Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Electronic Systems

Daniel Rahme

Physically Based Rendering on Mobile

Measuring Performance and Power

Master's thesis in Electronic Systems Design
Supervisor: Per Gunnar Kjeldsberg
Co-supervisor: Deepali Yemul
June 2023

NTNU
Norwegian University of
Science and Technology

Daniel Rahme

Physically Based Rendering on Mobile

Measuring Performance and Power

Master's thesis in Electronic Systems Design
Supervisor: Per Gunnar Kjeldsberg
Co-supervisor: Deepali Yemul
June 2023

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Electronic Systems

NTNU
Norwegian University of
Science and Technology

Daniel Rahme

Physically Based Rendering on Mobile

Measuring Performance and Power


Master thesis for the degree MSc in Electronic Systems Design

Trondheim, June 2023

NTNU
Innovation and Creativity

Norwegian University of Science and Technology

Faculty of Information Technology
and Electrical Engineering
Department of Electronic Systems

NTNU

Norwegian University of Science and Technology

Master thesis
for the degree of MSc in Electronic Systems Design

Faculty of Information Technology
and Electrical Engineering
Department of Electronic Systems

Supervisors
Deepali Yemul, Arm Norway AS
Per Gunnar Kjeldsberg, professor at NTNU

Abstract

Mobile phones are widely used and its hardware is now capable of rendering advanced graphics. Physically based rendering (PBR) is a technique which makes computer graphics look realistic and professional. PBR has been used by the movie industry for a long time and now it has become more prevalent in gaming as PBR can be rendered in real-time. The bidirectional reflectance distribution function (BRDF) is responsible for how light is reflected on a surface based on the material properties of the surface. Two BRDF shading models and two graphics APIs were measured and compared in this project in order to evaluate if the performance and energy consumption was suitable for mobile.

Google Filament is an open-source PBR engine which was used to create test cases for measuring power and performance on mobile. The BRDF of Unreal Engine 4 (UE4) was added into the Filament engine and compared to Filament's native BRDF. The graphics back-end APIs in Filament, OpenGLES and Vulkan, were compared.

The rendered image quality of UE4's BRDF implemented into Filament graphics engine was unsatisfactory and flawed. The image quality of using the Vulkan back-end was incorrect and indicated problems in the rendering, as the graphics API should not impact the render quality.

The results showed the implemented UE4 BRDF performed 6% worse than Filament's native BRDF. The Vulkan graphics back-end performed 50% worse than OpenGLES which was significant. The energy consumption of UE4's BRDF used 9% more energy than Filament's native BRDF and the Vulkan back-end used 32% more energy per frame than OpenGLES. The Vulkan back-end used less power and less of the central processing unit (CPU), but that was not enough to compensate for the low performance. Furthermore, the GPU memory bandwidth of the Vulkan back-end was twice of OpenGLES which indicated possible memory bottleneck. This was further evidence that the Vulkan back-end in Google Filament was incomplete.

Acknowledgement

I would first and foremost like to thank my excellent supervisors, professor Per Gunnar Kjeldsberg from NTNU and Deepali Yemul from Arm Norway AS, for their great support and guidance throughout the thesis work. A thanks to the external organization, Arm Norway AS and the Power, Performance and Area team in particular for their help.

My time in NTNU has been phenomenal and I would like to thank the university for the opportunity to study in Trondheim. Last but not least I would like to thank my mom.

Contents

List of abbreviations

API Application Programming Interface
APK Android Package Kit
AR Augmented Reality
GPU Graphics Processing Unit
CPU Central Processing Unit
PBR Physically Based Rendering
BRDF Bidirectional Reflectance Distribution Function
FPS Frames Per Second
UE4 Unreal Engine 4
V-Sync Vertical-Synchronization
VR Virtual Reality
DVFS Dynamic Voltage and Frequency Scaling

Chapter 1

Introduction

Better computer graphics has been a hot topic for a long time and the demand is ever increasing - especially now with mobile devices capable of running graphically intensive games and the rise of Virtual Reality and Augmented Reality [1]. In recent years, Physically Based Rendering (PBR) has gained popularity due to its photo-realistic look. It was first used by the movie industry but as technology has advanced PBR can now be rendered in real-time and thus suitable for gaming [2][3].

Using PBR, an object will be rendered by its physical attributes which are based on the real world. This is achieved by using the same light physics as in real life [4]. Mobile gaming is widely popular and today's hardware can support more advanced graphics. thus PBR has become relevant for mobile as it can add realism and improve the graphics of mobile games [3]. There has been a lot of research in this topic, however, for mobile phones the research is still lacking.

This thesis project was about PBR on mobile with focus on the physically based bidirectional reflectance distribution function (BRDF). The BRDF was responsible for calculating the realistic coloring of an object depending on the object's material properties and how light reflects on the object's surface [4].

A comparison of Vulkan and OpenGLES graphics application programming interfaces (API) was interesting due to the supposed reduced energy consumption and performance improvement that Vulkan offers [5][6].

1.1 Research Questions

The research question for the thesis was the following:

- What options are there for PBR on mobile?
- How does the underlying graphics application programming interface affect the performance and power?
- How do different PBR BRDFs compare in terms of image quality, performance and power on mobile?

1.2 Research Method Goals and Limitations

The research was conducted as follows:

- Perform a literature study on the topic of PBR with focus on shading models.
- Find or create a framework for PBR on mobile.
- Create test cases using different BRDFs and graphics APIs.
- Measure and compare the BRDFs and graphics APIs for performance, power and image quality.

The limitation for the research was the following:

- Simplistic and subjective rendered image quality analysis
- Use Android mobile device with an Arm Mali graphics processing unit (GPU)
- Due to limited time, only possible to compare two BRDFs

1.3 Main Contributions

The performance and power measurement results of Filament's PBR engine measured on a Asus ROG 6D mobile phone. The results covered in Chapter 4 showed that using Vulkan graphics API back-end gave a 50% reduced performance and 20% higher energy consumption than the OpenGLES graphics back-end. The implemented Unreal Engine 4 (UE4) BRDF performed worse by 6% and used 9% more energy than Filament's native BRDF. The implemented UE4 BRDF had faulty rendering which was clearly visible in the image quality analysis, see Figure 4.1 and 4.2.

A contribution to the official Google Filament repository was made by fixing a bug which enabled this thesis to create the test cases by loading the textures correctly. The details are described in Appendix B.1 and briefly discussed in Section 3.3.3.

Chapter 2

Theory

In order to understand how Physically Based rendering works and the experiment that was conducted in chapter 3, the following topics will be presented: light, computer graphics, physically based rendering, performance and power metrics, equipment and software tools and the last section will be previous work.

2.1 Light

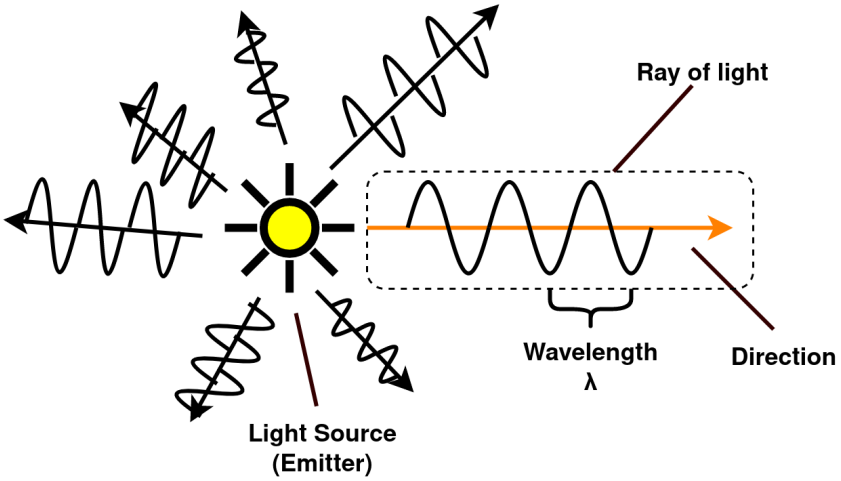
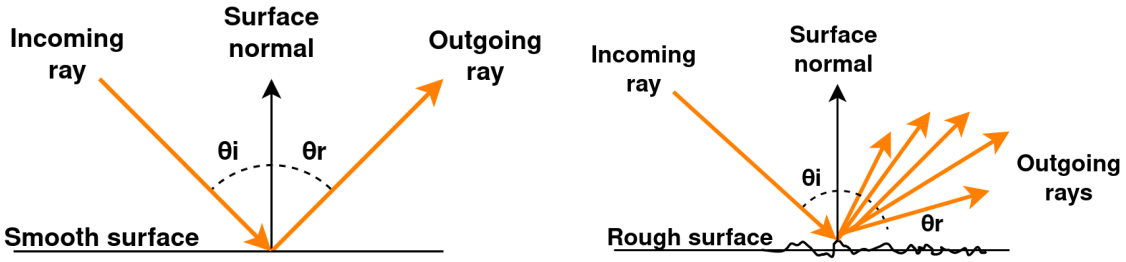


Figure 2.1: Light has a wavelength and a direction.

Light travels as a ray which has a direction from the light source and a wavelength. The wavelength of the light will determine the perceived color. The interesting properties of light in the context of this work is how it reflects of a surface as it varies depending on the material [7].



(a) Specular reflection

(b) Diffuse reflection

Figure 2.2: Light reflection and refraction. Illustrations inspired by J. F Blinn "Models of light reflection for computer synthesized pictures" [8].

When a ray of light hits a surface it can either reflect or refract. Specular reflection of light is when all incoming light is reflected off a smooth surface with the same angle which gives a shiny look. In other terms, when the incoming angle of incidence θ_i and the outgoing angle of reflectance θ_i are equal, see equation 2.1 and Figure 2.2a, this would be the case for a mirror [8].

$$\theta_i = \theta_r \tag{2.1}$$

Another type of reflection is called diffuse reflection which gives a matte look. This occurs when light hits a rough surface and scatters into many different directions than the angle of incidence θ_i, see Figure 2.2b. This means that the angle of incidence θ_i and reflectance θ_r are different from the surface normal [9].

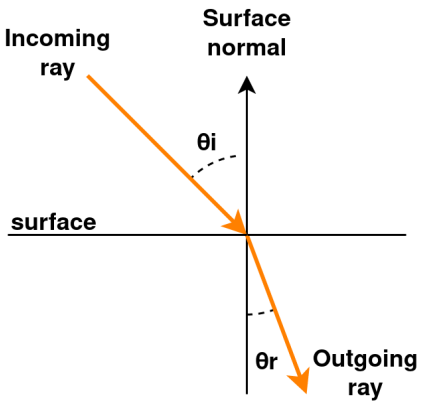


Figure 2.3: Refraction of light. Illustration inspired by J. F Blinn "Models of light reflection for computer synthesized pictures" [8].

Refraction is when a ray of light goes through the surface into the material instead of reflecting back. The light gets absorbed by the material and bends the light that travels inside the material [10], see Figure 2.3. An example would be an object put in water, as water refracts the light.

A combination of reflections and refraction can be had, i.e. light reflected on a surface can

have a specular and diffuse reflection or all three kinds with refraction. How specular and diffuse reflection can be used and combined will come in later Section 2.3.2.

2.2 Computer Graphics

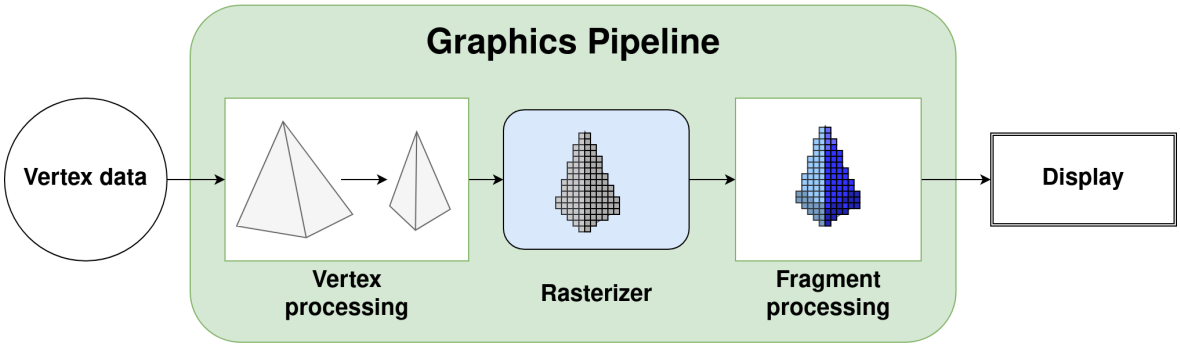


Figure 2.4: Simplified overview of a graphics pipeline. The process of rendering a frame can be broken down to a vertex processing stage and a fragment processing stage. Illustration inspired by M. Kenzel et al. "A high-performance software graphics pipeline architecture for the gpu" [11].

The graphics pipeline describes the process of rendering a frame from points in 3D-space onto a 2D-screen [11]. How it is done depends on the implementation of the graphics pipeline [12][13][14].

The two most commonly used primitives are triangles and lines. These primitives are made from vertices, where a vertex is a point in 3D-space (x, y, z). In computer graphics, a 3D-model is made up of triangles and lines forming a mesh [12][15].

Vertex processing operates on vertices in 3D-space. Examples of operations in this stage are rotations, enlargement or other transformations performed on 3D-models. These operations are written in a shader program which the graphics processing unit (GPU) executes. Shaders run in the vertex stage are known as vertex shaders [15][11].

The rasterizer takes the generated 3D-world of the vertex processing stage and transforms it into 2D-space. This is the process of going from vertices in 3D-space to pixels in 2D-space [11].

The fragment processing stage operates on the pixels. Usually this is where the colors (textures) are applied to the rendered frame, see Figure 2.4. This is also where the light and shadows are calculated. The fragment shaders are called by the GPU in this stage [11]. Things affecting the fragment shading are the rendering resolution and the textures. The rendering resolution will determine how many pixels will be rendered and the textures will be applied on to the pixels for coloring.

2.2.1 GPU and graphics API

A GPU is suitable for massive parallel computation and is often optimized for 3D graphics rendering and has proficient parallel computation capabilities for non-graphical applications [16].

A graphics API is a programming interface for graphics rendering. It is used to setup the graphics rendering pipeline and shaders used for the graphics application and is the interface for interacting with the GPU [11].

The most used and well known graphics application programming interface (API) used for Android is Open Graphics Library for Embedded Systems (OpenGLES) created by the Khronos group [17][18].

Vulkan is the newer graphics API also created by the Khronos group and can do both graphics rendering and compute. This API is explicit and the developer has to configure everything as there are no defaults. Furthermore, the developer is responsible for memory management, synchronization and run-time error checking which is not the case with OpenGLES [19]. The CPU uses a lot of energy and the Vulkan approach reduces the CPU usage, thus making it more energy efficient [6][20].

2.2.2 Graphics rendering engines

A graphics API is the interface for communicating with the GPU, as previously mentioned. It is tedious to write the same application for multiple graphics API. A graphics engine can be used to make the graphics application code portable. A graphics engine has 3D-rendering capabilities which supports multiple graphics API and acts as a layer of abstraction between the application and the underlying graphics API and rendering pipeline [21], see Figure 2.4.

A popular game engine is Unreal Engine which has appealing and advanced looking graphics. This game engine has been used for many known video games [22][23]. Another very popular game engine is Unity. It is cross-platform and supports many graphics API [24].

Google Filament is an open-source PBR real-time renderer which is cross-platform but with focus on Android. It supports platforms such as Android, Linux, Windows and iOS. It has support for graphics APIs such as OpenGL, Vulkan and Metal. Filament strives to be as efficient and lightweight as possible which makes it suitable for mobile. Filament is written in the programming language C++ and has supported API for C++ and Java [25].

2.3 Physically Based Rendering

This section will give an introduction to the topic of physically based rendering. The things that will be discussed is the rendering equation, the bidirectional reflectance distribution function, PBR textures and the history and use cases of PBR.

2.3.1 The rendering equation

The rendering equation solves the outgoing radiance of a surface, see Equation 2.2. In computer graphics, the radiance that the rendering equation solves is converted to RGB color for one pixel [26]. Therefore the rendering equation calculates the shading of the surface depending on the material and how the light reflects on the surface.

What makes this equation difficult to solve is due to the infinite recursion. As light bounces from object to object, the incoming light also has its own equation to solve [27].

The rendering equation was introduced by James Kajiya and David Immel et al in 1986 [28],

see Equation 2.2. Where L_o is outgoing radiance, L_e is emitted radiance, L_i is the incoming radiance, p is position, Ω is the surface area of a hemisphere, f_r is the BRDF function covered in Section 2.3.2, $n \cdot \omega_i$ is the surface normal times angle of incoming light, and $d\omega_i$ is the integrand over the incoming light.

$$L_o(p, \omega_o) = L_e + \int_\Omega f_r(p, \omega_i, \omega_o) L_i(p, \omega_i) \cdot n \cdot \omega_i \, d\omega_i \tag{2.2}$$

A general simplification of the rendering equation is to remove the emitted radiance L_e as it is trivial for computer graphics. The finite integral is replaced by a discrete Monte-Carlo integration, see Equation 2.3.

$$L_o(p, \omega_o) = \sum_\Omega f_r(p, \omega_i, \omega_o) L_i(p, \omega_i) \cdot n \cdot \omega_i \, d\omega_i \tag{2.3}$$

2.3.2 Bidirectional reflectance distribution function

The BRDF is the function which calculates how light reflects on an object based on its material properties [4]. From the rendering equation, see Equation 2.2, the BRDF function f_r has a specular and diffuse reflection component, see Equation 2.4.

$$f_r = f_{diffuse} + f_{specular} \tag{2.4}$$

A physically based BRDF needs to respect the following four principals [29]:

- Energy conservation
- Helmholtz reciprocity principle
- Microfacet model
- Fresnel effect

Energy conversation means that it cannot be more power reflected or absorbed than the incoming power [4]. The Helmholtz reciprocity principle states that light travels the same path from both directions [30]. The microfacet model and the Fresnel effect will be covered in this section.

Diffuse reflection

The diffuse reflection has a mate look which is the opposite of shiny, i.e wood or rubber. A common model for calculating diffuse lighting is the Lambertian diffuse model [29], see Equation 2.5. Diffuse reflection has been previously presented in Section 2.1.

$$f_{diffuse} = \frac{color}{\pi} \tag{2.5}$$

Specular reflection

The specular reflection function $f_{specular}$ is more complex and has two main parts, the microfacet model and the Fresnel effect.

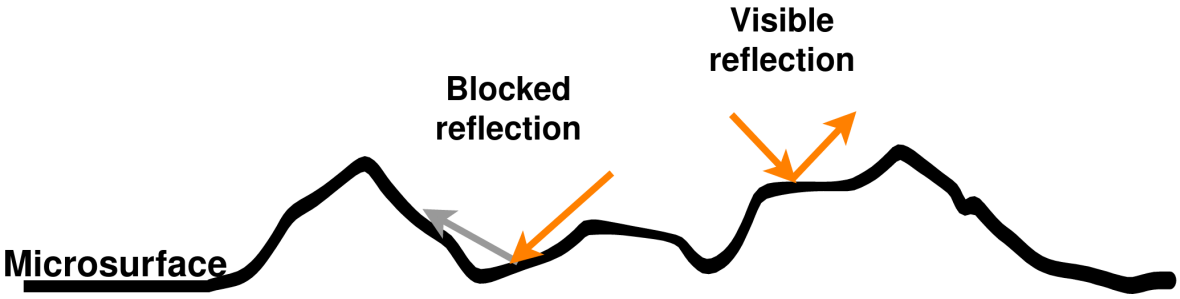


Figure 2.5: Microfacet theory showcasing blocked reflection. Illustration inspired by Torrance et al. "Microfacet Models for Refraction through Rough Surfaces" [31]

The microfacet model is used to calculate how light reflects on a rough surface. By modeling the surface at a microscopic level, light can either reflect or be blocked by the surface, see Figure 2.5. The blocked reflection is called shadowing-masking geometry [31]. A well known model for specular reflection is the Cook-Torrence microfacet model which has a distribution function (D) and a geometric visibility function (G) [29].

The Fresnel effect gives an object a glossy specular reflection when viewed from a low angle. If viewed from above, no glossy reflection would be seen [4], see Figure 2.6.

(a) Glossy reflection when viewed from a low angle.

(b) No glossy reflection when viewed directly from above.

Figure 2.6: Example of the Fresnel effect showing glossy reflection of a chair's leg.

The Cook-Torrence specular reflection model with the Fresenel effect is expressed as Equation 2.6 [29].

$$f_{specular} = \frac{DGF}{\pi cos(\omega_i)cos(\omega_o)} \tag{2.6}$$

BRDF Shading models

Examples of BRDF shading models used by Filament and UE4 will be presented in this section.

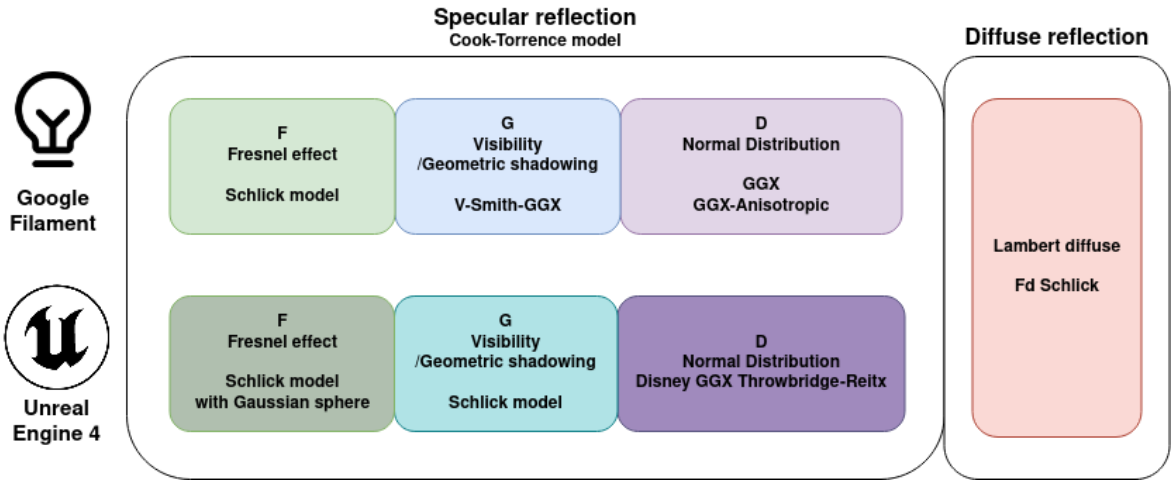


Figure 2.7: BRDF shading models of Google Filament and Unreal Engine 4.

Filament's standard shading model was called "Lit" and used the Cook-Torrence microfacet model and the Lambertian diffuse model. Filament had several other PBR shading models used for different purposes, i.e. the "Subsurface" shading model can be used for translucency and transparency effects. This shading model is useful for rendering human skin or wax candles.

Unreal Engine 4's shading model used the Cook-Torrence microfacet model for specular reflection and the Lambert diffuse model for diffused reflection. For diffuse reflection, the Lambertian diffuse Schlick function was used by both Filament and Unreal Engine 4. What differed was the geometric function and the normal distribution function [23], see Figure 2.7.

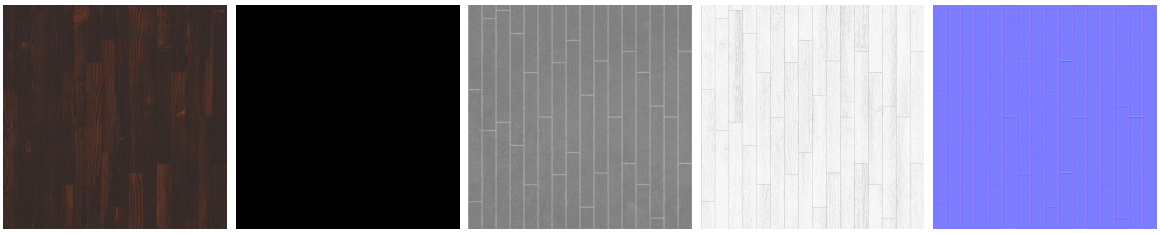
The original BRDF shader code of Filament and an implemented UE4 can be found in Appendix B Code B.2 and B.3.

2.3.3 PBR Textures

The BRDF uses the PBR textures for calculating the final color of a pixel. These textures give information about the color of a material, how light should reflect and how shadows should be applied for a more natural look [32].

There are five commonly used textures for PBR which describes a material's properties: base

color (also called albedo), metallic, roughness, ambient occlusion and normal map, see Figures 2.8 and 2.9.



(a) Base color (albedo) (b) Metallic (c) Roughness (d) Ambient Occlusion (e) Normal map

Figure 2.8: PBR texture of oak wood which is a non-metallic material. Source: Free PBR [33].

(a) Base color (albedo) (b) Metallic (c) Roughness (d) Ambient Occlusion (e) Normal map

Figure 2.9: PBR texture called "Pirate gold" which is a metallic material. Source: Free PBR [33].

The base color, or albedo, is a texture used for describing the main colors of the material. If the material is wood the the color will be brown, see Figure 2.8a. If the material is gold the the color will be yellow, see Figure 2.9a.

The metallic texture tells if a material is metallic or not. Black color means non-metallic and white means metallic, see Figures 2.8b and 2.9b.

The roughness texture gives information about the roughness of the materials surface, see Figures 2.8c and 2.9c.

The ambient occlusion texture is used for applying soft shadows on a surface for a more realistic look, see Figure 2.8d and 2.9d. This technique is used for fast calculations of shadows coming from indirect ambient light [34].

The normal map texture gives information about the depth of the surface. This texture provides the BRDF shading models with normal vectors of the material's surface, see Figure 2.8e and 2.9e.

2.3.4 History and motivation

In 1970 a computer scientist named Jim Blinn stated "As technology advances, rendering time remains constant" which became known as Blinns' law. This meant that rendering

more complex graphics was the objective instead of rendering the same thing but faster [35]. The research about PBR took off in the 80's when Turner Whitted published a paper about ray-tracing and lightning effects [4].

In the same decade, Cook and Torrence introduced the microfacet reflection model. This model made it possible for metals to look accurate in rendered computer graphics [36].

Goral et al. introduced radiosity which is an approach to solve the rendering equation by using finite element method (FEM). This approach incorporated global diffuse lightning in graphics, which took into account how surfaces reflected diffused lightning from object to object. A model of how objects correctly interact with each other in terms of diffuse reflectivity. Radiosity is a global illumination algorithm as light bouncing off objects is taken into account [37].

Diffuse lighting was further improved by Cohen, Greenberg, Nishita and Nakamae. At the time, this was a very computationally demanding effect but the results were satisfactory [38][39].

In 1984 Cook, Porter and Carpenter improved upon Whitted's ray-tracing model. The new model called distributed ray-tracing introduced depth of field and motion blur of moving objects and was capable of making objects look glossy and translucent [40].

In 1986 Kajiya developed the rendering equation which generalised and incorporated the different physically based rendering techniques. Path-tracing was introduced as an Monte-Carlo integral solution for the rendering equation [28].

Use cases and studies of PBR

In film production, physically based shaders have been used by Disney in movies like Tangled (2010) where the hair was physically based. For the movie Wreck It Ralph (2012), every surface used a physically based shader. The movie Wall-E used physically based shaders where the BRDF model was developed together with the people who used it. More movies using PBR are Gravity (2013) and The Hobbit: Battle of the five armies (2014) [4].

For augmented reality (AR), a study concerning teaching students about PBR thought AR has shown that PBR is well suited for AR and helped the students to better understand the concept of PBR [41]. Another study regarding rendering realistic smoke simulation based on physics for virtual reality (VR) has stated that hardware was a limiting factor for immersive VR experience and there can be improvements in optimizing algorithms and techniques to achieve a smoother experience [42].

Scene understanding is important for robotic navigation and human companion assistance. However, there is a limited amount of useful real images for training a convolutional neural network. A study used PBR to produce synthetic realistic looking images for training their neural-network which increased indoor scene understanding tasks [43]. Another study used a neural network to speed-up the rendering of PBR and the results looked photo-realistic [44].

Mobile has limited hardware capability for high-fidelity PBR rendering. An experiment has shown by having a server-client based system, the rendering time can be reduced significantly [45]. Another study has shown that using PBR game engines can speed-up computer

generated imagery (CGI) production time and reduce costs by a great deal [46].

2.4 Performance and Power Metrics

This section will present the most relevant metrics for measuring performance and power on a mobile phone device. The metrics will be the following: frame rate, bandwidth, processor cycles, power, dynamic voltage and frequency switching and energy.

2.4.1 Frame rate

Frame time is the measured time it takes to compute, render and display a frame. The inverse is the frame rate, measured in frames per second (fps) which is the pace to compute, render and display the frames [47][48].

Referring to the graphic pipeline in Figure 2.4, the total rendering time of a frame consists of the $t_{compute}$ which can be the vertex processing time, the t_{render} is the rasterization and fragment processing time and $t_{display}$ is the time it takes to send the final image to the display, see Equation 2.7.

$$t_{frame} = t_{compute} + t_{render} + t_{display} \tag{2.7}$$

The frame rate is the inverse of frame time measured as frames per second (fps), see Equation 2.8.

$$fps = \frac{1}{t_{frame}} \tag{2.8}$$

2.4.2 Bandwidth

Bandwidth and throughput are synonymous with the amount of data that can be transferred per unit time [49][50][51].

$$bandwidth = \frac{bytes}{second} \tag{2.9}$$

Another way to measure bandwidth is bytes per frame which is calculated by dividing the bandwidth with the frame rate to get the bandwidth per frame [52].

$$bandwidth_{frame} = \frac{bandwidth}{fps} = \frac{bytes}{second} \cdot \frac{second}{frame} = \frac{bytes}{frame} \tag{2.10}$$

2.4.3 Cycles

Processor cycles is a measure of how much work the processor needs to do for a certain task [53]. The metric can be used to calculate cycles per instruction (CPI) or processor usage as cycles per second (cycles/s) [54]. The relation between cycles, clock frequency and execution time can be seen in equation 2.11[55].

$$\text{Processor cycles} = f_{clock} \cdot t_{execution} \tag{2.11}$$

2.4.4 Power and DVFS

Power (P) is a rate of energy measured in joules per second and can be used to calculate how much heat is dissipated [56]. Electric power can be expressed in multiple ways, see equations 2.12, 2.13 and 2.14.

$$P = I \cdot V \tag{2.12}$$

$$P = I^2 \cdot R \tag{2.13}$$

$$P = \frac{V^2}{R} \tag{2.14}$$

In electronics, the power of a transistor is expressed as static and dynamic power, see equation 2.15. The static power (P_{static} is the leakage power from capacitors and leaking current. The dynamic power comes from the switching of the transistors and takes up most of the total transistor power [56].

$$P_{transistor} = \quad P_{dynamic} + P_{static} \tag{2.15}$$

Static power is expressed as the product of the supply voltage V_{dd}, the leakage current I_{leak} which is a technology dependant parameter, the number of transistors (N) and k_{design} which is a design dependant parameter [57][58], see Equation 2.16 .

$$P_{static} = Vdd \cdot I_{leak} \cdot N \cdot k_{design} \tag{2.16}$$

Dynamic power is expressed as the product of the transistor activity α, the effective capacitance C_{eff}, the supply voltage squared V_{dd}^2 and the clock frequency f_{clock}, see Equation 2.17 [56].

$$P_{dynmaic} = \quad \alpha C_{eff} Vdd^2 f_{clock} \tag{2.17}$$

Dynamic voltage and frequency scaling (DVFS) is a technique for reducing power consumption by changing the voltage and frequency dynamically [59].

2.4.5 Energy

Energy can be seen as the amount of work needed to fulfill a task. Potential electric energy is closely related to electric power which is a rate of energy, where E is energy in joules, P is power in Watt and t is time in seconds [56], see Equation 2.18.

$$E = P \cdot t \tag{2.18}$$

Energy efficiency, also referred to as "performance per watt" is a measure of performance rate per joule [58]. A common metric is frame efficiency which is measured as frame rate per Watt (fps/W) which simplifies down to frames per joule, see Equation 2.19.

$$\text{Energy Efficiency} = \frac{fps}{P} = \frac{\frac{frames}{second}}{\frac{Joules}{second}} = \frac{frames}{Energy} \quad \left[\frac{frames}{J}\right] \tag{2.19}$$

Another energy efficiency metric used is FLOPS per Watt, where FLOPS stands for floating-point operations per second [60] [61], see Equation 2.20.

$$Efficiency = \frac{FLOPS}{P} = \frac{\text{Floating-point operations}}{Energy} \quad \left[\frac{flops}{J}\right] \tag{2.20}$$

A better metric is frame energy, which is the inverse of frame efficiency and which tells how many joules per frame, see Equation 2.21. This metric is more descriptive and tells you the energy. The metric "performance per Watt" is ambiguous and harder to understand than energy per frame [62].

$$\text{Frame Energy} = \frac{P}{fps} = \frac{1}{\text{Energy Efficiency}} = \frac{Energy}{frame} \quad \left[\frac{J}{frame}\right] \tag{2.21}$$

2.5 Equipment and Software Tools

This section will address the equipment and tools that were used or considered for this research. First the latest generation of mobile phones will be presented then the methods for measuring performance and power will be discussed and last, the graphics rendering engines will be discussed.

2.5.1 Mobile phones

The phones using the external organization's latest and second latest generation of processors were Vivo X90 Pro and Asus ROG 6D. The specifications can be seen in Table 2.1.

Table 2.1: The specification of the phones Vivo X90 Pro and Asus ROG 6D.

	Vivo X90 Pro	**Asus ROG 6D**
Operating system	Android 13	Android 12
System-on-chip	Mediatek Dimensity 9200	Mediatek Dimensity 9000+
CPU chipset	1x Arm Cortex-X3 @3.05 GHz 3x Arm Cortex-A715 @2.85GHz 4x Arm Cortex-A510 @1.80 GHz	1x Arm Cortex-X2 @3.2GHz 3x Arm Cortex-A710 @2.85GHz 4x Arm Cortex-A510 @1.80GHz
GPU	Arm Immortalis-G715 MC11	Arm Mali-G710 MC10
Memory RAM	LPDDR5 12GB	LPDDR5X 16GB/12GB
Display	1260x2800, 120Hz AMOLED	2448x1080, 165Hz AMOLED
Battery	Li-Po 4870mAh	Li-Po 6000mAh

At the time of writing, the latest phone of the two was the Vivo X90 Pro. The Vivo X90 Pro had the latest Arm GPU, Immortalis-G715, and a newer CPU chipset, see Table 2.1. The Asus ROG 6D had the previous generation GPU, Mali-G710 and previous generation of CPU chipset. However, the Asus ROG 6D phone used the latest memory technology, LPDDR5X, while the Vivo X90 Pro used the older LPDDR5 technology.

2.5.2 Profiling and Data Acquisition System

Mobile phones run on battery and the lifetime is a limiting factor for applications and therefor it is important to optimize the applications to be more energy efficient. A profiler is a tool for profiling applications and computer systems by observing CPU activity, memory bandwidth, network traffic and energy [63].

Android Studio Profiler which is a system-wide profiler, meaning it can profile events and hardware at a system level [63]. Google's profiler called Perfetto is an open-source system-wide profiler for Android and considered a good alternative to Android Studio Profiler [64]. Arm, the external organization have an application level profiler called Streamline Performance Analyzer which is capable of measuring application level statistics like threads and had access to hardware counters for CPU, GPU and memory [65].

A data acquisition system (DAQ) is a tool used to measure real-world physical quantities such as electrical power, temperature, pressure and more. The DAQ samples the real-world quantities through sensors and transducers. The measured signals are converted into digital values and stored in a computer for further analysis and processing [66].

2.5.3 Graphics debugger

Renderdoc is a graphics debugger used for capturing and examining a rendered frame. All the assets in a frame can be viewed i.e. the 3D-model, the texture and the shader code. Renderdoc will show how the graphics pipeline was setup for the captured frame, the graphics API calls and shaders used [67].

2.6 Previous Work

Studies have shown that frame rate performance can be significantly increased and memory and energy consumption can be decreased by using Vulkan graphics API over the known

standard OpenGLES. A study about Vulkan rendering for mobile measured an increased frame rate up to 30%, decreased memory usage by 30% and less power consumption by 20% over using OpenGLES API [20]. However, the experiment was only tested for 2D-rendering and not 3D-rendering. Measuring for 3D-rendering on mobile using Vulkan API will conducted in this thesis project.

Another study about evaluating the performance and energy efficiency of OpenGLES and Vulkan on a desktop rendering server measured a power decrease up to 50% by using Vulkan. Vulkan's frame rate outperformed OpenGLES when power was not a concern and the GPU utilization was higher when using Vulkan. However, the development time for Vulkan is much longer and the learning curve is steeper than OpenGLES [5].

These studies set the expectation that Vulkan should increase performance and reduce power over using OpenGLES API. The following chapter will discuss how this will be tested.

Chapter 3

Experimental Implementation of PBR on Mobile Phone

The goal of this thesis project was to measure PBR on mobile and investigate how the graphics API and BRDF shading model affected the power and performance. In order to achieve this goal, three main components were needed: a mobile phone, a PBR graphics engine and measurement tools.

This chapter will discuss the choices made and the process of obtaining performance and power measurements of PBR on mobile. First, the tools used for measuring performance and power will be presented. Then the choice of PBR engine and the test cases used in the experiment. The last section will be an overview of how the experiment was conducted to obtain performance and power measurements.

3.1 Mobile Phone

The phone used for this project was the Asus ROG 6D, an Android phone marketed for gaming. It was provided by the external organization and it used the external organization's previous generation CPU chipset and GPU, see Table 2.1.

One of the reasons Asus ROG 6D was chosen over the newer generation phone, Vivo X90 Pro, was due to the better memory technology. Memory is often the bottle neck and thus the best memory technology was chosen [68]. On the same note, Vivo X90 Pro was released during the thesis work which meant that the driver could have issues which could give poor performance as the driver is still being optimized. Also the availability of the newer phone was low as it was the latest technology.

3.2 Measuring Tools

The measuring tools used for measuring performance and power were provided by the external organization and their usage will be presented below and throughout this chapter. The process of measuring PBR on mobile will be presented in a later section.

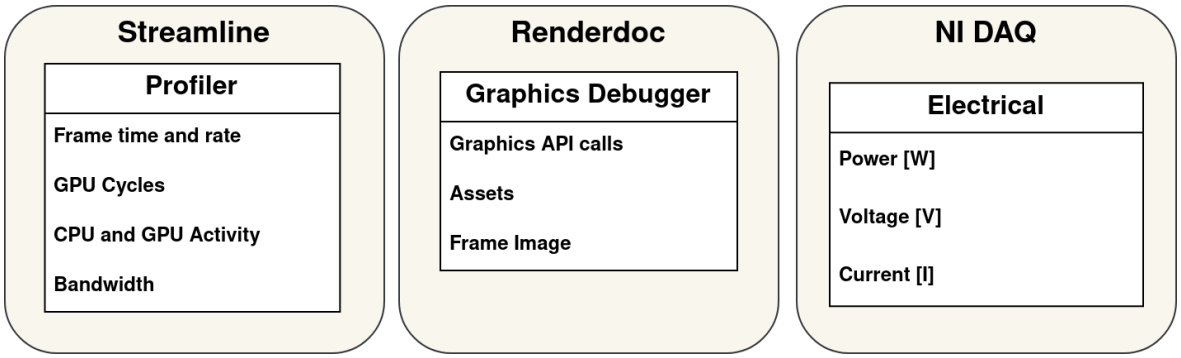


Figure 3.1: The measuring tools used to obtain performance measurements, rendered images and electrical power measurements.

The information of interest was rendered images, performance and power metrics, see Section 2.4:

- Image Quality
- Frame rate [fps]
- GPU memory bandwidth (read and write) [bytes/frame]
- GPU cycles [cycles/frame]
- CPU and GPU activity
- Battery power [W] and frame energy [mJ/frame]

In short, Streamline was used for measuring performance, Renderdoc was used for capturing images and National Instrument's (NI) DAQ for measuring battery power, see Figure 3.1.

3.2.1 Profiler for measuring performance metrics

The profiler Arm Streamline was used for measuring the performance metrics such as frame time, GPU memory bandwidth (read and write), GPU cycles and processor activity. This profiler was the natural choice as it was developed by the external organization for the processors used on the phone (Asus ROG 6D) and worked on application-level profiling.

Other alternative profilers considered were Perfetto and Android's profiler, see Section 2.5.2. However, those were system-level profilers and would not give as accurate measurements or interesting metrics specific for the GPU and CPU.

3.2.2 Graphics debugger for capturing frames

The graphics debugger used was Renderdoc which captured the rendered images used for image quality analysis. Renderdoc was also used for verification of which BRDF shading model was used, assets (3D-model and textures) and API-calls. Renderdoc was used for developing the test cases presented in next Section 3.3.

3.2.3 DAQ for power measurements

NI's DAQ was used for measuring the battery power of the phone under test (Asus ROG 6D). The external organization had a setup for automating power measurements of phones and was used for obtaining power measurements.

3.3 PBR Engine

Google Filament was the choice of PBR engine because it already had a framework setup for PBR on Android mobile. Unity, Unreal Engine and the external organization's own framework were considered alternatives, see Section 2.2.2. Unreal Engine had the most advanced graphics and would be the second choice.

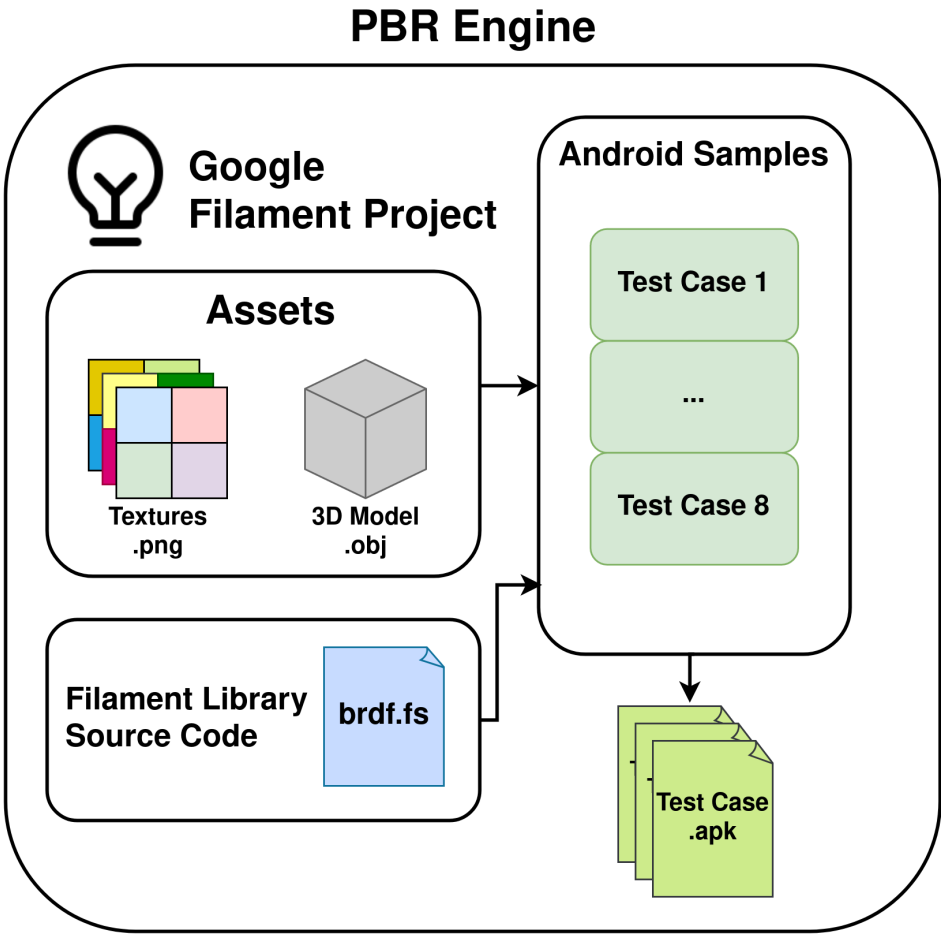


Figure 3.2: Google Filament PBR Engine Overview. The Filament project contained assets, samples and the Filament library. The Android samples used the Filament library and the assets which were built into an Android package kit (APK). The APK is the file used for installing the application into the Android phone.

The Google Filament project included everything needed to create a PBR application for Android. For this project, the assets, the filament library and an Android sample were used and modified to create test cases, see Figure 3.2.

The goal was to create test cases for comparing graphics API and BRDF shading models. Filament had an Android sample called "Textured Object" which was a PBR application displaying a rotating 3D-model with PBR textures on it. This sample was used as a baseline for creating the test cases used in this project.

The configurations for the experiment was the scene, the back-end graphics API and the BRDFs. The scene was either Suzanne or Lucy. The back-end graphics API was OpenGLES or Vulkan. The BRDF used was either Google Filament's native BRDF or an implementation of UE4's BRDF into the Filament library. In total, eight test cases were created using these three configurations, see Table 3.1. These configurations will be discussed below.

Table 3.1: The test cases from the configurations of scene, graphics API back-end and BRDF shading model.

Scene	Graphics API	BRDF
lucy	gles	filament
lucy	gles	ue4
lucy	vk	filament
lucy	vk	ue4
suzanne	gles	filament
suzanne	gles	ue4
suzanne	vk	filament
suzanne	vk	ue4

3.3.1 The scene

(a) The Suzanne scene with wooden texture.

(b) The Lucy scene with metallic golden texture.

Figure 3.3: The scenes called Suzanne and Lucy.

The purpose behind having two different scenes was to measure how the amount of vertices and the type of texture would affect the performance and power. The 3D models were already included in the Google Filament project and the textures used were from a free PBR texture website [33].

The Suzanne scene was composed of the 3D-model named Suzanne which had 47232 vertices and a non-metallic wooden PBR texture, see Figure 2.8, and an image based lighting texture providing background and light. The Lucy scene used the 3D-model named Lucy which had 100338 vertices (more than twice of Suzanne) and a metallic golden PBR texture, see Figure 2.9, and a different image based lighting, see Figure 3.3.

Each PBR texture pack consisted of an albedo (base color), metallic, roughness, ambient occlusion and normal map texture, covered in Section 2.3.3.

3.3.2 Adding UE4 BRDF shading model into Filament's BRDF shader

Google Filament had its own native physically based BRDF. Filament's BRDF was supposed to be visually appealing and energy efficient. To test this, an implementation of UE4's BRDF shading model was added into the Google Filament library.

This was achieved by adding the UE4's BRDF into Google Filaments BRDF shader source code (brdf.fs), see Figure 3.2. These two BRDF shading models have been covered in previously in Section 2.3.2. The UE4 shading model was from the article "Real Shading in Unreal Engine 4" by Brian Karis [23] and was expressed as mathematical formulas. This was translated into shader code and was added along-side Filament's native BRDF shading model in the same file. The shader was written in OpenGL shading language (GLSL).

The selected BRDF shading model was compiled into the Filament library which was used by the test application. In order to change the BRDF shading model, the Filament library had to be re-built from scratch every time. Rebuilding the Filament project took long compilation time compared to building the Android test application.

As previously mentioned, the graphics debugger Renderdoc was used to that the correct BRDF shading model was used in the test application. The effect of which BRDF was used could be seen visually on the display of the phone.

The original Filament's BRDF shading code can be found in B.2 and the implemented UE4's BRDF shading model in B.3.

3.3.3 The test case application

The original Filament Android sample called "Textured object" was used as a baseline for the test cases. The test cases were written in Kotlin which was a Java based programming language. The test cases was then compiled into Android applications (apk) and installed on a mobile phone, see Figure 3.2.

In order to create PBR applications used as test cases, a bug had to be fixed. By fixing this bug, multiple copies of the base sample application called "Textured Object" could be created and changed. Details about this contribution to the offical Google Filament project repository can be found in Appendix B.1.

The modifications done and how the test application worked will be presented below.

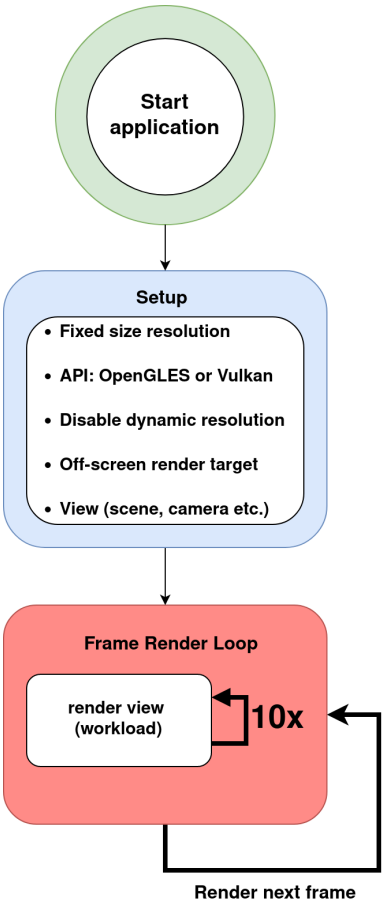


Figure 3.4: Simplified overview of the test case application data-flow.

The test case application had two main parts - the setup and the render-loop, see Figure 3.4.

Fixed size resolution and disabling dynamic resolution optimization

For better comparisons, a fixed frame resolution of 1080x1920 pixels was set and Filament's dynamic resolution optimization had to be disabled. This was to ensure that all test cases were rendered in the same resolution.

The dynamic resolution optimization would render at a lower resolution, i.e. 720x1280 pixels, and scale up the final image which was sent to the display at the set resolution of 1080x1920. This was detected in Renderdoc and the OpenGLES and Vulkan back-end used different resolutions for rendering. Disabling this optimization gave more deterministic test cases.

Off-screen rendering

For better comparisons and more accurate power measurements of the mobile processors, off-screen rendering was set up. This meant rendering the frame to memory instead of the display, see Figure 2.4. This took the screen out of the power measurements and gave more accurate results.

Graphics API back-end

The graphics API back-end was then chosen which was either OpenGLES or Vulkan. The default graphics back-end in Filament was OpenGLES but that could be changed to Vulkan. This was achieved by changing one line in the original code and no graphics API knowledge was needed to use either of the graphics back-end APIs. This was one of the benefits of using a graphics engine, see Section 2.2.2.

Scene

Then the view was set up containing the scene, camera and rotation matrix for rotating the 3D-model clockwise, see Figure 4.1. The scene had either the 3D-model Suzanne or Lucy along with their respective textures, see Section 3.3.1. In this thesis, the word scene and view were used synonymously. The scene (or view) was rendered in the render-loop, see Figure 3.4.

The vertical-synchronization limit

The vertical-synchronization (v-sync) limit was the maximum refresh rate of the display. This limited the GPU from rendering at its fastest potential. This limit was set to 120 fps (8.33 ms) and this was the fastest frame rate which could be measured. If the GPU rendered faster than 120 fps then the GPU had to wait idly for the display to finish. The initial test cases surpassed this limit and thus all test cases were measured at 120 fps.

The first solution was by setting up off-screen rendering. The off-screen rendering should by-pass the display and its v-sync limit. This did not work and the reason was not further investigated due to limited time.

The second solution was to decrease the frame rate by having the GPU do more work. What was rendered in a frame was the so called workload. To increase the workload, the same scene was rendered multiple times per frame. This lowered the frame rate and differences between the test cases could be measured.

At first, the workload was rendering the same scene 100 times per frame. The results were to unrealistic and the workload was lowered down to 10 scenes rendered per frame. The profiler Arm Streamline was used to detect the unrealistic performance values where the frame rate was less than 10 fps and the GPU bandwidth was very high. The graphics debugger Renderdoc was used to confirm the number of scenes render per frame were correct.

The workload of rendering the same scene 10 times per frame was used for the final results given in Chapter 4.

3.3.4 The project repository

The original Filament repository in Github was forked into a own repository where the test cases were developed. The repository for this project can be found in Github [69].

3.4 System Overview and Test Setup

Credit and thanks goes to the external organization for their guidance and help using their automated measuring system for obtaining the power and performance measurements of the

PBR test cases which ran on the mobile phone.

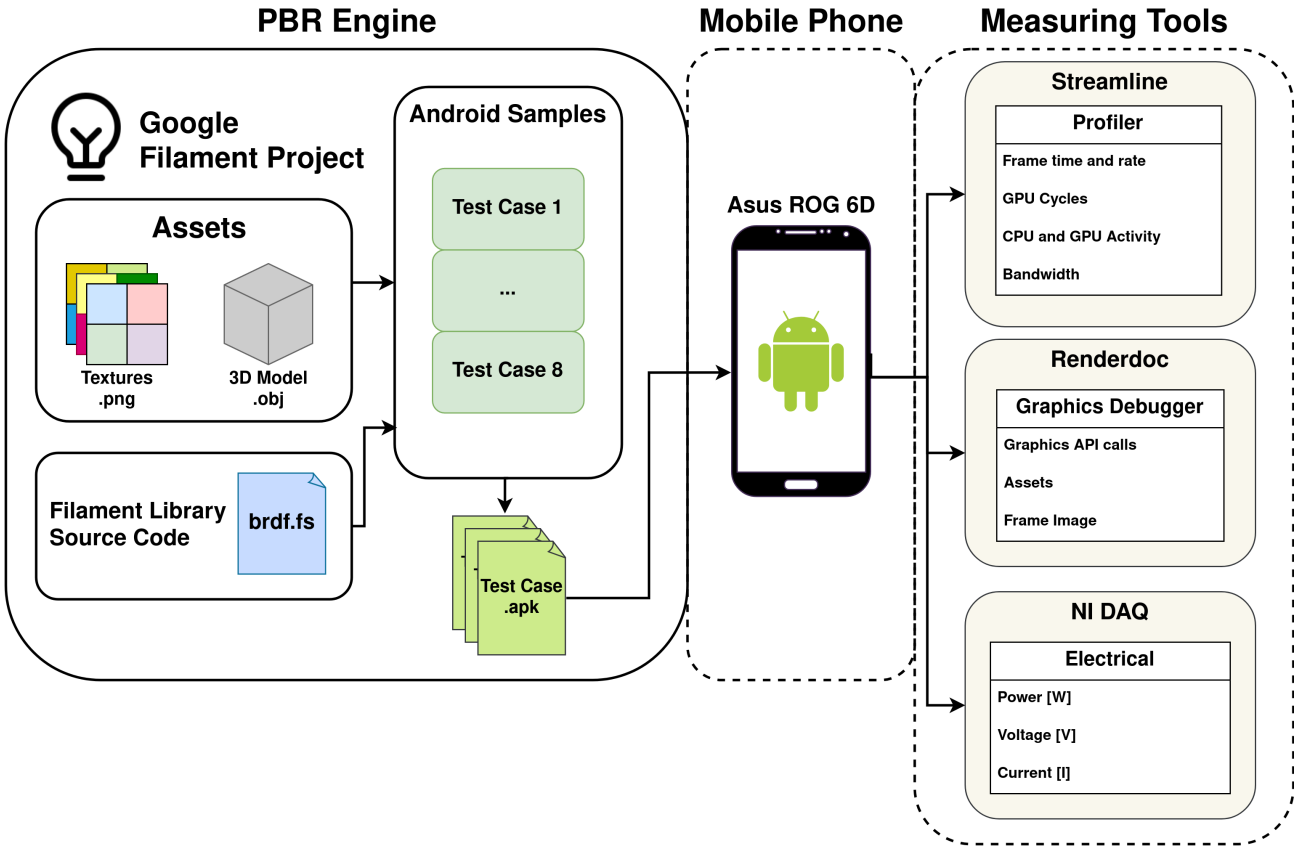


Figure 3.5: Overview of the system for conducting the experiment.

To recap this chapter, Google Filament PBR engine was used to build test cases for comparing the graphics back-end APIs Vulkan and OpenGLES and the BRDF shading models of UE4 and Filament. The test cases were installed on an Android phone, Asus ROG 6D, and power and performance was measured using the measuring tools: Arm Streamline profiler, Renderdoc graphics debugger and NI's DAQ, see Figure 3.5.

The test cases were measured for their peak performance and power. This meant that the phone was in a state which it could render at its fastest speed and output its maximum power. This phenomenon could be seen when the phone started at a lower temperature of 26 degrees Celsius.

The phone was cooled down using a fan and each test case ran three times for 60 seconds. A measurement started only when the phone was under 26 degrees Celsius.

Obtaining the measurement were done automatically but processing the performance and power results had to be done manually. The performance metrics were calculated from 10 consecutive frames measured in Arm Streamline profiler. A discussion about this will be covered in later Chapter 5.

The results of measuring the test cases will be presented in the next Chapter 4.

Chapter 4

Results

This chapter will present the results of the experiment conducted in this thesis project. The first section will present the image quality results, followed by performance and power and the last section will present the processor activity results. The end of each section will have an analysis part discussing the results.

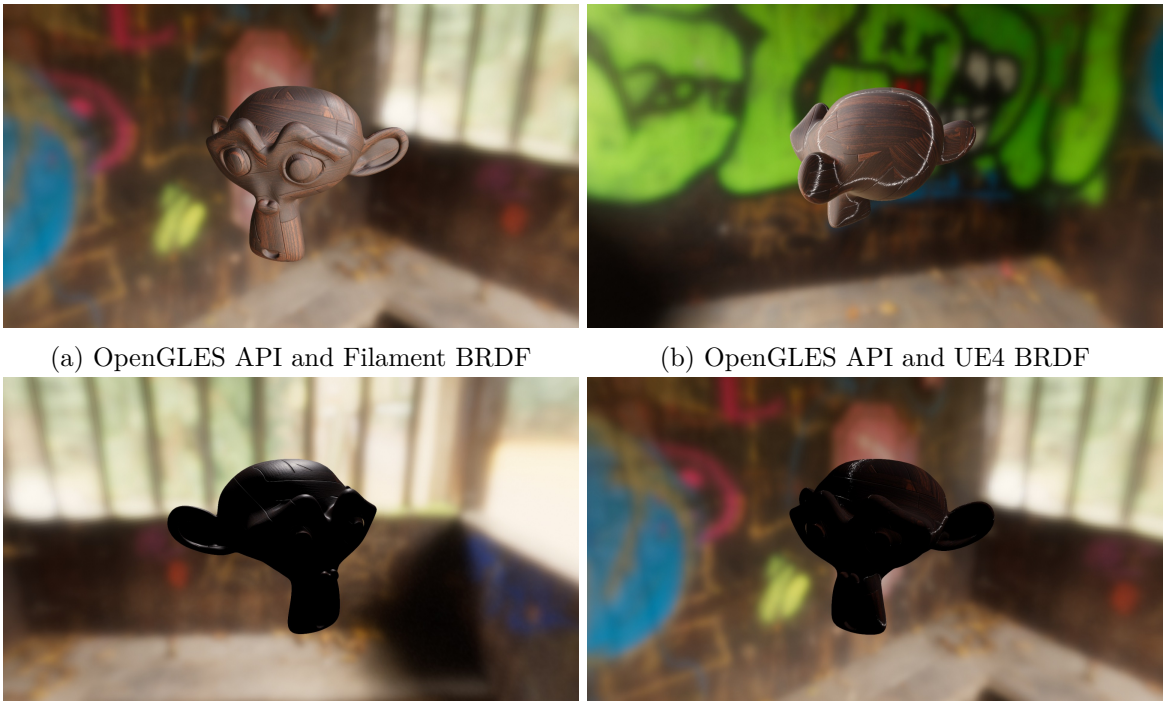
4.1 Image Quality

This section will present the rendered images of the eight test cases - starting with Suzanne, followed by the Lucy scene and the an analysis section. The figure is read as follows:

- Horizontally comparing BRDF: Filament (left) and UE4 (right)
- Vertically comparing graphics API: OpenGLES (top) and Vulkan (bottom)

The results of the image quality will be discussed more in depth later in chapter 5.

4.1.1 The Suzanne scene



(a) OpenGLES API and Filament BRDF

(b) OpenGLES API and UE4 BRDF

(c) Vulkan API and Filament BRDF

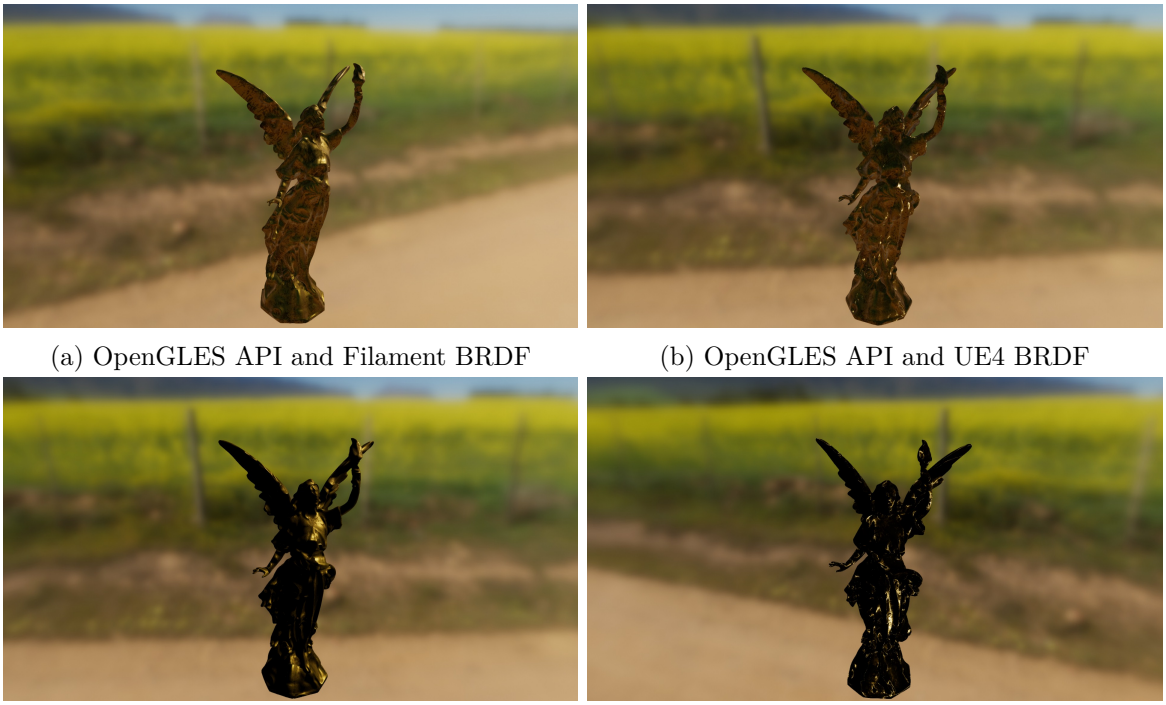
(d) Vulkan API and UE4 BRDF

Figure 4.1: Image quality comparison of the Suzanne scene.

Starting with the comparison of Filament's and UE4's BRDF. Filament's BRDF rendered a complete and satisfactory looking scene of Suzanne where the wooden texture could easily be identified, see Figure 4.1a. UE4's BRDF rendered frames which had a shiny white outline close to the edges of Suzanne model, see Figure 4.1b. Filament's BRDF gave a more shiny and glossy reflection on the wooden texture, see Figure 4.1c, compared to UE4's BRDF which made Suzanne look mate and dull, see Figure 4.1d. The white lines along the edges were more subtle in the Suzanne scene using Vulkan API and UE4's BRDF , see Figure 4.1d.

Looking at the graphics APIs OpenGLES and Vulkan. Suzanne had a clear wooden color using OpenGLES while the Vulkan test cases had very dark colors and dimmed lighting, see Figure 4.1c and 4.1d. It was harder to see the wooden texture on the Vulkan test cases for the Suzanne scenes. The graphics back-end should not affect the visuals of the rendered frame.

4.1.2 The Lucy scene



(a) OpenGLES API and Filament BRDF

(b) OpenGLES API and UE4 BRDF

(c) Vulkan API and Filament BRDF

(d) Vulkan API and UE4 BRDF

Figure 4.2: Image quality comparison of the Lucy scene.

Inspecting the results of the Lucy scenes by first comparing the BRDFs, Filament versus UE4 implementation, the biggest visible difference was how the direct light reflected on the model. With the Filament BRDF's, the direct light reflected properly on the 3D-model Lucy, see Figures 4.2a and 4.2c. The UE4's BRDF made Lucy look darker and the specular reflections were on different spots of the model and was missing the bulk of light shining on the 3D-model, see Figure 4.2b and 4.2d.

Comparing the graphics APIs, OpenGLES and Vulkan. The Vulkan back-end made the texture look darker and the details of the texture were lost, see Figures 4.2c and 4.2d. Looking at Lucy with Vulkan API and UE4's BRDF (Figure 4.2d, it was very dark and the material was not resembling gold as clearly as the other cases.

4.1.3 Analysis

Starting with the impact of changing the graphics API, Vulkan changed the image quality which was unexpected. The Vulkan API made the frames look darker and the textures lost details. The underlying graphics API should not affect the image quality and this was a strong indication of errors in Google Filament's Vulkan back-end.

The implementation of UE4's BRDF into the Google Filament project had worse image quality compared to Filament's native BRDF. The test cases using UE4's BRDF had a white line along the edges. A possible cause could be erroneous implementation of the UE4's BRDF. Another reason could be that the lighting model of UE4's PBR shading model was not taken into account, only the BRDF was implement. It is very likely that UE4's BRDF would look

better using UE4's graphics engine.

Comparison between the two textures, non-metallic (Suzanne) and metallic (Lucy), the Vulkan back-end worked better on the metallic scene Lucy over the non-metallic scene Suzanne. On the Suzanne scene, it was hard to see the wooden texture using Vulkan back-end. For the Lucy scene, the metallic texture could be identified. The test case using Vulkan back-end with UE4's BRDF was considered unacceptable for PBR, see Figure 4.1d and 4.2d.

A deeper discussion regarding the procedure will be covered in Chapter 5 Section 5.1.

4.2 Performance and Power

This section will cover the performance and power results. The setup of how the experiment was conducted and how the results were obtained has been covered in previous Section 3.4. The final results can be seen in Table 4.1.

Table 4.1: Performance and power results of the test cases. The configurations were the scenes: Lucy (lucy) and Suzanne (suzanne), the graphics back-end APIs: OpenGLES (gles) and Vulkan (vk) and the BRDFs: Filament (filament) and Unreal Engine 4 (ue4).

Scene	**API**	**BRDF**	**Frame Rate [fps]**	**GPU Bandwidth R/W [MB/frame]**	**Cycles / frame**	**Power [W]**	**Efficiency [fps/W]**	**Energy [mJ/frame]**
lucy	gles	filament	76.3	172.8	12800000.0	9.5	8.0	124.3
lucy	gles	ue4	76.3	170.6	12800000.0	8.8	8.6	115.7
lucy	vk	filament	37.5	350.5	20200000.0	6.2	6.1	164.2
lucy	vk	ue4	36.2	352.7	20700000.0	5.8	6.3	159.3
suzanne	gles	filament	84.7	166.1	11500000.0	9.3	9.1	110.0
suzanne	gles	ue4	79.4	174.3	12300000.0	9.5	8.4	119.7
suzanne	vk	filament	35.1	360.4	21300000.0	7.4	4.7	211.8
suzanne	vk	ue4	35.2	360.9	21900000.0	7.4	4.8	209.3

The results will be presented by looking at each metric in the following order: frame rate, GPU memory bandwidth, GPU cycles, battery power consumption and energy per frame.

Comparison results with less than 5% change were considered to have no change. Explained by the external organization, this range between 0 - 5% was due to the inner workings of Android OS which was out of control. All comparison plots of the results can be found in Appendix A Section A.1.

4.2.1 Frame rate - frames per second (fps)

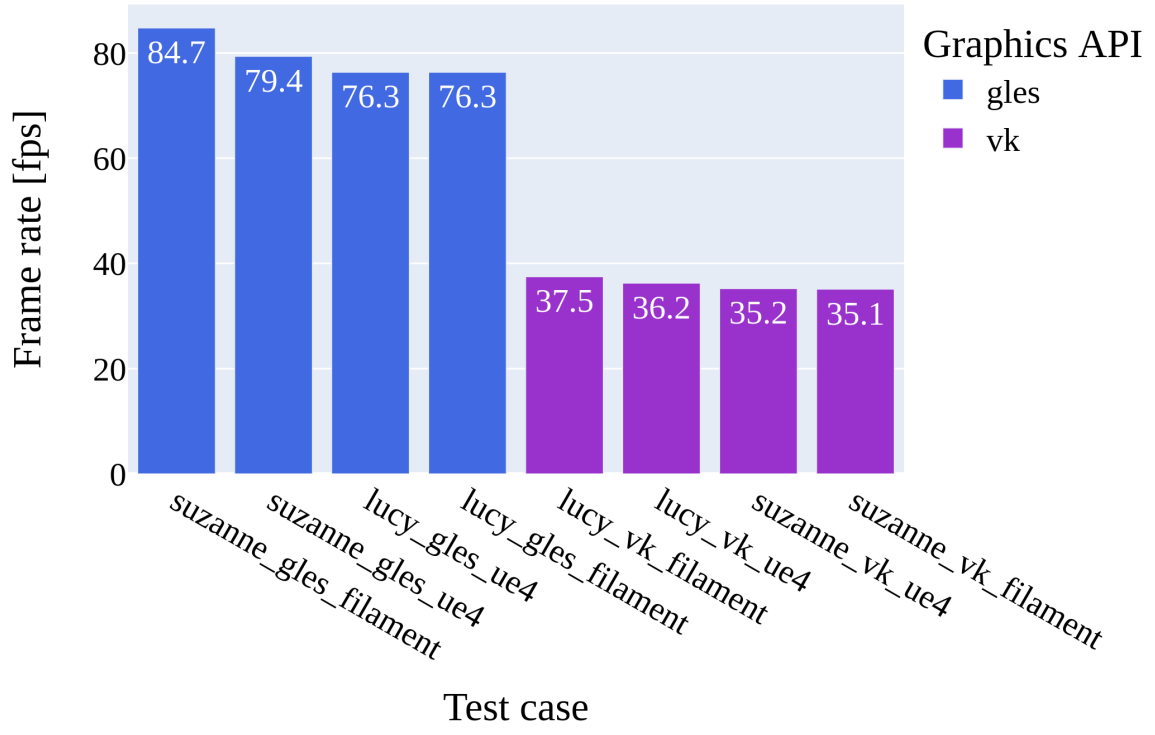


Figure 4.3: Frame rate results comparing OpenGLES (gles) and Vulkan (vk). Higher is better.

Looking at the absolute values, the OpenGLES test cases ran on average 80 fps and Vulkan around 40 fps, see Figure 4.3. The frame rate of both graphics back-end were above the acceptable range for real-time rendering which was 30 fps. A frame rate below 30 fps was considered to be too unpleasant for gaming. However, the Vulkan back-end was close to this minimum limit.

Comparing the graphics API's, Vulkan was 50% slower than OpenGLES. The results were surprising because the expectation was that the Vulkan back-end should perform better [5][20] which was not the case, see Section 2.6. These performance results of the Vulkan back-end further indicated that there are issues with the Vulkan back-end.

For the BRDFs, UE4's BRDF performed 6% worse than Filament's. The BRDF did not have a big impact on the performance as the back-end garphics APIs.

4.2.2 GPU memory bandwidth read-write - bytes per frame (MB/frame)

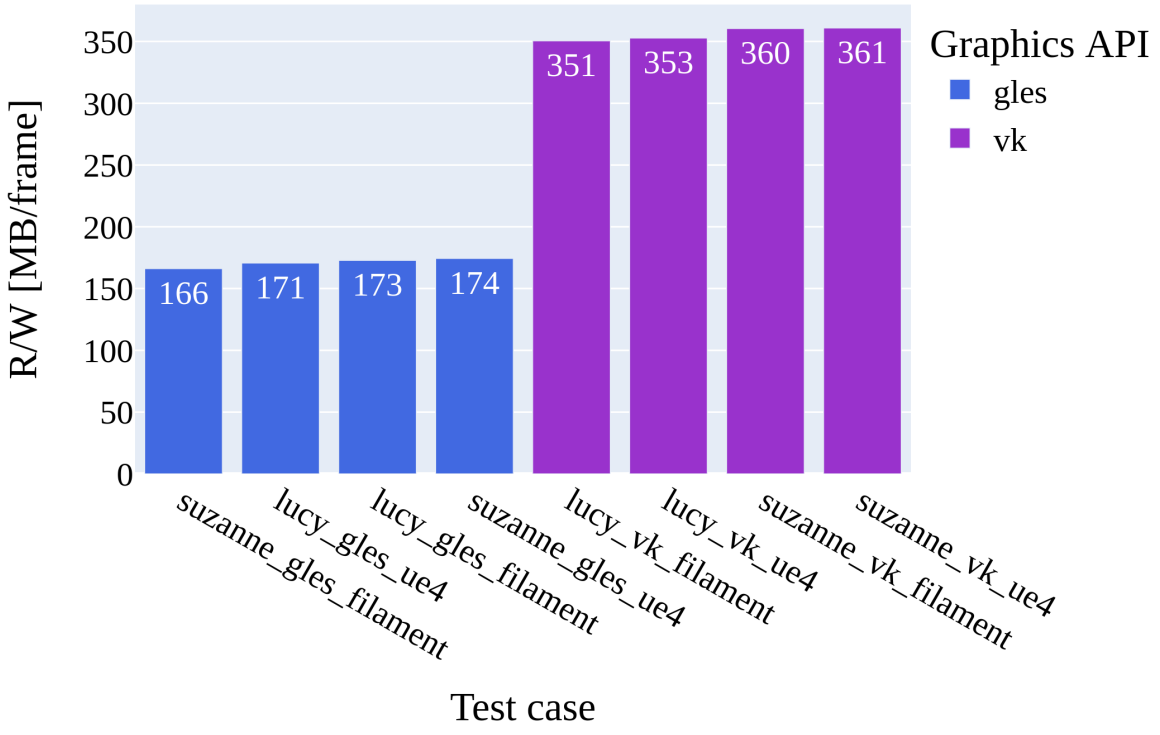


Figure 4.4: GPU memory bandwidth results comparing OpenGLES (gles) and Vulkan (vk). Lower is better.

Looking at the GPU bandwidth read and write results, the OpenGLES test cases had approximately 170 MB/frame and Vulkan around 355 MB/frame, see Figure 4.4.

Comparing the graphics API, Vulkan had more than twice as high bandwidth compared to the OpenGLES graphics back-end. The bandwidth results suggested that memory was the bottle-neck and could be one of the possible reasons for the reduced performance of the Vulkan back-end. As previously covered in Section 2.2.1, in Vulkan the programmer was responsible for managing memory. Why the Vulkan back-end used twice as much memory could be explained by sub-optimal memory management in the Filament project for the Vulkan back-end.

Comparing the BRDFs, there was no change in bandwidth as the change was under the acceptable range of 5%.

4.2.3 GPU cycles per frame (cycles/frame)

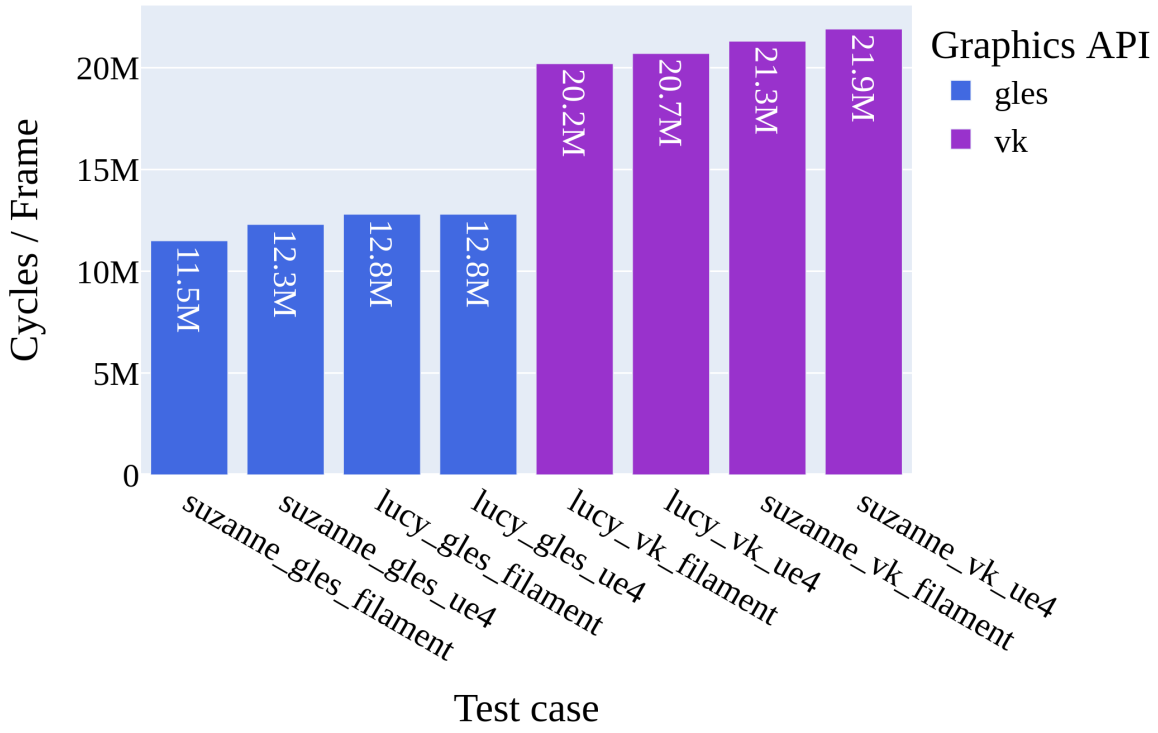


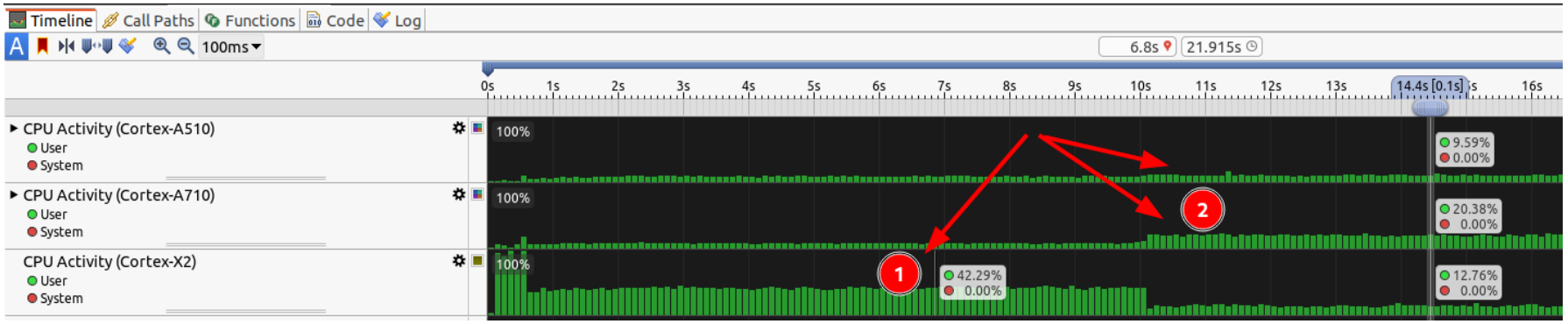
Figure 4.5: Cycles per frame results comparing OpenGLES (gles) and Vulkan (vk). Lower is better.

Looking at the GPU cycles per frame results, overall the OpenGLES test cases ran for 12 megacycles per frame and the Vulkan cases for 21 megacycles per frame, see Figure 4.5. Comparing the graphics API, Vulkan to OpenGLES, Vulkan used 58% to 85% more cycles than OpenGLES. These extra instructions used by Vulkan API could potentially be redundant memory read-write operations.

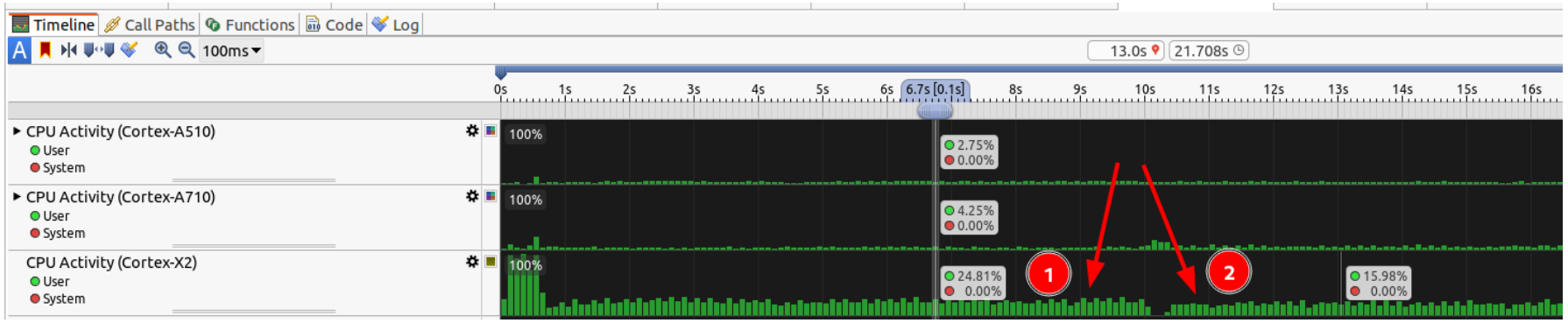
Comparing the BRDFs, UE4 to Filament's BRDF, UE4's BRDF used 7% more cycles than Filament meaning it performed worse.

4.2.4 Processor Activity

This section will take a closer inspection at the processor activity captured by the application profiler Arm Streamline. The processor activity of graphics API OpenGLES and Vulkan will be compared against each other.



(a) CPU activity of using OpenGLES back-end



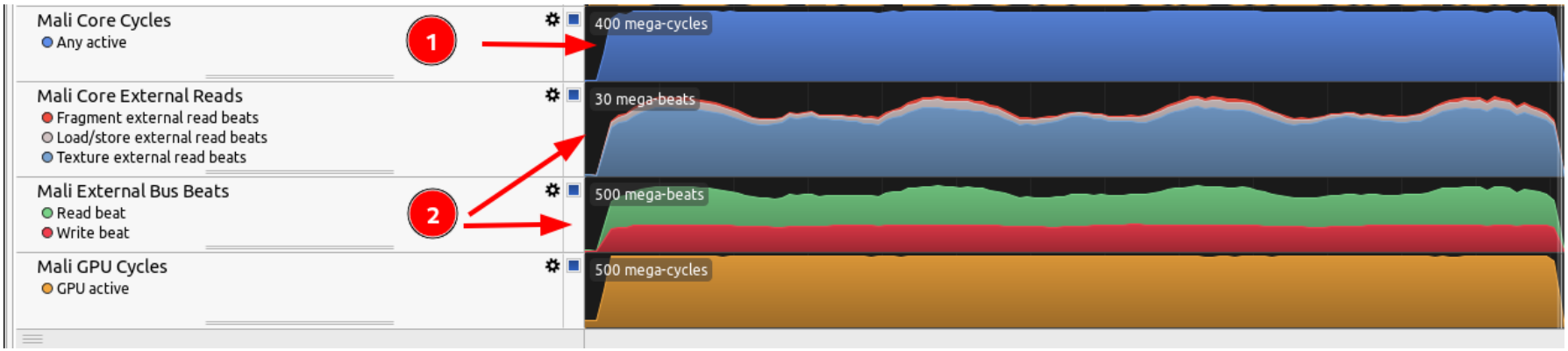
(b) CPU activity of using Vulkan back-end

Figure 4.6: CPU activity of OpenGLES and Vulkan graphics API captured in Arm Streamline profiler. Vulkan used less of the CPU than OpenGLES back-end graphics API.

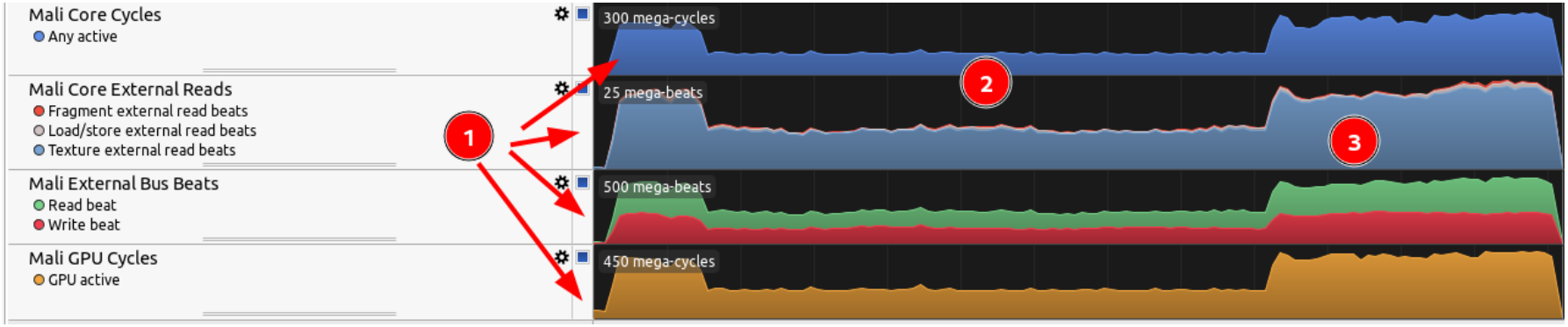
Starting with the CPU activity of using OpenGLES graphics back-end API, the big CPU core (Cortex X2) was used the first 10 seconds at 40%. After 10 seconds the smaller cores (Cortex A510 and A710) took over and used 10% and 20% of the CPUs respectively, see Figure 4.6a.

Looking at the Vulkan graphics back-end API, the big core (Cortex X2) was used for the whole run starting at 20% CPU usage. After the 10 second mark the CPU usage dropped down to 16%, see Figure 4.6b.

The processor activity results showed that the Vulkan graphics back-end API was more efficient by using less of the CPU compared to OpenGLES. This confirmed that the bottleneck was not the CPU.



(a) GPU activity of using the OpenGLES back-end



(b) GPU activity of using the Vulkan back-end

Figure 4.7: GPU activity (cycles and bandwidth) of OpenGLES and Vulkan graphics API captured in Arm Streamline profiler. OpenGLES shows a sinusoidal behaviour and Vulkan showed high and low activity regions.

Examining the GPU activity of OpenGLES graphics back-end API, the GPU cycles were constant while the bandwidth had a sinusoidal characteristic with peaks and lows, see Figure 4.7a.

Looking at the GPU activity of Vulkan back-end, the GPU activity was very high at the start of the test case application and after 7 seconds the activity dropped by half. This lower activity region remained until the activity went back up at the 45 seconds mark. The cycles and bandwidth shared the same characteristics, see Figure 4.7b. The measurement for Vulkan were all from the high GPU activity region at the start. It was observed that in the lower activity region the frame time reached the maximum measurable frame time of 8.33 ms (120 fps) which was due to the V-sync limit.

The behaviour of both The OpenGLES and Vulkan back-end could not be explained and further investigation would be required. All the test cases using the Vulkan back-end were measured in the high activity state.

4.2.5 Battery power (W)

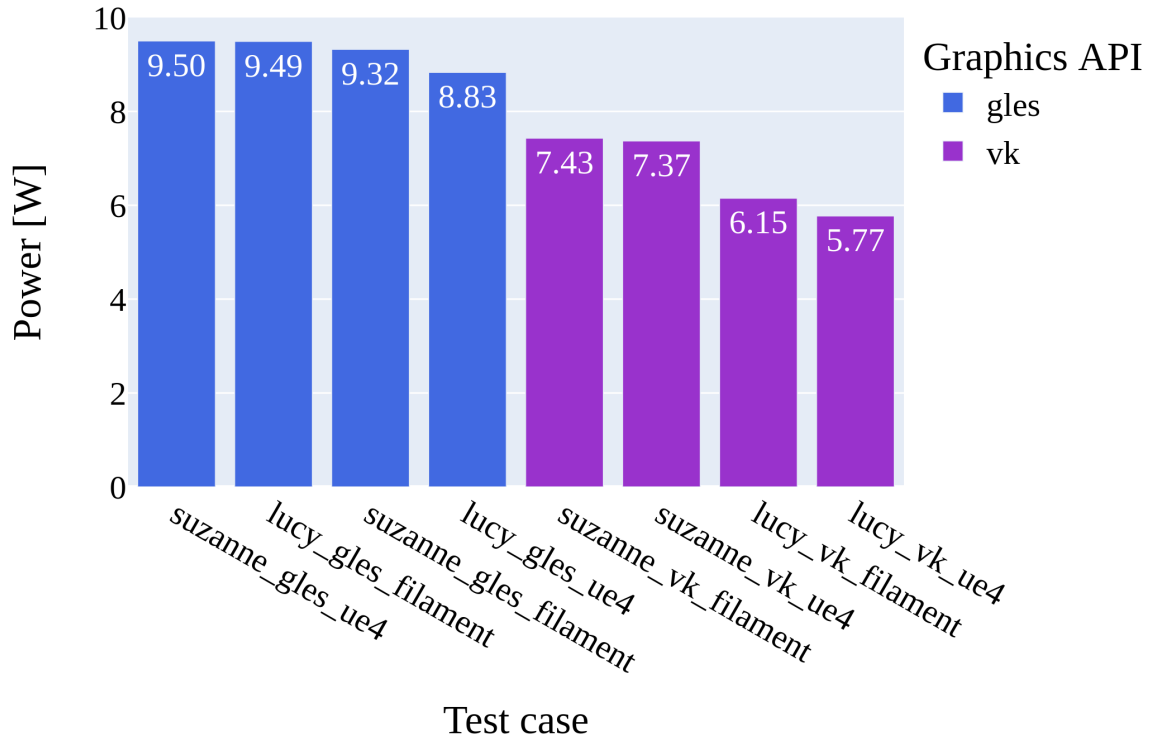
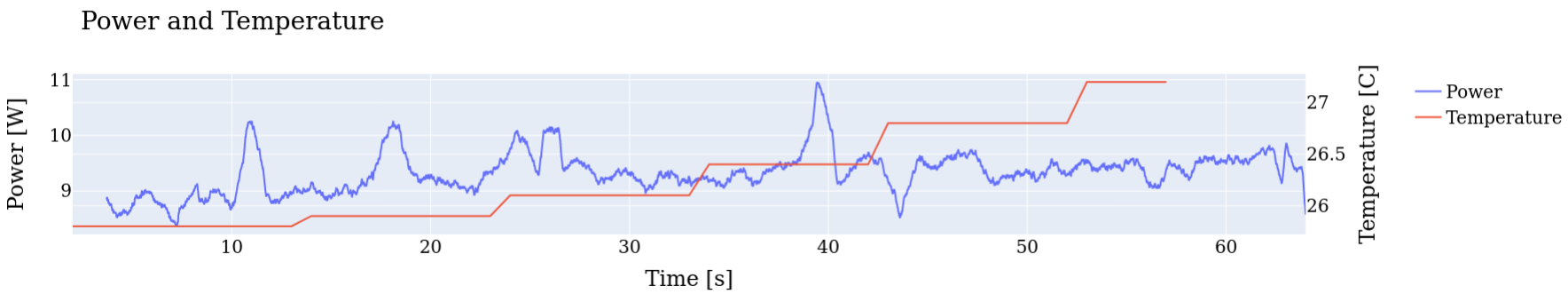


Figure 4.8: Average battery power consumption comparing OpenGLES (gles) and Vulkan (vk). Lower is better.

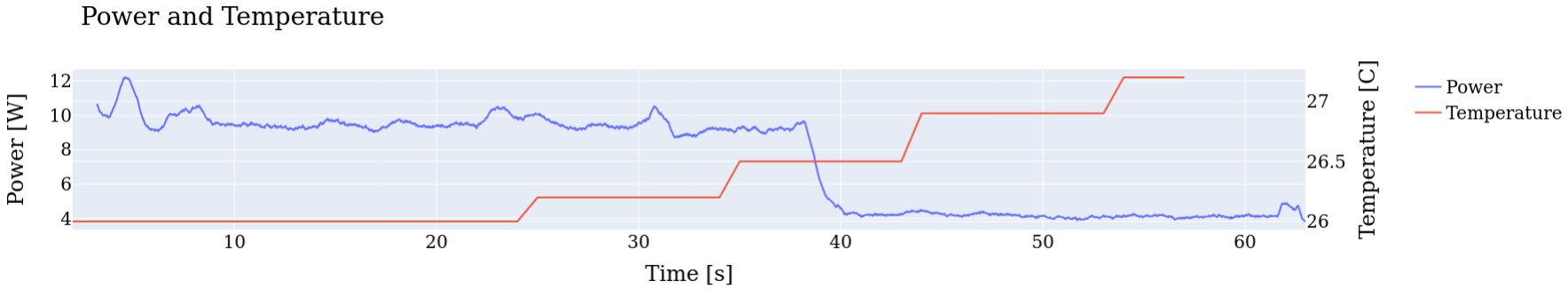
The power results showed that the OpenGLES graphics back-end used around 9 W and the Vulkan back-end around 6-7 W, see Figure 4.8. Comparing the graphics APIs, Vulkan to OpenGLES, Vulkan used 35% less power at best and 20% less at worst compared to OpenGLES. The Vulkan graphics back-end used less power and the reason was primarily the

lower usage of the CPU which was covered in the previous section. These results proved that the Vulkan API uses less of CPU and thus the power was lower.

Comparing the BRDFs, UE4 to Filament's, UE4 used 7% less power compared to Filament. UE4's BRDF saved more power and was better for reducing power consumption.



(a) Power and temperature measurement result of the Suzanne scene using OpenGLES API and Filament BRDF



(b) Power and temperature measurement result of the Suzanne scene using Vulkan API and Filament BRDF

Figure 4.9: Power measurement of OpenGLES and Vulkan graphics back-end.

Looking at the battery power measurements, OpenGLES started at 9 W and could have short spikes up to 11 W, see Figure 4.9a. With the Vulkan back-end, the power started at 10 W and after 40 seconds dropped down to 4 W, see Figure 4.9b. The temperature went from 26 to 27 degrees in 60 seconds, a one degree rise in temperature.

The battery power measurements showed that the OpenGLES graphics back-end started at a high power and remained constant with spikes. The Vulkan back-end started with high power and then dropped significantly lower. This gave the Vulkan back-end less average power consumption than OpenGLES. A reason why the Vulkan back-end had this drop could be compared with the processor activity drop covered in previous Section 4.2.4. However, the duration of the high power usage and the high GPU activity duration did not align time-wise.

Both the OpenGLES and Vulkan back-end had high initial power usage and the explanation was due to peak performance. As previously mentioned, this meant the device could run at its maximum performance and power when the device was at a lower temperature, which was 26 degrees. As the device heated up the performance and power dropped. All test cases were measured for their peak performance.

4.2.6 Energy per Frame (mJ/frame)

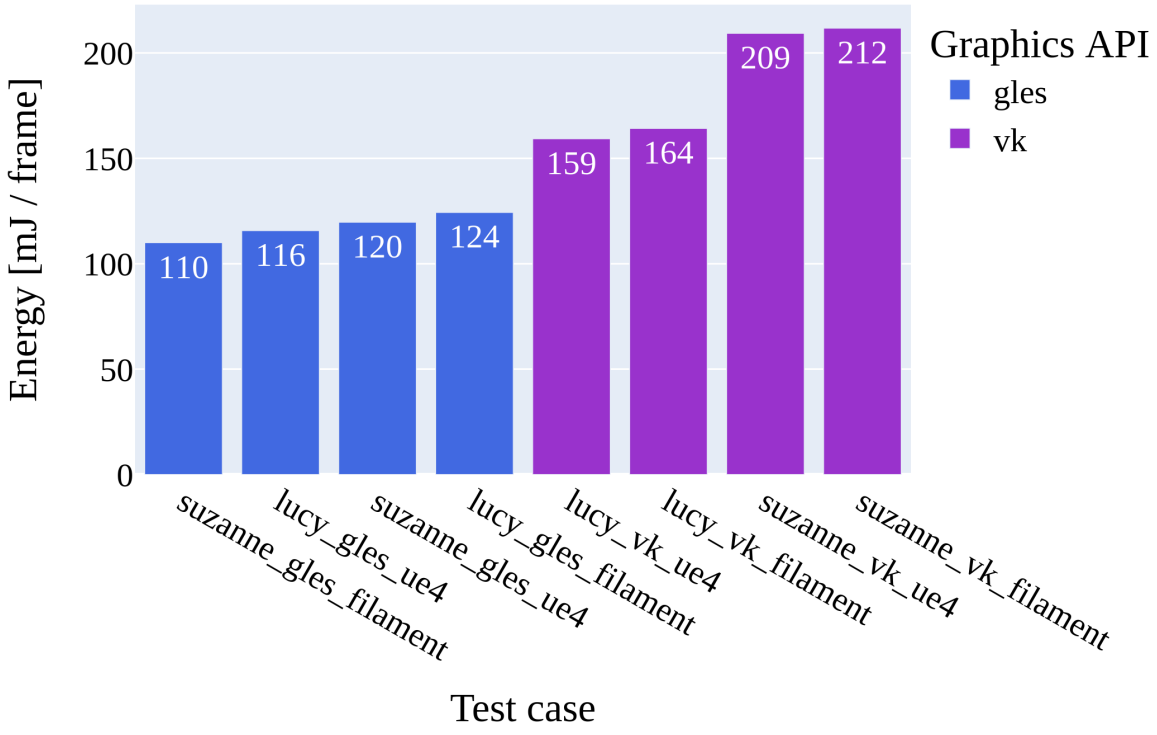


Figure 4.10: Energy per frame results comparing OpenGLES (gles) and Vulkan (vk). Lower is better

Looking at the frame energy, the OpenGLES API back-end used around 110-124 mJ per frame and the Vulkan back-end ranged from 160-212 mJ per frame, see Figure 4.10.

Comparing the graphics APIs, Vulkan to OpenGLES, the results showed Vulkan used at best 32% more energy and at worst 93% more energy than OpenGLES. Note that for Vulkan, the Lucy scenes used less energy than the Suzanne scenes. The Vulkan back-end used more energy per frame than OpenGLES even though the power consumption was lower. The lower power consumption of Vulkan was not enough to compensate for the slow frame rate.

Comparing the BRDFs, UE4 to OpenGLES, UE4's BRDF used 9% more energy compared to Filament's native BRDF. UE4's BRDF had worse energy consumption.

For frame efficiency results, see Appendix A Section A.2.

Chapter 5

Discussion

In this chapter, the results will be discussed at a higher level, reflection and improvements of the measuring process will be presented and possible future work will be suggested.

5.1 Image quality

The test cases with the most appealing image quality were the ones using the OpenGLES graphics back-end with Filament's native BRDF, see Figures 4.1a and 4.2a.

The Vulkan graphics back-end changed the shading of the frame which should not occur. This was the biggest indication that the Vulkan back-end had issues with incorrect rendering. This was a critical problem and should be solved before the Vulkan back-end could be used for practical graphics applications.

The implemented UE4 BRDF looked worse than the native Filament BRDF on all test cases. Reasons for this has already been discussed in Section 4.1.3. Better knowledge and experience was needed to pinpoint the root cause. More studies about image quality analysis would yield a deeper understanding of the topic and better comparisons would be made.

In order to change the BRDF used, the whole Filament project had to be built from scratch which could take several minutes. Due to the lengthy build time, solution on how to fix the poor image quality of the implemented UE4 BRDF into Filament was not explored.

An improvement to the image quality comparison process would be to have a static rendered frame, meaning the 3D-model would not rotate as it did in all test cases. However, the rendered frames of the test cases were visibly different and no tool was needed to compare the images. If the images were slightly different from each other then a tool for comparing images would be required for the differences to be seen.

5.2 Performance and Power

The Vulkan graphics back-end API in Google Filament performed worse than the OpenGLES back-end in all performance metrics except for power . The performance results strongly indicated that the memory bandwidth of the GPU was a possible reason for the worsened performance of choosing Vulkan over OpenGLES. The CPU usage was far less with the Vulkan

back-end and a big factor for the reduced power consumption. If the frame rate of Vulkan improved, then the energy consumption could potentially be far less than the OpenGLES back-end.

The GPU activity of both the Vulkan and OpenGLES graphics back-end APIs remained unanswered. Further investigation was needed and running the profiler for longer may give clues about these behaviours, see Figure 4.7. The test cases with the Vulkan back-end were measured only within the high activity state. If the test cases ran for a known amount of frames and the lower GPU activity state was present, then it would result in a lower average frame rate. Suggested improvement of running test cases with fixed amount of frames will be discussed in next section.

Before any measurements were conducted, the test cases were tried on an older generation phone (Samsung Galaxy S10 Exynos) and the Vulkan back-end had visibly low frame rate and the same image quality issues discussed previously. All official Android samples in the Filament repository used OpenGLES back-end by default. Many of the reported issues in the repository were Vulkan back-end related. The Google Filament project strongly suggests that the Vulkan back-end is still work in progress and not ready yet for practical use.

Investigating why the Vulkan back-end rendered incorrectly, how to improve GPU memory bandwidth bottleneck and frame rate would an interesting and important future work as the Vulkan back-end has the potential to surpass OpenGLES in terms of performance and energy consumption [5][20].

The implemented UE4 BRDF performed worse than Filament's native BRDF. As previously mentioned, the lighting model of UE4 was not implemented into the Filament project. It would be interesting to see if lighting model would correct the render quality and see how it would affect performance. Further studies and understanding was needed for better and more accurate BRDF measurements and comparisons. A proposed improvement will be given in the next section.

The initial experiment with the different scenes was to see how the number of vertices affected performance. The 3D-model Lucy had twice as many vertices than Suzanne but did not show any clear performance impact. Comparing the scenes were uninteresting for PBR and it was difficult to clearly state what was being compared. The focus was on the graphics back-end APIs and BRDFs. The comparison results of the two scenes can be found in Appendix A.3.

5.3 Potential Improvements of the Experiment

This section will discuss the measuring accuracy of the experiment, the phone used for measuring performance and power and a discussion about the PBR engine.

5.3.1 Measuring accuracy

Starting with measuring accuracy, this section will discuss the how the accuracy of the test cases and measurements could be improved.

The workload solution

Due to the V-sync frame rate limit of 120 fps, which meant that anything faster than 120 fps could not be measured, the workload was increased by rendering the same scene 10 times per frame. As previously mentioned, the initial workload was rendering the same scene 100 times per frame and later changed to 10 scenes per frame. It would be interesting to investigate how the performance and power scales with different number of scenes rendered per frame. It was a factor of 10 change in performance when changing workload from 100 scenes down to 10 scenes rendered per frame, this showed a linear characteristic. If the frame rate of rendering one scene could be estimated, then the OpenGLES back-end could render at 800 fps and Vulkan at 400 fps in theory.

When comparing workloads of rendering 100 scenes versus 10 scenes per frame, the power consumption showed similar results, see Appendix A Section A.4. This meant that power consumption did not scale linearly and the actual energy consumption per rendered scene could not be estimated. A better solution was needed to accurately estimate the energy consumption for rendering one scene per frame. Due to power not scaling linearly, due to the battery power having a dynamic and static part where the dynamic part is scaling, see Section 2.4.4. the test cases with a workload of rendering 100 scenes per frame were discarded from the final results.

The best solution for by-passing the V-sync limitation would be to render off-screen. This did not work, however this was not tested again after disabling the dynamic resolution optimization feature discussed in Section 3.3.3. Disabling the V-sync frame rate limit would give more accurate results and the Google Filament rendering overhead could potentially be measured. To render the overhead, the render-loop has to be empty (no scene rendered) and the overhead time could then be measured.

Fixed amount of frames

All the test cases ran indefinitely until the application was terminated. The issue was that the numbers of frames rendered during the run was unknown. If the test cases ran for a fixed number of frames, then the performance and energy calculations would be more accurate.

This would enable the processing of the measured results to be automated. The results would be more precise and the number of samples would could far more. The performance and power measurements were done manually which was very time consuming. Only ten consecutive frames were measured and the average frame rate, bandwidth and cycles was calculated from those ten frames. This was a very low amount of frames and by having fixed frames, the results would be much more accurate, i.e run a test case for 1000 frames while measuring time, performance and power.

Temperature and DVFS

The power measurements ran for 60 seconds and the temperature rise was only 1 degree Celsius which was too small of a change to see the temperature's impact on the power consumption. A longer measurement run would show how temperature affected the battery power consumption of the phone. The expectation would be that the power drops as temperature increases due to power consumption generates heat, see Section 2.4.4.

Another interesting characteristic would be to see how DVFS worked in practice. By measuring the frequency and voltage and see the impact of the consumption and how temperature impacts the frequency and voltage.

Frame resolution and PBR texture covering the whole frame

Instead of comparing two different scenes, a more relevant experiment for PBR would be to try different resolutions. This would work as the BRDF operates on pixels in the fragment processing stage and not on vertices in the vertex processing stage, see Figure 2.2. This experiment could tell how well the Filament engine would perform on higher resolutions.

A better way to test the BRDF shading models would be to occupy the whole frame with a PBR texture. This would mean that every pixel in the rendered frame would go through the BRDF shader and thus more accurate measurements of the BRDFs would be acquired.

5.3.2 Mobile phones

All the measurements were measured on a Asus ROG 6D phone and the other phone considered was the Vivo X90 Pro which had older memory technology, see Table 2.1. Given from the results, the GPU memory bandwidth was a possible bottleneck for the Vulkan back-end graphics API. By testing on the Vivo X90 Pro, the bandwidth bottleneck could further be verified as the phone has a newer GPU but worse memory technology than the ASUS ROG 6D. If the Vivo X90 Pro would have same or worse performance than the Asus ROG 6D - then the memory bandwidth bottleneck would be concluded as the possible root cause for the poor performance of the Vulkan graphics back-end.

Suggested future work would be to try on multiple mobile phones with different GPUs.

5.3.3 PBR engines

Google Filament engine was the only PBR engine used for measuring power and performance of PBR on mobile. Suggested future work would be to measure PBR using other engines such as the latest Unreal Engine or Unity. The main challenge would probably be to create good comparable test cases across the different engines.

5.4 Environmental Impact

As previously discussed, the Vulkan back-end has the potential to reduce the energy consumption if the performance issues were solved. With lower energy consumption comes longer battery life and less frequent charging. This means that energy could be saved and as most people own a mobile phone, the impact would be noticeable and less energy usage is better for the environment.

PBR is an easy rendering technique to implement which does not require any special dedicated hardware. This means it could run on older devices and there would be no need to buy a newer phone for using advanced looking graphics. This would improve the longevity of the mobile phone and reduce electronic waste as the phone does not need to be replaced by a newer one.

Chapter 6

Conclusion

The PBR engine Google Filament was used to conduct the experiment of measuring PBR on mobile. The choice of back-end graphics API impacted the image quality, performance and energy consumption severely. Choosing Vulkan over OpenGLES resulted in 50% worse performance, the image quality showed faulty rendering and the energy consumption was 32% higher than OpenGLES. The GPU bandwidth was twice as large using Vulkan which indicated a possible memory bandwidth bottleneck. The Vulkan graphics back-end was clearly still work in-progress and continues to improve in the Filament project.

The experiment of comparing Filament's native BRDF to the implemented UE4 BRDF showed that UE4's BRDF performed worse by 6% and used 9% more energy than Filaments native BRDF. The rendered image quality of UE4's BRDF with the Vulkan back-end was considered unacceptable for PBR. Reasons for the worsened image quality of UE4's BRDF could be erroneous implementation and inaccurate measurement procedure for comparing BRDFs.

Possible improvements of the experiments and future work would be to investigate how the Vulkan back-end could be optimized and conduct more accurate experiments for comparing BRDFs.

Bibliography

[1] T. Hiranyachattada and K. Kusirirat, *Using mobile augmented reality to enhancing students' conceptual understanding of physically-based rendering in 3d animation.* Nov. 2019. [Online]. Available: https://eric.ed.gov/?id=EJ1242139.

[2] A. Ace 3D Studio, *Physically-based rendering: Using pbr for games, animations, and more*, May 2023. [Online]. Available: https://3d-ace.com/blog/physically-based-rendering-using-pbr-for-games-animations-and-more/.

[3] M. Short, *Physically based shading on mobile*, Oct. 2018. [Online]. Available: https://medium.com/spaceapetech/physically-based-shading-on-mobile-d7d4e90bb4bd.

[4] M. Pharr, W. Jakob, and G. Humphreys, *Physically Based Rendering: From Theory to Implementation.* Elsevier Science, 2016, ISBN: 9780128007099.

[5] M. Lujan, M. Baum, D. Chen, and Z. Zong, "Evaluating the performance and energy efficiency of opengl and vulkan on a graphics rendering server," in *2019 International Conference on Computing, Networking and Communications (ICNC)*, Feb. 2019, pp. 777–781. DOI: 10.1109/ICCNC.2019.8685588.

[6] M. Lujan, M. McCrary, B. W. Ford, and Z. Zong, "Vulkan vs opengl es: Performance and energy efficiency comparison on the big.little architecture," in *2021 IEEE International Conference on Networking, Architecture and Storage (NAS)*, Oct. 2021, pp. 1–8. DOI: 10.1109/NAS51552.2021.9605447.

[7] X. D. He, K. E. Torrance, F. X. Sillion, and D. P. Greenberg, "A comprehensive physical model for light reflection," in *Proceedings of the 18th Annual Conference on Computer Graphics and Interactive Techniques*, ser. SIGGRAPH '91, New York, NY, USA: Association for Computing Machinery, 1991, pp. 175–186, ISBN: 0897914368. DOI: 10.1145/122718.122738.

[8] J. F. Blinn, "Models of light reflection for computer synthesized pictures," in *Proceedings of the 4th Annual Conference on Computer Graphics and Interactive Techniques*, ser. SIGGRAPH '77, San Jose, California: Association for Computing Machinery, 1977, pp. 192–198, ISBN: 9781450373555. DOI: 10.1145/563858.563893.

[9] J. F. Blinn, "Light reflection functions for simulation of clouds and dusty surfaces," *SIGGRAPH Comput. Graph.*, vol. 16, no. 3, pp. 21–29, Jul. 1982, ISSN: 0097-8930. DOI: 10.1145/965145.801255.

[10] P. Shirley, B. Smits, H. Hu, and E. Lafortune, "A practitioners' assessment of light reflection models," in *Proceedings The Fifth Pacific Conference on Computer Graphics and Applications*, 1997, pp. 40–49. DOI: 10.1109/PCCGA.1997.626170.

[11] M. Kenzel, B. Kerbl, D. Schmalstieg, and M. Steinberger, "A high-performance software graphics pipeline architecture for the gpu," *ACM Trans. Graph.*, vol. 37, no. 4, Jul. 2018, ISSN: 0730-0301. DOI: 10.1145/3197517.3201374.

[12] J. Foley, *Computer Graphics: Principles and Practice* (Addison-Wesley systems programming series). Addison-Wesley, 1996, ISBN: 9780201848403.

[13] J. Sugerman, K. Fatahalian, S. Boulos, K. Akeley, and P. Hanrahan, "Gramps: A programming model for graphics pipelines," *ACM Trans. Graph.*, vol. 28, no. 1, Feb. 2009, ISSN: 0730-0301. DOI: 10.1145/1477926.1477930.

[14] T. Akenine-Möller, E. Haines, and N. Hoffman, *Real-Time Rendering*. CRC Press, 2019, ISBN: 9781315362007.

[15] S. Buss, *3D Computer Graphics: A Mathematical Introduction with OpenGL* (3-D Computer Graphics: A Mathematical Introduction with OpenGL). Cambridge University Press, 2003, ISBN: 9780521821032.

[16] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips, "Gpu computing," *Proceedings of the IEEE*, vol. 96, no. 5, pp. 879–899, May 2008, ISSN: 1558-2256. DOI: 10.1109/JPROC.2008.917757.

[17] D. Ginsburg, B. Purnomo, D. Shreiner, and A. Munshi, *OpenGL ES 3.0 Programming Guide* (OpenGL). Pearson Education, 2014, ISBN: 9780133440126.

[18] H. Lee and N. Baek, "Implementing opengl es on opengl," in *2009 IEEE 13th International Symposium on Consumer Electronics*, May 2009, pp. 999–1003. DOI: 10.1109/ISCE.2009.5156990.

[19] G. Sellers and J. Kessenich, *Vulkan Programming Guide: The Official Guide to Learning Vulkan* (OpenGL). Pearson Education, 2016, ISBN: 9780134464688.

[20] M. Gambhir, S. Panda, and S. J. Basha, "Vulkan rendering framework for mobile multimedia," in *SIGGRAPH Asia 2018 Posters*, ser. SA '18, Tokyo, Japan: Association for Computing Machinery, 2018, ISBN: 9781450360630. DOI: 10.1145/3283289.3283336.

[21] A. H. Hormati, M. Samadi, M. Woh, T. Mudge, and S. Mahlke, "Sponge: Portable stream programming on graphics engines," in *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XVI, Newport Beach, California, USA: Association for Computing Machinery, 2011, pp. 381–392, ISBN: 9781450302661. DOI: 10.1145/1950365.1950409.

[22] A. Sanders, *An Introduction to Unreal Engine 4*. CRC Press, 2016, ISBN: 9781498765107.

[23] B. Karis, "Real shading in unreal engine 4," Epic Games, Tech. Rep., 2013. [Online]. Available: http://blog.selfshadow.com/publications/s2013-shading-course/karis/s2013_pbs_epic_notes_v2.pdf (visited on 02/05/2016).

[24] J. K. Haas, "A history of the unity game engine," *Diss. Worcester Polytechnic Institute*, vol. 483, no. 2014, p. 484, 2014.

[25] P. Rideout, *Getting started with filament on android*, May 2020. [Online]. Available: https://medium.com/@philiprideout/getting-started-with-filament-on-android-d10b16f0ec67.

[26] P. Shirley, R. K. Morley, P.-P. Sloan, and C. Wyman, "Basics of physically-based rendering," in *SIGGRAPH Asia 2012 Courses*, ser. SA '12, Singapore, Singapore: Association for Computing Machinery, 2012, ISBN: 9781450319133. DOI: 10.1145/2407783.2407785.

[27] J. R. Wallace, M. F. Cohen, and D. P. Greenberg, "A two-pass solution to the rendering equation: A synthesis of ray tracing and radiosity methods," *SIGGRAPH Comput. Graph.*, vol. 21, no. 4, pp. 311–320, Aug. 1987, ISSN: 0097-8930. DOI: 10.1145/37402.37438.

[28] J. T. Kajiya, "The rendering equation," in *Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques*, ser. SIGGRAPH '86, New York, NY, USA: Association for Computing Machinery, 1986, pp. 143–150, ISBN: 0897911962. DOI: 10.1145/15922.15902.

[29] M. Ashikmin, S. Premože, and P. Shirley, "A microfacet-based brdf generator," in *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*, ser. SIGGRAPH '00, USA: ACM Press/Addison-Wesley Publishing Co., 2000, pp. 65–74, ISBN: 1581132085. DOI: 10.1145/344779.344814.

[30] M. Capderou, "Confirmation of helmholtz reciprocity using scarab satellite data," *Remote Sensing of Environment*, vol. 64, no. 3, pp. 266–285, 1998, ISSN: 0034-4257. DOI: https://doi.org/10.1016/S0034-4257(98)00004-2.

[31] B. Walter, S. R. Marschner, H. Li, and K. E. Torrance, "Microfacet models for refraction through rough surfaces," in *Proceedings of the 18th Eurographics Conference on Rendering Techniques*, ser. EGSR'07, Grenoble, France: Eurographics Association, 2007, pp. 195–206, ISBN: 9783905673524.

[32] A. Kumar, *Beginning PBR texturing: Learn physically based rendering with allegorithmic's substance painter*. Apress, 2020, ISBN: 978-1-4842-5899-6. DOI: 10.1007/978-1-4842-5899-6.

[33] FreePBR, *Free pbr*. [Online]. Available: https://freepbr.com/.

[34] P. Shanmugam and O. Arikan, "Hardware accelerated ambient occlusion techniques on gpus," ser. I3D '07, Seattle, Washington: Association for Computing Machinery, 2007, pp. 73–80, ISBN: 9781595936288. DOI: 10.1145/1230100.1230113.

[35] T. Whitted, "An improved illumination model for shaded display," in *Proceedings of the 6th Annual Conference on Computer Graphics and Interactive Techniques*, ser. SIGGRAPH '79, Chicago, Illinois, USA: Association for Computing Machinery, 1979, p. 14, ISBN: 0897910044. DOI: 10.1145/800249.807419.

[36] R. L. Cook and K. E. Torrance, "A reflectance model for computer graphics," *ACM Trans. Graph.*, vol. 1, no. 1, pp. 7–24, Jan. 1982, ISSN: 0730-0301. DOI: 10.1145/357290.357293.

[37] C. M. Goral, K. E. Torrance, D. P. Greenberg, and B. Battaile, "Modeling the interaction of light between diffuse surfaces," *SIGGRAPH Comput. Graph.*, vol. 18, no. 3, pp. 213–222, Jan. 1984, ISSN: 0097-8930. DOI: 10.1145/964965.808601.

[38] M. F. Cohen and D. P. Greenberg, "The hemi-cube: A radiosity solution for complex environments," in *Proceedings of the 12th Annual Conference on Computer Graphics and Interactive Techniques*, ser. SIGGRAPH '85, New York, NY, USA: Association for Computing Machinery, 1985, pp. 31–40, ISBN: 0897911660. DOI: 10.1145/325334.325171.

[39] T. Nishita and E. Nakamae, "Continuous tone representation of three-dimensional objects taking account of shadows and interreflection," *SIGGRAPH Comput. Graph.*, vol. 19, no. 3, pp. 23–30, Jul. 1985, ISSN: 0097-8930. DOI: 10.1145/325165.325169.

[40] R. L. Cook, T. Porter, and L. Carpenter, "Distributed ray tracing," *SIGGRAPH Comput. Graph.*, vol. 18, no. 3, pp. 137–145, Jan. 1984, ISSN: 0097-8930. DOI: 10.1145/964965.808590.

[41] T. Hiranyachattada and K. Kusirirat, *Using mobile augmented reality to enhancing students' conceptual understanding of physically-based rendering in 3d animation.* Nov. 2019. [Online]. Available: https://eric.ed.gov/?id=EJ1242139.

[42] G. Psomathianos, N. Sourdakos, and K. Moustakas, "Smoke diffusion simulation and physically-based rendering for vr," in *2021 International Conference on Cyberworlds (CW)*, Sep. 2021, pp. 117–120. DOI: 10.1109/CW52790.2021.00025.

[43] Y. Zhang, S. Song, E. Yumer, *et al.*, "Physically-based rendering for indoor scene understanding using convolutional neural networks," in *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Jul. 2017, pp. 5057–5065. DOI: 10.1109/CVPR.2017.537.

[44] P. Dai, Z. Li, Y. Zhang, S. Liu, and B. Zeng, "Pbr-net: Imitating physically based rendering using deep neural network," *IEEE Transactions on Image Processing*, vol. 29, pp. 5980–5992, 2020, ISSN: 1941-0042. DOI: 10.1109/TIP.2020.2987169.

[45] M. Aranha, P. Dubla, K. Debattista, T. Bashford-Rogers, and A. Chalmers, "A physically-based client-server rendering solution for mobile devices," in *Proceedings of the 6th International Conference on Mobile and Ubiquitous Multimedia*, ser. MUM '07, Oulu, Finland: Association for Computing Machinery, 2007, pp. 149–154, ISBN: 9781595939166. DOI: 10.1145/1329469.1329489.

[46] K. H. Baek, Y. Ji, H. W. Jin, and T. S. Yun, "Game engine pbr for background cgi production of live-action contents," in *2019 IEEE Conference on Graphics and Media (GAME)*, Nov. 2019, pp. 18–21. DOI: 10.1109/GAME47560.2019.8980984.

[47] J. Y. C. Chen and J. E. Thropp, "Review of low frame rate effects on human performance," *IEEE Transactions on Systems, Man, and Cybernetics - Part A: Systems and Humans*, vol. 37, no. 6, pp. 1063–1076, Nov. 2007, ISSN: 1558-2426. DOI: 10.1109/TSMCA.2007.904779.

[48] H. K. Galoogahi, A. Fagg, C. Huang, D. Ramanan, and S. Lucey, "Need for speed: A benchmark for higher frame rate object tracking," in *2017 IEEE International Conference on Computer Vision (ICCV)*, 2017, pp. 1134–1143. DOI: 10.1109/ICCV.2017.128.

[49] R. Prasad, C. Dovrolis, M. Murray, and K. Claffy, "Bandwidth estimation: Metrics, measurement techniques, and tools," *IEEE Network*, vol. 17, no. 6, pp. 27–35, Nov. 2003, ISSN: 1558-156X. DOI: 10.1109/MNET.2003.1248658.

[50] D. Burger, J. R. Goodman, and A. Kägi, "Memory bandwidth limitations of future microprocessors," *SIGARCH Comput. Archit. News*, vol. 24, no. 2, pp. 78–89, May 1996, ISSN: 0163-5964. DOI: 10.1145/232974.232983.

[51] J.-C. Tuan, T.-S. Chang, and C.-W. Jen, "On the data reuse and memory bandwidth analysis for full-search block-matching vlsi architecture," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 12, no. 1, pp. 61–72, Jan. 2002, ISSN: 1558-2205. DOI: 10.1109/76.981846.

[52] M. Swanson, B. Zhu, and A. Tewfik, "Data hiding for video-in-video," in *Proceedings of International Conference on Image Processing*, vol. 2, Oct. 1997, 676–679 vol.2. DOI: 10.1109/ICIP.1997.638586.

[53] W. Yuan and K. Nahrstedt, "Energy-efficient cpu scheduling for multimedia applications," *ACM Trans. Comput. Syst.*, vol. 24, no. 3, pp. 292–331, Aug. 2006, ISSN: 0734-2071. DOI: 10.1145/1151690.1151693.

[54] Y. Yang, H. Jiang, Y. Wu, Y. Lv, X. Li, and G. Xie, "C2qos: Cpu-cycle based network qos strategy in vswitch of public cloud," in *2021 IFIP/IEEE International Symposium on Integrated Network Management (IM)*, May 2021, pp. 438–444.

[55] Y. Oyama, "How does malware use rdtsc? a study on operations executed by malware with cpu cycle measurement," in *Detection of Intrusions and Malware, and Vulnerability Assessment*, R. Perdisci, C. Maurice, G. Giacinto, and M. Almgren, Eds., Cham: Springer International Publishing, 2019, pp. 197–218, ISBN: 978-3-030-22038-9.

[56] A. von Meier, *Electric Power Systems: A Conceptual Introduction* (Wiley Survival Guides in Engineering and Science). Wiley, 2006, ISBN: 9780470036402.

[57] S. Mittal, "A survey of architectural techniques for improving cache power efficiency," *Sustainable Computing: Informatics and Systems*, vol. 4, no. 1, pp. 33–43, 2014, ISSN: 2210-5379. DOI: https://doi.org/10.1016/j.suscom.2013.11.001.

[58] D. Price, M. Clark, B. Barsdell, R. Babich, and L. Greenhill, "Optimizing performance per watt on gpus in high performance computing: Temperature, frequency and voltage effects," *Computer Science - Research and Development*, vol. 31, Nov. 2016. DOI: 10.1007/s00450-015-0300-5.

[59] S. Herbert and D. Marculescu, "Analysis of dynamic voltage/frequency scaling in chip-multiprocessors," in *Proceedings of the 2007 International Symposium on Low Power Electronics and Design*, ser. ISLPED '07, Portland, OR, USA: Association for Computing Machinery, 2007, pp. 38–43, ISBN: 9781595937094. DOI: 10.1145/1283780.1283790.

[60] R. A. Bridges, N. Imam, and T. M. Mintz, "Understanding gpu power: A survey of profiling, modeling, and simulation methods," vol. 49, no. 3, Sep. 2016, ISSN: 0360-0300. DOI: 10.1145/2962131.

[61] S. Mittal and J. S. Vetter, "A survey of methods for analyzing and improving gpu energy efficiency," *ACM Comput. Surv.*, vol. 47, no. 2, Aug. 2014, ISSN: 0360-0300. DOI: 10.1145/2636342.

[62] T. Akenine-Möller and B. Johnsson, "Performance per what?" *Journal of Computer Graphics Techniques (JCGT)*, vol. 1, no. 1, pp. 37–41, 2012, ISSN: 2331-7418.

[63] T. Hagos, "Android studio profiler," in *Android Studio IDE Quick Reference: A Pocket Guide to Android Studio Development*. Berkeley, CA: Apress, 2019, pp. 73–82, ISBN: 978-1-4842-4953-6. DOI: 10.1007/978-1-4842-4953-6_7.

[64] L. Ivankin, *Comparing perfetto with android profiler*, Sep. 2022. [Online]. Available: https://hackernoon.com/comparing-perfetto-with-android-profiler.

[65] A. Gutierrez, R. G. Dreslinski, T. F. Wenisch, *et al.*, "Full-system analysis and characterization of interactive smartphone applications," in *2011 IEEE International Symposium on Workload Characterization (IISWC)*, Nov. 2011, pp. 81–90. DOI: 10.1109/IISWC.2011.6114205.

[66] M. Di Paolo Emilio, *Data Acquisition Systems: From Fundamentals to Applied Design*. Jan. 2013, ISBN: 978-1-4614-4213-4. DOI: 10.1007/978-1-4614-4214-1.

[67] C. Doppioslash, "When shading goes wrong," in *Physically Based Shader Development for Unity 2017: Develop Custom Lighting Systems*. Berkeley, CA: Apress, 2018, pp. 217–224, ISBN: 978-1-4842-3309-2. DOI: 10.1007/978-1-4842-3309-2_17.

[68] H. Jang, S. Lee, J.-J. Lee, and K. Han, "Releasing the memory bottleneck to display video correctly," in *2022 19th International SoC Design Conference (ISOCC)*, Oct. 2022, pp. 340–341. DOI: 10.1109/ISOCC56007.2022.10031379.

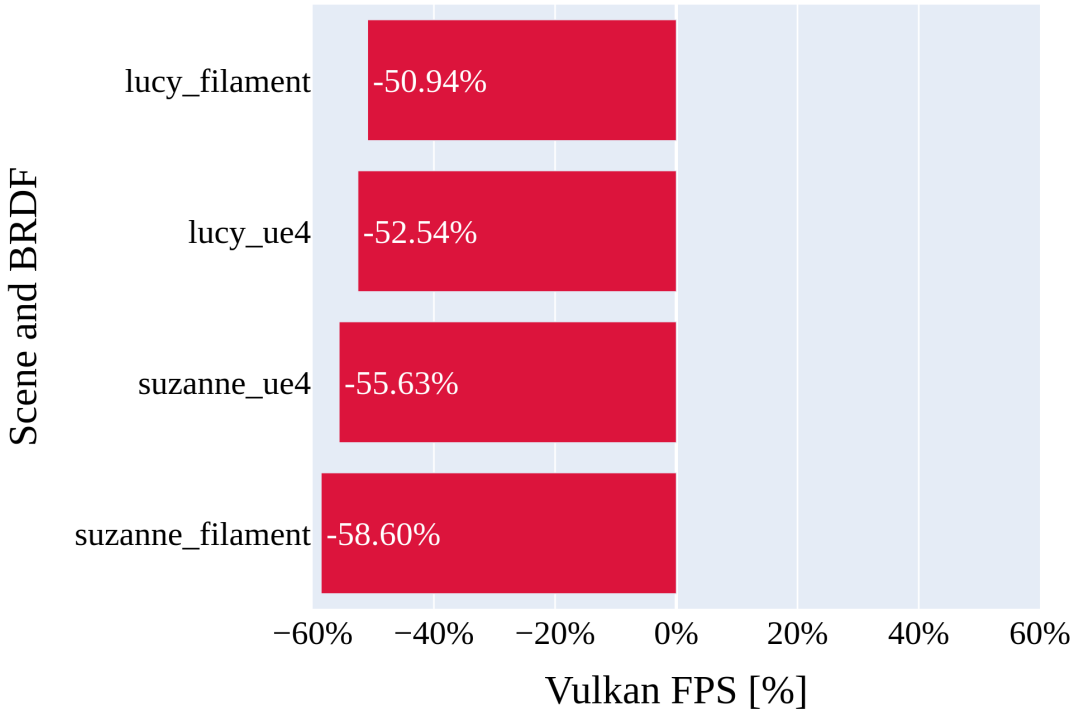
[69] D. Rahme, *Danielrahme/filament: Google filament repository*. [Online]. Available: https://github.com/DanielRahme/filament.

Appendix A

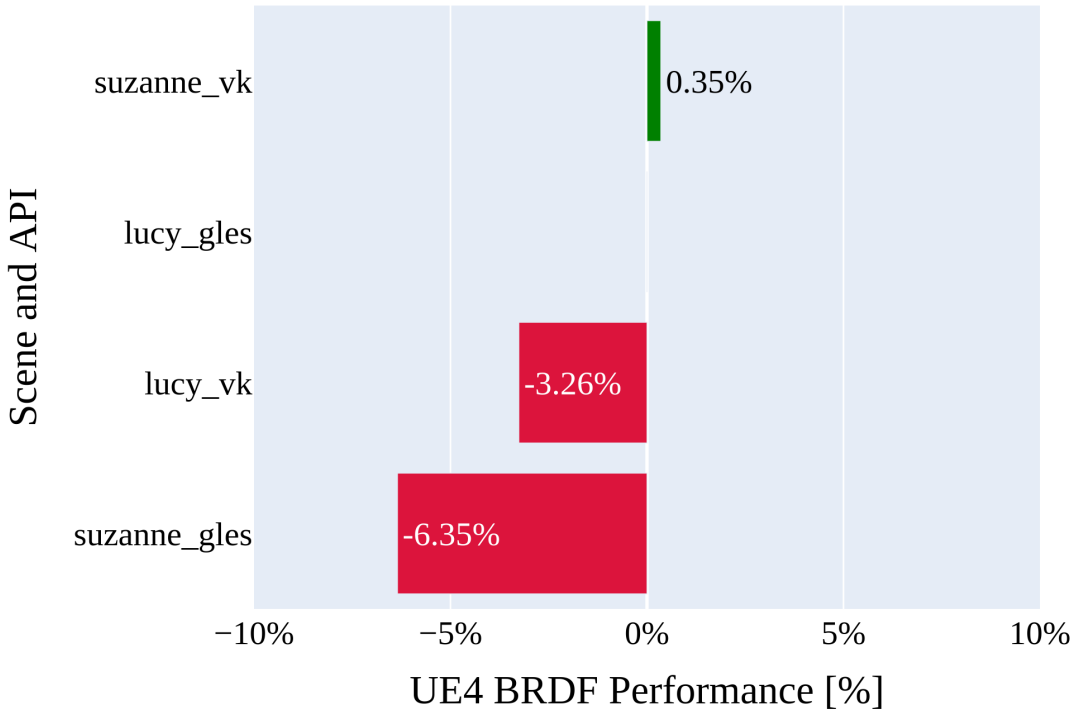
Appendix 1 Results

In this appendix chapter the figures of the results are presented.

A.1 Comparisons of graphics API and BRDF shaders

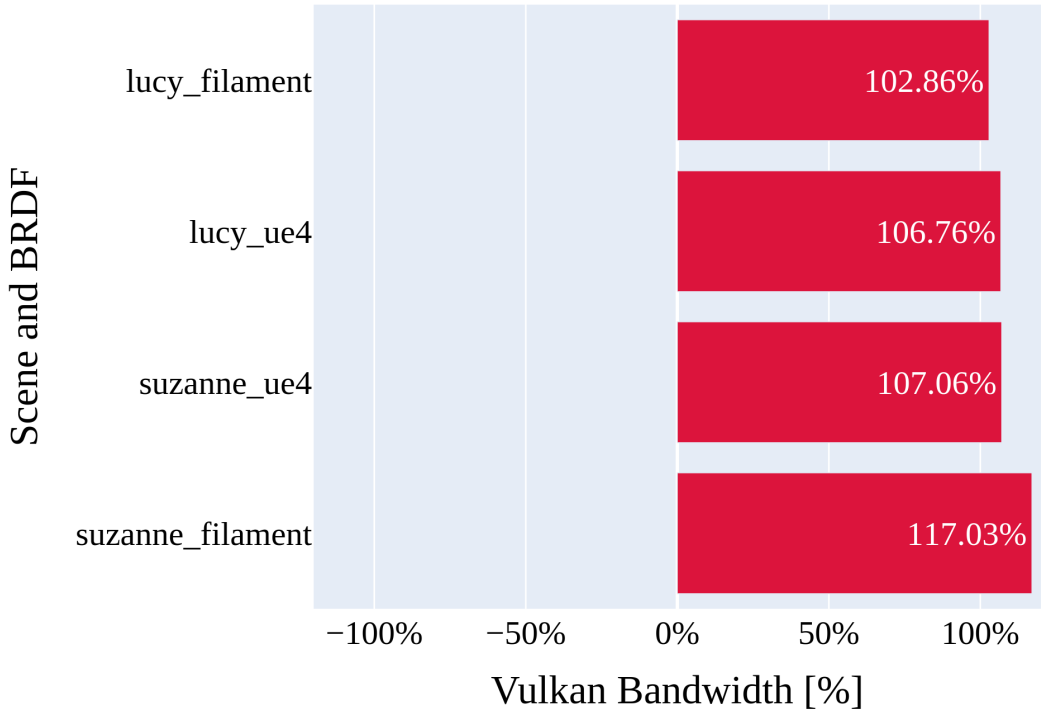


(a) Vulkan vs OpenGLES.

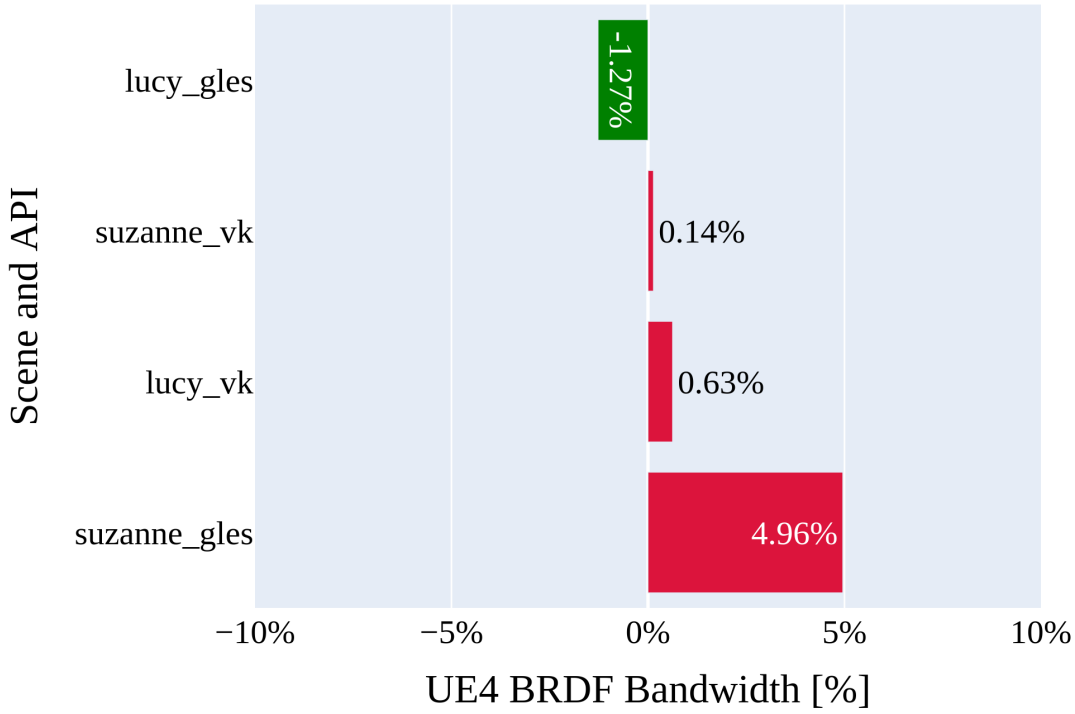


(b) UE4 vs Filament.

Figure A.1: Frame rate comparisons.

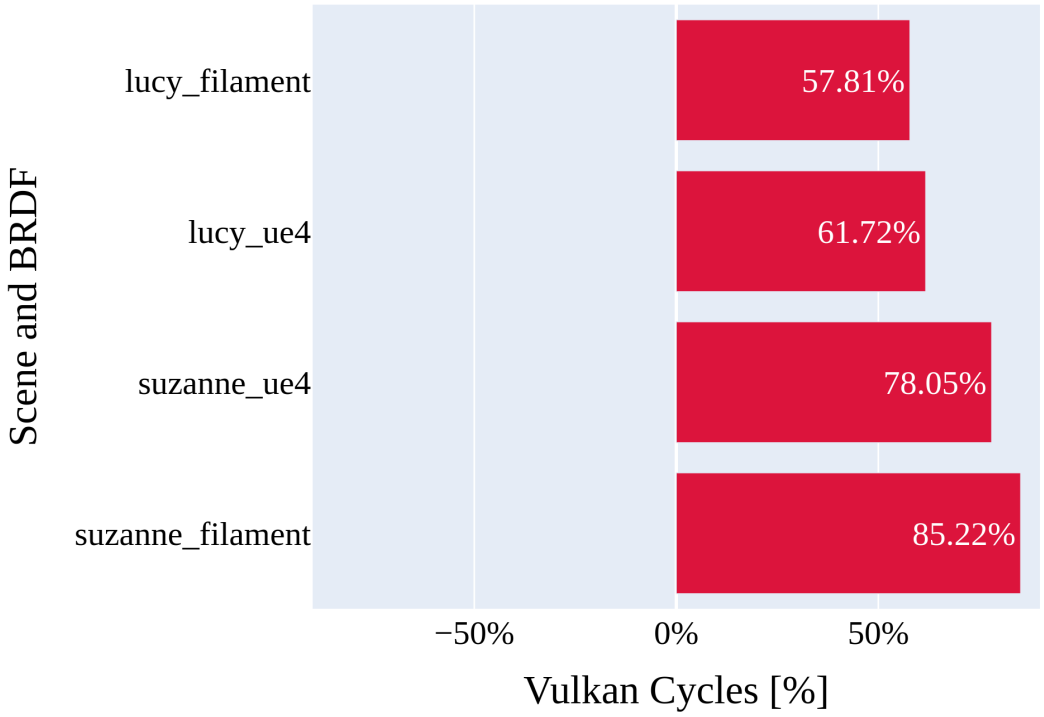


(a) Bandwidth Graphics APIs.

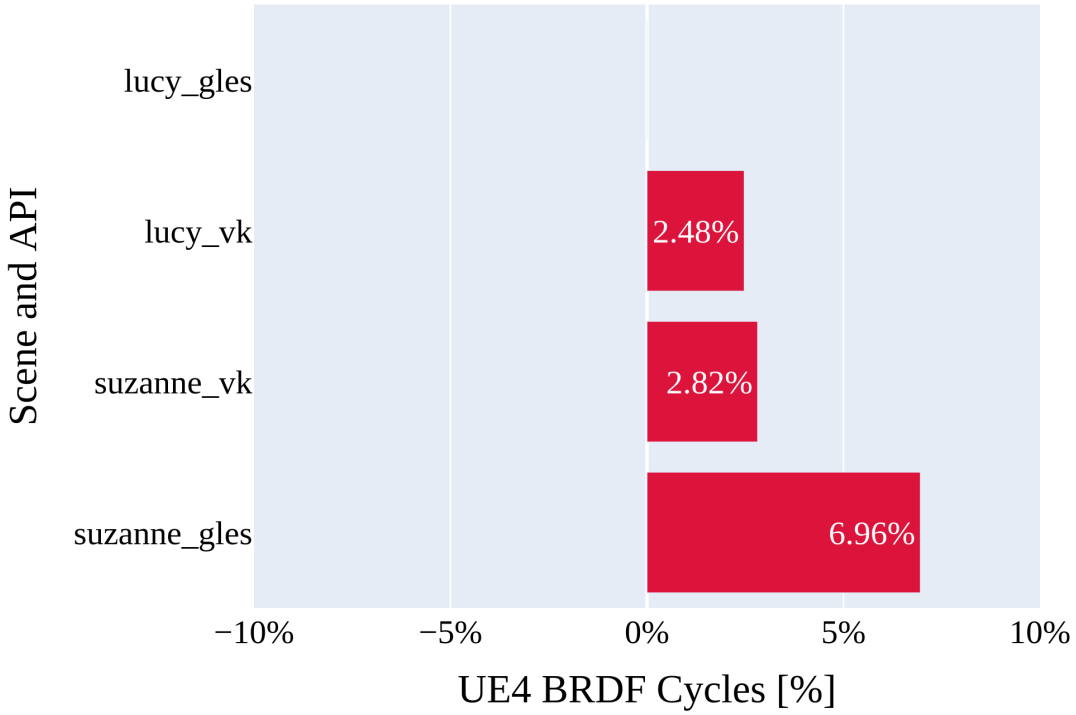


(b) Bandwidth BRDFs.

Figure A.2: Frame rate comparisons

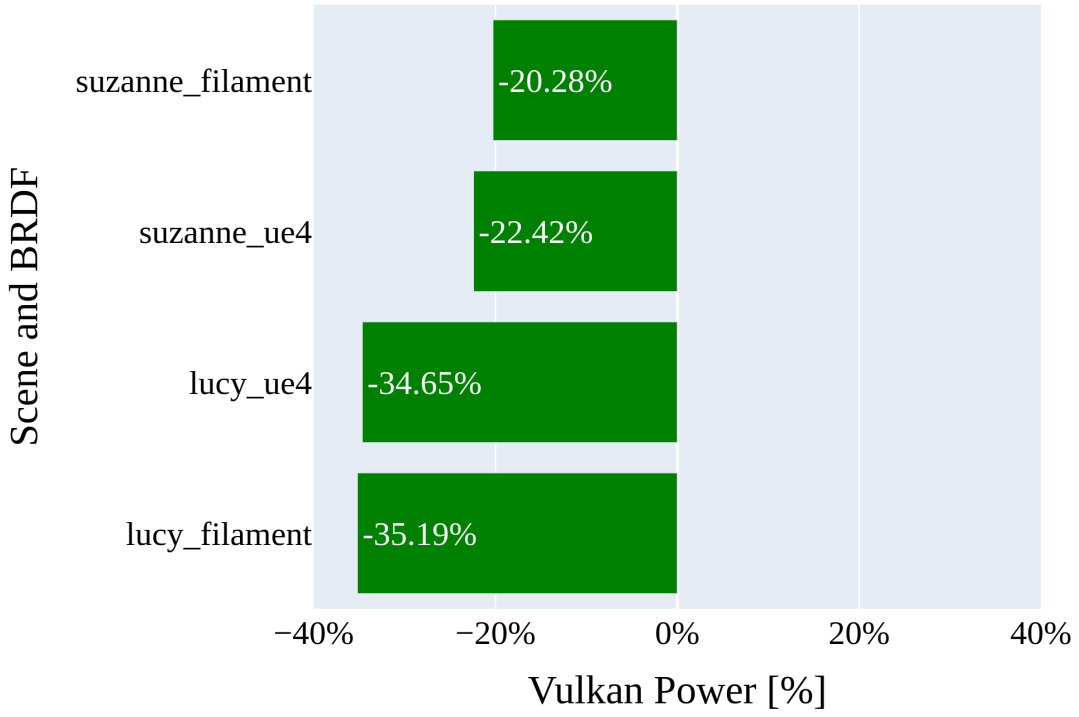


(a) Cycles Graphics APIs.

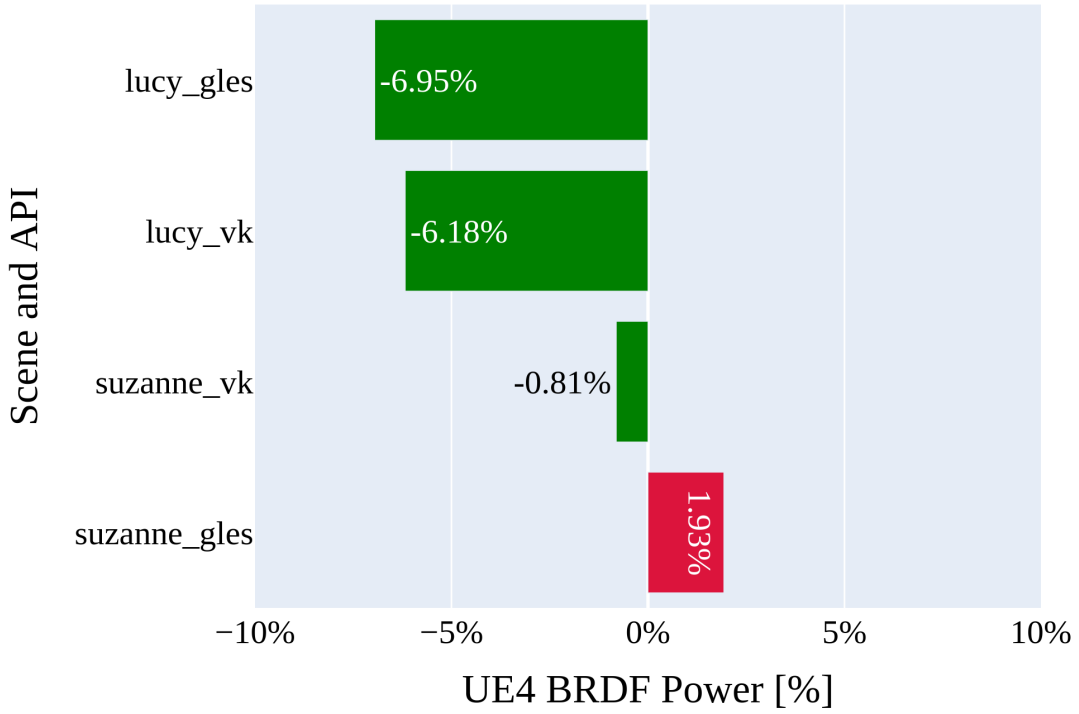


(b) Cycles BRDFs.

Figure A.3: GPU cycles results.

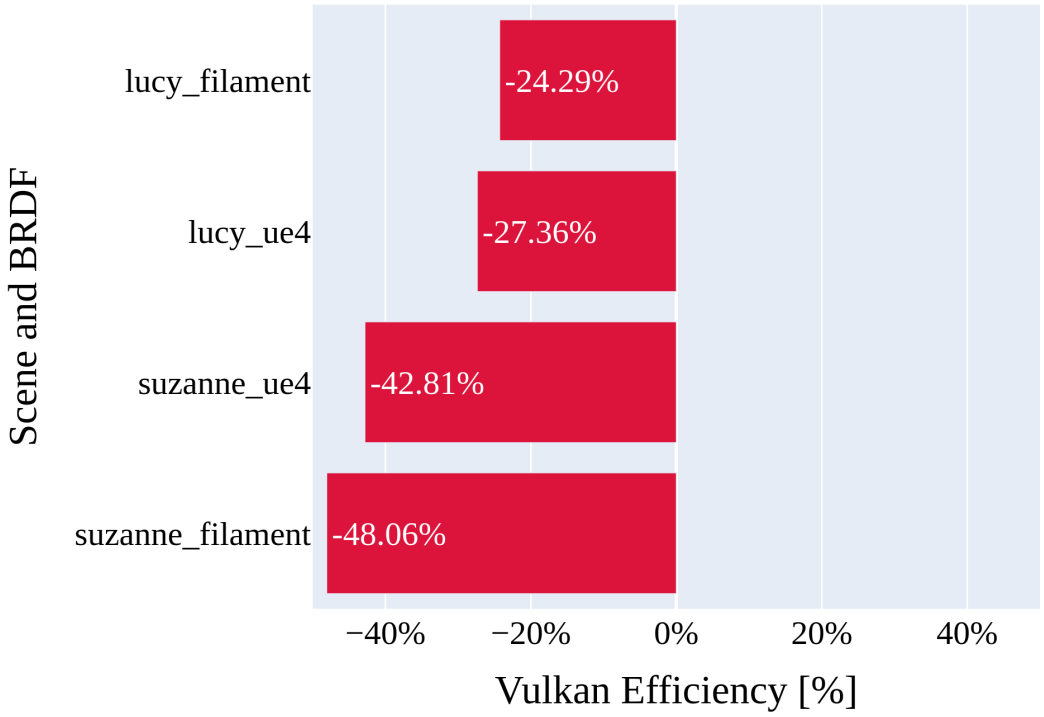


(a) Graphics API: Vulkan vs OpenGLES.

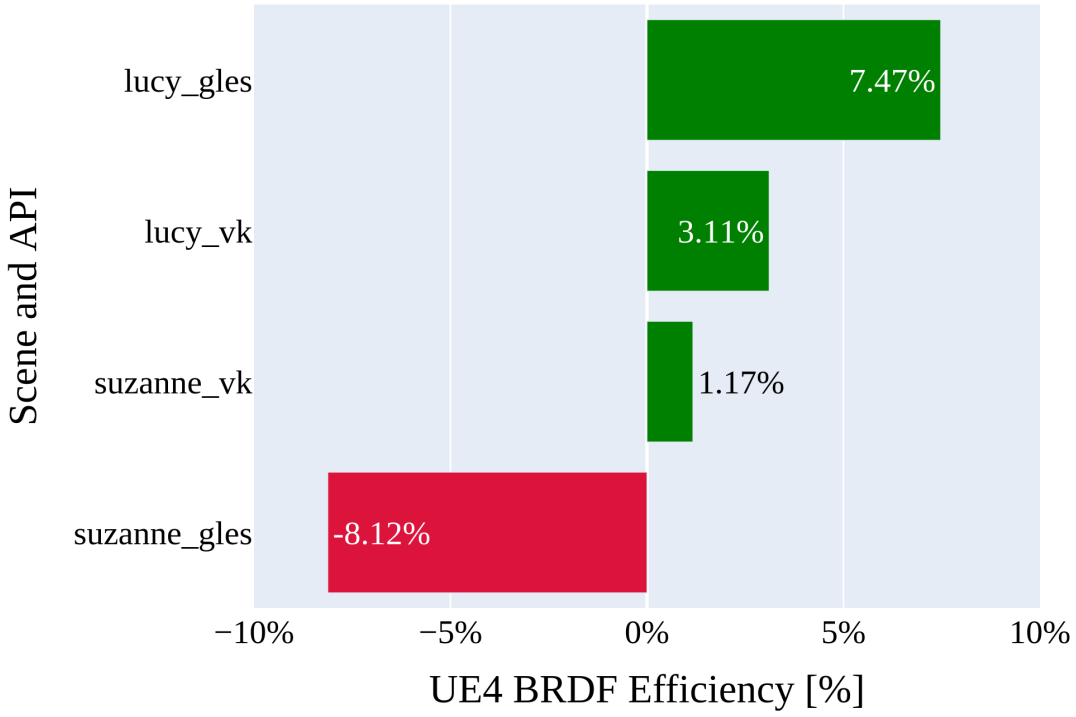


(b) BRDFs: UE4 vs Filament.

Figure A.4: Power results.

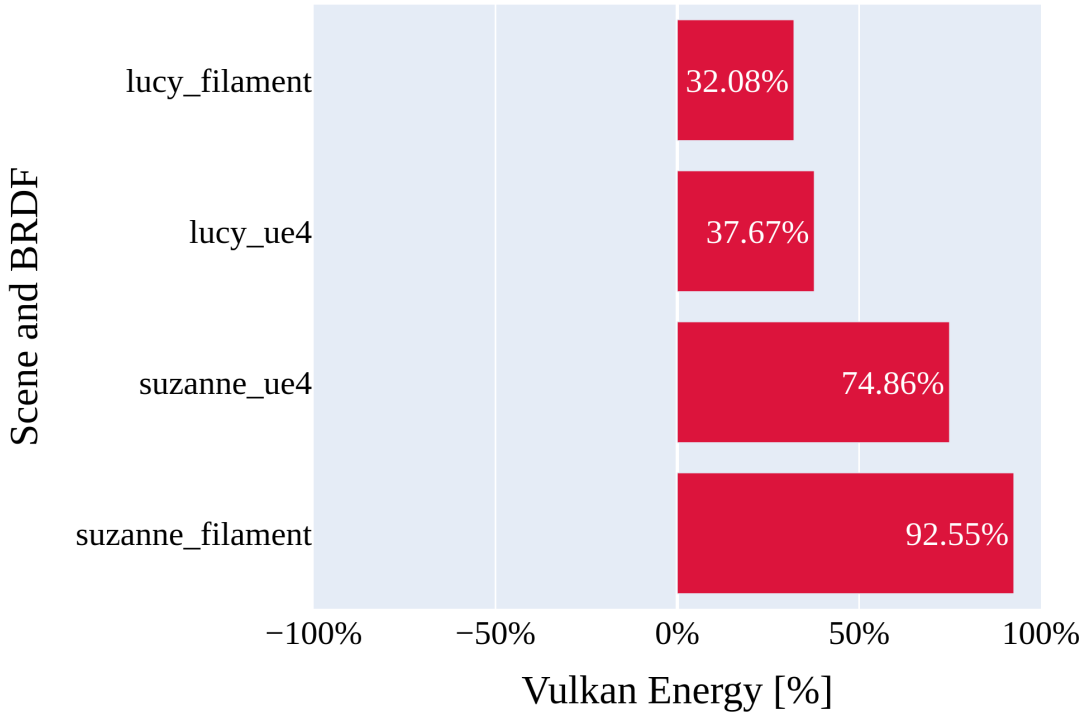


(a) Efficiency APIs.

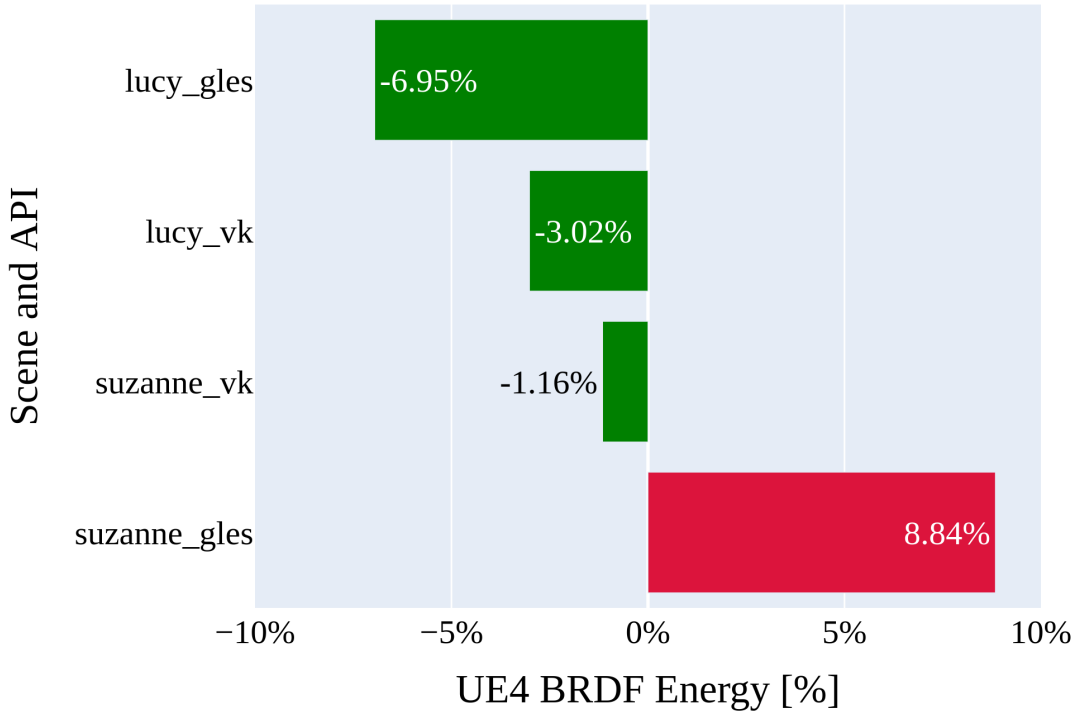


(b) Efficiency BRDFs.

Figure A.5: Frame efficiency results.



(a) Energy: Vulkan vs OpenGLES.



(b) Energy: Filament vs UE4.

Figure A.6: Frame energy results.

A.2 Frame efficiency

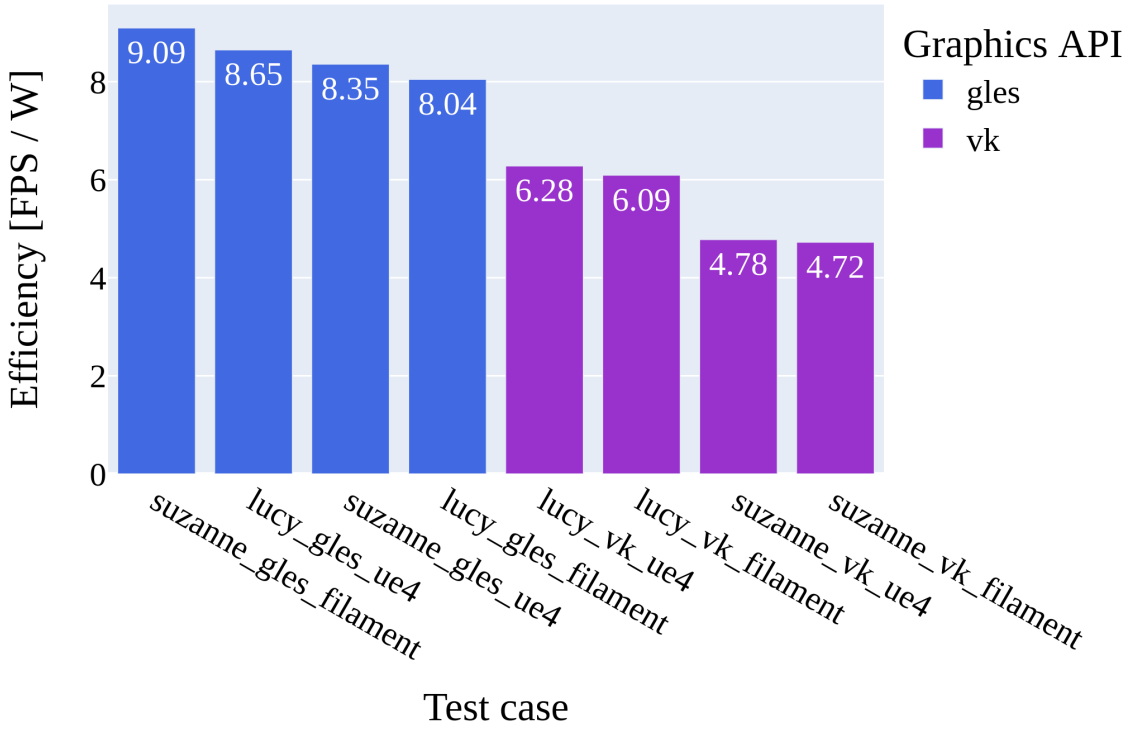


Figure A.7: Absolute frame efficiency.

Looking at the frame efficiency, the OpenGLES test cases had around 8-9 fps per Watt and Vulkan around 5-6 fps per Watt, see figure A.7. Comparing the graphics APIs, Vulkan to OpenGLES, the results showed Vulkan was at best 24% less efficient and at worst 48% less efficient than OpenGLES, see figure A.5a. Comparing the BRDF shaders, UE4 to Filament, the result showed that UE4 was 8% less efficient compared to Filament, see figure A.5b.

A.3 Plots of Scene Comparisons: Suzanne vs Lucy

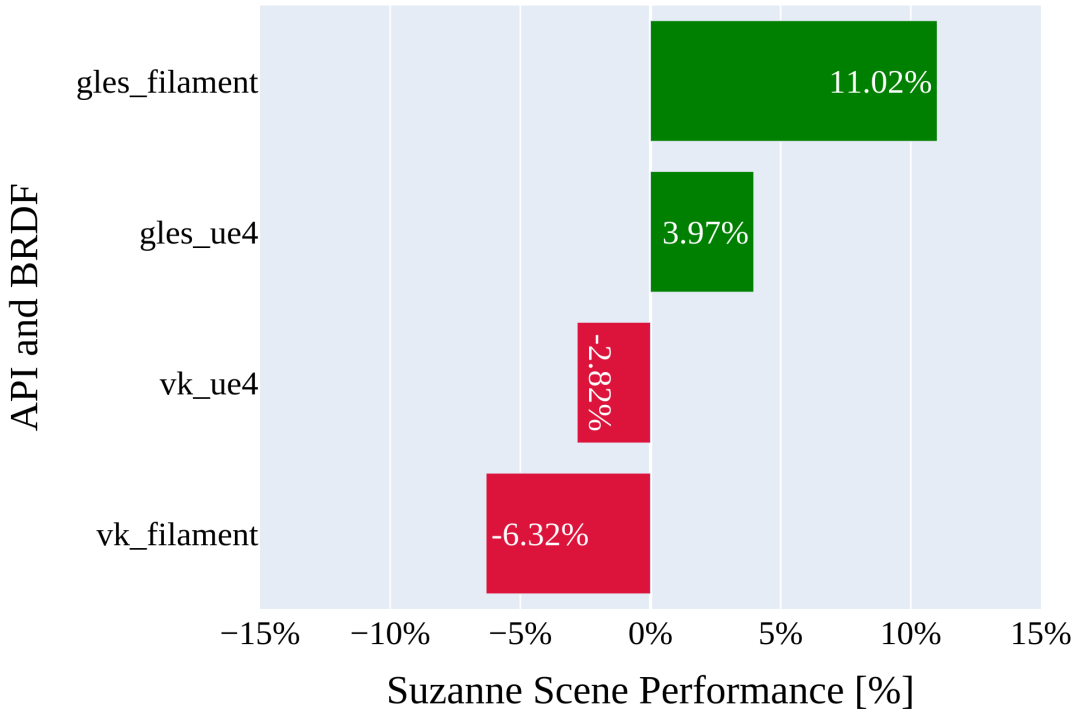


Figure A.8: Results FPS.

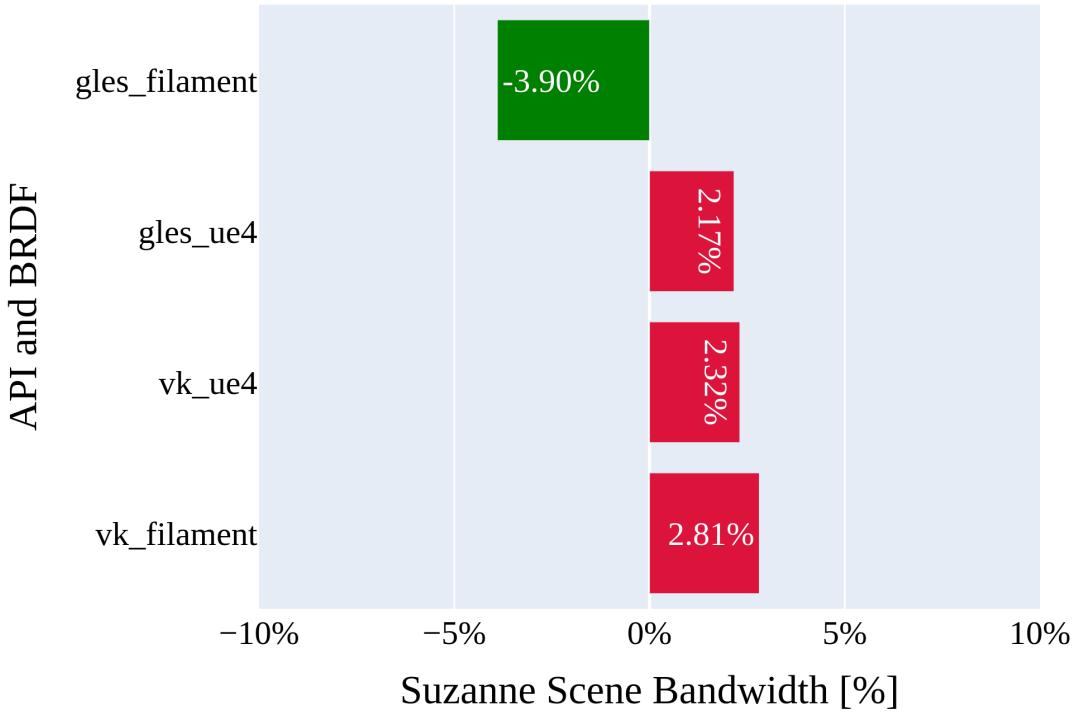


Figure A.9: Results bandwidth.

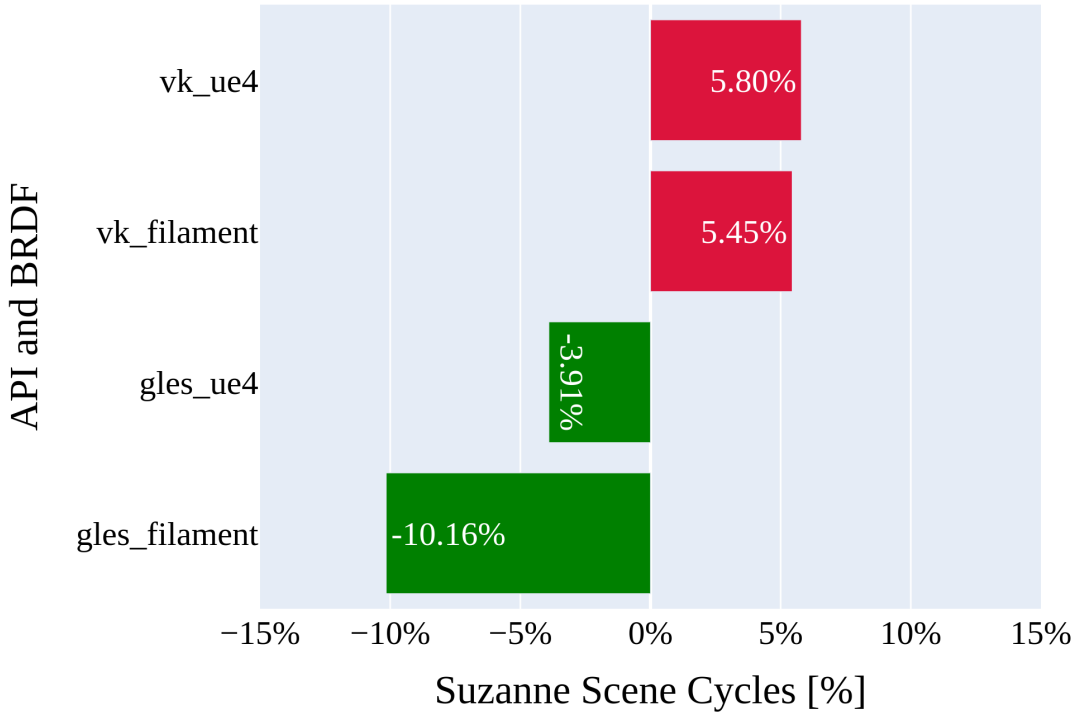


Figure A.10: Results cycles.

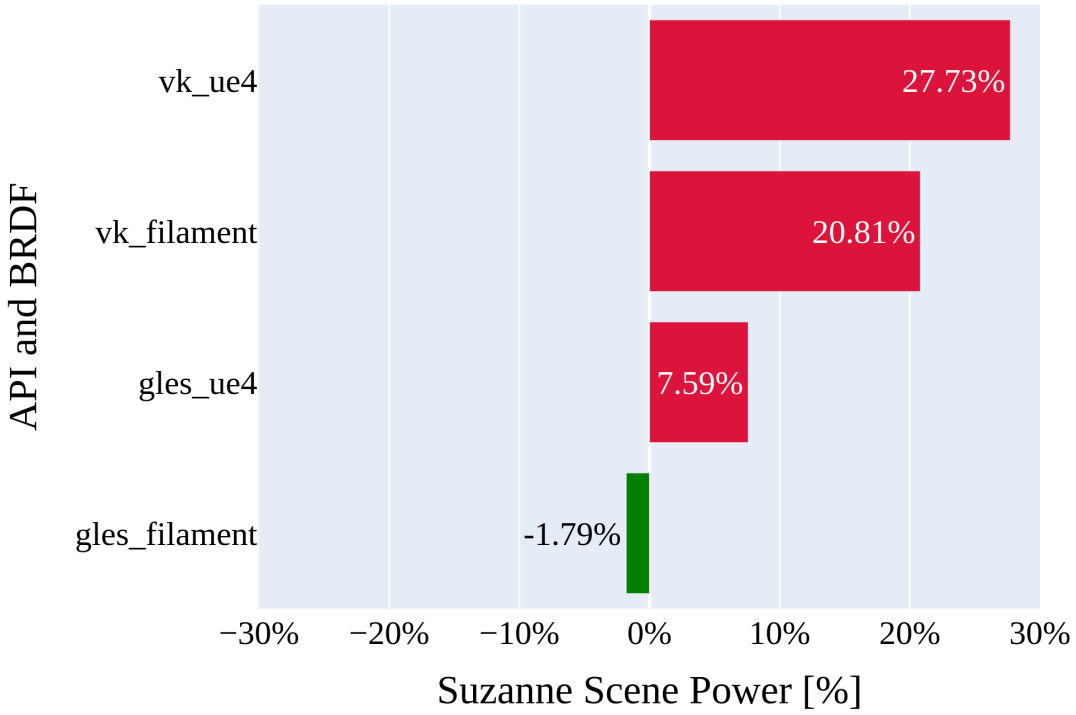


Figure A.11: Results power.

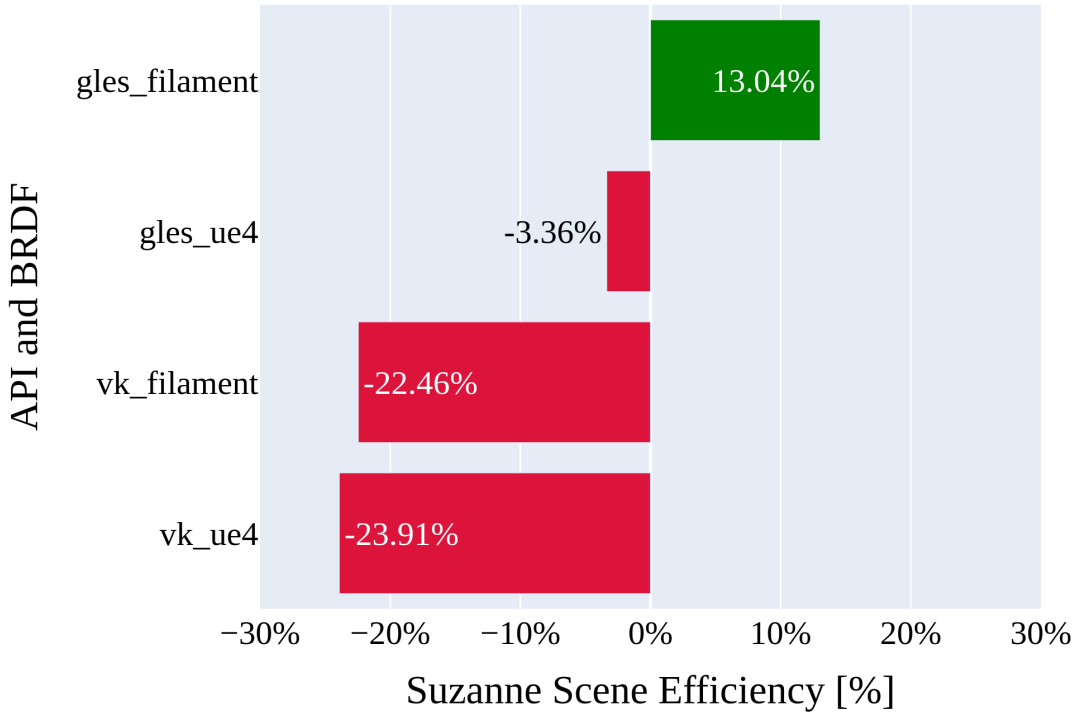


Figure A.12: Results frame efficiency.

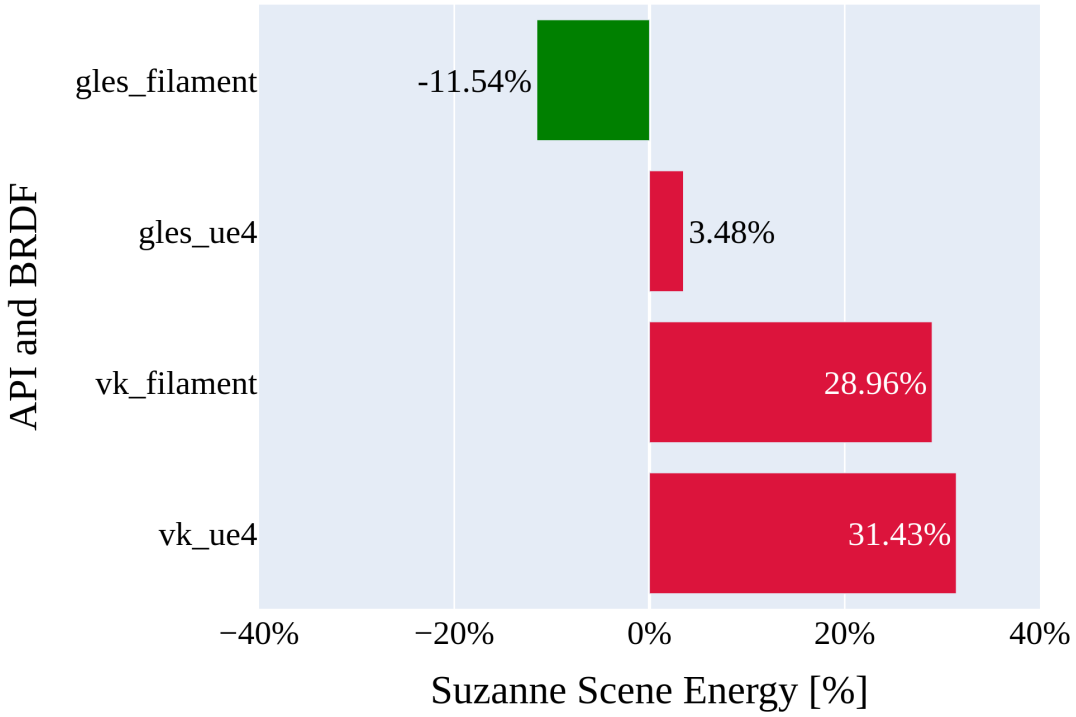


Figure A.13: Results comparison frame energy.

A.4 Workload Power Comparison of 100 Scenes vs 10 Scenes Per Frame

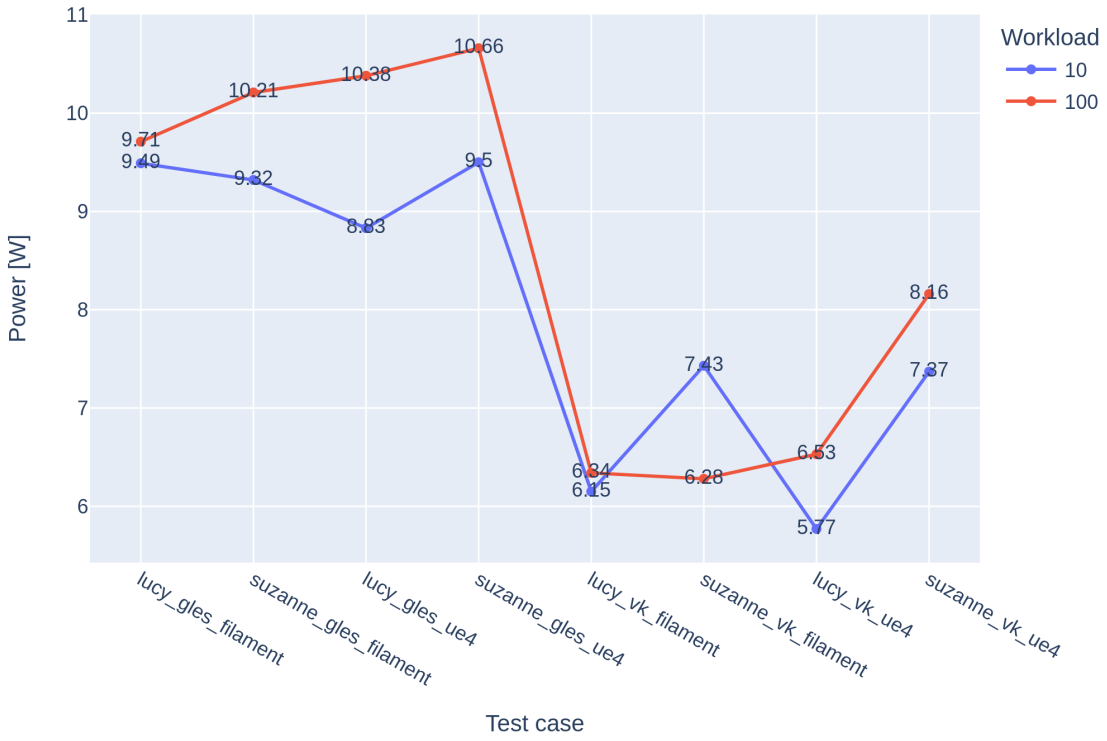


Figure A.14: Power comparison of workload rendering 10 scenes vs 100 scenes.

Appendix B

Appendix 2 Code

This Appendix chapter will have all the source code mentioned in the report

B.1 Appendix Bugfix Google Filament

In order to conduct the experiments, a pull-request was issued to the original Google Filament repository: `https://github.com/google/filament/pull/6634`

The code change was renaming the associated package name from *textured* to *utils*. This change enabled loading textures by using the utility library *utils*. Previously, the texture loading function was only accessibly for one specific sample application. This was a bug and the solution has made it possible to create the test cases used in this thesis.

Affected file:

android/filament-utils-android/src/main/java/com/google/android/filament/utils/TextureLoader.kt

Merged was completed in Mars 13th, 2023 with the commit *Fix wrong package name*.

Commit hash: 277a12dbf0b5ea22dc1e7aa1969999a208f931c1

Listing B.1: Google Filament Bugfix

```
-package com.google.android.filament.textured
+package com.google.android.filament.utils
```

Listing B.1: Google Filament Bugfix

B.2 Filament BRDF Code

Listing B.2: Filament BRDF shader code.

```
//------------------------------------------------------------------------
//  Filament native BRDF
//------------------------------------------------------------------------

float D_GGX(float roughness, float NoH, const vec3 h) {
    // Walter et al. 2007, "Microfacet Models for Refraction through Rough Surfaces"

    // In mediump, there are two problems computing 1.0 - NoH^2
    // 1) 1.0 - NoH^2 suffers floating point cancellation when NoH^2 is close to 1 (highlights)
    // 2) NoH doesn't have enough precision around 1.0
    // Both problem can be fixed by computing 1-NoH^2 in highp and providing NoH in highp as well

    // However, we can do better using Lagrange's identity:
    //      ||a x b||^2 = ||a||^2 ||b||^2 - (a . b)^2
    // since N and H are unit vectors: ||N x H||^2 = 1.0 - NoH^2
    // This computes 1.0 - NoH^2 directly (which is close to zero in the highlights and has
    // enough precision).
    // Overall this yields better performance, keeping all computations in mediump
#if defined(TARGET_MOBILE)
    vec3 NxH = cross(shading_normal, h);
    float oneMinusNoHSquared = dot(NxH, NxH);
#else
    float oneMinusNoHSquared = 1.0 - NoH * NoH;
#endif

    float a = NoH * roughness;
    float k = roughness / (oneMinusNoHSquared + a * a);
    float d = k * k * (1.0 / PI);
    return saturateMediump(d);
}


float V_SmithGGXCorrelated_Fast(float roughness, float NoV, float NoL) {
    // Hammon 2017, "PBR Diffuse Lighting for GGX+Smith Microsurfaces"
    float v = 0.5 / mix(2.0 * NoL * NoV, NoL + NoV, roughness);
    return saturateMediump(v);
}


vec3 F_Schlick(const vec3 f0, float VoH) {
    float f = pow(1.0 - VoH, 5.0);
    return f + f0 * (1.0 - f);
}

float Fd_Lambert() {
    return 1.0 / PI;
}
```

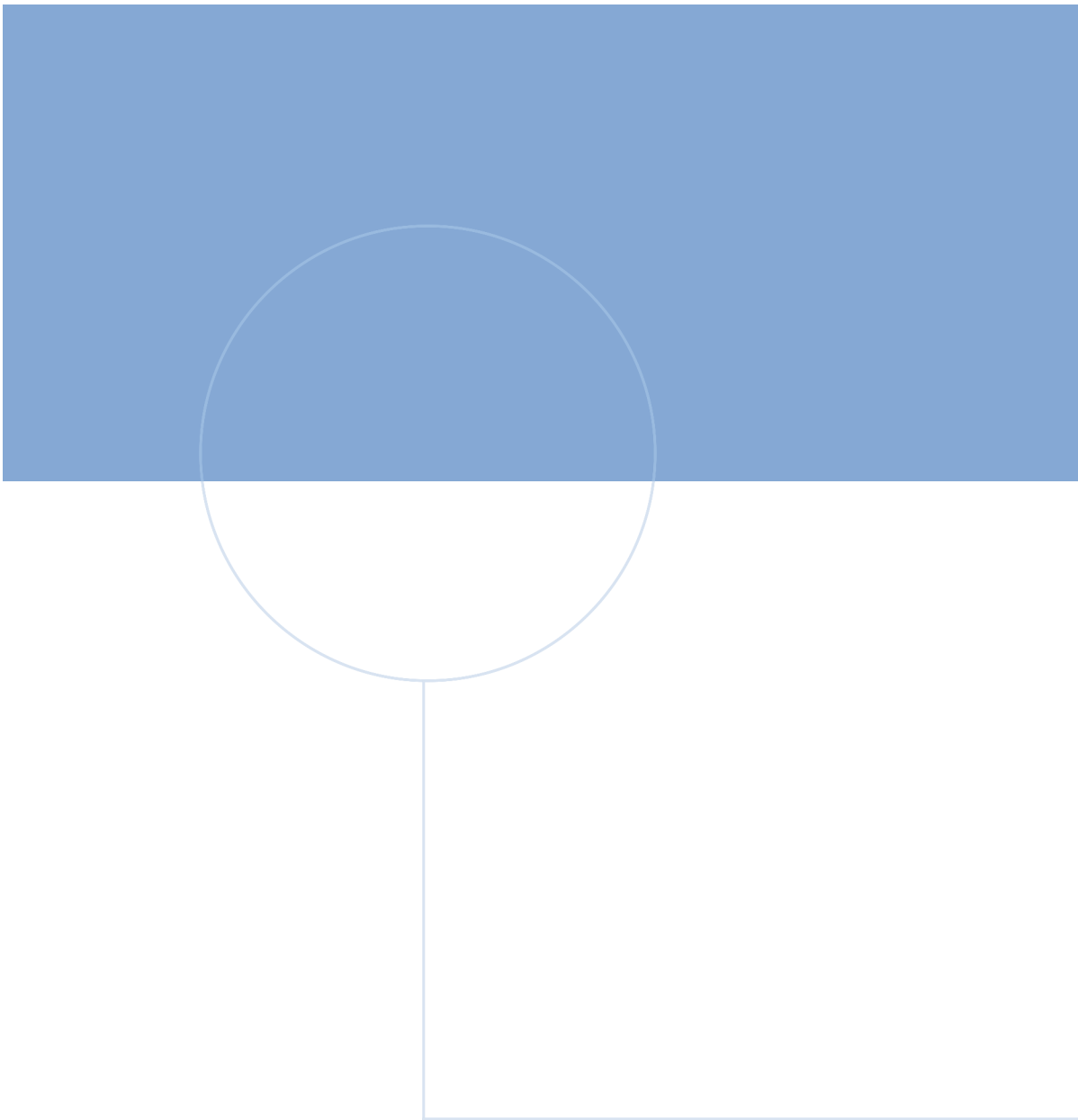
Listing B.2: Filament BRDF shader code.

B.3 UE4 BRDF Implementation Code

Listing B.3: Implementation of Unreal Engine 4 BRDF shader.

```
//-----------------------------------------------------------------------------
//  UE4 BRDF implementations
//-----------------------------------------------------------------------------

float D_GGX_UE4(float roughness, float NoH) {
    float a2 = roughness * roughness * roughness * roughness;
    return a2 / ((PI * (NoH * NoH) * (a2 - 1.0) + 1.0) * (PI * (NoH * NoH) * (a2 -
        1.0) + 1.0));
}


float V_Schlick_ue4(float roughness, float NoV, float NoL) {
    float k = (roughness + 1.0) * (roughness + 1.0) / 8.0;
    float g_v = NoV / (NoV * (1.0 - k) + k);
    float g_l = NoL / (NoL * (1.0 - k) + k);
    return g_v * g_l;
}


vec3 F_Schlick_ue4(const vec3 f0, float LoH) {
    float exponent = (-5.55473 * LoH - 6.98316) * LoH;
    return f0 + (1.0 - f0) * pow(2.0, exponent);
}


float Fd_Lambert() {
    return 1.0 / PI;
}
```

Listing B.3: Implementation of Unreal Engine 4 BRDF shader.



NTNU

Norwegian University of
Science and Technology