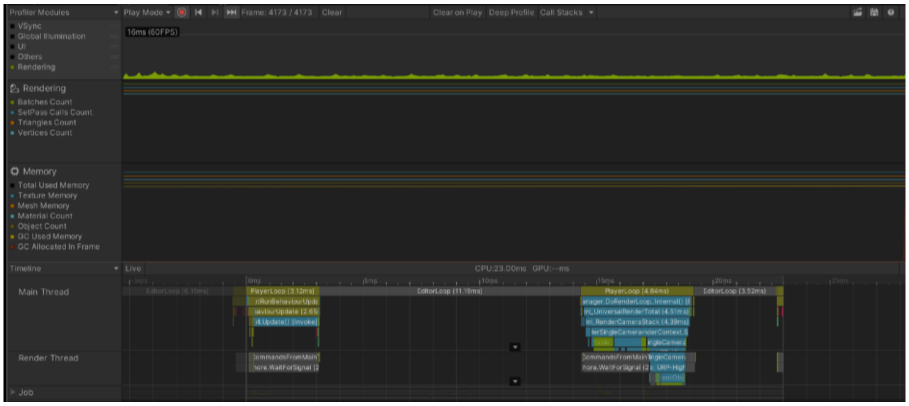
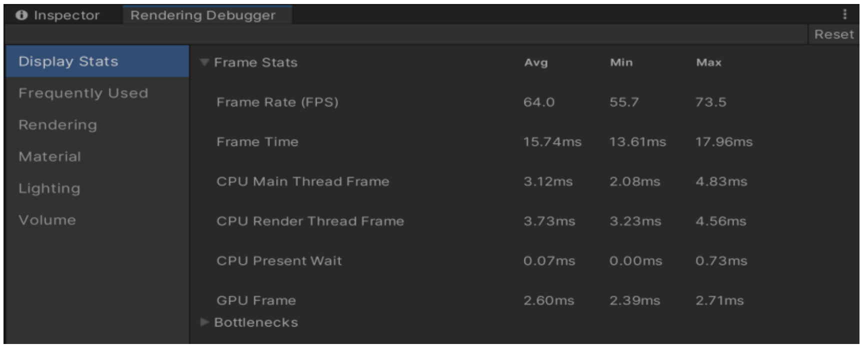


Figure 27 - Unity Profiler (top) and Timeline (bottom). (Styles, 2024)

In this case the interest is focused on the rendering and memory sections of that tool.

4.2.3 Rendering Debugger

The rendering debugger is a tool used to modify what is seen in Unity and apply masks and be able analyse visually specific points like triangles and vertices in the case of the wireframe mesh visualisation.



Frame Stats	Avg	Min	Max
Frame Rate (FPS)	64.0	55.7	73.5
Frame Time	15.74ms	13.61ms	17.96ms
CPU Main Thread Frame	3.12ms	2.08ms	4.83ms
CPU Render Thread Frame	3.73ms	3.23ms	4.56ms
CPU Present Wait	0.07ms	0.00ms	0.73ms
GPU Frame	2.60ms	2.39ms	2.71ms

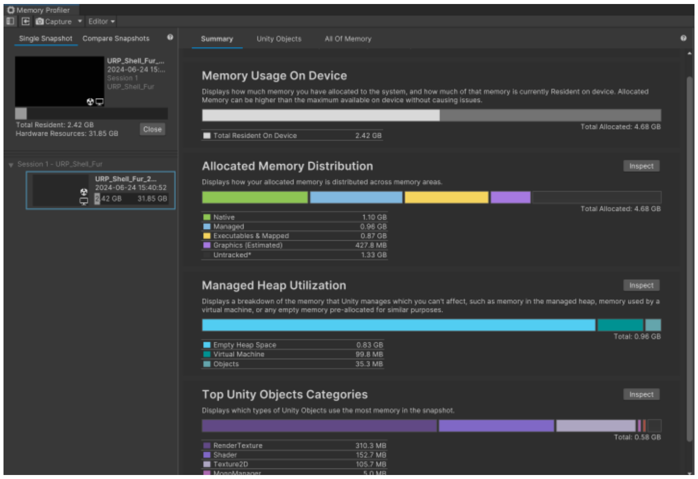
Figure 28 - Unity Rendering Debugger tool showing the Display Stats category. (Styles, 2024)

The values to focus on are in the “Display Stats” category where the different frame rates can be found.

To be able to have a good estimation of the GPU frame time, the average will be calculated from 1000 measurements obtained directly from the Unity projects. The values will be captured at run-time in the editor by storing them in an array of doubles thanks to a C# script (See Appendix P). The data collected in a .txt file to be reported to an excel spreadsheet to be averaged.

4.2.4 Memory Profiler

Unity’s memory profiler is a complete tool that is useful when it comes to measuring memory use (“How to use Unity’s memory profiling tools”, n.d.).



Total Resident: 2.42 GB
Hardware Resources: 31.85 GB

Memory Usage On Device

Displays how much memory you have allocated to the system, and how much of that memory is currently Resident on device. Allocated Memory can be higher than the maximum available on device without causing issues.

Total Allocated: 4.68 GB

Total Resident On Device	2.42 GB

Allocated Memory Distribution

Displays how your allocated memory is distributed across memory areas.

Total Allocated: 4.68 GB

Native	1.10 GB
Managed	0.96 GB
Executables & Mapped	0.87 GB
Graphics (Estimated)	427.8 MB
Untracked*	1.33 GB

Managed Heap Utilization

Displays a breakdown of the memory that Unity manages which you can't affect, such as memory in the managed heap, memory used by a virtual machine, or any empty memory pre-allocated for similar purposes.

Total: 0.96 GB

Empty Heap Space	0.83 GB
Virtual Machine	99.8 MB
Objects	35.3 MB

Top Unity Objects Categories

Displays which types of Unity Objects use the most memory in the snapshot.

Total: 0.58 GB

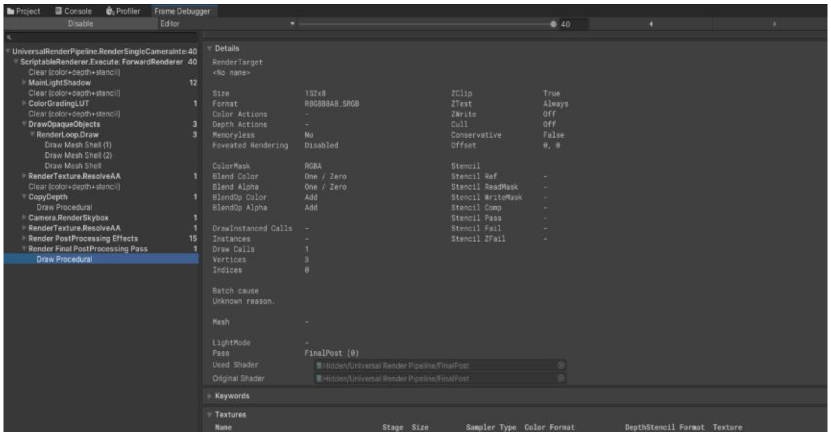
RenderTexture	310.3 MB
Shader	152.7 MB
Texture2D	105.7 MB
MonoManager	5.0 MB

Figure 29 - Unity editor caption of the Memory Profiler tool. (Styles, 2024)

A simple version of the profiler is already integrated in the Unity profiler 4.2.2 but in this case the use of the detailed version will give more information about the memory usage, especially regarding the graphics memory allocation.

4.2.5 Unity Frame debugger

Unity’s frame debugger is a useful tool when it comes to analysing specific frames. It exposes all the different passes or steps the rendering goes through to display the final image and gives all the data, values, and textures accounted for in the process.



```
UniversalRenderPipeline.RenderSingleCameraInte 40
  ScriptableRenderer.Execute: ForwardRenderer  40
    Clear (color+depth+stencil)
    MainLightShadow                            12
    Clear (color+depth+stencil)
    ColorGradingLUT                             1
    Clear (color+depth+stencil)
    DrawOpaqueObjects                           3
      RenderLoop.Draw                           3
        Draw Mesh Shell (1)
        Draw Mesh Shell (2)
        Draw Mesh Shell
    RenderTexture.ResolveAA                     1
    Clear (color+depth+stencil)
    CopyDepth                                   1
      Draw Procedural
    Camera.RenderSkybox                         1
    RenderTexture.ResolveAA                     1
    Render PostProcessing Effects              15
    Render Final PostProcessing Pass            1
      Draw Procedural
```

```
RenderTarget
<No name>

Size                 152x8                 ZClip              True
Format               R8G8B8A8_SRGB         ZTest              Always
Color Actions        -                     ZWrite             Off
Depth Actions        -                     Cull               Off
Memoryless           No                    Conservative       False
Foveated Rendering   Disabled              Offset             0, 0

ColorMask            RGBA                  Stencil
Blend Color          One / Zero            Stencil Ref        -
Blend Alpha          One / Zero            Stencil ReadMask   -
BlendOp Color        Add                   Stencil WriteMask  -
BlendOp Alpha        Add                   Stencil Comp       -
                                           Stencil Pass       -
DrawInstanced Calls  -                     Stencil Fail       -
Instances            -                     Stencil ZFail      -
Draw Calls           1
Vertices             3
Indices              0

Batch cause
Unknown reason.

Mesh                 -

LightMode            -
Pass                 FinalPost (0)
Used Shader          Hidden/Universal Render Pipeline/FinalPost
Original Shader      Hidden/Universal Render Pipeline/FinalPost
```

```
Name                 Stage  Size   Sampler Type  Color Format   DepthStencil Format  Texture
```

Figure 30 - Unity editor screenshot of the Frame Debugger tool. (Styles, 2024)

4.2.6 RenderDoc

RenderDoc is a tool that sensibly resembles Unity’s frame debugger as it also exposes all the passes. The main difference is that, since it is a software dedicated to frame analysis, it has more in-depth information for each step.

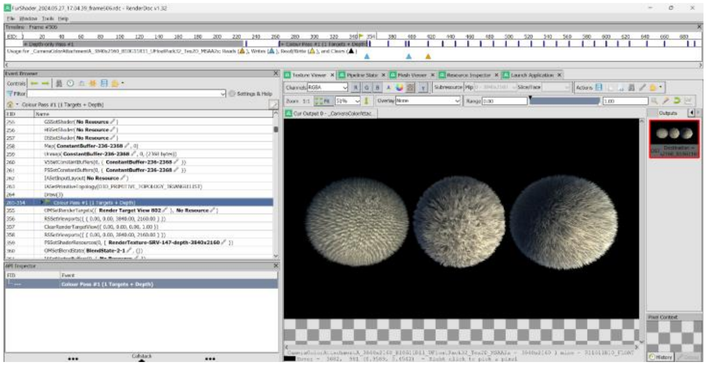
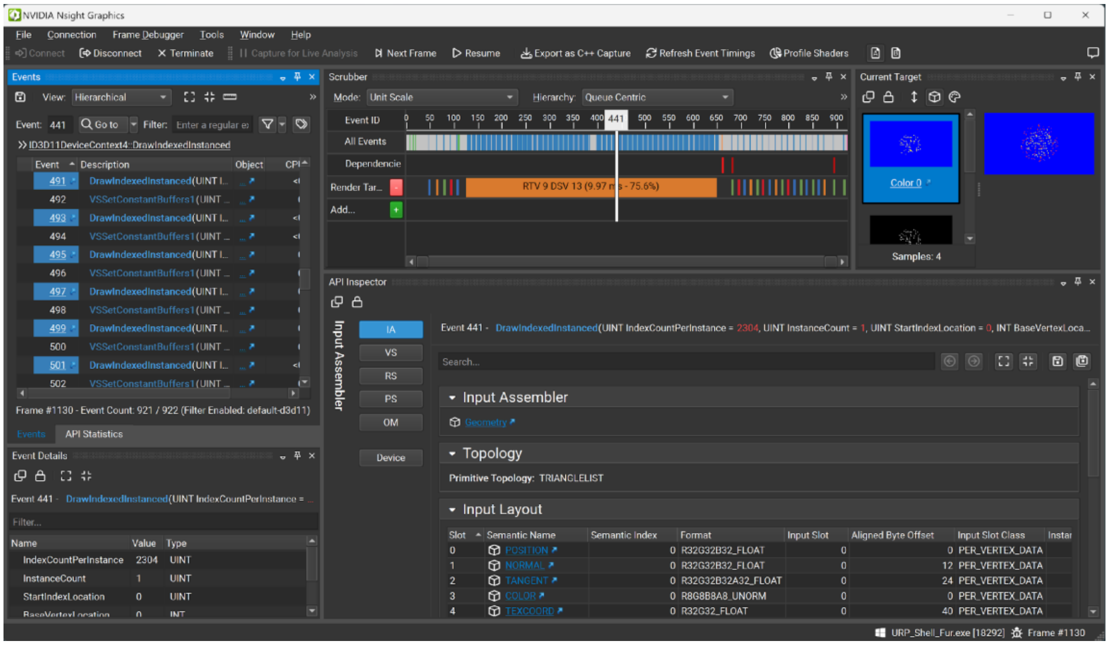


Figure 31 - RenderDoc used on Hecomi's project. (Styles, 2024)

This project might not need it but in case specific information cannot be found with Unity it is still a useful resource to consider.

4.2.7 Nvidia Nsight Graphics

Lastly, like Unity's frame debugger and RenderDoc, Nsight Graphics is a tool used to analyse frames and output precise information. Again, the project will probably not need to use it except on lower capacity machines where the former tools might not work.



Event: 441

ID3D11DeviceContext4::DrawIndexedInstanced

Event	Description
491	DrawIndexedInstanced(UINT I...
492	VSSetConstantBuffers1(UINT ...
493	DrawIndexedInstanced(UINT I...
494	VSSetConstantBuffers1(UINT ...
495	DrawIndexedInstanced(UINT I...
496	VSSetConstantBuffers1(UINT ...
497	DrawIndexedInstanced(UINT I...
498	VSSetConstantBuffers1(UINT ...
499	DrawIndexedInstanced(UINT I...
500	VSSetConstantBuffers1(UINT ...
501	DrawIndexedInstanced(UINT I...
502	VSSetConstantBuffers1(UINT ...

Frame #1130 - Event Count: 921 / 922 (Filter Enabled: default-d3d11)

Event 441 - DrawIndexedInstanced(UINT IndexCountPerInstance =

Name	Value	Type
IndexCountPerInstance	2304	UINT
InstanceCount	1	UINT
StartIndexLocation	0	UINT
BaseVertexLocation	0	INT

Event 441 - DrawIndexedInstanced(UINT IndexCountPerInstance = 2304, UINT InstanceCount = 1, UINT StartIndexLocation = 0, INT BaseVertexLoca...

Input Assembler

Geometry

Topology

Primitive Topology: TRIANGLELIST

Input Layout

Slot	Semantic Name	Semantic Index	Format	Input Slot	Aligned Byte Offset	Input Slot Class
0	POSITION	0	R32G32B32_FLOAT	0	0	PER_VERTEX_DATA
1	NORMAL	0	R32G32B32_FLOAT	0	12	PER_VERTEX_DATA
2	TANGENT	0	R32G32B32A32_FLOAT	0	24	PER_VERTEX_DATA
3	COLOR	0	R8G8B8A8_UNORM	0	0	PER_VERTEX_DATA
4	TEXCOORD	0	R32G32_FLOAT	0	40	PER_VERTEX_DATA

URP_Shell_Fur.exe [18292] Frame #1130

Figure 32 - NVIDIA Nsight Graphics used on the initial approach project. (Styles, 2024)

4.2.8 GPU Watch

The project will be tested on a Samsung Galaxy A50 phone. Samsung delivers a tool called GPUWatch which is said to be:

> *"A tool for observing GPU activity in your application. GPUWatch is made for developers who use Samsung devices to get GPU related information with the least effort. Detailed information is overlaid onto the screen in real-time. And it is very simple to enable -no PC required."* (Samsung Developers, n.d.)

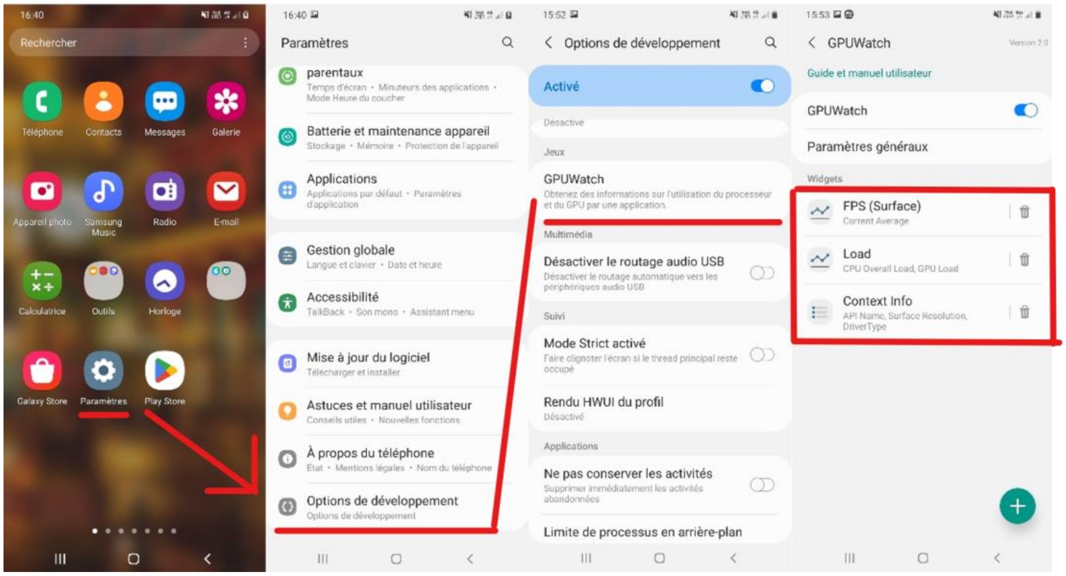

16:40
Rechercher
Téléphone
Contacts
Messages
Galerie
Appareil photo
Samsung Music
Radio
E-mail
Calculatrice
Outils
Horloge
Galaxy Store
Paramètres
Play Store
16:40
Paramètres
parentaux
Temps d'écran • Minuteurs des applications • Mode Heure du coucher
Batterie et maintenance appareil
Stockage • Mémoire • Protection de l'appareil
Applications
Applications par défaut • Paramètres d'application
Gestion globale
Langue et clavier • Date et heure
Accessibilité
TalkBack • Son mono • Assistant menu
Mise à jour du logiciel
Télécharger et installer
Astuces et manuel utilisateur
Conseils utiles • Nouvelles fonctions
À propos du téléphone
État • Mentions légales • Nom du téléphone
Options de développement
Options de développement
15:52
Options de développement
Activé
Désactivé
Jeux
GPUWatch
Obtenez des informations sur l'utilisation du processeur et du GPU par une application.
Multimédia
Désactiver le routage audio USB
Désactiver le routage automatique vers les périphériques audio USB
Suivi
Mode Strict activé
Faire clignoter l'écran si le thread principal reste occupé
Rendu HWUI du profil
Désactivé
Applications
Ne pas conserver les activités
Supprimer immédiatement les activités abandonnées
Limite de processus en arrière-plan
15:53
GPUWatch
Version 2.0
Guide et manuel utilisateur
GPUWatch
Paramètres généraux
Widgets
FPS (Surface)
Current Average
Load
CPU Overall Load, GPU Load
Context Info
API Name, Surface Resolution, DriverType
+

Figure 33 - Combination of screenshots resuming the process to activate GPUWatch on Samsung Galaxy A50. (Styles, 2024)

This tool will deliver indicative information such as the current average frame per second rate, CPU and GPU load (Figure 32, right part). The FPS value will mostly be considered as indicative since it depends on factors such as VSync and inaccurate averaging (phort99, 2015) but it will still gives a global idea of how the project behaves on the Galaxy A50 mobile phone.

4.3 Conclusion

In this chapter, the metrics used to measure the project have been defined and a list of tools that will be used have been exposed. The metrics, as seen in the first part of this chapter, are principally focused on visual performance. Any other parameters except for edge cases, where significant differences would lead to a frame fluidity problem, will not be retained. Regarding the list of tools, the main focus will be on those available in Unity. The others might end up being useful if the main set of tools does not allow sufficient analysis results.

5. Practical Project

The objective of the project is to be able to understand how shell-texturing is created and reproduce a shader mimicking the styles of *Genshin Impact*’s grass and create it in a way that will allow multiple usages across different scenes of diverse complexities. For that purpose, the shader will have to be able to adapt to the given objects, those being complex, like furry characters, or simple objects like planes, cubes or spheres. The project will ultimately allow measurements to be carried out on multiple devices and finally understand the shader’s limitations regarding scene complexity on different hardware.

5.1 Development environment

5.1.1 The hardware used

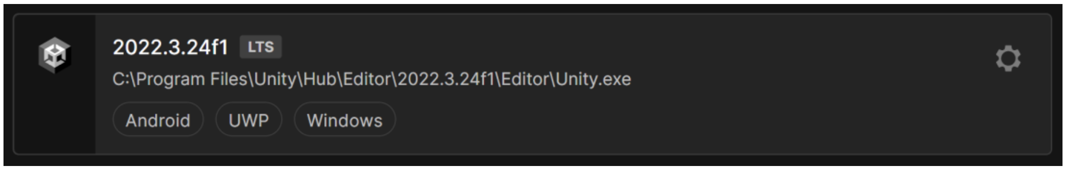
The personal computer used for creation and initial testing of the project is an Acer ConceptD with an Intel Core i7 9750H at 2.60GHz, which has 32 Gbytes of DDR4 memory, an Nvidia GeForce RTX 2060 graphics card and an 4K Ultra HD (3840 x 2160) 60Hz display.

The Samsung mobile phone used for the tests has the following description:

“The Galaxy A50 has a 6.4" FHD+ (2340 x 1080) 60 Hz Super AMOLED Infinity-U display, with an [sic] 19.5:9 aspect ratio. It is powered by an Octa-core, 4x2.3 GHz ARM Cortex-A73 and 4x1.7 GHz ARM Cortex-A53, 64-bit, 10 nm CPU and a Mali-G72 MP3 GPU.” (Wikipedia, 2024)

5.1.2 The software used

Regarding the software and tools, this project uses Unity version 2022.3.24f1, the latest LTS version available. All the sets of Unity tools used for measurement are related to that version.



2022.3.24f1 LTS

C:\Program Files\Unity\Hub\Editor\2022.3.24f1\Editor\Unity.exe

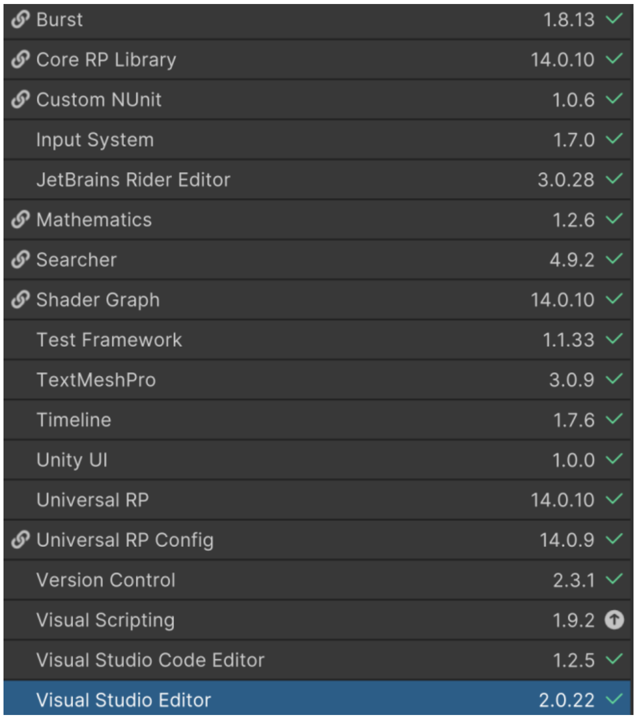
Android UWP Windows

Figure 34 - UnityHub screenshot of the Installs category showing the 2022.3.24f1 LTS version. (Styles, 2024)

The additional packages used for this project were the following:

- Input System 1.7.0
- TextMeshPro 3.0.9
- Universal RP 14.0.10 & Universal RP Config 14.0.9
- Visual Studio Editor 2.0.22

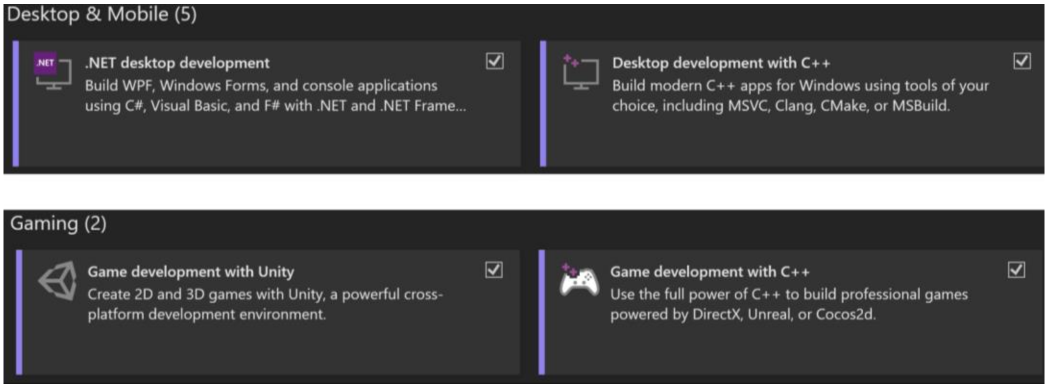
The JetBrains Rider editor 3.0.28 was installed in the Unity packages but disabled in the IDE.



🔗 Burst	1.8.13	✓
🔗 Core RP Library	14.0.10	✓
🔗 Custom NUnit	1.0.6	✓
Input System	1.7.0	✓
JetBrains Rider Editor	3.0.28	✓
🔗 Mathematics	1.2.6	✓
🔗 Searcher	4.9.2	✓
🔗 Shader Graph	14.0.10	✓
Test Framework	1.1.33	✓
TextMeshPro	3.0.9	✓
Timeline	1.7.6	✓
Unity UI	1.0.0	✓
Universal RP	14.0.10	✓
🔗 Universal RP Config	14.0.9	✓
Version Control	2.3.1	✓
Visual Scripting	1.9.2	⬆
Visual Studio Code Editor	1.2.5	✓
Visual Studio Editor	2.0.22	✓

Figure 35 - Unity editor Package manager tool showing the installed packages. (Styles, 2024)

The IDE used for development is the Visual Studio Community 2019 version 16.11.34 with the .NET desktop development and Game development with Unity packages installed.



Desktop & Mobile (5)

.NET desktop development ☑
Build WPF, Windows Forms, and console applications using C#, Visual Basic, and F# with .NET and .NET Frame...

Desktop development with C++ ☑
Build modern C++ apps for Windows using tools of your choice, including MSVC, Clang, CMake, or MSBuild.

Gaming (2)

Game development with Unity ☑
Create 2D and 3D games with Unity, a powerful cross-platform development environment.

Game development with C++ ☑
Use the full power of C++ to build professional games powered by DirectX, Unreal, or Cocos2d.

Figure 36 - Visual Studio IDE Installer showing the packages category. (Styles, 2024)

The RenderDoc version is the 1.33 using Qt version 5.15.2 and the Nvidia Nsight Graphics version is the 2024.1.0.0, build 34057410.

5.2 Limitations

Throughout this project some of the initial intentions were not fulfilled. Regarding the project itself, a limiting decision was pre-emptively made by following the stylised style of *Genshin Impact*, discarding more realistic texturing and lighting to focus of the main objective that was to find an optimal solution to the camera's perspective and compare it to a project that was effectively made in Unity.

In its current state the points described in 2.4.2.1 & 2.4.2.2 are successful but required a considerable amount of time between the learning of the HLSL and the transition of the basic CG implementation to the HLSL version.

Point 2.4.2.3 was attempted but was not successful due to a lack of expertise regarding the technical challenge of merging two distinct methods in a single shader and an underestimation of the time required for such a project. That element was the main reason for time-loss on the project.

The 2.4.2.4 was done despite the previous blocking objective.

The 2.4.2.5 was accomplished but not tested on a wide range of machines as initially intended since only the Acer ConceptD and the Samsung Galaxy A50 were continuously available. The learning process and high challenge coupled with time-loss due to compilation and coding errors did not allow the build to be done on Nintendo Switch.

The 2.4.2.6 was successful and added a layer of improvement to the shader with shadows and simple physical behaviour.

5.3 Projects procedure

The implementation of shell texturing as described in 2.2.1 relies on the extraction of layers from an object to colour them degressively according to a given noise. The initial project 2.4.2.1 used a Unity sphere object containing a camera in a scene with a directional light, a canvas prefab to draw UI and an event system component to be able to navigate in the UI.

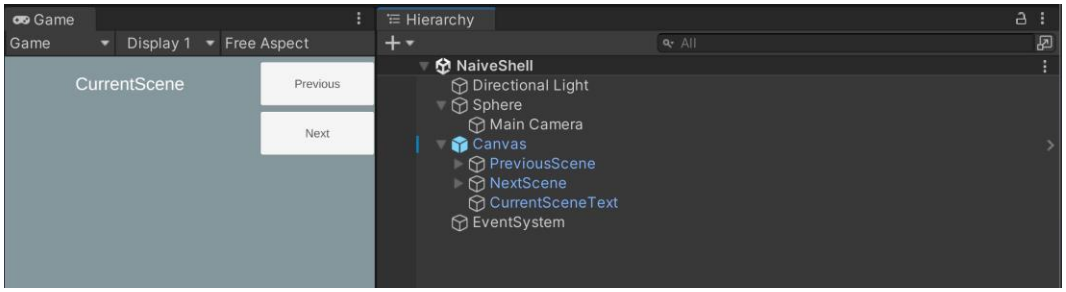
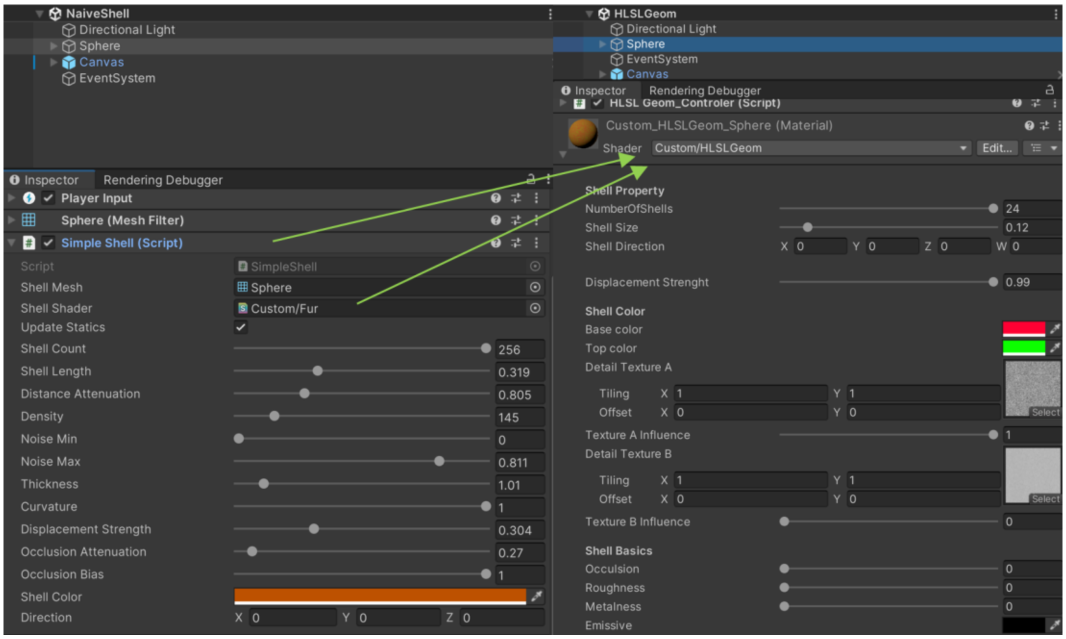


Figure 37 - Unity editor NaïveShell project exposing the Canvas prefab in the Hierarchy. (Styles, 2024)

The layer extraction was done with a C# script that communicated with a shader file for coloration written in CG. The full HLSL version 2.4.2.2 contains the same elements with the exception of the C# script and shader file which were replaced by a shader file written in HLSL.



NaiveShell
Directional Light
Sphere
Canvas
EventSystem

Inspector | Rendering Debugger

Player Input
Sphere (Mesh Filter)
Simple Shell (Script)

Script: SimpleShell
Shell Mesh: Sphere
Shell Shader: Custom/Fur
Update Statics: ✓
Shell Count: 256
Shell Length: 0.319
Distance Attenuation: 0.805
Density: 145
Noise Min: 0
Noise Max: 0.811
Thickness: 1.01
Curvature: 1
Displacement Strength: 0.304
Occlusion Attenuation: 0.27
Occlusion Bias: 1
Shell Color
Direction: X 0 Y 0 Z 0

HLSLGeom
Directional Light
Sphere
EventSystem
Canvas

Inspector | Rendering Debugger

HLSL Geom_Controler (Script)
Custom_HLSLGeom_Sphere (Material)
Shader: Custom/HLSLGeom

Shell Property
NumberOfShells: 24
Shell Size: 0.12
Shell Direction: X 0 Y 0 Z 0 W 0

Displacement Strenght: 0.99

Shell Color
Base color
Top color
Detail Texture A
Tiling: X 1 Y 1
Offset: X 0 Y 0
Texture A Influence: 1
Detail Texture B
Tiling: X 1 Y 1
Offset: X 0 Y 0
Texture B Influence: 0

Shell Basics
Occulsion: 0
Roughness: 0
Metalness: 0
Emissive

Figure 38 - Unity editor screenshots of two Inspectors exposing the SimpleShell script and HLSLGeom shader material parameters. (Styles, 2024)

The resolution of the problem of the camera's perspective described in 2.4.2.3 was attempted with a sphere containing two materials and an HLSL shader that combines both shell and fin texturing, but the results were not visually pleasing.

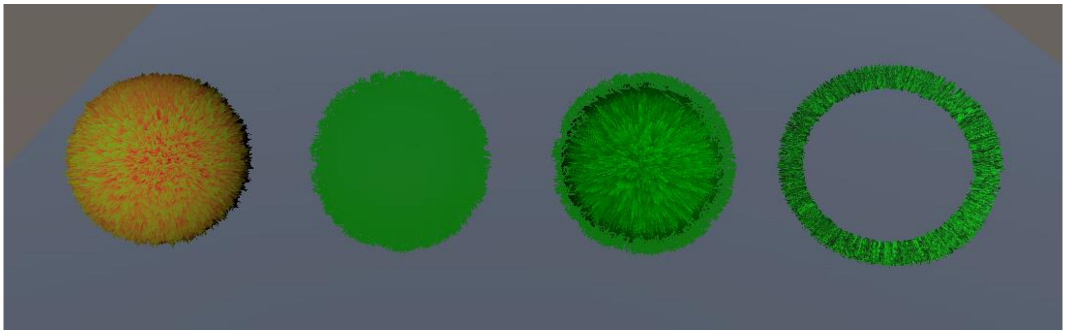
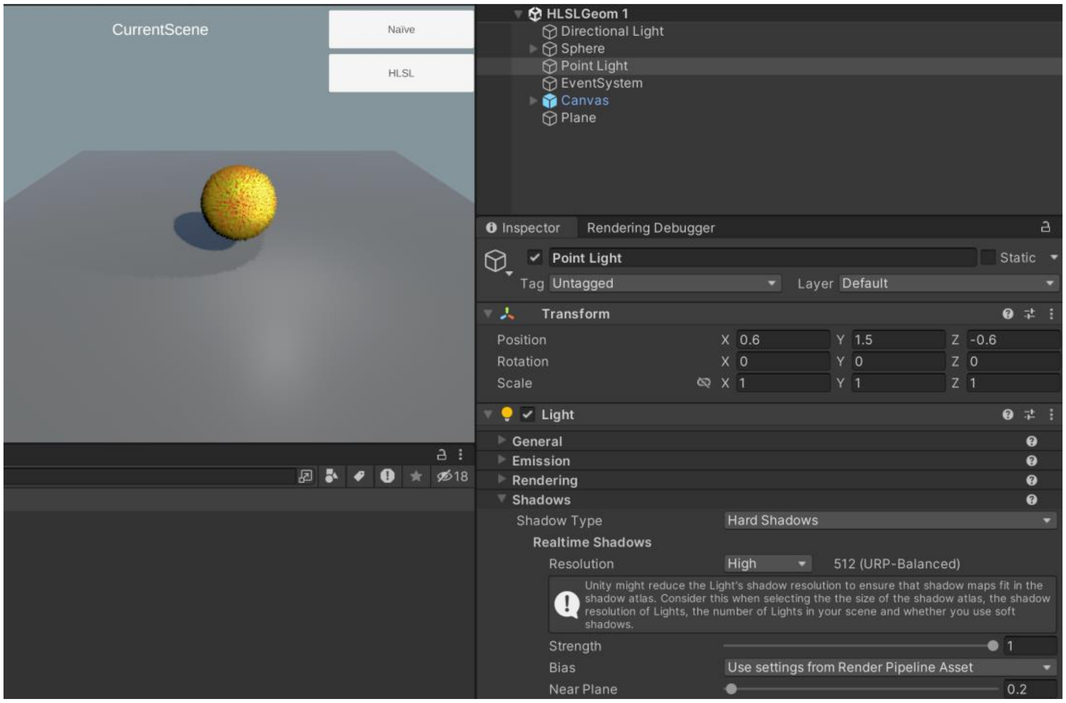


Figure 39 - Unity scene showing attempts at Shell and Fin shader merging. (Styles, 2024)

The implementation was abandoned at that point in order to advance on the project.

The objective described in 2.4.2.4 was first applied by adding a point light and a plane in the scene to add a layer of complexity and measure the impact of lighting and shadow projection.



Point Light — Static
Tag Untagged — Layer Default

Position X 0.6 Y 1.5 Z -0.6
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1

Shadow Type: Hard Shadows
Realtime Shadows
Resolution: High — 512 (URP-Balanced)

Unity might reduce the Light's shadow resolution to ensure that shadow maps fit in the shadow atlas. Consider this when selecting the the size of the shadow atlas, the shadow resolution of Lights, the number of Lights in your scene and whether you use soft shadows.

Strength: 1
Bias: Use settings from Render Pipeline Asset
Near Plane: 0.2

Figure 40 - Unity editor screenshot showing the HLSLGeom1 scene with light information in the inspector. (Styles, 2024)

The second level of diversification was achieved by adding a skybox, a plane and 18 spheres in the scene to simulate a complex environment and measuring how the shader impacts performance when widely used.

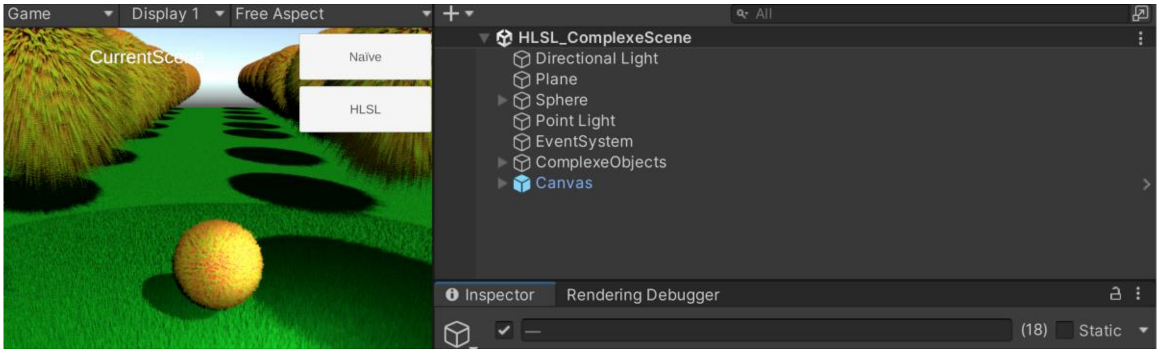
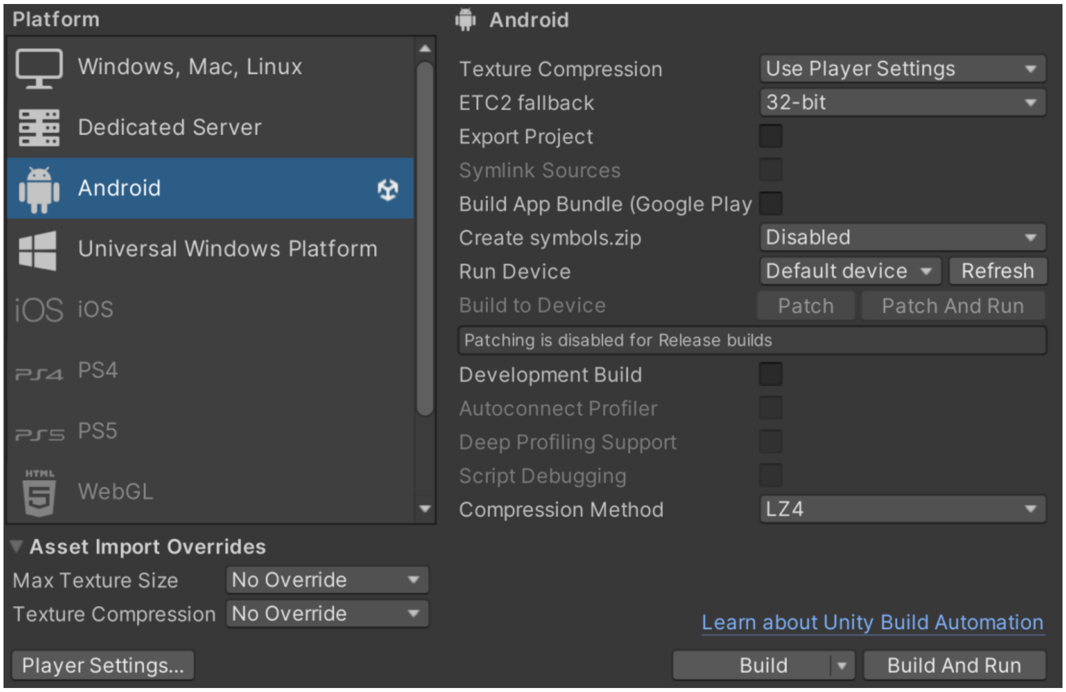


Figure 41 - Unity editor screenshot showing the HLSL_ComplexScene. (Styles, 2024)

The hardware implementation described in 2.4.2.5 was done by building the project for Android and windows through Unity's build settings.

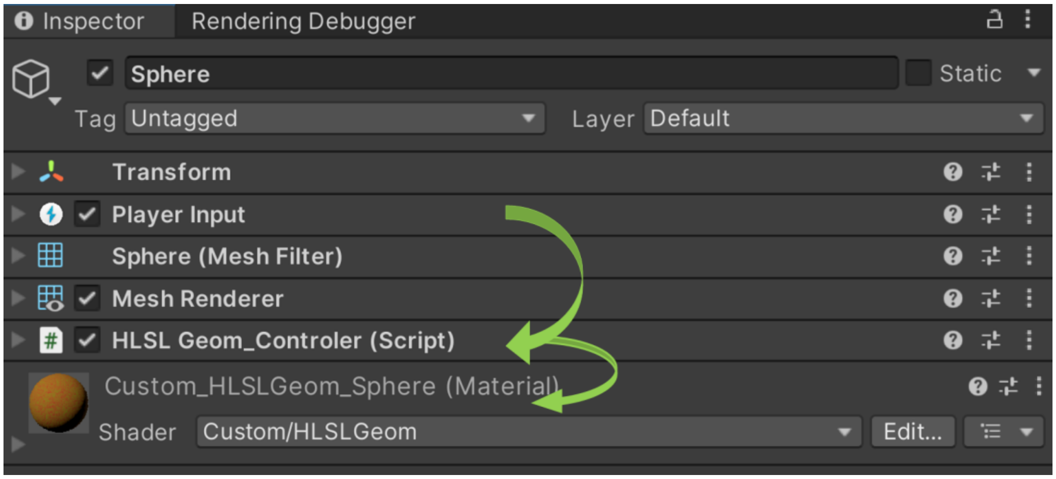


Texture Compression: Use Player Settings
ETC2 fallback: 32-bit
Export Project ☐
Symlink Sources ☐
Build App Bundle (Google Play ☐
Create symbols.zip: Disabled
Run Device: Default device
Build to Device
Patching is disabled for Release builds
Development Build ☐
Autoconnect Profiler ☐
Deep Profiling Support ☐
Script Debugging ☐
Compression Method: LZ4

Asset Import Overrides
Max Texture Size: No Override
Texture Compression: No Override

Figure 42 - Capture of the Unity Build Settings window. (Styles, 2024)

The improvements described in 2.4.2.6 were added after analysing shader code given by Hecomi for light interaction and shadows. The physical interaction was accomplished by gradually modifying the shader and transmitting the user’s input using Unity’s input system and a C# script to transmit information to the shader.



☑ Sphere ☐ Static
Tag: Untagged Layer: Default
Transform
☑ Player Input
Sphere (Mesh Filter)
☑ Mesh Renderer
☑ HLSL Geom_Controler (Script)
Custom_HLSLGeom_Sphere (Material)
Shader: Custom/HLSLGeom

Figure 43 - Unity editor screenshot of the Inspector window showing the relation between player input and HLSL shader. (Styles, 2024)

Additionally, an asset downloaded from TurboSquid (TurboSquid, n.d.) was integrated into the project and the collaboration of a game artist on the creation of textures coupled with some shader modifications brought the results presented in the figure below (Figure 43).

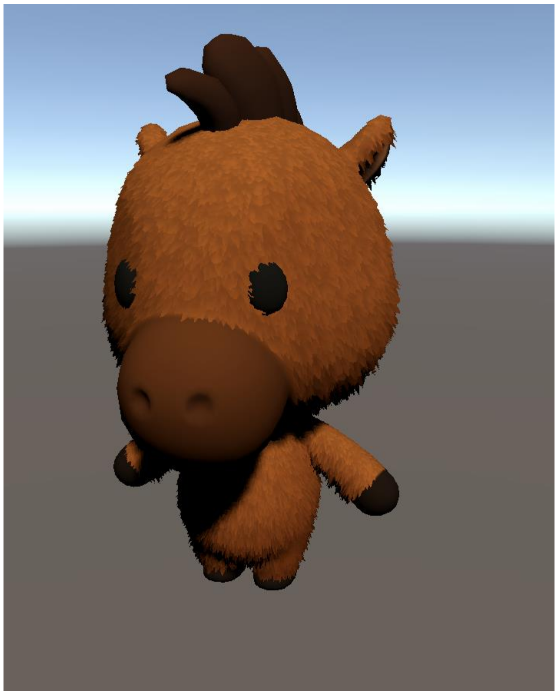


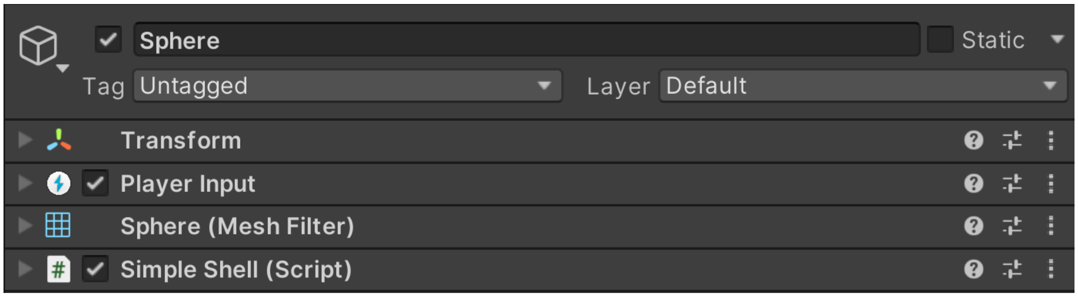
Figure 44 - Unity editor screenshot showing the rigger horse from TruboSquid with the HLSL shader on it. (Styles, 2024)

The last improvement added to the project was a scene manager with basic UI to be able to transition between scenes without having to build and run each scene independently.

5.4 Technical implementation

5.4.1 Initial approach

The initial project was created using a Unity Sphere Mesh Filter as the base mesh on which the fur is rendered. It was initially created thanks to a video of Garrett Gunnell that explained the process and the general idea for the code (Acerola, 2023).



☑ Sphere ☐ Static

Tag Untagged Layer Default

Figure 45 - Unity capture showing the Sphere object's inspector. (Styles, 2024)

The extraction of layers in the first version is done first with a C# script by adding extra objects containing MeshFilters and MeshRenderers, parenting them to the initial object and assigning them an index for each layer.

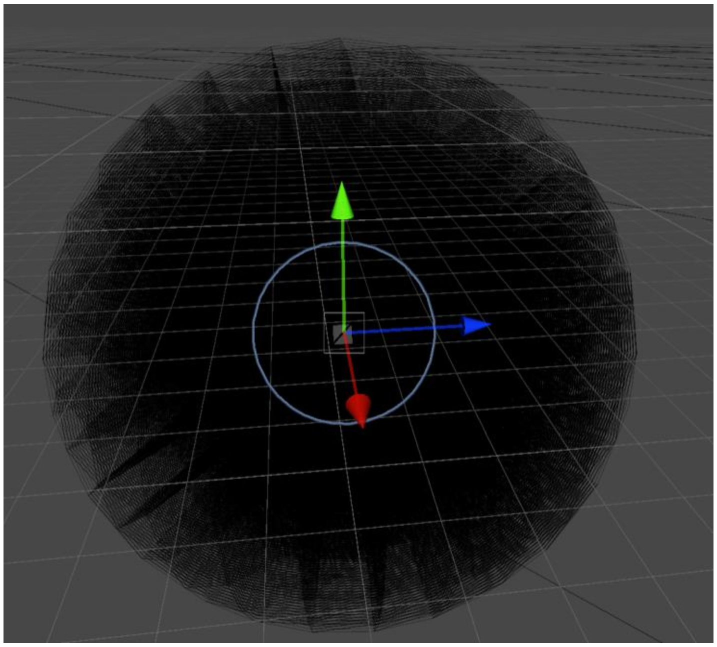
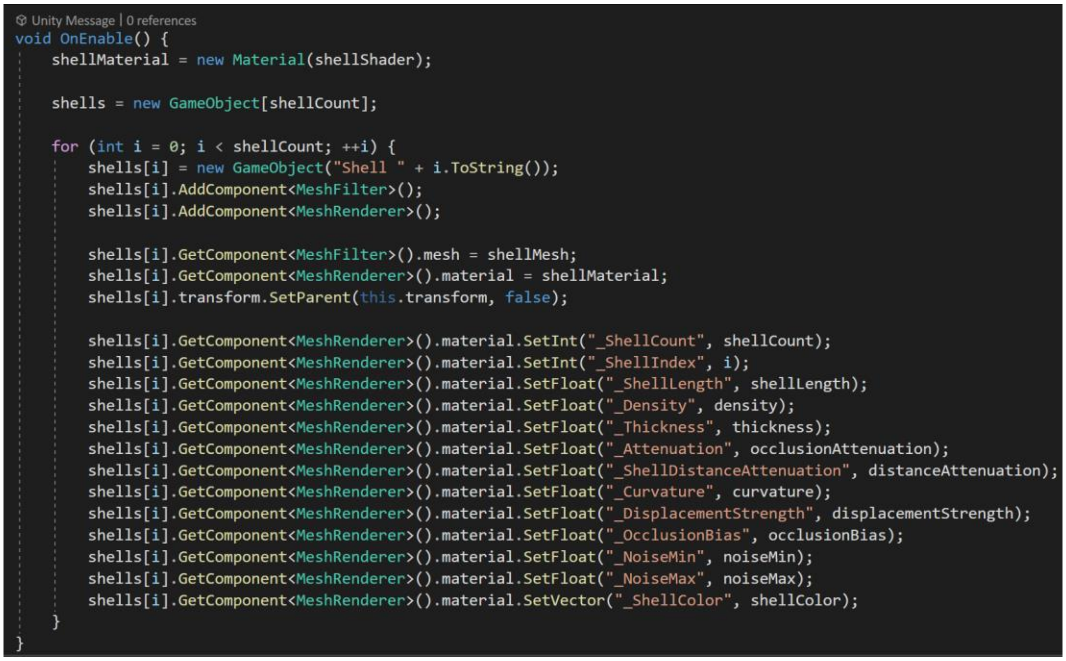


Figure 46 - Unity editor showing the mesh layering from the initial approach project. (Styles, 2024)

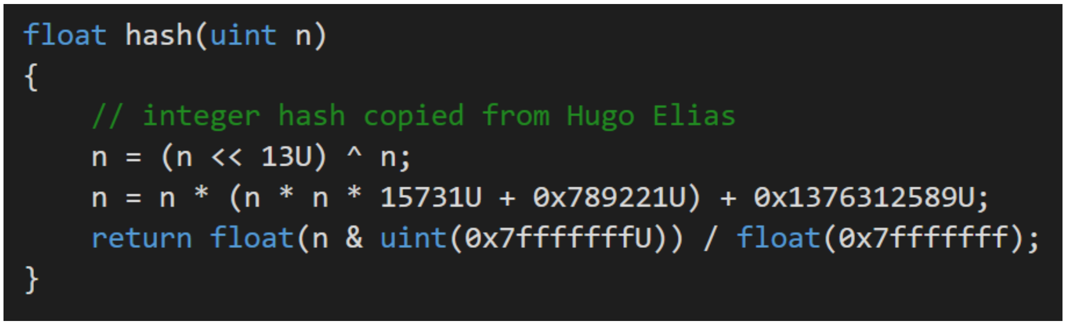
Afterwards the layer index is transferred to the HLSL shader so that it knows what object to modify. Additional parameters contained in the Simple Shell script are also transferred to the shader at the same time. (Styles, 2024) (see Appendix E).



```
void OnEnable() {
    shellMaterial = new Material(shellShader);

    shells = new GameObject[shellCount];

    for (int i = 0; i < shellCount; ++i) {
        shells[i] = new GameObject("Shell " + i.ToString());
        shells[i].AddComponent<MeshFilter>();
        shells[i].AddComponent<MeshRenderer>();

        shells[i].GetComponent<MeshFilter>().mesh = shellMesh;
        shells[i].GetComponent<MeshRenderer>().material = shellMaterial;
        shells[i].transform.SetParent(this.transform, false);

        shells[i].GetComponent<MeshRenderer>().material.SetInt("_ShellCount", shellCount);
        shells[i].GetComponent<MeshRenderer>().material.SetInt("_ShellIndex", i);
        shells[i].GetComponent<MeshRenderer>().material.SetFloat("_ShellLength", shellLength);
        shells[i].GetComponent<MeshRenderer>().material.SetFloat("_Density", density);
        shells[i].GetComponent<MeshRenderer>().material.SetFloat("_Thickness", thickness);
        shells[i].GetComponent<MeshRenderer>().material.SetFloat("_Attenuation", occlusionAttenuation);
        shells[i].GetComponent<MeshRenderer>().material.SetFloat("_ShellDistanceAttenuation", distanceAttenuation);
        shells[i].GetComponent<MeshRenderer>().material.SetFloat("_Curvature", curvature);
        shells[i].GetComponent<MeshRenderer>().material.SetFloat("_DisplacementStrength", displacementStrength);
        shells[i].GetComponent<MeshRenderer>().material.SetFloat("_OcclusionBias", occlusionBias);
        shells[i].GetComponent<MeshRenderer>().material.SetFloat("_NoiseMin", noiseMin);
        shells[i].GetComponent<MeshRenderer>().material.SetFloat("_NoiseMax", noiseMax);
        shells[i].GetComponent<MeshRenderer>().material.SetVector("_ShellColor", shellColor);
    }
}
```

Figure 47 - Capture of the SimpleShell.cs script in the Visual Studio IDE. (Styles, 2024)

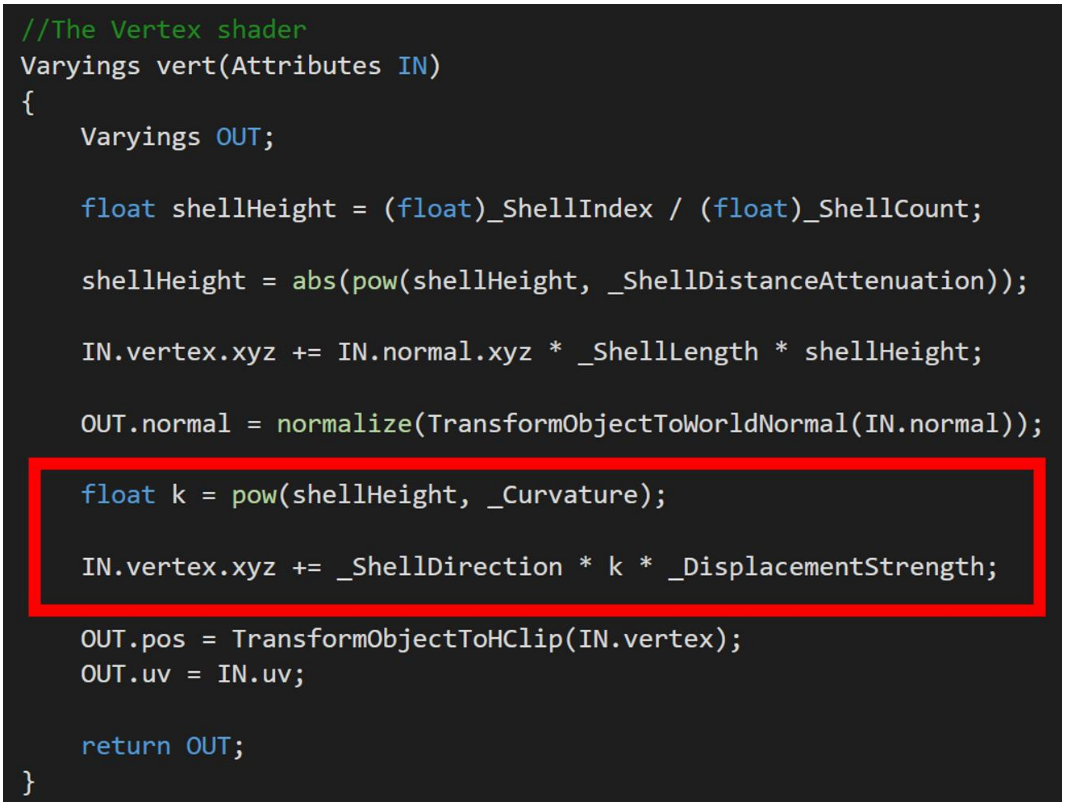
The shader, originally written in CG (Styles, 2024) (see Appendix C) was translated to HLSL (Styles, 2024) (see Appendix D) for the purpose of compatibility with the URP pipeline.



```
float hash(uint n)
{
    // integer hash copied from Hugo Elias
    n = (n << 13U) ^ n;
    n = n * (n * n * 15731U + 0x789221U) + 0x1376312589U;
    return float(n & uint(0x7fffffffU)) / float(0x7fffffff);
}
```

Figure 48 - Capture of Hugo Elias’ hashing function from the Shell.shader script in the Visual Studio IDE. (Styles, 2024)

A hash function was borrowed from Hugo Elias and found on ShaderToy (Beautypi, n.d.) to be able to rely on generated noise instead of a noise texture.

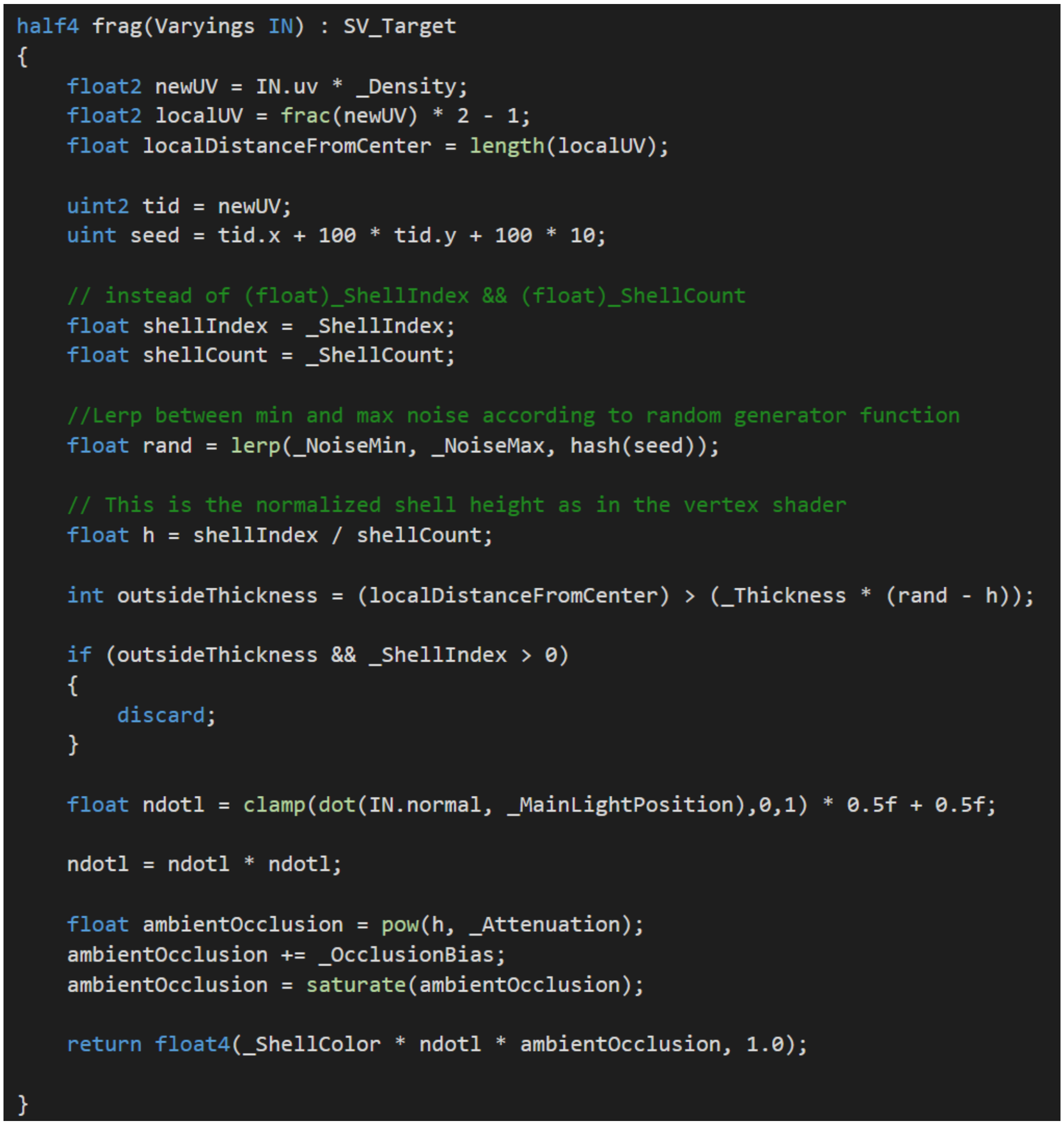


```
//The Vertex shader
Varyings vert(Attributes IN)
{
    Varyings OUT;

    float shellHeight = (float)_ShellIndex / (float)_ShellCount;

    shellHeight = abs(pow(shellHeight, _ShellDistanceAttenuation));

    IN.vertex.xyz += IN.normal.xyz * _ShellLength * shellHeight;

    OUT.normal = normalize(TransformObjectToWorldNormal(IN.normal));

    float k = pow(shellHeight, _Curvature);

    IN.vertex.xyz += _ShellDirection * k * _DisplacementStrength;

    OUT.pos = TransformObjectToHClip(IN.vertex);
    OUT.uv = IN.uv;

    return OUT;
}
```

Figure 49 - Capture of the Vertex shader code from the Shell.shader script in the Visual Studio IDE. (Styles, 2024)

The shader itself works in two parts. First the vertex shader, called Varyings in URP, is responsible for displacing the meshes along the normal vectors according to a previously calculated height obtained from the index of the currently treated shell divided by the number of shells. The displacement modifications (Figure 48, Red rectangle) change the orientation of the fur at run-time with a C# script and Unity’s input system (Styles, 2024) (see Appendix L).

The second part of the shader, the fragment shader, is responsible for the colouration of the different meshes.



```
half4 frag(Varyings IN) : SV_Target
{
    float2 newUV = IN.uv * _Density;
    float2 localUV = frac(newUV) * 2 - 1;
    float localDistanceFromCenter = length(localUV);

    uint2 tid = newUV;
    uint seed = tid.x + 100 * tid.y + 100 * 10;

    // instead of (float)_ShellIndex && (float)_ShellCount
    float shellIndex = _ShellIndex;
    float shellCount = _ShellCount;

    //Lerp between min and max noise according to random generator function
    float rand = lerp(_NoiseMin, _NoiseMax, hash(seed));

    // This is the normalized shell height as in the vertex shader
    float h = shellIndex / shellCount;

    int outsideThickness = (localDistanceFromCenter) > (_Thickness * (rand - h));

    if (outsideThickness && _ShellIndex > 0)
    {
        discard;
    }

    float ndotl = clamp(dot(IN.normal, _MainLightPosition),0,1) * 0.5f + 0.5f;

    ndotl = ndotl * ndotl;

    float ambientOcclusion = pow(h, _Attenuation);
    ambientOcclusion += _OcclusionBias;
    ambientOcclusion = saturate(ambientOcclusion);

    return float4(_ShellColor * ndotl * ambientOcclusion, 1.0);

}
```

Figure 50 - Capture of the fragment shader code from the Shell.shader script in the Visual Studio IDE. (Styles, 2024)

It uses the hash function with a previously calculated number obtained from the UV coordinates to generate a random height value and define what part of the different meshes are coloured of not. A check is then done to discard pixels that do not correspond to the current height threshold.

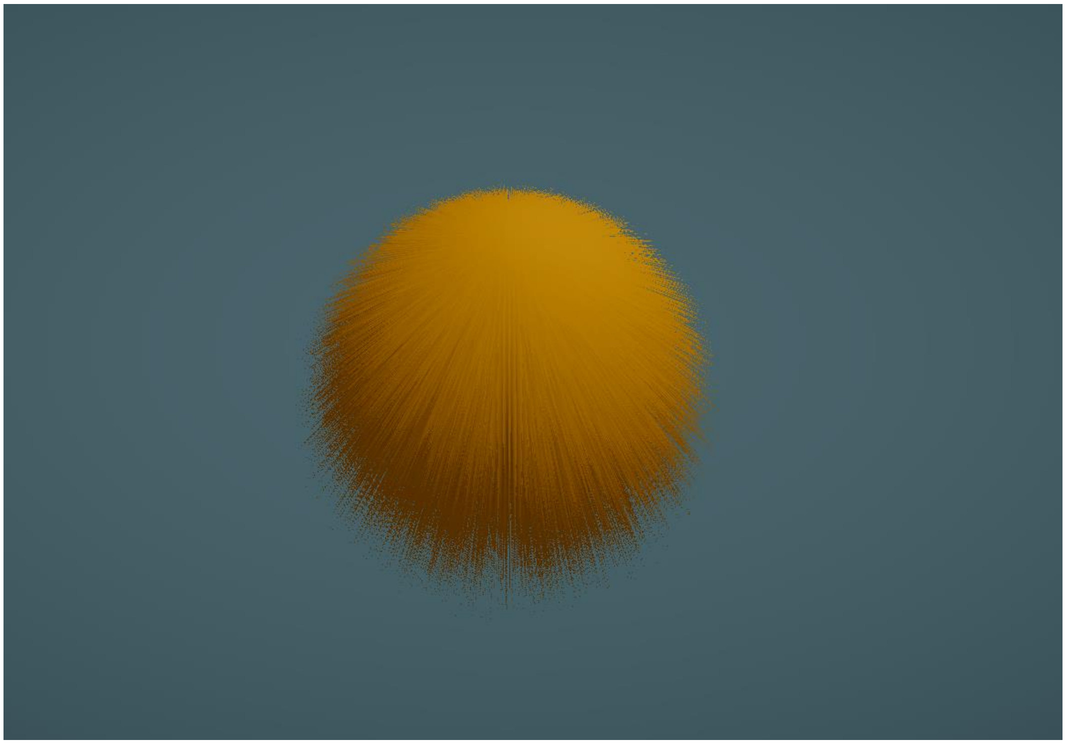
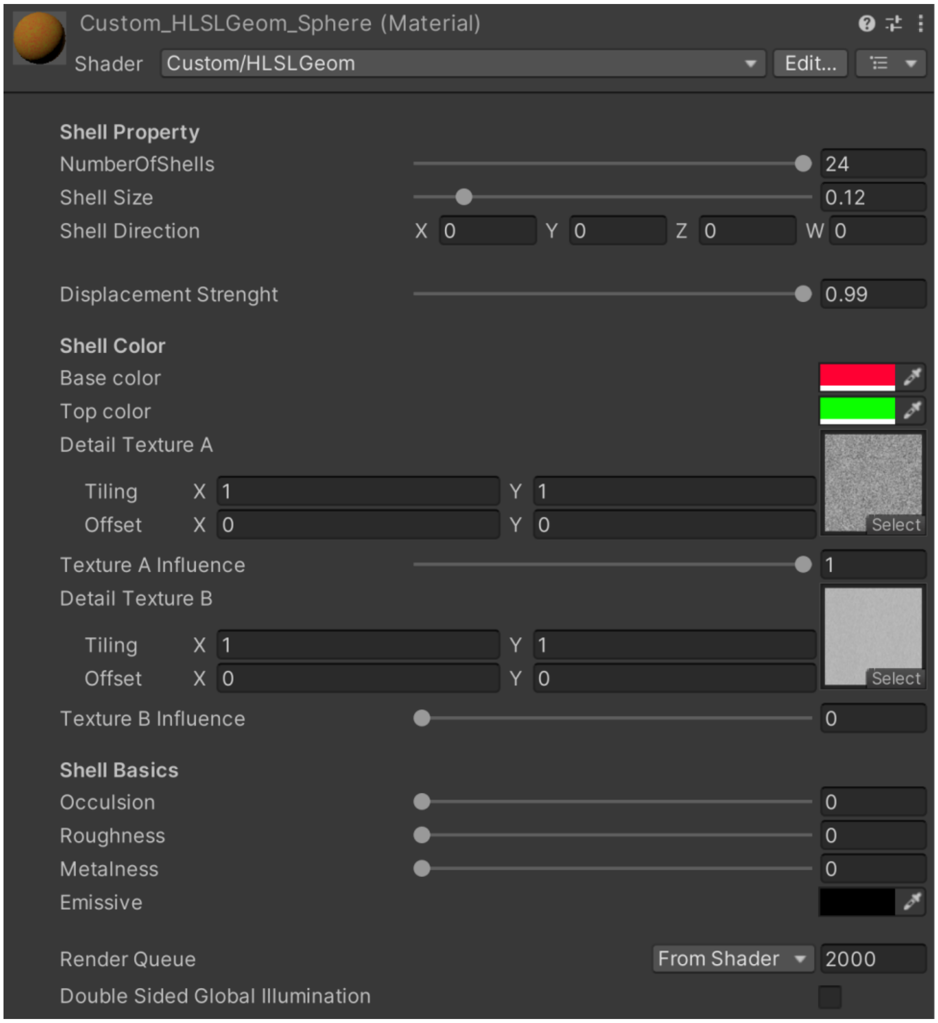


Figure 51 - Capture of the initial approach scene in the Unity editor. (Styles, 2024)

The colour is then applied to the non-discarded pixels using the half-lambert lighting model (Jordan Stevens, n.d.)

5.4.2 HLSL

The full HLSL version's parameters and layer extraction are all contained in the shader files, giving access to them directly in the material parameters of the inspector window.



Custom_HLSLGeom_Sphere (Material)
Shader Custom/HLSLGeom

Shell Property
NumberOfShells 24
Shell Size 0.12
Shell Direction X 0 Y 0 Z 0 W 0

Displacement Strenght 0.99

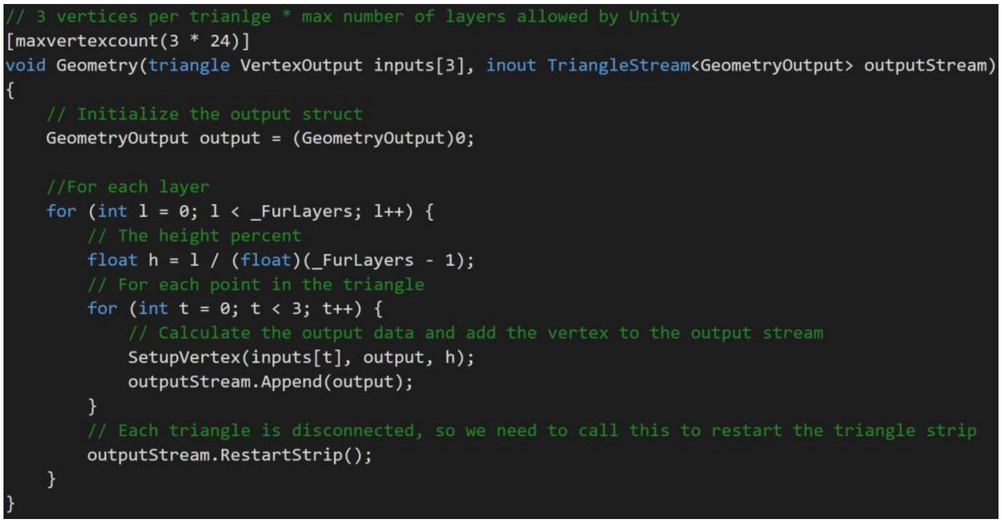
Shell Color
Base color 🟥
Top color 🟩
Detail Texture A
Tiling X 1 Y 1
Offset X 0 Y 0
Texture A Influence 1
Detail Texture B
Tiling X 1 Y 1
Offset X 0 Y 0
Texture B Influence 0

Shell Basics
Occulsion 0
Roughness 0
Metalness 0
Emissive ⬛

Render Queue From Shader 2000
Double Sided Global Illumination ☐

Figure 52 - Unity editor capture of the HLSLGeom shader's material showing parameters in the inspector window. (Styles, 2024)

This modification is achieved by first declaring the properties in the shader file (Styles, 2024) (see Appendix G). The body of the programme is declared in a separate file to avoid code repetition and improve readability.



```
// 3 vertices per trianlge * max number of layers allowed by Unity
[maxvertexcount(3 * 24)]
void Geometry(triangle VertexOutput inputs[3], inout TriangleStream<GeometryOutput> outputStream)
{
    // Initialize the output struct
    GeometryOutput output = (GeometryOutput)0;

    //For each layer
    for (int l = 0; l < _FurLayers; l++) {
        // The height percent
        float h = l / (float)(_FurLayers - 1);
        // For each point in the triangle
        for (int t = 0; t < 3; t++) {
            // Calculate the output data and add the vertex to the output stream
            SetupVertex(inputs[t], output, h);
            outputStream.Append(output);
        }
        // Each triangle is disconnected, so we need to call this to restart the triangle strip
        outputStream.RestartStrip();
    }
}
```

Figure 53 - Capture of the Geometry shader code from the Fur.hlsl script in the Visual Studio IDE. (Styles, 2024)

The extraction of the mesh previously done with the C# script is done with a geometry shader and the SetupVertex function and the fragment programme that used a pseudo-random generator is replaced by two noise textures (Styles, 2024) (see Appendix H).

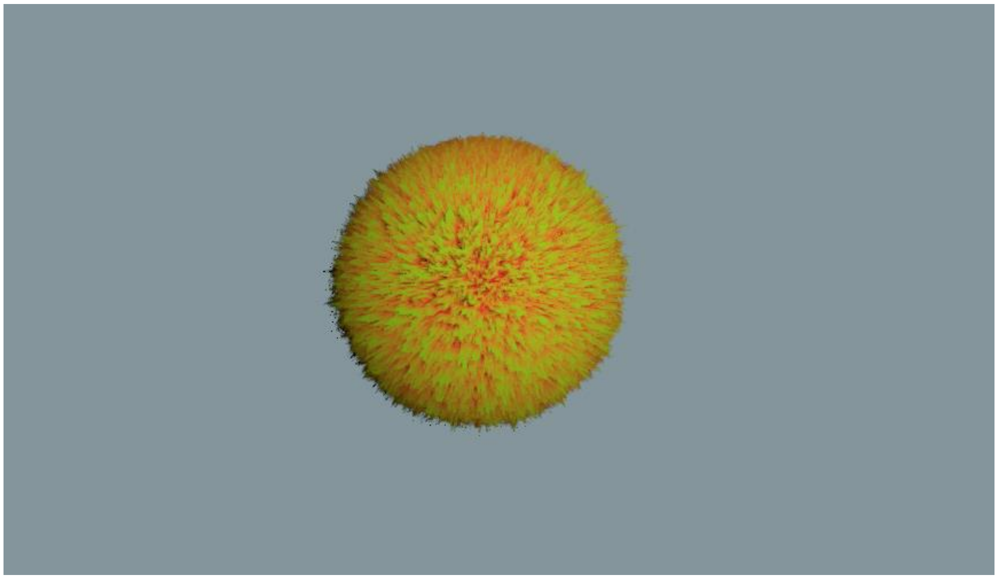
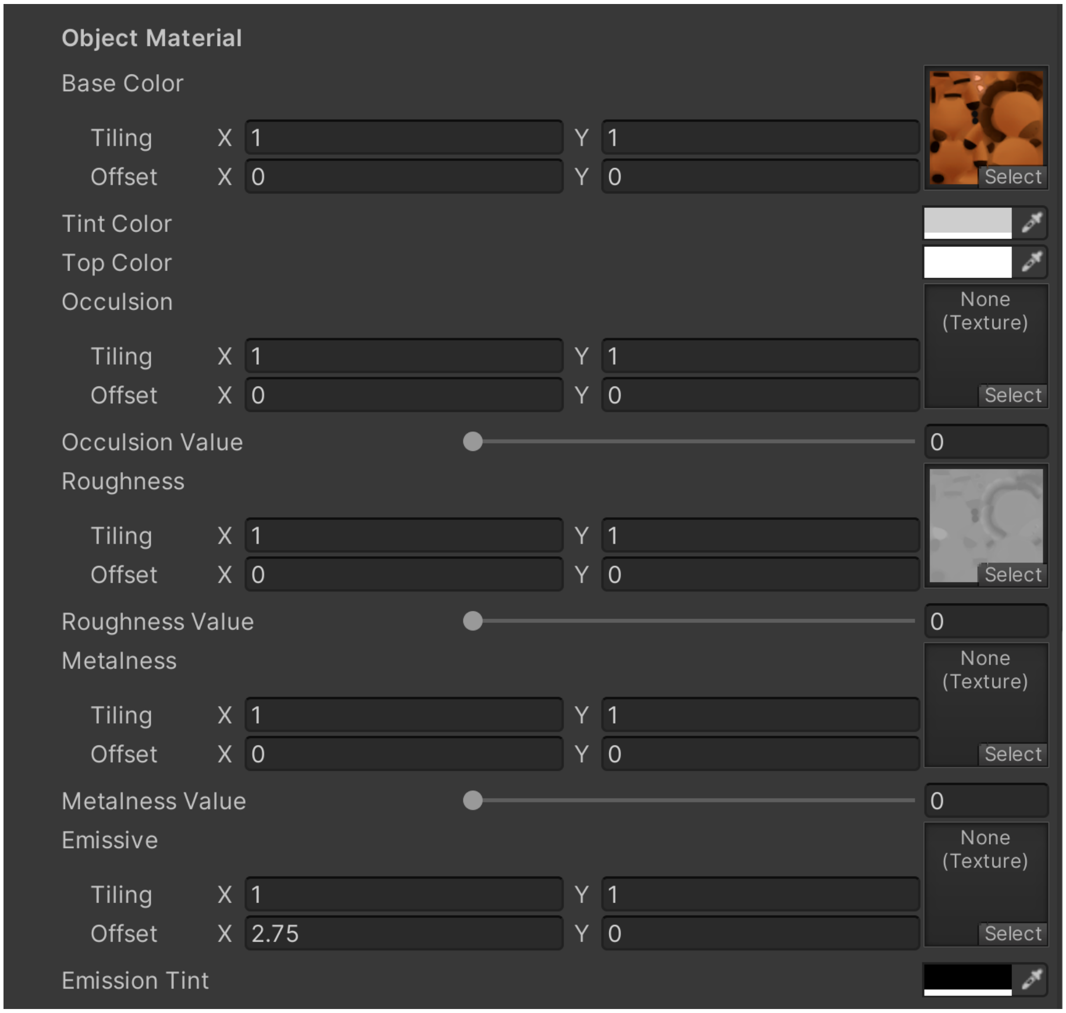


Figure 54 - Unity editor capture of the HLSLGeom scene. (Styles, 2024)

The analysis of Hecomi’s code added a layer of improvement regarding lighting and shadows with the addition of multiple Unity keywords, a shadow caster pass and functions for shadow calculations (Styles, 2024) (see Appendix I). Finally, the use of Unity’s UniversalFragmentPBR function replaces the previous lighting model (“Unity-Technologies/Graphics”, n.d.-h).

5.4.3 HLSL Complex object

The last version of the shader incorporates the use of textures for the base colour, occlusion, roughness, metalness and emissive by declaring 2D variables in the .shader file and TEXTURE2D and SAMPLER variables in the .hlsl file (Styles, 2024) (see Appendix J & K).



Object Material

Base Color
Tiling X 1 Y 1
Offset X 0 Y 0
Tint Color
Top Color
Occulsion
None (Texture)
Tiling X 1 Y 1
Offset X 0 Y 0
Occulsion Value 0
Roughness
Tiling X 1 Y 1
Offset X 0 Y 0
Roughness Value 0
Metalness
None (Texture)
Tiling X 1 Y 1
Offset X 0 Y 0
Metalness Value 0
Emissive
None (Texture)
Tiling X 1 Y 1
Offset X 2.75 Y 0
Emission Tint

Figure 55 - Unity editor capture of the material parameters in the inspector window. (Styles, 2024)

The previous values for each channel are used as influence modifiers to reduce the impact of each texture if necessary.

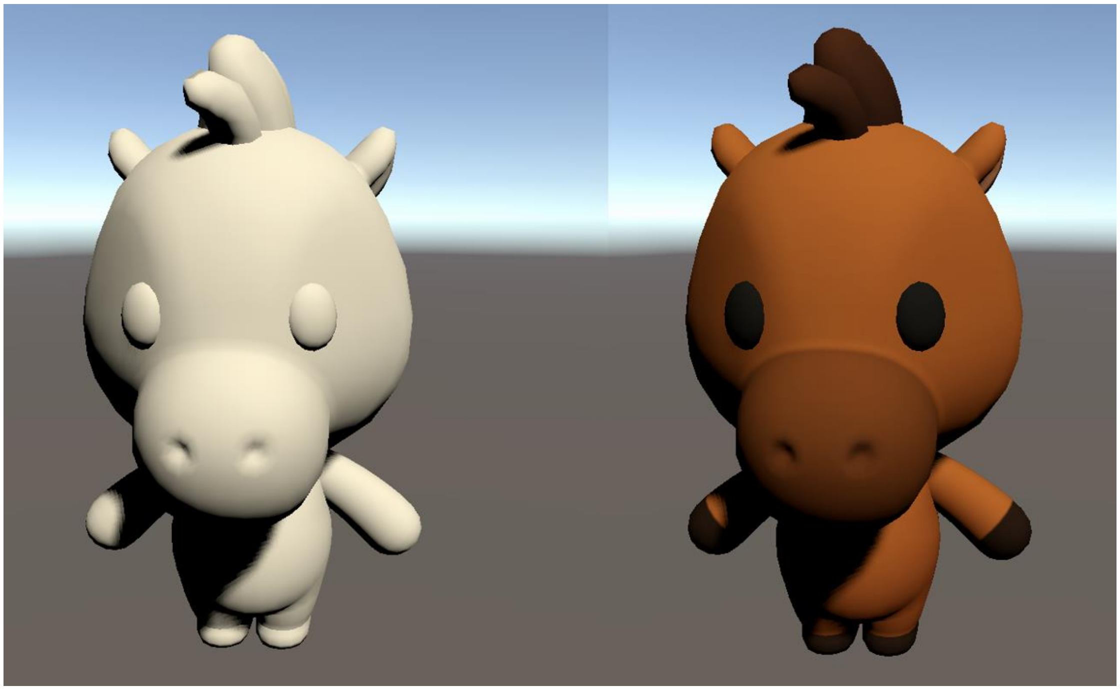
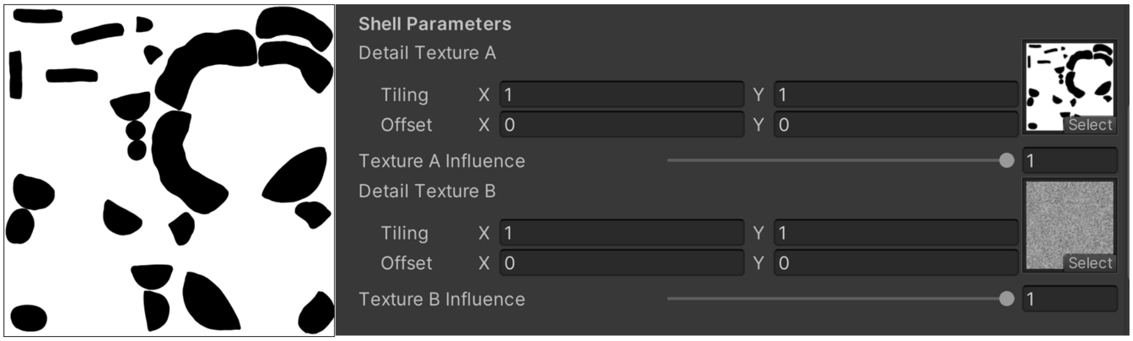


Figure 56 - Unity editor capture of the HLSL_ComplexObject scene showing the riggerd horse without colour (left) and with colour (right). (Styles, 2024)

This implementation allows complex meshes to be coloured according to the textures produced by an artist.



Shell Parameters

Detail Texture A

Tiling X 1 Y 1

Offset X 0 Y 0

Texture A Influence 1

Detail Texture B

Tiling X 1 Y 1

Offset X 0 Y 0

Texture B Influence 1

Figure 57 - Unity editor inspector window capture showing the shell detail parameters. (Styles, 2024)

The use of noise textures for the fur generation gives the possibility to define parts of the mesh that must be with or without fur by inputting a black and white texture in one of the detail channels.

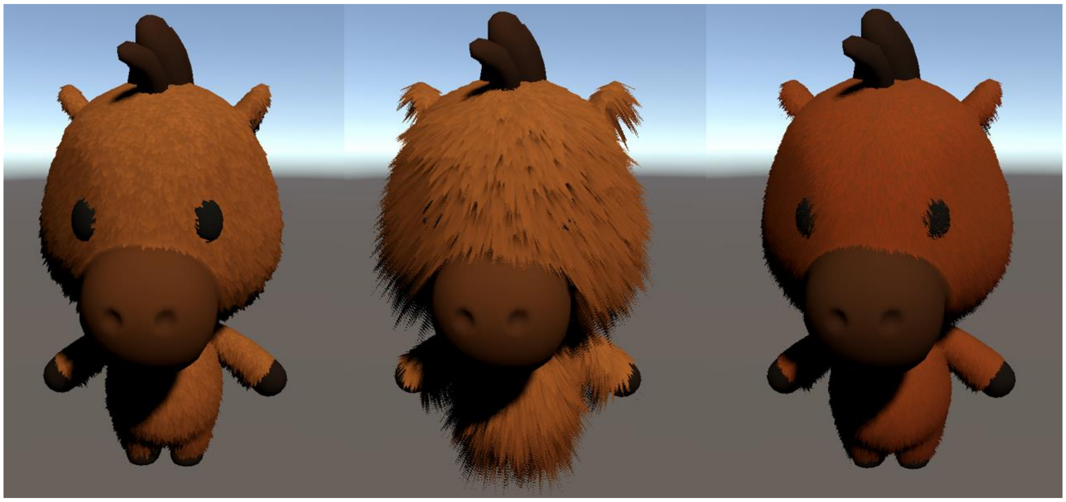
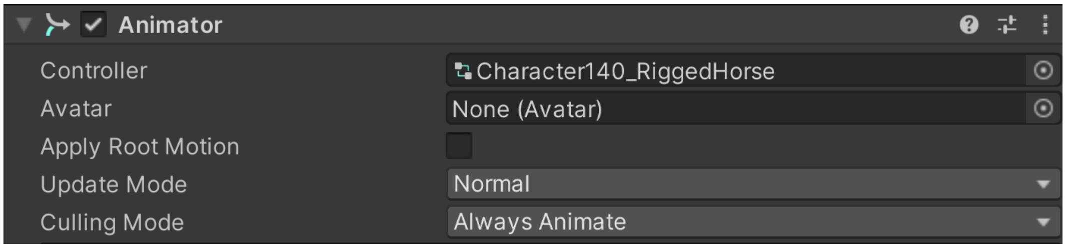


Figure 58 - Unity editor capture of three shader variations showing the rigged horse with short (right), long (middle), and red tinted (left) fur. (Styles, 2024)

With all these improvements, the users can modify the object's appearance at their convenience.

Since complex objects like the horse character (Figure 57) often use SkinMeshRenderers, a modification of the C# controller script was made to be able to take multiple renderers instead of one (Styles, 2024) (see Appendix M).



Animator

Controller	Character140_RiggedHorse
Avatar	None (Avatar)
Apply Root Motion	☐
Update Mode	Normal
Culling Mode	Always Animate

Figure 59 - Unity editor inspector capture showing the Animator component. (Styles, 2024)

Finaly, motion was added with an animator component and an animation clip to be able to visualise the fur's reaction to mesh movements in real-time (SamuelStyles, 2024).

5.4.4 UI and Scene Management

To be able to measure conveniently all the scenes exposed in 5.3 on all platforms described in 2.4.2.5, a UI was created with the Unity Canvas, Buttons and TextMeshPro – Text(UI) components (Figure 59).

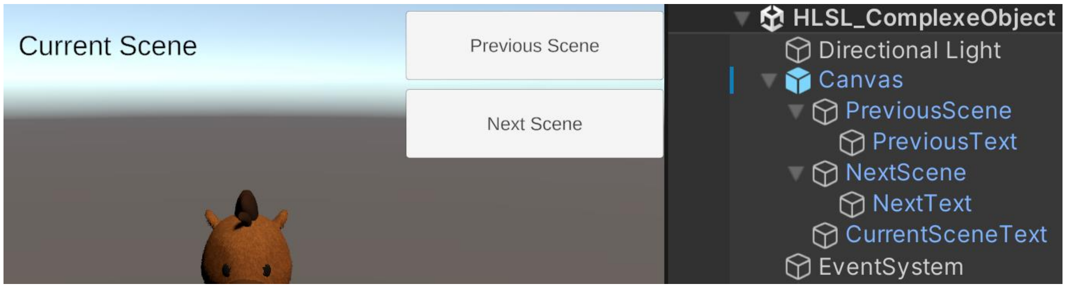
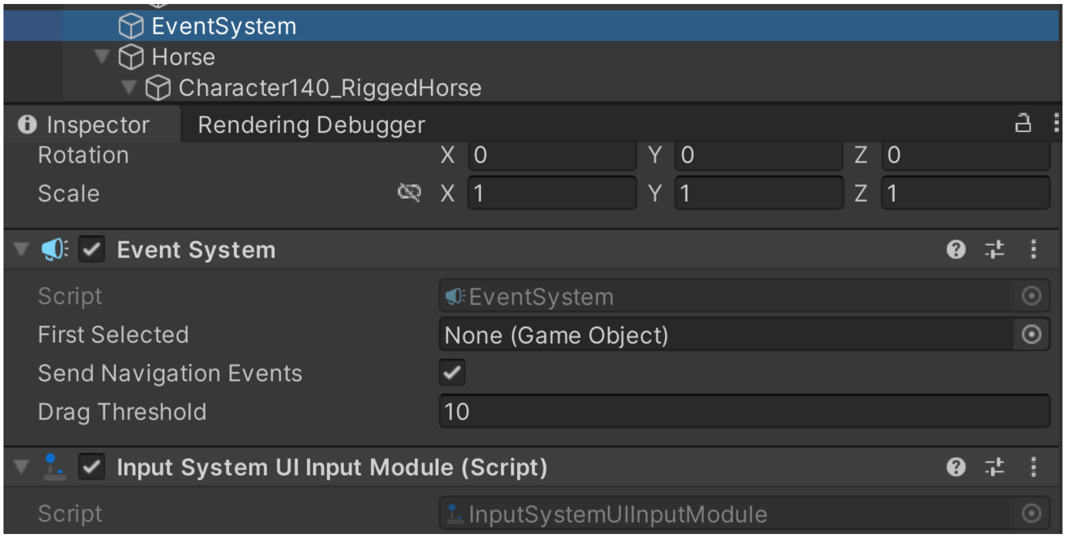


Figure 60 - Unity editor Hierarchy capture showing the Canvas prefab hierarchy. (Styles, 2024)

The user input was dealt with the Unity input system and an EventSystem component (Figure 60).



Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1

☑ Event System

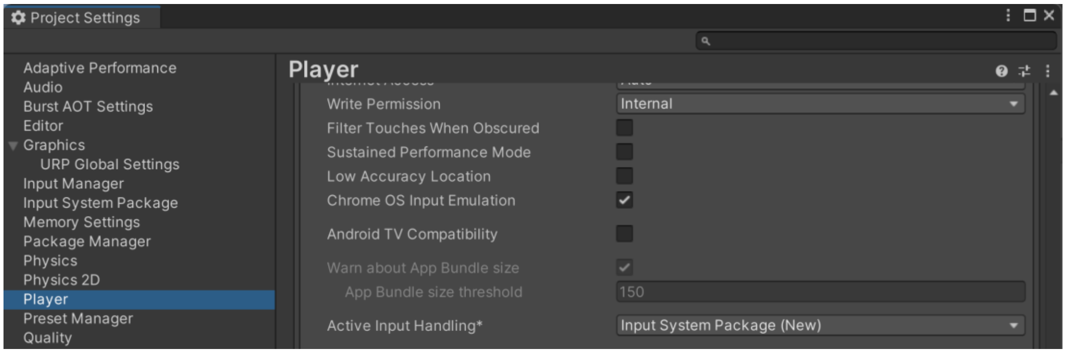
Script: EventSystem
First Selected: None (Game Object)
Send Navigation Events: ☑
Drag Threshold: 10

☑ Input System UI Input Module (Script)

Script: InputSystemUIInputModule

Figure 61 - Unity editor inspector window capture showing the EventSytem component. (Styles, 2024)

To deal with the scene management two C# scripts were created. One responsible for the loading of scenes in run-time (Styles, 2024) (see Appendix N) and the other for the creation of a scriptable object ("Unity - Manual: ScriptableObject", n.d.-f) that holds the current scene index across scenes (Styles, 2024) (see Appendix O).



Write Permission: Internal
Filter Touches When Obscured ☐
Sustained Performance Mode ☐
Low Accuracy Location ☐
Chrome OS Input Emulation ☑
Android TV Compatibility ☐
Warn about App Bundle size ☑
App Bundle size threshold: 150
Active Input Handling*: Input System Package (New)

Figure 62 - Unity editor capture of the Project Settings window showing the Active Input Handling parameter. (Styles, 2024)

The last modification that had to be made for cross-platform compatibility was to change the Active Input Handling in the Project Settings window under the Player menu to only tolerate the Input System Package (New) (Figure 61). Without that modification, the Android build does not compile.

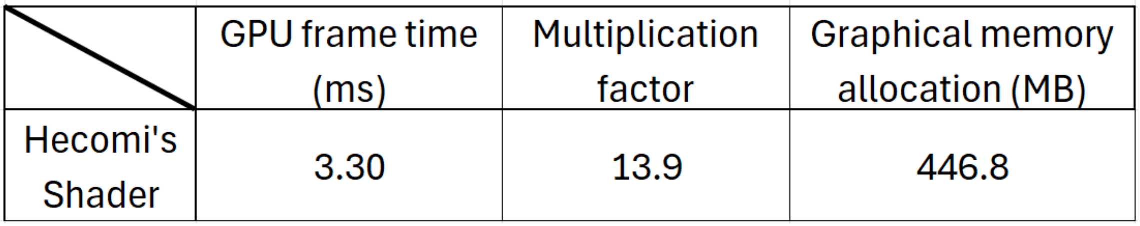
5.5 Conclusion

In this chapter the development of the project was exposed, describing the hardware and software in 5.1.1 and 5.1.2 respectively. The project primarily based on the visuals obtained from the project analysis of *Genshin Impact* in 3.2.1 was improved with the analysis of Hecomi’s code in 3.2.2.

The entire procedure leading to the final version of the shell texturing shader including the limitations and technical implementation was described in 5.2, 5.3 and 5.4.

6. Quantitative Analysis

The objective of this analysis is to use three versions of the shell-texturing shader and measure them in five different scenes, as described in section 5.3 with the tools described in section 4.2, the values of frame time, memory usage and vertex multiplication.



	GPU frame time (ms)	Multiplication factor	Graphical memory allocation (MB)
Hecomi's Shader	3.30	13.9	446.8

Figure 63 - Hecomi's project measurements, (Styles, 2024)

The limit values must not be over 14 times the original value for the vertex multiplication factor (Figure 63, middle) and 446.8 MB for the graphics total memory allocation (Figure 63, right) as measured in section 3.2.2.

Regarding the GPU frame time, the measurements, 3.30ms (Figure 63) taken on Hecomi’s project with the script developed and described in section 4.2.3, ended up being higher than the initial 2.16ms per GPU frame time exposed by the Unity render debugger. For that reason, the new value will be use as point of comparison for the following measurements.

The final measurement, taken on the Android mobile phone described in section 5.1.1, will give an indication on how the project runs on a lower capacity machine.

6.1 Measurement conditions

All the measurements are done the 24th of June 2024, in Lausanne, Switzerland.

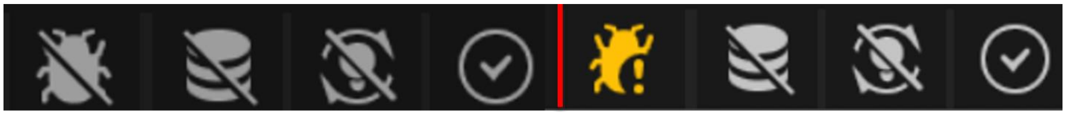


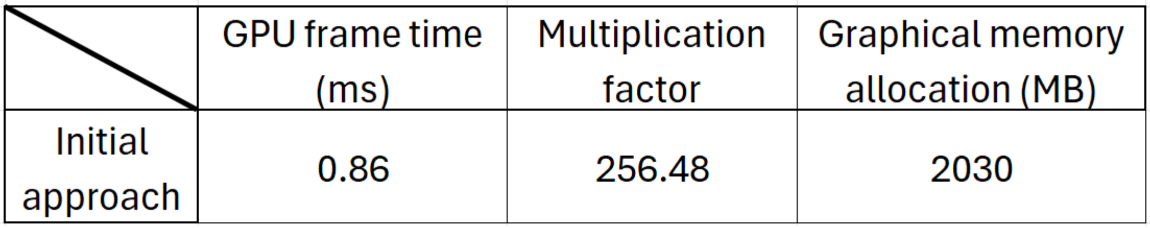
Figure 64 - Unity editor bottom left UI options showing the release (left) and debug (right) presets. (Styles, 2024)

The projects, as done with Hecomi's in section 3.2.2, are measured in the Unity editor in release mode (Figure 62, left) for the Acer ConceptD computer. The Android Galaxy measurements are made with a build of the project. All the measurements, although a script enabling fur movement was added, are taken in a static object state.

6.2 Project Measurements

6.2.1 Initial approach

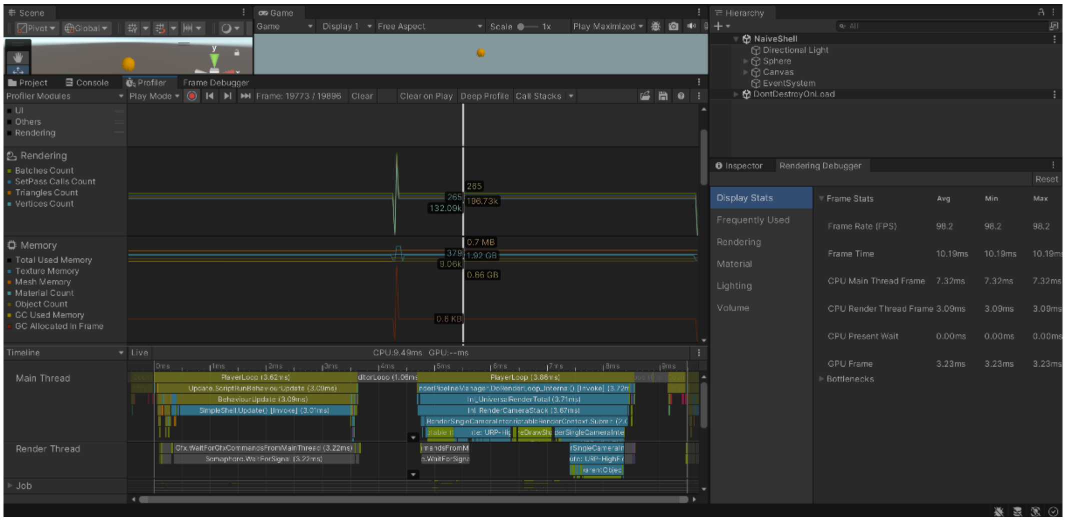
Measurements on the first version of the project give interesting results (Figure 65).



	GPU frame time (ms)	Multiplication factor	Graphical memory allocation (MB)
Initial approach	0.86	256.48	2030

Figure 65 - Measurement of the Initial approach project. (Styles, 2024)

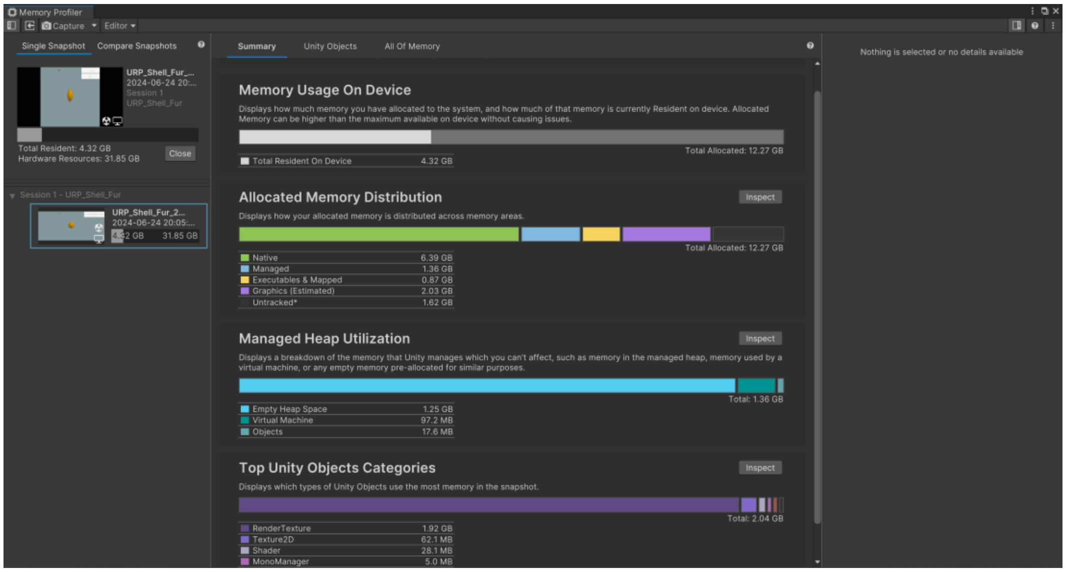
At an average of 3.23ms (Figure 66), the GPU frame time from the rendering debugger could be considered as over the limit compared to the initial 2.16ms. Considering the measurement of 0.86ms (Figure 65, left), the project happens to be more optimal than Hecomi's. This result could be explained by the four passes the rendering goes through on Hecomi's project.



Frame Stats	Avg	Min	Max
Frame Rate (FPS)	98.2	98.2	98.2
Frame Time	10.19ms	10.19ms	10.19m
CPU Main Thread Frame	7.32ms	7.32ms	7.32ms
CPU Render Thread Frame	3.09ms	3.09ms	3.09ms
CPU Present Wait	0.00ms	0.00ms	0.00ms
GPU Frame	3.23ms	3.23ms	3.23ms
Bottlenecks			

Figure 66 - Unity editor capture showing the measurement of the NaïveShell scene. (Styles, 2024)

The vertex multiplication factor criteria, with an output at 132'090 vertices for an original input of 515, exceeds the validation threshold. The memory usage given by the memory profiler, with a total of 2.03GB, also surpasses the targeted value.



Total Resident: 4.32 GB
Hardware Resources: 31.85 GB

4.32 GB 31.85 GB

Memory Usage On Device

Displays how much memory you have allocated to the system, and how much of that memory is currently Resident on device. Allocated Memory can be higher than the maximum available on device without causing issues.

Total Allocated: 12.27 GB

Total Resident On Device	4.32 GB

Allocated Memory Distribution

Displays how your allocated memory is distributed across memory areas.

Total Allocated: 12.27 GB

Native	6.39 GB
Managed	1.36 GB
Executables & Mapped	0.87 GB
Graphics (Estimated)	2.03 GB
Untracked*	1.62 GB

Managed Heap Utilization

Displays a breakdown of the memory that Unity manages which you can't affect, such as memory in the managed heap, memory used by a virtual machine, or any empty memory pre-allocated for similar purposes.

Total: 1.36 GB

Empty Heap Space	1.25 GB
Virtual Machine	97.2 MB
Objects	17.6 MB

Top Unity Objects Categories

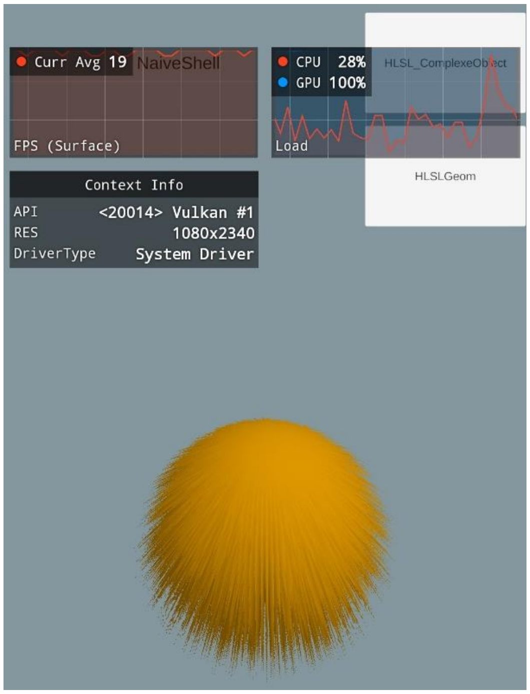
Displays which types of Unity Objects use the most memory in the snapshot.

Total: 2.04 GB

RenderTexture	1.92 GB
Texture2D	62.1 MB
Shader	28.1 MB
MonoManager	5.0 MB

Figure 67 - Unity editor capture showing the Memory Profiler on the NaïveShell scene. (Styles, 2024)

These two values, being over the limit of validation, clearly indicate a problem with the initial version of the project.



Curr Avg 19

CPU 28%
GPU 100%

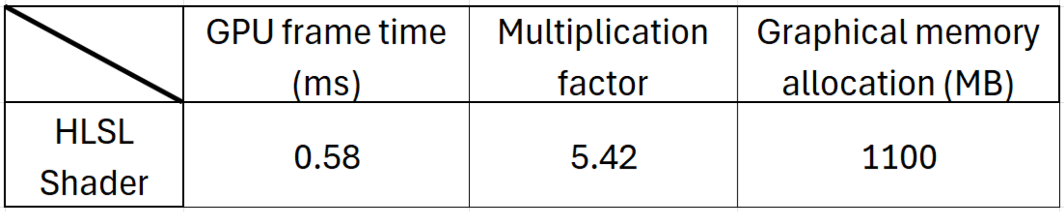
Context Info
API <20014> Vulkan #1
RES 1080x2340
DriverType System Driver

Figure 68 - Screen capture of the Android Galaxy A50 measurements for the NaïveShell scene. (Styles, 2024)

As expected, the values measured from the Android phone give a low frame rate at an average of 19 frames per second (Figure 68). In this state, the movement of the object is not fluid, and the flickering is noticeable.

6.2.2 HLSL

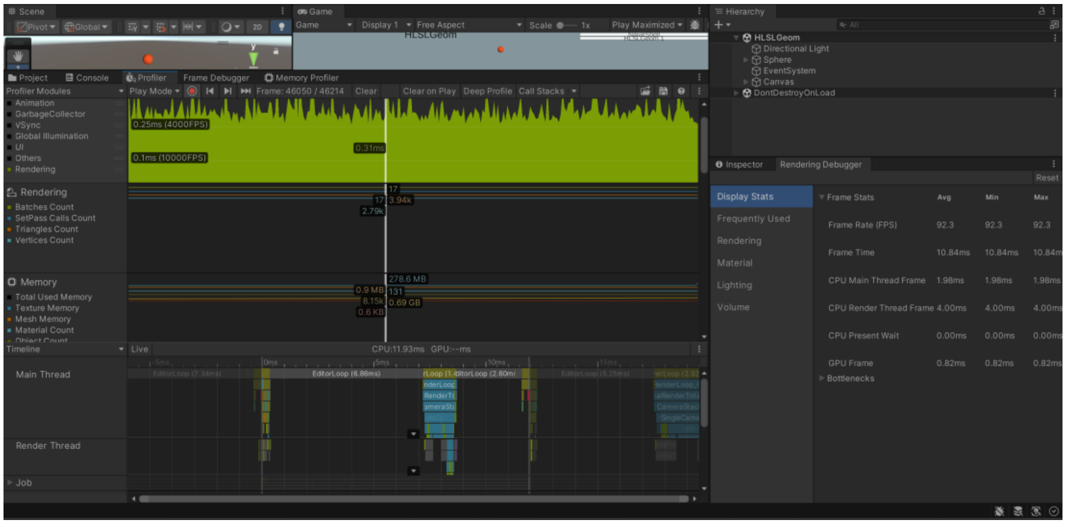
The second version of the project that uses only shader files and relies on HLSL instead of C# scripts for layer generation already gives better results than the previous version.



	GPU frame time (ms)	Multiplication factor	Graphical memory allocation (MB)
HLSL Shader	0.58	5.42	1100

Figure 69 - Measurements of the HLSL Shader project. (Styles, 2024)

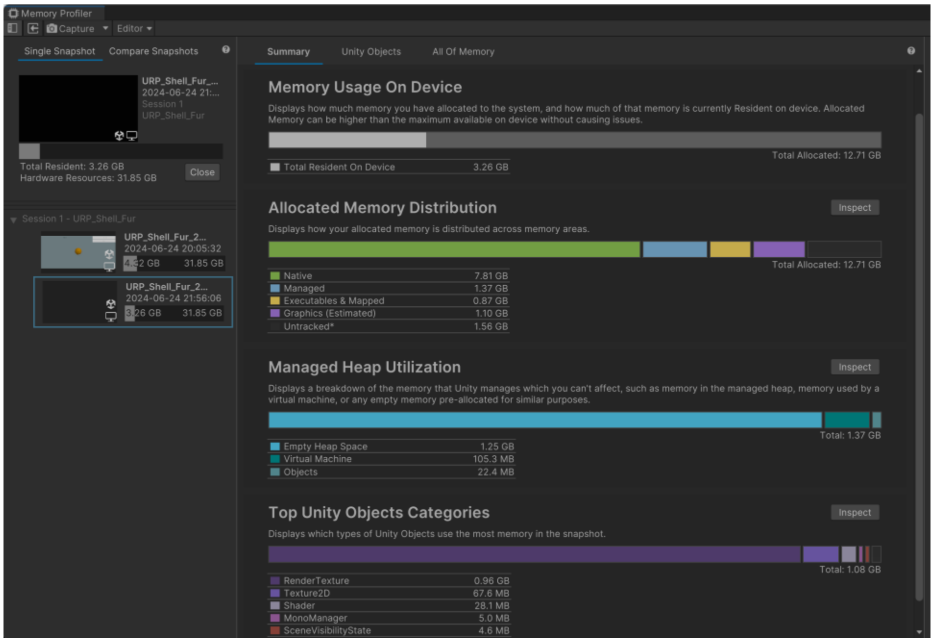
As expected, the use of a geometry shader not only reduces the time of a GPU frame but also impacts the multiplication factor and the memory allocation as exposed in figure 69.



Frame Stats	Avg	Min	Max
Frame Rate (FPS)	92.3	92.3	92.3
Frame Time	10.84ms	10.84ms	10.84m
CPU Main Thread Frame	1.98ms	1.98ms	1.98ms
CPU Render Thread Frame	4.00ms	4.00ms	4.00ms
CPU Present Wait	0.00ms	0.00ms	0.00ms
GPU Frame	0.82ms	0.82ms	0.82ms

Figure 70 - Unity editor capture showing the measurement of the HLSLGeom scene. (Styles, 2024)

The vertex value outputs 2,790 vertices for a factor of multiplication of 5.42, meaning that the validation criterion is reached. This change is mainly due to the difference between the use of a C# script layering meshes on top of each other and the use of the geometry shader to execute the same task.



URP_Shell_Fur_...
2024-06-24 21:...
Session 1
URP_Shell_Fur

Total Resident: 3.26 GB
Hardware Resources: 31.85 GB

Session 1 - URP_Shell_Fur

URP_Shell_Fur_2...
2024-06-24 20:05:32
4.32 GB 31.85 GB

URP_Shell_Fur_2...
2024-06-24 21:56:06
3.26 GB 31.85 GB

Memory Usage On Device

Displays how much memory you have allocated to the system, and how much of that memory is currently Resident on device. Allocated Memory can be higher than the maximum available on device without causing issues.

Total Allocated: 12.71 GB

Total Resident On Device	3.26 GB

Allocated Memory Distribution

Displays how your allocated memory is distributed across memory areas.

Total Allocated: 12.71 GB

Native	7.81 GB
Managed	1.37 GB
Executables & Mapped	0.87 GB
Graphics (Estimated)	1.10 GB
Untracked*	1.56 GB

Managed Heap Utilization

Displays a breakdown of the memory that Unity manages which you can't affect, such as memory in the managed heap, memory used by a virtual machine, or any empty memory pre-allocated for similar purposes.

Total: 1.37 GB

Empty Heap Space	1.25 GB
Virtual Machine	105.3 MB
Objects	22.4 MB

Top Unity Objects Categories

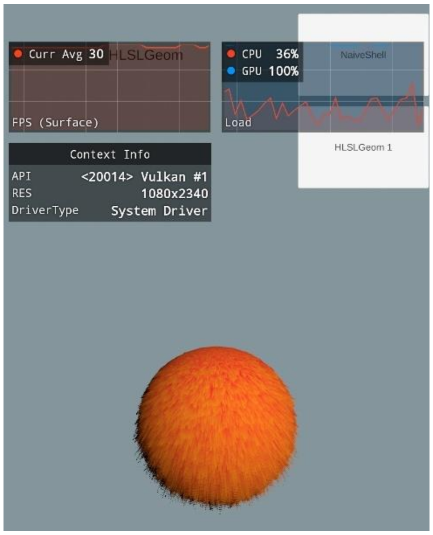
Displays which types of Unity Objects use the most memory in the snapshot.

Total: 1.08 GB

RenderTexture	0.96 GB
Texture2D	67.6 MB
Shader	28.1 MB
MonoManager	5.0 MB
SceneVisibilityState	4.6 MB

Figure 71 - Unity editor capture showing the Memory Profiler on the HLSLGeom scene. (Styles, 2024)

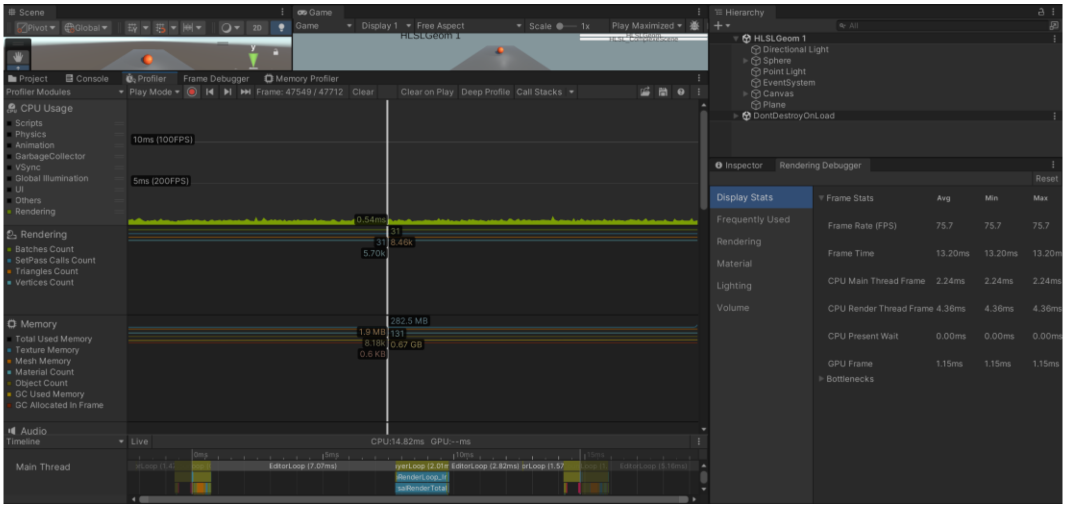
Regarding the graphics memory allocation, an improvement compared to the initial version is noticed but it still stays over the targeted value of 446.8 MB. This measurement can be explained by the reduction in the amount of meshes that are rendered.



API <20014> Vulkan #1
RES 1080x2340
DriverType System Driver

Figure 72 - Screen capture of the Android Galaxy A50 measurements for the HLSLGeom scene. (Styles, 2024)

With the Android mobile measurements, the project significantly improves in terms of fluidity with a current average at 30 frames per second and no noticeable flickering (Figure 72).



Frame Stats	Avg	Min	Max
Frame Rate (FPS)	75.7	75.7	75.7
Frame Time	13.20ms	13.20ms	13.20m
CPU Main Thread Frame	2.24ms	2.24ms	2.24ms
CPU Render Thread Frame	4.36ms	4.36ms	4.36ms
CPU Present Wait	0.00ms	0.00ms	0.00ms
GPU Frame	1.15ms	1.15ms	1.15ms

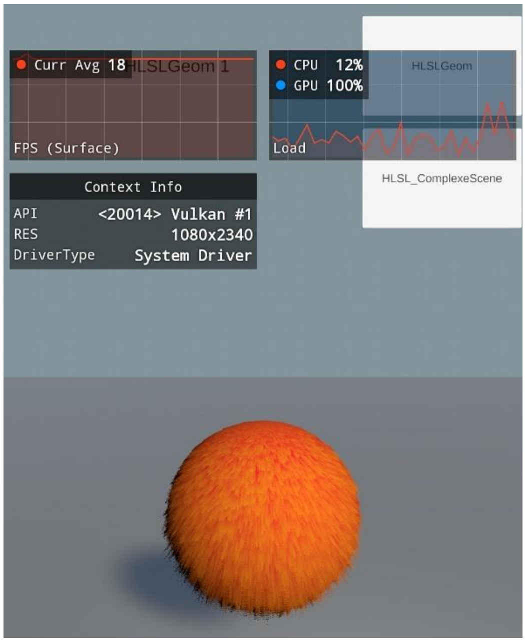
Figure 73 - Unity editor capture showing the measurement of the HLSLGeom1 scene. (Styles, 2024)

The following phenomenon happens in the third version when adding lighting to the scene and having shadows cast on a plane: the GPU frame time is not substantially impacted by the lighting, nor is the memory allocation. On the other hand, the multiplication factor is practically doubled.

	GPU frame time (ms)	Multiplication factor	Graphical memory allocation (MB)
HLSL Shader	0.58	5.42	1100
HLSL + Light	0.60	11.07	1100

Figure 74 - Measurement comparison on the HLSL Shader project with and without light

The hypothesis for this result is the duplication of passes, since the shader does twice its work when rendering shadows (See Appendix G).


Curr Avg 18
FPS (Surface)
CPU 12%
GPU 100%
Load

Context Info
API <20014> Vulkan #1
RES 1080x2340
DriverType System Driver

Figure 75 - Screen capture of the Android Galaxy A50 measurements for the HLSLGeom1 scene. (Styles, 2024)

The mobile measurements give an average frame rate of 18 which is comparable to the initial version of the shader without any lighting or shadow projection. One noticeable difference is that the additional lighting on the mobile build does not seem to be rendered (Figure 75).

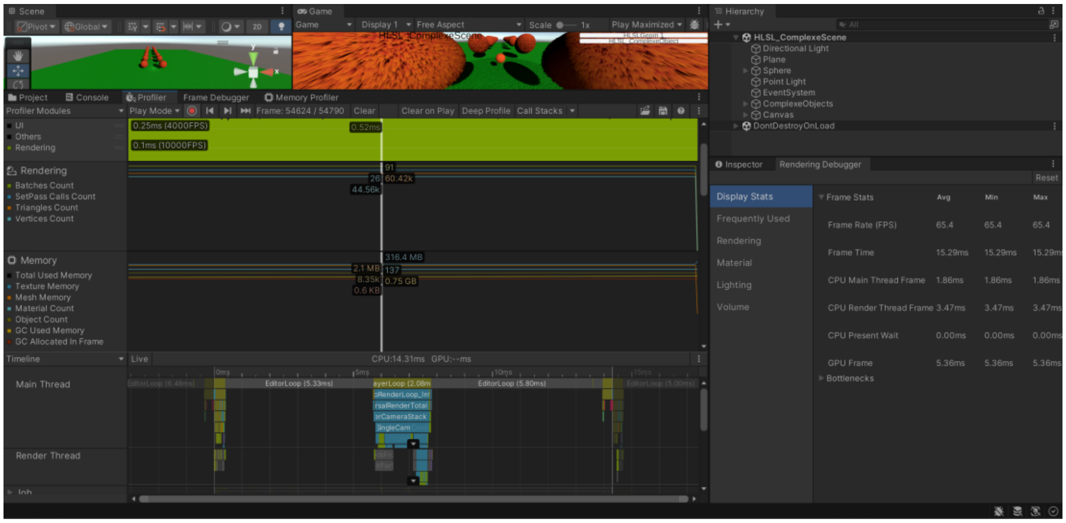

Frame Stats	Avg	Min	Max
Frame Rate (FPS)	65.4	65.4	65.4
Frame Time	15.29ms	15.29ms	15.29m
CPU Main Thread Frame	1.86ms	1.86ms	1.86ms
CPU Render Thread Frame	3.47ms	3.47ms	3.47ms
CPU Present Wait	0.00ms	0.00ms	0.00ms
GPU Frame	5.36ms	5.36ms	5.36ms

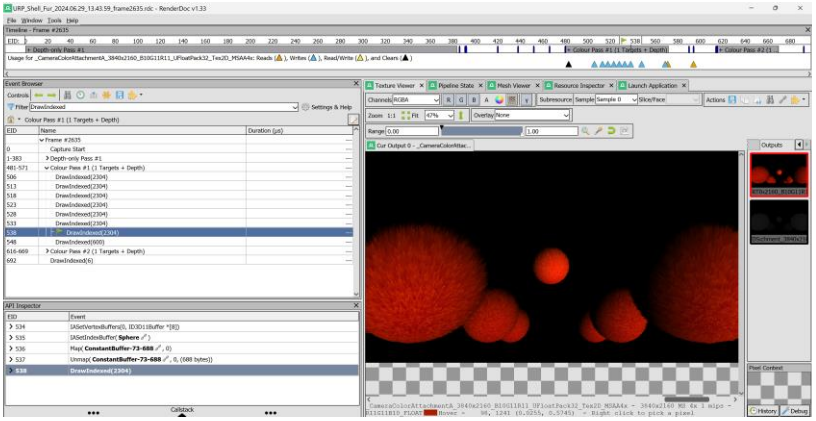
Figure 76 - Unity editor capture showing the measurement of the HLSL_ComplexeScene scene. (Styles, 2024)

For the fourth project, when applying the shader to an entire scene with 18 spheres and a 10 by 10 plane for a total count of 9391 vertices results in an output of 44560 vertices.

	GPU frame time (ms)	Multiplication factor	Graphical memory allocation (MB)
HLSL Scene	6.36	4.74	2140

Figure 77 - Measurement of the HLSL Shader in the complex scene project. (Styles, 2024)

The vertex multiplication factor at 4.47, is a better result than in the HLSL single sphere scene (Figure 77, middle).


URP_Shell_Fur_2024.06.29_13.43.59_frame2635.rdc - RenderDoc v1.33

Usage for _CameraColorAttachmentA_3840x2160_B10G11R11_UFloatPack32_Tex2D_MSAA4x: Reads (△), Writes (△), Read/Write (△), and Clears (▲)

Filter: DrawIndexed

Colour Pass #1 (1 Targets + Depth)

EID	Name	Duration (µs)
	Frame #2635	—
0	Capture Start	—
1-383	Depth-only Pass #1	—
481-571	Colour Pass #1 (1 Targets + Depth)	—
506	DrawIndexed(2304)	—
513	DrawIndexed(2304)	—
518	DrawIndexed(2304)	—
523	DrawIndexed(2304)	—
528	DrawIndexed(2304)	—
533	DrawIndexed(2304)	—
538	DrawIndexed(2304)	—
548	DrawIndexed(600)	—
616-669	Colour Pass #2 (1 Targets + Depth)	—
692	DrawIndexed(6)	—

EID	Event
534	IASetVertexBuffers(0, ID3D11Buffer *[8])
535	IASetIndexBuffer(**Sphere**)
536	Map(**ConstantBuffer-73-688** , 0)
537	Unmap(**ConstantBuffer-73-688** , 0, (688 bytes))
538	**DrawIndexed(2304)**

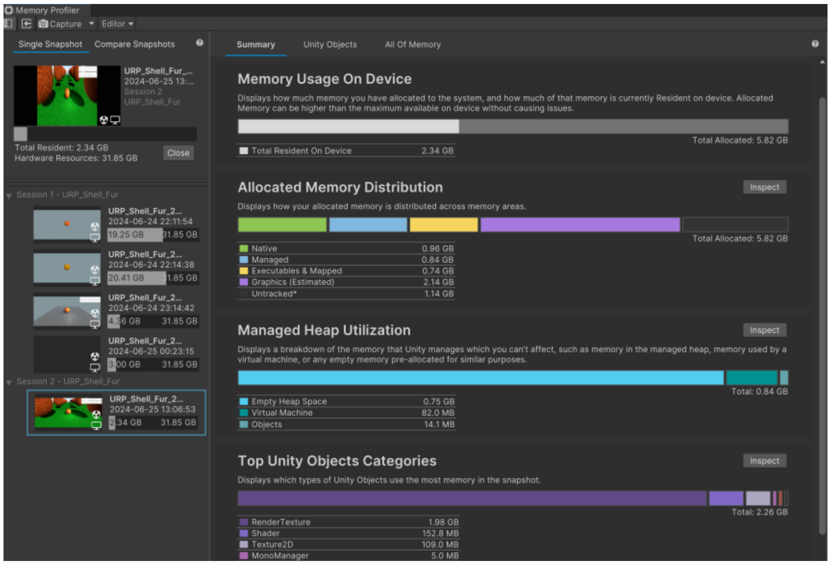
Channels: RGBA; Sample: Sample 0; Zoom: 1:1, Fit 47%; Overlay: None; Range: 0.00 – 1.00

_CameraColorAttachmentA_3840x2160_B10G11R11_UFloatPack32_Tex2D_MSAA4x - 3840x2160 MS 4x 1 mips - R11G11B10_FLOAT Hover = 98, 1241 (0.0255, 0.5745) - Right click to pick a pixel

Figure 78 - Capture of RenderDoc usage on the HLSL_ComplexeScene scene. (Styles, 2024)

RenderDoc gives information about the rendering process. The value is lower due to frustrum culling and other Unity built-in graphical optimisation techniques.

The value to acknowledge is the GPU frame that increases up to 6.36ms (Figure 77) or 92.59% (Figure 85) more than the targeted value. This result indicates the influence of a broad use of the shader and how it can influence fluidity when used on many objects.



URP_Shell_Fur_...
2024-06-25 13:...
Session 2
URP_Shell_Fur

Total Resident: 2.34 GB
Hardware Resources: 31.85 GB

Session 1 - URP_Shell_Fur

URP_Shell_Fur_2...
2024-06-24 22:11:54
19.25 GB 31.85 GB

URP_Shell_Fur_2...
2024-06-24 22:14:38
20.41 GB 31.85 GB

URP_Shell_Fur_2...
2024-06-24 23:14:42
4.36 GB 31.85 GB

URP_Shell_Fur_2...
2024-06-25 00:23:15
3.00 GB 31.85 GB

Session 2 - URP_Shell_Fur

URP_Shell_Fur_2...
2024-06-25 13:06:53
2.34 GB 31.85 GB

Memory Usage On Device

Displays how much memory you have allocated to the system, and how much of that memory is currently Resident on device. Allocated Memory can be higher than the maximum available on device without causing issues.

Total Allocated: 5.82 GB

Total Resident On Device	2.34 GB

Allocated Memory Distribution

Displays how your allocated memory is distributed across memory areas.

Total Allocated: 5.82 GB

🟩 Native	0.96 GB
🟦 Managed	0.84 GB
🟨 Executables & Mapped	0.74 GB
🟪 Graphics (Estimated)	2.14 GB
Untracked*	1.14 GB

Managed Heap Utilization

Displays a breakdown of the memory that Unity manages which you can't affect, such as memory in the managed heap, memory used by a virtual machine, or any empty memory pre-allocated for similar purposes.

Total: 0.84 GB

Empty Heap Space	0.75 GB
Virtual Machine	82.0 MB
Objects	14.1 MB

Top Unity Objects Categories

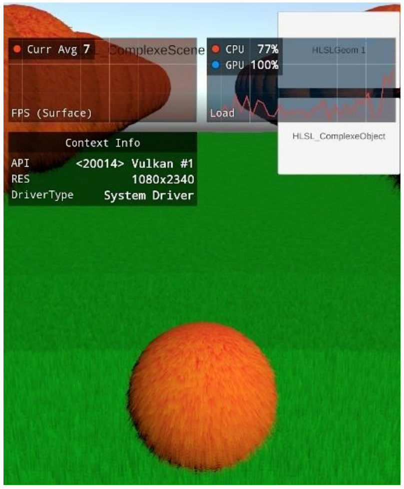
Displays which types of Unity Objects use the most memory in the snapshot.

Total: 2.26 GB

RenderTexture	1.98 GB
Shader	152.8 MB
Texture2D	109.0 MB
MonoManager	5.0 MB

Figure 79 - Unity editor capture showing the Memory Profiler on the HLSL_ComplexeScene scene. (Styles, 2024)

Regarding the memory allocation value, it is interesting to consider that for 18 times more objects, lights and shadows, the HLSL shader is at the same level as the first one.



Curr Avg 7

CPU 77%
GPU 100%

Context Info
API <20014> Vulkan #1
RES 1080x2340
DriverType System Driver

Figure 80 - Screen capture of the Android Galaxy A50 measurements for the HLSL_ComplexeScene scene. (Styles, 2024)

The mobile measurements, giving an average of seven frames per second with 77% of the CPU load, seem to indicate that the shader is not optimal when widely used. The results in terms of fluidity and flickering are dreadful despite the graphical optimisation techniques given by the Unity engine.

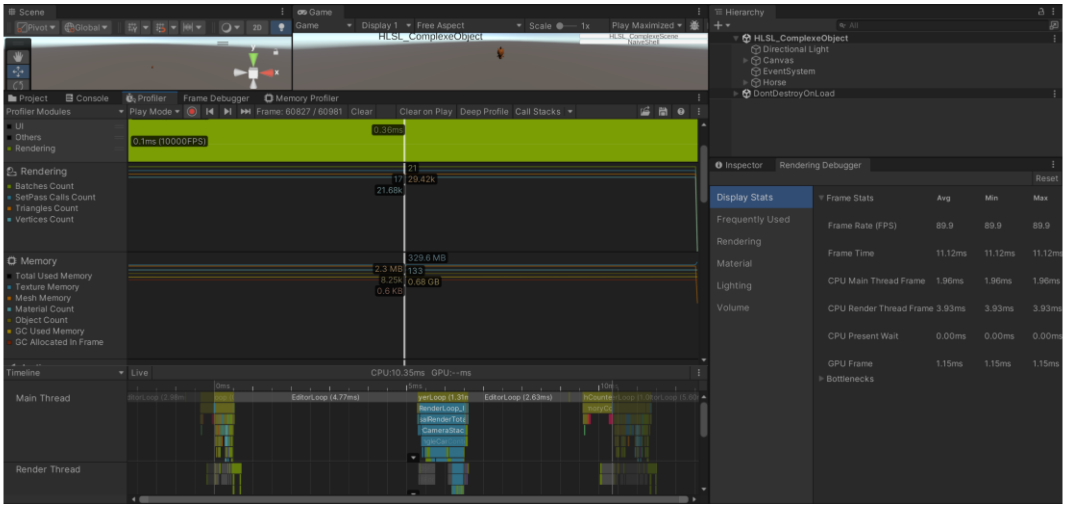
6.2.3 HLSL Complex object

The results gathered on the fifth project are probably the most representative of a normal use of the shader since it is targeted on an animated character instead of an entire scene.

	GPU frame time (ms)	Multiplication factor	Graphical memory allocation (MB)
HLSL Complex	0.80	6.63	456.2

Figure 81 - Measurements of the HLSL Complex object scene. (Styles, 2024)

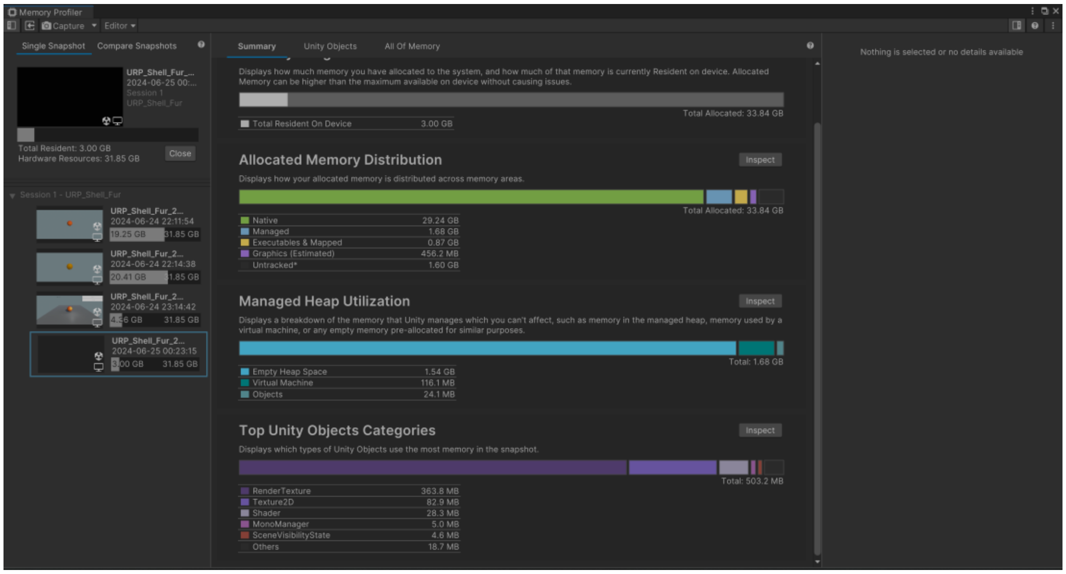
The measurements for the GPU frame time give 1.15ms (Figure 82), identical to the lighted version of the HLSL shader, when measured on the rendering debugger. The difference lies in the measurements gathered with the DataExtractor script (See Appendix P) where it appears to be 33% slower than the lighted version (Figure 81, left).



Frame Stats	Avg	Min	Max
Frame Rate (FPS)	89.9	89.9	89.9
Frame Time	11.12ms	11.12ms	11.12ms
CPU Main Thread Frame	1.96ms	1.96ms	1.96ms
CPU Render Thread Frame	3.93ms	3.93ms	3.93ms
CPU Present Wait	0.00ms	0.00ms	0.00ms
GPU Frame	1.15ms	1.15ms	1.15ms

Figure 82 - Unity editor capture showing the measurement of the HLSL_ComplexeObject scene. (Styles, 2024)

This difference can be explained by the amount of surface the shader covers going from 5700 vertices for the sphere to 21680 for the 3D horse model. With an input of 1862 vertices for the head, 1407 for the body and 3269 in total, the vertex multiplication factor is at 47% of the targeted value which meets the validation criterion.



URP_Shell_Fur_...
2024-06-25 00:...
Session 1
URP_Shell_Fur

Total Resident: 3.00 GB
Hardware Resources: 31.85 GB

Session 1 - URP_Shell_Fur

URP_Shell_Fur_2...
2024-06-24 22:11:54
19.25 GB 31.85 GB

URP_Shell_Fur_2...
2024-06-24 22:14:38
20.41 GB 31.85 GB

URP_Shell_Fur_2...
2024-06-24 23:14:42
4.36 GB 31.85 GB

URP_Shell_Fur_2...
2024-06-25 00:23:15
3.00 GB 31.85 GB

Displays how much memory you have allocated to the system, and how much of that memory is currently Resident on device. Allocated Memory can be higher than the maximum available on device without causing issues.

Total Allocated: 33.84 GB

Total Resident On Device	3.00 GB

Allocated Memory Distribution

Displays how your allocated memory is distributed across memory areas.

Total Allocated: 33.84 GB

Native	29.24 GB
Managed	1.68 GB
Executables & Mapped	0.87 GB
Graphics (Estimated)	456.2 MB
Untracked*	1.60 GB

Managed Heap Utilization

Displays a breakdown of the memory that Unity manages which you can't affect, such as memory in the managed heap, memory used by a virtual machine, or any empty memory pre-allocated for similar purposes.

Total: 1.68 GB

Empty Heap Space	1.54 GB
Virtual Machine	116.1 MB
Objects	24.1 MB

Top Unity Objects Categories

Displays which types of Unity Objects use the most memory in the snapshot.

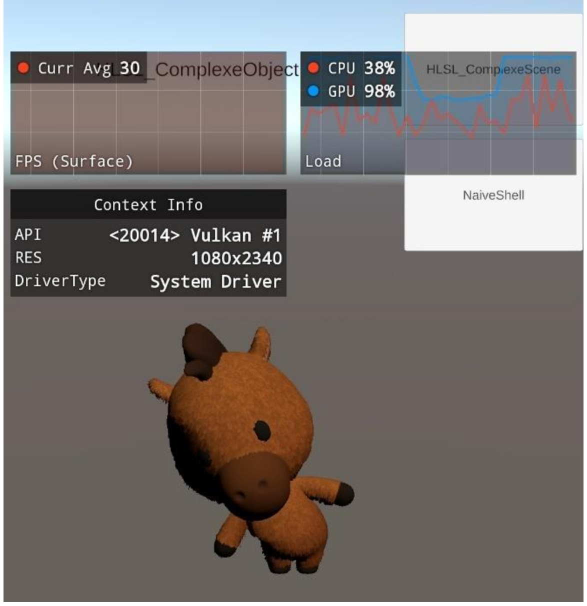
Total: 503.2 MB

RenderTexture	363.8 MB
Texture2D	82.9 MB
Shader	28.3 MB
MonoManager	5.0 MB
SceneVisibilityState	4.6 MB
Others	18.7 MB

Nothing is selected or no details available

Figure 83 - Unity editor capture showing the Memory Profiler on the HLSL_ComplexeObject scene. (Styles, 2024)

The memory allocation, with a value of 456.2MB, represents an increase of 2% compared to the original target value which is a considerable increase compared to the two previous shader versions.

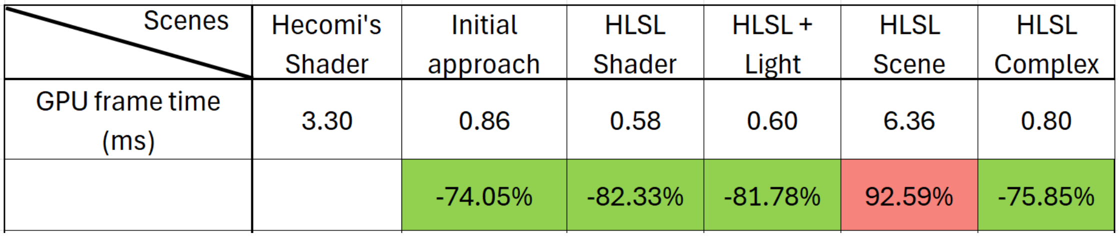


Context Info
API <20014> Vulkan #1
RES 1080x2340
DriverType System Driver

Figure 84 - Screen capture of the Android Galaxy A50 measurements for the HLSL_ComplexeObject scene. (Styles, 2024)

The mobile measurements seem to be as optimal as the first HLSL version even with an increase in the number of vertices and the use of an animator system.

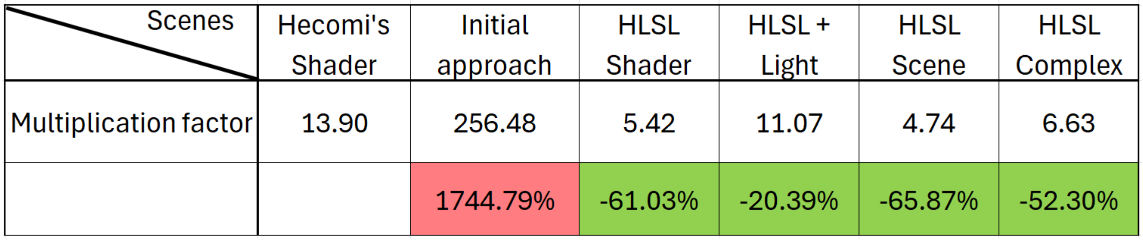
6.3 Project Analysis



Scenes	Hecomi's Shader	Initial approach	HLSL Shader	HLSL + Light	HLSL Scene	HLSL Complex
GPU frame time (ms)	3.30	0.86	0.58	0.60	6.36	0.80
		-74.05%	-82.33%	-81.78%	92.59%	-75.85%

Figure 85 - Measurements of the GPU frame time and comparison on all projects. (Styles, 2024)

Regarding the GPU frame time, measurements indicate its relationship with shader passes and shader usage. Hecomi’s shader, having four passes, appears to be less optimal than any other shader using two render passes, as long as the shader is restrained in its usage (Figure 85).

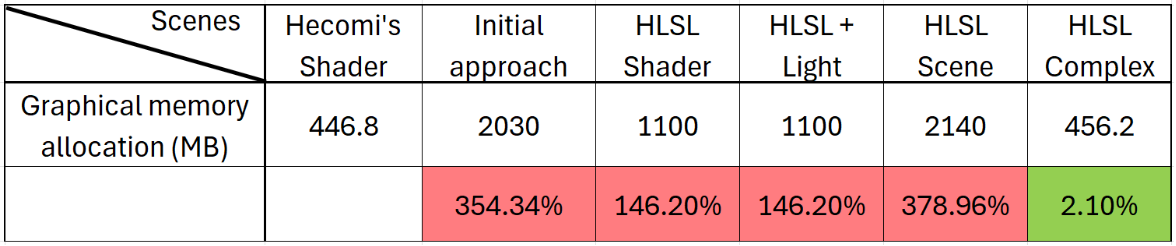


Scenes	Hecomi's Shader	Initial approach	HLSL Shader	HLSL + Light	HLSL Scene	HLSL Complex
Multiplication factor	13.90	256.48	5.42	11.07	4.74	6.63
		1744.79%	-61.03%	-20.39%	-65.87%	-52.30%

Figure 86 - Measurements of the multiplication factor and comparison on all projects. (Styles, 2024)

Concerning the multiplication factor, the influence of layering seems to be correlated with the way the shader processes the initial mesh. In the case of the initial approach (Figure 86) the addition of meshes, without any specific treatment after that process, gives unacceptable results in terms of optimisation. Projects relying on geometry shaders give better results despite the number of objects on which the shader is used (Figure 86).

An observation to consider is the impact of the number of passes used to render fur. Since the shader uses two passes and the multiplication factor is doubled, the measurements of the HLSL lighted scene (Figure 86) indicate a direct correlation between them.

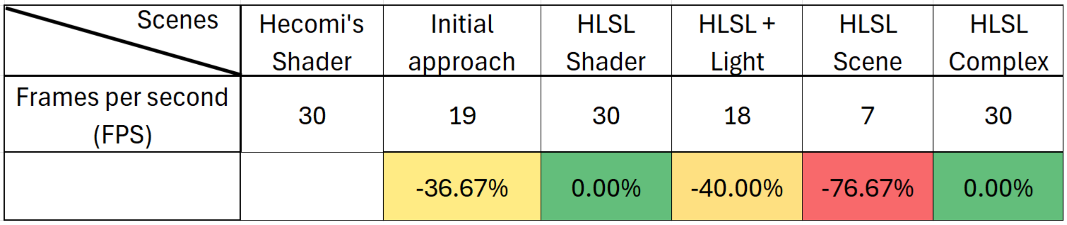


Scenes	Hecomi's Shader	Initial approach	HLSL Shader	HLSL + Light	HLSL Scene	HLSL Complex
Graphical memory allocation (MB)	446.8	2030	1100	1100	2140	456.2
		354.34%	146.20%	146.20%	378.96%	2.10%

Figure 87 - Measurements of the graphical memory allocation and comparison on all projects. (Styles, 2024)

Regarding the graphical memory allocation, shaders relying on the use of textures for the fur’s colour are less memory consuming than those using variables, despite the number of vertices contained in the object on which the shader is used. These measurements are counter-intuitive since the HLSL Complex shader does not remove the use of colour parameters but adds 2D textures multiplied by them as exposed in section 5.4.3.

One of the possible reasons behind these results could be related to texture compression or another optimisation done by the Unity engine (Technologies, n.d.-a & -b). Unfortunately, due to a lack of time, the reason behind this phenomenon couldn’t be demystified.



Scenes	Hecomi's Shader	Initial approach	HLSL Shader	HLSL + Light	HLSL Scene	HLSL Complex
Frames per second (FPS)	30	19	30	18	7	30
		-36.67%	0.00%	-40.00%	-76.67%	0.00%

Figure 88 - Measurements of the frames per second and comparison on all projects. (Styles, 2024)

In terms of frames per second, although the values in figure 88 cannot be considered as appropriate due to factors such as VSync, it is still relevant to acknowledge the impact of elements such as lighting and shadows and the repercussion of wide usage of the shader despite the graphical optimisations given by Unity.

6.4 Conclusion

In this chapter, some of the tools discussed in section 4.2 were employed to measure the metrics outlined in section 4.1 across all projects presented in section 5.3.

The collected data reveals that versions using HLSL shaders demonstrate superior performance compared to those relying on C# scripts for GPU data transfer. However, even with optimised HLSL shaders, complex scenes can still encounter performance bottlenecks due to the high output of vertices.

The initial approach project, while maintaining fluidity on portable computers, proves itself impractical for complex scenes or lower-end hardware usage. HLSL shaders significantly outperform the initial approach, particularly in scenarios involving minimal use of fur shaders and simple lighting setups.

The latest shader version reflects insights and improvements from a professional graphics programmer, notably enhancing performance, particularly in terms of memory management.

In terms of validation criteria, HLSL versions effectively manage GPU frame time when shader usage is limited to essential elements. The vertex multiplication factor criterion is consistent across all HLSL cases, likely due to Hecomi employing multiple passes for depth, light, and shadow calculations, all contributing to the layering effect produced by the geometry shader. While achieving visually similar results, the question of the purpose behind using so many passes could be raised.

7. Conclusion and further research

This study aimed to identify the cause of the visual problem encountered when using shell texturing in real-time environments and the possible improvements leading to its resolution.

The first chapter highlighted two examples of the use of this method in AAA games. It exposed the main differences between real-time and pre-calculated experiences, as well as the project's guidelines, its development, and the anticipation of certain limitations.

Based on this, the second chapter refined the concepts of fur, realistic rendering, and understanding what Unity’s engine is. It also presented different methods for rendering fur and four examples of use of shell texturing in the video game industry over the past 20 years. A rough plan of the project was also presented.

Interviews with an experienced professional, David Sena, and ChatGPT, along with their analyses, validated the interest in the topic and highlighted the importance of resource gathering when undertaking such a project. Some resources, not applicable to the use of Unity, were set aside but their analysis provided insights into the current state of methodologies for creating fur in real-time environments.

The empirical methodology employed led to the use of HLSL shaders, which were used to improve the initial project and allowed for adaptations meeting the validation criteria established in the fourth chapter, dealing with metrics, their threshold values, acceptance criteria, and the measurement tools for gathering these values. The resolution of the problem initially presented proved to be much more complex than initially anticipated and allowed a better understanding of the complexity of the role of a graphics programmer.

The quantitative chapter allowed for tests to be conducted under predefined conditions on all versions of the project. This quantitative study revealed that optimisation criteria such as GPU frame time and memory usage are important factors to consider when evaluating a graphics project. In contrast, the multiplication factor criterion is less recognised as relevant to the measurement itself due to technologies provided by the Unity engine, such as frustum culling, but it highlights the impact of the final vertex count in a scene.

Finally, the results of the project highlight several key aspects: It is possible to render fur in real-time, and multiple methodologies are available to do so regardless of the engine used. Designing a shader that addresses the issue of the camera’s perspective requires a much deeper understanding of the domain or a resolution using another method, which might be less optimal than shell texturing. Additionally, it is pertinent to emphasise the importance of the context for which these shaders are developed, as each project needs to be created with specific targets in mind for its final use, whether in terms of platform, capabilities, or visual rendering quality.

These elements are, however, subject to temporal and technological factors due to constant improvements in both hardware and software.

While the findings of this work provide insights and guidance on the subject, it is important to consider limiting factors such as a lack of professional experience in the field, the time taken to study graphic languages, and the laborious iterative process.

During the study, several points of interest emerged, such as the use of other game engines, the development of a custom rendering engine to facilitate the implementation and measurement of shaders, and the exploration of other methods that could provide development pathways for another Bachelor's student interested in the subject or for further research in the context of a Master's thesis.

8. References

8.1 Articles

Andersen, T.G., Falster, V., Frisvad, J.R. and Christensen, N.J., (2016). Hybrid fur rendering: combining volumetric fur with explicit hair strands. *The Visual Computer*, 32(6–8), pp.739–749. Available at: https://doi.org/10.1007/s00371-016-1252-x [Accessed 16 June 2024].

DoVale, E. and Motion Picture Science, Rochester Institute of Technology (RIT), (2021). High Frame Rate Psychophysics: Experimentation to Determine a JND for Frame Rate. *Motion Picture Science, Rochester Institute of Technology (RIT)*. Available at: https://s3.cad.rit.edu/cadgallery_production/storage/media/uploads/faculty-s-projects/1918/documents/177/framerate-visibility-thesis.pdf [Accessed 16 June 2024].

Döllner, J., Hinkenjann, A. and Wiemker, R., (2006). Shell-texturing for real-time fur rendering. *Proceedings of the 2006 ACM SIGGRAPH Symposium on Videogames*, pp.97-104. Available at: https://doi.org/10.1145/1174429.1174477.

Kajiya, J.T. and Lischinski, D., (1993). Anisotropic reflection models. *IEEE Computer Graphics and Applications*, 13(4), pp.25-34. Available at: https://doi.org/10.1109/38.252558.

Kajiya, T. and von Herzen, B., (1984). Ray tracing volume densities. *ACM SIGGRAPH Computer Graphics*, 18(3), pp.165-174. Available at: https://doi.org/10.1145/964965.808594.

Lengyel, J., Praun, E., Finkelstein, A. and Hoppe, H., (2001). Real-time fur over arbitrary surfaces. *Symposium on Interactive 3D Graphics (I3D) 2001*, pp.227-232. Available at: https://hhoppe.com/fur.pdf.

Marschner, S.R., Cornell University, Jensen, H.W., University of California—San Diego, Cammarano, M., Stanford University, Worley, S., Worley Laboratories, Hanrahan, P. and Stanford University, (2003). Light Scattering from Human Hair Fibers. Available at: http://www.graphics.stanford.edu/papers/hair/hair-sg03final.pdf [Accessed 16 June 2024].

Rapp, M., (2014). *Real-Time hair rendering* [Master Thesis, Stuttgart Media University]. In Stuttgart Media University, Computer Science and Media M.Sc.). Available at: http://markusrapp.de/wordpress/wp-content/uploads/hair/MarkusRapp-MasterThesis-RealTimeHairRendering.pdf [Accessed 16 June 2024].

Tariq, S., (2007). Fur (using shells and fins). *NVIDIA*. Available at: https://developer.download.nvidia.com/SDK/10/direct3d/Source/Fur/doc/FurShellsAndFins.pdf [Accessed 3 June 2024].

White, M., (2008). Real-Time Optimally Adapting Meshes: Terrain Visualization in Games. *International Journal of Computer Games Technology*, 2008, pp.1–7. Available at: https://doi.org/10.1155/2008/753584.

8.2 Bibliography

Akenine-Möller, T., Haines, E., Hoffman, N., Pesce, A., Iwanicki, M. and Hillaire, S., (2018). *Real-Time rendering*. In A K Peters/CRC Press eBooks. Available at: https://doi.org/10.1201/b22086.

NVIDIA Corporation, (2004). *GPU gems: Programming techniques, tips, and tricks for real-time graphics*. Addison-Wesley.

Shirley, P., Ashikhmin, M., Marschner, S. and Reinhard, E., (2021). *Fundamentals of Computer Graphics (4th ed.)*. A K Peters/CRC Press.

8.3 Filmography

Acerola, (2023). How are games rendering fur? [Video] YouTube, 30 October. Available at: https://www.youtube.com/watch?v=9dr-tRQzij4.

CG Cookie - Unity Training, (n.d.). Creating Polygon Hair for Game Characters [Video] YouTube. Available at: https://www.youtube.com/watch?v=6Wi4-fdeYyM.

Froyok, (2012). Shadow of the Colossus - Fur breakdown [Video] YouTube, 7 October. Available at: https://www.youtube.com/watch?v=taIuZAGOFTo.

GDC, (2022). Procedural Grass in “Ghost of Tsushima” [Video] YouTube, 14 July. Available at: https://www.youtube.com/watch?v=Ibe1JBF5i5Y.

SamuelStyles, (2024). MP Fur HorseDemo [Video] YouTube, 21 June. Available at: https://www.youtube.com/shorts/_C4lptNMAiU.

SimonDev, (2023). How do Major Video Games Render Grass? [Video] YouTube, 6 November. Available at: https://www.youtube.com/watch?v=bp7REZBV4P4.

8.4 Webography

80.lv. (2018). CGMA Student Project: Hair for Games. [online] Available at: https://80.lv/articles/005cg-cgma-student-project-hair-for-games/ [Accessed 30 Jun. 2024].

80.lv. (2021). Realistic Dog Portrait: Experimenting with Real-Time Fur. [online] Available at: https://80.lv/articles/realistic-dog-portrait-experimenting-with-real-time-fur/ [Accessed 30 Jun. 2024].

[ANSWERED], (n.d.). What game engine does Genshin Impact use? Available at: https://www.dragonflydb.io/faq/genshin-impact-game-engine [Accessed 6 June 2024].

Adobe, (n.d.). What is ray tracing & what does it do? Available at: https://www.adobe.com/products/substance3d/discover/what-is-ray-tracing.html [Accessed 3 June 2024].

Afanasev, G. (2018). Gen Afanasev | Graphics Programmer blog: Fancy Shaders - Part 2: Shell Rendering. [online] Gen Afanasev | Graphics Programmer blog. Available at: https://gen-graphics.blogspot.com/2018/04/fancy-shaders-shell-rendering.html [Accessed 30 Jun. 2024].

Artheroes, (n.d.). Realistic hair and grooming for game characters. Available at: https://artheroes.co/hair-cards [Accessed 6 June 2024].

Beautypi, (n.d.). Shadertoy. Available at: https://www.shadertoy.com/view/llGSzw [Accessed 16 June 2024].

Bluebird International, (n.d.). What is ray tracing? The future of graphics rendering. Available at: https://bluebirdinternational.com/what-is-ray-tracing/ [Accessed 3 June 2024].

Chaos, (n.d.). Photorealistic rendering software for artists & designers. Available at: https://www.chaos.com/photorealistic-rendering [Accessed 3 June 2024].

Creswell, J. (2021). Dark Souls: What Can You Make With the Soul of Sif? [online] CBR. Available at: https://www.cbr.com/dark-souls-sif-soul-guide/ [Accessed 30 Jun. 2024].

DARK SOULSTM III on Steam, (n.d.). Available at: https://store.steampowered.com/app/374320/DARK_SOULS_III/ [Accessed 6 June 2024].

Danielson, M. and Yonezawa, B., (2024). Genshin Impact: System requirements for PC & mobile. *ScreenRant*, 21 May. Available at: https://screenrant.com/genshin-impact-system-requirements-pc-mobile-minimum-configuration/#system-requirements-specs-on-android [Accessed 5 June 2024].

Donato, J., (2019). *Viva Piñata – Game review*. *Red Ring Circus*, 28 February. Available at: https://redringcircus.com/2008/04/16/viva-pinata-game-review-updated-2019/ [Accessed 6 June 2024].

Edgardlop (2013). Hair and fur Render Time on characters. [online] Available at: https://www.reddit.com/r/disney/comments/1t6q37/hair_and_fur_render_time_on_characters/ [Accessed 30 Jun. 2024].

For The Win. (2022). Genshin Impact characters: ages, heights, birthdays, and bios. [online] Available at: https://ftw.usatoday.com/lists/genshin-impact-characters-age-height-birthday.

Froyok, (n.d.). [Breakdown] Shadow of the Colossus (PAL - PS2) | Froyok - Léna Piquet. Available at: https://www.froyok.fr/blog/2012-10-breakdown-shadow-of-the-colossus-pal-ps2/ [Accessed 6 June 2024].

Hoyoverse.com. (2024). Genshin Impact – Step Into a Vast Magical World of Adventure. [online] Available at: https://genshin.hoyoverse.com/en/news/detail/103720 [Accessed 30 Jun. 2024].

GDCVault, (n.d.). Advanced Graphics Summit: Procedural grass in "Ghost of Tsushima." Available at: https://www.gdcvault.com/play/1027033/Advanced-Graphics-Summit-Procedural-Grass [Accessed 16 June 2024].

GiM. (n.d.). An Introduction to Shell Based Fur Technique. [online] Available at: https://gim.studio/animalia/an-introduction-to-shell-based-fur-technique/ [Accessed 30 Jun. 2024].

Gmh, (n.d.). GitHub - gmh5225/Genshin-EasyPeasy-Bypass: A simple bypass of Genshin anti-cheat. Just run it after starting the game. Available at: https://github.com/gmh5225/Genshin-EasyPeasy-Bypass [Accessed 5 June 2024].

Groom brushes | ZBrush Docs, (n.d.). Available at: https://help.maxon.net/zbr/en-us/#html/user-guide/3d-modeling/fibermesh/groom-brushes/groom-brushes.html?TocPath=User%2520Guide%257C3D%2520Modeling%257CFiberMesh%25C2%25AE%257C 2 [Accessed 16 June 2024].

Hecomi, (2021.-a). Unity で URP 向けのファーシェーダを書いてみた(シェル法). 凹みTips, 24 July. Available at: https://tips.hecomi.com/entry/2021/06/27/185835 [Accessed 16 June 2024].

Hecomi, (2021.-b). Unity で URP 向けのファーシェーダを書いてみた(フィン法). 凹みTips, 24 July. Available at: https://tips.hecomi.com/entry/2021/07/24/121420 [Accessed 16 June 2024].

Hecomi, (2021.-c). Unity で URP 向けのファーシェーダを書いてみた(毛ポリゴン生成). 凹みTips, 12 August. Available at: https://tips.hecomi.com/entry/2021/08/12/155948 [Accessed 16 June 2024].

Hecomi, (2021.-d). Unity で URP 向けのファーシェーダを書いてみた(動き・アニメーション連携). 凹みTips, 14 August. Available at: https://tips.hecomi.com/entry/2021/08/14/115756 [Accessed 16 June 2024].

Hecomi, (2024). 凹みTips, 31 May. Available at: https://tips.hecomi.com/ [Accessed 5 June 2024].

Hecomi, (n.d.). GitHub - hecomi/UnityFurURP: Fur shader implementation for URP. Available at: https://github.com/hecomi/UnityFurURP [Accessed 5 June 2024].

Imgur (n.d.). What Video Game Hair Looks Like Without Texture Painted. [online] Imgur. Available at: https://imgur.com/gallery/what-video-game-hair-looks-like-without-texture-painted-g8vLC [Accessed 30 Jun. 2024].

Jordan Stevens, (n.d.). Lighting Models in Unity. Available at: https://www.jordanstevenstechart.com/lighting-models [Accessed 16 June 2024].

Maajor, (n.d.). GitHub - maajor/Marschner-Hair-Unity: Implement Marschner Shading Model In Unity. Available at: https://github.com/maajor/Marschner-Hair-Unity [Accessed 16 June 2024].

Machkovech, S. (2019). Detective Pikachu film review: This is how you adapt a video game for theaters. [online] Ars Technica. Available at: https://arstechnica.com/gaming/2019/05/detective-pikachu-film-review-this-is-how-you-adapt-a-video-game-for-theaters/ [Accessed 30 Jun. 2024].

Medium. (2018). Tips & Tricks on hair for Games - 80Level - Medium, 30 June. Available at: https://medium.com/@EightyLevel/tips-tricks-on-hair-for-games-66367a1b8be7 [Accessed 16 June 2024].

NBC News. (2006). 'Viva Piñata' is kid-friendly, and fun for adults. [online] Available at: https://www.nbcnews.com/id/wbna15835251 [Accessed 30 Jun. 2024].

phort99 (2015). Why your game's FPS counter should measure milliseconds per frame, not frames per second. [online] Available at: https://www.reddit.com/r/gamedev/comments/34z6cv/why_your_games_fps_counter_should_measure/ [Accessed 12 Jul. 2024].

Porter, M., (2017). Dark Souls 3 is Bandai Namco's fastest-selling game ever. IGN, 2 May. Available at: https://www.ign.com/articles/2016/04/18/dark-souls-3-is-bandai-namcos-fastest-selling-game-ever [Accessed 6 June 2024].

Samsung Developers. (n.d.). GPUWatch. [online] Available at: https://developer.samsung.com/galaxy-gamedev/gpuwatch.html.

Softo, (2024.). How to show FPS on Android phones: Xiaomi, Samsung, and other brands, 22 March. Available at: https://www.softo.org/p/show-fps-on-android-phones-xiaomi-samsung-and-other-brands [Accessed 6 June 2024].

Souls modding, (n.d.). FromSoftware game engines. Available at: http://soulsmodding.wikidot.com/topic:engines [Accessed 6 June 2024].

Technologies, U. (n.d.-a). Unity - Manual: Optimizing graphics performance. [online] docs.unity3d.com. Available at:

https://docs.unity3d.com/2017.3/Documentation/Manual/OptimizingGraphicsPerformance.html [Accessed 14 Jul. 2024].

Technologies, U. (n.d.-b). Unity - Manual: Shader data types and precision. [online] docs.unity3d.com. Available at: https://docs.unity3d.com/2017.3/Documentation/Manual/SL-DataTypesAndPrecision.html [Accessed 14 Jul. 2024].

TheGamedev.Guru, (n.d.). Unity Overdraw: improving the GPU performance of your game. Available at: https://thegamedev.guru/unity-gpu-performance/overdraw-optimization/ [Accessed 16 June 2024].

The Ryg Blog, (2013). A trip through the Graphics Pipeline 2011: Index, 10 March. Available at: https://fgiesen.wordpress.com/2011/07/09/a-trip-through-the-graphics-pipeline-2011-index/ [Accessed 16 June 2024].

Tutorials made easy!, (n.d.). Generating Fur in DirectX or OpenGL Easily. Available at: https://www.xbdev.net/directx3dx/specialX/Fur/index.php [Accessed 16 June 2024].

TurboSquid, (n.d.). 3D Character140 Rigged Horse. Available at: https://www.turbosquid.com/3d-models/character140-rigged-horse-1761468 [Accessed 15 June 2024].

Tyler, J.E., (2021). Genshin Impact beats Fortnite, GTA 5 revenue in best first year ever. *ScreenRant*, 4 November. Available at: https://screenrant.com/genshin-impact-fortnite-gta5-first-year-revenue/ [Accessed 6 June 2024].

Unity, (n.d.-a). *Detecting performance bottlenecks with Unity Frame Timing Manager*. [online] Available at: https://unity.com/blog/engine-platform/detecting-performance-bottlenecks-with-unity-frame-timing-manager [Accessed 12 Jul. 2024].

Unity, (n.d.-b). *How to use Unity’s memory profiling tools*. Available at: https://unity.com/how-to/use-memory-profiling-unity [Accessed 6 June 2024].

Unity Forum, (n.d.). How to view used GPU’s memory. Available at: https://forum.unity.com/threads/how-to-view-used-gpus-memory.1176740/ [Accessed 5 June 2024].

Unity Technologies, (n.d.-a). Create high-quality graphics and stunning visuals. Available at: https://unity.com/features/high-definition-render-pipeline [Accessed 3 June 2024].

Unity Technologies, (n.d.-b). Download archive. Available at: https://unity.com/releases/editor/archive [Accessed 3 June 2024].

Unity Technologies, (n.d.-c). Fur shader. Available at: https://forum.unity.com/threads/fur-shader.4581/ [Accessed 6 June 2024].

Unity Technologies. (n.d.-d). LTS vs Tech Stream: Choose the right Unity release for you. [online] Available at: https://unity.com/releases/editor/whats-new/lts-vs-tech-stream [Accessed 3 June 2024].

Unity Technologies. (n.d.-e). Universal Render Pipeline overview. [online] Available at: https://docs.unity3d.com/Packages/com.unity.render-pipelines.universal@17.0/manual/index.html [Accessed 3 June 2024].

Unity Technologies. (n.d.-f). Unity - Manual: ScriptableObject. [online] Available at: https://docs.unity3d.com/Manual/class-ScriptableObject.html [Accessed 6 June 2024].

Unity Technologies. (n.d.-g). Using the Built-in Render Pipeline. [online] Available at: https://docs.unity3d.com/Manual/built-in-render-pipeline.html [Accessed 3 June 2024].

Unity-Technologies. (n.d.-h). Graphics/Packages/com.unity.render-pipelines.universal/ShaderLibrary/Lighting.hlsl at master · Unity-Technologies/Graphics. GitHub. [online] Available at: https://github.com/Unity-Technologies/Graphics/blob/master/Packages/com.unity.render-pipelines.universal/ShaderLibrary/Lighting.hlsl [Accessed 6 June 2024].

Wikipedia. (2024). Samsung Galaxy A50. [online] Available at: https://en.wikipedia.org/wiki/Samsung_Galaxy_A50 [Accessed 4 June 2024].

Wikipedia contributors. (2020). Dark Souls III. Wikipedia. [online] Available at: https://en.wikipedia.org/wiki/Dark_Souls_III [Accessed 6 June 2024].

Wikipedia contributors. (2023). Viva Piñata. Wikipedia; Wikimedia Foundation. [online] Available at: https://en.wikipedia.org/wiki/Viva_Pi%C3%B1ata [Accessed 6 June 2024].

Wikipedia contributors. (2024.-a). Shadow of the Colossus. Wikipedia. [online] Available at: https://en.wikipedia.org/wiki/Shadow_of_the_Colossus [Accessed 6 June 2024].

Wikipedia contributors. (2024.-b). Flicker fusion threshold. Wikipedia. [online] Available at: https://en.wikipedia.org/wiki/Flicker_fusion_threshold#Display_frame_rate [Accessed 5 June 2024].

Wikipedia contributors. (2024.-c.). Moore's law. Wikipedia. [online] Available at: https://en.wikipedia.org/wiki/Moore%27s_law [Accessed 5 June 2024].

Wikipedia contributors. (2024.-d). Genshin Impact. Wikipedia. [online] Available at: https://en.wikipedia.org/wiki/Genshin_Impact [Accessed 6 June 2024].

Wikipedia contributors. (2024.-e). Bandai Namco Entertainment. Wikipedia. [online] Available at: https://en.wikipedia.org/wiki/Bandai_Namco_Entertainment [Accessed 6 June 2024].

Wikipedia contributors. (2024.-f). Rare (company). Wikipedia. [online] Available at: https://en.wikipedia.org/wiki/Rare_(company) [Accessed 6 June 2024].

Wikipedia contributors. (2024.-g). Xbox Game Studios. Wikipedia. [online] Available at: https://en.wikipedia.org/wiki/Xbox_Game_Studios [Accessed 6 June 2024].

Wikipedia contributors. (2024.-h). MiHoYo. [online] Available at: https://en.wikipedia.org/wiki/MiHoYo [Accessed 6 June 2024].

Wikipedia contributors. (2024.-i). FromSoftware. Wikipedia. [online] Available at: https://en.wikipedia.org/wiki/FromSoftware [Accessed 6 June 2024].

X (formerly Twitter). (2024). x.com. [online] Available at: https://x.com/cosmicmatt/status/1731083457290555525/photo/2 [Accessed 30 Jun. 2024].

XBDEV.net. (n.d.). Fur rendering. [online] Available at: https://www.xbdev.net/directx3dx/specialX/Fur/index.php [Accessed 3 June 2024].

9. Figures

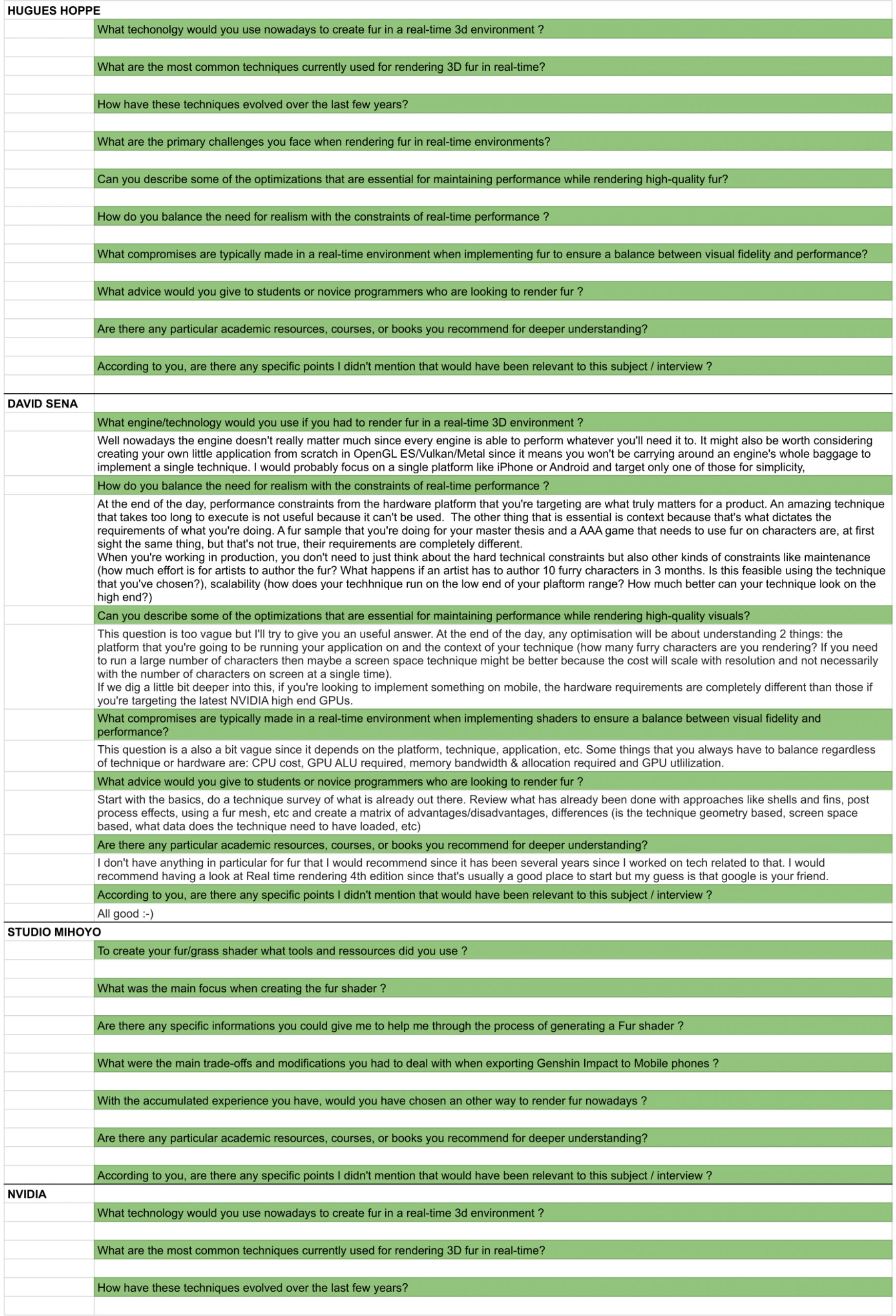
Figure 1 - Hair and fur Render Time on characters. (Edgardlop, 2013) 2
Figure 2 - Fur generation example from Real-Time Fur over Arbitrary Surfaces. (Lengyel et al., 2001) 5
Figure 3 - 80.lv student hair for games from Greg Mourino. (80.lv, 2018) & Detective Pikachu film caption. (Machkovech, 2019) 6
Figure 4 - Realistic Dog Portrait: Experimenting with Real-Time Fur. (80.lv, 2021) 7
Figure 5 - Unity download archive. (Unity Technologies, n.d.-b) 8
Figure 6 - Unity Hub 3.7.0 Installs screenshot, (Styles, 2024) 9
Figure 7 – A shell-based fur strand example. (GiM, n.d.) 10
Figure 8 - Visualisation of shells on low-poly mesh. (GiM, n.d.) 11
Figure 9 - Example of coloured Shell-based fur. (Afanasev, 2018) 11
Figure 10 - Illustration of fin extrusion. (Lengyel et al., 2001) 12
Figure 11 - Fin generation example and illustration. (Tariq, 2007) 13
Figure 12 - Polygonal hair example from The Last of Us. (Imgur, n.d.) 14
Figure 13 - Vertex shader with adaptive mesh according to LOD. (GDCVault, n.d.) 15
Figure 14 - Shadow of Colossus in game caption of the player looking at a Colossus. (Froyok, n.d.) 16
Figure 15 - Viva Piñata in Game Caption. (NBC News, 2006) 17
Figure 16 - Dark Souls III Wolf boss (Sif). (Creswell, 2021) 18
Figure 17 - Genshin impact illustration. (For The Win, 2022) 19
Figure 18 - In-game caption of Genshin Impact. (X (formerly Twitter), 2024) 19
Figure 19 - Genshin impact caption with coloured rectangle highlighting specific elements. (X (formerly Twitter), 2024) 26
Figure 20 - Caption in Unity editor of the cloned github project from Hecomi showing shell, fin, and geometry shader from left to right. (Styles, 2024) 27
Figure 21 - Caption in Unity editor of the cloned github project from Hecomi showing shell. (Styles, 2024) 27
Figure 22 - Caption in Unity editor of the cloned github project from Hecomi showing shell material parameters . (Styles, 2024) 28
Figure 23 - Caption in Unity editor of the project from Hecomi showing measures taken. (Styles, 2024) 29
Figure 24 - Capture of the Render Debugger from Unity. (Styles, 2024) 31
Figure 25 - Unity editor caption of the Memory Profiler tool indicating the difference memory allocations. (Styles, 2024) 32
Figure 26 - Unity editor Statistics tool. (Styles, 2024) 33
Figure 27 - Unity Profiler (top) and Timeline (bottom). (Styles, 2024) 34
Figure 28 - Unity Rendering Debugger tool showing the Display Stats category. (Styles, 2024) 34
Figure 29 - Unity editor caption of the Memory Profiler tool. (Styles, 2024) 35
Figure 30 - Unity editor screenshot of the Frame Debugger tool. (Styles, 2024) 36
Figure 31 - RenderDoc used on Hecomi's project. (Styles, 2024) 36
Figure 32 - NVIDIA Nsight Graphics used on the initial approach project. (Styles, 2024) 37

Figure 33 - Combination of screenshots resuming the process to activate GPUWatch on Samsung Galaxy A50. (Styles, 2024) 38
Figure 34 - UnityHub screenshot of the Installs category showing the 2022.3.24f1 LTS version. (Styles, 2024) 40
Figure 35 - Unity editor Package manager tool showing the installed packages. (Styles, 2024) 40
Figure 36 - Visual Studio IDE Installer showing the packages category. (Styles, 2024) 41
Figure 37 - Unity editor NaïveShell project exposing the Canvas prefab in the Hierarchy. (Styles, 2024). 42
Figure 38 - Unity editor screenshots of two Inspectors exposing the SimpleShell script and HLSLGeom shader material parameters. (Styles, 2024) 43
Figure 39 - Unity scene showing attempts at Shell and Fin shader merging. (Styles, 2024) 43
Figure 40 - Unity editor screenshot showing the HLSLGeom1 scene with light information in the inspector. (Styles, 2024) 44
Figure 41 - Unity editor screenshot showing the HLSL_ComplexScene. (Styles, 2024) 44
Figure 42 - Capture of the Unity Build Settings window. (Styles, 2024) 45
Figure 43 - Unity editor screenshot of the Inspector window showing the relation between player input and HLSL shader. (Styles, 2024) 45
Figure 44 - Unity editor screenshot showing the rigger horse from TruboSquid with the HLSL shader on it. (Styles, 2024) 46
Figure 45 - Unity capture showing the Sphere object's inspector. (Styles, 2024) 47
Figure 46 - Unity editor showing the mesh layering from the initial approach project. (Styles, 2024) 47
Figure 47 - Capture of the SimpleShell.cs script in the Visual Studio IDE. (Styles, 2024) 48
Figure 48 - Capture of Hugo Elias’ hashing function from the Shell.shader script in the Visual Studio IDE. (Styles, 2024) 48
Figure 49 - Capture of the Vertex shader code from the Shell.shader script in the Visual Studio IDE. (Styles, 2024) 49
Figure 50 - Capture of the fragment shader code from the Shell.shader script in the Visual Studio IDE. (Styles, 2024) 50
Figure 51 - Capture of the initial approach scene in the Unity editor. (Styles, 2024) 51
Figure 52 - Unity editor capture of the HLSLGeom shader's material showing parameters in the inspector window. (Styles, 2024) 52
Figure 53 - Capture of the Geometry shader code from the Fur.hlsl script in the Visual Studio IDE. (Styles, 2024) 53
Figure 54 - Unity editor capture of the HLSLGeom scene. (Styles, 2024) 53
Figure 55 - Unity editor capture of the material parameters in the inspector window. (Styles, 2024) 54
Figure 56 - Unity editor capture of the HLSL_ComplexObject scene showing the riggerd horse without colour (left) and with colour (right). (Styles, 2024) 55
Figure 57 - Unity editor inspector window capture showing the shell detail parameters. (Styles, 2024) 55
Figure 58 - Unity editor capture of three shader variations showing the rigged horse with short (right), long (middle), and red tinted (left) fur. (Styles, 2024) 56
Figure 59 - Unity editor inspector capture showing the Animator component. (Styles, 2024) 56
Figure 60 - Unity editor Hierarchy capture showing the Canvas prefab hierarchy. (Styles, 2024) 57

Figure 61 - Unity editor inspector window capture showing the EventSytem component. (Styles, 2024) 57
Figure 62 - Unity editor capture of the Project Settings window showing the Active Input Handling parameter. (Styles, 2024) 58
Figure 63 - Hecomi's project measurements, (Styles, 2024) 59
Figure 64 - Unity editor bottom left UI options showing the release (left) and debug (right) presets. (Styles, 2024) 60
Figure 65 - Measurement of the Initial approach project. (Styles, 2024) 60
Figure 66 - Unity editor capture showing the measurement of the NaïveShell scene. (Styles, 2024) 61
Figure 67 - Unity editor capture showing the Memory Profiler on the NaïveShell scene. (Styles, 2024) 61
Figure 68 - Screen capture of the Android Galaxy A50 measurements for the NaïveShell scene. (Styles, 2024) 62
Figure 69 - Measurements of the HLSL Shader project. (Styles, 2024) 63
Figure 70 - Unity editor capture showing the measurement of the HLSLGeom scene. (Styles, 2024) 63
Figure 71 - Unity editor capture showing the Memory Profiler on the HLSLGeom scene. (Styles, 2024) 64
Figure 72 - Screen capture of the Android Galaxy A50 measurements for the HLSLGeom scene. (Styles, 2024) 64
Figure 73 - Unity editor capture showing the measurement of the HLSLGeom1 scene. (Styles, 2024) 65
Figure 74 - Measurement comparison on the HLSL Shader project with and without light 65
Figure 75 - Screen capture of the Android Galaxy A50 measurements for the HLSLGeom1 scene. (Styles, 2024) 66
Figure 76 - Unity editor capture showing the measurement of the HLSL_ComplexeScene scene. (Styles, 2024) 66
Figure 77 - Measurement of the HLSL Shader in the complex scene project. (Styles, 2024) 67
Figure 78 - Capture of RenderDoc usage on the HLSL_ComplexeScene scene. (Styles, 2024) 67
Figure 79 - Unity editor capture showing the Memory Profiler on the HLSL_ComplexeScene scene. (Styles, 2024) 68
Figure 80 - Screen capture of the Android Galaxy A50 measurements for the HLSL_ComplexeScene scene. (Styles, 2024) 68
Figure 81 - Measurements of the HLSL Complex object scene. (Styles, 2024) 69
Figure 82 - Unity editor capture showing the measurement of the HLSL_ComplexeObject scene. (Styles, 2024) 69
Figure 83 - Unity editor capture showing the Memory Profiler on the HLSL_ComplexeObject scene. (Styles, 2024) 70
Figure 84 - Screen capture of the Android Galaxy A50 measurements for the HLSL_ComplexeObject scene. (Styles, 2024) 70
Figure 85 - Measurements of the GPU frame time and comparison on all projects. (Styles, 2024) 71
Figure 86 - Measurements of the multiplication factor and comparison on all projects. (Styles, 2024) 71
Figure 87 - Measurements of the graphical memory allocation and comparison on all projects. (Styles, 2024) 72
Figure 88 - Measurements of the frames per second and comparison on all projects. (Styles, 2024) 72

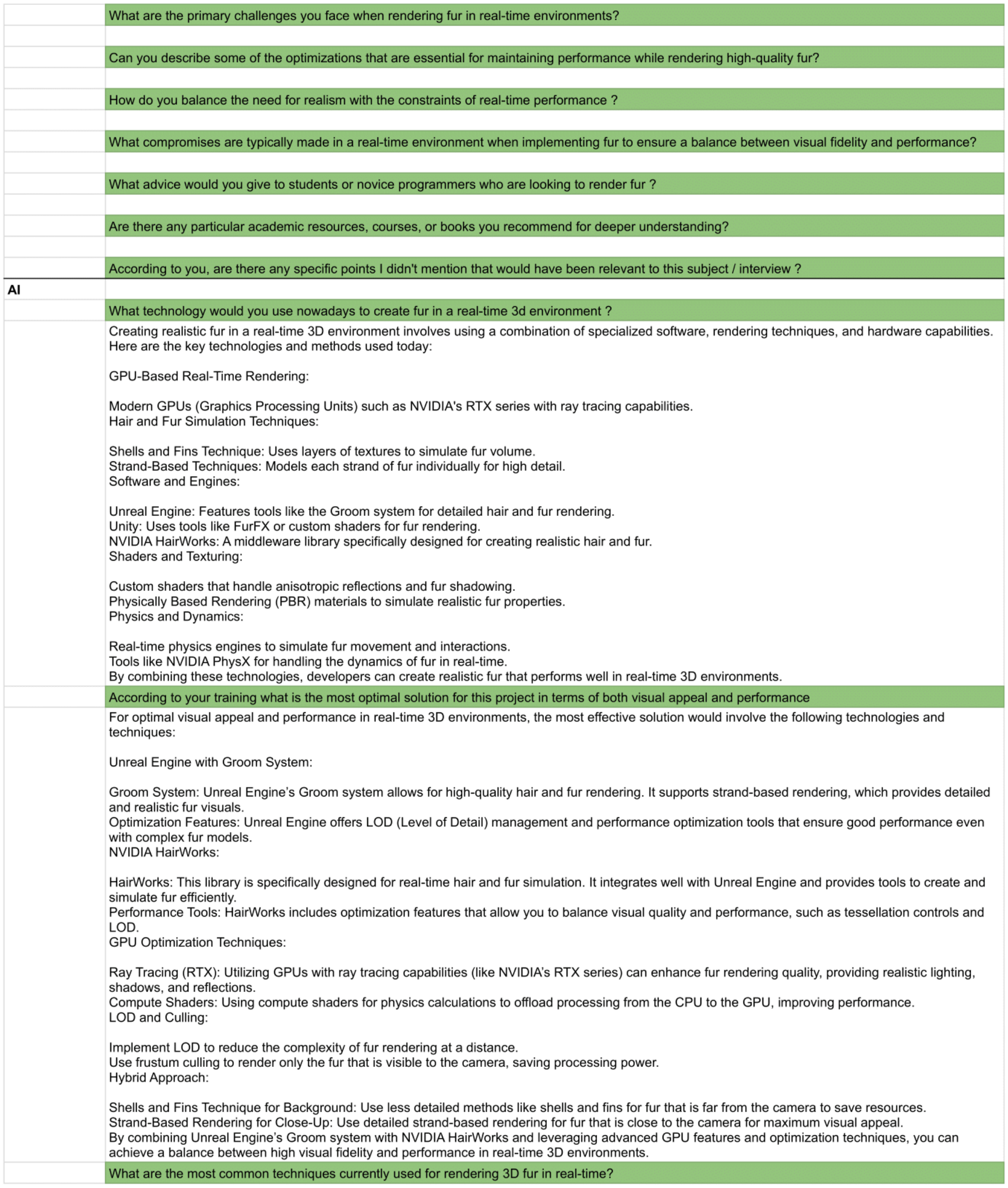
10. Appendices

Appendix A: Interview of David Sena

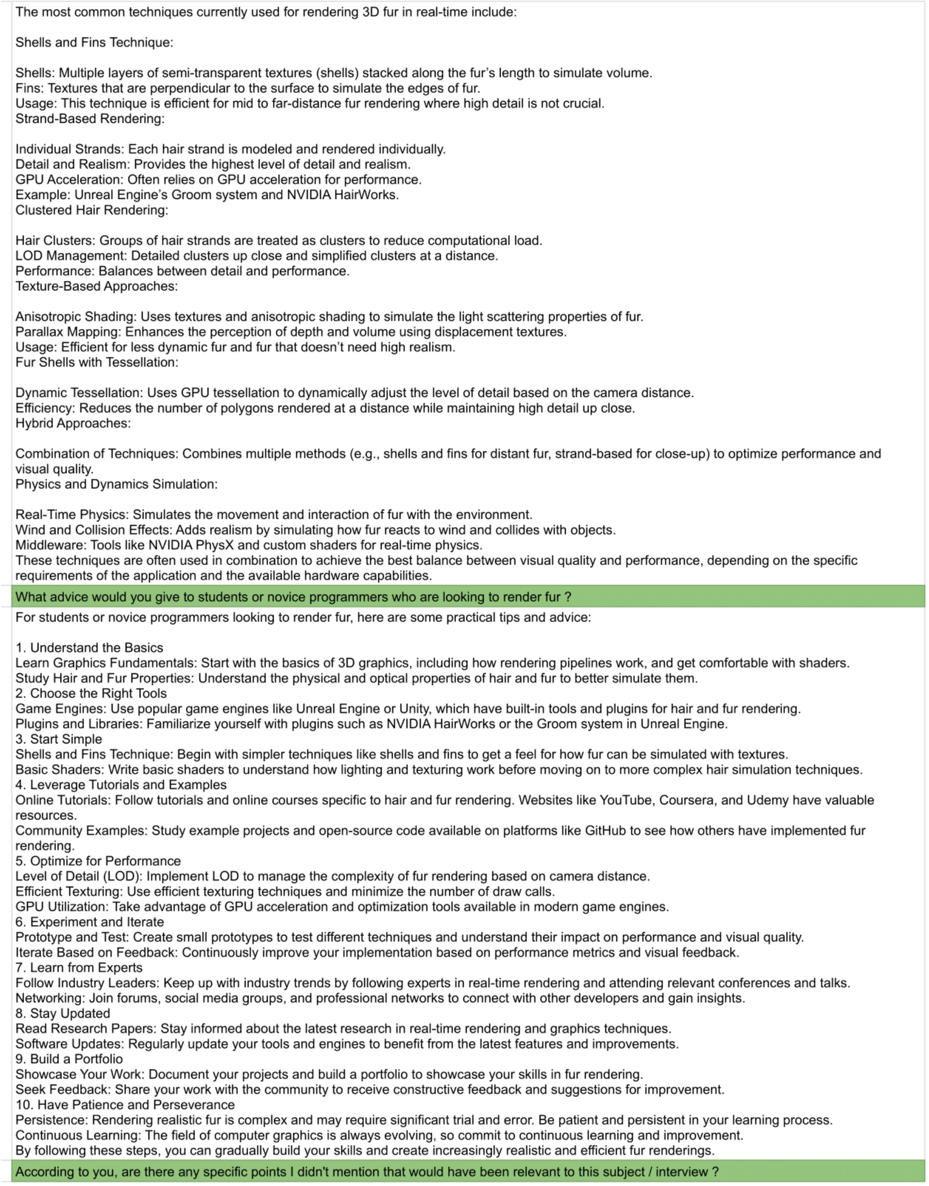


HUGUES HOPPE	
	What techonolgy would you use nowadays to create fur in a real-time 3d environment ?
	What are the most common techniques currently used for rendering 3D fur in real-time?
	How have these techniques evolved over the last few years?
	What are the primary challenges you face when rendering fur in real-time environments?
	Can you describe some of the optimizations that are essential for maintaining performance while rendering high-quality fur?
	How do you balance the need for realism with the constraints of real-time performance ?
	What compromises are typically made in a real-time environment when implementing fur to ensure a balance between visual fidelity and performance?
	What advice would you give to students or novice programmers who are looking to render fur ?
	Are there any particular academic resources, courses, or books you recommend for deeper understanding?
	According to you, are there any specific points I didn't mention that would have been relevant to this subject / interview ?
DAVID SENA	
	What engine/technology would you use if you had to render fur in a real-time 3D environment ?
	Well nowadays the engine doesn't really matter much since every engine is able to perform whatever you'll need it to. It might also be worth considering creating your own little application from scratch in OpenGL ES/Vulkan/Metal since it means you won't be carrying around an engine's whole baggage to implement a single technique. I would probably focus on a single platform like iPhone or Android and target only one of those for simplicity,
	How do you balance the need for realism with the constraints of real-time performance ?
	At the end of the day, performance constraints from the hardware platform that you're targeting are what truly matters for a product. An amazing technique that takes too long to execute is not useful because it can't be used. The other thing that is essential is context because that's what dictates the requirements of what you're doing. A fur sample that you're doing for your master thesis and a AAA game that needs to use fur on characters are, at first sight the same thing, but that's not true, their requirements are completely different. When you're working in production, you don't need to just think about the hard technical constraints but also other kinds of constraints like maintenance (how much effort is for artists to author the fur? What happens if an artist has to author 10 furry characters in 3 months. Is this feasible using the technique that you've chosen?), scalability (how does your techhnique run on the low end of your plaftorm range? How much better can your technique look on the high end?)
	Can you describe some of the optimizations that are essential for maintaining performance while rendering high-quality visuals?
	This question is too vague but I'll try to give you an useful answer. At the end of the day, any optimisation will be about understanding 2 things: the platform that you're going to be running your application on and the context of your technique (how many furry characters are you rendering? If you need to run a large number of characters then maybe a screen space technique might be better because the cost will scale with resolution and not necessarily with the number of characters on screen at a single time). If we dig a little bit deeper into this, if you're looking to implement something on mobile, the hardware requirements are completely different than those if you're targeting the latest NVIDIA high end GPUs.
	What compromises are typically made in a real-time environment when implementing shaders to ensure a balance between visual fidelity and performance?
	This question is a also a bit vague since it depends on the platform, technique, application, etc. Some things that you always have to balance regardless of technique or hardware are: CPU cost, GPU ALU required, memory bandwidth & allocation required and GPU utlilization.
	What advice would you give to students or novice programmers who are looking to render fur ?
	Start with the basics, do a technique survey of what is already out there. Review what has already been done with approaches like shells and fins, post process effects, using a fur mesh, etc and create a matrix of advantages/disadvantages, differences (is the technique geometry based, screen space based, what data does the technique need to have loaded, etc)
	Are there any particular academic resources, courses, or books you recommend for deeper understanding?
	I don't have anything in particular for fur that I would recommend since it has been several years since I worked on tech related to that. I would recommend having a look at Real time rendering 4th edition since that's usually a good place to start but my guess is that google is your friend.
	According to you, are there any specific points I didn't mention that would have been relevant to this subject / interview ?
	All good :-)
STUDIO MIHOYO	
	To create your fur/grass shader what tools and ressources did you use ?
	What was the main focus when creating the fur shader ?
	Are there any specific informations you could give me to help me through the process of generating a Fur shader ?
	What were the main trade-offs and modifications you had to deal with when exporting Genshin Impact to Mobile phones ?
	With the accumulated experience you have, would you have chosen an other way to render fur nowadays ?
	Are there any particular academic resources, courses, or books you recommend for deeper understanding?
	According to you, are there any specific points I didn't mention that would have been relevant to this subject / interview ?
NVIDIA	
	What technology would you use nowadays to create fur in a real-time 3d environment ?
	What are the most common techniques currently used for rendering 3D fur in real-time?
	How have these techniques evolved over the last few years?

Appendix B: Interview of ChatGPT



	What are the primary challenges you face when rendering fur in real-time environments?
	Can you describe some of the optimizations that are essential for maintaining performance while rendering high-quality fur?
	How do you balance the need for realism with the constraints of real-time performance ?
	What compromises are typically made in a real-time environment when implementing fur to ensure a balance between visual fidelity and performance?
	What advice would you give to students or novice programmers who are looking to render fur ?
	Are there any particular academic resources, courses, or books you recommend for deeper understanding?
	According to you, are there any specific points I didn't mention that would have been relevant to this subject / interview ?
AI	
	What technology would you use nowadays to create fur in a real-time 3d environment ?
	Creating realistic fur in a real-time 3D environment involves using a combination of specialized software, rendering techniques, and hardware capabilities. Here are the key technologies and methods used today: GPU-Based Real-Time Rendering: Modern GPUs (Graphics Processing Units) such as NVIDIA's RTX series with ray tracing capabilities. Hair and Fur Simulation Techniques: Shells and Fins Technique: Uses layers of textures to simulate fur volume. Strand-Based Techniques: Models each strand of fur individually for high detail. Software and Engines: Unreal Engine: Features tools like the Groom system for detailed hair and fur rendering. Unity: Uses tools like FurFX or custom shaders for fur rendering. NVIDIA HairWorks: A middleware library specifically designed for creating realistic hair and fur. Shaders and Texturing: Custom shaders that handle anisotropic reflections and fur shadowing. Physically Based Rendering (PBR) materials to simulate realistic fur properties. Physics and Dynamics: Real-time physics engines to simulate fur movement and interactions. Tools like NVIDIA PhysX for handling the dynamics of fur in real-time. By combining these technologies, developers can create realistic fur that performs well in real-time 3D environments.
	According to your training what is the most optimal solution for this project in terms of both visual appeal and performance
	For optimal visual appeal and performance in real-time 3D environments, the most effective solution would involve the following technologies and techniques: Unreal Engine with Groom System: Groom System: Unreal Engine’s Groom system allows for high-quality hair and fur rendering. It supports strand-based rendering, which provides detailed and realistic fur visuals. Optimization Features: Unreal Engine offers LOD (Level of Detail) management and performance optimization tools that ensure good performance even with complex fur models. NVIDIA HairWorks: HairWorks: This library is specifically designed for real-time hair and fur simulation. It integrates well with Unreal Engine and provides tools to create and simulate fur efficiently. Performance Tools: HairWorks includes optimization features that allow you to balance visual quality and performance, such as tessellation controls and LOD. GPU Optimization Techniques: Ray Tracing (RTX): Utilizing GPUs with ray tracing capabilities (like NVIDIA’s RTX series) can enhance fur rendering quality, providing realistic lighting, shadows, and reflections. Compute Shaders: Using compute shaders for physics calculations to offload processing from the CPU to the GPU, improving performance. LOD and Culling: Implement LOD to reduce the complexity of fur rendering at a distance. Use frustum culling to render only the fur that is visible to the camera, saving processing power. Hybrid Approach: Shells and Fins Technique for Background: Use less detailed methods like shells and fins for fur that is far from the camera to save resources. Strand-Based Rendering for Close-Up: Use detailed strand-based rendering for fur that is close to the camera for maximum visual appeal. By combining Unreal Engine’s Groom system with NVIDIA HairWorks and leveraging advanced GPU features and optimization techniques, you can achieve a balance between high visual fidelity and performance in real-time 3D environments.
	What are the most common techniques currently used for rendering 3D fur in real-time?



The most common techniques currently used for rendering 3D fur in real-time include:

Shells and Fins Technique:

Shells: Multiple layers of semi-transparent textures (shells) stacked along the fur’s length to simulate volume.
Fins: Textures that are perpendicular to the surface to simulate the edges of fur.
Usage: This technique is efficient for mid to far-distance fur rendering where high detail is not crucial.
Strand-Based Rendering:

Individual Strands: Each hair strand is modeled and rendered individually.
Detail and Realism: Provides the highest level of detail and realism.
GPU Acceleration: Often relies on GPU acceleration for performance.
Example: Unreal Engine’s Groom system and NVIDIA HairWorks.
Clustered Hair Rendering:

Hair Clusters: Groups of hair strands are treated as clusters to reduce computational load.
LOD Management: Detailed clusters up close and simplified clusters at a distance.
Performance: Balances between detail and performance.
Texture-Based Approaches:

Anisotropic Shading: Uses textures and anisotropic shading to simulate the light scattering properties of fur.
Parallax Mapping: Enhances the perception of depth and volume using displacement textures.
Usage: Efficient for less dynamic fur and fur that doesn’t need high realism.
Fur Shells with Tessellation:

Dynamic Tessellation: Uses GPU tessellation to dynamically adjust the level of detail based on the camera distance.
Efficiency: Reduces the number of polygons rendered at a distance while maintaining high detail up close.
Hybrid Approaches:

Combination of Techniques: Combines multiple methods (e.g., shells and fins for distant fur, strand-based for close-up) to optimize performance and visual quality.
Physics and Dynamics Simulation:

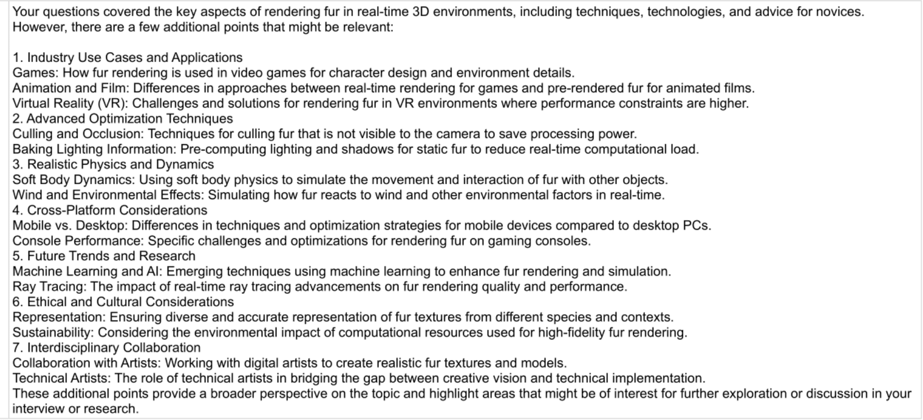
Real-Time Physics: Simulates the movement and interaction of fur with the environment.
Wind and Collision Effects: Adds realism by simulating how fur reacts to wind and collides with objects.
Middleware: Tools like NVIDIA PhysX and custom shaders for real-time physics.
These techniques are often used in combination to achieve the best balance between visual quality and performance, depending on the specific requirements of the application and the available hardware capabilities.

What advice would you give to students or novice programmers who are looking to render fur ?

For students or novice programmers looking to render fur, here are some practical tips and advice:

1. Understand the Basics
Learn Graphics Fundamentals: Start with the basics of 3D graphics, including how rendering pipelines work, and get comfortable with shaders.
Study Hair and Fur Properties: Understand the physical and optical properties of hair and fur to better simulate them.
2. Choose the Right Tools
Game Engines: Use popular game engines like Unreal Engine or Unity, which have built-in tools and plugins for hair and fur rendering.
Plugins and Libraries: Familiarize yourself with plugins such as NVIDIA HairWorks or the Groom system in Unreal Engine.
3. Start Simple
Shells and Fins Technique: Begin with simpler techniques like shells and fins to get a feel for how fur can be simulated with textures.
Basic Shaders: Write basic shaders to understand how lighting and texturing work before moving on to more complex hair simulation techniques.
4. Leverage Tutorials and Examples
Online Tutorials: Follow tutorials and online courses specific to hair and fur rendering. Websites like YouTube, Coursera, and Udemy have valuable resources.
Community Examples: Study example projects and open-source code available on platforms like GitHub to see how others have implemented fur rendering.
5. Optimize for Performance
Level of Detail (LOD): Implement LOD to manage the complexity of fur rendering based on camera distance.
Efficient Texturing: Use efficient texturing techniques and minimize the number of draw calls.
GPU Utilization: Take advantage of GPU acceleration and optimization tools available in modern game engines.
6. Experiment and Iterate
Prototype and Test: Create small prototypes to test different techniques and understand their impact on performance and visual quality.
Iterate Based on Feedback: Continuously improve your implementation based on performance metrics and visual feedback.
7. Learn from Experts
Follow Industry Leaders: Keep up with industry trends by following experts in real-time rendering and attending relevant conferences and talks.
Networking: Join forums, social media groups, and professional networks to connect with other developers and gain insights.
8. Stay Updated
Read Research Papers: Stay informed about the latest research in real-time rendering and graphics techniques.
Software Updates: Regularly update your tools and engines to benefit from the latest features and improvements.
9. Build a Portfolio
Showcase Your Work: Document your projects and build a portfolio to showcase your skills in fur rendering.
Seek Feedback: Share your work with the community to receive constructive feedback and suggestions for improvement.
10. Have Patience and Perseverance
Persistence: Rendering realistic fur is complex and may require significant trial and error. Be patient and persistent in your learning process.
Continuous Learning: The field of computer graphics is always evolving, so commit to continuous learning and improvement.
By following these steps, you can gradually build your skills and create increasingly realistic and efficient fur renderings.

According to you, are there any specific points I didn't mention that would have been relevant to this subject / interview ?



Your questions covered the key aspects of rendering fur in real-time 3D environments, including techniques, technologies, and advice for novices. However, there are a few additional points that might be relevant:

1. Industry Use Cases and Applications
Games: How fur rendering is used in video games for character design and environment details.
Animation and Film: Differences in approaches between real-time rendering for games and pre-rendered fur for animated films.
Virtual Reality (VR): Challenges and solutions for rendering fur in VR environments where performance constraints are higher.
2. Advanced Optimization Techniques
Culling and Occlusion: Techniques for culling fur that is not visible to the camera to save processing power.
Baking Lighting Information: Pre-computing lighting and shadows for static fur to reduce real-time computational load.
3. Realistic Physics and Dynamics
Soft Body Dynamics: Using soft body physics to simulate the movement and interaction of fur with other objects.
Wind and Environmental Effects: Simulating how fur reacts to wind and other environmental factors in real-time.
4. Cross-Platform Considerations
Mobile vs. Desktop: Differences in techniques and optimization strategies for mobile devices compared to desktop PCs.
Console Performance: Specific challenges and optimizations for rendering fur on gaming consoles.
5. Future Trends and Research
Machine Learning and AI: Emerging techniques using machine learning to enhance fur rendering and simulation.
Ray Tracing: The impact of real-time ray tracing advancements on fur rendering quality and performance.
6. Ethical and Cultural Considerations
Representation: Ensuring diverse and accurate representation of fur textures from different species and contexts.
Sustainability: Considering the environmental impact of computational resources used for high-fidelity fur rendering.
7. Interdisciplinary Collaboration
Collaboration with Artists: Working with digital artists to create realistic fur textures and models.
Technical Artists: The role of technical artists in bridging the gap between creative vision and technical implementation.
These additional points provide a broader perspective on the topic and highlight areas that might be of interest for further exploration or discussion in your interview or research.

Appendix C: Shell_Original.shader code

```
Shader "Custom/Fur_Original" {
       SubShader {
              Tags {
                     "LightMode" = "ForwardBase"
              }

              Pass {
            Cull Off

                     CGPROGRAM

                     #pragma vertex vp
                     #pragma fragment fp

                     #include "UnityPBSLighting.cginc"
            #include "AutoLight.cginc"

                     struct VertexData {
                            float4 vertex : POSITION;
                            float3 normal : NORMAL;
                float2 uv : TEXCOORD0;
                     };

                     struct v2f {
                            float4 pos : SV_POSITION;
                float2 uv : TEXCOORD0;
                            float3 normal : TEXCOORD1;
                            float3 worldPos : TEXCOORD2;
                     };

            int _ShellIndex;
                     int _ShellCount;
                     float _ShellLength;
                     float _Density;
                     float _NoiseMin, _NoiseMax;
                     float _Thickness;
                     float _Attenuation;
                     float _OcclusionBias;
                     float _ShellDistanceAttenuation;
                     float _Curvature;
                     float _DisplacementStrength;
                     float3 _ShellColor;
                     float3 _ShellDirection;


                     float hash(uint n) {
                            // Hash from Hugo Elias
                            n = (n << 13U) ^ n;
                            n = n * (n * n * 15731U + 0x789221U) + 0x1376312589U;
                            return float(n & uint(0x7fffffffU)) / float(0x7fffffff);
                     }


                     v2f vp(VertexData v) {
```

```
            v2f i;

            float shellHeight = (float)_ShellIndex / (float)_ShellCount;

            shellHeight = pow(shellHeight, _ShellDistanceAttenuation);

            v.vertex.xyz += v.normal.xyz * _ShellLength * shellHeight;

i.normal = normalize(UnityObjectToWorldNormal(v.normal));

            float k = pow(shellHeight, _Curvature);

            v.vertex.xyz += _ShellDirection * k * _DisplacementStrength;

i.pos = UnityObjectToClipPos(v.vertex);

i.uv = v.uv;

            return i;
    }

    float4 fp(v2f i) : SV_TARGET {
            float2 newUV = i.uv * _Density;

            float2 localUV = frac(newUV) * 2 - 1;

            float localDistanceFromCenter = length(localUV);

uint2 tid = newUV;
            uint seed = tid.x + 100 * tid.y + 100 * 10;

float shellIndex = _ShellIndex;
float shellCount = _ShellCount;

float rand = lerp(_NoiseMin, _NoiseMax, hash(seed));

float h = shellIndex / shellCount;

            int outsideThickness = (localDistanceFromCenter) > (_Thickness
            * (rand - h));

            if (outsideThickness && _ShellIndex > 0) discard;

            float ndotl = DotClamped(i.normal, _WorldSpaceLightPos0) *
            0.5f + 0.5f;

            ndotl = ndotl * ndotl;

            float ambientOcclusion = pow(h, _Attenuation);

            ambientOcclusion += _OcclusionBias;

            ambientOcclusion = saturate(ambientOcclusion);

return float4(_ShellColor * ndotl * ambientOcclusion, 1.0);
    }
    ENDCG}}}
```

Appendix D: Shell.shader code

```
Shader "Custom/Fur" {
       SubShader {
              Tags {
                     "RenderType" = "Transparent"
                     "RenderPipeline" = "UniversalPipeline"
                     "UniversalMaterialType" = "Lit"
                     "LightMode" = "UniversalForward"
                     "Queue" = "Transparent"
              }

              Pass
              {
                     //Shader parameters
                     LOD 100
                     Cull Back


                     Name "ForwardLit"
            Tags{"LightMode" = "UniversalForward"}


                     //Beginning of the program
                     HLSLPROGRAM

                     //Declaraction of Vertex and Fragment Shader
                     #pragma vertex vert
                     #pragma fragment frag

                     //Inclusion of Unity Libraries
                     #include "Packages/com.unity.render-
pipelines.universal/ShaderLibrary/Core.hlsl"
                     #include "Packages/com.unity.render-
pipelines.universal/Shaders/LitInput.hlsl"

                     //Attribute struct
                     struct Attributes
                     {
                            float4 vertex : POSITION;
                            float3 normal : NORMAL;
                float2 uv : TEXCOORD0;
                     };

                     //Varyings is the strucs that replaces v2f (Vertex to Frament)
                     struct Varyings
                     {
                            float4 pos : SV_POSITION;
                float2 uv : TEXCOORD0;
                            float3 normal : TEXCOORD1;
                            float3 worldPos : TEXCOORD2;
                     };


                     CBUFFER_START(UnityPerMaterial)
            int _ShellIndex;
```

```
                    int _ShellCount;
                    float _ShellLength;
                    float _Density;
                    float _NoiseMin, _NoiseMax;
                    float _Thickness;
                    float _Attenuation;
                    float _OcclusionBias;
                    float _ShellDistanceAttenuation;
                    float _Curvature;
                    float _DisplacementStrength;
                    float3 _ShellColor;
                    float3 _ShellDirection;
                    CBUFFER_END

                    float hash(uint n)
                    {
                            // integer hash copied from Hugo Elias
                            n = (n << 13U) ^ n;
                            n = n * (n * n * 15731U + 0x789221U) + 0x1376312589U;
                            return float(n & uint(0x7fffffffU)) / float(0x7fffffff);
                    }


                    //The Vertex shader
                    Varyings vert(Attributes IN)
                    {
                            Varyings OUT;

                            float shellHeight = (float)_ShellIndex / (float)_ShellCount;

                            shellHeight = abs(pow(shellHeight,
_ShellDistanceAttenuation));

                            IN.vertex.xyz += IN.normal.xyz * _ShellLength * shellHeight;

                OUT.normal = normalize(TransformObjectToWorldNormal(IN.normal));

                            float k = pow(shellHeight, _Curvature);

                            IN.vertex.xyz += _ShellDirection * k * _DisplacementStrength;

                            OUT.pos = TransformObjectToHClip(IN.vertex);
                OUT.uv = IN.uv;

                            return OUT;
                    }

                    half4 frag(Varyings IN) : SV_Target
                    {
                            float2 newUV = IN.uv * _Density;
                            float2 localUV = frac(newUV) * 2 - 1;
                            float localDistanceFromCenter = length(localUV);

                            uint2 tid = newUV;
                            uint seed = tid.x + 100 * tid.y + 100 * 10;

                            // instead of (float)_ShellIndex && (float)_ShellCount
                float shellIndex = _ShellIndex;
                float shellCount = _ShellCount;

                            //Lerp between min and max noise according to random generator
function
                float rand = lerp(_NoiseMin, _NoiseMax, hash(seed));

                            // This is the normalized shell height as in the vertex shader
                float h = shellIndex / shellCount;

                            int outsideThickness = (localDistanceFromCenter) > (_Thickness
* (rand - h));

                            if (outsideThickness && _ShellIndex > 0)
                            {
                                   discard;
                            }

                            float ndotl = clamp(dot(IN.normal, _MainLightPosition),0,1) *
0.5f + 0.5f;

                            ndotl = ndotl * ndotl;

                            float ambientOcclusion = pow(h, _Attenuation);
                            ambientOcclusion += _OcclusionBias;
                            ambientOcclusion = saturate(ambientOcclusion);

                            return float4(_ShellColor * ndotl * ambientOcclusion, 1.0);

                     }
                     ENDHLSL
              }
       }
       }
```

Appendix E: SimpleShell.cs code

```
using System;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.InputSystem;

public class SimpleShell : MonoBehaviour {
    public Mesh shellMesh;
    public Shader shellShader;

    public bool updateStatics = true;

    [Range(1, 256)] public int shellCount = 16;
    [Range(0.0f, 1.0f)] public float shellLength = 0.15f;
    [Range(0.01f, 3.0f)] public float distanceAttenuation = 1.0f;
    [Range(1.0f, 1000.0f)] public float density = 100.0f;
    [Range(0.0f, 1.0f)] public float noiseMin = 0.0f;
    [Range(0.0f, 1.0f)] public float noiseMax = 1.0f;
    [Range(0.0f, 10.0f)] public float thickness = 1.0f;
    [Range(0.0f, 1.0f)] public float curvature = 1.0f;
    [Range(0.0f, 1.0f)] public float displacementStrength = 0.1f;
    [Range(0.0f, 5.0f)] public float occlusionAttenuation = 1.0f;
    [Range(0.0f, 1.0f)] public float occlusionBias = 0.0f;
    public Color shellColor;

    private Material shellMaterial;
    private GameObject[] shells;

    public Vector3 direction = new Vector3(0, 0, 0);
    private Vector3 displacementDirection = new Vector3(0, 0, 0);

    void OnEnable() {
        shellMaterial = new Material(shellShader);

        shells = new GameObject[shellCount];

        for (int i = 0; i < shellCount; ++i) {
            shells[i] = new GameObject("Shell " + i.ToString());
            shells[i].AddComponent<MeshFilter>();
            shells[i].AddComponent<MeshRenderer>();

            shells[i].GetComponent<MeshFilter>().mesh = shellMesh;
            shells[i].GetComponent<MeshRenderer>().material = shellMaterial;
            shells[i].transform.SetParent(this.transform, false);

            shells[i].GetComponent<MeshRenderer>().material.SetInt("_ShellCount",
shellCount);
            shells[i].GetComponent<MeshRenderer>().material.SetInt("_ShellIndex", i);
            shells[i].GetComponent<MeshRenderer>().material.SetFloat("_ShellLength",
shellLength);
            shells[i].GetComponent<MeshRenderer>().material.SetFloat("_Density",
density);
            shells[i].GetComponent<MeshRenderer>().material.SetFloat("_Thickness",
thickness);
```

```
            shells[i].GetComponent<MeshRenderer>().material.SetFloat("_Attenuation",
occlusionAttenuation);

shells[i].GetComponent<MeshRenderer>().material.SetFloat("_ShellDistanceAttenuation",
distanceAttenuation);
            shells[i].GetComponent<MeshRenderer>().material.SetFloat("_Curvature",
curvature);

shells[i].GetComponent<MeshRenderer>().material.SetFloat("_DisplacementStrength",
displacementStrength);
            shells[i].GetComponent<MeshRenderer>().material.SetFloat("_OcclusionBias",
occlusionBias);
            shells[i].GetComponent<MeshRenderer>().material.SetFloat("_NoiseMin",
noiseMin);
            shells[i].GetComponent<MeshRenderer>().material.SetFloat("_NoiseMax",
noiseMax);
            shells[i].GetComponent<MeshRenderer>().material.SetVector("_ShellColor",
shellColor);
        }
    }

    void Update() {
        float velocity = 1.0f;
        Vector3 currentPosition = this.transform.position;
        direction.Normalize();
        currentPosition += direction * velocity * Time.deltaTime;
        this.transform.position = currentPosition;

        displacementDirection -= direction * Time.deltaTime * 10.0f;
        if (direction == Vector3.zero)
            displacementDirection.y -= 10.0f * Time.deltaTime;

        if (displacementDirection.magnitude > 1) displacementDirection.Normalize();

        if (updateStatics) {
            for (int i = 0; i < shellCount; ++i) {
                shells[i].GetComponent<MeshRenderer>().material.SetInt("_ShellCount",
shellCount);
                shells[i].GetComponent<MeshRenderer>().material.SetInt("_ShellIndex", i);
                shells[i].GetComponent<MeshRenderer>().material.SetFloat("_ShellLength",
shellLength);
                shells[i].GetComponent<MeshRenderer>().material.SetFloat("_Density",
density);
                shells[i].GetComponent<MeshRenderer>().material.SetFloat("_Thickness",
thickness);
                shells[i].GetComponent<MeshRenderer>().material.SetFloat("_Attenuation",
occlusionAttenuation);

shells[i].GetComponent<MeshRenderer>().material.SetFloat("_ShellDistanceAttenuation",
distanceAttenuation);
                shells[i].GetComponent<MeshRenderer>().material.SetFloat("_Curvature",
curvature);

shells[i].GetComponent<MeshRenderer>().material.SetFloat("_DisplacementStrength",
displacementStrength);
shells[i].GetComponent<MeshRenderer>().material.SetFloat("_OcclusionBias",
occlusionBias);
                shells[i].GetComponent<MeshRenderer>().material.SetFloat("_NoiseMin",
noiseMin);
                shells[i].GetComponent<MeshRenderer>().material.SetFloat("_NoiseMax",
noiseMax);
                shells[i].GetComponent<MeshRenderer>().material.SetVector("_ShellColor",
shellColor);

shells[i].GetComponent<MeshRenderer>().material.SetVector("_ShellDirection",
displacementDirection);
            }
        }
    }

    void OnDisable() {
        for (int i = 0; i < shells.Length; ++i) {
            Destroy(shells[i]);
        }

        shells = null;
    }

    public void OnMove(InputValue value)
    {
        Vector2 movement = value.Get<Vector2>();
        direction.x = movement.x;
        direction.z = movement.y;
    }
    public void OnElevate(InputValue value)
    {
        float movement = value.Get<float>();
        direction.y = movement;
    }
}
```

Appendix F: SimpleCameraController.cs code

```
using UnityEngine;

namespace UnityTemplateProjects
{
    public class SimpleCameraController : MonoBehaviour
    {
        void Update()
        {
            Vector3 direction = new Vector3();
            if (Input.GetKey(KeyCode.W))
            {
                direction.z += Time.deltaTime;
            }
            if (Input.GetKey(KeyCode.S))
            {
                direction.z -= Time.deltaTime;
            }
            if (Input.GetKey(KeyCode.A))
            {
                direction.x -= Time.deltaTime;
            }
            if (Input.GetKey(KeyCode.D))
            {
                direction.x += Time.deltaTime;
            }
            if (Input.GetKey(KeyCode.Q))
            {
                direction.y -= Time.deltaTime;
            }
            if (Input.GetKey(KeyCode.E))
            {
                direction.y += Time.deltaTime;
            }
            transform.position += direction.normalized * Time.deltaTime;

            if (Input.GetKey(KeyCode.Escape))
            {
                Application.Quit();
                            #if UNITY_EDITOR
                            UnityEditor.EditorApplication.isPlaying = false;
                            #endif
            }
        }
    }
       }
```

Appendix G: HLSLGeom.shader code

```
Shader "Custom/HLSLGeom" {
    Properties
    {

        [Header(Shell Property)][Space]
        [IntRange] _FurLayers("NumberOfShells", Range(2, 24)) = 24
        _ShellSize("Shell Size", range(0.01, 1.0)) = 1.0
        _ShellDirection("Shell Direction", vector) = (0, 0, 0, 0)
        _DisplacementStrength("Displacement Strenght", range(0.0, 0.99)) = 0.5

        [Header(Shell Color)][Space]
        _BaseColor("Base color", Color) = (0, 0.5, 0, 1) // Color of the lowest layer
        _TopColor("Top color", Color) = (0, 1, 0, 1) // Color of the highest layer
        _DetailTextureA("Detail Texture A", 2D) = "white" {} // Texture A used to clip
layers
        _TextureInfluenceA("Texture A Influence", Range(0, 1)) = 1 // The influence of
Texture A
        _DetailTextureB("Detail Texture B", 2D) = "white" {} // Texture B used to clip
layers
        _TextureInfluenceB("Texture B Influence", Range(0, 1)) = 1 // The influence of
Texture B

        [Header(Shell Basics)][Space]
        _occlusion("Occulsion", Range(0, 1)) = 0
        _smoothness("Roughness", Range(0, 1)) = 0
        _metallic("Metalness", Range(0, 1)) = 0
        _emission("Emissive", Color) = (0, 0, 0, 0)
    }

    SubShader {


        // URP prerequisites
        Tags{"RenderType" = "Opaque" "RenderPipeline" = "UniversalPipeline"
"IgnoreProjector" = "True"}

        // Forward Lit Pass
        Pass {

            Name "ForwardLit"
            Tags{"LightMode" = "UniversalForward"}
            Cull Back

            HLSLPROGRAM
            // Signal this shader requires geometry function support
            #pragma prefer_hlslcc gles
            #pragma exclude_renderers d3d11_9x
            #pragma target 2.0
            #pragma require geometry

            // Lighting and shadow keywords
            #pragma multi_compile _ _MAIN_LIGHT_SHADOWS
            #pragma multi_compile _ _MAIN_LIGHT_SHADOWS_CASCADE
            #pragma multi_compile _ _ADDITIONAL_LIGHTS
            #pragma multi_compile _ _ADDITIONAL_LIGHT_SHADOWS
            #pragma multi_compile _ _SHADOWS_SOFT

            // Register our functions
            #pragma vertex Vertex
            #pragma geometry Geometry
            #pragma fragment Fragment

            // Incude our logic file
            #include "Fur.hlsl"

            ENDHLSL
        }

        Pass
{

            Name "ShadowCaster"
            Tags {"LightMode" = "ShadowCaster"}

            HLSLPROGRAM
            // Signal this shader requires geometry function support
            #pragma prefer_hlslcc gles
            #pragma exclude_renderers d3d11_9x
            #pragma target 2.0
            #pragma require geometry

            // Support all the various light types and shadow paths
            #pragma multi_compile_shadowcaster

            // Register our functions
            #pragma vertex Vertex
            #pragma geometry Geometry
            #pragma fragment Fragment

            // A custom keyword to modify logic during the shadow caster pass
            #define SHADOW_CASTER_PASS
            // Incude our logic file
            #include "Fur.hlsl"

            ENDHLSL
        }
    }
}
```

Appendix H: Fur.hlsl code

```
#include "Packages/com.unity.render-pipelines.universal/ShaderLibrary/Lighting.hlsl"
#include "Common.hlsl"

struct Attributes {
    float4 positionOS   : POSITION;
    float3 normalOS     : NORMAL;
    float4 tangentOS    : TANGENT;
    float2 uv           : TEXCOORD0;
};

struct VertexOutput {
    float3 positionWS   : TEXCOORD0;
    float3 normalWS     : TEXCOORD1;
    float2 uv           : TEXCOORD2;
};

struct GeometryOutput {
    float3 uv           : TEXCOORD0;
    float3 positionWS   : TEXCOORD1;
    float3 normalWS     : TEXCOORD2;
    float4 positionCS   : SV_POSITION;
};

// Properties
//Values used to change "transform" of the Shells
float _ShellSize;
float4 _ShellDirection;
float _DisplacementStrength;
float _Curvature;

//Color properties
float4 _BaseColor;
float4 _TopColor;
// These two textures are combined to create the fur pattern in the fragment function
TEXTURE2D(_DetailTextureA); SAMPLER(sampler_DetailTextureA); float4 _DetailTextureA_ST;
float _TextureInfluenceA;
TEXTURE2D(_DetailTextureB); SAMPLER(sampler_DetailTextureB); float4 _DetailTextureB_ST;
float _TextureInfluenceB;

half _metallic;
half3 _specular;
half _smoothness;
half _occlusion;
half3 _emission;
half _alpha;

uint _FurLayers;


// Vertex functions
VertexOutput Vertex(Attributes IN)
{
    // Initialize the output struct
    VertexOutput output = (VertexOutput)0;
    // Calculate position and normal in world space
    VertexPositionInputs vertexInput = GetVertexPositionInputs(IN.positionOS.xyz);
    VertexNormalInputs normalInput = GetVertexNormalInputs(IN.normalOS, IN.tangentOS);
    output.positionWS = vertexInput.positionWS;
    output.normalWS = normalInput.normalWS;

    // Pass through the UV
    output.uv = IN.uv;
    return output;
}

// Geometry functions

// This function sets values in output after calculating position based on height
void SetupVertex(in VertexOutput input, inout GeometryOutput output, float height)
{

    // Extrude the position from the normal based on the height
    float3 positionWS = input.positionWS + input.normalWS * (height * _ShellSize);

    //Move shell positions according to the _ShellDirection higher layers are more
impacted by the height
    positionWS -= _ShellDirection * (height * _ShellSize * _DisplacementStrength);

    output.positionWS = positionWS;

    output.normalWS = input.normalWS;
    output.uv = float3(input.uv, height); // Store the layer height in uv.z
    output.positionCS = CalculatePositionCSWithShadowCasterLogic(positionWS,
input.normalWS);
}

// 3 vertices per trianlge * max number of layers allowed by Unity
[maxvertexcount(3 * 24)]
void Geometry(triangle VertexOutput inputs[3], inout TriangleStream<GeometryOutput>
outputStream)
{
    // Initialize the output struct
    GeometryOutput output = (GeometryOutput)0;

    //For each layer
    for (int l = 0; l < _FurLayers; l++) {
        // The height percent
        float h = l / (float)(_FurLayers - 1);
        // For each point in the triangle
        for (int t = 0; t < 3; t++) {
            // Calculate the output data and add the vertex to the output stream
            SetupVertex(inputs[t], output, h);
            outputStream.Append(output);
        }
        // Each triangle is disconnected, so we need to call this to restart the triangle
strip
        outputStream.RestartStrip();
    }
}
```

```
// Fragment functions

half4 Fragment(GeometryOutput input) : SV_Target
{

    float2 uv = input.uv.xy;
    float height = input.uv.z;

    // Sample the two noise textures, applying their scale and offset
    float detailNoiseA = SAMPLE_TEXTURE2D(_DetailTextureA, sampler_DetailTextureA,
TRANSFORM_TEX(uv, _DetailTextureA)).r;
    float detailNoiseB = SAMPLE_TEXTURE2D(_DetailTextureB, sampler_DetailTextureB,
TRANSFORM_TEX(uv, _DetailTextureB)).r;
    // Combine the textures together using these scale variables. Lower values will
reduce a texture's influence
    detailNoiseA = 1 - (1 - detailNoiseA) * _TextureInfluenceA;
    detailNoiseB = 1 - (1 - detailNoiseB) * _TextureInfluenceB;
    // If detailNoise * smoothNoise is less than height, this pixel will be discarded by
the renderer
    clip(detailNoiseA * detailNoiseB - height);

    // If the code reaches this far, this pixel should render

#ifdef SHADOW_CASTER_PASS
    // If we're in the shadow caster pass, it's enough to return now. We don't care about
color
    return 0;
#else
    // Gather some data for the lighting algorithm
    InputData lightingInput = (InputData)0;
    lightingInput.positionWS = input.positionWS;
    lightingInput.normalWS = NormalizeNormalPerPixel(input.normalWS); // Renormalize the
normal to reduce interpolation errors
    lightingInput.viewDirectionWS = GetViewDirectionFromPosition(input.positionWS); //
Calculate the view direction
    lightingInput.shadowCoord = CalculateShadowCoord(input.positionWS, input.positionCS);
// Calculate the shadow map coord

    // Lerp between the two grass colors based on layer height
    float3 _diffuseColor = lerp(_BaseColor, _TopColor, height).rgb;

    //InputData inputData,
    //half3 albedo,
    //half metallic,
    //half3 specular,
    //half smoothness,
    //half occlusion,
    //half3 emission,
    //half alpha
    //return UniversalFragmentPBR(lightingInput, _diffuseColor, _metallic, _specular,
_smoothness, _occlusion, _emission, _alpha);
    return UniversalFragmentPBR(lightingInput, _diffuseColor, _metallic, _specular,
_smoothness, _occlusion, _emission, 1);

#endif
       }
```

Appendix I: Common.hlsl code

```
#ifndef COMMON
#define COMMON

// Returns the view direction in world space
float3 GetViewDirectionFromPosition(float3 positionWS) {
    return normalize(GetCameraPositionWS() - positionWS);
}

// URP Helpers

// If this is the shadow caster pass, we also need this variable, which URP sets
#ifdef SHADOW_CASTER_PASS
float3 _LightDirection;
#endif

// Calculates the position in clip space
float4 CalculatePositionCSWithShadowCasterLogic(float3 positionWS, float3 normalWS) {
    float4 positionCS;

#ifdef SHADOW_CASTER_PASS
    // From URP's ShadowCasterPass.hlsl
    positionCS = TransformWorldToHClip(ApplyShadowBias(positionWS, normalWS,
_LightDirection));
#if UNITY_REVERSED_Z
    positionCS.z = min(positionCS.z, positionCS.w * UNITY_NEAR_CLIP_VALUE);
#else
    positionCS.z = max(positionCS.z, positionCS.w * UNITY_NEAR_CLIP_VALUE);
#endif
#else
    // This built in function transforms from world space to clip space
    positionCS = TransformWorldToHClip(positionWS);
#endif

    return positionCS;
}

// Calculates the shadow texture coordinate for lighting calculations
float4 CalculateShadowCoord(float3 positionWS, float4 positionCS) {
    // Calculate the shadow coordinate depending on the type of shadows currently in use
#if SHADOWS_SCREEN
    return ComputeScreenPos(positionCS);
#else
    return TransformWorldToShadowCoord(positionWS);
#endif
}

       #endif
```

Appendix J: HLSLComplex.shader code

```
Shader "Custom/HLSLComplex" {
    Properties {

        [Header(Shell Property)][Space]
        [IntRange] _FurLayers("NumberOfShells", Range(2, 24)) = 24
        _ShellSize("Shell Size", range(0.01, 1.0)) = 1.0
        _ShellDirection("Shell Direction", vector) = (0, 0, 0, 0)
        _DisplacementStrength("Displacement Strenght", range(0.0, 0.99)) = 0.5


        [Space(20)][Header(Shell Parameters)][Space]
        _DetailTextureA("Detail Texture A", 2D) = "white" {} // Texture A used to clip 
layers
        _TextureInfluenceA("Texture A Influence", Range(0, 1)) = 1 // The influence of 
Texture A
        _DetailTextureB("Detail Texture B", 2D) = "white" {} // Texture B used to clip 
layers
        _TextureInfluenceB("Texture B Influence", Range(0, 1)) = 1 // The influence of 
Texture B

        [Space(20)][Header(Object Material)][Space(10)]
        [MainTexture] _BaseColor("Base Color", 2D) = "white" {}
        _TintColor("Tint Color", Color) = (0, 0.5, 0, 1)
        _TopColor("Top Color", Color) = (0, 1, 0, 1)
        _Occlusion("Occulsion", 2D) = "white" {}
        _OcclusionValue("Occulsion Value", Range(0, 1)) = 0
        _Smoothness("Roughness",  2D) = "white" {}
        _SmoothnessValue("Roughness Value", Range(0, 1)) = 0
        _Metalness("Metalness", 2D) = "white" {}
        _MetalnessValue("Metalness Value", Range(0, 1)) = 0
        _Emissive("Emissive", 2D) = "white" {}
        _EmissionTint("Emission Tint", Color) = (0, 0, 0, 0)

    }

    SubShader {

        Tags{"RenderType" = "Opaque" "RenderPipeline" = "UniversalPipeline" 
"IgnoreProjector" = "True"}

        // Forward Lit Pass
        Pass {

            Name "ForwardLit"
            Tags{"LightMode" = "UniversalForward"}
            Cull Back

            HLSLPROGRAM
            // Signal this shader requires geometry function support
            #pragma prefer_hlslcc gles
            #pragma exclude_renderers d3d11_9x
            #pragma target 2.0
            #pragma require geometry
            // Lighting and shadow keywords
            #pragma multi_compile _ _MAIN_LIGHT_SHADOWS
            #pragma multi_compile _ _MAIN_LIGHT_SHADOWS_CASCADE
            #pragma multi_compile _ _ADDITIONAL_LIGHTS
            #pragma multi_compile _ _ADDITIONAL_LIGHT_SHADOWS
            #pragma multi_compile _ _SHADOWS_SOFT

            // Register our functions
            #pragma vertex Vertex
            #pragma geometry Geometry
            #pragma fragment Fragment

            // Incude our logic file
            #include "FurComplex.hlsl"

            ENDHLSL
        }

        Pass {

            Name "ShadowCaster"
            Tags {"LightMode" = "ShadowCaster"}

            HLSLPROGRAM
            // Signal this shader requires geometry function support
            #pragma prefer_hlslcc gles
            #pragma exclude_renderers d3d11_9x
            #pragma target 2.0
            #pragma require geometry

            // Support all the various light types and shadow paths
            #pragma multi_compile_shadowcaster

            // Register our functions
            #pragma vertex Vertex
            #pragma geometry Geometry
            #pragma fragment Fragment

            // A custom keyword to modify logic during the shadow caster pass
            #define SHADOW_CASTER_PASS

            #include "FurComplex.hlsl"

            ENDHLSL
        }
    }
       }
```

Appendix K: FurComplex.hlsl code

```
#include "Packages/com.unity.render-pipelines.universal/ShaderLibrary/Lighting.hlsl"
#include "Common.hlsl"

struct Attributes {
    float4 positionOS   : POSITION;
    float3 normalOS     : NORMAL;
    float4 tangentOS    : TANGENT;
    float2 uv           : TEXCOORD0;
};

struct VertexOutput {
    float3 positionWS   : TEXCOORD0;
    float3 normalWS     : TEXCOORD1;
    float2 uv           : TEXCOORD2;
};

struct GeometryOutput {
    float3 uv           : TEXCOORD0;
    float3 positionWS   : TEXCOORD1;
    float3 normalWS     : TEXCOORD2;
    float4 positionCS   : SV_POSITION;
};

// Properties
float _ShellSize;
float4 _ShellDirection;
float _DisplacementStrength;
float _Curvature;

//Color properties
TEXTURE2D(_BaseColor); SAMPLER(sampler_BaseColor); float4 _BaseColor_ST;
float4 _TintColor;
float4 _TopColor;
half3 _Specular;
TEXTURE2D(_Occlusion); SAMPLER(sampler_Occlusion); float4 _Occlusion_ST;
half _OcclusionValue;
TEXTURE2D(_Smoothness); SAMPLER(sampler_Smoothness); float4 _Smoothness_ST;
half _SmoothnessValue;
TEXTURE2D(_Metalness); SAMPLER(sampler_Metalness); float4 _Metalness_ST;
half _MetalnessValue;
TEXTURE2D(_Emissive); SAMPLER(sampler_Emissive); float4 _Emissive_ST;
half3 _EmissionTint;

// Textures for the fur pattern
TEXTURE2D(_DetailTextureA); SAMPLER(sampler_DetailTextureA); float4 _DetailTextureA_ST;
float _TextureInfluenceA;
TEXTURE2D(_DetailTextureB); SAMPLER(sampler_DetailTextureB); float4 _DetailTextureB_ST;
float _TextureInfluenceB;

uint _FurLayers;


VertexOutput Vertex(Attributes IN) {
    VertexOutput output = (VertexOutput)0;

    VertexPositionInputs vertexInput = GetVertexPositionInputs(IN.positionOS.xyz);
    VertexNormalInputs normalInput = GetVertexNormalInputs(IN.normalOS, IN.tangentOS);
    output.positionWS = vertexInput.positionWS;
    output.normalWS = normalInput.normalWS;

    output.uv = IN.uv;
    return output;
}

void SetupVertex(in VertexOutput input, inout GeometryOutput output, float height) {

    float3 positionWS = input.positionWS + input.normalWS * (height * _ShellSize);

    positionWS -= _ShellDirection * (height * _ShellSize * _DisplacementStrength);

    output.positionWS = positionWS;

    output.normalWS = input.normalWS;
    output.uv = float3(input.uv, height);
    output.positionCS = CalculatePositionCSWithShadowCasterLogic(positionWS,
input.normalWS);
}

[maxvertexcount(3 * 24)]
void Geometry(triangle VertexOutput inputs[3], inout TriangleStream<GeometryOutput>
outputStream) {

    GeometryOutput output = (GeometryOutput)0;

    for (int l = 0; l < _FurLayers; l++) {
        float h = l / (float)(_FurLayers - 1);
        for (int t = 0; t < 3; t++)
        {
            SetupVertex(inputs[t], output, h);
            outputStream.Append(output);
        }

        outputStream.RestartStrip();
    }
}

half4 Fragment(GeometryOutput input) : SV_Target {

    float2 uv = input.uv.xy;
    float height = input.uv.z;

    float detailNoiseA = SAMPLE_TEXTURE2D(_DetailTextureA, sampler_DetailTextureA,
TRANSFORM_TEX(uv, _DetailTextureA)).r;
    float detailNoiseB = SAMPLE_TEXTURE2D(_DetailTextureB, sampler_DetailTextureB,
TRANSFORM_TEX(uv, _DetailTextureB)).r;

    detailNoiseA = 1 - (1 - detailNoiseA) * _TextureInfluenceA;
    detailNoiseB = 1 - (1 - detailNoiseB) * _TextureInfluenceB;

    clip(detailNoiseA * detailNoiseB - height);
```

```
#ifdef SHADOW_CASTER_PASS
    return 0;
#else

    InputData lightingInput = (InputData)0;
    lightingInput.positionWS = input.positionWS;
    lightingInput.normalWS = NormalizeNormalPerPixel(input.normalWS);
    lightingInput.viewDirectionWS = GetViewDirectionFromPosition(input.positionWS);
    lightingInput.shadowCoord = CalculateShadowCoord(input.positionWS, input.positionCS);

    float3 _diffuseColor = lerp(
        SAMPLE_TEXTURE2D(_BaseColor, sampler_BaseColor, TRANSFORM_TEX(uv, _BaseColor)) *
_TintColor,
        SAMPLE_TEXTURE2D(_BaseColor, sampler_BaseColor, TRANSFORM_TEX(uv, _BaseColor))
*_TopColor,
        height).rgb;
    half _occlusion = SAMPLE_TEXTURE2D(_Occlusion, sampler_Occlusion, TRANSFORM_TEX(uv,
_Occlusion)) * _OcclusionValue;
    half _smoothness = SAMPLE_TEXTURE2D(_Smoothness, sampler_Smoothness,
TRANSFORM_TEX(uv, _Smoothness)) * _SmoothnessValue;
    half _metallic = SAMPLE_TEXTURE2D(_Metalness, sampler_Metalness, TRANSFORM_TEX(uv,
_Metalness)) * _MetalnessValue;
    half3 _emissive = SAMPLE_TEXTURE2D(_Emissive, sampler_Emissive, TRANSFORM_TEX(uv,
_Emissive)) * _EmissionTint;

    //InputData inputData,
    //half3 albedo,
    //half metallic,
    //half3 specular,
    //half smoothness,
    //half occlusion,
    //half3 emission,
    //half alpha
    return UniversalFragmentPBR(lightingInput, _diffuseColor, _metallic, _Specular,
_smoothness, _occlusion, _emissive, 1);

#endif
       }
```

Appendix L: HLSLGeom_Controller.cs script

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.InputSystem;

public class HLSLGeom_Controller : MonoBehaviour
{
    private Vector3 displacementDirection = new Vector3(0, 0, 0);

    public Vector3 direction = new Vector3(0, 0, 0);

    public Material shellMaterial;

    // Start is called before the first frame update
    void Start()
    {
        shellMaterial = GetComponent<Renderer>().material;
    }

    // Update is called once per frame
    void Update()
    {
        float velocity = 1.0f;

        Vector3 currentPosition = this.transform.position;
        direction.Normalize();
        currentPosition += direction * velocity * Time.deltaTime;
        this.transform.position = currentPosition;

        displacementDirection += direction * Time.deltaTime * 10.0f;
        if (direction == Vector3.zero)
            displacementDirection.y += 10.0f * Time.deltaTime;

        if (displacementDirection.magnitude > 1) displacementDirection.Normalize();

        shellMaterial.SetVector("_ShellDirection", displacementDirection);
    }

    public void OnMove(InputValue value)
    {
        Vector2 movement = value.Get<Vector2>();
        direction.x = movement.x;
        direction.z = movement.y;
    }
    public void OnElevate(InputValue value)
    {
        float movement = value.Get<float>();
        direction.y = movement;
    }

}
```

Appendix M: HLSLComplex_Controller.cs script

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.InputSystem;

public class HLSLComplex_Controller : MonoBehaviour
{
    private Vector3 displacementDirection = new Vector3(0, 0, 0);

    public Vector3 direction = new Vector3(0, 0, 0);

    public List<Renderer> shellRenderers = new List<Renderer>();

    // Update is called once per frame
    void Update()
    {
        float velocity = 1.0f;

        Vector3 currentPosition = this.transform.position;
        direction.Normalize();
        currentPosition += direction * velocity * Time.deltaTime;
        this.transform.position = currentPosition;

        displacementDirection += direction * Time.deltaTime * 10.0f;
        if (direction == Vector3.zero)
            displacementDirection.y += 10.0f * Time.deltaTime;

        if (displacementDirection.magnitude > 1) displacementDirection.Normalize();

        foreach(Renderer render in shellRenderers)
        render.material.SetVector("_ShellDirection", displacementDirection);
    }

    public void OnMove(InputValue value)
    {
        Vector2 movement = value.Get<Vector2>();
        direction.x = movement.x;
        direction.z = movement.y;
    }
    public void OnElevate(InputValue value)
    {
        float movement = value.Get<float>();
        direction.y = movement;
    }

}
```

Appendix N: FurSceneManager.cs script

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.SceneManagement;
using TMPro;

public class FurSceneManager : MonoBehaviour
{
    [SerializeField] private TMP_Text m_PreviousSceneText;
    [SerializeField] private TMP_Text m_CurrentSceneText;
    [SerializeField] private TMP_Text m_NextSceneText;

    [SerializeField] private SceneData m_SceneData;

    public void IncreaseScene()
    {
        m_SceneData.SceneIndex = GetNextSceneIndex();

        UpdateTexts();
        LoadSceneAtIndex(m_SceneData.SceneIndex);
    }

    public void DecreaseScene()
    {
        m_SceneData.SceneIndex = GetPreviousSceneIndex();

        UpdateTexts();
        LoadSceneAtIndex(m_SceneData.SceneIndex);
    }

    private void LoadSceneAtIndex(int index)
    {
        SceneManager.LoadScene(index);
    }

    private int GetNextSceneIndex()
    {
        if (m_SceneData.SceneIndex >= SceneManager.sceneCountInBuildSettings-1)
            return 0;
        else
        {
            return m_SceneData.SceneIndex + 1;
        }
    }

    private int GetPreviousSceneIndex()
    {
        if (m_SceneData.SceneIndex <= 0)
            return SceneManager.sceneCountInBuildSettings-1;
        else
        {
            return m_SceneData.SceneIndex - 1;
        }
    }
    private void UpdateTexts()
    {
        m_PreviousSceneText.text = GetSceneName(GetPreviousSceneIndex());
        m_CurrentSceneText.text = GetSceneName(m_SceneData.SceneIndex);
        m_NextSceneText.text = GetSceneName(GetNextSceneIndex());
    }

    private string GetSceneName(int sceneIndex)
    {
        string sceneName = SceneUtility.GetScenePathByBuildIndex(sceneIndex).ToString();
        sceneName = sceneName.Remove(0, sceneName.LastIndexOf('/')+1);
        sceneName = sceneName.Remove(sceneName.LastIndexOf('.'));
        return sceneName;
    }

    private void Awake()
    {
        m_SceneData.SceneIndex = SceneManager.GetActiveScene().buildIndex;
        UpdateTexts();
    }
}
```

Appendix O: SO_SceneData.cs script

```
using UnityEngine;

[CreateAssetMenu(fileName = "SceneData", menuName = "ScriptableObjects/SceneData", order
= 1)]
public class SceneData : ScriptableObject
{
    public int SceneIndex;
       }
```

Appendix P: DataExtractor.cs script

```
using Unity.Profiling;
using UnityEngine;
using System.IO;
using UnityEditor;
using UnityEngine.InputSystem;
using UnityEngine.SceneManagement;

public class DataExtractor : MonoBehaviour
{

    FrameTiming[] m_FrameTimings = new FrameTiming[1];
    double[] m_gpuFrameTimes = new double[1000];

    bool isMeasuring = false;
    int m_DataCount = 0;


    string sceneName;

    [SerializeField] int m_DataToCollect = 1000;
    [SerializeField] TextAsset m_TextAsset;

    private void Start()
    {
        sceneName =
SceneUtility.GetScenePathByBuildIndex(SceneManager.GetActiveScene().buildIndex).ToString(
);
        sceneName = sceneName.Remove(0, sceneName.LastIndexOf('/') + 1);
        sceneName = sceneName.Remove(sceneName.LastIndexOf('.'));
    }
    void Update()
    {
        if (isMeasuring && m_DataCount < m_DataToCollect)
        {
            FrameTimingManager.CaptureFrameTimings();
            var ret = FrameTimingManager.GetLatestTimings((uint)m_FrameTimings.Length,
m_FrameTimings);
            if (ret > 0)
            {
                {
                    //Frame capture logic
                    //Debug.Log(m_FrameTimings[0].gpuFrameTime);
                    m_gpuFrameTimes[m_DataCount] = m_FrameTimings[0].gpuFrameTime;
                    m_DataCount++;
                }
            }
        }
        if(m_DataCount >= m_DataToCollect && isMeasuring)
        {
            EditorUtility.SetDirty(m_TextAsset);
            for (int i = 0; i < m_DataCount; i++)
            {
                File.AppendAllText(AssetDatabase.GetAssetPath(m_TextAsset), 
m_gpuFrameTimes[i].ToString() + "\n");
            }

            Debug.Log("Data for: " + sceneName + " taken");
            isMeasuring = false;
        }
    }

    public void OnMeasure()
    {
        isMeasuring = true;
        Debug.Log("Measuring started on scene: " + sceneName);
    }
       }
```

Appendix Q: Extracted Data spreadsheet

MP_Data.xlsx