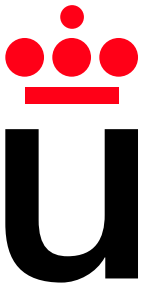


Universidad
Rey Juan Carlos

Escuela de Másteres Oficiales

**Master en Informática Gráfica, Juegos y Realidad Virtual**

**2022-2023**

**Trabajo Fin de Máster**

# DEVELOPING A HIGH PERFORMANCE 3D ENGINE FOR INTERACTIVE APPLICATIONS

**Autor: Antonio Arian Silaghi**
**Tutor: Julio Guillén García**


BY NC SA

# Acknowledgments

To my friends, family, Micky, Cookie and Pepe.

# Resumen

Proponemos un diseño e implementación de un motor 3D en tiempo real para aplicaciones interactivas, como juegos, que integra diferentes técnicas con el objetivo de lograr un alto rendimiento.

Para cumplir con este objetivo, implementamos sistemas que permiten el preprocesamiento de recursos, un modelo de objetos de Entidad-Componente-Sistema (ECS) que aprovecha la arquitectura del hardware de la CPU en cuanto a cachés y posibilidades multihilo. Además, integramos sistemas de renderizado que implementan técnicas de Renderizado Basado en Físicas (PBR) y aprovechan API gráficas modernas de bajo nivel, como Vulkan, y sistemas de alto nivel como FrameGraph.

Presentamos varios casos que validan los sistemas del motor, el rendimiento del modelo de objetos y la calidad gráfica de los sistemas de renderizado.

**Palabras clave:** Motor de Videojuegos, Gestión de recursos, Entity-Component-Systems (ECS), Renderizado Basado en Físicas (PBR), Gráficos por Ordenador en Tiempo Real.

# Abstract

We propose a design and implementation of a real-time 3D engine for interactive applications, such as games, that integrates different techniques with the goal of achieving high performance.

To meet this goal, we implement systems that allow asset pre-processing, an Entity-Component-System object model that leverages the CPU hardware architecture regarding caches and multithreading. Additionally, we integrate rendering systems that implement Physically-based Rendering techniques and leverage modern low-leve graphics API, such as Vulkan, and high-level systems such as FrameGraph.

We present multiple cases that showcase and validate engine systems such as the object model performance and rendering systems graphic quality.

**Key Words:** Game Engine, Resource Pipeline, Entity-Component-Systems (ECS), Physically-Based Rendering (PBR), Real-Time 3D Computer Graphics.

# Contents

# List of Figures

# Chapter 1

# Introduction

## 1.1 Historical Analysis

In the current game development industry, most developed games have a *mostly well defined* separation between the game code and the engine code. That is to say that they either use a general purpose game engine like Unreal, Unity, etc. A in-house custom one like Northlight (Remedy), Decima (Guerrilla Games), Anvil & Dunia (Ubisoft), Frostbite (EA). This a very different landscape when we compare it with how the first digital games were developed.

Games were mostly built from zero each time and were tailored to use the scarce platform resources as well as possible. Given the real-time requirements it was not feasible to build general-purpose reusable systems, as they would not have been sufficiently performant. Only when hardware improved over time developers started to have the necessary leeway to make general-purpose game systems feasible.

Simultaneously, as hardware advancements took place, game graphics underwent a transformative shift from 2D/2.5D to polygon-based 3D. This progression fueled the rapid development of real-time 3D computer graphics. Within this context, it becomes crucial to delve into an exploration of the most noteworthy early game engines and systems that hold relevance to the project at hand.

### 1.1.1 Early Game Engines

The early era of game engines brought forth several notable examples, showcasing significant advancements in the gaming industry. One remarkable instance is the side-scrolling engine developed by Shigeru Miyamoto's team at Nintendo for the Nintendo Entertainment System (NES). Initially employed in Excitebike (1984), this engine later played a pivotal role in the development of Super Mario Bros (1985).

Throughout the 1980s, numerous 2D "game creation systems" emerged, catering to independent developers. Some noteworthy creations include Pinball Construction Set (1983), ASCII's War Game Construction Kit (1983), Thunder Force Construction (1984), Adventure Construction Set (1984), Garry Kitchen's GameMaker (1985), Wargame Construction Set (1986), Shoot-Em-Up Construction Kit (1987), Arcade Game Construction Kit (1988), and early versions of the popular RPG Maker engine series.

While the term "game engine" was not widely used during this time, various systems that align with its description were developed. Examples include Sierra's Adventure Game Interpreter (AGI) and SCI systems, LucasArts' SCUMM system, and Incentive Software's Freescape Engine.

It was not until the mid-1990s that the term "game engine" gained popularity, with the emergence of engines such as Unreal Engine (1998) and the Quake III Arena engine. These engines pioneered the separation of engine systems from game content, leading to significant advancements in game development practices.

### 1.1.2 Reflection Models for Local Illumination

We define local illumination as a method that illuminates a surface only taking into account the incoming light directly emitted from the light sources, ignoring the effect of objects occluding the light sources and the indirect light emitted from other objects. In the following sections we will discuss different real-time methods that describe how light reflects at surfaces.

#### Phong Reflection Model

The Phong Reflection model is a local illumination model that describes how a surface reflects light as a combination of diffuse reflection and specular reflection. It generally also includes an ambient term. This is one of the popular early models used to represent materials in real time.[57]

The model was developed by Bui Tuong Phong at the University of Utah who published it in his 1975 Ph.D. dissertation[41]. Alongside this reflection model, an accompanying interpolation model was presented named **Phong Shading**.[56]

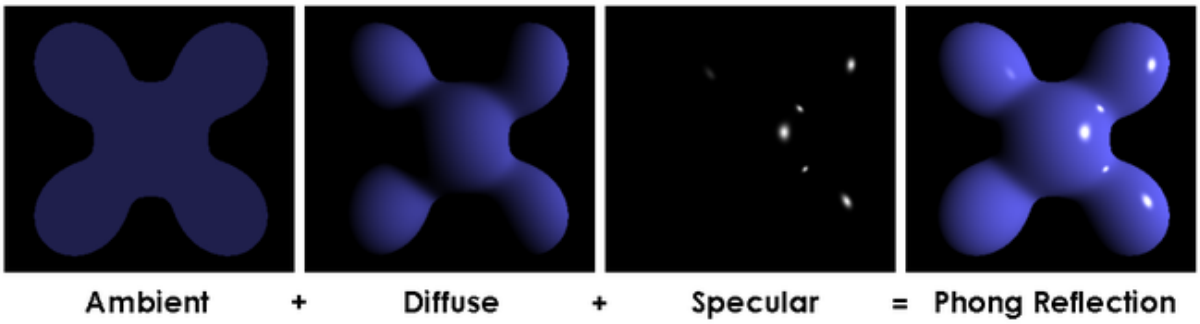



Figure 1.1: Visualization of Phong equation terms[57]

It defines the following per material parameters:

- **Ks:** Ratio of reflection of the specular term of incoming light.
- **Kd:** Ratio of diffuse Lambertian reflectance.
- **Ka:** Ambient reflection constant.
- **Alpha:** Scaling factor of the specular highlight.

This reflection model was used in earlier versions of OpenGL and Direct3D's fixed-function pipeline, although it used the Gouraund shading model instead of the more expensive Phong shading model.[56]

There exists variations of the Phong reflection model like the **Blinn-Phong reflection model** that uses a halfway vector between the viewer and the light source for performance reasons and to solve wrong behaviour with the specular highlight at specific angles.

#### Cook-Torrance Model

The Cook-Torrance reflectance model was presented in 1982 by Robert Cook and Kenneth Torrance[12] as a model that would represent the light reflectance of a surface in a more physically accurate way than the Phong and Blinn-Phong models.[11]

Its a model based on microfacet theory, it approximates a surface as a collection of small reflective mirrors called microfacets. The model uses three main equations that can be defined in multiple ways.

- **Normal Distribution Function:** Defines the percentage of facets that reflect light to the viewer.
- **Geometric Attenuation Function:** Accounts for shadowing and masking effects produced by the microfacets.

- **Fresnel Function:** Handles the Fresnel effect that causes a strong reflection when rays hit with a high angle of incidence.

### Lambertian Diffuse Model

Lambertian reflectance is a property that defines an diffusely reflecting surface. The brightness of a Lambertian surface doesn't depend on the viewer's view direction. The luminous intensity obeys Lambert's cosine law. In other words, this model assumes that the light is reflected equally in all directions when from a surface, it is a simple model that is commonly used as part of more complex models.[54]

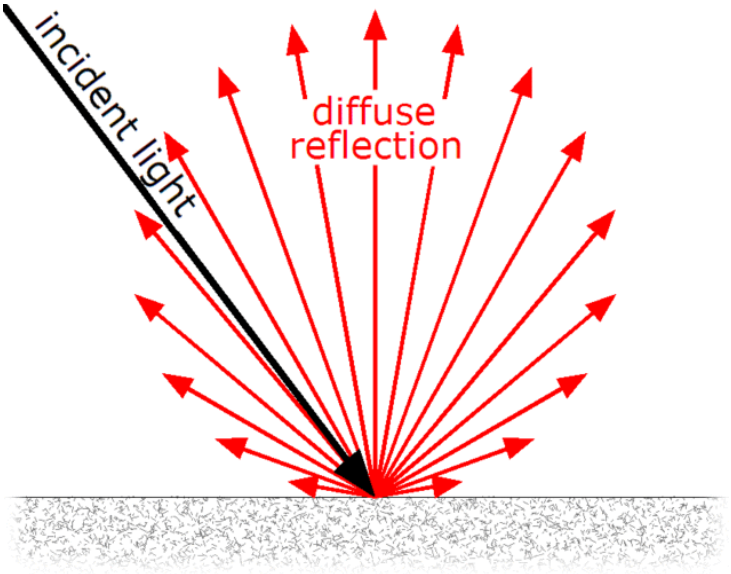



Figure 1.2: Illustration of the Lambertian globe[54]

Regarding real time 3D Computer graphics, the Lambert reflection model is widely used to model the diffuse reflection of a material or surface. It has the big advantage of being very cheap to calculate, requiring only a dot product between the surface normal and the light direction vector.

### Oren–Nayar Reflectance Model

The Oren-Nayar Reflectance Model is used to accurately represent the diffuse reflection of rough surfaces. It was introduced by Michael Oren and Shree K. Nayar in 1994.[39]


Figure 1.3: Comparison of the Lambertian and Oren-Nayar reflectance models[55]

The model is based in microfacet theory, it takes into account complex physical phenomena such as masking and inter-reflections between the surface microfacets. It can be seen as a generalization of Lambert's Law. It is widely used in computer graphics for rendering rough surfaces.

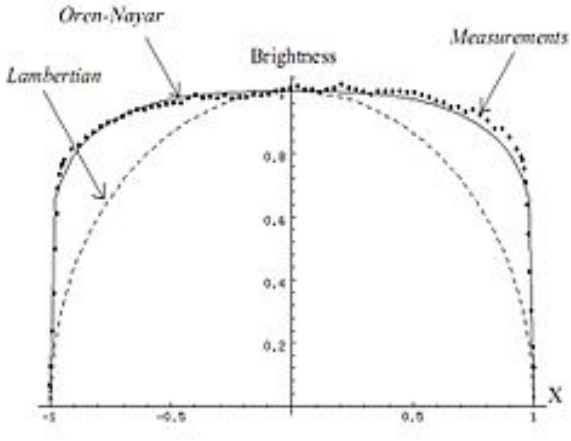



Figure 1.4: Comparison of the brightness value measured across an horizontal slice in the objects in figure 1.3

### 1.1.3 Global Illumination

In this section, we will explore global illumination techniques, which involve calculating the contribution of indirect light from the scene. While local illumination techniques focus on calculating lighting on objects without considering indirect light, global illumination takes into account the overall lighting in the scene. However, due to its computational complexity, accurately calculating global illumination in real-time environments can be challenging. To overcome this, various approximation and pre-calculation methods are commonly employed. In the following subsections, we will delve into some of these techniques in detail.

#### Light-mapping

In many games, the world geometry and lights can be static and dynamic. A common approach to rendering static objects lit by static lights is to pre-calculate most, if not all of the illumination applied on these objects by the static light sources. This is accomplished by calculating the light contribution in a view-independent manner by storing it into extra textures or *lightmaps* associated with the static geometry.

#### Precomputed Radiance Transfer

Precomputed Radiance Transfer or PRT is a technique originally presented in 2002 by Peter-Pike Sloan, Jan Kautz and John Snyderthat.[45] It aims to provide an efficient method illuminate dynamic objects in real time. The general process is to pre-compute the incident irradiance at many points in the scene, with this information, we can choose any dynamic object in the scene and approximate its received irradiance as a interpolation between the neighbouring pre-calculated points[44]. To reduce the memory footprint of the pre-calculated data, **Spherical Harmonics** are commonly used to encode the incoming irradiance at each point.

### 1.1.4 Forward & Deferred Rendering

When rendering and lighting the objects of a scene there are two major methods to do so, forward and deferred rendering. In the following sections we will discuss these two methods.

### Forward Rendering

Forward rendering is the most straightforward method of rendering meshes. Geometry data is supplied to the GPU, it gets processed passing through all of the pipeline stages, finally generating its associated fragments. This sequence is repeated for all of the geometry in the scene, producing the final rendered image.

### Deferred Rendering

At the time when it was first presented. Deferred Rendering was a method that tried to solve the drawbacks of existing forward rendering techniques like poor scaling with light count and overdraw. It does this by separating the rendering process in a **Geometry Pass** and a **Lighting pass**.[13]

In the geometry pass, the necessary geometric data is calculated for each texel in the screen, this is stored in textures called **G-Buffers**. This data is later used in the lighting pass to illuminate all of the texels using their geometric properties. This allowed to render a scene with a lot more lights at the cost of the memory of storing the GBuffers.[53]

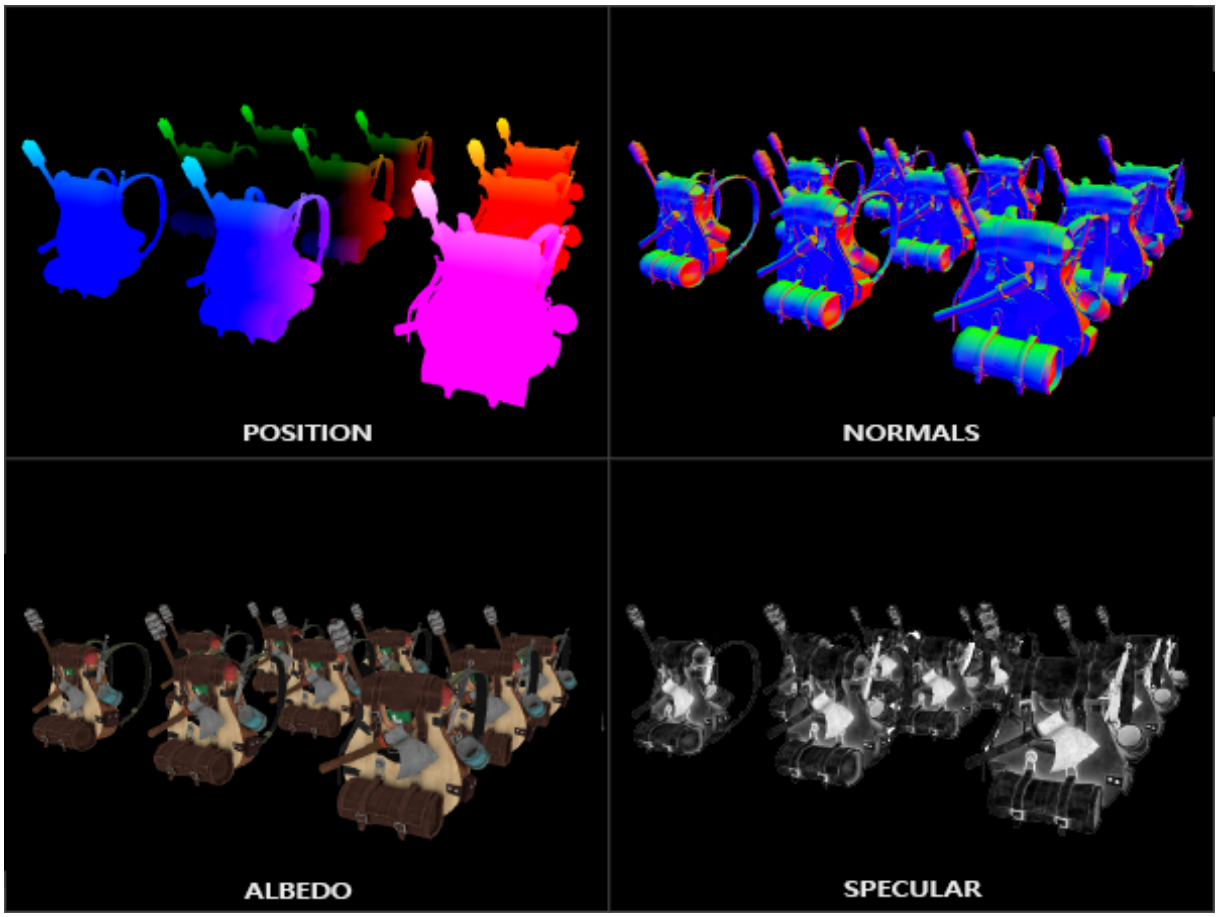



Figure 1.5: Example of G-Buffer textures[33]

This method has many disadvantages like not being able to handle transparency directly, difficulty with using multiple materials, limited MSAA options, etc.

## 1.2 Existing Solutions

The landscape of game engines offers a range of complete general-purpose options, with many commercial and free choices available to developers. Here, we will discuss some of the most popular solutions.

**Unreal Engine** Unreal Engine, developed by Epic Games, is a widely adopted game engine. It provides support for various platforms, including desktop, mobile, console, and virtual reality. Unreal Engine stands out with its advanced commercial systems for developing realistic virtual worlds and character-based games. These systems include dynamic global illumination (*Lumen*), virtualized geometry (*Nanite*), film-quality particles (*Niagara*), destruction (*Chaos*), and more[48].

**Unity Engine** Unity Engine, created by Unity Technologies, is a cross-platform game engine known for its versatility. It supports mobile, desktop, web, console, and virtual reality platforms. Unity has gained popularity in the Android and iOS mobile game market, offering solutions like *Unity Ads* for managing advertisements and monetization in apps[47]. The engine provides tools for developing both 2D and 3D real-time applications, along with features like visual scripting, multiplayer systems, and more[49].

**Godot** Godot is a fully free and open-source game engine initially developed by Juan Linietsky and Ariel Manzur. It offers support for multiple platforms, including desktop, mobile, web, and virtual reality. Godot provides tools for creating 2D and 3D games using its own scripting language, GDScript. However, it also offers bindings for other languages such as Rust, Python, and JavaScript, expanding its flexibility and usability[27]. While Godot may not have cutting-edge systems like Unreal Engine, its advantages lie in being a lightweight and highly extensible engine, complemented by a rich selection of community-created libraries.

Apart from these complete engines, there are numerous libraries available that implement engine-specific systems, enabling developers to accelerate custom engine development. Popular rendering libraries include Google's Filament, BGFX, Ogre3D, Diligent Engine, and Raylib. For skeletal animation, libraries like OOZ Animation are available, and for game UI, options like NoesisGUI are widely used. Additionally, there are several popular libraries that implement the Entity Component System (ECS) object model, such as EnTT (used in Minecraft: Bedrock Edition) and Flecs.

## 1.3 State Of The Art

In this section, we will explore various state of the art techniques that are relevant to the implementation of real-time 3d game engines.

**Radiance Fields** A prominent technique in this domain is the concept of neural radiance fields or NeRFs [36]. These are fully-connected neural networks capable of generating new perspectives of complex 3D scenes based on a limited set of 2D images. By taking a continuous 5D coordinate (comprising spatial location and viewing direction) as input, a neural radiance field produces an output that represents volume density and view-dependent emitted radiance at that specific location. The output is then projected into an image using classic volume rendering techniques.

Notably, projects such as Plenoxels [23] have explored the radiance field concept without relying on neural networks, yielding higher quality results compared to many neural network implementations. Additionally, variants like variable bitrate neural fields [46] demonstrate promising outcomes, showcasing mesh compression and streamable level-of-detail (LOD) capabilities.

**Neural Materials** Significant progress has also been made in rendering complex materials, as evidenced by the "Real-Time Neural Appearance Models" paper by NVIDIA [60]. This paper introduces a comprehensive system for rendering objects with intricate appearances in real-time, an achievement previously limited to offline environments.

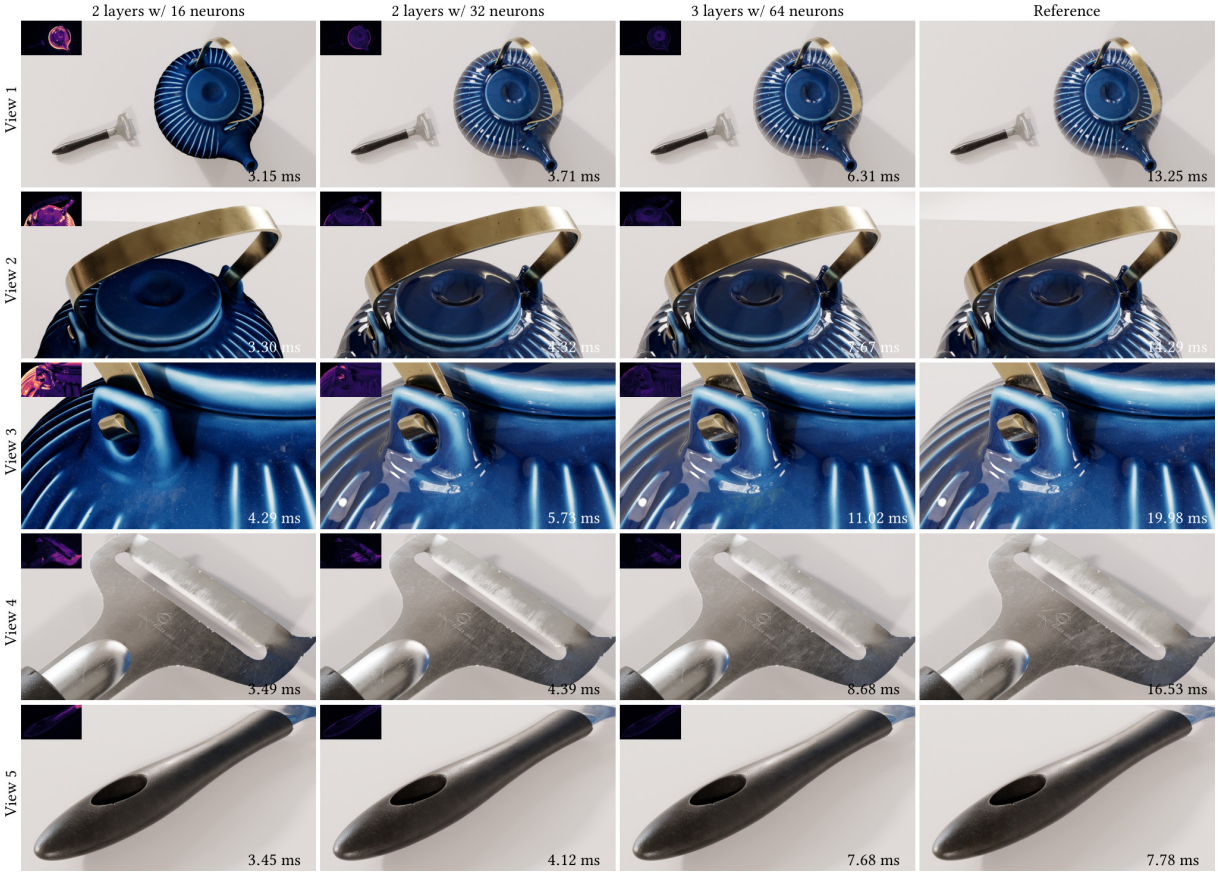



Figure 1.6: Showcase of material appearances with varying neural network architectures

**Real-Time Global Illumination** Real-Time Global Illumination techniques, such as "Global Illumination based on Surfels" (GIBS) [6], provide efficient solutions for calculating indirect diffuse illumination in real-time environments. GIBS combines hardware ray tracing with a discretization of scene geometry, enabling caching and amortization of lighting calculations across time and space. This approach eliminates the need for pre-computation, special meshes, or special UV sets, while delivering high-fidelity lighting that can adapt to scenes of any scale.

Another notable method is ReSTIR [9], which offers an efficient approach to rendering direct lighting from numerous dynamic light sources using Monte Carlo integration. ReSTIR achieves interactive rendering with high-quality results, without the need for complex data structures. It accomplishes this by iteratively resampling a set of candidate light samples and applying spatial and temporal resampling techniques to leverage information from nearby samples.

# Chapter 2

# Objectives

## 2.1 Problem Description

We present a custom real-time 3D game engine, designed with a strong emphasis on performance and simplicity. It leverages the precise control of Vulkan for rendering, providing precise control over the graphics pipeline. It incorporates several essential systems, such as a data-oriented Entity Component System (ECS) object model and an efficient asset processing pipeline, all geared towards achieving optimal runtime performance.

The rendering systems of our engine comprise a set of key components: the *Render Hardware Interface*, the *FrameGraph*, and a *Deferred Rendering Pipeline*. These components support physically based rendering (PBR) materials, along with commonly used post-processing effects. Additionally, our engine features an asset pipeline driven by an offline asset compiler. This compiler converts source assets into a efficient format that can be seamlessly integrated into the engine runtime.

To facilitate the creation of game worlds and entities within them, we have developed a comprehensive suite of generic, data-oriented tools as part of our object model. Furthermore, the engine incorporates core libraries, including serialization, logging, and mathematics, to provide essential functionality.

It is important to note that our engine is not intended to be a one-size-fits-all solution that attempts to fulfill every possible project requirement. Instead, we define it as a collection of libraries that can be replaced or reused to implement different versions of the engine. This deliberate design choice allows us to simplify certain requirements and align the project scope with the available development resources.

## 2.2 Motivation

The game development market offers a multitude of established game engines, such as Unreal or Unity, which come equipped with a wide range of powerful systems ready for developers to utilize. Opting for an existing general-purpose engine can be a cost-effective approach, as it minimizes the need for developing an in-house engine from scratch. However, relying solely on a general-purpose engine can pose challenges if specific functionalities are lacking or if the performance requirements of the project cannot be met.

Alternatively, creating a custom in-house engine tailored precisely to the project's requirements can yield superior results. A custom engine has the advantage of being more streamlined and focused, avoiding unnecessary complexities that might be present in general-purpose engines. The decision between using a general-purpose engine or investing in a custom engine is ultimately a trade-off. Depending on the project's priorities, cutting costs by using a general-purpose engine may be a reasonable choice, while other projects may recognize the value in investing resources to develop a custom engine that perfectly aligns with their specific requirements.

In this project, we have chosen to embark on the path of building a custom game engine that caters

to specific arbitrary requirements. Our aim is to thoroughly analyze the fundamental design principles of game engines and develop systems with a strong emphasis on performance. By pursuing this approach, we seek to create an engine that not only meets the desired functionalities but also operates efficiently within the constraints of the project.

## 2.3 Objectives

For this project, we define the following objectives:

- Analyze existing game engines and design an custom engine based on the analysis.
- Develop a custom game engine based on the previously defined designs.
- Validate the functionality and performance of the engine.

## 2.4 Development Methodology

During the development of this project, we adopted a systematic approach to ensure the successful creation of the game engine. Our methodology consisted of the following steps:

1. Defining Overall Requirements and Engine Architecture: We began by defining the requirements of the engine, encompassing all desired features and functionalities. Simultaneously, we designed the general architecture of the engine, considering its modules and their interactions.
2. Iterative Development: Our development process follows an iterative methodology, allowing us to refine the engine's systems progressively. Each system within the engine underwent the following stages:
   (a) System Requirements Specification: We started by specifying the requirements for each individual system. Outlining the functionalities and objectives that the system needed to fulfil.
   (b) High-Level Design: Based on the specified requirements, we created a high-level design for the system. This served as a blueprint for implementation, considering the overall structure and organization of the system.
   (c) Prototype Implementation: We implemented a prototype based on the high-level design. Developing the necessary components for the system.
   (d) Testing and Analysis: After the prototype implementation, we rigorously tested the system to evaluate its performance and identify any issues or areas for improvement. We analyzed these results to gain insights and further refine the system.
   (e) Iteration and Refinement: Using the test results and analysis as feedback, we refined the systems by iterating on the design and implementation. We repeated the previously defined steps, continuously improving the system until all system requirements were met.
3. Completion of Requirements: Through the iterative development process, we ensure that each system of the game met all of the specified requirements. This comprehensive allowed us to create robust and functional game engine systems.

By following this development methodology, we maintained a structured workflow, enabling us to achieve our objectives effectively while ensuring the quality of the engine.

## 2.5 Project Planification

In order to effectively develop the multiple systems required for this project, we have devised a layered approach, inspired by insights gained from the Game Engine Architecture book[28], that progresses from lower-level to higher-level systems.

The initial focus will be on developing the first and lowest level system in the core layer. Once these system are in an acceptable state, we will proceed to develop the higher level systems in the engine layer that handle features such as rendering, input, assets, object model, and more. Subsequently, we will develop the engine layer that connects all of these systems in a specific configuration.

While our primary focus will be on developing one system at a time, in practice, we will begin by slowly developing all the systems concurrently, aiming to create a vertical slice encompassing the entire collection of systems. This approach may require constant refactoring; however, it will provide valuable insights into the necessary connections between systems and their individual requirements, which may not have been evident if we had developed each system in isolation.

By adopting this layered and concurrent development approach, we aim to foster a comprehensive understanding of the interdependencies and requirements of the various systems. This method will enable us to build a robust and interconnected game engine while continually refining and optimizing its components.

## 2.6 Report Structure

This report is structured in the following 6 main sections:

1. **Introduction:** Presents an historical analysis about game engines and real-time computer graphics. Subsequently realises an analysis of existing solutions and explores state of the art systems regarding game engines and 3D graphics.
2. **Objectives:** Presents the project motivation, objectives, development methodology and planification.
3. **Theoretical Frame:** Presents multiple sources of information that were analysed and studied for the development of this project.
4. **Development:** Presents the project requirements, systems designs and implementation details of the engine.
5. **Validation:** Presents many of the tests realized to validate the functionality and results of the engine, compares them with existing solutions and discusses the observed results.
6. **Conclusion:** Realizes an analysis about the development of the project, accomplished goals, lessons learned and future work.

# Chapter 3

# Theoretical Framework

In this chapter we will talk about the existing information regarding game engines and their subsystems that we analysed and studied as research for the development of the project this project. We discuss the overall architecture of an engine (§3.1), notable core systems (§3.2), resource systems and asset pipelines (§3.3), object models and some of the common design patters used to develop them (§3.4). Lastly, we analyse common rendering techniques and systems used in game engines (§3.5).

## 3.1 Game Engine Architecture

A game engine's general goal is to streamline the process of developing an interactive experience. To accomplish this, an engine provides a wide variety of tools, libraries and functionalities to implement game mechanics, render graphics, manage audio, handle physics simulations, build virtual worlds, and much more.

An engine can be defined as collection of runtime systems and a tool suite. Even though these two components are related we will start by defining the architecture of the runtime first. A game engine is generally built in layers, as seen in figure 3.1, where upper layers depend on lower layers, but not vice versa. This design some nice properties like avoiding circular dependencies.

Starting from the lowest layer, we have the hardware, drivers and operating system. It is common for engines, specially modern ones to abstract platform-specific functionality in a platform independence layer. This layer handles low level functionality like threading, time management, file system, and more. On top of this, we have the *"Core Layer"*, this layer contains general purpose systems that handle profiling, parsing files, logging, math, memory management, and more. Next we have the resources layer that defines and handles all of the resources like textures, materials, models, etc. In practice this functionality is sometimes directly implemented in the own systems that need it. Then we have all of the big engine systems that implement features like rendering, multiplayer, audio, gameplay object models, etc. Finally, on top of these we have the gameplay layer. This layer implements all of the game specific creatures like AI, player state machines, scripted cameras, etc.[28]

## 3.2 Core Systems

### 3.2.1 Logging

The logging system offers functionalities to record and output messages to files, console or send them through the network to a development server. They assist in the developer in recording information about the engine state and help with tracking and debugging engine systems, monitoring systems performance, and more.

Some of the feature that they usually provide are: message filtering by levels of severity, such as

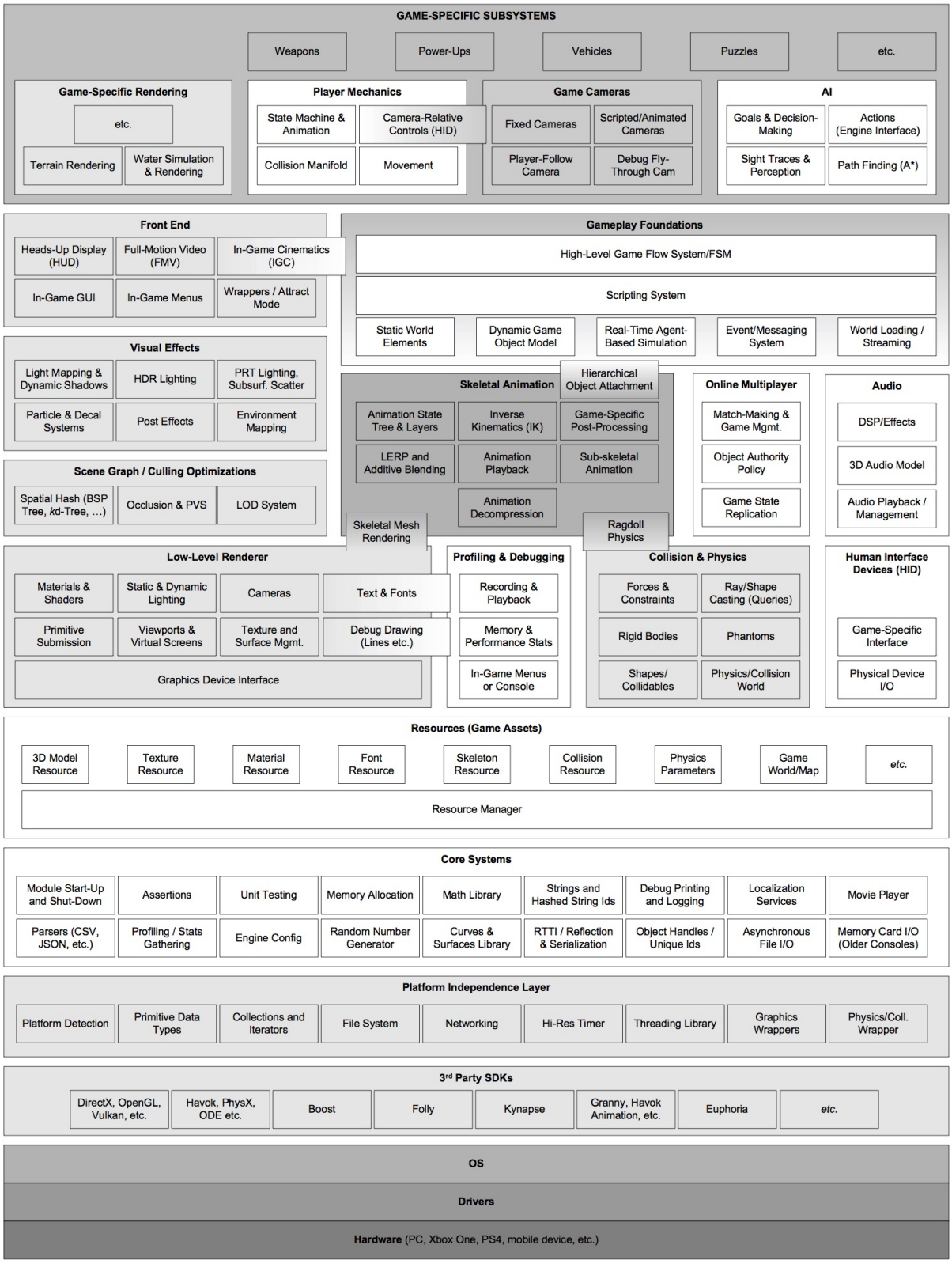



Figure 3.1: Runtime engine architecture [28]

debug, info, warning, error, etc. message tagging with custom labels (such as *"physics", "render", ...*) and, if necessary, networking capabilities that allow the passing of messages between consoles and the development PC's.

### 3.2.2 Memory Systems

A memory system offers low-level functionality over the platform's memory operations. Their goal is to provide efficient and flexible memory management capabilities while minimizing fragmentation, achieving high performance and providing memory debugging and profiling tools.

They provide tools like memory allocators that handle blocks of memory using specific schemes based on the requirements. Some of the common ones are linear, pool, stack and frame allocators. Alongside these, they also provide memory tagging and tracking functionalities that aid the developer in finding memory leaks and keeping the systems within their memory budget.

The implementation of a memory systems widely varies depending on the engine. Some prefer to have fine control by pre-allocating most of the necessary memory for the engine and subdividing it hierarchically for each system. While others don't require the benefits of such fine control and prefer to avoid the complexities that come with it.

### 3.2.3 Mathematics & 3D Linear Algebra

Mathematics libraries play a crucial role in game engines, enabling a wide range of operations necessary for tasks such as representing a 3D world, performing physics calculations, interpolating animations, generating random numbers for gameplay mechanics, implementing ray-tracing algorithms, and more. These libraries provide an interface for essential mathematical tools, including vectors, matrices, quaternions, and various geometric operations.

### 3.2.4 Generic Data Structures & Containers

In game engines, custom implementations of basic data containers are often provided to meet specific memory constraints and performance requirements. These custom-made containers offer specialized functionalities and optimizations tailored to the needs of game development. Here are some examples of these containers:

- Inline Vector/Array: An inline vector or array is typically implemented as a fixed-size stack-allocated array that can dynamically allocate to the heap if more memory is required. They combine the advantages of a stack-allocated array with the flexibility of dynamic resizing when necessary.
- Maps: Maps, also known as dictionaries or associative arrays, associate each element with a unique key, allowing for efficient lookups. High-performance map implementations often leverage CPU cache architecture to minimize cache misses. They store the keys and values in separate but contiguous arrays, ensuring that retrieving an element involves iterating over the key array until the desired key is found. The position of the key in the array corresponds to the position of the associated value in the value array. This design allows the processor to load into cache only the necessary data to find the key, which is then used to access the corresponding value efficiently.
- Stack/Inline Strings: Traditional string implementations use dynamic memory allocation on the heap to accommodate dynamically growing strings. In contrast, stack or inline strings pre-allocate a maximum buffer on the stack where the string is stored. Some implementations allow the buffer to grow dynamically by allocating memory on the heap, similar to an inline array.

To address the performance and memory demands of game development, there are popular libraries like EASTL [15], which provide efficient implementations of many performant data structures. These libraries offer optimized and specialized containers designed specifically for game-related scenarios, ensuring efficient memory usage and high-performance operations.

### 3.2.5 Runtime-Type Information

Runtime-type Information systems, also known as *RTTI*, offer mechanisms that enable the identification and manipulation of object types during runtime. They allow the engine to make decisions and operate on objects based on their type properties. They include features such type identification, reflection, dynamic casting, polymorphism and virtual functions.

Even though some of these features are implemented to some degree *"by default"* in languages like C++ (although with a lack of important features such as reflection), many engines decide to build a custom implementations with the goal of optimizing the performance of type-related operations and reducing the memory overhead of storing all of the necessary type information. Its worth noting that

*"Reflection"* is a fundamental aspect of RTTI systems, it enables the examination and modification of an object's properties at runtime. It allows the engine to inspect and manipulate data members, invoke methods dynamically and perform introspection to query information like function signatures, member names, types, and more. This feature is specially important for developing serialization systems (e.g., Serializing to JSON), scripting systems (e.g., dynamic binding, automatically querying an object's, methods and members) and tools (e.g., automatic creation of editing menus for any object types)

Some common implementation methods of these systems heavily rely in C++ macros, hand placed by the developer, to create and store type information for runtime usage. Other systems leverage the functionalities of libraries such as LLVM and Clang's LibTooling to parse C++ code, extracting type information directly from the generated Abstract Syntax Tree (AST), and storing it in generated headers or lightweight databases [24]. Some implementations develop a custom Domain Specific Language (DSL) where the developer defines C++ classes and structures. This DSL is subsequently parsed, generating the header files of the defined class alongside with its type information [20].

### 3.2.6 Serialization

A serialization system allows data to be converted to a format that can be stored or transmitted and later reconstructed back to its original form. The main purpose of this system is to allow saving and loading data such as configurations, scenes, entities, and more.

A serialization system can be divided into two main components: serialization and deserialization. Serialization refers to the process of converting data structures into a format suitable for storage or transmission (such as binary, JSON or MessagePack). Deserialization, on the other hand, handles the reconstruction of the data structure taking as input the serialized representation, restoring the data.

These systems generally provide features like versioning to facilitate backwards compatibility when loading serialized data from a different version. They also allow de developer to specify custom serialization logic for certain data types and define what data should be serialized or excluded.

Many serialization libraries like *Cereal* are implemented using C++ macros that generate the necessary functions to serialize an arbitrary type. Another approach is to use an RTTI system, this allows the serialization system to query the format and structure of the data type through reflection. Using this information to automatically generate the necessary logic to serialize and deserialize the data type to the supported formats.

## 3.3 Resource Systems & Asset Pipelines

Games are built from a wide variety of assets and resources. These include meshes, textures, shaders, materials, animations, audio clips, prefabs, gameplay zones, and more. Efficiently managing these resources is crucial at every stage of the development process, starting from their creation in Digital Content Creation (DCC) tools to their utilization in by the engine runtime.

Through the development of a game, a systematic approach is required to handle all of these assets effectively. This involves organizing, storing and manipulating the resources in a way that ensures seamless

integration into the game engine. Furthermore, in the engine runtime, proper management of these resources is essential to meet the performance requirements of a real-time application.

An asset pipeline plays a big part in defining the workflow of game developers, it encompasses many interconnected processes and tools that handle the creation and integration of assets through the development cycle. A robust asset pipeline allows seamless collaboration between multiple users in the development team. It optimizes the iteration cycles, allowing developers to quickly iterate on assets, make changes, and test them in real-time. It also enables a smooth transfer of assets between all of the team members.

An asset can be defined as a combination of various pieces of data. These include the source data (such as a texture or a 3D model), references to other assets (like a material referencing textures), and metadata that specifies how the asset should be handled by the runtime (e.g., Texture format when uploading it to the GPU).

When developing a game, a flexible asset system is required, capable of handling hundreds of gigabytes of assets, accommodating constant changes by multiple contributors, supporting hot-reloading, and ensuring fast asset iteration times. On the other hand, when releasing a game, a performant system is required, one that can package the assets efficiently and load them as fast as possible to be used in the runtime. Because of these vastly different and contrasting requirements, many game engines opt to implement separate back-ends to each situation accordingly.

During development, the asset system that is typically designed as a server that handles request by the development PC's, enabling the retrieval, modification and conversion/processing of assets. On the other hand, for the release of the game, the asset system design focuses on packaging assets as efficiently as possible, aiming to strike a balance between file size and loading time. While this packaging process accelerates asset operations during runtime, it slows down iteration times, making it less suitable for the development stage.

In both scenarios, the resource system of a game engine can be divided into two main components: the conversion module and the loading module. The conversion module takes as input the source assets and their accompanying metadata, converting them into a format that the engine runtime can subsequently load through the loading module. In this conversion, the module ensures that the assets are properly prepared and optimized for runtime operations.

## 3.4 Object Models

The object model can be defined as the underlying system used to represent the game world and all its residing objects. This system provides the basic building blocks to make the virtual world and manages the connections between various engine systems such as physics, rendering, particle systems, and more. The majority of gameplay mechanics and game-specific functionalities are built on top of this system. In this section we will discuss common design patterns employed in these types of systems.

### 3.4.1 Class-Based Model

The class-based model leverages the features of object-oriented programming languages, such as classes, objects, inheritance and polymorphism, to construct the game world. In this model, developers define explicit classes for each object type, using inheritance to combine object behaviours. Although is relatively simple to implement for smaller-scale games, most of the code tends to be highly game-specific and difficult to reuse. This approach often results in deep class hierarchies and can lead to issues like diamond inheritance. Additionally, this model requires the creation of special-purpose engine tools tailored to the specific game being developed. As a consequence of these reasons, this model has become less prevalent in modern game engines.

### 3.4.2 Object-Component Model

The object-component model is currently one of the most commonly used objects models. Adopted by engines like UE5, Unity, CryEngine, and others. This model defines a general-purpose *object* that is just a collection of *components*, responsible for storing data and behaviour. Components can hold references to other objects and components to facilitate communication between them.

This model has the advantages of being very simple and easy to understand. However, some of the biggest problems with it come with the way that components communicate with each other. This functionality can be easily misused and its one of the reasons its hard to multithread general gameplay code. Additionally, the model struggles with handling initialization and execution order dependencies. Although solutions like providing multiple initialization callbacks (e.g., Unity's awake, start) and execution order priority can address these issues to some extent, they often lead to increased complexity, poor scaling and a higher likelihood of bugs.

The aforementioned *"reference problems"* can be mitigated by maintaining a strict control over references and their usage. Initialization and execution order dependencies can be handled by introducing additional components that manually initialize components. Nevertheless, these solutions often serve as workarounds rather than addressing the limitations of the object model's functionality in effectively managing these challenges.

### 3.4.3 ECS Model

This model takes inspiration in data-oriented programming and existing database technology with the goal of offering a performant way to represent the game world. It defines the concept of an *entity* that is generally just implemented as an identifier or a key (akin to a database primary key). To assign data to entities we create *components* and assign them to these entity identifiers, these components are just containers of plain data. Lastly, to add behaviour we add *systems* that can query the components of entities that match some specified requirements and operate on their data. The systems in essence just operate on the component data. A key aspect of the model is that it optimizes sequential access to multiple entity components by storing them in a cache-friendly way in memory based on their access patterns.

This model is generally implemented in such a way that it can collect information about the data accesses from the entity systems code before their execution, either by querying it automatically from the execution code or by requiring the developer to specify the accesses before-hand. This is valuable information that can be used to automatically parallelize entity systems.

Some of the advantages of this system is the performance it allows with its cache-aware design and built-in multithreading functionality. That said, it has a steeper learning curve because it diverges from the object-oriented design classically taught in academia. The analysed implementations have problems like not providing basic features like transform hierarchies of objects.

### 3.4.4 An Experimental Multithreaded Component Model

This is an experimental model presented by Bobby Anguelov in his own engine Exoterica[5]. Its a model that tries to solve the order dependencies and multithreading problems of the previously explained object-component model. It mixes ideas of the object-component and ECS models as we will discuss in the next paragraphs.

In this model we have the following key elements:

- Entity: An entity is an object that contains a collection of components associated with that entity, as defined in the object-component model.
- Component: A component is a container of data that contains callbacks that can be used to define behaviours that modify its own data, in these callbacks, the developer only has access to the own component data.

- Systems: A system can contain data and behaviour, it can access entity components and its main purpose is to serve as a way of communication between them. It closely resembles an ECS entity system.

The key features of this system are that, because components are defined in such a way that they only mutate their internal data, all of the component behaviours can be executed in parallel in as many cores as available. After the components update, the systems get executed as they would in an ECS model, handling all of the inter-component communication. This sequence of updating components and then systems can be executed multiple times in a frame allowing *component, system, component* communication without having to wait for the next frame to call the component update.

This system has the advantage of being trivially parallelizable at the component level and easily parallelizable at the system level using techniques discussed in the ECS model. It also defines a clear workflow to the developer where the internal behaviour of components is separated from the communication between them. Allowing components to truly be independent units of data and behaviour while the systems act as glue between them.

## 3.5 Rendering Systems

Rendering the game world within soft real-time constraints is a fundamental functionality of a 3D game engine. Depending on the desired style and quality of the final render, the rendering pipeline can become quite complex, involving graphics API abstractions, material systems, particle systems, frame graphs, and specialized debugging tools, among others. In the following sections, we will explore multiple systems related to the rendering process.

### 3.5.1 Physically-Based Rendering

Physically-Based Rendering (PBR) encompasses a collection of techniques aimed at emulating the lighting and material properties of the real world. It seeks to replicate the behaviour of light in a physically accurate manner, ensuring that materials appear correct under various lighting conditions. This stands in contrast to rendering pipelines that do not adhere to PBR principles[34].

A key component of the material model is the Bidirectional Scattering Distribution Function (BSDF), which mathematically describes the material's behaviour. The BSDF consists of two parts: the Bidirectional Reflectance Distribution Function (BRDF) and the Bidirectional Transmittance Distribution Function (BTDF) [19] [7].

In many cases, the BRDF alone is sufficient for modelling most surfaces encountered in rendering. This simplification significantly reduces the complexity of the model but imposes certain limitations, such as accurately representing only reflective, isotropic, dielectric, or conductive surfaces with a short mean free path[19].

A common approach is to use a microfacet BRDF, which enables the representation of light interactions with irregular surfaces. The microfacet model assumes that all surfaces can be described as an assemblage of tiny reflective mirrors known as microfacets. The alignment of these microfacets varies based on the surface's roughness, allowing for the simulation of a wide range of surface characteristics.

### 3.5.2 Graphics API Abstractions

Certain game engines need to support multiple graphics APIs, such as Vulkan, Direct3D 12, Metal, OpenGL, and more, depending on the target platform. To accomplish this, engines create an abstraction layer that sits atop these platform-specific graphics APIs. This layer allows developers to write rendering code that is agnostic to the underlying API, while still providing access to platform-specific features when necessary.

Designing and implementing these abstraction systems pose a challenge in striking a balance between abstraction and control over the underlying APIs, ensuring that performance is not significantly compromised. The complexity of this task varies depending on the APIs that need to be supported. Some

modern graphics APIs, like Vulkan, DirectX 12, and Metal, follow similar principles, making abstraction relatively straightforward. However, older APIs like OpenGL operate in a fundamentally different manner, requiring more intricate handling. We have analyzed multiple implementations and designs of graphics API abstractions, including:

**Unreal Engine 5** Unreal Engine is one of the most popular game engines in the market. It offers a high fidelity renderer with many state-of-the-art technologies like **Lumen**[1] that allows real time global illumination and **Nanite**[2] that efficiently manages rendering meshes with high triangle counts. A complete analysis of just the rendering systems of this engine is out of the scope of this project, but just looking at the high level architecture and system design could provide us with important insights.

Unreal implements the previously defined RHI that abstracts platform specific API's. It provides a high level shader definition system that allows the user to define and sync shader information like binding layouts between the proper HLSL/GLSL shader code and the C++ code. On top of all of this it defines a FrameGraph-like system called Render Dependency Graph **(RDG)**[16]. This system implements all of the core features described in the FrameGraphs section. Unreal's implementation of this system offers the lambda-based API presented by **FrameGraph, O'Donnell**[37]

**Diligent Engine** Diligent Engine is a lightweight cross-platform 3D graphics library and rendering framework, its built with the goal of taking advantage of DirectX 12, Vulkan and Metal, while still supporting older API's like OpenGL and DirectX 11. It uses HLSL as a unified shading language that gets converted to the required format depending of the underlying platform that is in use.

**Wicked Engine** Wicked Engine is an open-source C++ engine that focuses on modern rendering techniques and high-performance. Its supports Windows and Linux, offers DirectX12 and Vulkan backends for rendering, and implements its own editor with LUA scripting facilities.[26]

This engine has been selected for analysis because it implements a high performance modern API abstraction that only supports modern API's like DirectX12 and Vulkan. Its designed so that is as close as possible with the underlying API but offers a lot of quality of life improvements and abstractions. It abstracts some of the most used operations like managing buffers, textures and creating, populating and submitting command lists in a multithreaded context.[17]

**BGFX** BGFX is a popular open source cross-platform graphics API abstraction that supports a multitude of render backends like DirectX 9/11/12, Metal, OpenGL, Vulkan, WebGL. It supports platforms like Android, Linux, MacOS, PS4, Windows, UWP, etc.[25]

Studying the provided API, we noticed that its design is close to the OpenGL API. This has the advantage that the API is simple and straightforward to use but it has the big disadvantage that, for applications with high performance and graphic fidelity requirements. A lot of the fine tuning is lost. Even though we can use a Vulkan or DirectX 12 backend, we can't exploit their fine tuning capabilities through the simple BGFX API. Of course in the end this is just a design choice. This library was designed with portability and ease of use as the primary goals.[8]

**Vulkan & DirectX 12** Even though analysing existing implementations of these kinds of systems can give us insights into important design concepts. We must not forget the actual problem that we are trying to solve in our case. Which is to create an render API abstraction that must work with Vulkan and should be easy to integrate to DirectX 12. This means that analysing the design of these APIs could provide us with important information that will aid us in creating an abstraction over both of them.

Vulkan is a modern explicit API that targets high performance real time 3D graphics applications. It offers fine tuning controls to allow the developer to optimize its application as much as possible. Next we will discuss the general architecture of the API, focusing on the most important parts.

The first thing we must setup and configure is the **Instance**. This is a object that serves as the initial connection between the application and the Vulkan Library[51]. This instance will handle high

level application info, additional layers configurations (like debugging layers), and most importantly it will allow us to select a **Physical Device** and create a **Logical Device** from it.

The **Logical Device** will allow us to manage resources like **Buffers** and **Textures**, **Pipelines** that will use user defined **Shaders**, **Descriptor Sets** that will allow us to bind resources to shaders, etc. It will also give us access to **Queues** where we can submit **CommandLists** specifying the rendering operations we want to execute.

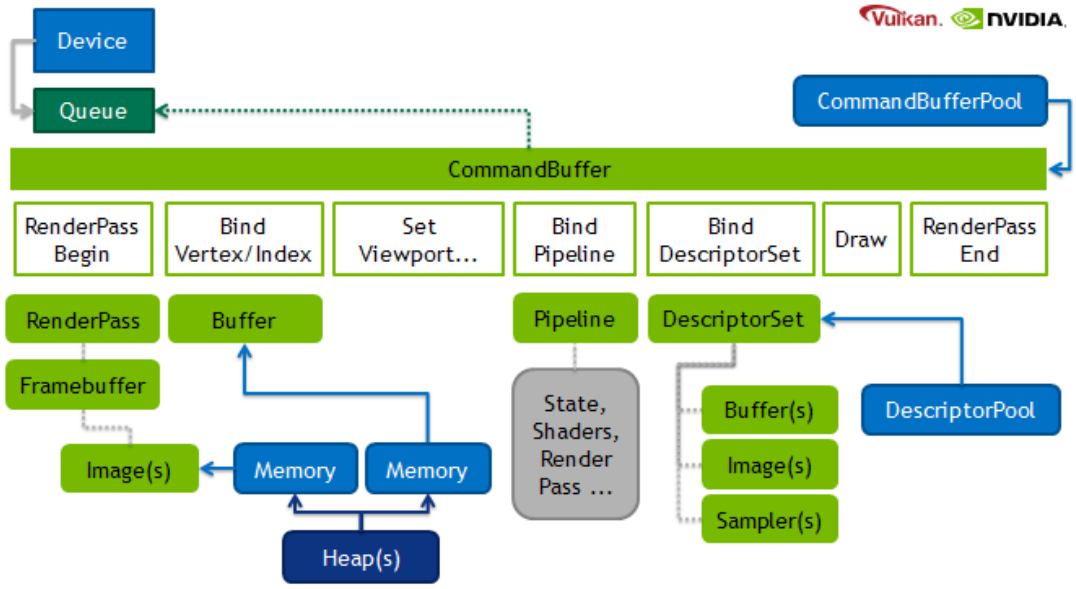



Figure 3.2: Vulkan Commands API Architecture[14]

The Vulkan API also provides important synchronization mechanism like **Pipeline Barriers**, **Fences** and **Semaphores**. Alongside this it also requires the developer to explicitly manage the memory of different Vulkan resources through **CommandBufferPools**, **DescriptorPools** and **General GPU Heap Memory**.

### 3.5.3 FrameGraphs

When designing the high-level rendering pipeline for an engine, one of the primary objectives is to create a system that is both modular and user-friendly, while ensuring optimal performance and reusability across games with varying requirements. Achieving this goal in modern graphics APIs like DirectX 12 or Vulkan is a non-trivial task. To attain high performance in these APIs, developers must explicitly and manually declare rendering operations such as pipeline barriers for resource transitions, memory aliasing, render target clears or discards, multi-queue synchronization, and more. This manual approach can result in monolithic and difficult-to-reuse rendering code, as each piece must be individually designed and fine-tuned for a specific rendering pipeline.

The standard solution to this problem, employed by most engines, is a concept called **FrameGraph**[37]. FrameGraph systems separate rendering code into modular units known as *Render Passes* and simplify many low-level operations that can be automated.

This is accomplished by requiring Render Passes to declare their resource accesses beforehand. This information is then used calculate resource dependencies and automatically generate optimal resource transitions, memory barriers, calculate resource lifetimes, manage memory aliasing, and more. Finally, all of the render passes are executed in an order based on their dependencies, at this stage, the passes record and submit the actual render commands to the graphics API.

At a high level, the FrameGraph can be broken down into three phases:

1. **Setup:** Declare all the passes and resource dependencies.
2. **Compilation:** Calculate the required resources, lifetimes, transitions and synchronization points.
3. **Execution:** Execute all the graph passes.

Individually, each render pass consists of two phases:

1. **Dependencies/Setup:** Declare resource accesses in the pass (Read&Writes) and transient resource creation.
2. **Execute:** Rendering commands that will be recorded and submitted for execution.

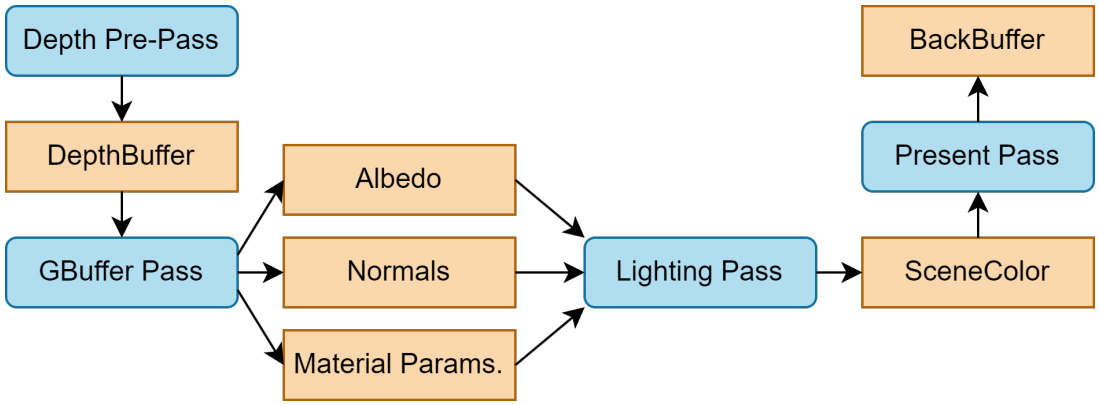



Figure 3.3: Example of a basic deferred rendering graph, expressed as an acyclic directed graph

By predeclaring all resource accesses, the FrameGraph provides comprehensive information about the frame. It enables us to know which resources will be accessed, how they will be used, and when they will be accessed. With this information, many operations like resource barriers and layout transitions can be automatically handled. Additionally, since we have full frame information, we can efficiently batch operations such as barriers.

**Transient Resources** Many render passes require resources that have a limited lifetime within a frame, the most common ones being textures being used as render targets. There are cases when these resources could be reused by different passes that execute at distinct times in the frame without overlapping. However, manually optimizing this scenario is non-trivial and prone to introducing bugs if the rendering pipeline configuration changes. Fortunately, using a FrameGraph architecture, we possess complete frame information, including details about transient resource creation and usage. This enables us to accurately calculate the lifetimes of all resources and automatically manage their memory aliasing or reuse.

### 3.5.4 Forward & Deferred Rendering and Variations

There are many high level real time rendering techniques but they can be mostly categorized into two types, forward and deferred rendering. [31]

The traditional **Forward rendering** is the simplest and most straightforward method. Each mesh in the scene is individually rendered and its lighting is calculated by checking the contribution of all of the lights. A depth buffer is used to resolve object overlapping.

This has the advantage of being very simple, allows multi-sampling, multiple types of shaders, transparency, etc. Some of the big disadvantages are that for each fragment we calculate, we must check the light contribution of all of the lights, this means that it scales poorly with the number of lights in the scene. There is also a lot of overdraw produced when rendering.

**Traditional Deferred rendering** tries to get a better scaling with the number of lights in the scene and reduce overdraw. It accomplishes this by decoupling the geometry and the shading process. Initially,

per pixel geometry information like albedo, normals, roughness is calculated in a **Geometry Pass**. This information is later used in a **Lighting Pass** that evaluates the materials and lighting accordingly.

Modern versions of these techniques include *Tiled Forward/Deferred* (Tiled forward is also commonly known as Forward+ [29]) and *Clustered Forward/Deferred* rendering [38]. Tiled rendering is based on the principle of dividing the screen into 2D tiles and performing lighting calculations only on the relevant objects within each tile. This approach can greatly reduce the number of lighting calculations which is especially important in forward rendering. Clustered rendering is based on the same idea but it divides the camera frustum intro 3D clusters which can further reduce the number of lighting calculations.

### 3.5.5 Post-Processing

The following section describes all of the post-processing effects that have been analysed for this project. Most of the presented effects try to cheaply emulate otherwise costly light phenomena.

#### Bloom

Bloom is an effect that tries to reproduce the glow of bright objects or light sources in an image when captured with real-world cameras. Even though it is possible to implement this effect using low precision colour buffers when rendering the scene, it works best with HDR images.

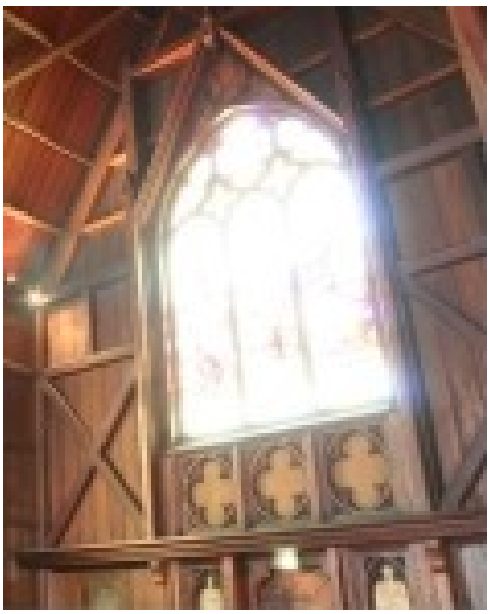


Figure 3.4: Bloom effect seen in a picture taken with a real-world camera[52]

In some of the original implementations the general process is as follows:

1. Render the scene as usual in an HDR texture.
2. Select the sections in the image that are deemed bright enough to produce light bleeding or bloom.
3. Blur the selected sections with a convolution filter.
4. Compose the result with the original scene texture.

This process has the drawback of requiring the use of big and expensive convolution filters to be able to render the light bleeding at a distance. Because of this, modern models try to avoid this by using better filtering schemes.[32]

#### Screen-Space Ambient Occlusion

Ambient occlusion can be defined as a technique that tries to calculate how much ambient light reaches any point in the scene. Algorithms like the ones presented by **Crytek** try to calculate this phenomena using only a post-processing step, this is the screen-space ambient occlusion algorithm (**SSAO**). Some modern solutions try to calculate this efficiently using hardware ray-tracing facilities.

For this project. We studied a variation of the **Crytek** algorithm. Which conceptually works the same way but uses a surface normal oriented semi-sphere instead of a complete sphere.

#### Screen-Space Reflections & Refractions

Also known as **SSR**, these effects try to emulate material reflections and refractions. At a high level they accomplish this by ray-casting in clip-space.

#### Screen-Space Global Illumination

This effect follows the same ray-casting principles as **SSR** but tries to approximate global illumination instead. It is mostly used as a secondary source of global illumination alongside more precise techniques such as pre-computed light maps and lighting caches.

### 3.5.6 Lighting Caches and Spherical Harmonics

Spherical harmonics are special functions defined over the surface of a sphere, they form a complete set of orthogonal functions an thus an orthonormal basis, Which means that we can define any function on a sphere as the sum of these spherical harmonics[58]. Some common usages include signal encoding, decoding and convolution/product.

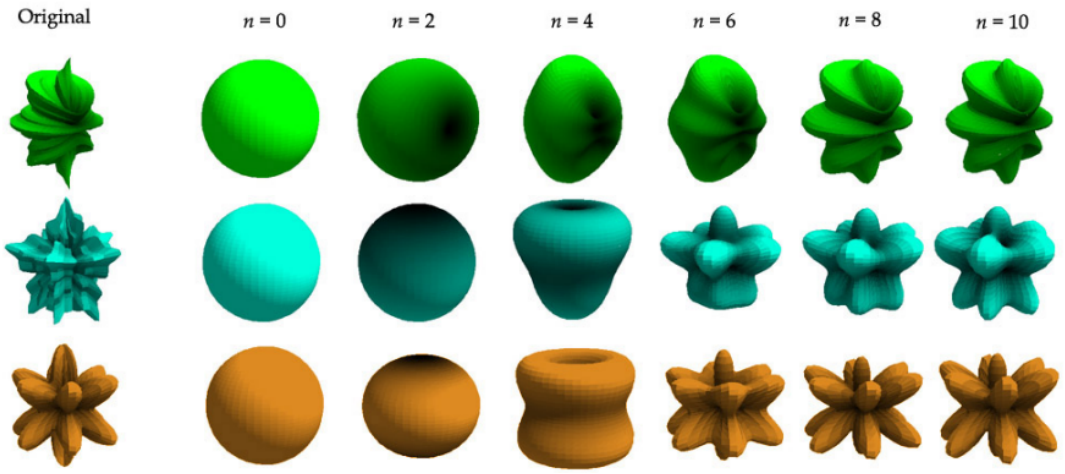



Figure 3.5: Example of function reconstruction with SH[40]

Relating to the field of rendering, Ramamoorthi's paper [42] establishes a relationship between radiance and irradiance in terms of Spherical Harmonics, these results allow us to estimate the diffuse irradiance of a point in space as ah harmonic function.[40]

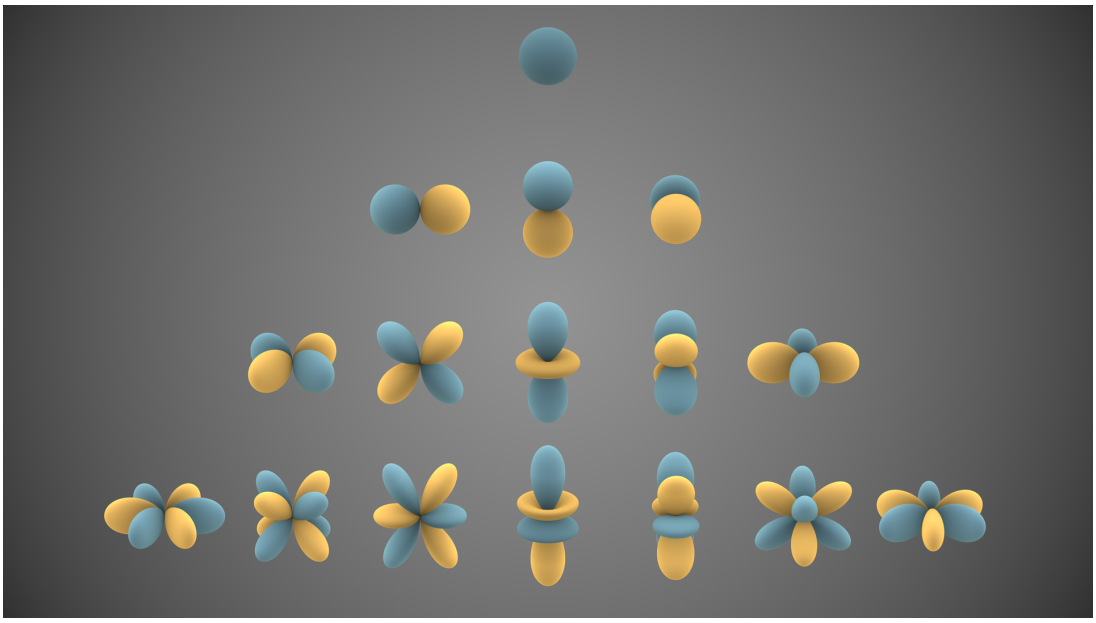


Figure 3.6: Visual representation of the first few real spherical harmonics[58]

# Chapter 4

# Development

In this chapter we will discuss the requirements of the project (§4.1), the overall design of the engine (§4.2) and its most notable systems like asset management (§4.2.5), rendering (§4.2.6) and the object model (§4.2.8). Alongside this we also talk about some of the design philosophies that were followed in these designs (§4.2.1). Lastly, we will cover some of the implementation choices we made (§4.3).

## 4.1 Project Requirements

In this section we will define the functional **(FR)** and non-functional **(NFR)** requirements of the engine and some of its most notable systems like asset management, rendering and the object model.

#### General Engine Requirements

**FR1:** Systems that manage serialization and de-serialization of data to/from disk.

**FR2:** Centralized logging system for comprehensive log management.

**FR3:** Memory management systems with tracking and debugging capabilities.

**FR4:** Performance profiling at the frame level, allowing measurement and analysis of frame execution times.

**FR5:** Math tools and utilities for handling vectors, matrices, and mathematical operations.

**FR6:** High-precision time management systems, providing accurate time measurement and synchronization.

**FR7:** Random number generation, supporting simple number generation, seeded generation, and generation using uniform distributions.

**FR8:** Build system must be able to individually compile engine modules as libraries.

**FR9:** Keyboard and mouse input management.

**FR10:** Task-based threading system.

**NFR1:** Serialize and deserialize arrays of data in a performant manner.

#### Asset Pipeline Requirements

**FR11:** Individually convert/compile engine assets offline

**FR12:** Default converters/loaders for 3D models, textures, materials and shaders

**FR13:** Track resource dependencies in asset conversion process

**Rendering Systems Requirements**

**FR14:** Abstract platform-specific graphics API (RHI)

**FR15:** RHI must offer control over buffers, textures, pipelines, layouts, pipeline bindings and command lists.

**FR16:** High-level rendering code must be handled by a FrameGraph-like system.

**FR17:** FrameGraph must handle automatically texture layout transitions with pipeline barriers

**FR18:** FrameGraph must work with a separate graphics and transfer queue and it must automatically sync the command lists implicitly submitted to these queues.

**FR19:** FrameGraph must allow precise control over resource access, must define the pipeline stage and the access type.

**FR20:** FrameGraph must detect and remove unnecessary read-to-read barriers.

**FR21:** FrameGraph must batch command lists defined by multiple passes into a single gpu submit if possible.

**FR22:** FrameGraph must support the usage of buffers and textures defined outside of the graph context.

**FR23:** FrameGraph must pre-calculate all of the necessary graph data only once and reuse it over the rest of the frames if it doesn't change.

**FR24:** FrameGraph must update only the data that changed and not recalculate the entire graph if its not necessary.

**FR25:** Physically-based Cook-Torrance material model.

**FR26:** Deferred rendering architecture for opaque objects.

**FR27:** Physically-based bloom post-process effect.

**FR28:** Screen-space ambient occlusion effect.

**FR29:** Depth only pre-pass to reduce overdraw.

**NFR2:** Achieve 300+ FPS in the test machine in scenes with low mesh count and complexity.

**Object Model Requirements**

**FR30:** The object model must be based on the general ECS architecture.

**FR31:** Register new component types.

**FR32:** Create/destroy entities & add/remove data components to them.

**FR33:** Add/remove data components associated to the entity world.

**FR34:** Define iterators used to operate over entities that match a specified component tuple.

**FR35:** Add/remove tag components that have no data but can be used identify entity groups for querying purposes.

## 4.2 Systems Design

In this section we discuss the high-level system designs developed for this project, alongside some of the design philosophies used when creating these designs.

### 4.2.1 Design Principles

As we mentioned in earlier sections, a game engine is a complex system that is formed by, and must coordinate, multiple subsystems like user input, rendering, game logic, etc. Each of these systems must be performant enough to fulfil the real-time requirements of a game or application yet flexible enough to allow rapid development speed and to support unending changes that occur when developing a game. To manage all of these requirements we decided to implement the following design principles:

1. Don't introduce unnecessary complexity to solve problems: We attempt to solve problems using the simplest solution that we can conceive. When possible, we prefer implementing simple solutions that can be easily replaced by better ones if necessary, than creating a complex solution that attempts to be extensible in every possible way.

2. Prefer designing tools and utilities to aid the developer instead of future-proof generic systems: We prefer to create simple, general and specific purpose tools that a developer could use to implement simple and performant solutions to his problems, instead of trying to hack and workaround a general-purpose system that might not fit the requirements of the developer.

3. Take into account real-world usages when developing a system: When developing a system, we attempt to implement a *"realistic"* use case as early as possible, this permits us to gain insights on how the system would be used, allowing us to tweak it according. In many cases we initially mock an use case, before developing any of the internal behaviour.

4. Module-based design: We chose to divide the engine into multiple modules/systems, these behave as individual libraries that define a public and a private interface. This convention allows us to define data and functionality that is completely public within the module, but private outside of it. Instead of only relying on C++ class privacy modifiers.

### 4.2.2 High-Level Engine Architecture

For the design of the engine runtime, we chose a layered architecture where each layer is a collection of libraries/modules/systems (these terms are often used interchangeably). In this architecture, systems from a layer can only depend on systems in the same layer or below it. Overall, we treat each module/library as a individually compilable project. The module-based design allows us to better define module boundaries and its permits us to have a stronger control over dependencies because they have to be declared in the build system and, for the developer, is not as easy as just adding an *"#include library.h"* to a file. On top of this, the layered architecture makes it easier to reuse code for different engine versions without having to juggle all of the individual system dependencies. For example, if a project requires a different configuration of systems in the superior layers, this new configuration can be developed on top of the remaining layers.

we define a set of libraries and command-line tools that handle asset conversion. Lastly we define multiple automatic tests that check the functionality of many runtime systems and testing environments that can be used by the developer to check the functionality of hard to automatically test systems like rendering.

### 4.2.3 Build System

We use CMake as the primary build system for our project. This choice stems from its expensive adoption within the open-source C++ community which allows us to easily integrate external projects into our engine. However, we only test and support the Visual Studio generated solution. In line with this decision, We leverage many Visual Studio specific features, such as folders and filters, to configure the project solution effectively. To facilitate the definition of new engine modules, we maintain a standardized We use a standardized project file structure, enabling us to create CMake functions to streamline the process.

### 4.2.4 Core Systems

Subsequently, we will discuss some of the core systems that we deem important enough to be presented.

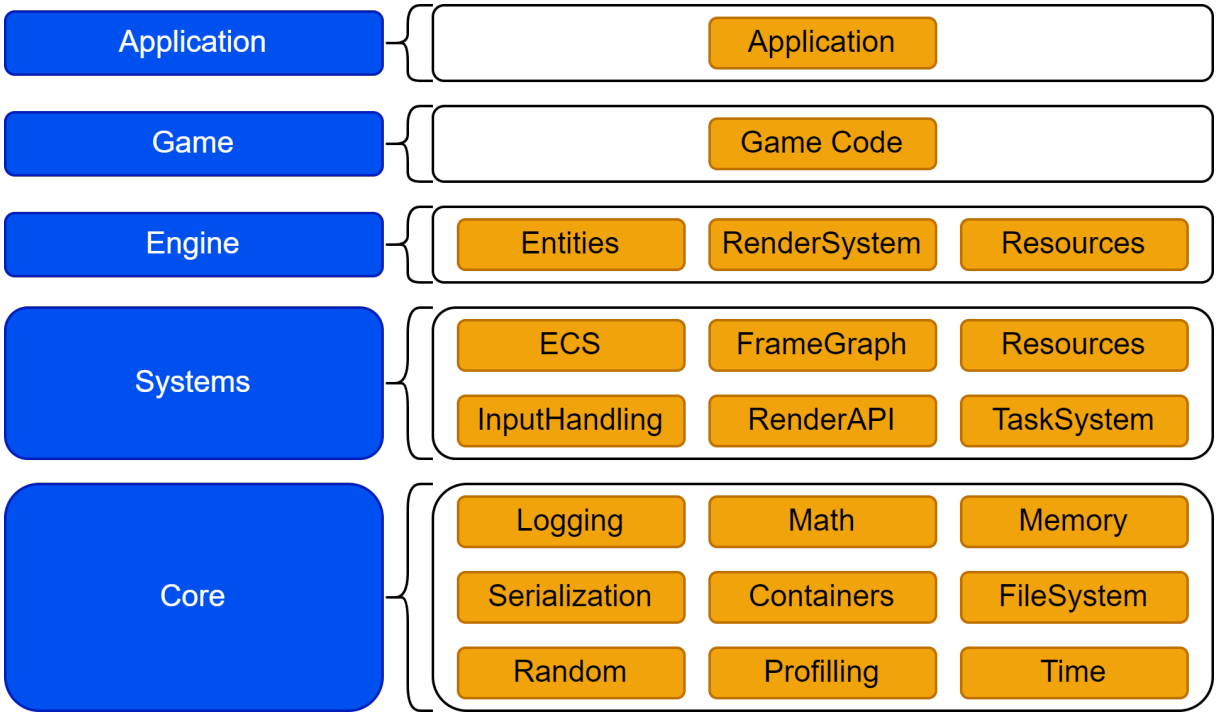



Figure 4.1: Layered architecture of CKE

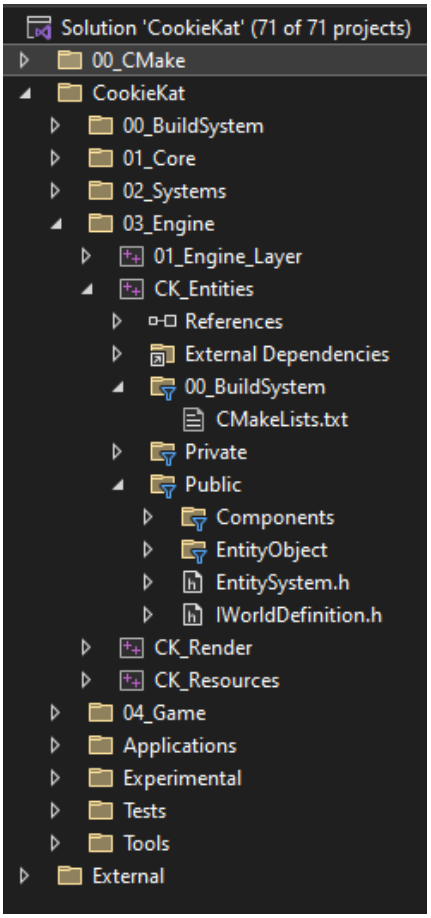



Figure 4.2: Visual Studio solution overview of CookieKat

### Serialization

We implement binary serialization functionality that handles the serialization of structs and simple classes with limited inheritance. We could have implemented JSON serialization but we required the specific type data that would have been provided by an RTTI system that we couldn't implement for this project. We could have accomplished this with certain workarounds but we decided that it wasn't adequate.

Our design is very simple and heavily inspired in Exoterica Engine's implementation and the serialization library Cereal. As seen in figure 4.3.

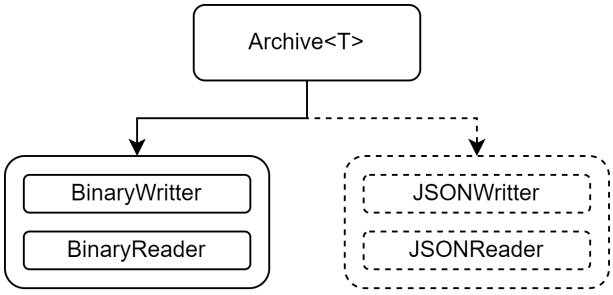



Figure 4.3: Serialization system design

### Containers and Data Structures

To ease development we decided to use, at least momentarily, containers defined in the standard library such as vectors, maps, and more. We implement many types of structures to represent strings such as:

- StringID: Hashes the supplied string at compile time.
- InlineString: Stack allocated string container.

### Logging

We define a logging system that leverages the formatting functionality of the standard library. On top of this we provide per-system log tagging and various log levels for filtering messages such as info, warning, error and debug.

### Memory Management

We implement a memory system that provides replacements to all of the basic memory operations (such as allocations, new, and more) and offers debugging capabilities such as logging memory operations. On top of this, we provide tools to implement allocators with basic memory management schemes such as linear, pool and stack.

In the final engine, these systems weren't widely used, instead relying on stack allocations and managed structures such as vectors, with the goal of accelerating development time.

### Profiling

We integrate Optick as our instrumentation tool of choice, it also serves as a sampling profiler that allows us to efficiently analyse overall engine performance.

We use platform-specific tools such as NSight Systems for GPU profiling, this tool allows us to profile both the CPU-side Vulkan calls and the GPU-side workload execution.

To precisely measure low-level CPU metrics, we utilize AMD uProf given that we work with AMD CPU's. This tool offers recording of CPU events that provide information about instruction and data cache-misses, pipeline stalls, retired instructions, misaligned loads and more.

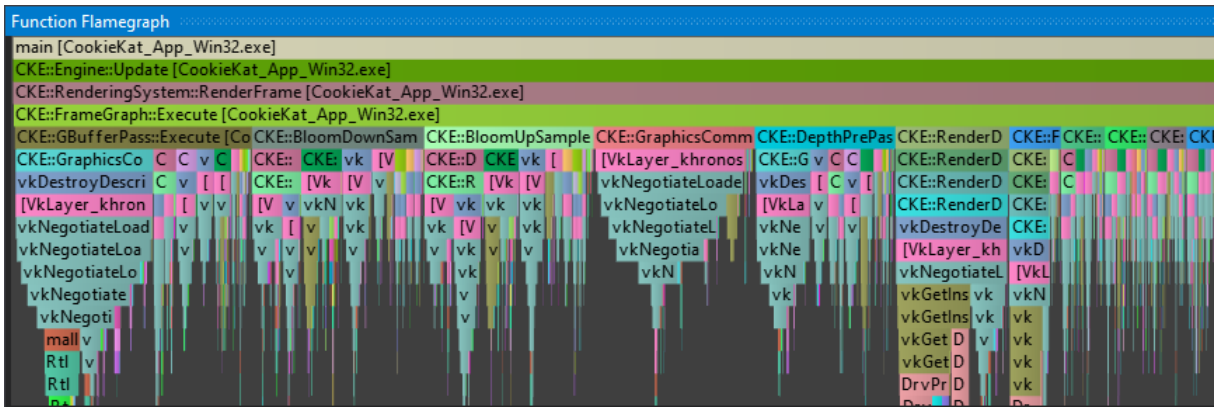



Figure 4.4: Partial frame capture of functions, recorded and visualized in Optick

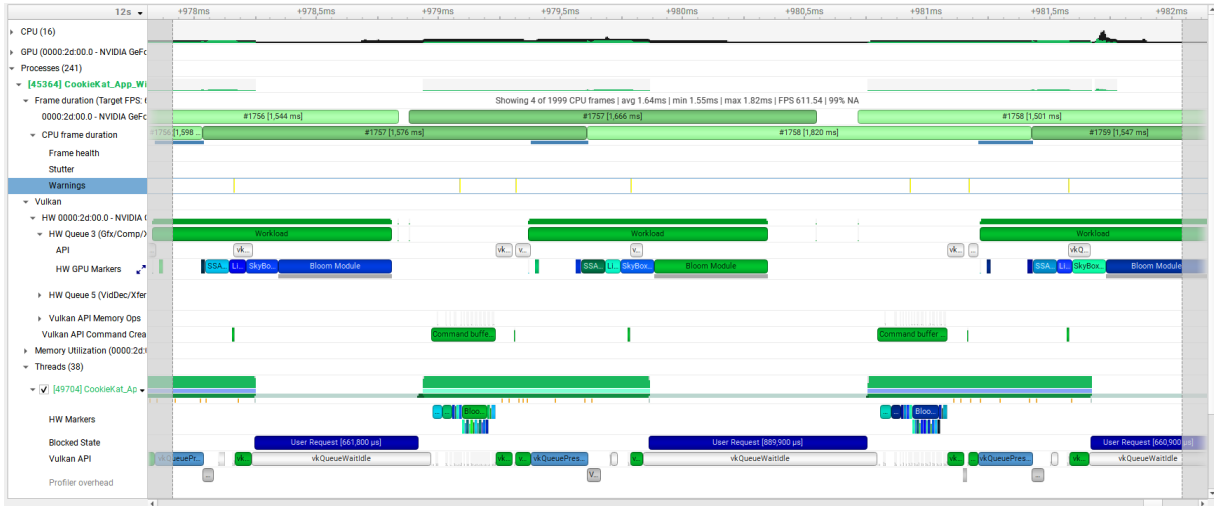


Figure 4.5: Engine capture view of multiple frames in NSight Systems

| Functions | Modules | RETIRED_BR_INST_MISP (PTI) | %RETIRED_BR_INST_MISP | L1_DC_ACCESSES (PTI) | L1_DC_MISSES (PTI) | %L1_DC_MISSES |
|---|---|---|---|---|---|---|
| glm::vec<3,float,0>::operator+=<float>(struct glm::vec<3,float,0> const &) | cookiekat_app_ | 0.09 | 0.06 | 436.34 | 0.19 | 0.04 |
| CKE::CubeMapWrapper::AreaIntegral(float,float) | cookiekat_app_ | 0.13 | 0.08 | 328.21 | 0.02 | 0.00 |
| CKE::CubeMapWrapper::GetPixelSphericalDirection(int,int,int) | cookiekat_app_ | 0.11 | 0.09 | 418.74 | 0.50 | 0.12 |
| CKE::SphericalHarmonicsUtils::SHProjectCubeMap(struct glm::vec<3,float,0> * cor | cookiekat_app_ | 0.15 | 0.10 | 390.28 | 0.07 | 0.02 |
| ValidateCmdBufDrawState(class CMD_BUFFER_STATE const &, enum CMD_TYPE, | vklayer_khrono | 5.73 | 3.07 | 559.87 | 21.16 | 3.78 |
| ValidatePipelineDrawtimeState(struct LAST_BOUND_STATE const &, [, ...) | vklayer_khrono | 7.27 | 3.91 | 535.86 | 23.90 | 4.46 |
| CKE::CubeMapWrapper::GetPixelSolidAngle(int,int,int) | cookiekat_app_ | 0.07 | 0.04 | 379.81 | 0.04 | 0.01 |
| CKE::CubeMapWrapper::GetPixelColor(int,int,int) | cookiekat_app_ | 0.16 | 0.10 | 252.03 | 0.04 | 0.02 |

Figure 4.6: Partial function hotspots and profiling data recorded and visualized in AMD uProf

### 4.2.5 Resource System

We chose an resource system design based on asset conversion/compilation. This means that source resources like textures, materials, etc, get converted to a, generally more efficient, *"runtime format"*. This format includes metadata about resource dependencies and types.4.7

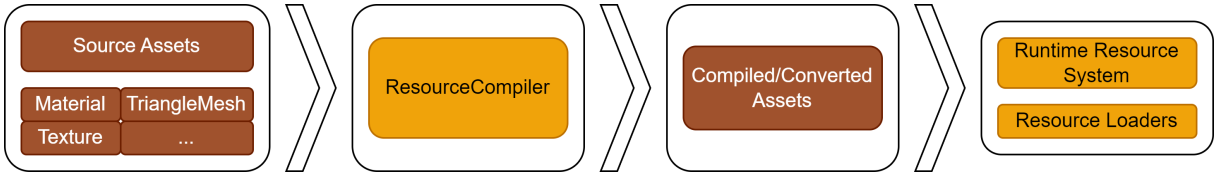



Figure 4.7: Overview of CKE asset processing pipeline

Our asset system is formed by two distinct parts, a *Resource Compiler* and a *Runtime Resource System*. We offer a general purpose resource compilation library and a command-line tool that leverages it. For the runtime engine, we offer a general purpose resource system designed to be extended to support more resource types. We implement the following resources in our engine:

- Texture & CubeMap: Due to development constrains, we only implement a simple compression to *"png"* in the compilation process. Its worth noting that better compression formats exist, such as DXT1 / DXT5, commonly used in engines like Unreal and Unity.
- Material: We implement a material as a container of texture references for a PBR material.
- Model: We currently don't apply any processing to mesh data.
- Pipeline: A pipeline is a container of shader code in SPIRV format, on load, it is reflected using *"SPIRV-Reflect"* to extract the vertex formats and resource bindings, used to automatically create the required Vulkan descriptor layout.

In the Resource Compiler, the source asset is taken as input to generate a binary file. This binary file includes a resource header that identifies the resource and maintains a record of its dependencies on other resources. Subsequently, this binary file is loaded by the appropriate resource loader registered with the resource system.

During resource loading, the resource system retrieves the resource and forwards it to the corresponding loader based on the file extension. The loader then parses the resource and integrates it into the engine. If any dependencies exist, the system recursively attempts to load them.

We implement a two phase loading design. In the first phase, "Load", we pass the source binary data to the resource loader so that it can extract the underlying resource data. In the second phase, "Install", we provide the previously extracted resource and all of the dependencies so that the loader can initialize the appropriate references and do extra processing. When loading a resource, the system first executes the load phase for the resource itself and all its dependencies, and then proceeds to the install phase. The unloading process follows the same design.

Resource data memory is exclusively handled by the resource system. Resources are identified by their unique file path and a runtime lightweight ID. When loading a resource, its runtime ID is returned to the requester, which can be used to retrieve the underlying resource data.

### 4.2.6 Rendering Systems

We chose separate the rendering-related systems of the engine into many different modules.

#### Render-Hardware-Interface

As part of the lower-level rendering systems. We have designed and developed platform-independent graphics API abstraction called the Render-Hardware-Interface (RHI). Currently, we only support Vulkan as a backend, but our API has been designed with the option to expand support to DirectX12.

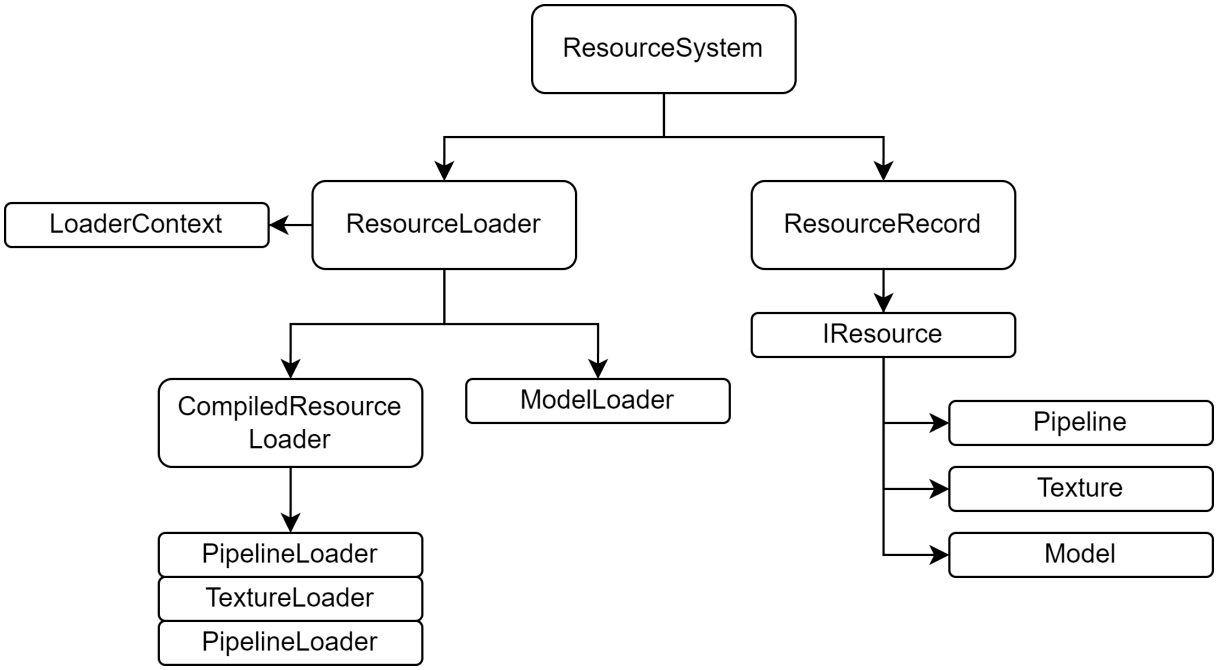



Figure 4.8: Resource System Design Diagram

When developing the RHI, we considered various options, as discussed in the **Theoretical Frame** section. We decided to develop and API that adheres to the following principles:

- Allow low-level precise control while platform-independence.
- Provide access to platform-specific functionality when required for special cases.
- Reduce the verbosity of common graphics operations, such as texture and buffer creation.
- Minimize and control the memory footprint of the system.

We decided to create an API where all of the resources are identified an accessed by their **Handle**. A handle is just a lightweight integer that uniquely identifies a resource in the system. These handles are used to identify the resources that will be used for the provided rendering operations. The underlying representation of the resource remains hidden and the system manages all of the memory allocated for the resources.

We chose this design with the goal to provide a core API as lightweight and small as possible that could later be extended with facilities like reference counted resources, resource caches, etc. This is a recurring design decision in all of the engine systems.

To allow as much control as possible in the API, we designed it to leverage most of the important features in Vulkan but we also wanted the API to be mostly compatible with DirectX12. Luckily for us both APIs are quite close in their design overall, one of the differences between them can be seen in the setup of a graphics pass. Vulkan by default requires the developer to create and cache many structures like render passes and framebuffers while DirectX12 offers a simpler and easier to use API. To avoid these differences and to reduce the complexity of the system when using Vulkan render passes, we decided to use the **Dynamic Rendering** Vulkan extension that became part of the core in Vulkan 1.3. This extension presents a rendering API closer in design to the one in DirectX12. The usage of this extension means that we lose

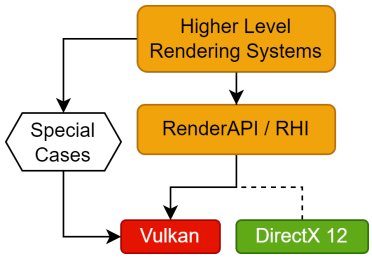



Figure 4.9: Overview of RenderAPI/RHI Design in CKE

some of the precision of the original API, but while researching we found that the usage of the more complex render pass API is only relevant in Tiled-Renderers found in mobile devices, which we don't plan supporting.

The **RenderDevice** is the main entry point for the API it manages creating and destroying all of the GPU resources like **Textures, Views, Buffers, Pipelines, Semaphores, Fences, Samplers, Descriptor Sets and Command Lists**. It handles the SwapChain and offers some quality of life utilities like automatic handling of per-frame resources in the context of having multiple frames in flight. It also hides most of the complexity of descriptor set layouts and memory pools.

### FrameGraph

In order to streamline the development of high-level rendering code, we have implemented a FrameGraph system. Given the time constraints of our project, we designed a simplified version inspired by O'Donnel's FrameGraph [37], featuring the following key features:

- Automatic resource barriers and layout transitions
- Automatic management of transient resources

Our implementation currently has limited support to buffer barriers and it does not include memory aliasing in the transient resource system or automatic queue selection and workload distribution based on availability, although these are planned extensions for future development.

The FrameGraph system consists of two important concepts: the **FrameGraph** itself and **Render Passes**. A render pass is defined as a unit of rendering code that is submitted to a specific queue type and defines input and output resources. A collection of render passes is used to create a graph, which is then compiled to generate the necessary texture layout transitions and synchronization points. After compilation, the graph can be executed to record and submit all the commands for the render passes.

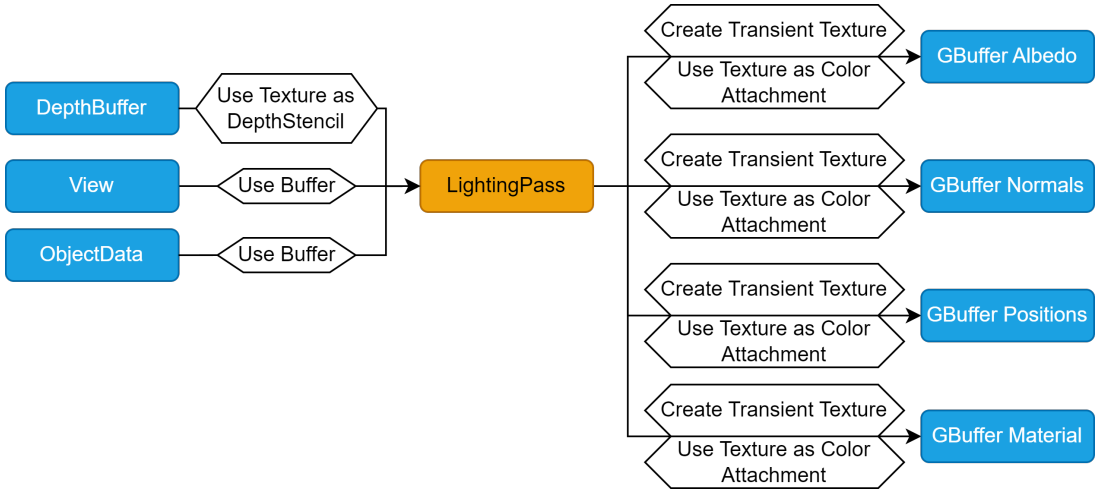



Figure 4.10: Example of the resource accesses defined by the GBufferPass in CKE

When creating a pass, the developer must define a **Setup Context**, where all resource accesses required by the pass are declared. Initially, we considered only requiring the declaration of read and write accesses, but we wanted to provide finer control without making the API excessively verbose and cumbersome. As a result, we implemented a precise base API and developed shortcuts on top of it for common operations.

To access the declared resources in the **Execute** stage, we use a **Resource Context**, where the resources are already in the required state for direct usage.

To calculate the necessary texture layout transitions, we iterate over all the render passes, acquire their resource accesses, and create transition barrier data based on the previous and new access requirements.

This transition data is used to submit barriers when recording the command lists during the execution stage. In both the setup and execute stages, all resources are identified by a **ResourceID**.

There are two main types of resources: **Transient** and **Imported**. Transient resources are created by passes and have a lifespan limited to a single frame. The most common type is transient textures used as intermediary output attachments in the rendering pipeline. Resources defined outside the FrameGraph can be accessed by render passes through importing. The system does not manage the lifetime of imported resources, which are typically used to store data spanning multiple frames, such as temporal buffers or externally created resources like the window backbuffer texture.

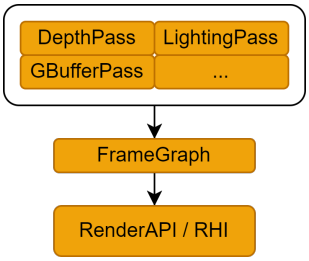



Figure 4.11: Overview of FrameGraph in CKE

When defining a transient texture as an output attachment, it is useful to specify its size relative to the window backbuffer size. To facilitate this, the system offers special flags that can be used for automatic resizing if the backbuffer size changes.

We define two types of **Render Passes**: **Graphics Pass** and **Transfer Pass**. A Graphics Pass is provided with a graphics command list for execution, which will be submitted to the general-purpose graphics queue. For accessing the transfer-optimized queue, the option of defining a Transfer Pass is available, the same is provided for the compute-optimized queue.

All internal memory allocations performed by the system are managed by the **FrameGraphDB**. This class oversees the allocation and release of most of the heap-allocated data used by the system.

### 4.2.7 Rendering a Frame

#### Data to Render a Scene

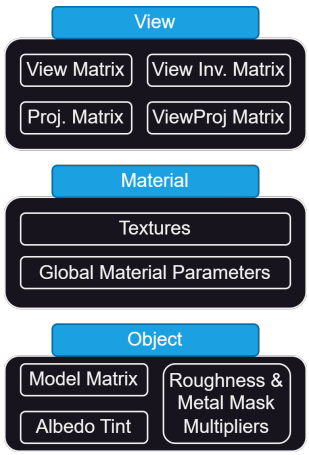



Figure 4.12: Separation of rendering-related scene data in different buckets

Based on our analysis of existing real-time 3D graphics engines and games, we developed systems that efficiently gather and process scene objects, preparing the necessary data to render them in the GPU. We leverage our Entity-Component-System (ECS) architecture to query all scene objects that posses render-related components and perform the required processing on them.

To take advantage of the shader resource binding architecture offered by modern graphics APIs such as Vulkan, we have chosen to organize the scene data into three major buckets: Per-View, Per-Material and Per-Object.

While the Per-View and Per-Material data is bound as needed, we optimize the number of descriptor bindings required for each object by maintaining a buffer on the GPU that contains all Per-Object data for the entire scene, this includes data like model matrices and material modifiers.

Within the Per-View bucket we primarily store information such as the camera view matrix, projection matrix, the CPU-computed View-Projection matrix, and their inverses.

As for the Per-Material data, we currently store only the essential textures required for the PBR shading model, including the albedo, roughenss, metalness and normal maps.

This separation of scene data into distinct buckets allows for efficient resource management and minimizes the number of descriptor bindings necessary during rendering, ultimately optimizing performance.

### Rendering Pipeline

We use a **Deferred Rendering** model to handle the rendering of all opaque objects within the scene. Transparent objects are currently not supported. The complete rendering pipeline, depicted in figure 4.13, outlines the rendering steps of our engine.

We begin with an initial depth pass to generate a complete scene depthbuffer that enables efficient culling of occluded fragments in subsequent passes. We chose a simple deferred rendering architecture, where we output the albedo, position, normals and various materials parameters such as roughness, metalness, and reflectiveness to the GBuffers. The SSAO algorithm we utilize leverages the previously computed depth buffer to reconstruct view-space positions for fragments.

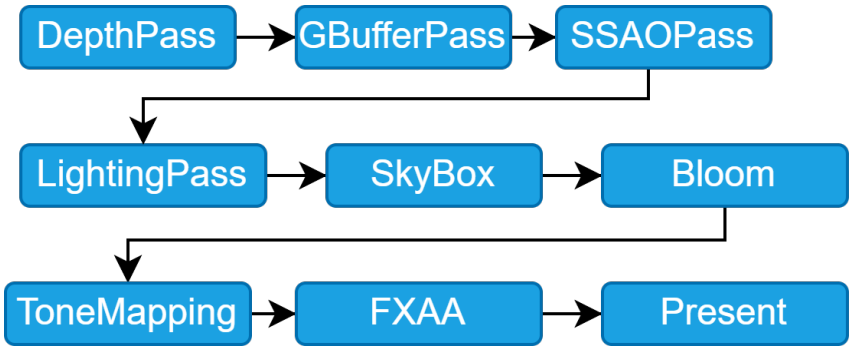



Figure 4.13: Default PBR Rendering pipeline implemented in CKE

We realize an initial depth pass to generate a scene depthbuffer that will later be used to more efficiently discard all of the occluded fragments in the following passes. We use a simple deferred rendering architecture where we output the albedo, position, normals, and material parameters like roughness, metal mask and reflectiveness. The SSAO algorithm that we implement uses the previously calculated depth buffer to reconstruct the view-space positions of the fragments. Although we execute the GBuffer pass and SSAO pass sequentially, we could execute them in parallel because of their resource accesses.

We use a Physically-Based Rendering (PBR) material model to light the scene. This includes incorporating a diffuse environmental component (Implementing a specular component is out of the scope of this project so we use a constant value instead). Following the lighting stage, we draw the scene's skybox and proceed to apply post-processing effects like bloom, tone-mapping and FXAA. Finally, we copy the final render to the backbuffer and we present it for display.

### Physically-Based Rendering

In order to achieve a more accurate representation of materials and their interactions with light, we chose to integrate Physically-Based Rendering (PBR) techniques into our game engine. Our approach focuses on implementing a simple material system where we focus on the Bidirectional Reflectance Distribution Function (BRDF) to characterize materials. For our implementation, we take inspiration from existing research such as EA's "Moving Frostbite to PBR" course [10] and Filament's design [19].

The BRDF describes the behaviour of a surface by combining two main components: the diffuse and specular terms. In our model, we have chosen to adopt a standard microfacet BRDF as the foundation for our materials.

For the diffuse term, we have implemented both the Lambertian and Burley's model (In the final engine materials use Burley's diffuse term). Lambertian diffuse it's characterized by the following equation (where $\rho$ represents albedo):

$$f_{\mathrm{d}} = \frac{\rho}{\pi}$$

While Burley's model is described by the following equation:

$$f_{\mathrm{d}} = \frac{\rho}{\pi} F_{\mathrm{Schlick}}(n, l, 1, f_{90}) F_{\mathrm{Schlick}}(n, v, 1, f_{90})$$

where:

$$f_{90} = 0.5 + 2 \cdot \alpha \cos^2(\theta_{\mathrm{d}})$$

As for the specular component, we have employed the Cook-Torrance approximation, which is a widely used model in real-time graphics. It combines the GGX Normal Distribution Function (NDF) with the Smith Geometric term to achieve accurate specular reflections. Additionally, we have utilized the Schlick approximation for the Fresnel term, which provides a practical and efficient solution for simulating Fresnel reflectance. The specular term can be described by the following equation:

$$f_r(v, l) = \frac{F(v, h, f_0, f_{90})G(v, l, \alpha)D(h, \alpha)}{4(n \cdot v)(n \cdot l)}$$

By adopting these Physically-Based Rendering techniques, we aim to enhance the realism and visual fidelity of our rendered scenes, ensuring that materials exhibit more accurate behaviour under various lighting conditions.

### Bloom

To approximate the glowing/light bleeding effect similar to that produced by bright light sources, we decided to incorporate a bloom technique based on the approach presented by Jorge Jimenez in Advanced Post-Processing in COD: Advanced Warfare [32]. Although it might be more efficient to implement this technique with compute shaders to avoid begin/end rendering overheads, we chose to employ multiple full-screen quad post-processing passes due to the limitations of the engine during the development phase.

Our implementation consists of five downsample steps followed by four upsample and a combine step. We utilize the HDR scene color without applying any threshold filter to the initial input. Upon completing the downsampling and upsampling stages, we blend the resulting color with the original scene color using a fixed factor of 0.04.

### SSAO

We implement Crytek's screen-space ambient occlusion algorithm, using a surface normal oriented semi-sphere to estimate collisions (and therefore, light occlusion zones) instead of a complete sphere. We pre-generate all of the required noise patterns on startup, with this data, we calculate the occlusion in a fragment shader of a SSAO graphics pass. Subsequently, we apply a box blur filter to eliminate possible noise patterns.

### Fast-Approximate Anti-Aliasing

We integrate the original Fast-Approximate Anti-Aliasing (FXAA) implementation of Timothy Lottes. through its application, we can reduce the visual imperfections caused by aliasing, commonly described as the jagged edges that appear in a render. The algorithm analyzes the image, identifying sections where these artefacts appear and applying a subtle blur to smooth out said edges.

We considered other options such as Temporal Anti-Aliasing (TAA) and Multi-Sample Anti-Aliasing (MSAA) but we chose to implement FXAA due to its ease of integration and the compatibility with our deferred rendering model.

### Environment Map

For the creation of our skybox, we decided to implement a straightforward approach utilizing a CubeMap. In order to capture the diffuse irradiance that objects receive from the world environment map, we employ spherical harmonics. We utilize an approximation method that uses 9 coefficients. Although, given that we only have one environment map in the world. We could have used more coefficients to achieve a higher fidelity result for more complex environment maps.

### 4.2.8 Object Model Design

In essence, a game world is composed by an extensive amount of data and logic that transforms it in various ways. An object model's job is to provide general tools to handle this complexity. After analysing numerous designs, we have concluded that our goal is to provide developers with a set of tools that offer the flexibility to choose the desired level of abstraction when describing the game world.

For our object model, we aimed to develop a design that combines the low-level control, simplicity, flexibility, and multi-threading capabilities of an Entity-Component-System (ECS). However, we recognized that relying exclusively on an ECS system for gameplay code development is not efficient, despite its suitability for low-level data processing tasks. It can become cumbersome when implementing higher-level gameplay features like AIs, character state machines, player inventory, and more. Consequently, we have adopted the following design approach:

As our low-level gameplay system, we use an archetype-based ECS model that is heavily inspired by database design principles. Instead of using this model exclusively for all gameplay code, we plan to utilize it to implement higher-level, purpose-specific systems designed to ease the development of high-level features like AI, player behaviour, etc. This approach is used by game engines such as Overwatch's. which utilizes an ECS system as low-level object model and glue between all other engine systems [21], while offering a high-level graph-based visual scripting system built top of this ECS model [43].

In this design, the ECS model is used to define the game world data and declare its accesses through ECS systems, allowing us to define these operations in such a way that we can benefit from features such as automatic parallelization of game logic (not implemented in the final project). All of this while the systems offer API's designed for developing high-level gameplay features.

We offer core functionality such as:

- Single-Multi Component Iterators
- Global Components
- Job System Integration (Early prototype)

The ECS model is designed as a table-based database with the following data elements alongside their database counterpart if they have any:

- EntityID (Primary Key): An EntityID serves as a unique identifier used to access the related data associated with an entity in the system.
- Component (Entry): A Component is a plain data structure identified by a component ID and linked to a specific entity.
- ComponentSet: A ComponentSet represents a collection of unique components.
- Archetype (Table): An Archetype can be visualized as a matrix, where each row represents an entity, and each column corresponds to a specific component type. Identified by a ComponentSet, an Archetype stores only the data of entities that share the same component set. In the system, there can be an archetype for each combination of component sets.
- Singleton Components: Singleton Components are plain data structures identified by a singleton component ID and associated with the world as a whole.
- Archetype Component **[Not Implemented]**: An Archetype component is a plain data structure associated to an archetype, not implemented in the final project.

These data elements form the foundation of the ECS model, facilitating the organization and management of entities, components, and their relationships within a game or simulation system.

In addition to the aforementioned elements, our implementation includes single and multi-component iterators, which allow querying and operating on a list of components containing at least the components specified in a provided component set.

To ensure flexibility and maintain a decoupled design, we initially developed a type-less API and later introduced a typed templated version. This approach allowed us to separate the complexities of managing a templated API from the core system, enabling easier extensibility and customization.

Regarding memory management, our system follows a straightforward scheme where we predeclare the maximum entity count. This information is utilized to allocate and manage complete archetypes whenever we encounter a new one. While this approach may not be the most sophisticated, it was chosen due to its simplicity and adherence to the constraints at hand.

To aid in debugging and analysis, we have implemented useful tools that log and provide snapshots of the database state. These debugging tools enable developers to monitor and understand the state of the ECS system during runtime and aid in the development of automatic tests.

Additionally, we have integrated component iterators directly into the engine's job system as **ECS Jobs**. This experimental functionality allows for enhanced performance and parallelization by leveraging the benefits of the job system in conjunction with the ECS model. This integration opens up new avenues for optimizing and scaling the system's performance, although it should be noted that this feature is still in the experimental phase.

## 4.3 Implementation Philosophy

Considering the project's intricate nature, extensive scope, and performance demands, our approach to implementing these systems is guided by the following principles, taken from experience game developers in the industry [3] [4] [22] [35] as well as established embedded programming practices [59]:

1. Prefer simplicity in solving problems: Whenever possible, we prioritize using simple language tools to address challenges.
2. Control the use of Templates: We limit the use of templates and keep their design simple to avoid increased compilation times and the complexity. If we deem that they improve the system in development and we decide to use them, we apply different methods to minimize their impact in compile times.
3. Limit the use of operator overloading: We avoid the use of operator overloading unless the operations are deemed obvious (such as vector and matrix multiplications) to minimize complexities and maintain code clarity.
4. Avoid exceptions: We adopt a no-exceptions policy given the performance and memory constrains we have. We implement other mechanisms such as error codes and toggle-able asserts.
5. Limit data inheritance and avoid multiple data inheritance: To avoid potential complications and cognitive complexity, we restrict data inheritance and refrain from multiple data inheritance. Prefer composition over inheritance in such cases.
6. Clear memory ownership and per-system memory management: Each system must manage its own memory space according with its requirements and clearly define ownership transfers when necessary.

By adhering to these principles, we aim to develop a well-structured and efficient game engine given our constrains and context.

We don't follow a single programming paradigm exclusively, instead, we treat them as sets of tools, using the practices they present accordingly where we see fit, based on the context and the problem to solve. [50] [18]

## 4.4 Everything can be Data

When developing the FrameGraph, one of the fundamental principles of the system is treating all operations as data and declaring them beforehand. This approach enables the system to manipulate and

optimize this data effectively. In many ways, this resembles how compilers operate. They have a comprehensive view of program data and operations before execution, allowing them to optimize the code to the best of their abilities.

The concept of treating resources and operations in a system as pure data, which can be inspected and manipulated, offers significant benefits. It enables systems to automatically eliminate redundant operations, self-diagnose their state, and even use information about data access to automatically schedule multithreaded work. This capability forms the foundation of both FrameGraph and ECS models, leveraging the power of a data-oriented design.

# Chapter 5

# Validation

To validate the systems developed in this project, we conducted a comprehensive range of tests. This chapter delves into the methods employed and presents the results obtained. Specifically, we will discuss module-level tests designed to validate the behaviour of individual modules (§5.1). We will also explore various tests aimed at assessing the overall performance of the engine, including specific tests for performance-critical systems (§5.2). Lastly, we will provide comparisons with existing engines, focusing primarily on performance and visual disparities (§5.4).

## 5.1 Functionality Tests

To validate the functionality of engine modules and ensure that the systems behave correctly even after refactors or changes, we have implemented an extensive suite of functionality tests (unit tests) for various engine modules.

For our testing framework, we opted to use *Google Test*, as it provides the necessary features we require and integrates well with Visual Studio. Currently, our focus has been primarily on implementing tests for core engine modules such as serialization, containers, and others. Additionally, we have conducted tests for higher-level systems like the ECS object model, addressing various cases where bugs were previously identified.

## 5.2 Performance Tests

In order to validate that our systems perform within the defined performance constraints, we conducted a series of comprehensive tests. These tests encompassed both engine-wide performance evaluations and system-specific assessments.

During the development of the ECS object model, we recognized that the performance of the system under heavy load was a important metric to have. This metric helped us fine-tune different implementation methods and evaluate the performance impact of our design choices. We conducted separate tests for the core ECS API and the templated & lambda-based API to measure any potential performance overhead associated with the latter. To gain a comparative perspective on the performance results, we also implemented the same tests using other ECS-based libraries such as *EnTT*.

To assess the overall performance per frame, we employed *Optick*, a profiling tool specifically designed for game engines. We utilize profiling markers on all of the main engine systems.

To measure more precisely the performance of GPU related tasks, such as rendering passes, we utilized *NSight Systems*. This tool proved invaluable in identifying bottlenecks and performance issues within the rendering systems, allowing us to optimize GPU performance effectively.

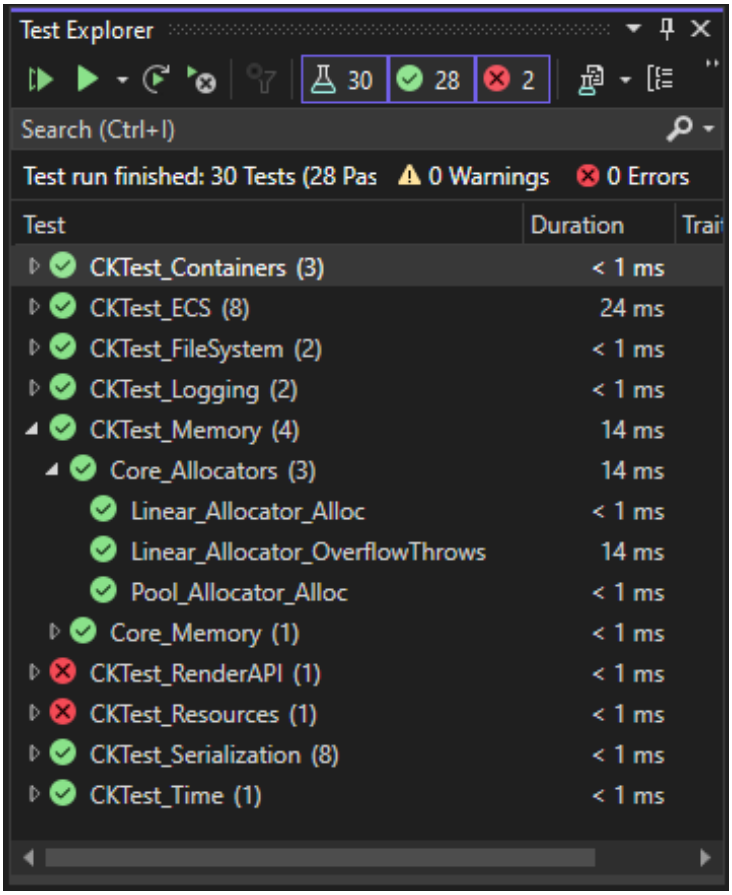



Figure 5.1: Test explorer in Visual Studio

## 5.3 Rendering Tests

We planned to develop an automatic testing framework specifically designed for rendering code. This system would allow the developer to write blocks of graphics code that output to a final image. This image would then be compared (within a threshold) with the predefined, expected image to check if the test passed.

Due to development constrains, this system is still in an experimental state. It offers a configured sandbox environment where the developer can create small rendering applications with the purpose of testing rendering code. Its was mainly used to validate FrameGraph's multi-queue synchronization.

## 5.4 Comparisons With Existing Solutions

To evaluate the capabilities of our engine, we realized different types of comparisons between our engine and existing solutions. In this section we will discuss some of these comparisons regarding the object-model of the engine and the graphic quality of the rendering pipeline.

### 5.4.1 Object Model Comparisons

To compare the performance of object models, we conducted three test cases in our engine, Unity, and EnTT. Each test case involved defining a set of components, assigning them to a specific number of entities, and implementing logic to update the component data. We then measured the execution time of updating all entities (excluding setup time).

For these tests, we defined three arbitrary components: "Position," "Velocity," and "Acceleration." Each component consisted of three 32-bit floating-point numbers. We created a total of 750,000 entities, divided into three archetypes or buckets based on their components:

- 250'000 entities with only a "Position" component.
- 250'000 entities with both "Position" and "Velocity" components.
- 250'000 entities with "Position", "Velocity" and "Acceleration" components.

We designed the following processes for each test scenario:

- Pos: This process involves iterating over entities with a position component and incrementing their values by a fixed amount.
- PosVel: In this process, we iterate over entities with both position and velocity components, performing basic operations that access and modify the data of both components.
- PosVelAccel: For this process, we iterate over entities with position, velocity, and acceleration components, carrying out basic operations that access and modify the data of all three components.

In these processes, we primarily focus on simple arithmetic operations. This choice allows us to measure the iteration and data access overheads of each system, rather than the time it takes to execute complex data transformations.

We selected these specific cases to cover common scenarios of low-level gameplay programming encountered in game development. They enable us to evaluate important aspects, particularly when comparing multiple ECS models. For example, the "Pos" and "PosVel" cases allow us to assess the iteration over a single component or a reduced tuple when some entities possess additional components. On the other hand, the "PosVelAccel" case helps us evaluate the performance of iterating over an entire archetype, minimizing the overhead associated with switching between multiple archetypes during iteration.

#### CookieKat ECS VS Entt

We implemented a standalone executable that tests the same configuration of components and systems in both the engine ECS model, and the popular ECS library Entt. We use the previously presented entity

numbers of 750'000 (Pos), 500'000(PosVel) and 250'000(PosVelAccel) for each case.

| **Framework** | **Pos** | **PosVel** | **PosVelAccel** |
|---|---|---|---|
| CKE | 1.78333ms | 3.86667ms | 3.125ms |
| CKE Templated API | 1.45833ms | 4ms | 3.15833ms |
| Entt | 1ms | 6.96667ms | 6ms |

Table 5.1: Performance Comparison

### CookieKat ECS VS Unity GameObjects

Comparing different types of object models can be challenging due to the wide variation in results based on the specific implementation methods employed. It is possible to develop alternative systems that mitigate the limitations of Unity's object model, but this often leads to creating an additional object model layer on top of the existing one, resulting in unnecessary complexity. Consequently, for the purpose of comparison, we decided to develop Unity's test using the base features of its object model. It's important to note that there are more performant ways to design the tests, such as batching all data transformations into a single update callback and avoiding expensive null checks on Unity objects. However, we chose to utilize the standard features of Unity's object model to ensure a fair comparison between the default behaviour of our CookieKat ECS and Unity's GameObjects.

| Number of Entities | Time (ms) |
|---|---|
| 750,000 | 91.86 |
| 500,000 | 68.55 |
| 250,000 | 29.96 |
| 100,000 | 13.30 |
| 20,000 | 2.61 |
| 1,000 | 0.12 |

Table 5.2: Results of the *"Pos" case in Unity*

We create a *"Monobehaviour"* component for each ECS counterpart, these hold the data and the behaviour to transform it in the *"Update"* callback. We handle the script update order by using unity's own system.

| **Framework** | **Pos** | **PosVel** | **PosVelAccel** |
|---|---|---|---|
| CKE | 1.78333ms | 3.86667ms | 3.125ms |
| Unity | 91.86 | 122.46ms | 110.47ms |

Table 5.3: Results of the test cases in Unity compared with our solution

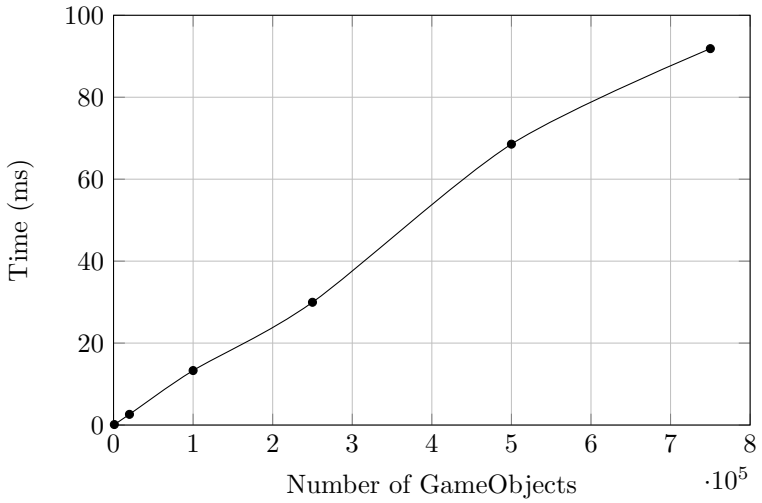



Figure 5.2: Unity scaling with number of GameObjects in "Pos" case

**Discussion**

Looking at the results between our implementation of ECS and Entt, we can see that we lack behind in single component iteration (Pos case) but we achieve much better results in multi-component iteration (PosVel and PosVelAccel cases). In this test case we iterate over 750'000 entities that we split in 3 archetypes, 250'000 entities have only a position component while the rest are assigned extra components. We believe that the performance differences come from overheads in our system when iterating over entities of different archetypes. That said, we also believe that this *extra work* that we have to do allows us to achieve close to 2x performance increase in the multi-component test cases. Its worth noting that for our tests, we didn't leverage advanced EnTT features such as groups that in specific cases perform much better.

We achieve expected results in the case of the Unity comparisons. Even though there might be an argument of Unity's model of gameobjects leading to poor data accesses that lead to data and instruction cache misses. We believe that in this case, the test performed poorly because of major overheads when calling the *Update* method in each object. We believe that this is caused in part because of the way Unity is structured as a C++ core runtime and a C# scripting side, which, using the mono scripting backend that we believe the editor uses, introduces overheads when calling the C# Update methods from the C++ engine side. That said, this is still a major problem in engines like Unreal that use exclusively C++. This is because of the overhead and instruction cache misses of calling a high amount of virtual *Update/Tick* methods. This is a known problem of the Component-Object model and there possible solutions like manually batching/aggregating update behaviour [30].

### 5.4.2 Rendering Comparisons

We created multiple test scenes to validate and compare the graphics capabilities of our engine. We chose Unity's Universal Render Pipeline (URP) renderer as a target for comparisons. We employ the same models and textures in both engines for all of the rendering comparisons, we attempt to match the graphic features of both engines (SSAO, Bloom and such) even though we have limited information regarding Unity's implementation and the parameters they use.

We realized *Roughness and Metallic Comparisons* to showcase the material behaviour of each engine when using the same parameters. We also showcase a complex model with texture-driven material parameters to visualize the graphic differences of each engine in a common scenario.

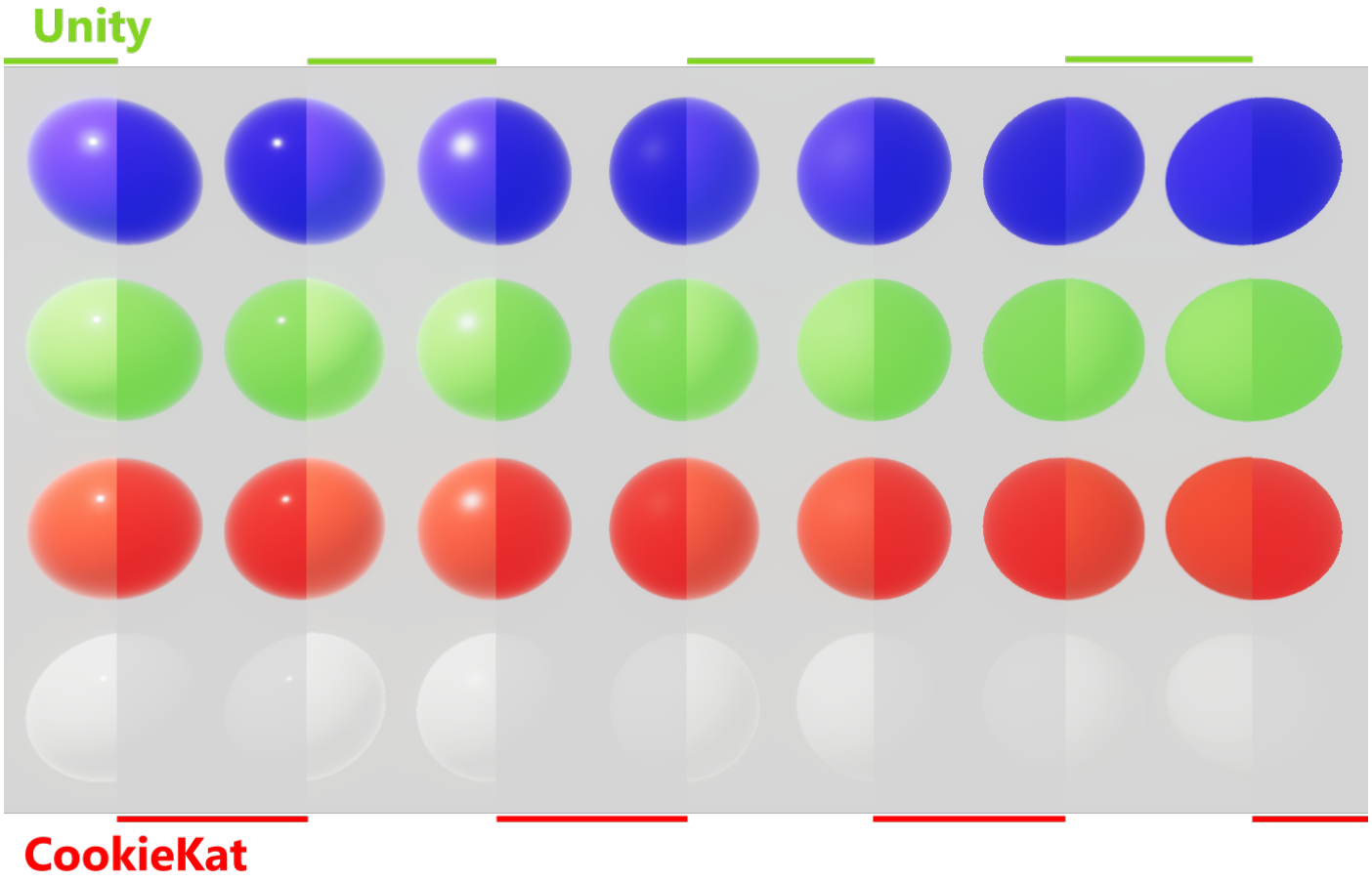



Figure 5.3: Roughness comparisons between CookieKat and Unity

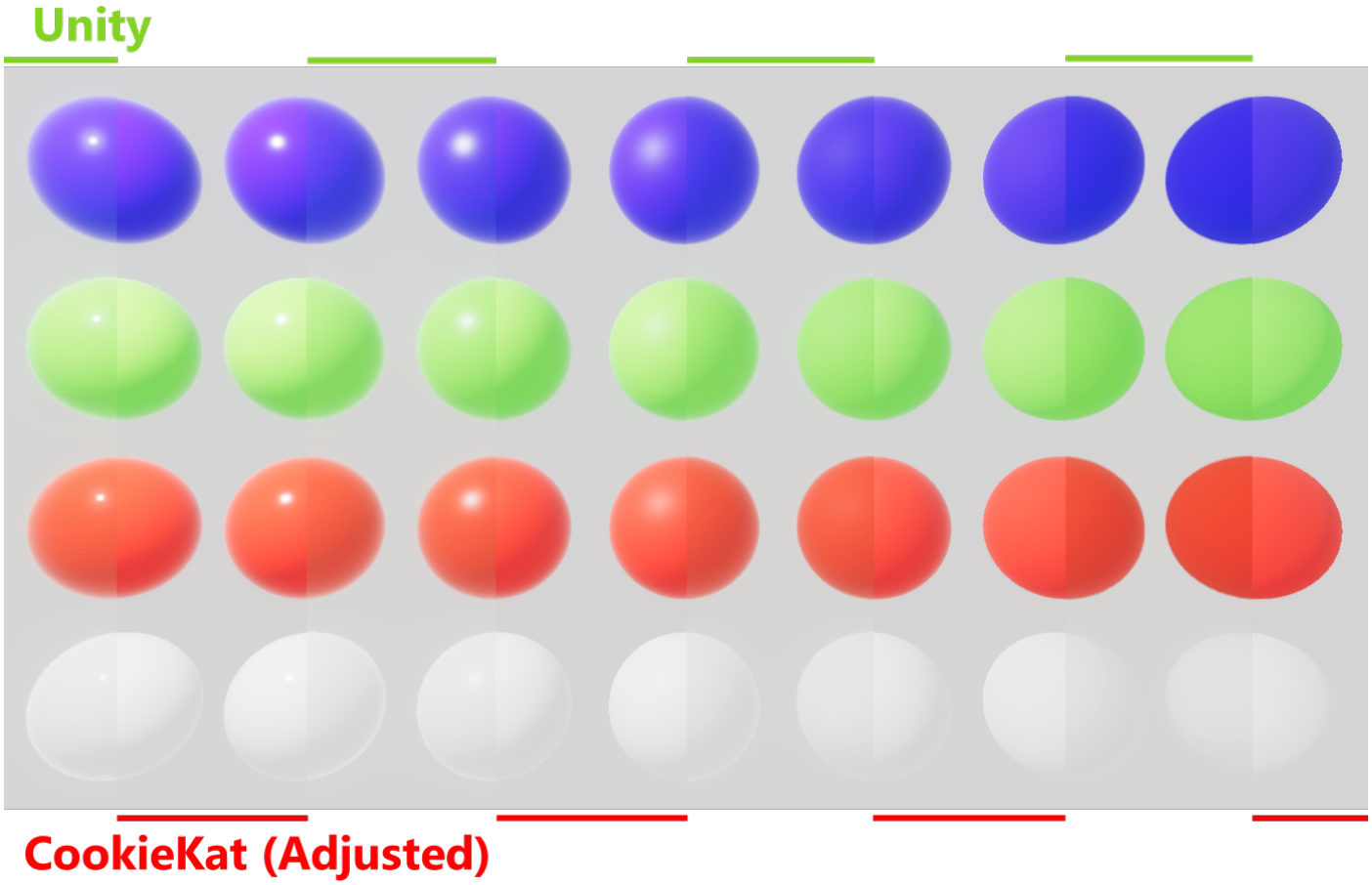



Figure 5.4: Roughness comparisons between CookieKat (Adjusted) and Unity

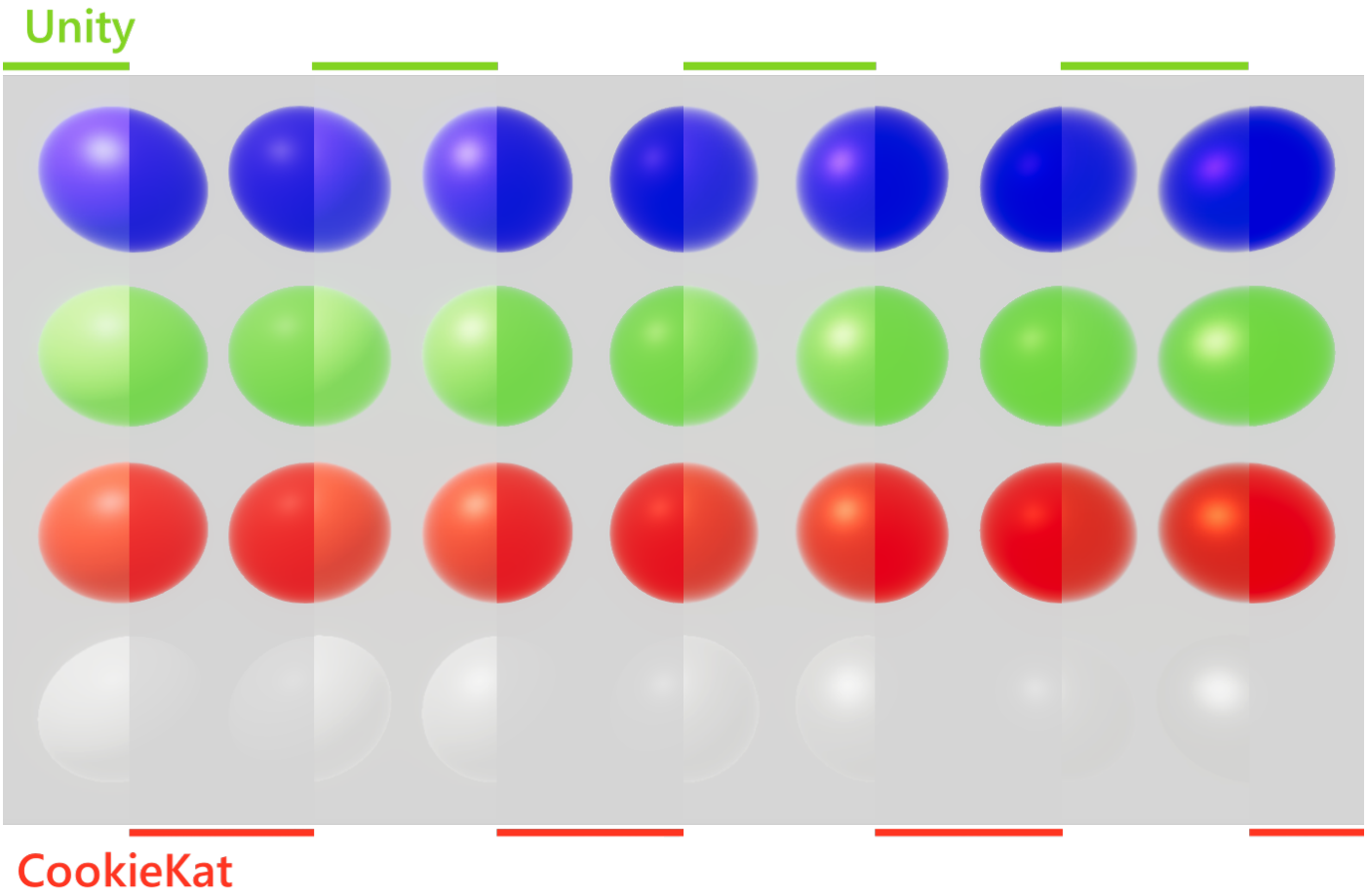



Figure 5.5: Metallic comparisons between CookieKat and Unity

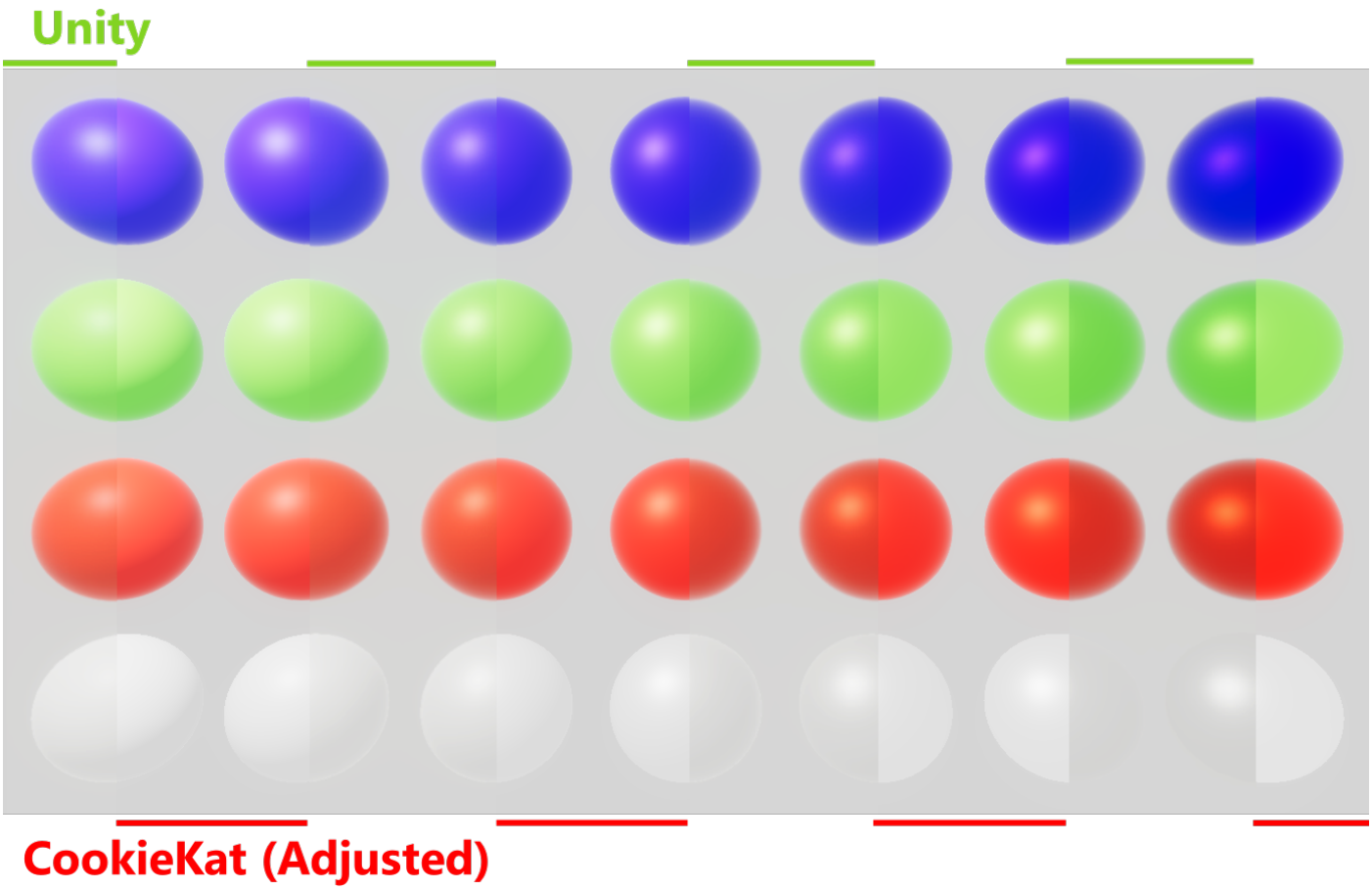



Figure 5.6: Metallic comparisons between CookieKat (Adjusted) and Unity

**Roughness and Metallic Grid Comparison**

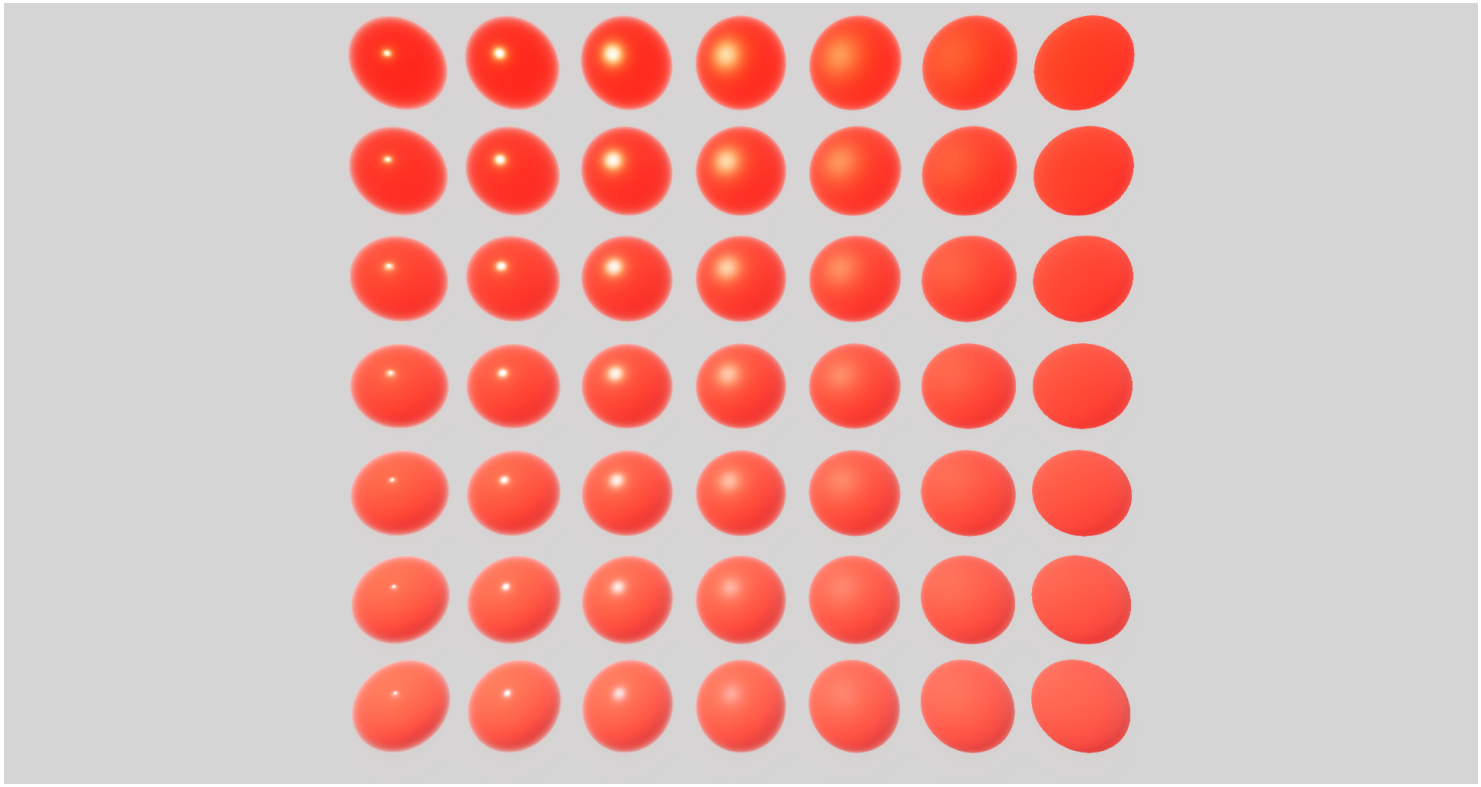


Figure 5.7: Sphere Grid in CookieKat (Adjusted): 0.05 to 1 Roughness (left to right), 0 to 1 Metallic (bottom to top)

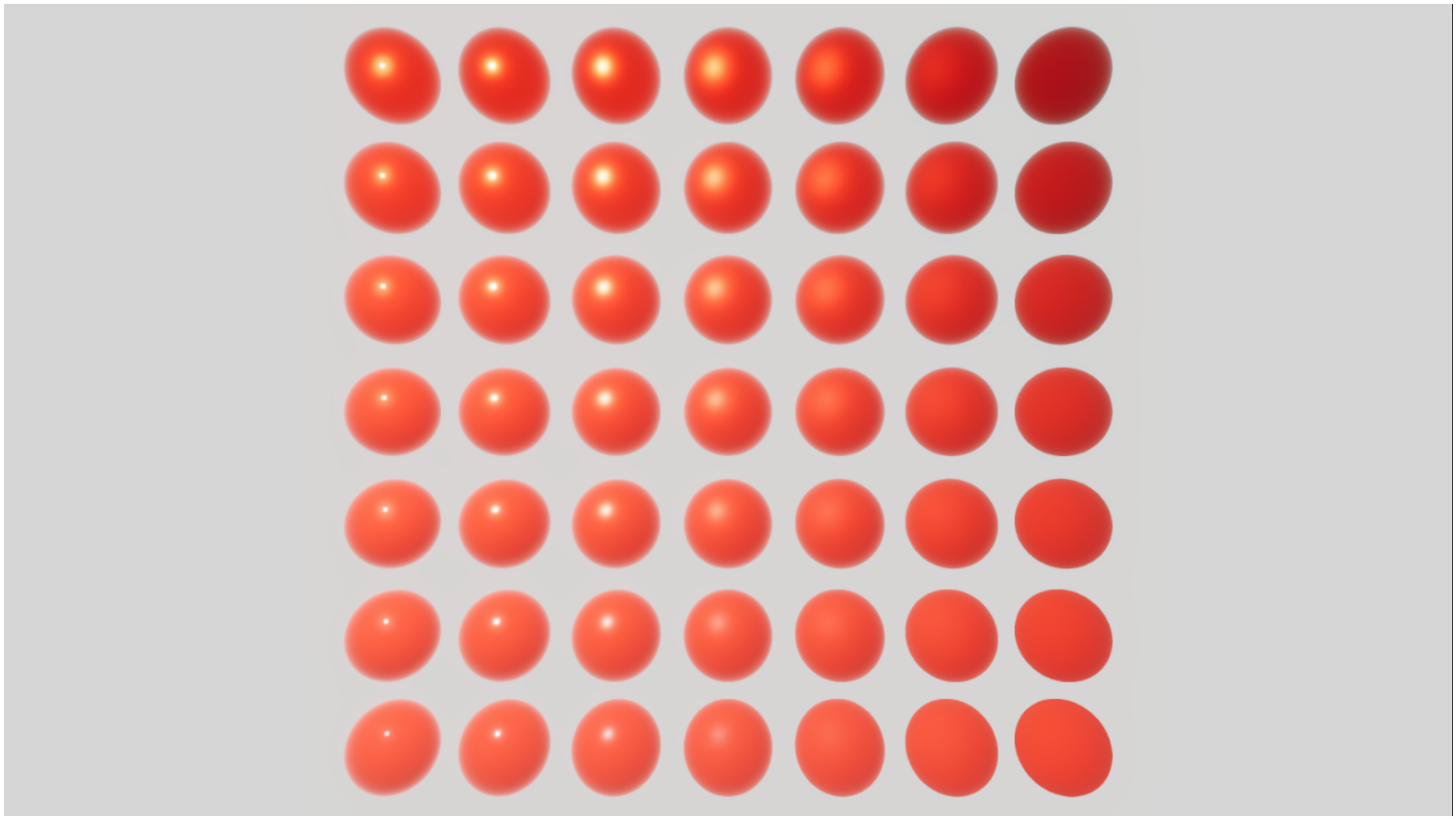


Figure 5.8: Sphere Grid in Unity: 0.05 to 1 Roughness (left to right), 0 to 1 Metallic (bottom to top)


Figure 5.9: Cerberus comparisons between CookieKat and Unity

### Discussion

Even though we tried to match object and light properties in each test scene, we can see clear differences in material behaviour using the same parameters, as showcased in figures 5.5 and 5.3. We attribute these differences to implementation details regarding how the light intensities get processed, On top of this, we suspect of possible differences in the tone-mapping implementation of ACES curve.

When we adjust the light intensity and the environment specular contribution (which our engine doesn't fully implement and instead relies on a constant fixed white value), we can see the final graphic result becoming quite similar to Unity's results as showcased in figures 5.6 and 5.4. We observe overall brightness differences that we believe are caused by not matching exactly the light intensities in both engines. More importantly, we observe quite a different intensity in the spheres borders, caused by the Fresnel effect from the environment light. We can specially observe the lack of a real specular environment factor when comparing metallic materials (fig. 5.7 and 5.8).

Additionally, we observe clear differences in general brightness values in cases 5.9. We believe this effect might be caused either by errors we might have made converting the PBR textures to Unity's required format, or differences in the tone-mapping implementation.

# Chapter 6

# Conclusions

In this chapter we will discuss the accomplishments made with the project, some of the lessons learned in the development of it and finally we will talk about the planned future work.

## 6.1 Achievements

At the beginning of this project, we established a set of overarching goals. Although some of these goals evolved during the project's development, we will now examine each one individually and evaluate the extent to which we believe we have achieved them.

- **Analyse existing game engines and design an custom engine based on the analysis:** We analyzed some of the more popular game engines and standalone libraries in the theoretical frame section. Furthermore, we designed a real-time engine based on these analysis.
- **Develop a custom game engine based on the previously defined designs:** We developed a real-time engine that integrates an resource system based on a asset conversion workflow, a high performance object model based on ECS architecture that leverages modern CPUs, and finally, an advanced rendering system that integrates a graphics API abstraction layer, and a high level rendering framework based on a FrameGraph architecture.
- **Validate the functionality and performance of the developed engine:** We implemented many functionality/unit tests to validate the correct functionality of individual engine components. Realized and integrated multiple performance tests to measure the overall performance of the engine and of some individual systems like the object model. Finally created multiple scenes to test and compared the graphic results of our material model and rendering pipeline with other renderers like Unity's Universal Render Pipeline (URP).

Given the scope of the project and the available resources, we believe that we've met the all of the goals that we had for this project.

## 6.2 Lessons Learned

This was a project where every time we were developing or changing a system we were presented with so many choices of how to do things, we tried to study these choices carefully, trying to find the perfect choice and, while this is good to some degree, sometimes it led to *decision paralysis*. In the end we learned the obvious lesson that all choices have pro's and con's, even if they are not obvious at the moment of making them. If we are not sure about how to implement some system or how to design some API, the most straightforward and simple thing to do is to just choose a path and see where it leads to.

Developing a project like this that seemingly has no end can be daunting some times. You can spend weeks developing something when you realize that there is still a very long way to have something to

show for. This can sometimes incite to rush things to just have something barely working even if its not *"good"*, and while this is actually a good thing sometimes, its also necessary to remember to, at some point, fix the mess that was made in a rush because if not, this technical debt will end up drowning the project and it will be harder and harder to work with it.

## 6.3 Future Work

There is still a significant amount of work remaining in the project, this section will focus on exploring some of the more interesting topics such as:

- Build Pipeline: Expanding and improving the build system is also a notable option, for this project we integrated a simple build scheme where we manually do a lot of the work, this could be vastly improved. We could create a complete automatic build pipeline that builds, tests and packages the engine and its libraries and applications, using popular tools such as Jenkins or Travis CI.

- Rendering: When it comes to rendering, there are numerous options for improving visual quality and performance. These include implementing advanced materials with features like anisotropy, subsurface scattering and clear-coat effects, effectively handling transparencies in the rendering pipeline and designing efficient culling algorithms and LOD systems to optimize scene rendering. Additionally, implementing global illumination systems based on light probes, light-mapping or ray-tracing can enhance the realism of lighting and shadows in the scene, creating more immersive and visually appealing environments.

- Object Model: Regarding the object model, there is still substantial work required to achieve full integration of the ECS system with the Job System, enabling automatic scheduling of jobs based on the data accesses of systems. Additionally, numerous improvements can be made to enhance the existing system, such as implementing a better chunk-based memory management scheme for components, a more complete component querying system, an entity relationship model that makes it possible to describe entity graphs natively, and more.

- Resource Management: Even though we implement the foundation of a resource management system that allows asset preprocessing, we only integrate a few basic assets and we do little to no pre-processing in some cases. This is also an area that could be vastly improved, implementing better processing for textures and meshes. With the integration of RTTI systems, we could create scene assets, prefabs, and more. Our implementation is also greatly limited by being blocking, asynchronous asset loading is a must in any game that is not based in small, closed *"levels/zones"*.

- Tooling: Even though we developed a basic asset conversion tool, we mostly focused our development resources in creating the engine runtime. The tooling side of the engine could be vastly improved, integrating systems such as in-game debug UI's that could leverage libraries such ass Dear ImGui, developing a separated, out of process, level editor that leverages frameworks such as QT or WPF. Implementing an asset server that handles conversion requests from different clients, etc.

- Testing: Even though we integrate an automatic testing framework using *Google Test*, multiple benchmarks for the object model, and a prototype framework to create manual rendering tests, there is still much work to do. Improving and expanding the rendering test framework, comparing our real-time methods to path-traced results and developing tests for some of the engine systems that handle resource processing, input management, etc.

- State of the Art: Our game engine is constructed as a comprehensive collection of libraries that can be easily interchanged as needed. This flexibility enables us to utilize it as a solid foundation for developing and testing new systems and showcasing tech demos. One intriguing possibility is the integration of cutting-edge systems such as neural materials (§1.3) or NeRFs (§1.3). Additionally, we can design an asset pipeline that leverages neural networks as compression tools or implement experimental object models to significantly reduce development time in the long run. These various avenues present opportunities to integrate and enhance the functionality of our engine.

# Papers & Books


[7] Frederick O Bartell, Eustace L Dereniak, and William L Wolfe. "The theory and measurement of bidirectional reflectance distribution function (BRDF) and bidirectional transmittance distribution function (BTDF)". In: *Radiation scattering in optical systems*. Vol. 257. SPIE. 1981, pp. 154–160.

[9] Benedikt Bitterli et al. "Spatiotemporal reservoir resampling for real-time ray tracing with dynamic direct lighting". In: *ACM Transactions on Graphics (Proceedings of SIGGRAPH)* 39.4 (July 2020). DOI: 10/gg8xc7.

[12] Robert L Cook and Kenneth E. Torrance. "A reflectance model for computer graphics". In: *ACM Transactions on Graphics (ToG)* 1.1 (1982), pp. 7–24.

[13] Michael Deering et al. "The Triangle Processor and Normal Vector Shader: A VLSI System for High Performance Graphics". In: *Proceedings of the 15th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '88. New York, NY, USA: Association for Computing Machinery, 1988, pp. 21–30. ISBN: 0897912756. DOI: 10.1145/54852.378468. URL: https://doi.org/10.1145/54852.378468.

[18] Richard Fabian. *Data-Oriented Design*. Accessed: 2023-07-11. 2018. URL: https://www.dataorienteddesign.com/dodbook/.

[23] Sara Fridovich-Keil et al. "Plenoxels: Radiance fields without neural networks". In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2022, pp. 5501–5510.

[28] Jason Gregory. *Game engine architecture*. crc Press, 2018.

[29] Takahiro Harada, Jay McKee, and Jason C Yang. "Forward+: Bringing Deferred Lighting to the Next Level". In: (2012). URL: https://takahiroharada.files.wordpress.com/2015/04/forward_plus.pdf.

[36] Ben Mildenhall et al. "Nerf: Representing scenes as neural radiance fields for view synthesis". In: *Communications of the ACM* 65.1 (2021), pp. 99–106.

[38] Ola Olsson, Markus Billeter, and Ulf Assarsson. "Clustered Deferred and Forward Shading". In: (2012). URL: https://diglib.eg.org/xmlui/bitstream/handle/10.2312/EGGH.HPG12.087-096/087-096.pdf?sequence=1&isAllowed=y.

[39] Michael Oren and Shree K Nayar. "Generalization of Lambert's reflectance model". In: *Proceedings of the 21st annual conference on Computer graphics and interactive techniques*. 1994, pp. 239–246.

[41] Bui Tuong Phong. "Illumination for computer generated pictures". In: *Seminal graphics: pioneering efforts that shaped the field*. 1998, pp. 95–101.

[42] Ravi Ramamoorthi and Pat Hanrahan. "On the relationship between radiance and irradiance: determining the illumination from images of a convex Lambertian object". In: *JOSA A* 18.10 (2001), pp. 2448–2459.

[44] Peter-Pike Sloan, Jan Kautz, and John Snyder. "Precomputed Radiance Transfer for Real-Time Rendering in Dynamic, Low-Frequency Lighting Environments". In: (2005). Accessed: 2023-07-11. URL: https://jankautz.com/publications/prtSIG02.pdf.

[45] Peter-Pike Sloan, Jan Kautz, and John Snyder. "Precomputed radiance transfer for real-time rendering in dynamic, low-frequency lighting environments". In: *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*. 2002, pp. 527–536.

[46] Towaki Takikawa et al. “Variable bitrate neural fields”. In: *ACM SIGGRAPH 2022 Conference Proceedings*. 2022, pp. 1–9.

[60] Tizian Zeltner et al. “Real-Time Neural Appearance Models”. In: *arXiv preprint arXiv:2305.02678* (2023).

# Talks & Presentations


[3] Mike Acton. *CppCon 2014: Data-Oriented Design and C++*. Accessed: 2023-07-11. 2014. URL: https://www.youtube.com/watch?v=rX0ItVEVjHc.

[4] Mike Acton. *Solving the Right Problems for Engine Programmers*. Accessed: 2023-07-11. 2020. URL: https://www.youtube.com/watch?v=4B00hV3wmMY.

[5] Bobby Anguelov. *Exoterica Engine: Game Engine Entity/Object Models*. Accessed: 2023-07-11. 2020. URL: https://www.youtube.com/watch?v=jjEsB611kxs&.

[6] Electronic Arts. *SIGGRAPH 21: Global Illumination Based on Surfels*. Accessed: 2023-07-11. URL: https://www.ea.com/seed/news/siggraph21-global-illumination-surfels.

[10] Sébastian Lagarde Charles de Rousiers. *SIGGRAPH 2014: Moving Frostbite to PBR*. Accessed: 2023-07-11. 2014. URL: https://www.ea.com/frostbite/news/moving-frostbite-to-pb.

[20] Nicolas Fleury. *CppCon 2014: C++ in Huge AAA Games*. Accessed: 2023-07-11. URL: https://www.youtube.com/watch?v=qYN6eduU06s&.

[21] Timothy Ford. *GDC 2017: Overwatch Gameplay Architecture and Netcode*. Accessed: 2023-07-11. 2017. URL: https://www.youtube.com/watch?v=zrIY0eIyqmI&.

[22] Andreas Fredriksson. *Context is Everything*. Accessed: 2023-07-11. URL: https://vimeo.com/644068002.

[24] Arvid Gerstmann. *CppCon 2018: Building a C++ Reflection System in One Weekend Using Clang and LLVM*. Accessed: 2023-07-11. URL: https://www.youtube.com/watch?v=DUiUBt-fqEY.

[30] Jon Holmes. *Unreal Fest Europe 2019: Aggregating Ticks to Manage Scale in Sea of Thieves*. Accessed: 2023-07-11. 2019. URL: https://www.youtube.com/watch?v=CBP5bpwkO54&.

[31] Matthew Ireland. *Cambridge Computer Science Talks: Forward and Deferred Rendering*. Accessed: 2023-07-11. 2021. URL: https://youtu.be/n5OiqJP2f7w.

[32] Jorge Jimenez. *Advanced Post-Processing in Call of Duty Advanced Warfare*. Accessed: 2023-07-11. URL: http://www.iryoku.com/next-generation-post-processing-in-call-of-duty-advanced-warfare.

[35] Casey Muratori Mike Acton. *HandmadeCon 2015: Mike Acton*. Accessed: 2023-07-11. 2015. URL: https://www.youtube.com/watch?v=qWJpI2adCcs.

[37] Yuriy O'Donnell. *FrameGraph: Extensible Rendering Architecture in Frostbite*. Accessed: 2023-07-11. URL: https://www.gdcvault.com/play/1024612/FrameGraph-Extensible-Rendering-Architecture-in.

[43] Dan Reed. *GDC 2017: Networking Scripted Weapons and Abilities in Overwatch*. Accessed: 2023-07-11. 2017. URL: https://www.youtube.com/watch?v=5jP0z7Atww4&.

[50] Anjana Vakil. *GOTO 2017: Programming Across Paradigms*. Accessed: 2023-07-11. 2017. URL: https://www.youtube.com/watch?v=Pg3UeB-5FdA&.

[59] Michael Wong. *CppCon 2020: Modern Software Needs Embedded Modern C++ Programming*. Accessed: 2023-07-11. URL: https://youtu.be/885TI3jnB7g.

# Miscelaneous


[1] Unreal Engine 5. *Lumen Documentation*. Accessed: 2023-07-11. URL: https://docs.unrealengine.com/5.0/en-US/lumen-global-illumination-and-reflections-in-unreal-engine/.

[2] Unreal Engine 5. *Nanite Documentation*. Accessed: 2023-07-11. URL: https://docs.unrealengine.com/5.0/en-US/nanite-virtualized-geometry-in-unreal-engine/.

[8] BGFX. *Documentation*. Accessed: 2023-07-11. URL: https://bkaradzic.github.io/bgfx/overview.html#what-is-it.

[11] Graphics Compendium. *Chapter 47: PBR, Cook-Torrance Reflectance Model*. Accessed: 2023-07-11. URL: https://graphicscompendium.com/gamedev/15-pbr.

[14] NVIDIA Developer Docs. *Engaging Voyage Vulkan*. Accessed: 2023-07-11. URL: https://developer.nvidia.com/engaging-voyage-vulkan.

[15] *EASTL: EA Standard Template Library*. Accessed: 2023-07-11. URL: https://github.com/electronicarts/EASTL.

[16] Unreal Engine. *Unreal's Render Dependency Graph (RDG)*. Accessed: 2023-07-11. URL: https://docs.unrealengine.com/5.0/en-US/render-dependency-graph-in-unreal-engine/.

[17] Wicked Engine. *Render API Abstraction*. Accessed: 2023-07-11. URL: https://wickedengine.net/2021/05/06/graphics-api-abstraction/.

[19] Filament. *Material Model*. Accessed: 2023-07-11. URL: https://google.github.io/filament/Filament.html#materialsystem/standardmodel.

[25] Github. *BGFX*. Accessed: 2023-07-11. URL: https://github.com/bkaradzic/bgfx.

[26] Github. *Wicked Engine*. Accessed: 2023-07-11. URL: https://github.com/turanszkij/WickedEngine.

[27] *Godot*. URL: https://godotengine.org/features/.

[33] LearnOpenGL. *Deferred Shading*. Accessed: 2023-07-11. URL: https://learnopengl.com/Advanced-Lighting/Deferred-Shading.

[34] LearnOpenGL. *Physically-Based Rendering*. Accessed: 2023-07-11. URL: https://learnopengl.com/PBR/Theory.

[40] Patapom. *Notes on Spherical Harmonics*. Accessed: 2023-07-11. URL: https://patapom.com/blog/SHPortal/.

[47] *Unity Ads*. URL: https://unity.com/products/unity-ads.

[48] *Unity Engine Features*. URL: https://www.unrealengine.com/en-US/features.

[49] *Unity Wikipedia Article*. URL: https://en.wikipedia.org/wiki/Unity_(game_engine).

[51] VulkanTutorial. *Instance Management*. Accessed: 2023-07-11. URL: https://vulkan-tutorial.com/Drawing_a_triangle/Setup/Instance.

[52] Wikipedia. *Bloom Effect*. Accessed: 2023-07-11. URL: https://en.wikipedia.org/wiki/Bloom_(shader_effect).

[53] Wikipedia. *Deferred Shading*. Accessed: 2023-07-11. URL: https://en.wikipedia.org/wiki/Deferred_shading.

[54] Wikipedia. *Lambertian Reflectance*. Accessed: 2023-07-11. URL: https://en.wikipedia.org/wiki/Lambertian_reflectance.

[55] Wikipedia. *Oren-Nayar Reflectance*. Accessed: 2023-07-11. URL: https://en.wikipedia.org/wiki/Oren%E2%80%93Nayar_reflectance_model.

[56] Wikipedia. *Phong Reflection*. Accessed: 2023-07-11. URL: https://en.wikipedia.org/wiki/Blinn%E2%80%93Phong_reflection_model.

[57] Wikipedia. *Phong Shading*. Accessed: 2023-07-11. URL: https://en.wikipedia.org/wiki/Phong_shading.

[58] Wikipedia. *Spherical Harmonics*. Accessed: 2023-07-11. URL: https://en.wikipedia.org/wiki/Spherical_harmonics.

# Appendix A

# Rendering Tests

This appendix provides high resolution figures of all of the rendering tests we realized.

Figure A.1: CookieKat: White Sphere, 0.5 Roughness, 0 Metallic

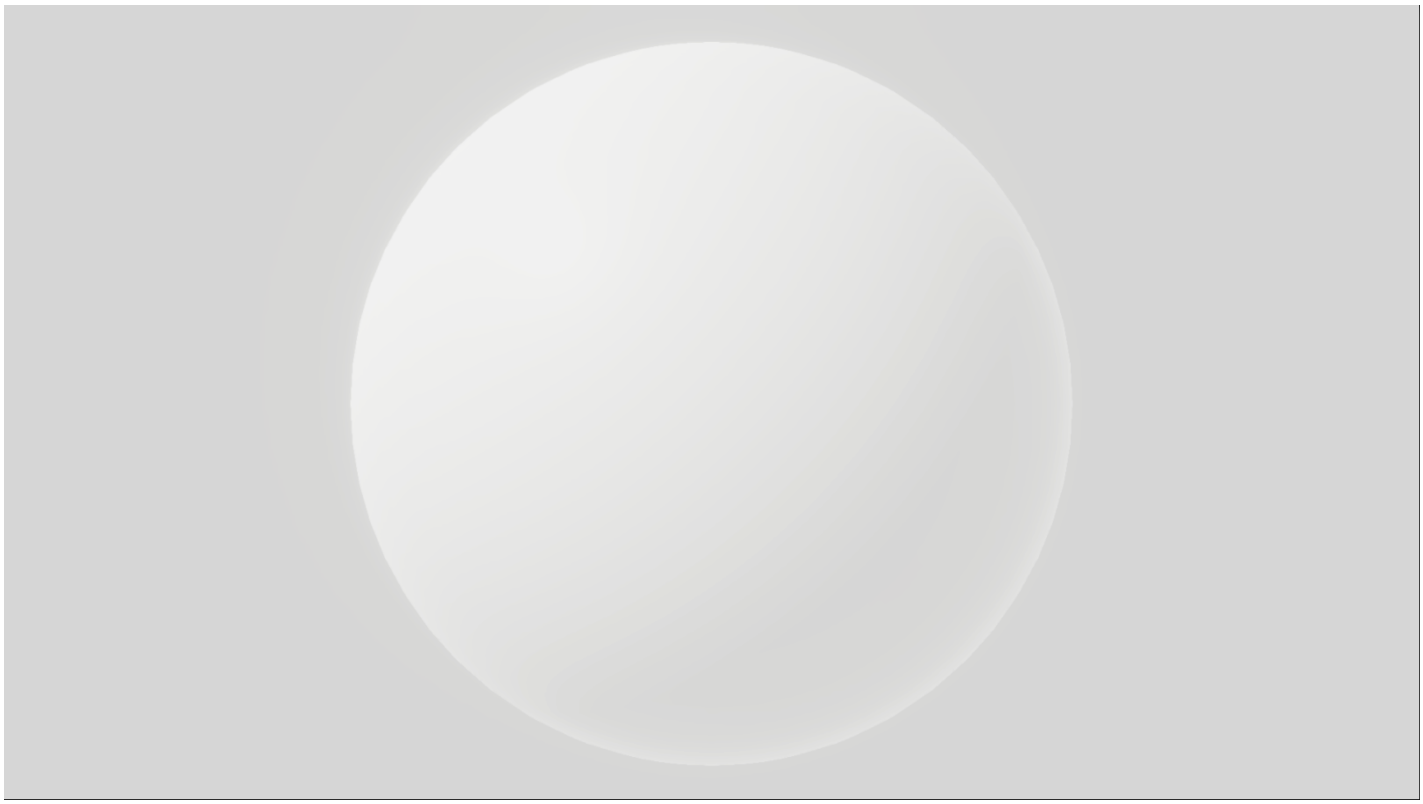


Figure A.2: Unity: White Sphere, 0.5 Roughness, 0 Metallic

Figure A.3: CookieKat (Adjusted): White Sphere, 0.5 Roughness, 0 Metallic

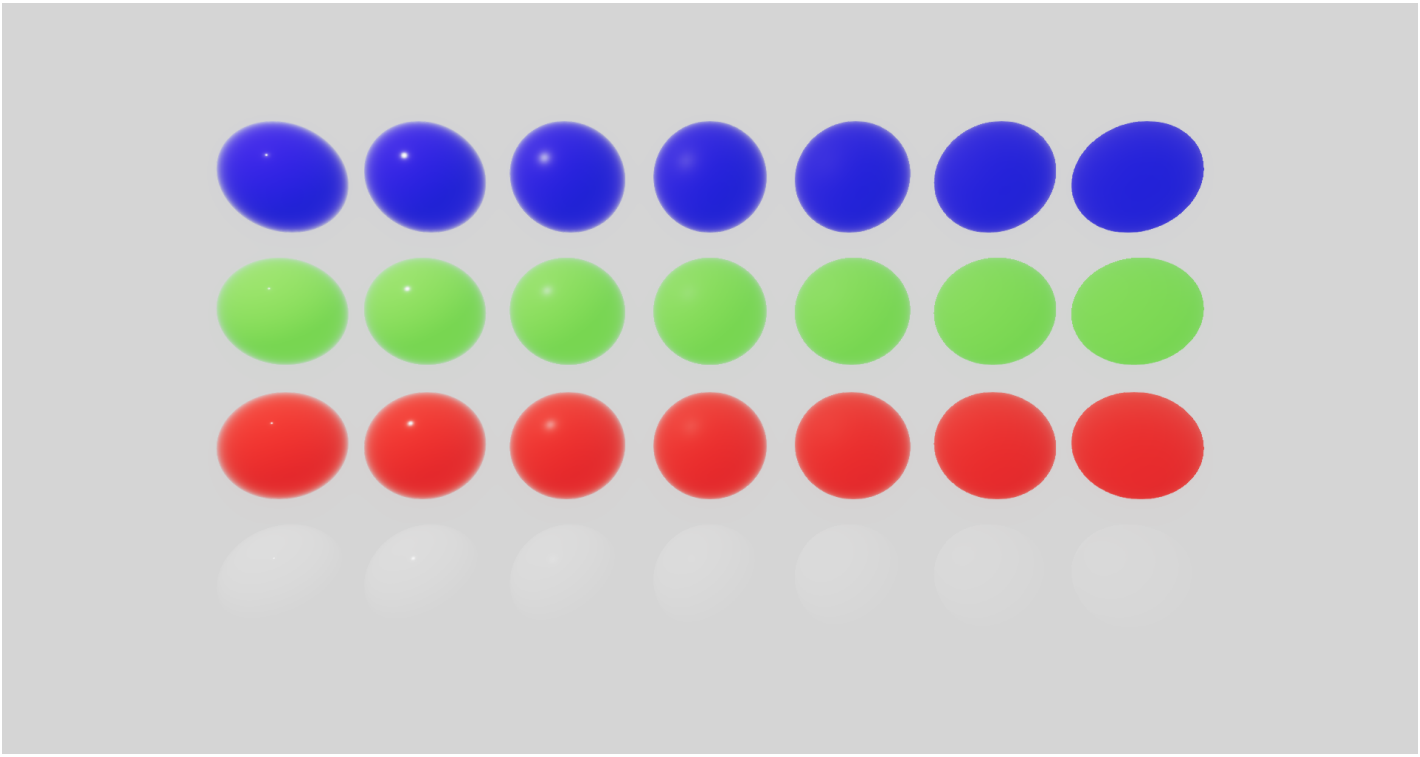


Figure A.4: CookieKat: 0.05 to 1 Roughness from left to right, 0 Metallic

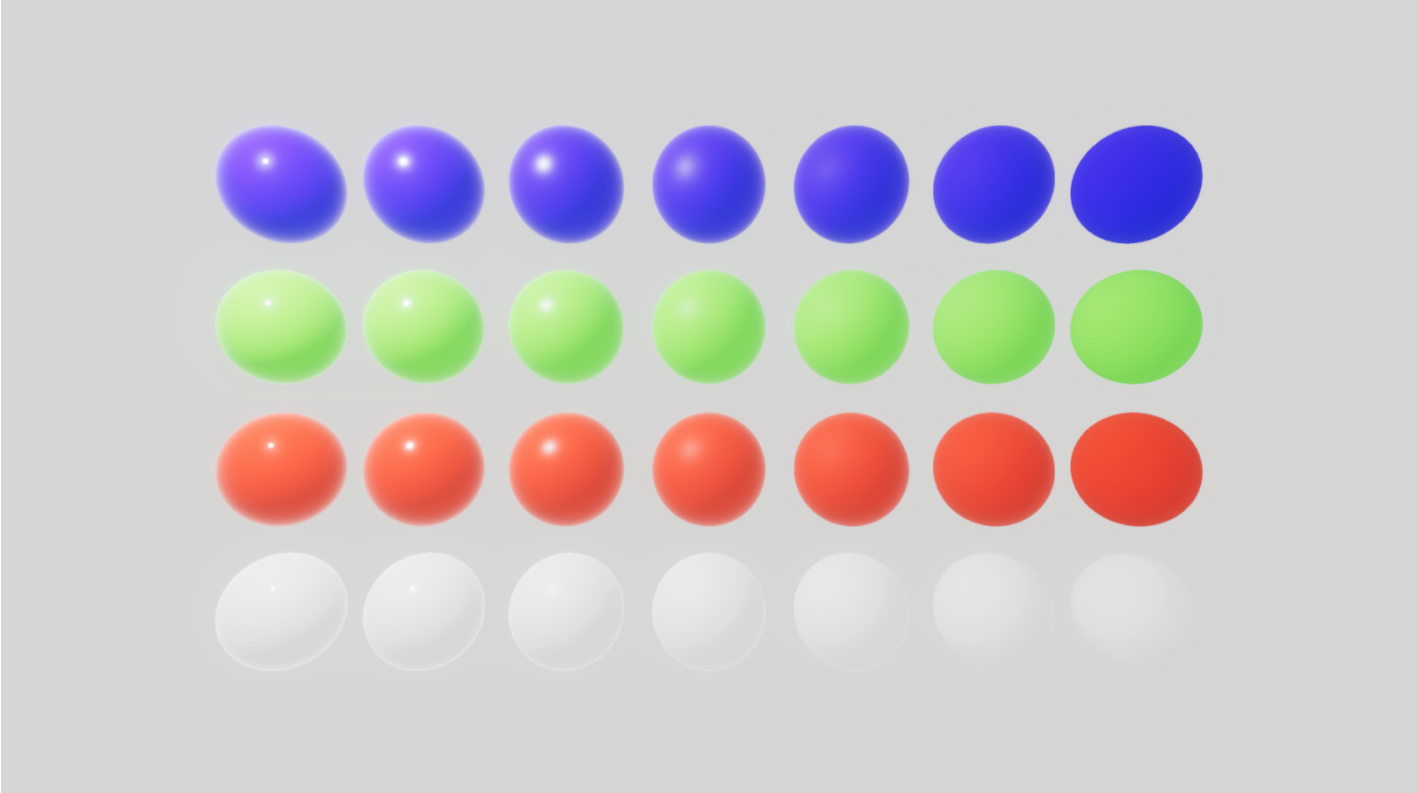


Figure A.5: Unity: 0.05 to 1 Roughness from left to right, 0 Metallic

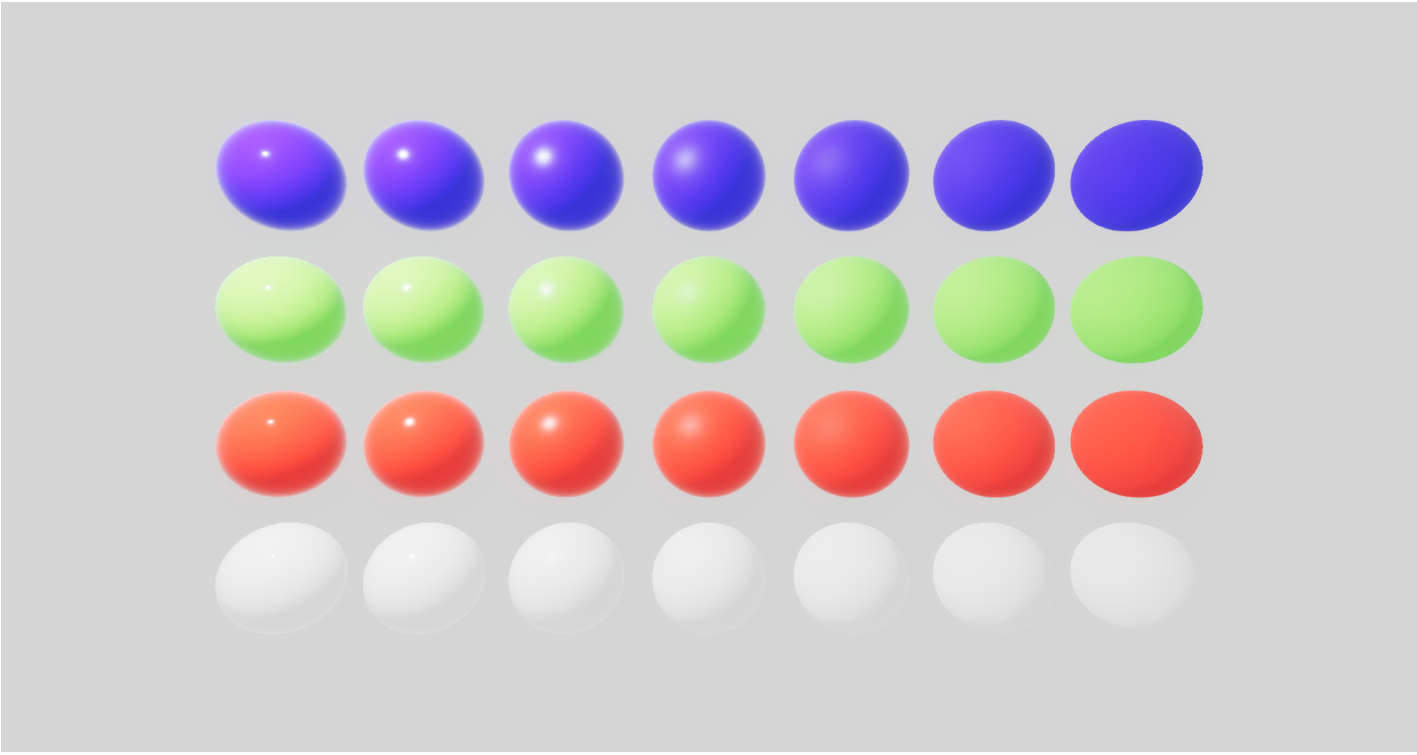


Figure A.6: CookieKat (Adjusted): 0.05 to 1 Roughness from left to right, 0 Metallic

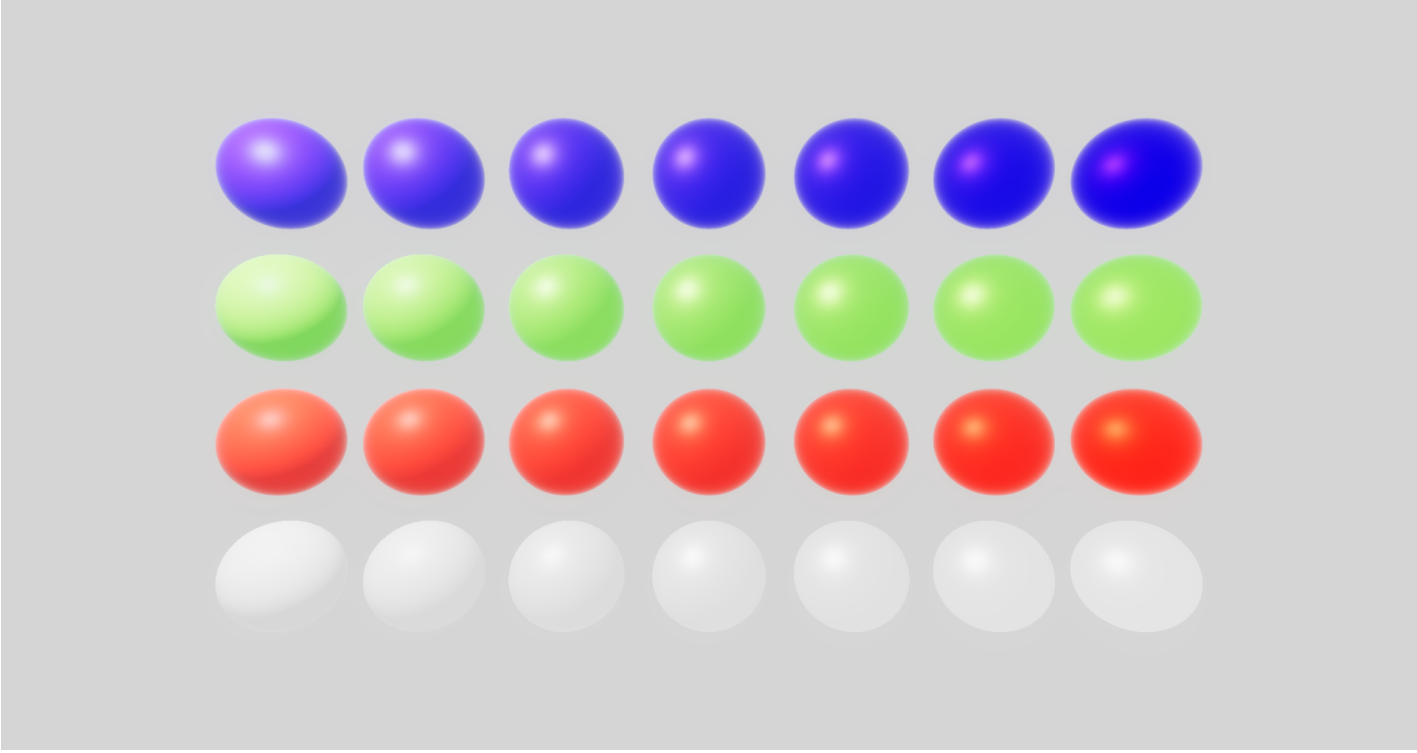


Figure A.7: CookieKat (Adjusted): 0.5 Roughness, 0 to 1 Metallic from left to right

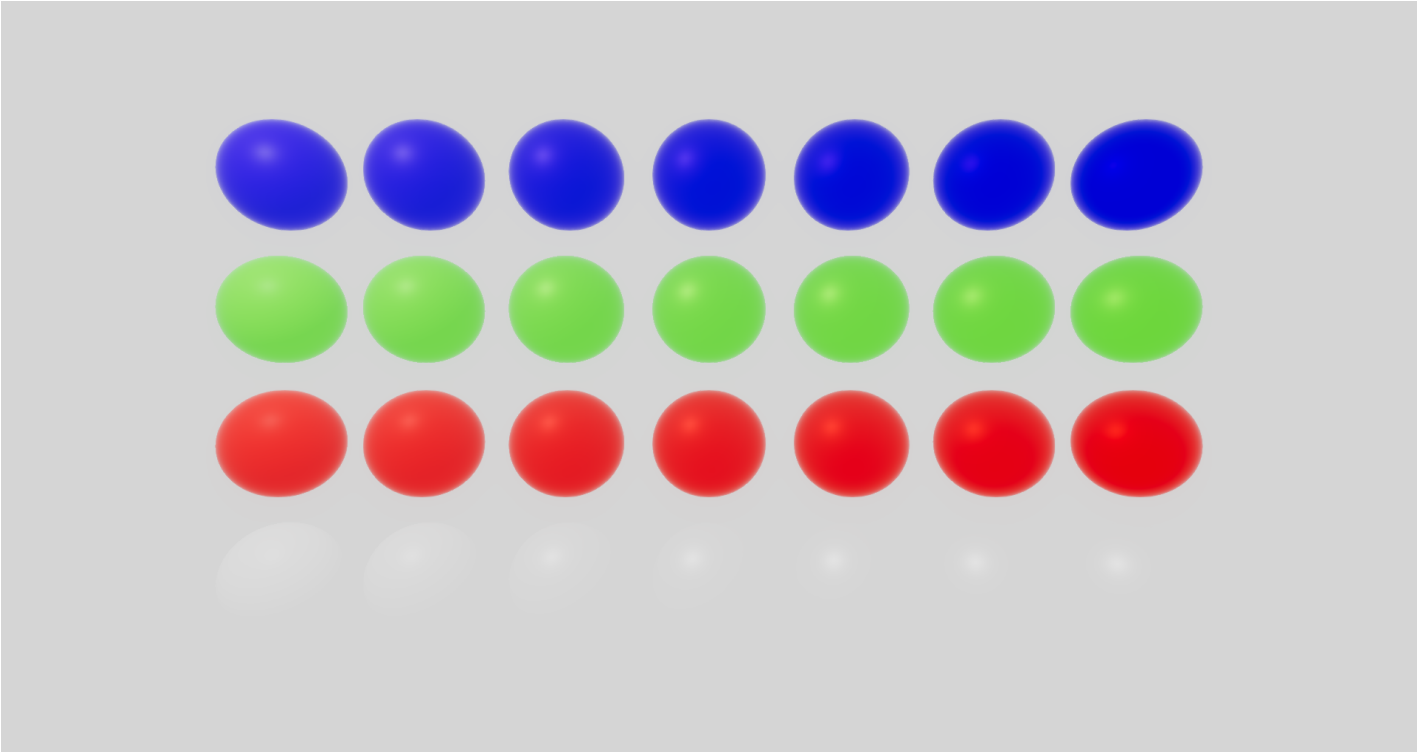


Figure A.8: CookieKat: 0.5 Roughness, 0 to 1 Metallic from left to right

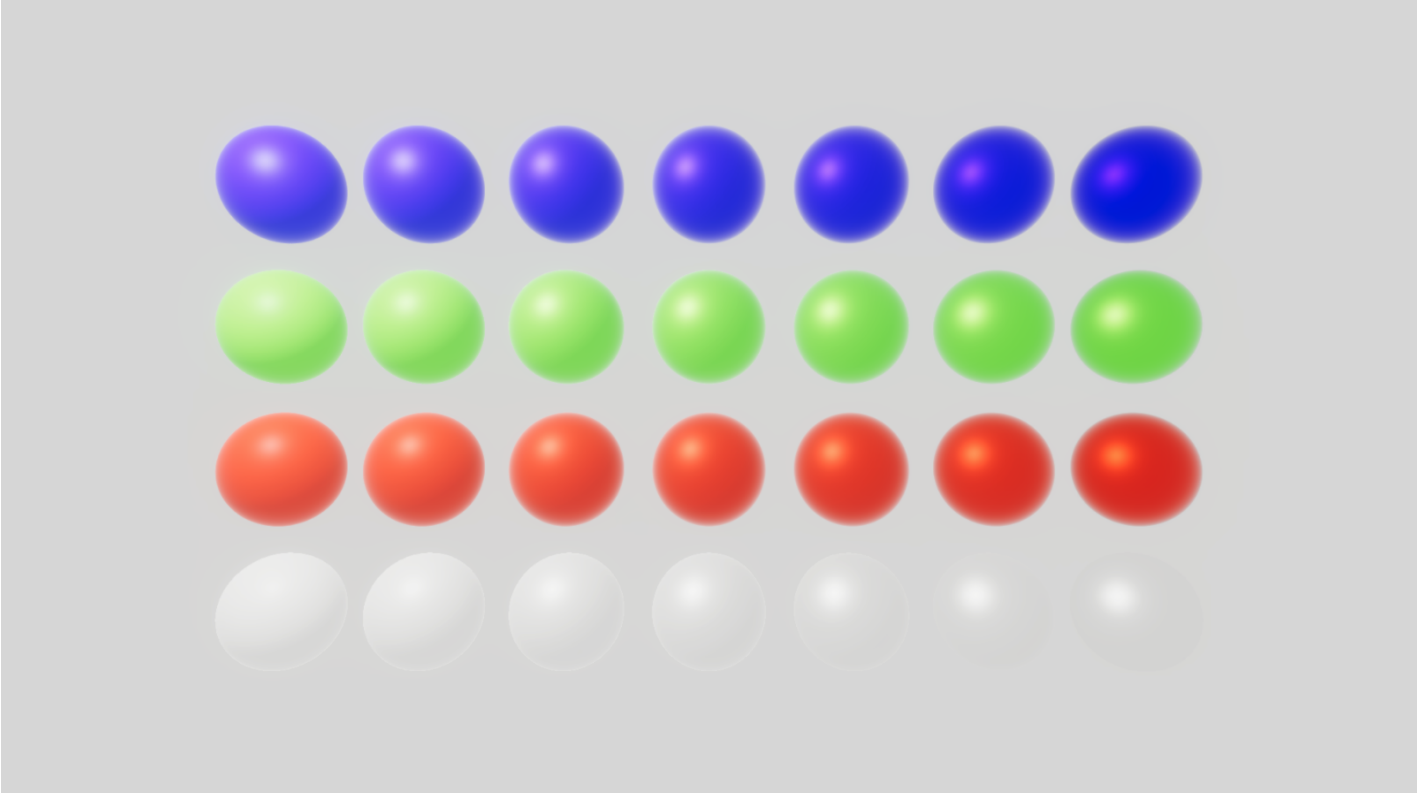


Figure A.9: Unity: 0.5 Roughness, 0 to 1 Metallic from left to right

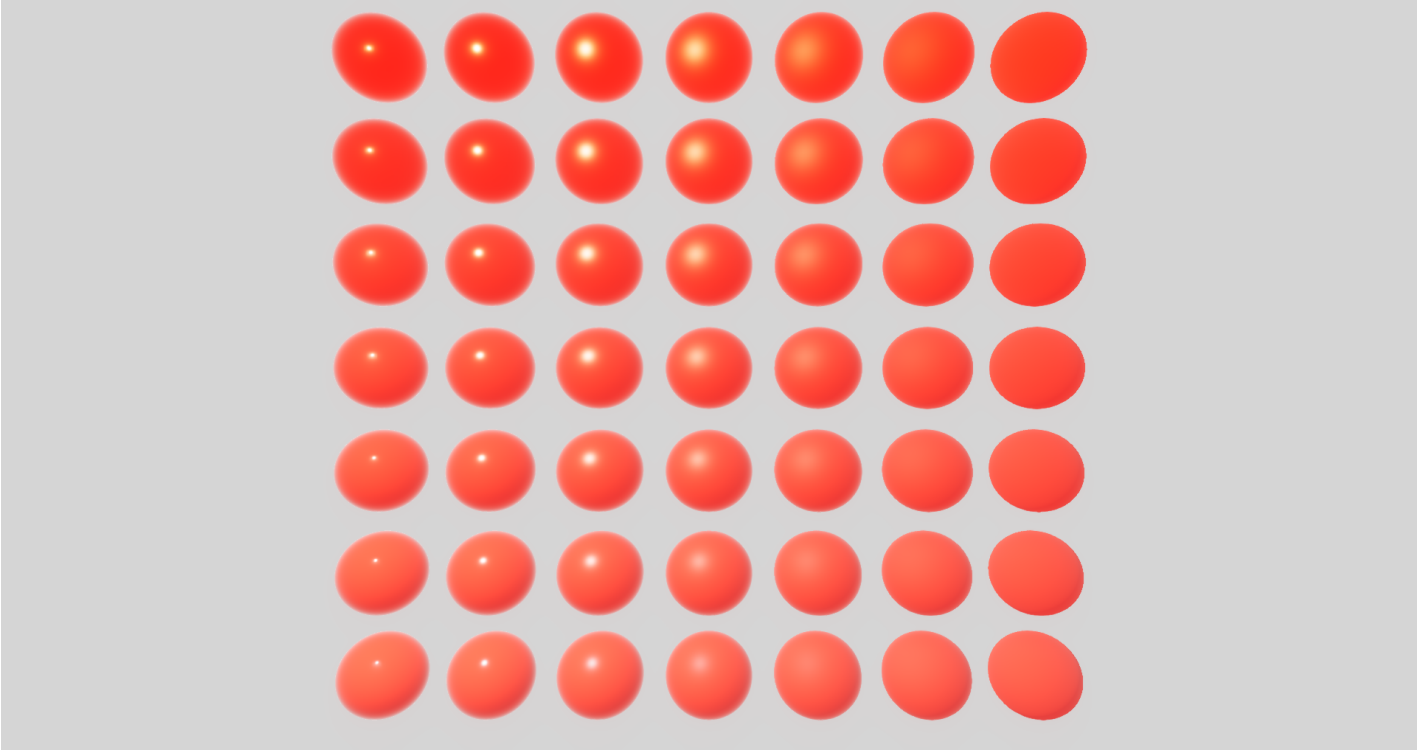


Figure A.10: CookieKat (Adjusted): 0.05 to 1 Roughness (X- to X+), 0 to 1 Metallic (Y- to Y+)

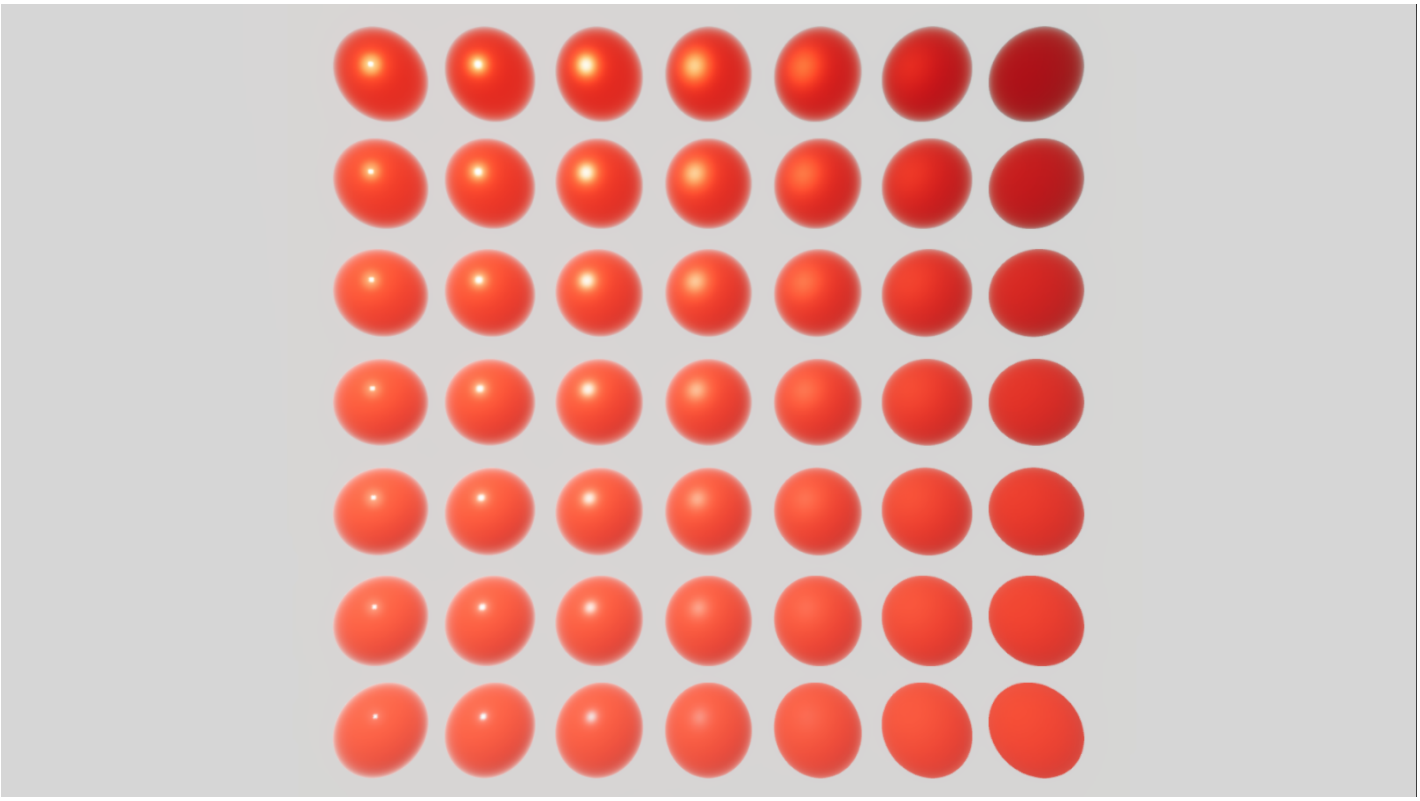
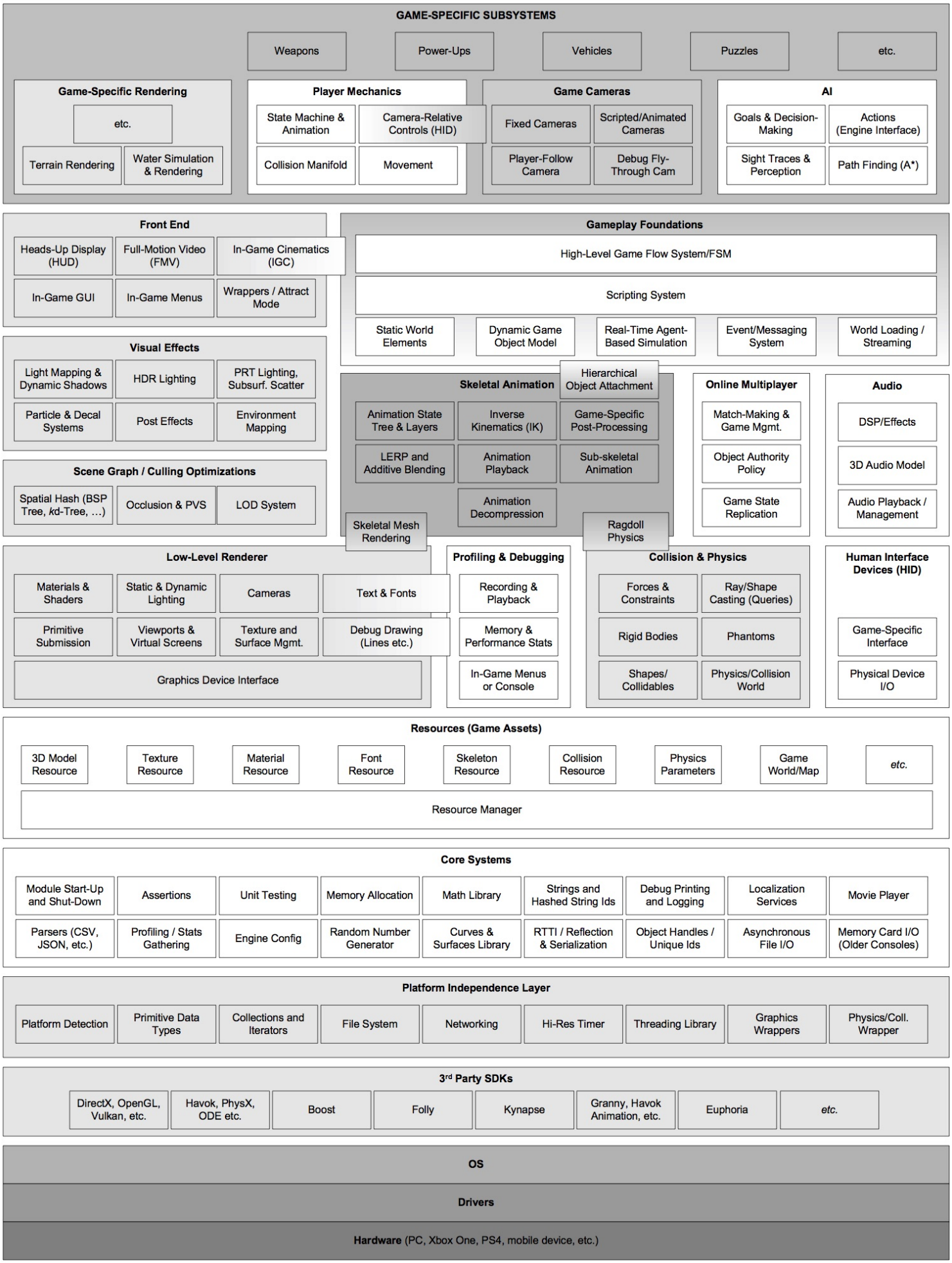


Figure A.11: Unity: 0.05 to 1 Roughness (X- to X+), 0 to 1 Metallic (Y- to Y+)

Figure A.12: CookieKat (Adjusted): Cerberus textured PBR model

Figure A.13: Unity: Cerberus textured PBR model


GAME-SPECIFIC SUBSYSTEMS
Weapons
Power-Ups
Vehicles
Puzzles
etc.
Game-Specific Rendering
etc.
Terrain Rendering
Water Simulation & Rendering
Player Mechanics
State Machine & Animation
Camera-Relative Controls (HID)
Collision Manifold
Movement
Game Cameras
Fixed Cameras
Scripted/Animated Cameras
Player-Follow Camera
Debug Fly-Through Cam
AI
Goals & Decision-Making
Actions (Engine Interface)
Sight Traces & Perception
Path Finding (A*)
Front End
Heads-Up Display (HUD)
Full-Motion Video (FMV)
In-Game Cinematics (IGC)
In-Game GUI
In-Game Menus
Wrappers / Attract Mode
Gameplay Foundations
High-Level Game Flow System/FSM
Scripting System
Static World Elements
Dynamic Game Object Model
Real-Time Agent-Based Simulation
Event/Messaging System
World Loading / Streaming
Visual Effects
Light Mapping & Dynamic Shadows
HDR Lighting
PRT Lighting, Subsurf. Scatter
Particle & Decal Systems
Post Effects
Environment Mapping
Skeletal Animation
Hierarchical Object Attachment
Animation State Tree & Layers
Inverse Kinematics (IK)
Game-Specific Post-Processing
LERP and Additive Blending
Animation Playback
Sub-skeletal Animation
Animation Decompression
Skeletal Mesh Rendering
Ragdoll Physics
Online Multiplayer
Match-Making & Game Mgmt.
Object Authority Policy
Game State Replication
Audio
DSP/Effects
3D Audio Model
Audio Playback / Management
Scene Graph / Culling Optimizations
Spatial Hash (BSP Tree, kd-Tree, …)
Occlusion & PVS
LOD System
Low-Level Renderer
Materials & Shaders
Static & Dynamic Lighting
Cameras
Text & Fonts
Primitive Submission
Viewports & Virtual Screens
Texture and Surface Mgmt.
Debug Drawing (Lines etc.)
Graphics Device Interface
Profiling & Debugging
Recording & Playback
Memory & Performance Stats
In-Game Menus or Console
Collision & Physics
Forces & Constraints
Ray/Shape Casting (Queries)
Rigid Bodies
Phantoms
Shapes/ Collidables
Physics/Collision World
Human Interface Devices (HID)
Game-Specific Interface
Physical Device I/O
Resources (Game Assets)
3D Model Resource
Texture Resource
Material Resource
Font Resource
Skeleton Resource
Collision Resource
Physics Parameters
Game World/Map
etc.
Resource Manager
Core Systems
Module Start-Up and Shut-Down
Assertions
Unit Testing
Memory Allocation
Math Library
Strings and Hashed String Ids
Debug Printing and Logging
Localization Services
Movie Player
Parsers (CSV, JSON, etc.)
Profiling / Stats Gathering
Engine Config
Random Number Generator
Curves & Surfaces Library
RTTI / Reflection & Serialization
Object Handles / Unique Ids
Asynchronous File I/O
Memory Card I/O (Older Consoles)
Platform Independence Layer
Platform Detection
Primitive Data Types
Collections and Iterators
File System
Networking
Hi-Res Timer
Threading Library
Graphics Wrappers
Physics/Coll. Wrapper
3rd Party SDKs
DirectX, OpenGL, Vulkan, etc.
Havok, PhysX, ODE etc.
Boost
Folly
Kynapse
Granny, Havok Animation, etc.
Euphoria
etc.
OS
Drivers
Hardware (PC, Xbox One, PS4, mobile device, etc.)