GAME ENGINE ARCHITECTURE DEVELOPMENT FOR REALTIME ENGINEERING SIMULATIONS

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

BURAK EREN DERE

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
COMPUTER ENGINEERING

JANUARY 2023

Approval of the thesis:

GAME ENGINE ARCHITECTURE DEVELOPMENT FOR REALTIME ENGINEERING SIMULATIONS

submitted by **BURAK EREN DERE** in partial fulfillment of the requirements for the degree of **Master of Science in Computer Engineering Department, Middle East Technical University** by,

Prof. Dr. Halil Kalıpçılar
Dean, Graduate School of **Natural and Applied Sciences** _____

Prof. Dr. Halit Oğuztüzün
Head of Department, **Computer Engineering** _____

Prof. Dr. Halit Oğuztüzün
Supervisor, **Computer Engineering, METU** _____

Examining Committee Members:

Prof. Dr. Ahmet Oğuz Akyüz
Computer Engineering, METU _____

Prof. Dr. Halit Oğuztüzün
Computer Engineering, METU _____

Prof. Dr. Tolga Kurtuluş Çapın
Computer Engineering, TED University _____

Date:19.01.2023

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Surname: Burak Eren Dere

Signature :

ABSTRACT

GAME ENGINE ARCHITECTURE DEVELOPMENT FOR REALTIME ENGINEERING SIMULATIONS

Dere, Burak Eren
M.S., Department of Computer Engineering
Supervisor: Prof. Dr. Halit Oğuztüzün

January 2023, 140 pages

Real-time simulations are one of the most important tools for engineers from many fields to analyze their models. In the industry, professionals are extensively using software which are capable of running real-time simulations. The existence of these software helps engineers to quickly implement their projects without considering what is under the hood so they can focus on their actual work.

One of the most promising tools for real-time simulations are Game Engines. They allow engineers to simulate physics with cutting-edge graphics and even implement the logic for the behavior of the objects in the scene via scripting. There are many famous or homemade game engines and even some of them are open source. However, most of them are huge general-purpose engines so their architectures have a lot of complicated subsystems. In this thesis, we propose a lightweight and scalable architecture for engines that aims to fulfill the needs for implementing real-time engineering simulations. We have built this architecture by implementing and integrating its subsystems from low level to high level. Firstly, we started by Windowing and Input systems, then the GUI. Secondly, have implemented the Entity Component System and the Scene System. Thirdly, we went into detail with the Rendering Sys-

tem, Resource, and Physics Manager. Lastly, we wrote a Scripting Manager on top of the architecture. After completing the architecture, we tested the integrity of the architecture with a case study.

Keywords: Real-time Simulation, Game Engine Architecture, Computer Graphics

ÖZ

GERÇEK ZAMANLI MÜHENDİSLİK SİMULASYONLARI İÇİN OYUN MOTORU MİMARİSİ GELİŞTİRME

Dere, Burak Eren
Yüksek Lisans, Bilgisayar Mühendisliği Bölümü
Tez Yöneticisi: Prof. Dr. Halit Oğuztüzün

Ocak 2023 , 140 sayfa

Gerçek zamanlı simülasyonlar, mühendislerin oluşturdukları modelleri inceleyebilmeleri için kullandıkları en önemli araçlardan biridir. Bunun için endüstride uzmanlar gerçek zamanlı simülasyonları çalıştırma yeteneğine sahip yazılımlar kullanmaktadır. Bu yazılımların varlığı mühendislere fazla detaya inmeden projelerini yapabilmelerini ve böylece kendi işlerine odaklanmalarını sağlar.

Gerçek zamanlı simülasyon araçları arasında en öne çıkanlardan biri Oyun Motorlarıdır. Oyun Motorları, mühendislere fiziksel olayları son teknoloji grafiklerle simüle edebilmelerini ve hatta sahnedeki objelerin davranışlarını betik programlamayla tanımlamalarına olanak tanır. Ünlü veya ünsüz birçok Oyun motoru mevcuttur ve bunlardan bazıları açık kaynaklıdır. Fakat, bu motorlardan birçoğu devasa genel amaçlıdır ve mimarileri çok fazla karmaşık alt sistem içermektedir. Bu tezde, gerçek zamanlı mühendislik simülasyonları yapmayı hedefleyen hafif ve ölçeklendirilebilir bir mimari önerilmektedir. Bu mimari aşağıdan yukarı seviyelere doğru alt sistemlerin geliştirilip entegre edilmesiyle oluşturulmuştur. İlk önce, Pencere ve Girdi sistemleri, sonra Kullanıcı Arayüzü geliştirilmesiyle başlanmıştır. Sonrasında, Varlık Bile-

şen Sistemi ve Sahne Sistemi oluşturulmuştur. Daha sonra, Render Sistemi, Fizik ve Kaynak Yöneticisi yazılmıştır. En sonunda, mimarinin en üst seviyesine Betik Programlama Yöneticisi eklenmiştir. Mimari tamamlandıktan sonra, bütünlüğü bir vaka çalışması ile test edilmiştir.

Anahtar Kelimeler: Gerçek Zamanlı Simülasyon, Oyun Motoru Mimarisi, Bilgisayar Grafikleri

To Kutay ECEVIT

ACKNOWLEDGMENTS

I and my friends grew up in an era when game and simulation technologies had just started thriving. So, building a game engine is my childhood dream. I was always eager to learn how these engines or real-time applications used to work under the hood. With this thesis work, despite the lack of practical knowledge of game engine architectures, we have managed to develop a simulation/game engine from scratch with an easily scalable architecture that can help students and engineers to learn or take references.

First of all, I would like to thank my thesis advisor Prof. Dr. Halit Oğuztüzün for his support. He showed me great respect and trusted me during my study. This thesis work is finished thanks to his guidance.

Secondly, I would like to express my gratitude to all brilliant content creators on the internet who share their wisdom with strangers without any outcome. I would specifically like to thank TheCherno, thebennybox, OGLDEV, and Joey de Vries. I would never have had a chance to learn any of these techniques I have used without you. Now I have a solid understanding which I can, and definitely will improve in the future. Thanks, guys!

I would also thank to my mother Mehtap DERE, my father Güven DERE, and my brother Emre DERE for their support and unconditional love.

Lastly, I want to mention one of my oldest true friends since the early 2000s. We lived in the same place, played basketball, rode bikes and went to internet cafes, and had a jolly good time together. His father was an officer and his mother was a teacher, just like mine. Our paths crossed in Cyprus and Diyarbakır, but we separated after I moved to Ankara for high school and he went to Çanakkale. Now, the only thing left from him is his wedding invitation which they were handing out with his fiancée on the day of the accident. I will never stop thinking about you Kutay. Thank you for everything, my brother.

TABLE OF CONTENTS

ABSTRACT v

ÖZ vii

ACKNOWLEDGMENTS x

TABLE OF CONTENTS xi

LIST OF TABLES xvii

LIST OF FIGURES xviii

LIST OF ABBREVIATIONS xxii

CHAPTERS

1 INTRODUCTION 1

1.1 Motivation 2

1.2 Contributions and Novelties 2

1.3 The Outline of the Thesis 3

2 BACKGROUND AND RELATED WORK 5

2.1 Background and Related Work 5

2.1.1 Overview of Engine Archictectures 6

2.1.2 OP Engine's Architecture 8

2.1.2.1 Interaction Between Subsystems During Runtime 9

2.1.2.2 OP Engine as a Static Library 10

2.1.2.3 OP Editor 10
2.1.2.4 Third Party Libraries 11
3 CORE SYSTEM COMPONENTS OF OP ENGINE 15
3.1 Application Class 16
3.1.1 Application Runtime 17
3.2 Entry Point 18
3.3 Layering System 19
3.4 Windowing System 20
3.4.1 Callbacks 21
3.5 Event System 23
3.5.1 Event Dispatcher 25
3.6 Input System 26
4 GRAPHICAL USER INTERFACE (GUI) OF OP ENGINE 29
4.0.1 GUI Layer in the Application 32
4.0.2 OnAttach 33
4.0.3 OnDetach 34
4.0.4 Begin and End 34
4.0.5 Icon Fonts 35
5 ENTITY COMPONENT SYSTEM OF OP ENGINE 37
5.1 OP Engine's ECS 41
5.1.1 OP Engine's Entity Class 42
5.1.2 OP Engine's Components 43
5.2 Scene Management 50

5.2.1 Callbacks for Nested Entities 51
5.2.2 Scene Utility Functions 53
6 OP ENGINE'S RENDERING SYSTEM 57
6.1 API Agnosticism 57
6.1.0.1 Abstracting API Commands 58
6.2 Structure of OP Rendering System 59
6.2.1 Buffers 60
6.2.1.1 OpenGL Vertex Buffer 60
6.2.1.2 OpenGL Index Buffer 61
6.2.2 Vertex Array 61
6.2.2.1 Vertex Buffer Layout for Vertex Array 62
6.2.3 Texture 65
6.2.4 Framebuffer 68
6.2.5 Uniform Buffer 71
6.2.6 Shader 73
6.2.6.1 Shader Compilation and Struture 74
6.2.6.2 Shader Include System 76
6.2.7 Overall Structure of Rendering System 78
6.3 Meshes and Geometry 79
6.3.1 Tangents and Bitangents 80
6.4 Material System 83
6.5 Renderpass System 84
6.6 Cameras and Light Sources 86

6.6.1 How Camera Data is Managed 86
6.6.2 How Light Source Data is Managed 87
6.7 Shadow Mapping 88
6.7.1 Light Space Matrices, Cascade Planes, and Depth Maps 90
6.7.1.1 Light Space Matrices and Cascade Planes 90
6.7.1.2 Depth Maps 92
6.7.1.3 Examples from OP Engine 92
6.8 Physically Based Rendering and Environment Lighting 94
6.8.1 Texture Maps 95
7 RESOURCE MANAGER OF OP ENGINE 99
7.1 Data 99
7.1.1 Resource Allocation 99
7.2 Initialization of Resource Manager 100
7.3 Shader Compilation 100
7.4 Loading Models 101
7.5 Generic Resources 102
7.5.1 Generic Meshes 102
7.5.2 Generic Textures 102
7.6 Environment Maps and Textures 103
7.6.1 Environment Maps 103
7.6.2 Textures 104
7.7 Materials 105
8 OP ENGINE'S PHYSICS SYSTEM 107

8.1 Initiating Physics Manager . . . 107
8.2 Starting Physics World . . . 108
8.3 Rigid Bodies . . . 109
8.3.1 Collider Shapes . . . 109
8.3.2 Adding Rigid Bodies . . . 111
8.3.3 Deleting Rigid Bodies . . . 112
8.4 Clearing Physics World . . . 112
9 OP ENGINE'S SCRIPTING SYSTEM . . . 113
9.1 Mono . . . 114
9.1.1 Embedding Mono into C++ . . . 114
9.2 Script Manager . . . 116
9.2.1 Initialize Mono . . . 116
9.2.2 Load Runtime Assembly . . . 116
9.2.3 Calling Script Functions . . . 117
9.3 OP Script . . . 118
9.3.1 Structure of OP Script . . . 119
9.4 A Script Example . . . 120
9.5 Reloading Scripts . . . 123
10 A CASE STUDY: SWARM ROBOTS . . . 125
10.1 Ground and Maze . . . 125
10.2 Swarm Robot . . . 127
10.3 Scene . . . 129
10.4 Boids Algorithm . . . 130

10.5 Scripts 132

10.6 Performance Test 133

11 CONCLUSION & FUTURE WORK 135

11.1 Conclusion 135

11.2 Future Work 135

REFERENCES 137

LIST OF TABLES

TABLES

Table 10.1 Performance metrics for case study 134

LIST OF FIGURES

FIGURES

Figure 2.1 Game Engine Reusability Gamut. From Game Engine Architecture (3rd ed., p. 12), by J. Gregory, 2018, K Peters/CRC Press. Copyright 2019 by Jason Gregory. Reprinted with permission 6

Figure 2.2 Runtime Game Engine Architecture. From Game Engine Architecture (3rd ed., p. 39), by J. Gregory, 2018, K Peters/CRC Press. Copyright 2019 by Jason Gregory. Reprinted with permission 7

Figure 2.3 OP Engine Architecture 8

Figure 2.4 OP Engine Runtime Sequence Diagram 9

Figure 2.5 OP Editor 10

Figure 2.6 Example figure from Spdlog 11

Figure 2.7 Position Gizmo 12

Figure 2.8 Orientation Gizmo 12

Figure 2.9 Scale Gizmo 12

Figure 3.1 Application Initialization 16

Figure 3.2 Layer Stack Structure 20

Figure 3.3 Event Flow 23

Figure 3.4 Event Class Diagram 24

Figure 3.5 Event Dispatcher Logic 26

Figure 4.1 OP Engine GUI 29
Figure 4.2 ImPlot Line Plot Demo 31
Figure 4.3 ImPlot with ImGui Tables Demo 31

Figure 5.1 Multiple Inheritance for a simple scene 38
Figure 5.2 Multiple Inheritance for a more complex scene 39
Figure 5.3 Non Tightly packed array 40
Figure 5.4 Tightly packed array 41
Figure 5.5 Outlined Capsule 43
Figure 5.6 An object between outlined capsule and camera 44
Figure 5.7 Relationship Between Entities 45
Figure 5.8 Transform Component in GUI 45
Figure 5.9 Camera Component in GUI 46
Figure 5.10 Script Component with Null Script 46
Figure 5.11 Script Component with a Script 47
Figure 5.12 Directional Light Component 47
Figure 5.13 Spot Light Component 47
Figure 5.14 Point Light Component 48
Figure 5.15 Mesh Component 48
Figure 5.16 Material Component 49
Figure 5.17 Physics 3D Material Component 50
Figure 5.18 Physics 3D Collider Component 50
Figure 5.19 Rotating Pivot Point on X axis 52

Figure 5.20 Camera Pivoting Scene Graph 52

Figure 6.1 Vertex Buffer Layout 63

Figure 6.2 Flow of the Programmable Shader Stages 74

Figure 6.3 OP Engine's Shader Source Code Structure 77

Figure 6.4 Rendering System Structure 78

Figure 6.5 Tangent of a Surface 80

Figure 6.6 Primitive Tangent 81

Figure 6.7 Shadow Mapping Depth Test 89

Figure 6.8 Scene with only environment light 92

Figure 6.9 Scene with a point light 93

Figure 6.10 Scene with a spot light 93

Figure 6.11 Scene with a directional light 94

Figure 6.12 Metalness and roughness values 95

Figure 6.13 Texture Mapping of Roughness Albedo and AO 96

Figure 6.14 Texture Mapping of Metalness and Normal 96

Figure 6.15 Height Map added 97

Figure 8.1 Box Collider 109

Figure 8.2 Sphere Collider 110

Figure 8.3 Box Collider 110

Figure 8.4 Capsule Collider 111

Figure 9.1 Communication between C# and C++ 115

Figure 10.1 Ground and Maze 127

Figure 10.2 Robot body and its collider 128

Figure 10.3 Robot's Sensor's Collider 129

Figure 10.4 Complete Test Scene 130

LIST OF ABBREVIATIONS

OP	Over-Powered
ECS	Entity Component System
CSM	Cascaded Shadow Mapping
2D	2 Dimensional
3D	3 Dimensional
HDR	High Dynamic Range
PBR	Physically Based Rendering
GLSL	OpenGL Shading Language
ECS	Entity Component System

CHAPTER 1

INTRODUCTION

Real-time simulation software are one of the essential tools in the pockets of engineers in the industry. They need them in order to analyze and check the reliability of their models and scenarios inside their workflow. This way, they can fix the problems in their models before actually testing them with real hardware. Professionals in the industry often use modeling and simulation tools such as Matlab/Simulink[1] and OpenModelica[2] for these purposes. In addition, there are also simulation and computational analysis tools such as SU2[3] for specific simulation problems like fluid dynamics.

On the other hand, real-time simulations are also being used in creative industries like game development. This led to the birth of game engines. Game engines helped game devs to put their ideas to work as quickly as possible. Today, they are being used in most of the products in creative industries. Some studios or independent developers use game engines such as Unreal Engine[4] and Unity[5]. There are also some open-source engines such as Godot[6]. Furthermore, some programmers have developed their own engines for creating their games such as ETEngine[7] by Leah Lindner or the public branch of 2D HazelEngine[8] developed by Yan Chernikov.

With this thesis, we have developed the OP Engine[9] which supports agent-based real-time simulations with 3D Graphics and Physics. This is an open-source project that everyone can use freely. We are proposing a lightweight and scalable architecture. This means that OP Engine's capabilities are the ultimate basics for building real-time 3D simulations with rigid body dynamics. The integrity of our architecture has also been evaluated with a case study of swarm robots.

1.1 Motivation

Many of the tools that we can see online are huge general-purpose software with complex architectures. This makes it harder to extend their functionalities and add new algorithms or try new approaches. We also see that most of the real-time simulation software we can find are not open source and have some limiting licenses. As a result, researchers or engineers develop a dependency on these software and it affects their productivity.

Furthermore, for open-source and non-complex engines, we can say that most of their architectures have been designed for specific purposes. For example, the public version of Yan Chernikov's Hazel Engine is developed only for making 2D Games so its architecture utilizes a batch renderer and Box2D[10] as the physics library.

With OP Engine, we aimed to develop an engine that is capable of handling rigid body dynamics in the 3D world with modern rendering approaches, also giving developers the flexibility to handle objects in their scenarios by giving them different behaviors via scripting. Every low-level module of the architecture is easily extensible which helps developers to try out their own approaches if they want to. This way, we are giving a lightweight and open-source architecture that is practically implemented in OP Engine.

1.2 Contributions and Novelties

The main contributions of this thesis are as follows:

- In the field of real-time simulations and game engines, there was a lack of practical knowledge so with this thesis, we have combined theoretical knowledge with practical work and as a result, came up with a game/simulation engine that is capable of handling 3D physics/graphics and scripting
- OP Engine is an open source engine and it has a highly permissive license. So it can be used an manipulated without permission.
- OP Engine works as a testbed for real-time simulations. This is helpful for both

educational and professional purposes. Researchers and students can easily test their simulations and see the results visually and since the architecture is not complex, they can modify existing subsystems and create their own pipelines.

- In the architecture, external APIs and libraries are also abstracted at higher levels which means that developers can also change them according to their needs.

1.3 The Outline of the Thesis

The outline of the remaining parts of the thesis is as follows:

- Chapter 2 gives background information, related work, how the architecture of OP Engine is planned, and which external libraries have been used.
- Chapter 3 describes some of the core elements of OP Engine's architecture, which are the Entry point, Application Class, Layering, Windowing, Event, and Input systems.
- Chapter 4 explains how GUI is handled in OP Engine and OP Editor.
- Chapter 5 is about the scene and ECS(Entity Component System), their importance, and how they are used inside the engine architecture.
- In Chapter 6, the rendering part is explained. In this part:
  - Which graphics API is used and how rendering classes are organized to be API-Agnostic?
  - How mesh data is handled?
  - Material system. This part describes how shaders and materials are correlated.
  - How renderpass system is built for implementing graphics techniques easier and in a cleaner way?
  - How cameras and different light sources are handled and their data sent to the GPU?

  - How shadow mapping is implemented for point, spot, and directional lights? This part also explains how OP Engine handles CSM(Cascaded Shadow Mapping) as well.
  - How OP Engine manages image-based environment lighting and PBR rendering. In this part, texturing, and normal-parallax mapping are also explained.

- In Chapter 7, the way of handling resources such as shaders and textures with the Resource Manager is explained.
- Chapter 8 is about how 3D Physics is implemented using Bullet[11] library.
- In Chapter 9, the scripting system which gives developers a way to define the behaviors of objects is explained.
- In Chapter 10, there is a case study of multiple swarm robots in a scene with obstacles. This case study shows the real-time behavior of these robots and also tests the integrity of the proposed architecture. The performance of the scene is also measured using different numbers of robots in the scene.
- Finally, Chapter 11 is the conclusion. Future work is discussed in this chapter.

CHAPTER 2

BACKGROUND AND RELATED WORK

2.1 Background and Related Work

Modern game engines are actually libraries being used by some applications which are built on top of them. These applications might be games and simulations or tools to create them like level editors. They consist of multiple subsystems from low-level to high-level functionalities which work together in runtime.

In the past, game developers were rewriting or using low-level code from their previous games each time they were building a newer game. Therefore, this created a need for gathering most of the low-level core functionalities which are going to be needed for applications together and reusing them in an organized manner. According to the book Game Engine Architecture by Jason Gregory(2018),[12] who is the lead programmer of the well-known studio Naughty Dog which produced the game The Last of Us Part II, winning the EE Game of The Year Award(2021)[13], this reusing pattern which leads to modern game engines emerged with old games like Quake III Arena. In the book, he also states that sometimes, the difference between a game engine and a game might be tiny so the line between them becomes blurry. Hard-coded game rules and logic inside an engine turn this line even blurrier. Therefore, if an engine has a more general and data-driven architecture, it can be reused to build a large variety of different games or applications(Gregory, 2018). Jason Gregory describes this reusability issue with the following figure which shows that engine architectures decide the variety of products that are going to be created:

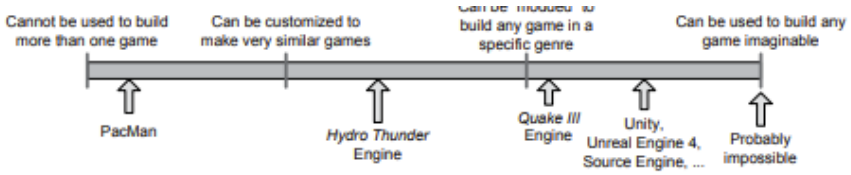


Figure 2.1: Game Engine Reusability Gamut. From Game Engine Architecture (3rd ed., p. 12), by J. Gregory, 2018, K Peters/CRC Press. Copyright 2019 by Jason Gregory. Reprinted with permission

Architectures of game engines like Unity[5] and Unreal Engine[4] have been built in such a way that any application related to real-time simulations (including games) can be created using their systems. But, as we stated in Chapter 8 their architectures are hard to reproduce or tweak to some extent. The reason behind this is these software are not open source. Even if they are open source like Godot Engine[6], it is still hard for developers who are trying to test new algorithms or run their simulations with special scenarios to fully integrate their ideas. Because the things they can do with these engines are limited to the capabilities of their architectures. Therefore, we have formalized an architecture, which can be extended with additional modules or completely overhauled by developers, giving them the flexibility to try their algorithms or scenarios. We have developed OP Engine[9] which handles rigid body simulations in real-time with 3D Graphics.

2.1.1 Overview of Engine Archictectures

As a whole, a modern real-time simulation engine consists of a great number of different subsystems working together in runtime. Since modern engines need to work in different environments and devices simultaneously, this adds extra layers to their architectures. For example, because of the differences in hardware for different devices, engines need to utilize systems that are capable of transforming data to be used in different devices. This can be explained with the endianness concept in computer science which is explained in this article[14]. As a result, this problem introduces extra subsystems which do the conversion to the data written with big-endian and little-endian architectures. In Game Engine Architecture book[15](2018), Jason Gregory

explains the bigger picture for game engine architecture with the following figure:

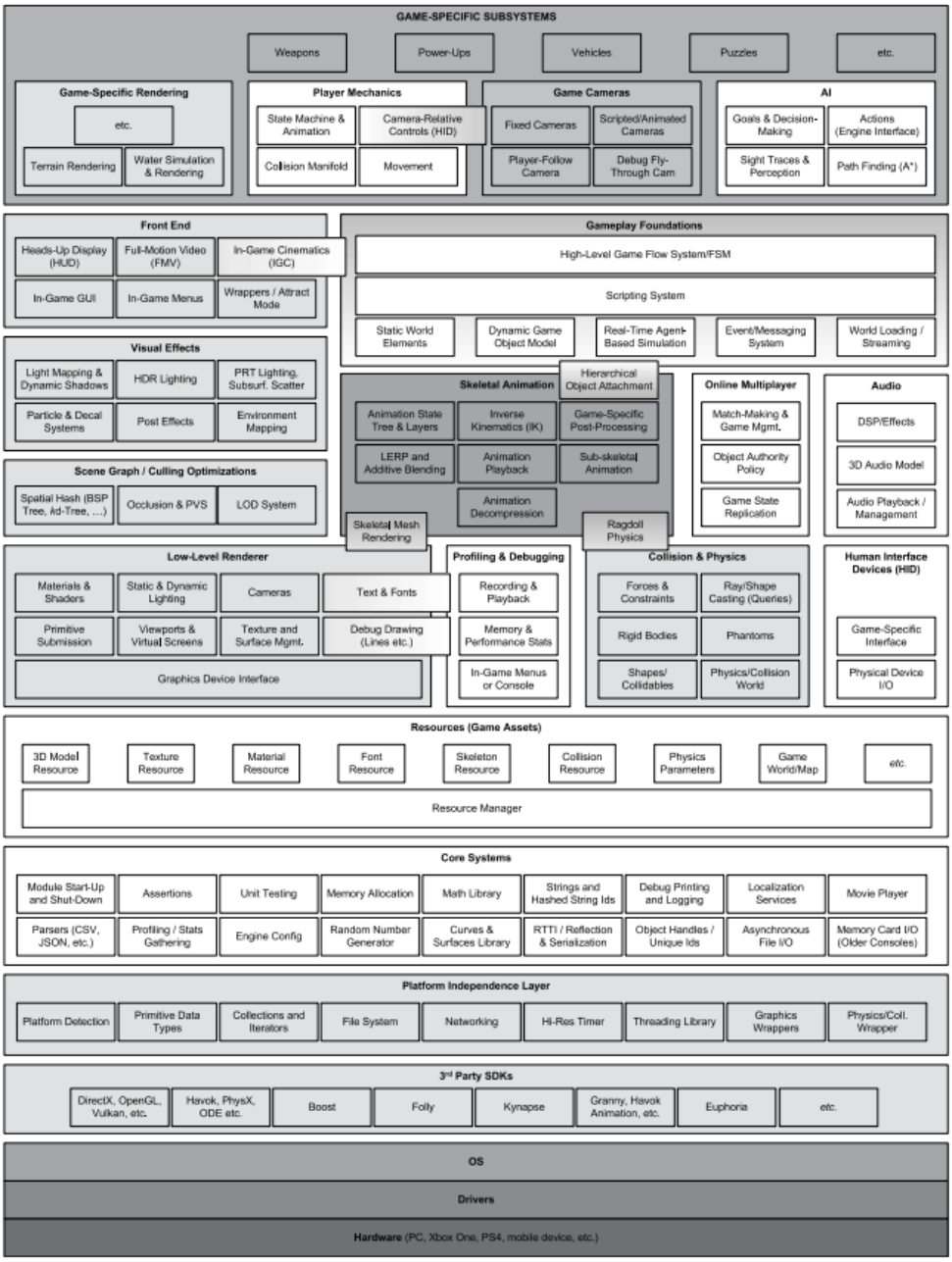


Figure 2.2: Runtime Game Engine Architecture. From Game Engine Architecture (3rd ed., p. 39), by J. Gregory, 2018, K Peters/CRC Press. Copyright 2019 by Jason Gregory. Reprinted with permission

2.1.2 OP Engine's Architecture

In OP Engine's architecture, we have taken the essential subsystems from Jason Gregory's architecture from Figure 2.2. It is mainly composed of low-level systems for handling windows, inputs, events, layer systems, and the entry point for applications using OP Engine. As we go to higher levels in the Architecture, there are systems for handling scripts, physics, and low-level rendering elements such as an abstracted Renderer API for integrating different graphics APIs such as vulkan[16] or metal[17]. After that, there are Renderer classes which are responsible for providing graphics functionalities coming from low-level renderer API. On the highest layer of the architecture, there are scene-related classes and functions which utilize most of the elements in the architecture.

The architecture of OP Engine can be described in the following figure:

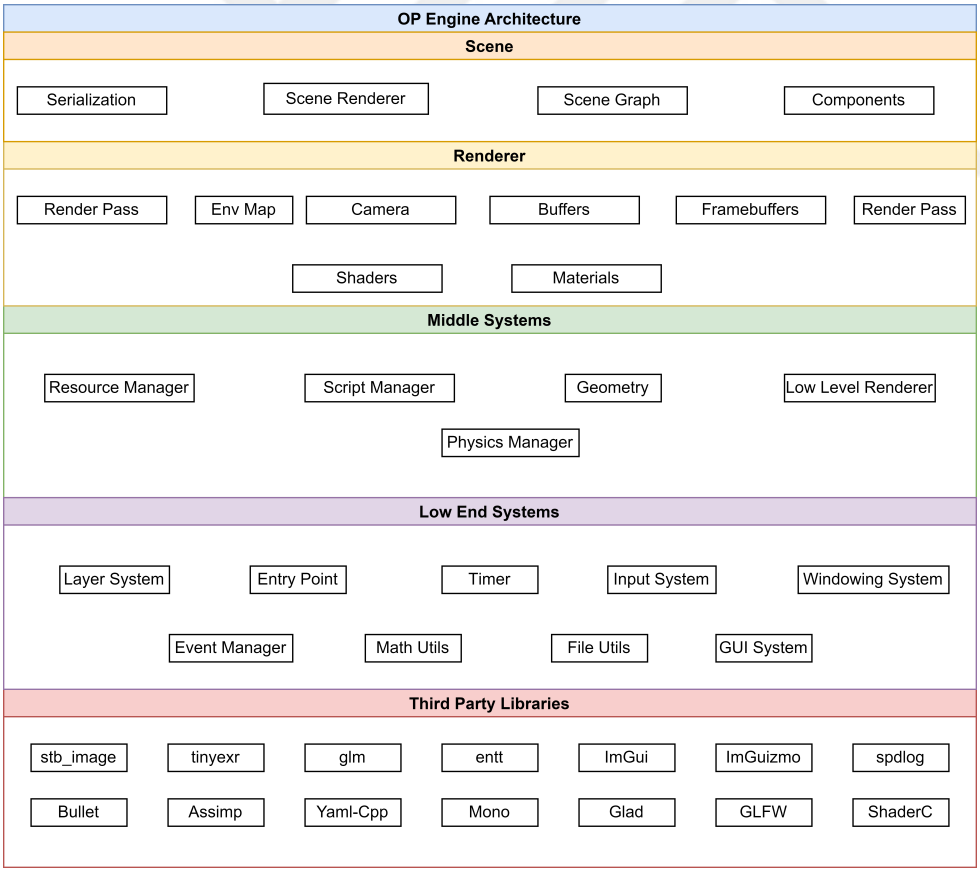


Figure 2.3: OP Engine Architecture

2.1.2.1 Interaction Between Subsystems During Runtime

During the runtime of the engine, each entity located in the scene is updated and during this process, different subsystems interact with each other. The sequence diagram below shows how these subsystems behave during runtime.

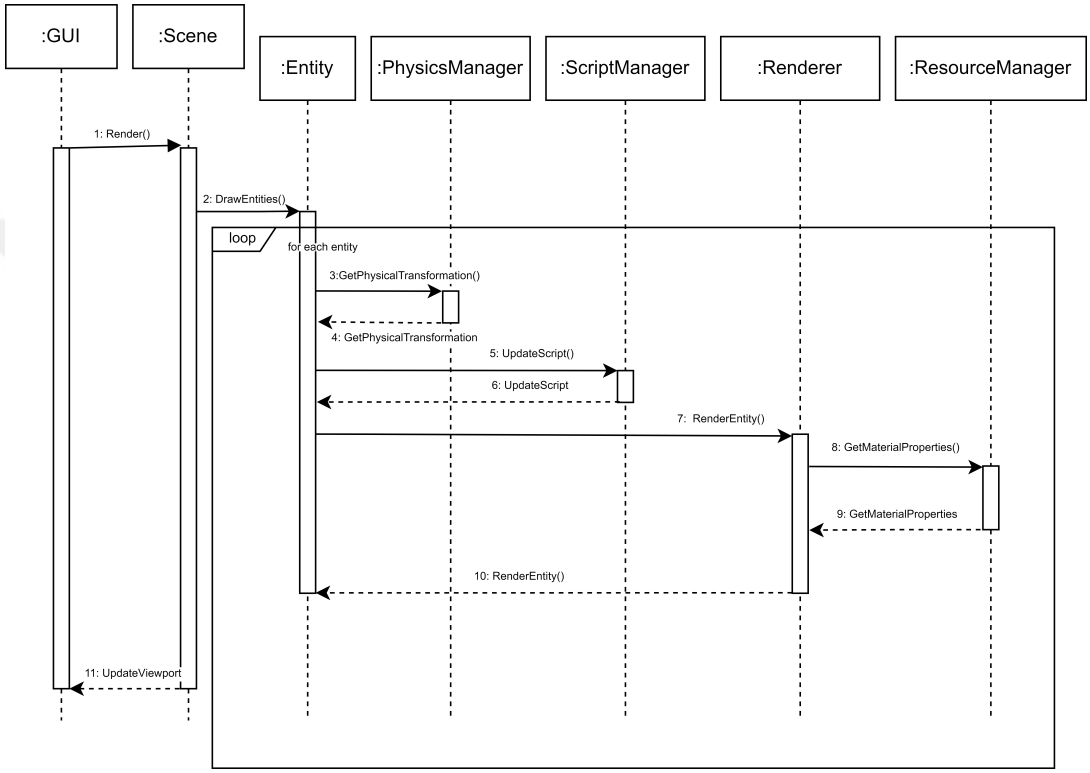


Figure 2.4: OP Engine Runtime Sequence Diagram

As is seen in the figure above, the graphical user interface signals the scene to update all the entities registered. And then, the scene iterates through all the entities, and for each entity, their updated transforms and values from physics and scripting systems are acquired lastly, after reading their transformation matrices, the renderer is signaled for drawing the entities mesh. While drawing continues, the renderer checks the material properties of the entity using the resource manager. After the loop ends, the scene returns a message for GUI to update the viewport.

2.1.2.2 OP Engine as a Static Library

OP Engine is built as a static library to its core and by using this library, different applications for different purposes can be built using the library. In the thesis study, in order to work on the engine, we have also built a level editor OP Editor which uses OP Engine as a dependency.

OP Engine is designed as a static library. The reason behind this is to have tighter control over the code and there is a need for using a small number of binary files after the compilation process[18]. This way, we can bundle most of the low-level libraries and core functionalities together in a single static library to be used by applications that are dependent on OP Engine.

2.1.2.3 OP Editor

OP Editor is the level editor program that has been built with OP Engine, which puts its systems into use and provides an interactive environment to build, save and load scenes and test new algorithms for agents inside the scene by scripting or tweaking the components of different entities. In the following parts of the thesis, all algorithms and subsystems have been tested in OP Editor.

The following figure describes the Runtime GUI for OP Editor:

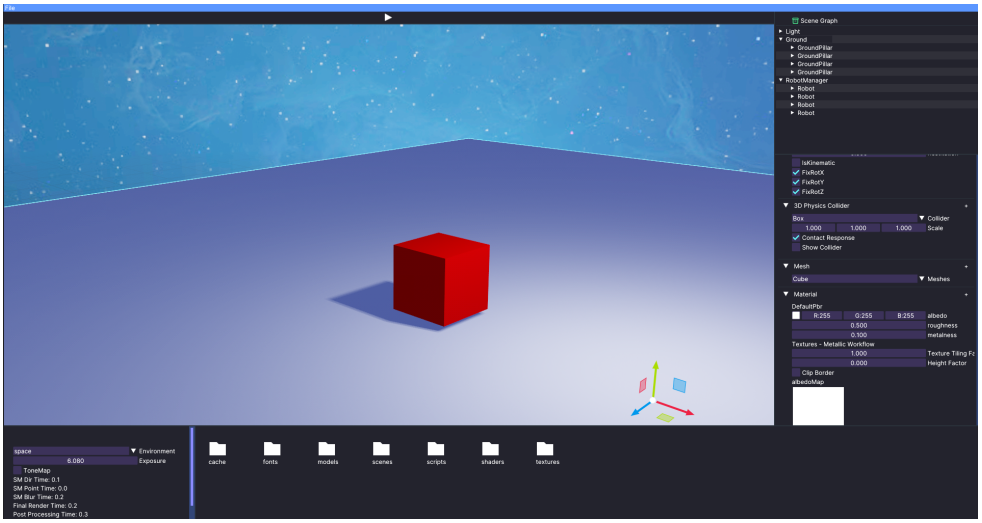


Figure 2.5: OP Editor

2.1.2.4 Third Party Libraries

In OP Engine, there are 3rd party libraries that have been integrated for various reasons. Some of these libraries are just header files, some of them are static libraries and a couple of them are dynamic libraries.

Third-party libraries which are integrated with a header file are:

- **Stb Image[19]:** This library is used for reading images such as textures in different formats.
- **Spdlog[20]:** Spdlog is a logging library to use in the command prompt for easier debugging during the development of OP Engine and also OP Editor.

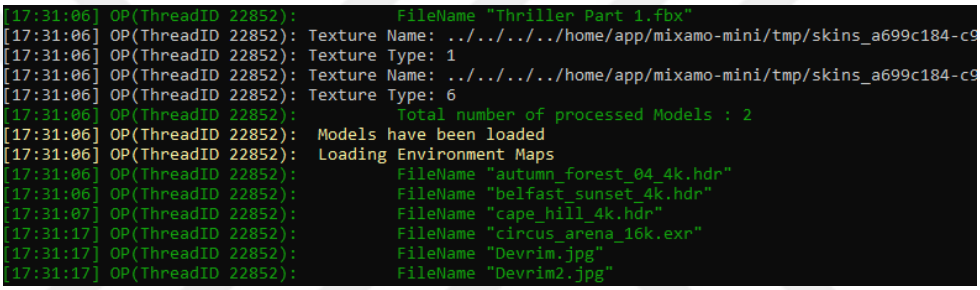


Figure 2.6: Example figure from Spdlog

- **TinyExr[21]:** Tinyexr is used for reading images in HDR. These images are primarily used for building environment maps for PBR rendering because they can express light sources with high radiance.
- **GLM[22]:** GLM is a header-only math library for graphics. It is compatible with types in GLSL and widely used in OP Engine.
- **Entt[23]:** Entt is a library that is used for building an ECS inside OP Engine. By using this library, we can define new components such as *physics materials*, *directional light*, *mesh* etc., and easily create entities having multiple of these components at the same time. This way, we do not deal with the complexity of class inheritance for the relationship between components and entities.
- **ImGui[24]:** ImGui is a GUI library which is written in C++. It can easily be integrated into graphics projects. Therefore OP Engine also utilizes imgui's ca-

pabilities and gives an opportunity to use it in applications that use OP Engine. One thing to note is that for OP Engine, ImGui's docking branch has been used. Figure 2.5 shows what a GUI built with ImGui's docking branch looks like.

- **ImGuizmo[25]:** ImGuizmo is a library that is used together with ImGui that gives Gizmos for manipulating objects sizes, positions, and orientations in the scene by clicking and dragging. In OP Engine, the Position gizmo looks like this:

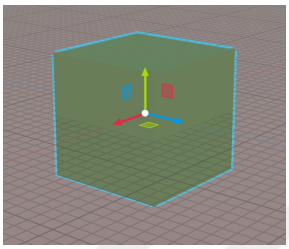


Figure 2.7: Position Gizmo

Orientation gizmo:

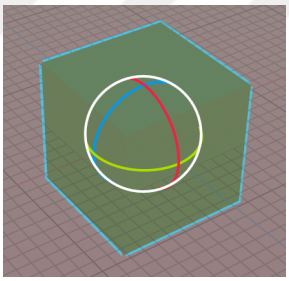


Figure 2.8: Orientation Gizmo

Scale gizmo:

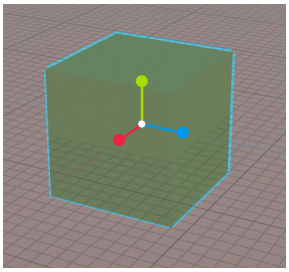


Figure 2.9: Scale Gizmo

- **Bullet[11]:** Bullet is a physics library that is used in OP Engine to evaluate physical collisions and other events. When the simulation is started, a physics world is created which is in parallel to the graphical world, and positions and orientations of objects are updated according to the calculations in the physics world. In OP Engine, there are also **custom callback functions** implemented which are called when an object:
  - Start Colliding
  - Collision continues
  - End Colliding

  These callback functions mentioned above are directly connected to the **Scripting System** so they can be controlled with C# scripts.

- **Assimp[26]:** Assimp is a library for reading mesh files. Assimp supports different formats for these files such as **.fbx** and **.mtl**. Inside these files, there is also information about the material, animation, and other scene properties that can be extracted. OP Engine supports loading models and animations.

- **Yaml Cpp[27]:** Yaml Cpp is a library that is used for serializing and deserializing scene data as **.yaml** files. Inside the OP Editor, it is used for opening scenes that are created in the last session or saving scenes currently under work. Inside these yaml files, every entity and their components are stored as well as scene properties.

- **Mono[28]:** Mono is used inside the **Scripting System** just like it is used in [5] game engine. By using mono, OP Engine can read C# scripts that are written, and use them together with C++. This way, we do not need to compile the application every time when there is a change in a script. Instead, we just change the script and Mono reloads the new behavior.

- **Glad[29]:** Glad is a library that is for using Modern OpenGL[30]'s functions.

- **GLFW[31]:** GLFW is a cross-platform windowing library that is used for the windows that are created in the application that are using OP Engine.

- **ShaderC[32]:** ShaderC is used for compiling and storing the GLSL shaders in the disk for caching and reading them when they are needed.

CHAPTER 3

CORE SYSTEM COMPONENTS OF OP ENGINE

As stated in previous chapters, OP Engine is designed as a library, written in C++. This means that its system is being used by applications that are dependent on OP Engine. So, the library needs to provide some basic functionalities for applications. To expose some of the public classes and functionalities in a systematic way, OP Engine gathers all subsystems that are going to be used by applications in a single header file. This way, a developer whose job is to build a tool using OP Engine's functions can easily import a single header file to begin their project. While designing the core systems that are necessary, we have come up with the idea that there have to be 6 core subsystems of OP Engine, to begin with. These are:

- Application Class.
- Entry Point.
- Layering System.
- Windowing System.
- Input System.
- Event System.

Except for the Entry point, all the others are a part of the header file to be included in OP Engine's architecture.

3.1 Application Class

Application class is a **singleton** element that controls the runtime of the application. It performs the lowest-level functions in the application's main loop. In the constructor of Application Class, we initialize **Window**, set event callback for the window created, and then initialize other subsystems. With setting event callback, we actually bind the **OnEvent** function inside the Application Class directly to the events which are being controlled by the window. More detailed information about the event system can be found in 3.5. These events are being captured and then propagated to all layers if the current layer allows propagation. This process is explained in 3.3. Lastly, a layer for **GUI** is created and pushed to the back of the layer system. The figure below shows the process of initialization in the Application Class:

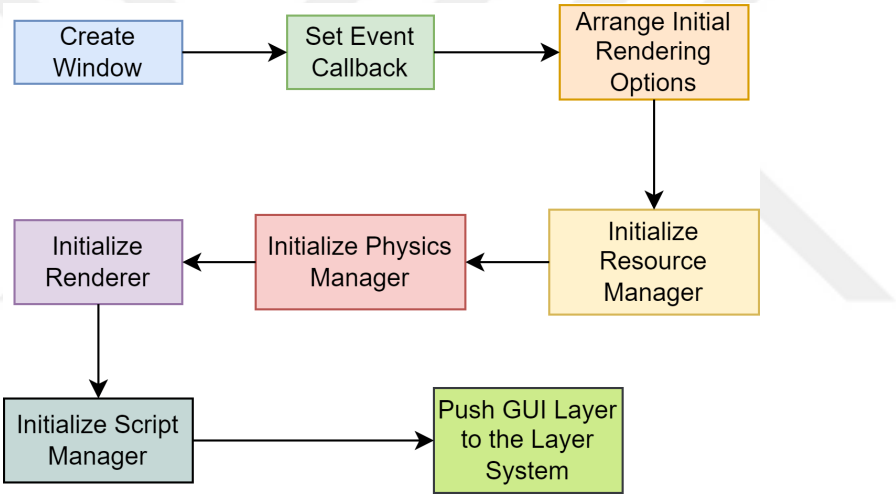


Figure 3.1: Application Initialization

After initializing the necessary subsystems in Application Class's Constructor, we invoke the **Run** function and start the main loop of the application. One thing to mention here is that we do not actually use the **Application** class directly while creating an application using OP Engine. Instead, we create another class that is derived from the **Application** class. In this derived class, we just implement **constructor** and the **destructor**. In the constructor and destructor, developers can add their own layers to the layer system of the application. For example, in **OP Editor** application which uses OP Engine, there is an extra layer called **EditorLayer** that has been pushed to

the **Layer System**.

Application class contains **Layering System** using a member variable with the type **Layer Stack**. Newly created layers can be pushed to the stack normally or pushed as an overlay. If they are pushed as an overlay, we add the layer to the bottom of the stack. Popping from the stack means that the layer on the top of the stack is deleted. A detailed explanation of the layering system can be found in 3.3.

3.1.1 Application Runtime

The runtime of the application is being controlled by invoking the **Run** function which is implemented in the Application Class. This is the main loop that is used in the application to invoke **OnUpdate** functions of the **layers** which are present in the **layer stack**. At the end of each loop, we also call the **OnUpdate** function which is written in **Window** class. Since we use GLFW[31] as our windowing library, in this **OnUpdate** function, we simply do:

- *glfwPollEvents* function to poll the window events.
- *swapBuffers* function to allow double buffering in the application.

Below, the algorithm we use for Running an application is shown:

Algorithm 1 Run function in Application Class

```
1: while is_running do
2:     time ← CurrentTime
3:     timestep = time − LastFrameTime
4:     if window is not minimized then
5:         for layer in layer_stack do
6:             layer → OnUpdate(timestep)
7:     window → OnUpdate()
```

3.2 Entry Point

For applications using OP Engine, we need to provide an entry point inside the library. This entry point is being controlled by OP Engine. Since it is written in C++, the entry point will be the $main()$ function. This means that application developers will not need to define a main function inside their projects. In OP Engine, entry point logic is inspired by Yan Chernikov's Hazel Engine[8].

Developer needs to bind the application they are building to the entry point of the OP Engine by adding a new class which is derived from **Application** class and providing the **CreateApplication** function in their code. Inside the entry point, **CreateApplication** function is being declared as an *extern* function which returns an **Application** pointer. It is written as below:

```
extern Application* CreateApplication(int argc, char** argv);
```

One thing is to note that, every function inside the OP Engine is actually wrapped inside the namespace OP. Therefore, function calls and declarations are widely used such as **OP::CreateApplication(argc, argv)** both inside the engine and applications. Wrapping all of them inside a namespace makes the code easier to read and understand.

The algorithm below shows how applications are started inside the entry point:

Algorithm 2 How Applications Start

1: Initialize Logging System
2: $App \leftarrow CreateApplication(\{argc, argv\})$
3: $App \rightarrow Run()$
4: delete App

As shown above, OP Engine takes control of the runtime of the application. This way, the app developer does not need to think about the runtime.

3.3 Layering System

OP Engine uses a layering system which is essential for propagating different events. The layering system has two different elements:

- Layer
- LayerStack

The layer is an abstract class that defines some functions to be invoked. These functions are:

- **OnAttach**: This function is called when a layer is bound to the system first time.
- **OnDetach**: It is called when a layer is removed from the system.
- **OnUpdate**: In each timestep, this function is called inside the application's main loop. We define what that layer needs to do inside this function.
- **OnImGuiRender**: This function is also called in the application's main loop and it defines GUI commands to be run if that layer utilizes ImGui.
- **OnEvent**: Inside this function, we dispatch the events that are caught and distribute them to the appropriate function inside the layer.

LayerStack is a structure for storing layers for iteration during the main loop. The last inserted stack is on the top of the stack, which means they will receive events earlier. One thing to note is that the layers which receive the event earlier can decide whether the event is propagated to the next layer or not.

The structure of the layer stack is shown in the figure below:



Figure 3.2: Layer Stack Structure

3.4 Windowing System

If we are not trying to build an application that is not just dependent on the console, we need a windowing system. Inside OP Engine, windowing utilizes GLFW[31]. The reason why we wanted to use GLFW is that it is cross-platform. So we can use this library on multiple platforms. But since OP Engine is being designed to be flexible in terms of selected libraries and APIs, in the architecture of the engine, we created an abstract class for windowing which has a static **Create** function in it.

After that, we created another class derived from the window class which defines all of the **virtual** functions inside the base **Window** class. The functions defined in these classes are as follows:

- **OnUpdate**: This function is called inside the main loop of the application which is defined in algorithm 1.
- **GetWidth, GetHeight**: Returns the current height and width of the window respectively.
- **SetVSync**: Sets vertical synchronization which limits the fps of the system to the refresh rate of the monitor.
- **HideMouseCursor, ShowMouseCursor**: In the application, we may want to show or hide the mouse cursor, these functions are for controlling that. They can also be controlled from the **Scripting System**.
- **Init, Shutdown**: They respectively initialize and shutdown the windowing system.

- **SetEventCallback**: This function acts like the bridge between the event system of OP Engine and the event system of the windowing library(in our case it is GLFW). We set the event callback function which is an `std::function` like described in figure 3.1 when we are initializing the application after creating the window.

One thing to mention about the windowing system is how we capture events using the windowing system. We define a struct called **WinData**. This struct is set as **user pointer** in GLFW for the window and whenever a glfw callback function is triggered, we get the data as a WinData struct by explicit casting and invoke the Event Callback function which is inside the struct. In the code, this struct is defined as below.

```
struct WinData
{
    std::string Title;
    unsigned int Width, Height;
    bool VSync;
    EventCallbackFunction EventCallback;
};
```

This struct is stored as a member variable called **m_Data** inside the Window class. When we initiate the window within **Init** function, we set the **m_Data** as user pointer using `glfwSetWindowUserPointer(m_Window, &m_Data)` statement. In this statement, it is seen that m_Data's address is being passed. This way, we can acquire m_Data anytime we want and manipulate it during runtime. And this is mostly done inside callbacks.

In m_Data, since we store the **Event Callback** of the Application, which is propagated to different layers as explained in layering section 3.3, we can call the **Event Callback** function to initiate this propagation from the base layer.

3.4.1 Callbacks

In GLFW, there are callback functions for various events. The callbacks used in OP Engine are as follows:

- **WindowSizeCallback**: It is triggered when the size of the window is changed.
- **WindowCloseCallback**: When we close the window, this callback is called.
- **SetKeyCallback**: This callback is called when we press a key on the keyboard. In addition, GLFW has different actions for key presses. These are:
  - **GLFW_PRESS**: When we press a key.
  - **GLFW_RELEASE**: When we release the key that is being pressed.
  - **GLFW_REPEAT**: If a key is being pressed.

  For each case, we trigger a different event in our **EventCallback** function which is stored in **m_Data**.
- **SetCharCallback**: It is triggered when we type a character. It is used for key typing events. One thing to mention here is that this callback is different from the set key callback.
- **SetMouseButtonCallback**: This callback is invoked when one of the mouse buttons is being pressed(left, right, middle, or other mouse buttons). There are two different actions for this callback which are as follows:
  - **GLFW_PRESS**: When mouse button is pressed.
  - **GLFW_RELEASE**: When mouse button is released.

  Just as in the **SetKeyCallback** part, a different event is triggered for different actions.
- **SetScrollCallback**: It is triggered when we scroll inside the window. The callback is called with the information of **xOffset** and **yOffset** so that we can feed this information to the **EventCallback**.
- **SetCursorPosCallback**: This callback is invoked whenever the mouse is moved to a new location. It is called with **xPos** and **yPos** variables which are the positions of the mouse on the x-axis and y-axis respectively. The current location of the mouse is then passed to the using the **EventCallback** function.

To sum up, **Windowing System** is one of the backbones of **OP Engine**. Every event and input query inside the application is first fetched from the Windowing System

using the logic explained above. As it is stated above, **OP Engine** uses GLFW as the windowing library, but a different windowing library can also be used because GLFW-related codes and callbacks are being abstracted with class inheritance. So, a different library can also be used for window management, but the logic will stay the same.

3.5 Event System

As explained in the Windowing System section 3.4, we bind an **Event Callback** function inside a struct which is used as a **User Pointer** by the windowing library. Then, whenever an event occurs, the same event is being propagated from one layer to another until it is handled. A generalized version of the event flow is shown in the figure below:



Figure 3.3: Event Flow

While constructing the Event System, we have first written an **Event** class to form a base for all different Event Categories. As discussed above, all these events are actually generated using the windowing library and then propagated into deeper layers inside the application. There are various Event types that are derived from the base **Event** Class.

- **Window Resize Event:** It is an application category event. It is raised when the windowing library detects a size change.
- **Window Close Event:** Closing the window triggers an application category event.
- **Key Event:** A key event is triggered whenever a certain event where the keyboard is involved. There are multiple event classes that derive from the Key Event class which are explained below. A key event consists of a **KeyCode** variable for keeping track of which key is being pressed or released.

- **Key Pressed Event:** It is derived from **Key Event** class and it is raised whenever a key is being pressed. It has a function called `GetRepeatCount()` which returns how many times that key is repeated.
- **Key Released Event:** It derives from **Key Event** class and it is triggered when a key is released.
- **Key Typed Event:** A key typed event can be confused with **Key Pressed Event** but it is for detecting especially typing. For example, writing something with the keyboard.

In the figure below, the complete class diagram of OP Engine's Event System can be seen;

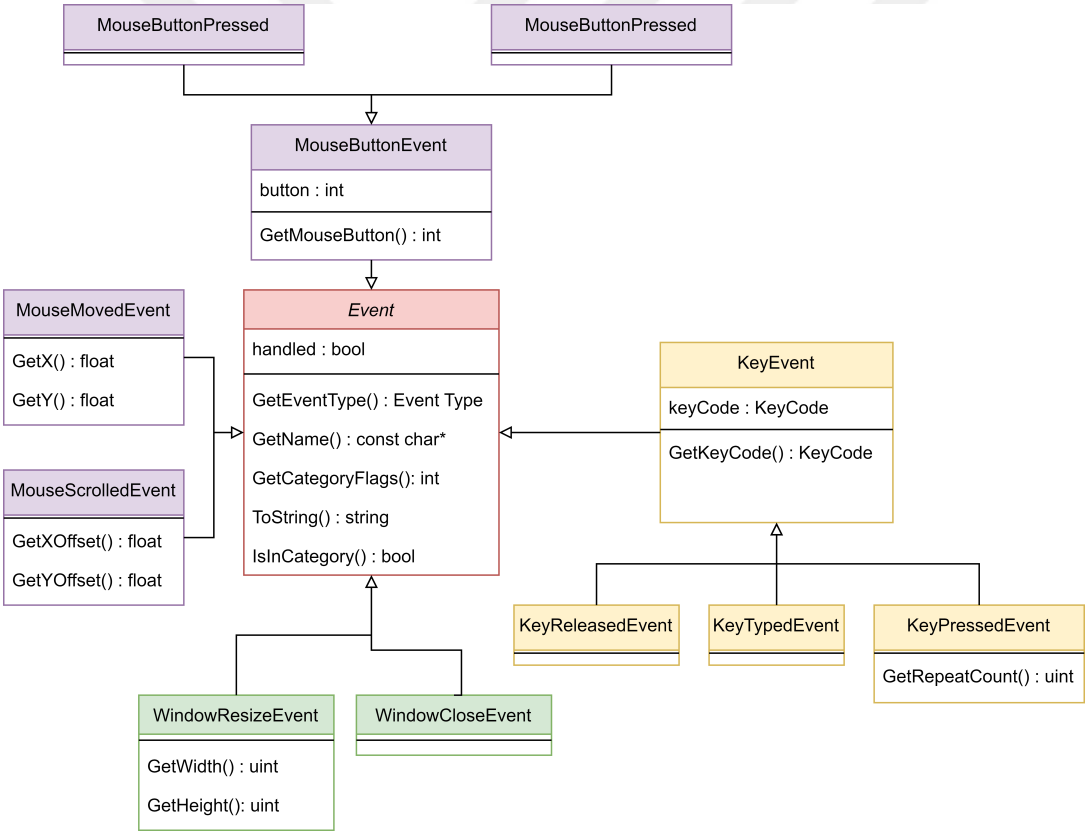


Figure 3.4: Event Class Diagram

In the above figure, the colors on the classes describe the categories of events. There is also a way to distinguish events by the categories in OP Engine which might be

useful when dealing with Events. Furthermore, we also need to distribute these event types into different custom functions that we write for runtime. For this purpose, we have to introduce Event Dispatcher3.5.1.

3.5.1 Event Dispatcher

In Op Engine, Event Dispatcher logic is used for distributing different events to the member functions of the particular layer. For this purpose, a new class called **Event Dispatcher** has been introduced. This system in OP Engine is derived from Yan Chernikov's Hazel Engine's event system[8].

Event Dispatcher holds a member variable which is a reference to an event itself. The only way to construct an Event Dispatcher object is the give it an event as a reference. This way, we do not build another Event object while creating an Event Dispatcher.

An Event Dispatcher utilizes a function called **Dispatch()** which expects a **function** which is returning a boolean variable. This function is also declared as a template function and the type we need to give is the event type.

In runtime, when an event is triggered and we need to distribute that to different functions, we create an Event Dispatcher and invoke **Dispatch()** function for all the Event Types we need to work with. We also give the function we want to invoke as an argument. And then, **Dispatch()** function checks the static type of the template and the actual event. If they match, the function is invoked. In the figure below, this process is illustrated.

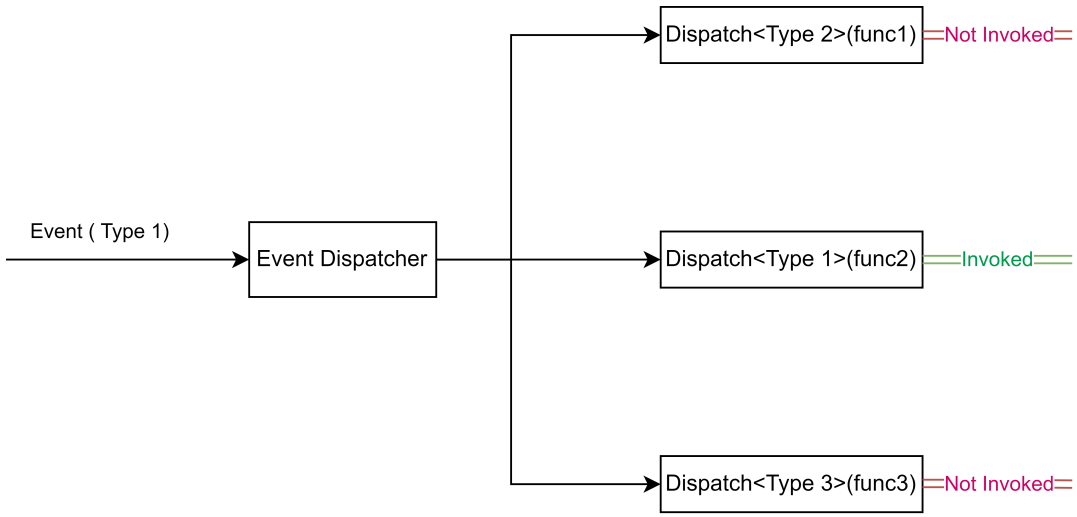


Figure 3.5: Event Dispatcher Logic

3.6 Input System

The input system of OP Engine is loosely connected with GLFW[31]. For keycodes and mouse button codes, there are different enum classes and every button and the key is encoded in these enum classes. There is also a base Input Class which has **static** functions. These static functions are only declared in Input's header file but not defined. The reason behind that is to provide a base for cross-platform support in the engine.

The idea is to implement these static functions in a separate source file and decide whether compile them or not with preprocessor commands. For now, since OP Engine only uses GLFW, there is only one source file which implements the functions of the Input class.

These functions are;

- IsKeyPressed
- IsMouseButtonPressed
- GetMouseX
- GetMouseY

- GetMousePos

All of these functions get the window associated with the application and call the appropriate GLFW function for retrieving the information and returning the expected value. In the future, if developers decide to introduce a new windowing library to the Engine, they are only going to write a new source file for defining input functions and control the compilation of multiple source files regarding inputs for different windowing libraries or platforms with preprocessor commands. This feature gives OP Engine flexibility to be enhanced. Because since the functions will stay the same, other subsystems of the Engine will not be affected by the change in the windowing system. The systems affected by this will only be the Windowing system and the Input System.

CHAPTER 4

GRAPHICAL USER INTERFACE (GUI) OF OP ENGINE

OP Engine utilizes the GUI by using the docking branch ImGui[24] library which is inspired by the public branch of Hazel Engine[8]. With the help of the docking branch, we can "dock" the ImGui windows inside the main window or other ImGui windows. Before starting the implementation of GUI with this library, we first need to compile the project with **backend** files provided by ImGui. In OP Engine, **imgui_impl_opengl3.cpp** and **imgui_impl_glfw.cpp** are being used. In future studies, when we introduce new APIs and Windowing libraries, we may control this with preprocessor commands. Overall, with this, we create a neat-looking graphical user interface. Here is an example of the OP Editor's GUI with an example scene with multiple game objects having different material properties:

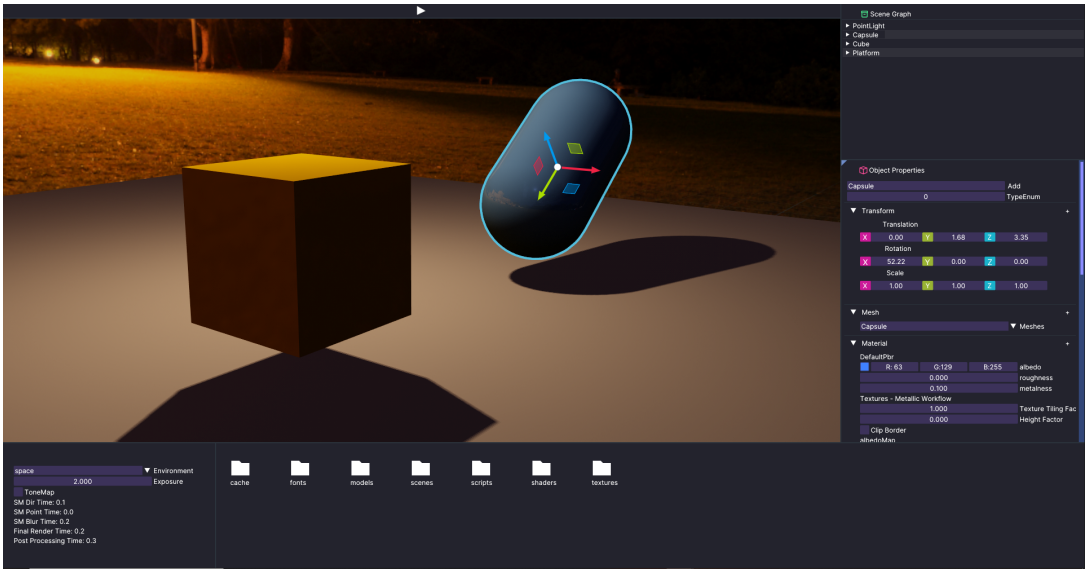


Figure 4.1: OP Engine GUI

In the figure above, we see different ImGUI windows which are docked in the main window of the application. The components used in the GUI of OP Editor are:

- **SceneGraph:** Scene graph controls and displays the game objects which are present inside the scene, it works together with the ECS(Entity Component System) which will be explained in detail in future chapters.
- **Properties Window:** This window is docked below the **Scene Graph** and it shows the properties of the **selected** object. Its properties can be manipulated from this window. In the figure above, we can see the properties of the **Capsule** object since it is selected by clicking the Capsule object in Scene Graph or clicking the object itself in the viewport.
- **Viewport:** Viewport displays the scene and it is docked on the left side. In the figure above, we can see that the capsule object is selected since it is highlighted with blue and there is a gizmo for manipulating its position. We can drag and drop the gizmo to change the position of the object.
- **Scene Play-Stop:** On the top of the viewport, there is a button to start or stop the scene. When we start the scene, OP Editor changes to the **Runtime** mode instead of **Editor** mode. With this, the viewport starts using the active camera in the scene graph(if it exists) instead of the editor camera and the physics world starts running for the objects having physical properties.
- **File Explorer:** It is docked in the below part of the window and it displays the files and the folders within the project's base folder. We can drag and drop different items such as scene files or textures to the **Viewport** or to the **Properties Window**.
- **Scene Properties:** This window is docked in the left part of the **File Explorer** and it shows scene properties like tone-mapping, environment map, exposure amount, and some metrics for rendering.

Alongside with ImGui, ImGuizmo[25] and ImPlot[33] are also used. The purpose of using ImGuizmo is to visualize gizmos which are presented in figures representing Orientation Gizmo 2.8, Position Gizmo 2.7 and Scale Gizmo 2.9. On the other hand,

Implot is used for visualizing plots. In further studies, it is planned to use ImPlot more extensively for performance calculations. Here are some example figures from the demo window generated with ImPlot in an application written by OP Engine (OP Editor):

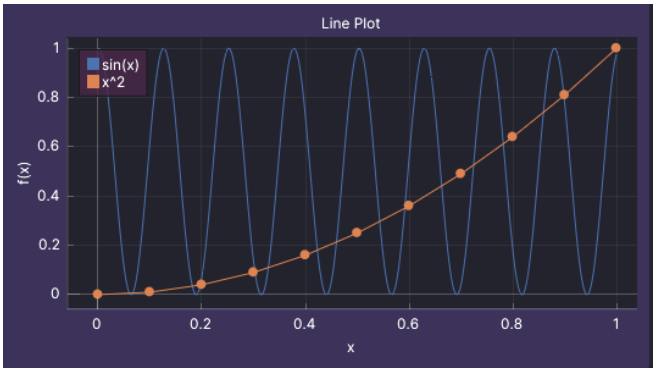


Figure 4.2: ImPlot Line Plot Demo

In the example above, we can use line plots as shown. We can also use ImPlot within some ImGui structures like tables, which is the case in the figure below:



Figure 4.3: ImPlot with ImGui Tables Demo

All the GUI events are being controlled with a specific **layer** which is put on the top of the **layer stack** of the application. In application runtime, GUI operations are separately propagated to the next member of the layer stack. Each layer inside the stack has a function for controlling GUI behavior. But these functions which control

the GUI behavior are invoked after Updating the state of the whole application. This means that ImGui functions for generating and displaying windows are actually called in every time step while the application is running. This might seem tedious work but it does not create much difference in the performance of the editor itself. And in future studies, OP Editor will be able to create standalone runtime applications of the projects implemented with itself which does not use the ImGui library at all. So the load coming from the ImGui usage will not be an issue.

4.0.1 GUI Layer in the Application

Application class generates a layer as an "overlay" which is always on the top of the **layer stack**. This layer is responsible for the commands coming from ImGui. Every other layer which wants to use ImGui windows or widgets implements a function called **OnGuiRender**. In the main loop, the application looks to traverse all the layers in the stack and calls that function for every one of them. This action is being done **after** updating the state of all layers which is explained in algorithm 1. So in addition to that algorithm, we also utilize the algorithm below inside the main loop to manage the GUI:

Algorithm 3 Managing GUI for layers

1: $GuiLayer \rightarrow Begin()$
2: **for** $layer$ in $layer_stack$ **do**
3: $\quad layer \rightarrow OnGuiRender()$
4: $GuiLayer \rightarrow End()$

Since the GUI layer is actually a class that is derived from the base **layer** class, it must implement the functions **OnAttach**, **OnDetach**. In addition to these, the GUI layer also introduces new **Begin** and **End** functions which are being called in every time step for creating the frames.

4.0.2 OnAttach

The algorithm which is utilized when the GUI layer is attached to the application is shown below:

Algorithm 4 Gui Layer Attach

1: $ImGui :: CreateContext()$
2: $ImPlot :: CreateContext()$
3: $io \leftarrow ImGui :: GetIO()$
4: $io \rightarrow EnableKeyboard$
5: $io \rightarrow EnableDocking$
6: $io \rightarrow EnableViewport$
7: $LoadFonts()$
8: $SetDefaultFont$
9: **SetDarkThemeColors()**
10: **SetGlobalStyles()**
11: $app \leftarrow GetApplication()$
12: $window \leftarrow app \rightarrow GetWindow()$
13: $initGLFWOpengl(window, true)$
14: $initOpengl3(\text{"}\#version450\text{"})$

In the algorithm above, **OnAttach** function of the GUI Layer is explained. The first thing the OnAttach function does is create contexts for **ImGui** and **ImPlot**, and then there are some configurations made for enabling certain features like keyboard, docking, and viewports. After that, we load the fonts for different sizes and styles. The styles are *light*, *regular*, *medium*, *bold* and sizes are 6,8,10,12,14,16,18 and 20. In OP Engine, **Inter**, **Ubuntu** and **OpenSans** fonts are being used from Google Fonts [34]. The default font is defined as **Inter-Light-12** which means Inter font with a light style and 12 as its size.

And then we define the custom theme colors. The custom theme color for OP Engine is a purplish color which can be seen in figure 4.1. By manipulating the statements which are written in **SetDarkThemeColors** function, we can change these colors according to their needs. After defining the colors, we define the global styles with the function **SetGlobalStyles**. We define parameters like sizes of the texts, borders of

windows, scrollbar sizes, and frame roundings. Lastly, we have to call the initialization functions written by the backend which we included before implementing our own GUI. For now, OP Engine uses GLFW and OpenGL3 (version 450) implementations. But as said before these can be controlled with preprocessor commands in the future if we introduce new graphics APIs and windowing libraries. ImGui has multiple implementations for various APIs and windowing libraries.

4.0.3 OnDetach

When we detach the GUI layer, we simply shut down the GUI system in the reverse order we did when we attached it. It is shown in the algorithm below:

Algorithm 5 Gui Layer Detach

1: $shutdownOpengl3()$
2: $shutdownGLFWOpengl()$
3: $ImPlot :: DestroyContext()$
4: $ImGui :: DestroyContext()$

4.0.4 Begin and End

In every time step, we call **Begin** and **End** functions as described in the algorithm 3. Begin function is straightforward, we call functions from the ImGui and ImGuizmo libraries. We do not call functions from ImPlot because all necessary commands are included in the ImGui library. With these two functions, we have direct control over the GUI commands in the application on the engine side. GUI Commands written in the different layers introduced in the layer stack are called between the **Begin** and **End** functions. The algorithm for Begin function is shown below.

Algorithm 6 Gui Layer Begin

1: $Opengl3NewFrame()$
2: $GLFWNewFrame()$
3: $ImGui :: NewFrame()$
4: $ImGuizmo :: BeginFrame()$

After that, in the **End** function, we match the display size with the window size of our application update viewports if they are enabled. The algorithm is shown below:

Algorithm 7 Gui Layer End

1: $io \leftarrow ImGui :: GetIO()$
2: $app \leftarrow GetApplication()$
3: $io.displaySize \leftarrow GetApplicationWindowSize()$
4: $ImGui :: Render()$
5: $Opengl3RenderDrawData(ImGui :: GetDrawData())$
6: **if** Viewports Enabled **then**
7: $\quad currentContext \leftarrow glfwGetCurrentContext()$
8: $\quad ImGui :: UpdatePlatformWindows()$
9: $\quad RenderPlatformWindowsDefault()$
10: $\quad glfwMakeContextCurrent(currentContext)$

4.0.5 Icon Fonts

OP Engine also loads some custom fonts generated for icons. Icon fonts are expressed in three sizes which are:

- Big icons which have the size 32.
- Medium icons which have the size 24.
- Text-sized icons which have the size 14.

While loading icons, the system reads a **.ttf** file that contains the icons and we also define the codes of these icons in a header file called **Font.h** and give them hexadecimal numbers. In the loading stage, we provide a range with the maximum and minimum values of the icons. This way, ImGui loads these icons. After that, whenever we want to use an icon, we can use the hexadecimal numbers defined in the header and ImGui retrieves the correct icon corresponding to that. Icons can be seen in figure 4.1 where we show the GUI of the OP Editor.

CHAPTER 5

ENTITY COMPONENT SYSTEM OF OP ENGINE

In a real-time agent-based simulation application or engine, we need a way to represent the agents or any object and update their state according to their properties at every time step. For example, some of the objects in the scene might be changing in certain events or they might carry some kind of information that is distinct from other objects in the scene. For example, let's think about a simple scene like below:

- A red sphere which stands at point (0.0, 1.0, 0.0).
- A point light object emitting blue color which stands on the top of the red cube (0.0, 3.0, 0.0) point.
- A thin white platform at the origin point.

By looking at this simple scene, we can conclude that **red sphere** and **white platform** can be represented as objects of the same class because they both have to have a mesh for their shapes (sphere and a rectangular shape), a material for representing their colors and their positions. In each time step, the renderer will look at their positions (more formally their transformation matrices) with their material properties and display them on the screen.

However, we can not say the same thing for the **point light** object. Because this object has a position but does not have a mesh or a material. In addition, it has a different purpose which is emitting light. This raises the question. How can we represent all of these objects sharing only a portion of their properties?

The messiest solution would be creating a class for every unique behavior. This means a distinct class for sphere and platform, having material properties, a mesh and a po-

sition, and another distinct class for the light object which has point light properties and a position. If we want to improve this approach, we can implement a base class for transformation which holds the position, orientation, and scale information, secondly, a class for holding the mesh information, thirdly a class for materials and lastly another class for point light properties. The class diagram will look something like this:

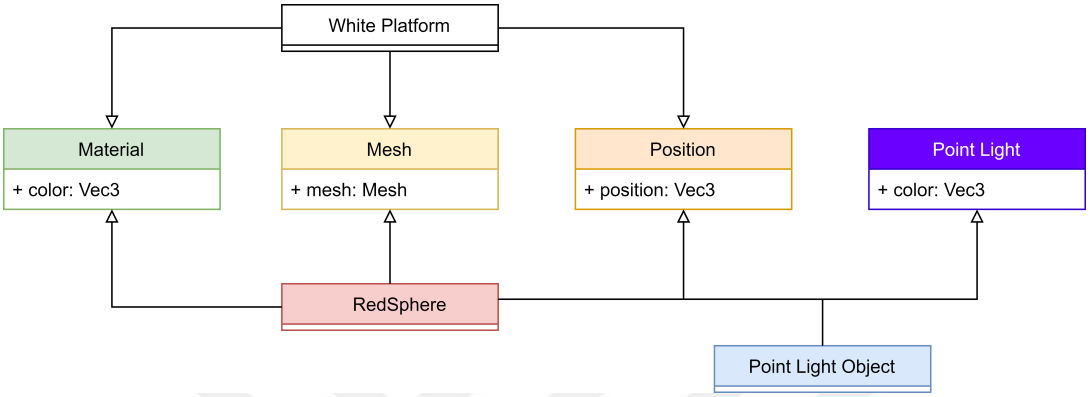


Figure 5.1: Multiple Inheritance for a simple scene

In the end, when we start creating actual game objects, we derive from these base classes using multiple inheritances. This might seem the right thing to do if we think about very simple scenes. But what if we increase the complexity of our scene like this:

- A red sphere at which stands at point (0.0, 1.0, 0.0). The sphere also has physical properties and it has to move according to physics laws when we run the simulation.
- A point light object emitting blue color which stands at point (0.0, 3.0, 0.0). The point light object is also the active camera, which means when we run the simulation, we have to see from the eye of the point light. The point light also has to change its orientation when we move the mouse and its position when we press keys (w,a,s,d).
- A thin white platform at the origin point. It also has physical properties so the red sphere has to bounce back when it hits this platform.

As seen in the above scene, we have introduced some new properties which are physics properties. For physics properties, we need the physical material and a collider for testing collisions. We also introduced the camera concept. So we need a class for cameras as well. In addition to that, we have also added a way to change the camera's orientation and position. This is the scripting behavior where we add new behaviors to objects in runtime. So we need to add a scripting class as well. The new class diagram will eventually look like this.

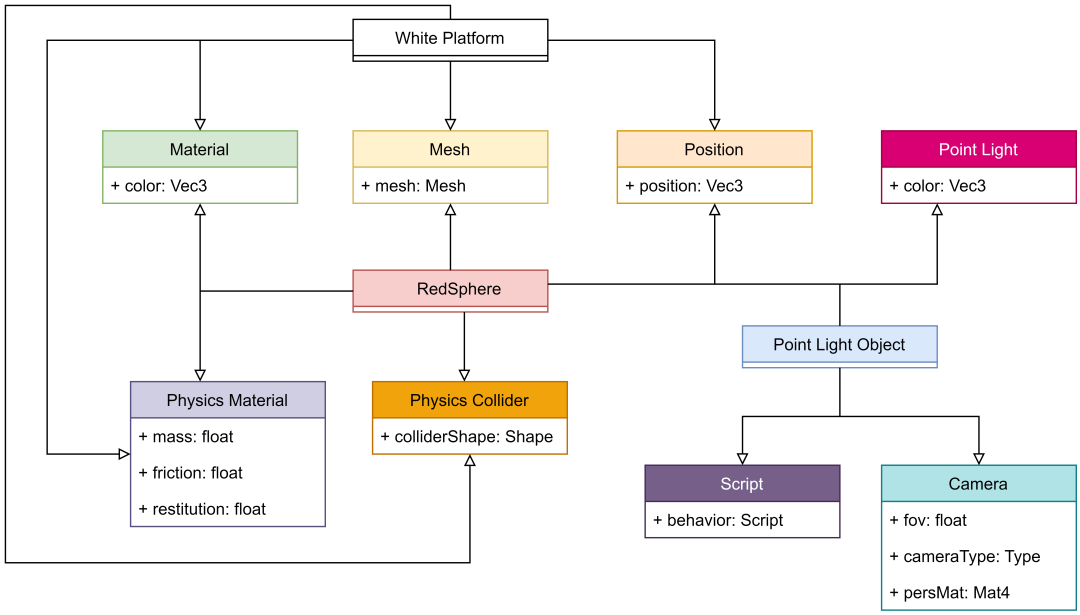


Figure 5.2: Multiple Inheritance for a more complex scene

From the figure above, we can see that we are introducing a lot more inheritance relationships when we create more objects having compound properties. In the engine code, if we want to update these objects, we need to write specific codes for every `update()` function for all classes which is tedious work.

This solution is also going to have some different complications because we are actually filling up the **vtable** of these classes with multiple inheritance relationships which might cause performance problems because processors need extra cycles to look up to these virtual functions. In addition, having a function to update the state in the game object class is not a good idea. Because there are different systems inside the engine which need to do specific things with the properties of the object. So, these

systems need to iterate all objects in the scene separately and do the work.

So, this introduces an issue of how we store these game objects in the memory. For example, let's think that we store all game objects in the scene inside a special array. For this case, we do not have `update()` functions in the classes which represent game objects. We have these functions inside the abstract classes which game objects derive from like **point light** class. For example, a subsystem like the **renderer** may want to process the objects which only have light properties in order to send them to the **shaders** in a tightly packed buffer.

Let's think about the addresses of these game objects in the main array when the renderer is trying to find the objects with a point light property. There is no guarantee that all the objects with point light properties are packed together in the memory. It is more likely for them to be sparsely distributed in the array. If we want to illustrate this situation. For simplicity, let's assume that each object in the array has a size of **1 byte**. In the below figure, this situation in the array is explained. In addition;

- **PL:** Point Light
- **SL:** Spot Light
- **CAM:** Camera
- **DL:** Directional Light

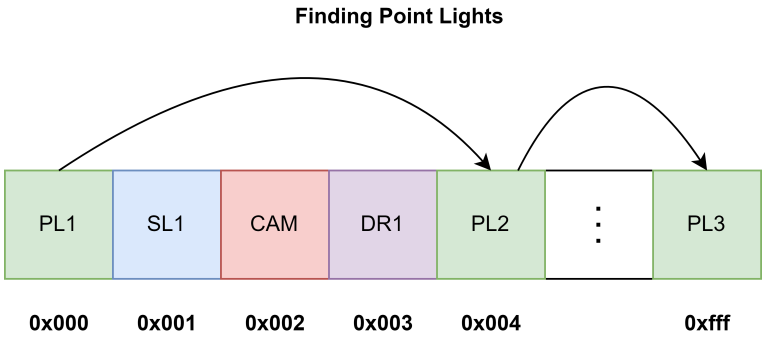


Figure 5.3: Non Tightly packed array

In the case above as illustrated in figure 5.3, we see that point light objects are not tightly packed.

This is an important performance problem. Because when a processor reads a value from the memory, it actually saves some of the adjacent addresses to the **caches**. This way, if we try to reach the adjacent memory address in the next step, the processor directly reads the value from the cache instead of doing a memory read operation which is costly. As illustrated in the figure below, reading values from adjacent memory addresses is going to be less costly.

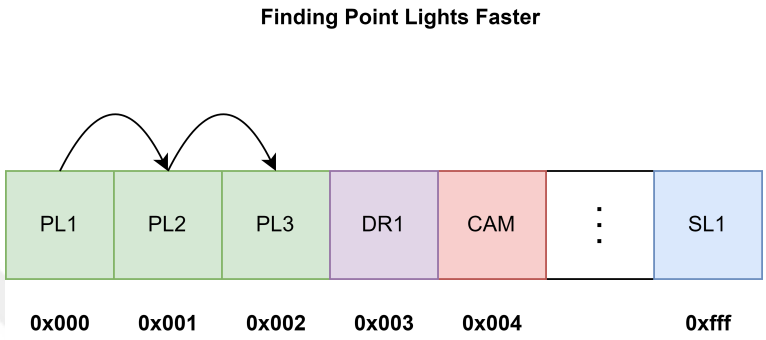


Figure 5.4: Tightly packed array

So, we need to find a way to pack all the objects having similar properties together. This way, we can deal with the data more efficiently. This problem is solved by creating an ECS (Entity - Component System). Entity Component Systems are essential architectural patterns used in modern game engines like Unreal Engine[4], Unity[5], and Godot[6]. Each of these engines and their derivations has found a way to utilize the concept of ECS. In OP Engine, Entt[23] library is used for Entity Component System.

5.1 OP Engine's ECS

In OP Engine, Entt[23] library is used for Entity Component System. Entt is a header-only library for integrating entity component systems and that's why it is used in OP Engine. In OP Engine's architecture, class inheritance is not used for defining different behaviors of the object. Instead, we behave each object in the scene as an **Entity** and we can add or remove **Components** from it.

Components are connected to the corresponding object. However, they are tightly

packed together. By using entt's utilities, specific components can be traversed efficiently each time step, which increases the performance. For example, we can separately process entities that have script components that will invoke functions from the script manager class and physical properties that are going to be processed by the physics system.

5.1.1 OP Engine's Entity Class

In OP Engine, every object in the scene must be an Entity. Entities have:

- A unique ID that is provided by entt library.
- A pointer to the scene to which that entity is connected.

We also need to define some templated functions for managing components that are attached or going to be attached to the Entity which are:

- **AddComponent:** This function adds the component class which is written in as a template. If that component class is already attached to the component, the function raises an exception. This case is also handled in OP Editor which is a level editor designed with OP Engine.
- **AddOrReplaceComponent:** It works just like **AddComponent** function. The difference is, if there is already a component of that class attached to the entity, that component is being replaced by the newer one.
- **Patch:** Patch function is required to inform entt's event system if some components are manually changed in the code. This way, we can trigger some custom functions to deal with the situation.
- **GetComponent:** This function returns the component which is attached to the entity. If there is no such component, it raises an exception.
- **RemoveComponent:** It removes the attached component from the entity. If there is no such component, an exception is raised.

- **HasComponent:** It returns whether an entity has a specific component or not.

There are also functions for updating the transform values of the entity. These functions are required in order to recursively update every child entity starting from the parent because if a parent entity is being moved, then all of its children and grandchildren also have to move. These functions are deeply connected with the **Transform Component** and **Relationship Component**.

5.1.2 OP Engine's Components

In order to work with ECS in OP Engine, we first have to add components as structs. After that, we can use them in the scene. Every component which is used in OP Engine is defined in a single header file in the source code which is named **Components.h**. The components that are defined are:

- **Outline Component:** For now, it is mainly used for outlining the selected objects in the scene in OP Editor. However, it may have different use cases. A game or simulation developer might want to tag an object which is seen through other objects. There are various examples of this in different games. For example, in OP Engine, an object which has an outline component looks like the following figure:

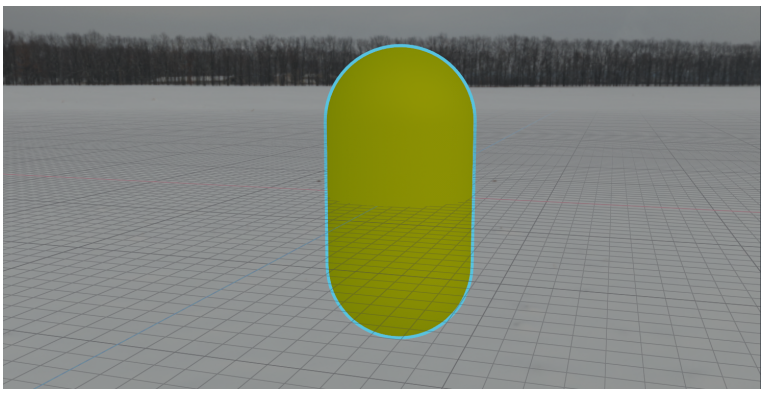


Figure 5.5: Outlined Capsule

If another object is between the camera and the outlined object, the outline is also seen. It is illustrated in the figure below:

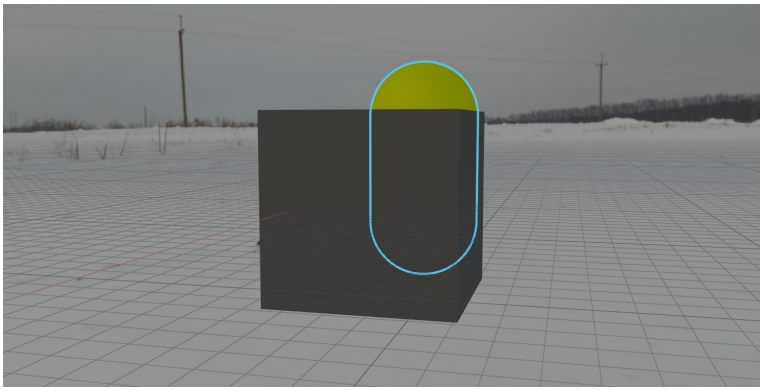


Figure 5.6: An object between outlined capsule and camera

- **ID Component:** This component is added to every entity that is created by default. Inside this component, a unique id is stored. This id is different than what is given by the entt library. It is for finding and processing entities in other subsystems of OP Engine.

- **Root Component:** In OP Engine, we can have **nested components**. Root Component is added to an entity whether it has no parents. It is mainly used in the **scene graph** for determining the root entities.

- **Relationship Component:** This component also exists in all entities in the scene. It is for determining the children, parents, and siblings of the entity. Each relationship component has:

  - The number of its children
  - ID of the parent
  - ID of the next sibling
  - ID of the previous sibling

  By using these IDs, we can traverse the whole tree of an entity's relatives. The relationship between entities is being displayed in OP Editor's scene graph GUI like the figure below:



Figure 5.7: Relationship Between Entities

- **Tag Component:** Every component has a tag component that defines a specific name for the entity. It can be useful to group certain entities under a tag.
- **Transform Component:** Transform Component is also being found in every entity in the scene. It stores the information of:
  - Local translation, rotation, and scale. These values are according to the parent of the entity.
  - Translation, rotation, and scale. These are the global values according to the world.

  By using these values, a **transformation matrix** is being generated for **graphics system** and **physics system** to use. There is also a function to get a view matrix for entities that have camera components as well. This component can be modified inside the OP Editor and looks like the figure below in GUI. The design is inspired by Hazel Engine's public branch[8]:

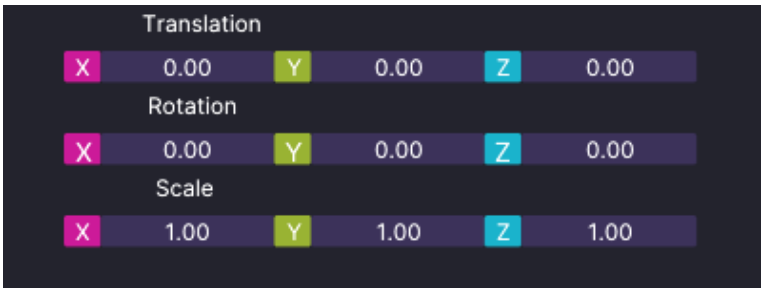


Figure 5.8: Transform Component in GUI

- **Camera Component:** Entities having a camera component act as cameras during the runtime of the simulation. They have:

- Scene Camera object
- A flag for determining whether that camera is primary or not
- A flag for determining whether the aspect ratio is fixed or not

In OP Editor's GUI, a camera component is displayed in the figure below:

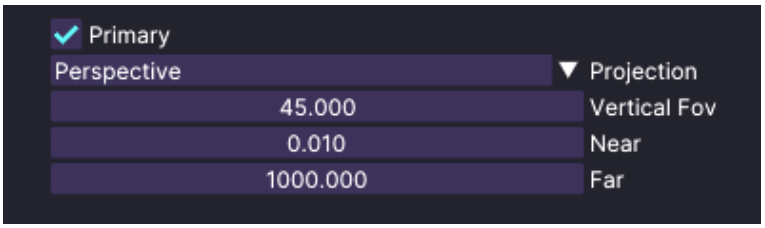


Figure 5.9: Camera Component in GUI

- **Script Component:** This component is directly connected to the script manager of OP Engine. It has:
  - A name field for the script module's name
  - An unordered map which stores the public fields inside the script

When we first add a script component to an entity, normally a **null script** is attached as a module. This **null script** is predefined in OP Engine's scripting system and it does nothing. The initial script component with **null script** looks like the figure below:



Figure 5.10: Script Component with Null Script

However, when we actually drag and drop a **C#** script into that null script, the editor will display that script's name and its public fields. We can change these fields easily just like it is being done in Unity[5]. For example, the script component looks like the following figure of a script named **Free First Person Camera** is added:

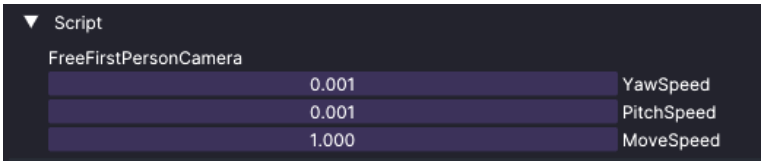


Figure 5.11: Script Component with a Script

As is seen in the figure above, we can change the public fields **Yaw Speed**, **Pitch Speed**, and **Move Speed**.

- **Dir Light Component:** This component is added if an entity has a directional light property. In OP Engine, the technique of **cascaded shadow mapping** is being used for generating shadows of directional lights. Because of that we also have some properties like cascade size and frusta dist factor for directional light components. We can also adjust light intensity and the color of the light. The directional light component looks like the figure below in OP Editor:

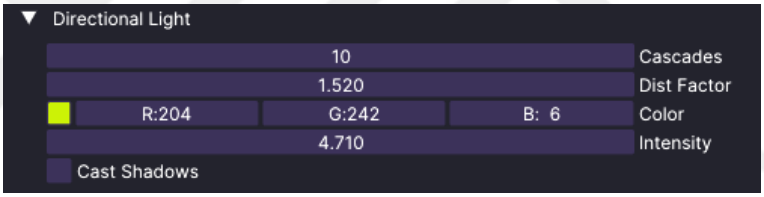


Figure 5.12: Directional Light Component

- **Spot Light Component:** Spot light component has properties of a spot light and it looks like the figure below in OP Editor:

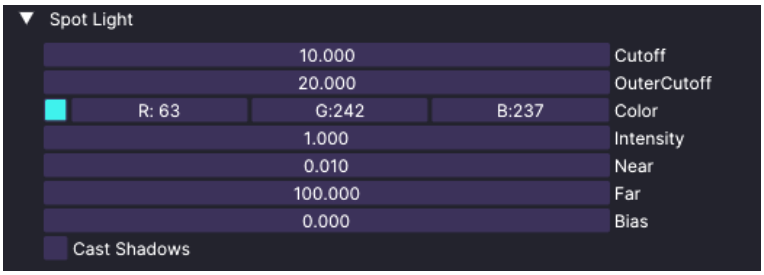


Figure 5.13: Spot Light Component

- **Point Light Component:** Every entity having a point light property is being

described with a **point light component** attached to it. The point light component looks like the figure below in OP Engine's GUI:

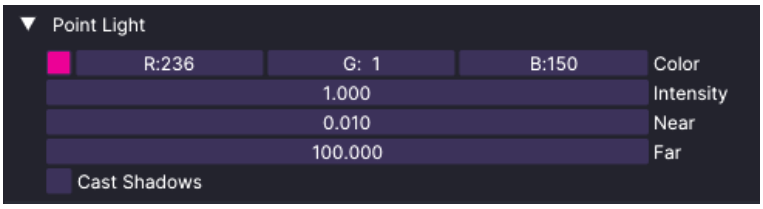


Figure 5.14: Point Light Component

- **Mesh Component:** Mesh component is added to the object if we want to display the object graphically in the scene. We select the mesh we want from the menu in OP Editor's GUI which is given in the figure below:

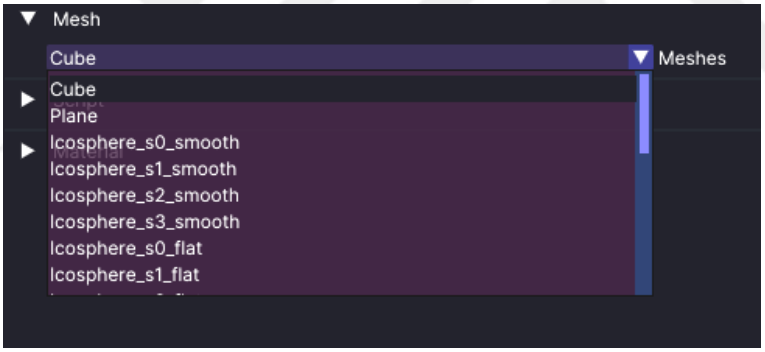


Figure 5.15: Mesh Component

As it is seen, the appropriate mesh can be selected from the drop-down menu.

- **Material Component:** Materials are connected to the shaders in game engines. This means that each material has a corresponding shader. We can define the materialistic properties of objects with this component. The material component looks like the figure below in OP Editor's GUI:



Figure 5.16: Material Component

In the figure above, the white box below the name **albedo map** is a drag-and-drop display for textures. We can drag and drop different textures not just for albedo, but other maps such as **normal maps**, **height maps**, **roughness**, **metallness** and **ambient occlusion maps**. For now, every material is directly connected to an **uber shader** which utilizes physically based rendering. But in future work, custom shaders can be also added to the materials for different visual behaviors.

- **Physics3D Material:** OP Engine deals with rigid body simulations in 3D environments. So for that reason, the objects which are going to be registered in the physical world have to have a component for determining their physical properties. These properties are:

  - Mass
  - Friction
  - Rolling and Spinning Friction
  - Restitution

  We also need to determine whether the object is **kinematic**, **dynamic**, or **static**. Lastly, we should also be able to fix rotations for some axes. In OP Editor's GUI, the **physics material** component looks like this:

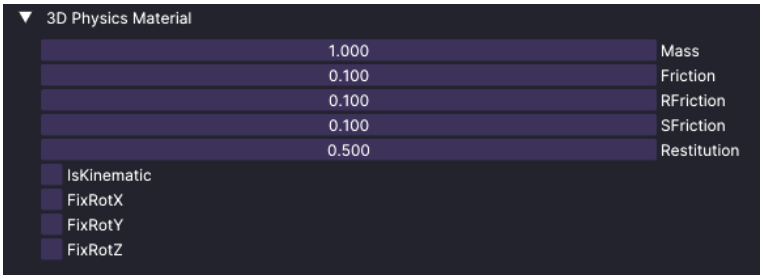


Figure 5.17: Physics 3D Material Component

- **Physics3D Collider:** Every physical object also needs a collider in the scene. In every time step, the physics manager checks for collisions according to the colliders provided by the engine. These colliders can be in various shapes in OP Engine. For now, OP Engine supports colliders with shapes:
  - Box
  - Sphere
  - Cylinder
  - Capsule

  We may also want to have colliders but instead of physically interacting with other colliders, only check whether they are colliding with them or not. For that, we also have a flag for a contact response. In OP Editor's GUI, physics 3d collider properties look like the figure below:



Figure 5.18: Physics 3D Collider Component

5.2 Scene Management

As stated in section 5.1, OP Engine uses the concept of ECS. In addition, Entities and Components are also needed to be processed and managed by a class. Since

every object is added to a certain scene, we have implemented this logic in **scene** class. Thus, the scene object is responsible for creating, removing, copying, and manipulating entities. In the main loop of the engine, the scene has two main modes which are:

- **Editor Mode:** In Editor Mode, simulation is idle and we only display objects in the scene just using the **SceneRenderer** singleton object which is responsible for rendering the scene. Scripts and the physics world do not work inside the editor mode. There is a special function called **OnUpdateEditor** and this function takes **timestep** and **editorCamera** objects as arguments and sends them to the **SceneRenderer**.
- **Runtime Mode:** Before starting the runtime mode, we need to make some changes to entities that have **script** and **physical** components. After preparation, when we update the scene in runtime mode, we also need to update script components and step physics world and render the scene from the eyes of the entity which has a primary camera component if it exists.

5.2.1 Callbacks for Nested Entities

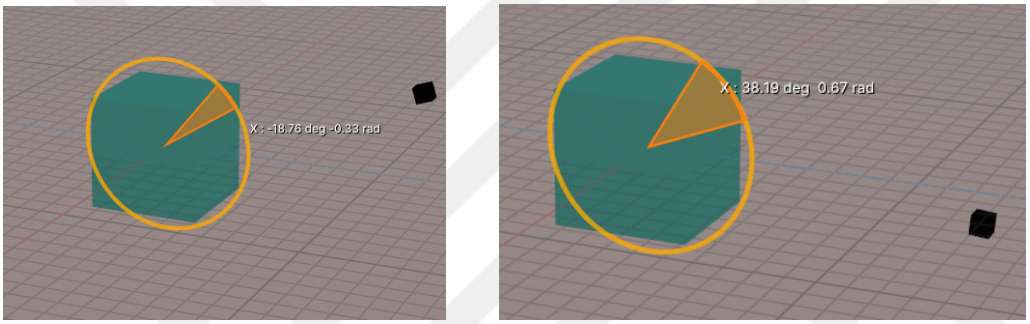
An entity's transform can be updated by scripts, physics interactions, or manually from OP Editor's GUI. However, sometimes we might have nested entities and if we update a parent's transform, we also have to update all of its children and grandchildren recursively. For that, we have to inform the entt registry that a change in a parent entity is happening.

For this, in **Scene** class, there is a function called `TransformChangeCallback` and this function is being called whenever a transform update happens for an entity. We bind this function to the system by using the **connect** function from the **entt** library. This callback function calls `UpdateTransforms` function of the entity and it also triggers a recursive change in its children.

One of the most useful cases where we use nested entities for transform updates is camera pivoting. We first create an empty entity and create another entity inside the

first entity. Secondly, we add a camera component to the second entity and change its location and orientation. If we change the orientation and location of the pivot point via scripting or manually from OP Editor, the entity with the camera component is also going to move according to its parent. This is mostly used for third-person cameras in games and simulations.

For example, let's think about a case where we have an entity with a cube turquoise mesh at the origin and there is also an empty entity at the origin which is the **pivot point**, we also add a new entity as the children of the **pivot point** entity. We also add a black cube mesh to the entity to see its position clearly. If we rotate the **pivot point** on **X** axis by a certain degree, the black cube entity will also be going to be affected. This change can be seen in the two figures below:



(a) Camera Pivoting 1 (b) Camera Pivoting 2

Figure 5.19: Rotating Pivot Point on X axis

From the two figures above, we can see that the turquoise cube does not rotate because it is another entity and we are only rotating the **pivot point** entity. The structure of the scene is seen in the figure below:

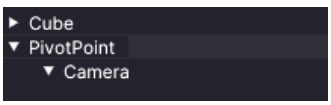


Figure 5.20: Camera Pivoting Scene Graph

This kind of nested entity structure can be used in a more exaggerated manner with more complex scene structures. We can also load models and add some parts of the

models as entities. OP Engine has experimental model loading features but this work is left for future studies.

5.2.2 Scene Utility Functions

While managing the scene, there are some functions that serve as utility functions implemented. These functions are:

- **Scene Name Getters and Setters:** We use these functions for manipulating the name of the scene
- **Skybox Getters and Setters:** Skyboxes are imported elements in a scene that provides the **environment light**. Skyboxes enhance the visual quality of the scene drastically. These functions help to use skyboxes in the scene.
- **Tonemap Getters and Setters:** Tone mapping is actually related to the rendering system. However, it is logically connected to the scene structure. Tone mapping is done at the end of the rendering step. We keep a flag as a member variable inside the scene class for determining whether we use tone mapping or not for the current time step. These functions are for manipulating tone mapping behavior.
- **Exposure Getters and Setters:** OP Engine uses HDR (High Dynamic Range) for the frames created. As a result, for some scenes where the luminance difference is really high, we might lose some of the details in places with very high or low luminance. Setting an exposure helps us to define which interval we need to focus on. If we lower the exposure, we will see darker places better and if we increase the exposure, lighter places will be seen easier.
- **CopyComponent:** When we change from runtime mode to editor mode or vice versa, we need to copy components from one entity to a new entity created in another scene as a clone. Copy component is a templated function that works for this purpose.
- **CopyScriptComponent:** Copying script components is a little bit different than copying other components. Because of that, there is a separate function

called **CopyScriptComponent** which specifically copies the script component.

- **Copy:** This function is for copying a scene into another scene. This function iterates all entities in the scene and runs **CopyComponent** functions for all of its components. This way, a scene is copied.

- **CreateEntity:** This function creates an empty entity having a root component and id component and transform component.

- **CreateChildEntity:** With this function, we can create entities inside other entities. These entities do not have a root component.

- **GetEntityWithUUID:** Sometimes, we might need to find Entities having a certain unique ID. For example, while traversing nested entities, we look at relationship components and use the id's to find the actual children entities.

- **AddChildEntity:** This function adds an entity that already exists as a child into another entity. If the child entity has former parents and siblings, the relationship components are all updated accordingly.

- **CreateEntityWithUUID:** We sometimes need to create entities with predefined unique IDs. For example, when we are deserializing a scene that we have saved before, the scene contains entities that already have unique IDs. If we would call **CreateEntity** functions, we would lose all the unique IDs. To prevent that, this function is being used while deserializing the scene.

- **DetachChild:** This function detaches the child entity and adds it to the root of the scene.

- **GiveAllChildren:** This function adds all children of an entity to a vector which is passed as a reference to the function.

- **RemoveEntity:** This function is used for removing entities from the scene.

- **OnViewportResize:** When a viewport resize event is triggered, this function is called and changes are made for the entities which have camera components. Because we need an aspect ratio while rendering the scene seen from the primary camera.

- **DuplicateEntity:** For the sake of simplicity, we might need to duplicate a certain entity. This function clones an existing entity and creates another one.

CHAPTER 6

OP ENGINE'S RENDERING SYSTEM

In chapter 5, **Scenes** and **Entity Component System** of OP Engine have been discussed. As a simulation engine, OP Engine relies on visual feedback for its simulations. This means that rendering the scene graph is important.

In OP Engine, Renderer System is responsible for visually describing the scene according to the properties of objects. In the core of the rendering system, there is a graphics API that handles all the commands for rendering. There are various graphics APIs and well-known ones are listed below:

- OpenGL [30]
- Metal [17]
- Vulkan [16]
- DirectX [35]

Each of these APIs has slightly different implementations and architectures. In some cases they are designed for specific platforms such as DirectX [35] is designed for Windows and Metal [17] for iOS platforms.

6.1 API Agnosticism

A fully functional game/simulation engine has to be platform-independent so that different APIs can be used for different platforms and the behavior of the renderer does not change. In OP Engine, only OpenGL[30] is being used but the software

architecture is built in a way that different APIs can also be used. This approach is called **API Agnosticism** which states that the engine is built in a way that it does not care which API is currently being used.

6.1.0.1 Abstracting API Commands

The first step to ensure API Agnosticism is abstracting the commands which have to use API-specific commands. For that, OP Engine uses two different classes for abstracting API commands. These are:

- RenderCommand
- RendererAPI

classes. This approach is inspired by Yan Chernikov's Hazel Engine [8]. RenderCommand class holds a RendererAPI object and they together invoke some basic functions. These are:

- **Init:** Initialization codes of different APIs are abstracted here. For example, in the OpenGL context, OP Engine enables blending mode and depth test in initialization.
- **SetViewport:** This function sets the viewport to the according size.
- **SetClearColor:** This function determines the color of the main framebuffer when it is cleared.
- **Clear:** This command clears the active framebuffer.
- **DrawIndexed:** This function and the next one are for drawing geometry. They use a **VertexArray** object as a reference.
- **DrawIndexBinded:** This function does the same thing just like the **DrawIndex** function but the difference is that the **VertexArray** object is bonded in the beginning and unbound at the end of the function.

- **Enable and Disable:** These two functions are for abstracting some commands which require enabling and disabling certain features such as **stencil** and **depth**.
- **DepthFunc:** We sometimes want to change how the API perceives depth values. We might want to show a certain object always without its depth values and testing it. For this purpose **DepthFunc** function is used.
- **PolygonMode:** This function determines how we render the geometry. It takes two parameters, one of them is **Face** and the other one is **Polygonmode**. The face has three modes and they determine which faces should we render. These three modes are:
  - Front
  - Back
  - Back and Front

  This is useful for backface culling. The **Polygonmode** has also three different modes and they determine how we represent the polygons. These are:
  - Point
  - Line
  - Fill

6.2 Structure of OP Rendering System

In the Rendering System, API Agnosticism is implemented as described in 6.1. With that approach, the components of the rendering system have merged together. In this section, each component will be explained in the order below:

- Abstract Class
- Corresponding OpenGL Class

As seen above, if we want to add different APIs rather than OpenGL in the future, the only thing that has to be done is to add a new class that uses the new API which is derived from the Abstract Class.

6.2.1 Buffers

In the graphics world, we need to prepare some data to be transferred into GPU's memory. We manage this by holding these data inside buffers. In OP Engine, there are two buffers which are:

- **Vertex Buffers:** They carry information about each vertex. They might have different layouts such as position, normal vector, and texture coordinates.
- **Index Buffers:** Index buffers hold the order of vertices. This order determines how a primitive (such as a triangle) is formed. Index buffers are also an optimization because it prevents redundant repetitions of the same vertices.

6.2.1.1 OpenGL Vertex Buffer

In OpenGL, vertex buffers are created in a straightforward manner. We have to store a variable to reference the buffer whenever we need it. This reference is an unsigned integer value given by OpenGL after creating the buffer. This value is being held as a member variable in VertexBuffer class.

OP Engine allows the creation of Vertex Buffers as follows:

```
OpenGLVertexBuffer(uint32_t size)
{
    glCreateBuffers(1, &id);
    glBindBuffer(GL_ARRAY_BUFFER, id);
    glBufferData(GL_ARRAY_BUFFER, size, nullptr, GL_DYNAMIC_DRAW);
}
```

In the snippet above:

- **glCreateBuffers:** This command creates the buffer and assigns it to the unsigned integer **id** which is passed as a reference
- **glBindBuffer:** This command binds the buffer which we pass as an argument (which is **id** in this case) to the **Array Buffer** which is bind before the creation of the buffer. From that time on, this buffer belongs to the **Array Buffer**.

- **glBufferData:** This command fills the buffer that is bind to the **Vertex Array**. The code actually allocates memory for the buffer. **GL_DYNAMIC_DRAW** indicates that this buffer might change during the runtime so it stores the data accordingly.

6.2.1.2 OpenGL Index Buffer

In OpenGL, index buffers are also created easily. Just like any buffer object, an id is stored in the **Index Buffer** class. Index buffers are created in OpenGL as follows:

```
OpenGLIndexBuffer(uint32_t size)
{
    glCreateBuffers(1, &id);
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, id);
    glBufferData(GL_ELEMENT_ARRAY_BUFFER, size, nullptr, GL_STATIC_DRAW);
}
```

- **glCreateBuffers:** This command creates the buffer and assigns it to the unsigned integer **id** which is passed as a reference. This is the same as **Vertex Buffers**
- **glBindBuffer:** This command binds the buffer which we pass as an argument (which is **id** in this case) to the **Array Buffer**'s Element buffer which is bound before the creation of the buffer. From that time on, this buffer belongs to the **Array Buffer**. This is also not different from **Vertex Buffers**.
- **glBufferData:** This command fills the buffer that is bind to the **Vertex Array**. The code actually allocates memory for the buffer. **GL_DYNAMIC_DRAW** indicates that this buffer might change during the runtime so it stores the data accordingly.

6.2.2 Vertex Array

Vertex Array is an object which holds **Vertex** and **Index** buffers which are explained in 6.2.1. They are created just like **Vertex Buffers** and **Index Buffers**. In OpenGL,

Vertex Arrays are created as follows:

```
OpenGLVertexArray(uint32_t size)
{
    glCreateVertexArrays(1, &id);
}
```

This way a Vertex Array object is created inside the program. A Vertex Array object has two important functions which are **AddVertexBuffer** and **AddIndexBuffer**. These functions are essential to get Vertex Arrays working in the first place. As their names suggest, these functions bind Vertex and Index Buffers to the Vertex Array. But before doing that, there is also one thing that we have to take care of, which is the **Layout** of the data in Vertex Buffer.

6.2.2.1 Vertex Buffer Layout for Vertex Array

Before adding the Vertex Buffer to the Vertex Array, we have to determine the layout of its data. For example, let's think about a vertex buffer where we store:

- Positions as Vec3
- Normals as Vec3
- Texture Coordinates as Vec2

Vec3 is a vector with three floating point elements and **Vec2** is a vector with two floating point elements. While defining the layout, we have to look at the sizes of each component and calculate the total **stride** and **offsets** of every component. **Stride** determines how many bytes we have to pass in order to get into the territory of the next vertex. Moreover, **offsets** determine how many bytes we have to move forward from the beginning of the vertex in order to reach the territory of the corresponding component. The figure below visually explains this layout:

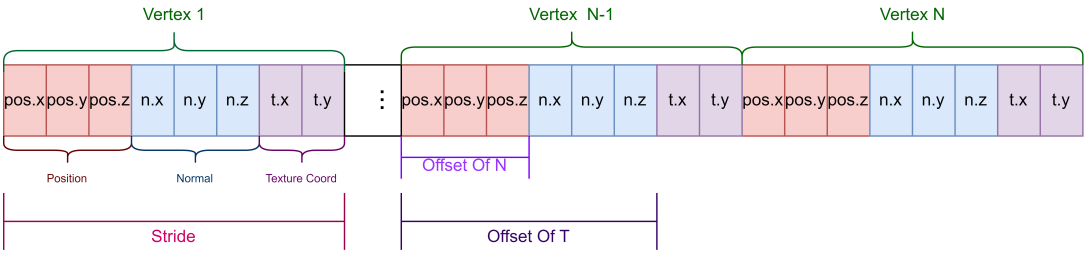


Figure 6.1: Vertex Buffer Layout

In the layout described above, we assume that fs is the size of a floating point and it is **4 bytes**. So according to this assumption:

- Stride is $3 * fs + 3 * fs + 2 * fs = 32$ bytes.
- Offset of Position is 0 bytes.
- Offset of Normal is $3 * fs = 12$ bytes.
- Offset of Texture Coordinate is $3 * fs + 3 * fs = 24$ bytes.

This layout information is given to the Vertex Array and Vertex Array applies it so that **Shaders** which work with this data fetch true values. In OpenGL, this is being applied to position as follows:

```
/* For Position Values  */
glEnableVertexAttribArray(0)
glVertexAttribPointer(0,
                      3,
                      GL_FLOAT,
                      GL_FALSE,
                      32,
                      (const void*)0);
```

Normal values are also added accordingly:

```
/* For Normal Values  */
glEnableVertexAttribArray(1)
```

```
glVertexAttribPointer(1,
                      3,
                      GL_FLOAT,
                      GL_FALSE,
                      32,
                      (const void*)12);
```

And lastly, texture coordinates are added with:

```
/* For Texture Coordinate Values  */
glEnableVertexAttribArray(2)
glVertexAttribPointer(2,
                      2,
                      GL_FLOAT,
                      GL_FALSE,
                      32,
                      (const void*)24);
```

The **glEnableVertexAttribArray** function is for determining which element we are enabling at that moment to fill in. In the example layout above, **Position** is the first element so its index is **0**. So, **Normal**'s index is **1** and **Texture Coordinate**'s index is **2**.

glEnableVertexAttribPointer is the function where we apply the layout that we have defined in the **Vertex Buffer** object. Its parameters mean (first to the last parameter in order):

- **Index:** Index of the element
- **Component Numer:** How many components are in that element? In the case above, **Position** and **Normal** have 3 components and **Texture Coordinate** has 2 components.
- **Component Type:** What is the type of the components? In the case above, it is **GL_FLOAT** for all elements.
- **Normalize:** This parameter is for determining whether OpenGL should apply normalization on the data or not. In the case above, there is no need for a normalization so it is set to **GL_FALSE**.

- **Stride:** This parameter is for **stride**. As explained above, **stride** is for determining how many bytes we have to move forward to reach the next vertex data. In the case above, it is **32** bytes, the full size of one vertex.

- **Offset:** We define how many bytes we have to move forward to reach the data of the corresponding component by setting the **offset value**. In the case above, it is **0** for **Position**, **12** for **Normal** and **24** for **Texture Coordinate**.

6.2.3 Texture

Textures are essential in the Graphics World. They are used for various purposes. Sometimes, procedural textures are created for certain effects, sometimes they enhance the visual quality of the rendering with techniques such as **normal mapping** and **parallax mapping**. They are also used to store information about the scene to be used in later stages of the rendering. Another use case of Textures is creating environment maps. Textures can be 3D or 2D and can be defined as **Arrays**.

In OP Engine, Textures have certain functions such as:

- **GetWidth:** Width of the texture is returned.

- **GetHeight:** Height of the texture is returned.

- **SetData:** Data of the texture is being set with this function.

- **Bind:** The texture is enabled with this function.

Normally, textures are reached inside shaders via a **sampler**. For 2D Textures, we give the sampler a **texture coordinate** (x,y), and for 3D Textures, we give a vector (x,y,z). And then, the sampler yields the pixel at that coordinates. There are some parameters we give while creating textures. For example, in OpenGL, while creating a texture, we give parameters such as:

- **GL_TEXTURE_WRAP_S:** This parameter is for determining how the sampler is going to work when a texture coordinate's x value is greater than 1.0.

- **GL_TEXTURE_WRAP_T:** This parameter determines how the sampler is going to work in the shader if a texture coordinate's y value is greater than 1.0.
- **GL_TEXTURE_MIN_FILTER:** This is the minifying filter. When the fragment that is going to be renderer with a certain texture is smaller than the texture itself. The sampler works according to this parameter. The parameters are:
  - **GL_NEAREST:** This yields the pixel whose coordinates are the nearest.
  - **GL_LINEAR:** This parameter tells the sampler to perform a linear interpolation with the neighboring texture elements.

  Mipmaps are also can be used with this parameter. Mipmaps are smaller versions of a texture that can be generated while creating the texture. Normally mipmaps are half of the size of the normal texture. Mipmaps can be generated more than 1 time. For example, for an image that has 512x512 size, mipmaps can be like follows:
  - 1st mipmap 256x256
  - 2nd mipmap 128x128
  - 3nd mipmap 64x64
  - 4th mipmap 32x32

  If we want to use mipmap for the minifying filter:
  - **GL_NEAREST_MIPMAP_NEAREST:** With this parameter, the sampler gives the pixel from the closest mipmap texture and yields the pixel of the nearest texture element.
  - **GL_LINEAR_MIPMAP_NEAREST:** The sampler applies a linear interpolation using the nearby mipmaps and from that resulting texture, it gives the nearest texture element
  - **GL_NEAREST_MIPMAP_LINEAR:** This parameter suggests that sampler should pick the nearest mipmap and apply linear filtering using linear interpolation to the resulting texture.
  - **GL_LINEAR_MIPMAP_LINEAR:** With this parameter, the sampler first applies a linear interpolation to nearby mipmaps and from that texture, it again applies a linear interpolation to the nearby texture elements

to yield the final pixel value. This parameter is the slowest one but it gives realistic results while using **texture mapping**.

As an example, this is how a texture is created using OpenGL in OP Engine. In this example, the texture has 3 channels (red-green-blue) and the values are 8-bit unsigned integers which means red, green, and blue elements can have values between 0-255:

```
/* Texture Creation  */
glGenTextures(1, &id);
glBindTexture(GL_TEXTURE_2D, id);
glTexParameteri(GL_TEXTURE_2D,
                GL_TEXTURE_WRAP_S, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D,
                GL_TEXTURE_WRAP_T, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0,
             GL_RGB, GL_UNSIGNED_BYTE, data);
```

In the example code snippet above. The created texture repeats itself if the x or y values of the coordinates are greater than 1 and the sampler fetches the nearest pixel according to these coordinates. With **glTexImage2D** data is loaded into the texture and we also specify:

- **Type of Texture:** We are using a 2D Texture. So it is GL_TEXTURE_2D in this case.
- **Level:** Defines the mipmap level. In this case, it is 0.
- **Internal Format:** This is for defining how many channels we want in the data. It is GL_RGB in this case.
- **Width and Height:** The next two parameters are width and height.
- **Border:** This parameter is for enabling the border if we want. In this case, there is no border.
- **Format:** This is for defining the format of the image that we are using. It is GL_RGB in this case.

- **Type:** With this parameter, we define what is the data type of the pixels in the image. In our case, we are using 8-bit unsigned integers so it is going to be GL_UNSIGNED_BYTE.

6.2.4 Framebuffer

Framebuffers are structures that hold data about the scene. They normally use **textures** as attachments. For example, if we look at the viewport in OP Engine, the visual representation of the scene can be seen. This representation is actually taken from **main framebuffer**'s texture attachment. In addition, **depth** and **stencil** values are also stored in framebuffers and they can be reached.

Another importance of framebuffers is that we can save the intermediate results from various framebuffers and then use them in other stages of the rendering. Effects like shadows and post-processing are done by rendering the scene into some framebuffers and then using their attachments as textures in different **render passes**.

In OP Engine there are various texture formats for different purposes and while creating a framebuffer, texture formats of the attachments are also given. The most used texture formats of OP Engine are as follows:

- **RGBA8:** This texture holds red, green, and blue values as 8-bit unsigned integers. Its main purpose is to hold the final rendered image.

- **RGBA32F:** This texture holds red, green, and blue values as 32-bit floats. They are normally used for intermediate rendering results where we deal with **high dynamic range**.

- **RED_INTEGER:** This texture is mainly used for mouse picking. Inside this texture, there is only one component for each pixel and these components are 32-bit integer values. In each pixel, the id of the corresponding element is written in a separate rendering without lighting calculations. When the user clicks on an object in the scene, this texture is read and the corresponding entity is selected.

- **DEPTH24STENCIL8:** This is a standard attachment. It tells graphics API that the depth attachment which is being used has 32 bits for each pixel, 24 of that is for depth and 8 is for the stencil. The depth part is used for depth testing and the stencil is used for stencil testing.
- **SHADOWMAP_ARRAY_DEPTH:** This attachment is used for shadow mapping. OP Engine supports shadow mapping with multiple lights so that each **spot** and **directional** light has its own depth texture inside a texture array.
- **SHADOWMAP_OD_ARRAY_DEPTH:** This texture is also for storing multiple lights' depth textures. But this time it is for **point lights**. The reason why **point lights** are treated separately is they are **omnidirectional**. This means for each point light, we have to store **6** different shadow map textures and each of them resembles a direction in 3D space.
- **SM_POINT_LIGHT_BLUR:** This texture is used as an intermediate step for creating smoother shadows for **point lights**. It is produced from a raw shadow map coming from a **point light**.

As an example, a framebuffer with **RGBA8** and **DEPTH24STENCIL8** attachments is created as the following in OpenGL:

```
/* Framebuffer Creation */
glCreateFramebuffers(1, &id);
glBindFramebuffer(GL_FRAMEBUFFER, id);
/*------ RGBA8 ATTACHMENT ------*/
glCreateTextures(GL_TEXTURE_2D, 1, texid);
glBindTexture(GL_TEXTURE_2D, texid);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA8, width, height,
             0, GL_RGBA, GL_UNSIGNED_BYTE, nullptr);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_R, GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
glFramebufferTexture2D(GL_FRAMEBUFFER, 0, GL_TEXTURE_2D, id, 0);
/*-------- DEPTH24STENCIL8 ATTACHMENT ---------*/
glCreateTextures(GL_TEXTURE_2D, 1, texid2);
```

```
glBindTexture(GL_TEXTURE_2D, texid2);
glTexStorage2D(GL_TEXTURE_2D, 1, GL_DEPTH24_STENCIL8, width, height);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_R, GL_CLAMP_TO_BORDER);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_BORDER);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_BORDER);
float borderColor[] = { 1.0, 1.0, 1.0, 1.0 };
glTexParameterfv(GL_TEXTURE_2D, GL_TEXTURE_BORDER_COLOR, borderColor);
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_DEPTH_STENCIL_ATTACHMENT,
                       GL_TEXTURE_2D, id, 0);

GLenum buffers[1] = { GL_COLOR_ATTACHMENT0 };
glDrawBuffers(1, buffers);
glBindFramebuffer(GL_FRAMEBUFFER, 0);
```

As it is seen in the above code snippet, first the framebuffer is created and bound, and then textures for color and depth_stencil attachments are created and bound to the framebuffer. And then color attachments that are created are registered as draw buffers so that shaders can write into them without any problems. And at the end, we disable the framebuffer which is a good practice. In OP Engine, this is how a framebuffer is created.

There are certain rules while creating a framebuffer. In OpenGL, if an array texture is used as an attachment, then every attachment must be an array attachment. The integrity of the framebuffer is checked with **glCheckFramebufferStatus** function. If this function does not return **GL_FRAMEBUFFER_COMPLETE**, then it means there is a problem while creating the framebuffer. Rules for attachment completes are found in khronos wiki [36].

There can also be some framebuffers where we do not need any color attachments. This is normally done for shadow mapping but OP Engine uses a different shadow mapping technique for now so there are no framebuffers without color attachments for now. But in future studies, this is going to change.

6.2.5 Uniform Buffer

There are variables that are sent to shaders during runtime. These are called **uniforms**. Uniforms are mainly **matrices** or **vectors**. They can also be materialistic properties. However, if we want to use uniforms, we have to send them to every shader separately. This is a tedious job. In addition, it requires a lot more time because we have to minimize the number of API calls for performance.

To eliminate this problem, **Uniform Buffers** are being used. Uniform buffers have a **SetData** function where we set the data we want to send. While we create them, we also give the size of the block we want to send. In OpenGL, OP Engine creates a uniform buffer like the following:

```
/* Uniform Buffer Creation */
glCreateBuffers(1, &id);
glNamedBufferData(id, size, nullptr, GL\_DYNAMIC\_DRAW);
glBindBufferBase(GL\UNIFORM\_BUFFER, binding, id);
```

The code snippet above creates the buffer, allocates its memory, and determines a **binding**. This binding will be able to access from every shader working in the program. Normally **GL_DYNAMIC_DRAW** is used because uniform buffers are changed a lot of times during the runtime.

As an example from OP Engine, **transform data** is used as a uniform buffer object because it is widely used by multiple shaders at the same time. For creating a uniform buffer holding **transform data**, we first create a struct like the following;

```
struct TransformData
{
    glm::mat4 Model;
} TransformBuffer;

UniformBuffer TransformUniformBuffer;
```

In the above code, a struct for holding transform data is defined and it has a single 4x4 matrix which is the model matrix. And then we have also declared a Uniform Buffer

object. Secondly, when **renderer** is initialized, this **TransformUniformBuffer** is being created as follows:

```
TransformUniformBuffer = UniformBuffer::Create(sizeof(TransformData), 2);
```

After that, when we want to update the buffer, we use the **SetData** function as follows:

```
TransformUniformBuffer->SetData(&TransformData, sizeof(TransformData));
```

And lastly, in the shaders, we also have to define the same structure. In GLSL, it is achieved like this:

```
layout(std140, binding = 2)  uniform Transform
{
mat4 u_Model;
};
```

In the glsl code above, we define a struct that has a **std140** layout and binding on **2**. After defining this, the 4x4 transformation matrix is achievable from all of the shaders if the code above is included. In OP Engine, some of the most used uniform buffers are:

- **Camera:** Camera is used as a uniform buffer. It contains view and projection matrices and some other important values such as near and front values.
- **DirLightData:** Multiple directional lights are being held as arrays in uniforms buffers objects. But of course the amount of allowed directional lights are fixed.
- **SpotLightData:** Multiple spot lights are handled with arrays in uniform buffer objects just like directional lights.
- **PointLightData:** Same as directional and spot lights, point lights are also held within arrays in uniform buffer objects.

There are also other uniform buffer objects which are going to be discussed in later parts.

6.2.6 Shader

Shaders are programs that run on the **GPU** and they do specific operations on data they receive from the **CPU**. There are different types of shaders. The most used ones are as follows:

- **Vertex Shader:** Inside the vertex shader, the data of a vertex is processed. Normally vertices are multiplied with model, view, and projection matrices to simulate the camera behavior and the object's position, orientation, and scale in 3D space.
- **Fragment Shader:** Inside the fragment shader, we deal with the clipped version of the scene according to the viewport. According to the polygon mode which is explained in 6.1.0.1, **rasterizer** which works just before **fragment shader** is being run and fills the parts between the vertices. The parts we have at the end are called **fragments**. So, **fragment shader** deals with the **fragments** that are created by the **rasterizer**. In the fragment shader, normally, lighting operations are handled.
- **Tesselation Shader:** Tesselation shaders create new vertex points between already existing vertices. They are mainly used for creating detailed geometry for terrains which is a great optimization. Because if a terrain is zoomed out, the amount of tesselation will decrease and the **GPU** will not be able to deal with a lot of vertex data. **Tesselation Shaders** are currently not used in **OP Engine**
- **Geometry Shader:** Geometry shaders add new primitives to the geometry. In OP Engine, they are widely used for creating environment maps. They are also used for creating shadow maps for multiple lights with **layered rendering**.
- **Compute Shader:** Sometimes, developers might need to make some calculations on the GPU because their algorithms allow them to run highly parallelized code. They work great with lighting algorithms such as **Forward Plus Rendering** [37]. However, **Compute Shaders** are not currently supported in OP Engine, but in future studies, it is planned to be supported as well.

Except for the **Compute Shader**, these shaders are actually representing a certain

stage in the **Graphics Pipeline**. In the following figure, the flow of the stages according to the graphics pipeline can be seen. The red ones must be programmed in order to create a shader. The others are optional:

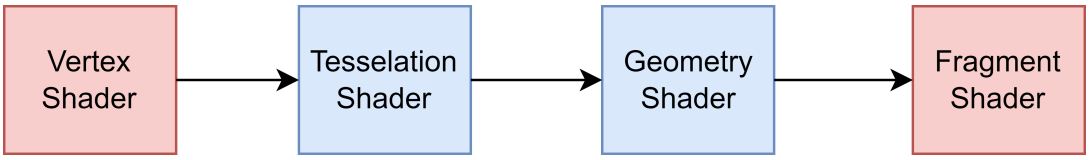


Figure 6.2: Flow of the Programmable Shader Stages

In the current stage of OP Engine, **Compute Shaders** and **Tesselation Shaders** are not used. But in future studies, they are also going to be included.

6.2.6.1 Shader Compilation and Struture

In Hazel Engine [8], Yan Chernikov has developed a way for writing shader code which is going to be used in the same shader program by defining some tokens. For example, let's think about a shader program where only **vertex shader** and **fragment shader** are being used. With this approach, the code of **vertex shader** and **fragment shader** are written in the same source file. After that, this file is read by the **shader compilation system**. This system differentiates **vertex** and **fragment** shaders and adds them into the newly created shader object which contains the program.

In OP Engine, a shader creation follows the following steps:

- Parsing the source file. OP Engine supports **vertex**, **fragment**, and **geometry** shaders. In the same source file, they are separated with **#type** token. For example, if we want to specify a **vertex shader** part in the source code, we write **#type vertex** at the beginning of that line. After that, the shader compilation system assumes that everything which is written under that line contains **#type** token until another **#type** token is read belongs to the **vertex shader**. After that, these separated sources are saved into an **unordered map**.
- Separated sources are compiled using **shaderc**[32] and their binaries are saved into a cache. If there are already shader binaries that are saved in the cache,

the system does not compile that shader again, instead, it uses the precompiled binary. This works as a good optimization.

- After binaries are acquired, they are linked to the shader program that is being created.

In the subsection 6.2.5, it was explained that **uniform buffers** are structs that can be reached from every shader. This was one way of sharing data between the CPU and the GPU. In addition to that, we can also add uniforms to each shader. These uniform variables are specific for each shader. They are mainly used for textures that are going to be used by objects or material properties.

The shader class supports setting **Integer**, **Integer Array**, **Float**, **Float2**, **Float3**, **Float4** and **Mat4** uniform variables. We can either add these uniform variables by their name inside the shader or by their binding locations. However, since OP Engine uses **shaderc** for compilation and turns source code into binaries, and then links the binaries, it is impossible to use them with their names. Thus, their binding locations are used.

For example, uniform variables are declared as below in OP Engine's shader system:

```
layout (location = 0) uniform float u_Roughness;
layout (location = 1) uniform float u_Metalness;
layout (location = 2) uniform vec3 u_Albedo;
layout (location = 3) uniform float u_TilingFactor;
layout (location = 4) uniform float u_HeightFactor;
layout (location = 5) uniform int u_ClipBorder;
```

As seen above, the naming convention for uniform variables is that their names start with **u_**. We access these uniforms and change them in the shader by using their **location** numbers. For example, if we want to change **u_Metalness** variable to the value of **0.77f**, we have to call:

```
shader->SetFloat(1, 0.77f);
```

This way we efficiently update the values for all uniform variables in the shader.

6.2.6.2 Shader Include System

Just like in languages such as C++ and Python, shader languages do not have a standard include system. Most of the time, we need a way to reorganize and reuse the code segments that we have written over and over again. For example, we might write some utility functions and use them in different shaders. Or some global variables and uniform buffers. Without a proper **include system**, the same code will be written over and over again. OP Engine uses a custom designed and recursive include system for writing more organized shader code. In the source files, **#include** keyword can be used for including the necessary file. If that included file contains **#include** keywords as well, it recursively resolves the code and returns the final source code.For example, in a file called **Defines.glsl**, we have written some definitions:

```
#define MAX_DIR_LIGHTS 2
#define MAX_SPOT_LIGHTS 4
#define MAX_CASCADE_SIZE 15
#define MAX_POINT_LIGHTS 10
#define MAX_BONES 10
#define MAX_BONE_INFLUENCE 4
```

If we include this Defines.glsl like the following:

```
// ------------------ DEFINES ----------------- //
#include Defines.glsl
// -------------------------------------------- //
```

The final code will be resolved into:

```
// ------------------ DEFINES ----------------- //
#define MAX_DIR_LIGHTS 2
#define MAX_SPOT_LIGHTS 4
#define MAX_CASCADE_SIZE 15
#define MAX_POINT_LIGHTS 10
#define MAX_BONES 10
#define MAX_BONE_INFLUENCE 4
// -------------------------------------------- //
```

With this system, shaders are written a lot more organized and it saves development time. As an example, OP Editor currently uses the shader source structure like the figure below:

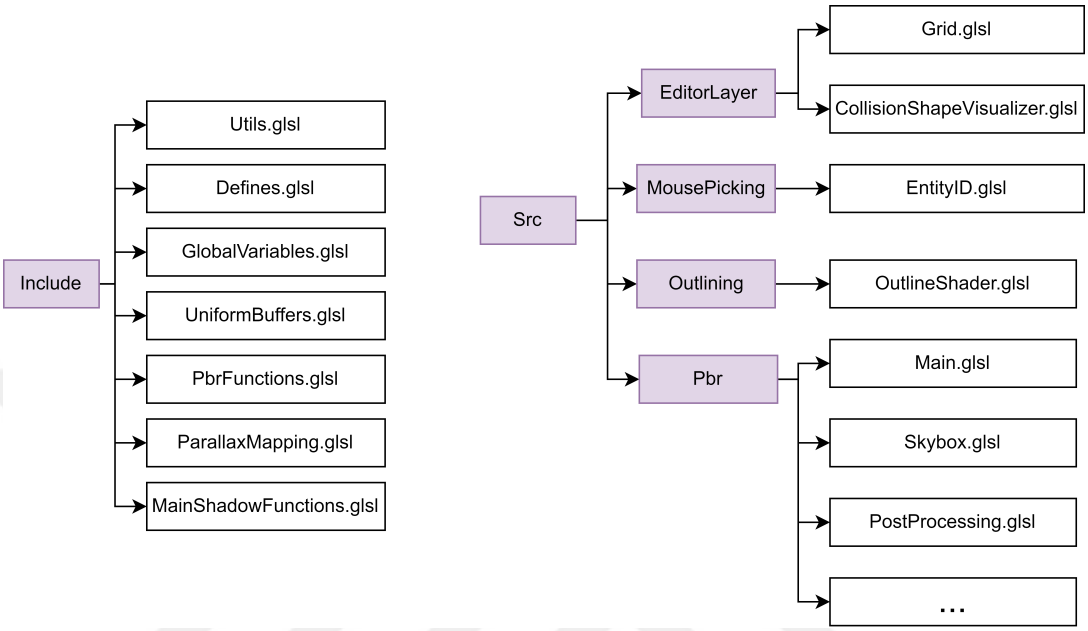


Figure 6.3: OP Engine's Shader Source Code Structure

In the figure above, **Purple** boxes represent directories and other boxes represent source files. In this structure, code is used in multiple files in the **Src** directory. Every **.glsl** file is used for different reasons in the engine. For example, in **EditorLayer** directory, we have **Grid.glsl** which is responsible for creating the grid that can be seen in figure 5.19a. In the **Outlining** directory, we have a shader for writing outlines with the name **OutlineShader.glsl**. And this shader is responsible for creating outlines when a certain entity has an outline component which is shown in figure 5.5.

In the **Include** directory, we can see that included files are also organized according to their use cases. For example, **GlobalVariables.glsl** has some variables that we might need in our source files. Moreover, inside **UniformBuffers.glsl** there are all uniform buffers that can be accessed from every shader. This file is also included in all of the source files where we use one or more uniform variables that are defined.

6.2.7 Overall Structure of Rendering System

In the last subsection of the Rendering System structure, a class diagram of the overall rendering system structure can be seen in the figure below:

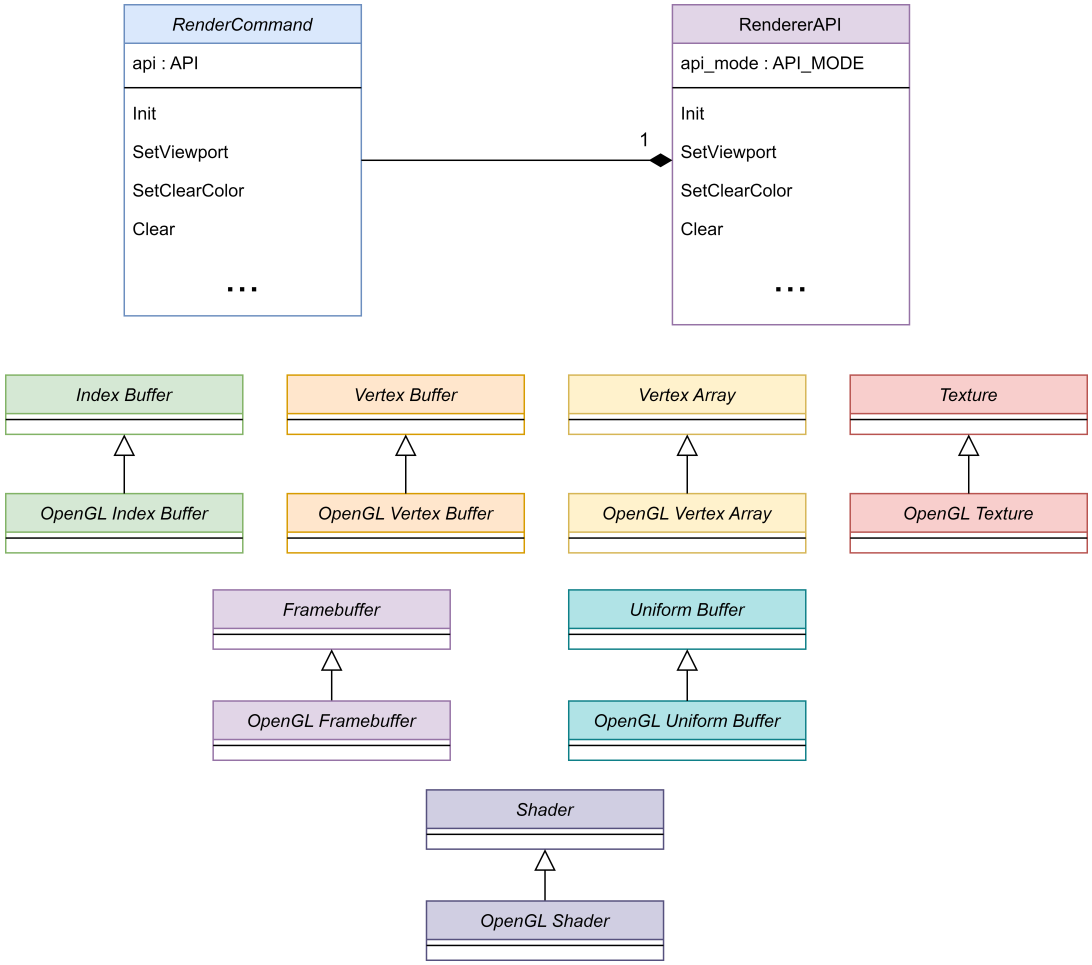


Figure 6.4: Rendering System Structure

This figure explains the **API Agnosticism** which is explained in subsection 6.1. If we want to add different graphics APIs such as DirectX [35], all we need to do is write classes named like **DirectXFramebuffer** etc. and make them derived from it superclass **Framebuffer**. This way, the renderer will only call commands using virtual methods and we will not need to change any other code.

6.3 Meshes and Geometry

OP Engine has its own way of handling meshes. Meshes are a set of vertices and each vertex has its own properties. In OP Engine, a mesh vertex has the following properties which are passed to the layout of the **Vertex Buffer** as it is explained in 6.2.2.1.

The layout of the vertex data for all meshes is as follows:

- Position which is a 3-float Vector
- Normal vector which is a 3-float Vector
- Texture Coordinate which is a 2-float Vector
- Tangent vector which is a 3-float Vector
- Bitangent which is a 3-float Vector
- BoneID which is a 4-integer Vector
- Bone Weights which is a 4-float Vector

In this layout, Bone ID and Bone Weights are for skeletal animations of models. This feature is currently under work in OP Engine and it is not included in the thesis work because of that. Position, Normal and Texture Coordinate values are straightforward and they are for calculating fragment positions and the pixel from the texture to be filled. In addition, Tangent and Bitangent vectors are also calculated before registering the Vertex Array.

Currently, OP Engine supports primitive meshes such as **Capsule**, **Cube**, **Cylinder**, **Radial Sphere**, **Icosphere**, **Plane**, **Quad** and **Skybox**. OP Engine also supports model loading using Assimp Library [26], however, this topic is beyond the scope of this thesis. All these classes are derived from the base **Mesh** class.

Moreover, except the **Model** class, all of them are created by calculating the vertex points, normals, texture coordinates, tangent, and bitangent values according to their shapes in their constructors. The **Model** class loads mesh files such as **.mtl** or **.obj**.

6.3.1 Tangents and Bitangents

Tangents are bitangents are mainly used for creating a coordinate space at a fragment part for calculating the final normal value of the fragment if we are using **Normal Mapping**. They are created using the texture coordinates and positions of three triangle vertices.

As seen in the figure below, if we look at the tangent plane on a point in a sphere's surface, it is seen that we can create an infinite amount of **Tangent** and **Bitangent** vectors which are perpendicular to each other:

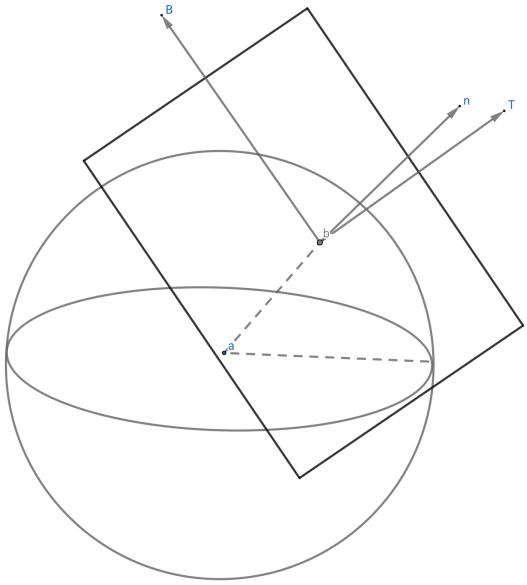


Figure 6.5: Tangent of a Surface

The goal is to calculate the Tanget and Bitangent vectors that we need. In our case, they have to match with the **U** and **V** of the **Normal Map** texture. In mesh data, our vertices form primitives which are triangles. Triangles have three vertex positions and each vertex has a normal. In the graphics pipeline, the **rasterizer** calculates the normal vector of the fragment which is inside the primitive by **interpolation**. Whether it is interpolated or not, they all reside in the same plane, so their tangents and bitangents must be the same.

Thus, we can extract Tangent and Bitangent information from 3 points of a triangle.

We know the:

- Position of each vertex in 3D Space $P1(x, y, z)$, $P2(x, y, z)$ and $P3(x, y, z)$
- Their texture coordinates $Pt1(u, v)$, $Pt2(u, v)$ and $Pt3(u, v)$

This primitive is illustrated in the figure below:

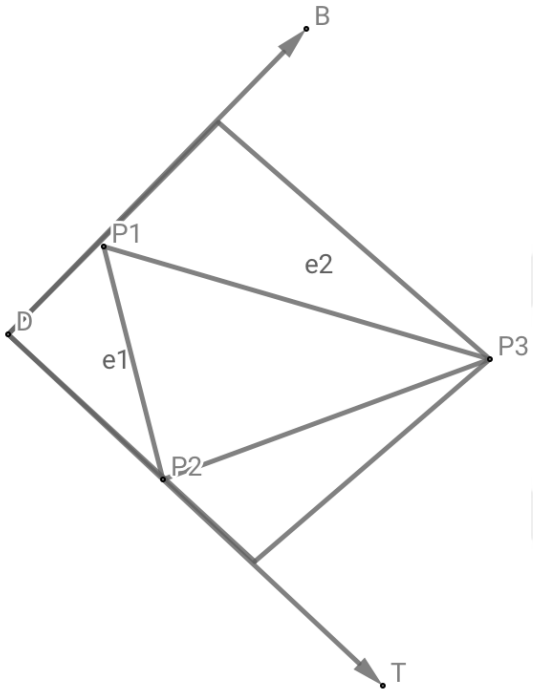


Figure 6.6: Primitive Tangent

We first have to write edges $e1$ and $e2$ in terms of T and B. The tangent vector T is associated with the u axis of the texture map and the bitangent vector B is associated with the v axis. By using this knowledge, we can derive $e1$ and $e2$ such as:

$$e1 = (Pt2_u - Pt1_u)T + (Pt2_v - Pt1_v)B \tag{6.1}$$

$$e2 = (Pt3_u - Pt1_u)T + (Pt3_v - Pt1_v)B \tag{6.2}$$

We already know real values of e_1 and e_2 because of their position values in 3D Space $P_1(x, y, z)$, $P_2(x, y, z)$ and $P_3(x, y, z)$. So, the only unknown variables in the

equations above are T and B. The above equation resolves into the linear system below:

$$\begin{bmatrix} e1_x & e1_y & e1_z \\ e2_x & e2_y & e_2z \end{bmatrix} = \begin{bmatrix} Pt2_u - Pt1_u & Pt2_v - Pt1_v \\ Pt3_u - Pt1_u & Pt3_v - Pt1_v \end{bmatrix} * \begin{bmatrix} T_x & T_y & T_z \\ B_x & B_y & B_z \end{bmatrix} \tag{6.3}$$

In order to solve this system, we can take the inverse of the 2x2 matrix and multiply it by the left of each equation. As a result, this gives us the matrix which contains T and B vectors. In order to calculate the inverse of a 2x2 matrix, we have to calculate its determinant first. The determinant D is as follows:

$$D = (Pt2_u - Pt1_u) * (Pt3_v - Pt1_v) - (Pt3_u - Pt1_u) * (Pt2_v - Pt1_v) \tag{6.4}$$

Thus, the inverse of the matrix is:

$$\frac{1}{D} * \begin{bmatrix} Pt3_v - Pt1_v & Pt1_v - Pt2_v \\ Pt1_u - Pt3_u & Pt2_u - Pt1_u \end{bmatrix} \tag{6.5}$$

Let's call this matrix in the above equation A^{-1}. So our system becomes:

$$\begin{bmatrix} T_x & T_y & T_z \\ B_x & B_y & B_z \end{bmatrix} = A^{-1} * \begin{bmatrix} e1_x & e1_y & e1_z \\ e2_x & e2_y & e_2z \end{bmatrix} \tag{6.6}$$

By using this, we get the T and B vectors.

After calculating T and B vectors while constructing meshes, we also have to re-orthogonalize T and B vectors according to the normal vector we have for precision errors. They might introduce some artifacts in rendering if we are using a more complex mesh. This is done via **gram-schmidt** orthogonalization which is explained in Yousef Saad's book about iterative methods for sparse systems [38].

In OP Engine, this is used with the following code snippet:

```
tangent = normalize(tangent - normal * dot(normal, tangent));
```

normalize and **dot** are functions that are defined in glm library [22]. After calculating tangents and bitangents, all of the data is sent to the vertex array object that is dedicated to the current mesh.

6.4 Material System

Every object which physically exists in the scene has to have a material attached to it. Materials are associated with **shaders**. The properties of the material are the uniform variables that are being passed to the shader that is associated with the material. In OP Engine, materials are flexible. They have a vector of pairs for different uniform types. These types are currently:

- Floats
- 2 Component Float Vectors
- 3 Component Float Vectors
- 4X4 Matrices
- Integers
- Textures

OP Engine has a distinct material class and this class holds all these variables in **vectors** for each uniform type. One material may be used in a lot of places and we also need to blueprint a material type and generate instances of it. For that, a struct named **MaterialInstance** has been added. This material instance has the same components in vectors and it is passed to the **MaterialComponent** of the entity. In the main loop, the values inside these material instances are updated and sent to their shader.

The default material that is given by the OP Engine itself is the **pbr** material. While creating a material, we have to define a **MaterialSpec** and add values to the vectors for each data type. For example, in **pbr** material there are:

- Roughness value which is a float between 0.0 and 1.0.
- Metalness value which is also a float between 0.0 and 1.0.
- A diffuse color is called **Albedo** and it is a 3-component Float Vector holding red, green, and blue values of an object's diffuse color.
- An albedo map that is a texture. If an object has a diffuse texture this field is being overwritten. By default, it is a simple white texture.
- Roughness map which determines roughness values with a texture. It is also a white texture by default.
- Metalness map that determines metalness values with a texture. By default, it is a white texture.
- Ambient Occlusion map. This is also a texture that determines which places of the object must be shadowy. It is a white texture by default.
- Normal Map. We define normal values with a texture to distort normal values created by vertices to create a more realistic-looking object. By default, it is a blueish color that resolves into vectors having $(0.0, 0.0, 1.0)$ value.
- Height map. This texture is needed for parallax mapping. It is also a white texture by default.

6.5 Renderpass System

While dealing with more complex scenes having different techniques being applied, we have to define an easy method to invoke shader and drawing commands. The reason is, some steps or **passes** can be dependent on another one. In OP Engine, in order to mitigate this complexity of dependencies, a **RenderPass** system has been created.

In this system, each **Renderpass** object has its own framebuffer. Their framebuffers are created in the constructors of **Renderpass** objects. They also have getters and setters for their framebuffers' color and depth attachments. The most important function in the **Renderpass** class is the **InvokeCommands** function. This function takes a **std::function** and when it is invoked, it:

- Binds its framebuffer
- Invokes the **std::function** given as argument
- Unbinds the framebuffer

If a **Renderpass** object depends on another **Renderpass** object's framebuffer information, it can be retrieved using **GetColorInput** and **GetDepthInput** functions. In OP Engine, the renderer uses **Renderpass** objects step by step like the following:

- **Depth Pass For Spot and Directional Lights:** In this pass, we render the scene with only depth information from the perspective of the camera. The result can be held in-depth attachment of the framebuffer. However, OP Engine uses **variance shadow mapping** technique and because of that, we have to save the depth values and additional information to the color buffer by taking advantage of green, blue, and alpha channels.
- **Depth Pass for Point Lights:** Point lights use a slightly different algorithm than spot and directional lights since they are omnidirectional. Because of that, another depth testing is being done for point lights as well.
- **Spot and Directional Light Blur Pass:** For creating soft shadows, with this pass, we are blurring the resulting depth map. This pass uses a special **renderpass** object which is called **pingpong renderpass**. This **renderpass** object holds two framebuffers and they are mainly used for blurring the x and y axes one after another.
- **Entity ID Pass:** With this pass, we are storing the id values of entities that are shown on the viewport. These IDs are used for mouse picking.

- **Scene Render Pass:** Inside this pass, **pbr** calculations and actual shading is being done. Depth maps that are coming from blurring and point lights are also being used in this pass.
- **Post Processing Pass:** This pass is for mainly tone-mapping the image acquired from the previous pass. The image we get from **Scene Render Pass** is actually a **high dynamic range** image. This pass reverts color values back to **low dynamic range**, specifically 8-bit unsigned integers between 0-255.

6.6 Cameras and Light Sources

OP Engine handles Cameras and Light Sources under **uniform buffers** as it is discussed in 6.2.5.

6.6.1 How Camera Data is Managed

Camera data is defined as follows:

```
struct CameraData
{
    glm::mat4 ViewProjection;
    glm::mat4 View;
    glm::mat4 Projection;
    alignas(16) glm::vec3 ViewPos;
    float Near;
    float Far;
} CameraBuffer;
```

And in the shader, its uniform buffer equivalent version is as follows:

```
layout(std140, binding = 1) uniform Camera
{
mat4 u_ViewProjection;
mat4 u_View;
mat4 u_Projection;
vec3 u_ViewPos;
```

```
float u_Near;
float u_Far;
};
```

If we look closely at the **glm::vec3** ViewPos in the C++ Struct, we see that **alignas(16)** is used. There is a reason behind this. In the shader, we have defined the layout of the uniform buffer as **std140** and in this layout, **vec3** values are aligned as 4 component floating point values, so they have to be aligned in memory addresses that are multiples of **16**.

In cameras, the View matrix is the inverse of the camera's transformation matrix. If this matrix is applied to a position in the space, it transforms it into the space of the camera such that the camera becomes the new origin. The projection matrix projects the 3D Position values to the 2D viewport. Camera uniform holds View, Projection, and ViewProjection matrices because all of them are needed in certain calculations. Viewpos, Near, and Far are mainly used in shadow mapping. Viewpos is additionally used in PBR rendering for calculating **view rays** as well.

6.6.2 How Light Source Data is Managed

In OP Engine, Spot and Directional lights are evaluated together. As discussed before, OP Engine supports multiple spot and directional lights. A dir light is defined as follows:

```
struct DirLight
{
    int CascadeSize;
    float FrustaDistFactor;
    alignas(16) glm::vec3 LightDir;
    alignas(16) glm::vec3 Color;
};
```

In the above code snippet, it is seen that we have again used **alignas(16)** before **glm::vec3** values. The shader equivalent of this struct is:

```
struct DirLight
```

```
{
int CascadeSize;
float FrustaDistFactor;
vec3 LightDir;
vec3 Color;
};
```

In order to support multiple of these Directional lights, we create the uniform buffer using the struct below:

```
struct DirLightData
{
    int Size;
    DirLight DirLights[MAX_DIR_LIGHTS];
} DirLightsBuffer;
```

and its shader equivalent is:

```
layout(std140, binding = 4) uniform DirLightData
{
int u_DirLightSize;
DirLight u_DirLights[MAX_DIR_LIGHTS];
};
```

Spot lights and point lights also follow this convention. They have their own structs and lastly, there is a uniform buffer that contains an array.

6.7 Shadow Mapping

OP Engine uses supports shadow mapping for multiple **spot**, **directional**, and **point** lights. The first step of applying the shadow mapping technique is getting a **light space matrix** for each light. We then render the scene like lights are the camera and save the depth values in a framebuffer. Later, we use those **light space matrices** to transform world coordinate points to the coordinate system of the light and check their depth values. If their depth values are greater than the according pixel's depth

value, this means that the fragment should be in shadow. The figure below illustrates this situation:

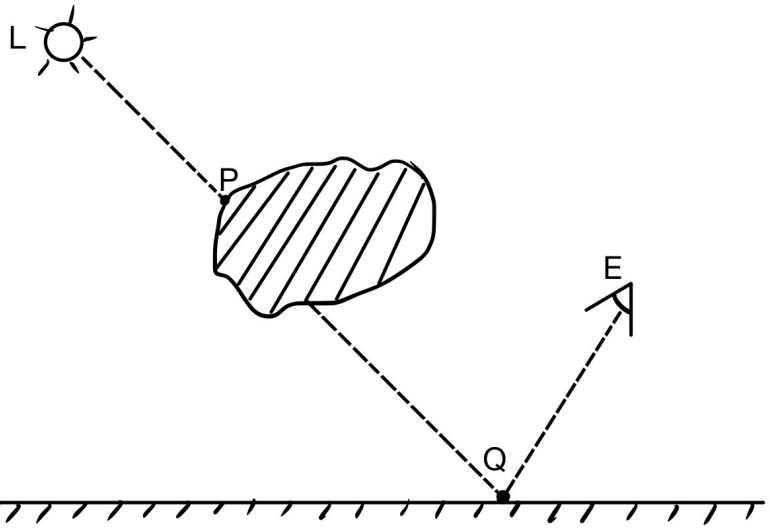


Figure 6.7: Shadow Mapping Depth Test

In the figure above:

- L is a point light
- E is the main camera that we use to render the scene
- Q is the fragment that is going to be depth tested
- P is the fragment on the depth map of L which is on the same line with Q

L has a depth map that was already written in the previous render passes. When we try to render the scene from the eyes of the camera, we test every fragment with available shadow maps. But these fragments are according to the world's coordinate system. So we need a way to transform these coordinates into the light's coordinate space.

Before rendering the scene, the transformation matrices that we need to transform coordinates in world space to light space are already sent to shaders as **Uniform Buffers**. We multiply these with **light space matrices** and we get the coordinate in light space. The next step is calculating the aligned pixel. This is done by doing

perspective division and transforming the resulting vector's values between 0 and 1. We sample from the depth map using x and y coordinates of this vector. The z coordinate of the first **light space** before we made perspective division gives the depth value that we want to test.

If we consider the figure 6.7, while testing the point Q, the depth value we sample from the depth map belongs to the point P. In the figure, it is seen that according to the light, point Q is further away so its depth value is greater than the value sampled which belongs to point P. By testing this, we decide that point Q is under the shadow. In OP Engine, **point lights**, **directional lights**, and **spot lights** shadow testing is done according to this approach.

6.7.1 Light Space Matrices, Cascade Planes, and Depth Maps

As it is discussed in previous sections, OP Engine supports multiple light sources. The light data for each light is filled into arrays inside **uniform buffer objects** and used later in the **fragment shader**. We also need a way to store **light space matrices** and **depth maps**.

First of all, we need to define a maximum number of **Directional Lights**, **Spot Lights**, **Point Lights** and **Cascade Size** for **cascaded shadow mapping**. In the current state of OP Engine, it is:

- $MAX_DIR_LIGHTS = 2$
- $MAX_SPOT_LIGHTS = 4$
- $MAX_POINT_LIGHTS = 10$
- $MAX_CASCADE_SIZE = 15$

6.7.1.1 Light Space Matrices and Cascade Planes

We first build our uniform buffers according to these maximum values for light space matrices. Light space matrices are divided into two categories.

- Light space matrices for **Directional** and **Spot** lights
- Light space matrices for **Point Lights**

The reason behind this is the similarities and differences between these light types. The two difference of **Directional** and **Spot** lights are:

- Directional lights use **orthographic** projection in light space matrix and Spot lights use **perspective** projection.
- In addition, Directional lights use **cascaded shadow mapping**, so that there should be additional light space matrices for different cascade levels.

So in total, the size of the array which holds the data equals:

$$MAX_DIR_LIGHTS * MAX_CASCADE_SIZE + MAX_SPOT_LIGHTS \tag{6.7}$$

So, each directional light has **max cascade size** amount of light space matrix slots attached and spot lights will have only one.

Above, it was mentioned that point lights have a different uniform buffer object for storing matrices. The reason behind that is we are producing depth maps for point lights in a slightly different fashion in the shaders. Each point light has **6** different light space matrices that show **6** directions in the 3D space which are **FRONT**, **BACK**, **UP**, **DOWN**, **LEFT** and **RIGHT**. So, the array size that holds the light space matrix data equals:

$$MAX_POINT_LIGHTS * 6 \tag{6.8}$$

In addition, we hold another uniform buffer for keeping track of **cascade planes**. These planes are produced by dividing the space between **near** and **far** planes of the camera. These cascade planes allow us to select the desired depth map for directional lights according to the viewer's distance from the fragment. The array size of cascade planes is as follows:

$$(MAX_CASCADE_SIZE - 1) * MAX_DIR_LIGHTS \tag{6.9}$$

6.7.1.2 Depth Maps

The sizes of depth maps for **Directional**, **Spot** and **Point** lights are the same as light space matrices. The difference is, these depth maps are actually produced in **Geometry Shaders** with **layered rendering** and stored in a framebuffer which holds a **Texture Array**.

Geometry shaders allow us to create new primitives from an existing primitive and layered rendering allows us to render a scene with multiple layers, so we use this to our advantage and create texture arrays for depth maps. As a result, **Directional** and **Spot** lights use the same framebuffer and point lights have another framebuffer attached.

6.7.1.3 Examples from OP Engine

In OP Engine, let's see how shadows are cast with different light sources. In the next figures, a scene with a **platform** and a **cube** has been used. In each figure, a different light source has been added. In the figure below, there is only environment light and none of the directional, spot, or points lights have been used.



Figure 6.8: Scene with only environment light

In the figure above, since there are no shadows, it is harder to take grasp the depth of the scene. Now, let's add a point light to the scene and see how it casts shadows. Point lights are omnidirectional lights, they use perspective projection in their light space matrices. It can be seen in the figure below:

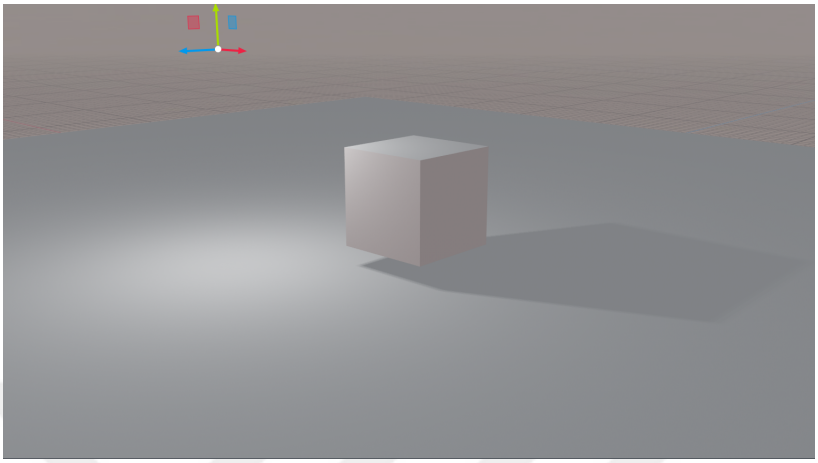


Figure 6.9: Scene with a point light

We see that shadows give a greater feel of the depth of the scene. Now let's look at how spot lights cast shadows in the figure below:

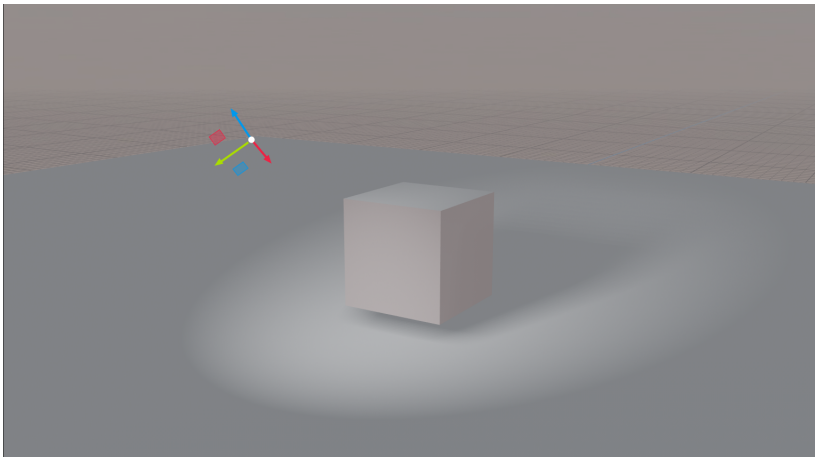


Figure 6.10: Scene with a spot light

In the two figures above, we can see how point lights and spot lights illuminate the scene and cast shadows. They both use perspective projections in the light space matrices. However, as is discussed, each spot light has one depth map, and point lights

have 6 depth maps attached. Lastly, let's look at how directional lights illuminate the scene:

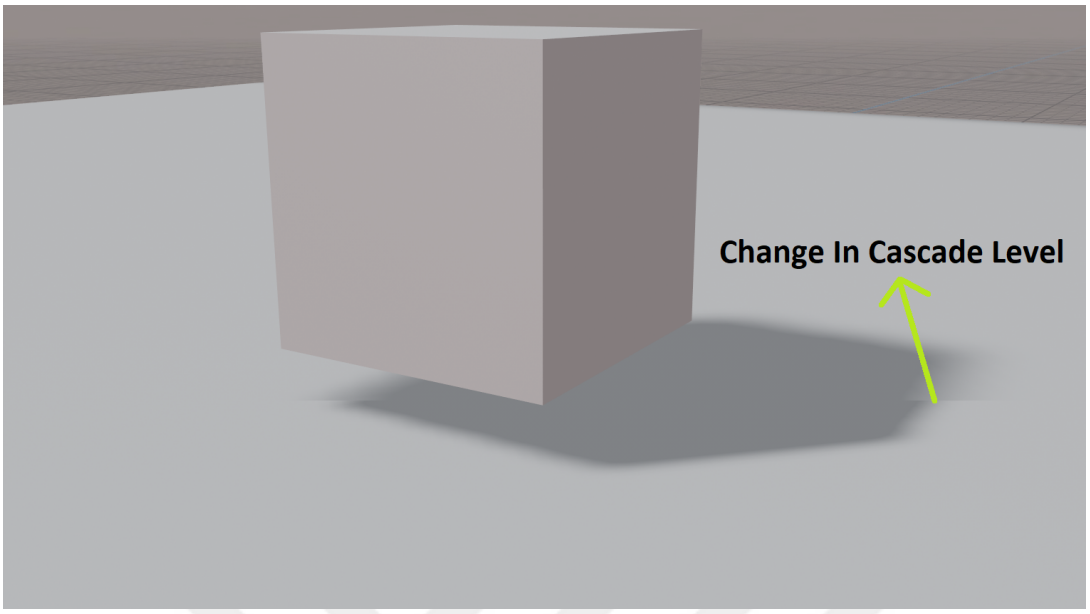


Figure 6.11: Scene with a directional light

From the figure above, we can see that the quality of the shadow is decreased as the processed fragment's distance is bigger than the threshold. This is because of the cascaded shadow mapping technique. In the figure above, we can see that change in the cascade level which is shown with a green arrow.

6.8 Physically Based Rendering and Environment Lighting

OP Engine uses its material system according to the **metallic workflow** for physically based rendering. For each material, there are **roughness** and **metalness** values. They both change the diffuse and specular lighting. For example, let's think about a grid of different spheres with different roughness and metalness values. The specifications are:

- The lower left sphere has 0.0 roughness and 0.0 metalness.
- As we go right, roughness increases by 0.25 for each sphere. So the rightmost spheres have 1.0 roughness.

- As we go upward, metalness increases by 0.25 for each sphere. So the uppermost spheres have 1.0 metalness.

According to these specifications, the figure below shows how OP Engine's PBR renderer handles these materials:

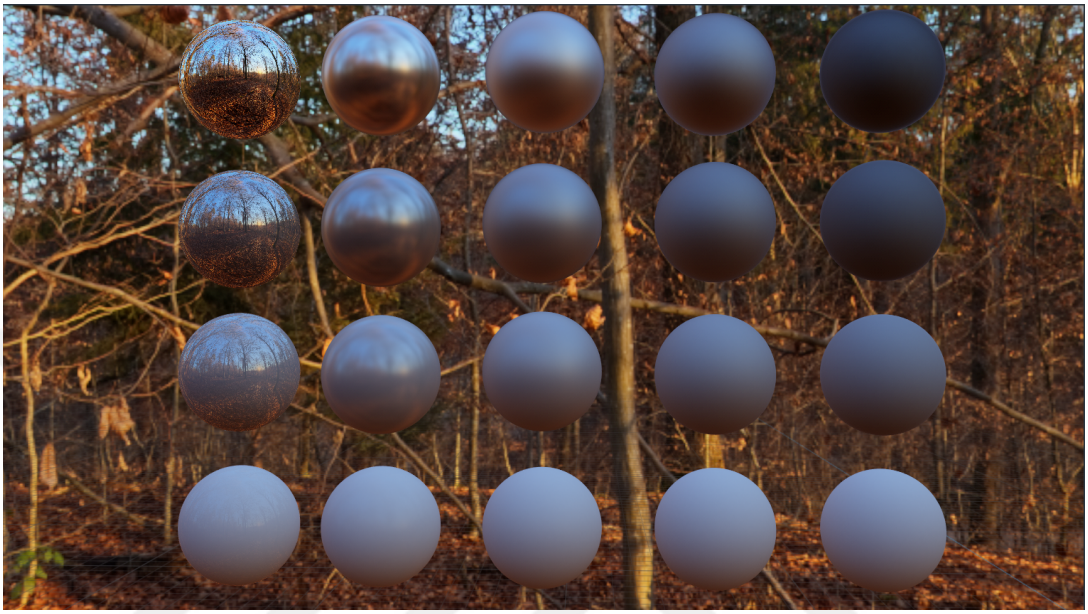


Figure 6.12: Metalness and roughness values

In addition, OP Engine's PBR material system also supports environment mapping. For example, in the figure above an equirectangular hdr environment map was posted by Alexander Scholten [39].

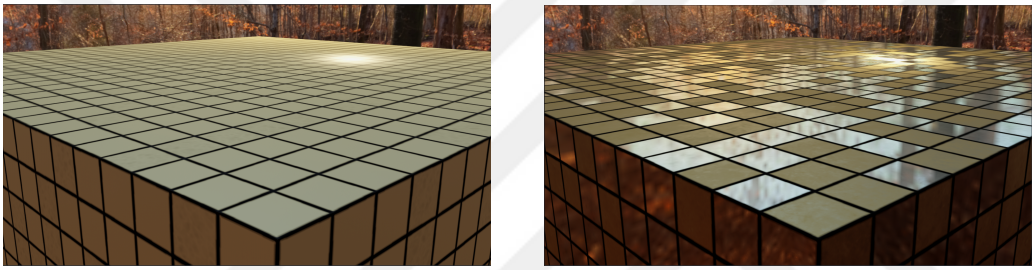
6.8.1 Texture Maps

OP Engine uses **metallic workflow** with Pbr materials which is explained by Brian Karis in 2013 [40]. As it is explained in the material system 6.4. There are different texture maps that are being used together to create a realistic material effect. The texture maps that are used by OP Engine's Pbr renderer are:

- **Albedo and Ambient Occlusion Maps:** Albedo specifies diffuse color. The ambient Occlusion map determines which places in the diffuse color should be

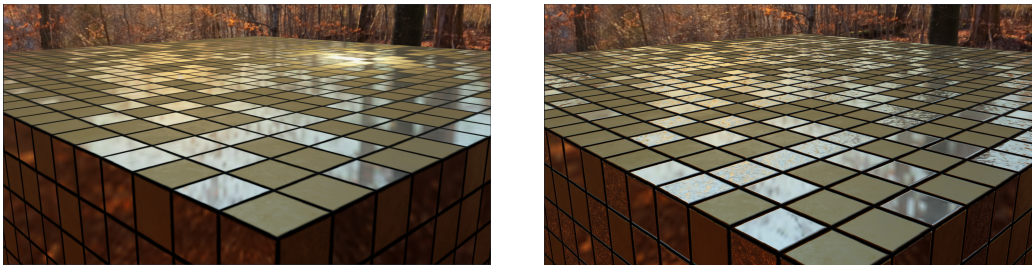
in shadow.

- **Roughness Map:** Specifies roughness values
- **Metalness Map:** Specifies metalness values
- **Normal Map:** As discussed in section 6.3.1, normal maps take advantage of the tangent space and distort the normals, creating a more detailed look without spending a lot more vertices.
- **Height Map:** Height maps give the illusion of depth by tweaking the texture coordinates in certain places. This also enhances the quality of the rendering. It can work together with normal maps.



(a) Albedo and AO Maps Added (b) Roughness map added

Figure 6.13: Texture Mapping of Roughness Albedo and AO



(a) Metalness map added (b) Normal map added

Figure 6.14: Texture Mapping of Metalness and Normal

Lastly, the figure below shows the final rendering of the cube with metal tile material after adding the height map. As it is seen, these texture maps collectively add a lot of

detail to the simple cube with only **6** vertices. This way, a huge variety of different materials can be introduced to the scenes that are going to be created. These texture maps are taken from Artem Lebenev's free PBR texture set [41]. In the figure below, some tiny artifacts may be seen if we move the camera a lot closer and parallel to the cube. The reason is OP Engine currently uses **parallax mapping** with **height maps** and they do not work properly when the camera is very close and parallel to objects with height maps.

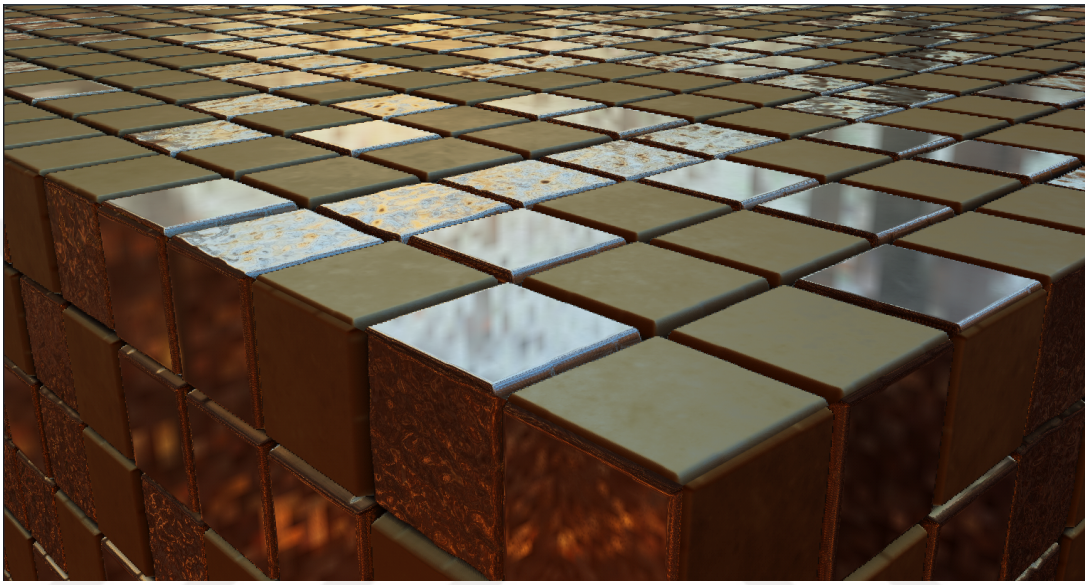


Figure 6.15: Height Map added

CHAPTER 7

RESOURCE MANAGER OF OP ENGINE

Managing the resources is a crucial part of a simulation - game engine. In OP Engine, resources are mainly regulated using the **Resource Manager** class. This class is designed as a singleton class. Normally, the resource manager starts collecting the necessary items at the startup of the program that uses OP Engine.

7.1 Data

Resource manager holds data in a static structure that consists of different unordered maps. The resource manager gives each resource registered an ID and also saves its name. There are two lookup tables for conversion between names and ids. These are:

- **String Lookup Table:** This is an unordered map where values are IDs and keys are the names of resources.
- **ID Lookup Table:** This unordered map has values as resource names and keys as IDs.

7.1.1 Resource Allocation

Resource manager uses a function which is called **Allocate** for allocating space for the resource. This function takes a string name of the resource and fills up the **StringLookupTable** with that name. Resource manager holds a **counter** for determining IDs of different resources that are going to be allocated. The ID of the resource becomes the current value of the counter. Moreover, At the end of the allo-

cation, the current value of the counter is incremented by 1 and the function returns the previous value of the counter which is the ID of the newly allocated resource. The algorithm of allocating a resource is shown below:

Algorithm 8 Resource Allocation

1: $StringLookupTable[name] \leftarrow counter$
2: $IDLookupTable[counter] \leftarrow name$
3: $counter \leftarrow counter + 1$
4: **return** $counter - 1$

7.2 Initialization of Resource Manager

The resource manager is initialized when an application is started which is shown in figure 3.1. At the beginning of the initialization, **rootFilePath** is given to the resource manager so it knows where is the root directory of the project. After that, paths for:

- **AssetPath:** rootFilePath/assets
- **ModelPath:** AssetPath/models
- **ShaderPath:** AssetPath/shaders
- **ShaderIncludePath:** ShaderPath/include
- **MaterialPath:** assetPath / materials
- **TexturePath:** assetPath / textures

Then, these paths are used to load all resources. It starts with loading and compiling shaders.

7.3 Shader Compilation

Shaders are discussed in section 6.2.6. They are programs that run on the GPU which are deeply connected with the **Graphics Pipeline** except compute shaders. The steps for handling shader resources are:

- Loading all **include shaders** to a map called **IncludeShaderSources** with their file names as string buffers. In this process, every file residing inside the **include** directory is recursively traversed and loaded to the map.
- Loading all **shader sources** to another map called **ShaderSources** with their file names as string buffers. Just as loading **include shaders**, this time every source file inside **src** directory is recursively traversed and loaded into the map.
- Traverse all shader sources one by one and extract their **vertex**, **geometry**, and **fragment** parts according to the **#type** token. After that, run a function called **ResolveIncludes** which triggers the custom-created include algorithm which is explained in 6.2.6.2. The resolved sources are stored in a vector as string buffers and then they are given to the shader constructor.
- Lastly, allocate memory for the shader and get an ID, then use that id to load the **shaders** map and save a shared pointer there.

After following these steps, shaders that are required in the program are compiled and ready to go.

7.4 Loading Models

OP Engine is capable of loading models using Assimp Library [26]. However, that feature is out of the scope of this thesis. But it is also a part of the resource manager's initialization process. So it is briefly going to be explained in this section. OP Engine can deal with model files with extensions such as:

- gltf
- fbx
- dae

Every file residing inside the **models** folder is recursively traversed and the models are loaded using the constructor of the **Model** class. This class loads the vertex,

index values, and also **skeletal animation** data using Assimp. In the end, memory is allocated and resulting models' shared pointers are loaded to a map called **model** with their IDs generated in allocation.

7.5 Generic Resources

The next step of resource manager initialization is loading some generic resources.

7.5.1 Generic Meshes

Generic meshes that are loaded are as follows:

- Simple Cube
- Plane
- Flat Icospheres with subdivision values 1, 2 and 3
- Smooth Icospheres with subdivision values 1, 2 and 3
- Flat Radial Spheres with the sector, stack counts 16, 32, 64 and 128
- Smooth Radial Spheres with sector stack counts 16, 32, 64 and 128
- Pyramid
- Smooth and Flat Cylinders
- Smooth and Flat Capsules

7.5.2 Generic Textures

The generic textures that are loaded in the beginning are:

- **White Texture:** This is a 1x1 texture that is used when we expect a texture but do not use it. So it has no effect on the material.

- **Black Texture:** This is another 1x1 texture that nullifies the effect of other parameters

- **Default Normal Map:** A 1x1 normal map with a blueish color. This texture is used when there are no normal maps specified for a material. It has no effect on the real normal of the surface.

- **Bayer Dithering Matrix:** This is actually a matrix that is used for **Dithering Technique** but it is saved as 8x8 texture and sent to the final fragment shader for creating **dithering effect** that reduces the banding artifact of renderings. The matrix is as follows:

$$\begin{bmatrix} 0 & 32 & 8 & 40 & 2 & 34 & 10 & 42 \\ 48 & 16 & 56 & 24 & 50 & 18 & 58 & 26 \\ 12 & 44 & 4 & 36 & 14 & 46 & 6 & 38 \\ 60 & 28 & 52 & 20 & 62 & 30 & 54 & 22 \\ 3 & 35 & 11 & 43 & 1 & 33 & 9 & 41 \\ 51 & 19 & 59 & 27 & 49 & 17 & 57 & 25 \\ 15 & 47 & 7 & 39 & 13 & 45 & 5 & 37 \\ 63 & 31 & 55 & 23 & 61 & 29 & 53 & 21 \end{bmatrix} \tag{7.1}$$

7.6 Environment Maps and Textures

The next step of initialization is loading Environment Maps and Textures. In reality, environment maps are also textures, but they have different purposes. So, we first deal with environment maps.

7.6.1 Environment Maps

Environment maps are mainly **hdr** images having color values with much higher precision than standard 8-bit unsigned integers. They are 32-bit floating point numbers. OP Engine currently supports environment maps with **equirectangular** format. We have to process this image before loading it as an environment map. Step by step, the following is applied to the map:

- **Preparing Shaders:** We need to prepare some shaders that we want to use in order to:
  - Convert equirectangular map into a **cubemap**
  - Convolute **cubemap** for generating **irradiance map**
  - Generate the **prefilter map** that we use for specular lighting. This prefilter map will have **mip** levels that are used for roughness values at specific ranges.
  - Generate **BRDF LUT** map. In this texture, vertical coordinates are roughness values and horizontal coordinates are the dot product of the normal and the view direction. Red and Green values of this map are used while calculating the **reflectance equation**.
  - Prepare a skybox shader that is going to be used to implement the skybox effect. In this shader, a cube with inverted normals is always rendered inside the camera and behind every object.
- **Loading Textures:** Every environment map residing in the environment map folder is recursively traversed and loaded as **hdr** textures.
- **Environment Map Creation:** Give prepared shaders and **hdr** textures to the constructor of **Environment Map** class and create the **Environment map** object.

7.6.2 Textures

Textures are also loaded by traversing every texture file residing in the textures folder recursively. OP Engine uses Stb Image library [19] for loading **non hdr** textures. The supported formats are:

- png
- jpeg
- bmp
- tga

7.7 Materials

As discussed in the material system section 6.4 OP Engine's material system is designed in such a way that each material is associated with a shader and its uniforms are stored inside the material. In each time step, these values are updated.

Materials are designed to be resources so developers might write their own materials just like in Unreal Engine [4], Unity [5] or Godot [6]. However, for now, OP Engine loads only a generic material called the **pbr** material. The properties of this material are:

- $Roughness = 0.5$
- $Metalness = 0.1$
- A three-component float for **Albedo**
- A white texture for **Albedo Map**
- A white texture for **Roughness Map**
- A white texture for **Metalness Map**
- A white texture for **Ambient Occlusion Map**
- A default normal map for **Normal Map**
- A white texture for **Height Map**

The shader associated with this material is the **Main** uber shader that is responsible for all the rendering of the engine currently.

CHAPTER 8

OP ENGINE'S PHYSICS SYSTEM

OP Engine uses Bullet Physics library [11] for physics calculations. OP Engine stores a **physics world** inside **Physics System**. In each timestep during runtime, this physical world is being put one step ahead.

The objects that are registered to the physics world are the entities having **Physics Material** and **Colliders**. In each timestep of the runtime mode, the positions and orientations of these entities affected by the physics world are updated in the **graphics world** as well.

8.1 Initiating Physics Manager

As discussed in the figure 3.1, **Physics Manager** is initialized after the initialization of the **Resource Manager** is finished. In initialization, OP Engine does not create a physics world, this is done when the runtime mode starts. Instead, in the initialization of the **Physics Manager**, three **callback functions** are defined. These are:

- **gContactAddedCallback:** This function is being called when a certain entity is currently colliding with another entity. We define a custom function for defining the behavior in this situation. In OP Engine, if a collision occurs between certain entities, the script manager calls **OnCollision** function. If a script is defined to that entity and **OnCollision** function is defined, that function is being called from C# side.
- **gContactStartedCallback:** Just like **gContactAddedCallback**, this function is for only keeping track of the collision that has started. If a collision is started

between certain entities, the custom function calls **OnCollisionStarted** function via the **scripting system**. This way, the behavior of entities when a collision starts can be programmed.

- **gContactEndedCallback:** Like the ones above, this function also calls a function from the **scripting system**. But this time, this function is called when a collision ends between certain entities. If **OnCollisionEnded** functions are defined in C# scripts, then these functions are called when a collision ends.

8.2 Starting Physics World

A physics world is started when the state of the application changes to the **runtime mode**. In this mode, a physics world is started. In OP Engine, there is a static struct for holding the pointers of the data for the physics world. This struct has:

- Collision Shapes inside an array
- Broadphase Interface
- Collision Dispatcher
- Constraint Solver
- Collision Configuration
- Discrete Dynamics World

When starting a new physics world, the data above is being defined. With the Bullet Physics library, this is done as follows:

- Define a collision configuration. This is selected as **btDefaultCollisionConfiguration** in Bullet's library.
- Define the dispatcher. We give collision configuration to the constructor of the **btCollisionDispatcher** object and create the dispatcher.
- Define the broadphase.

- Define the constraint solver. It is defined as **btSequentialImpulseConstraintSolver** in Bullet physics.
- Create the dynamics word. We give dispatcher, broadphase, solver, and collision configuration to the constructor of **btDiscreteDynamicsWorld** object. After that, we set the gravity for the dynamics world. It is $(0.0, -10.0, 0.0)$ by default.

8.3 Rigid Bodies

The entities which have **physics material** and **collider** components are added as rigid bodies to the dynamics world. These components are explained in 5.1.2.

8.3.1 Collider Shapes

There are 4 collider shapes that are being used by the physics manager. These colliders become the boundaries of rigid bodies while testing collisions. They can also be visualized inside the OP Editor. These are:

- Box shape. This is modeled as a standard cube. It is visualized as follows:

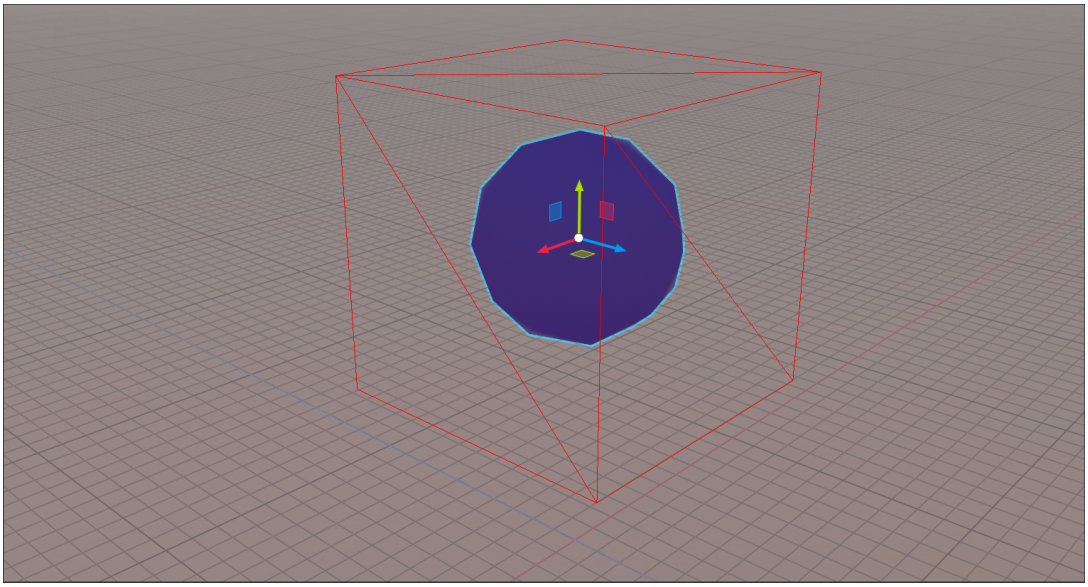


Figure 8.1: Box Collider

- Sphere collider. This is also a standard sphere from the mesh class. It is visualized as:

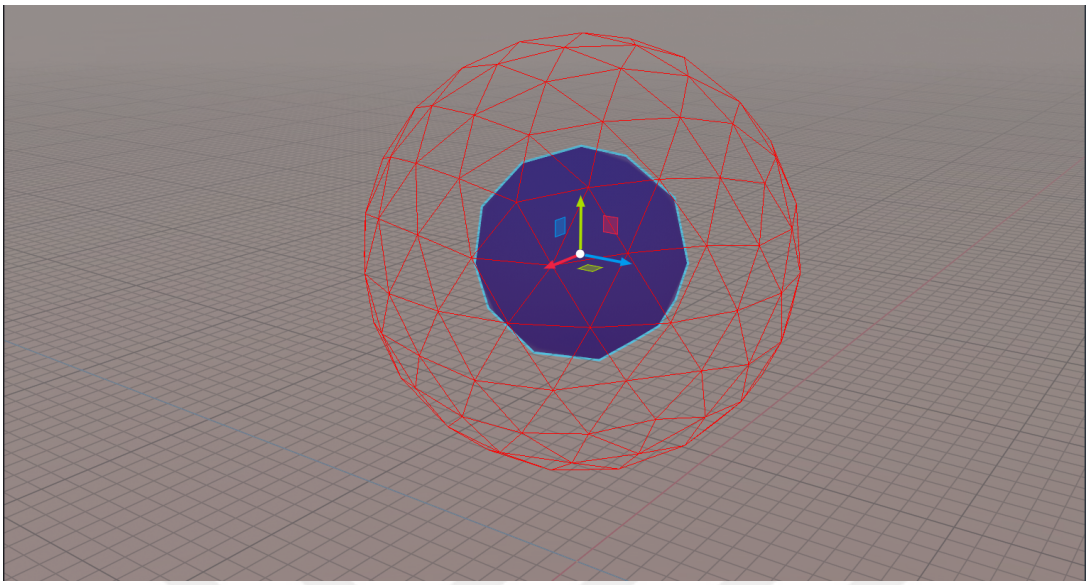


Figure 8.2: Sphere Collider

- Cylinder collider. A standard cylinder from the mesh class. It is shown as:

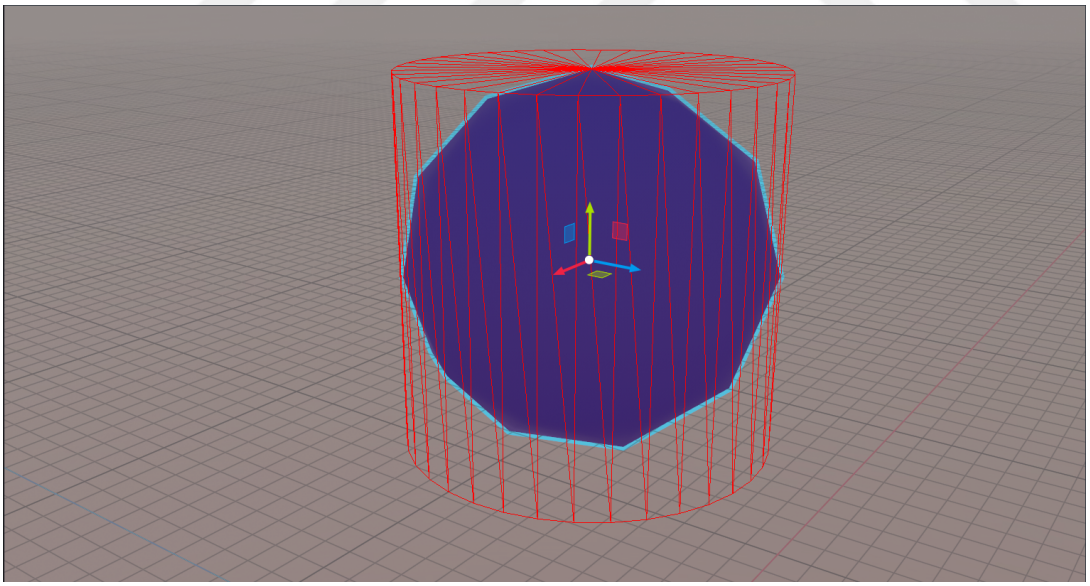


Figure 8.3: Box Collider

- Capsule collider. Just like the others, it is from the mesh class. The visualization is as follows:

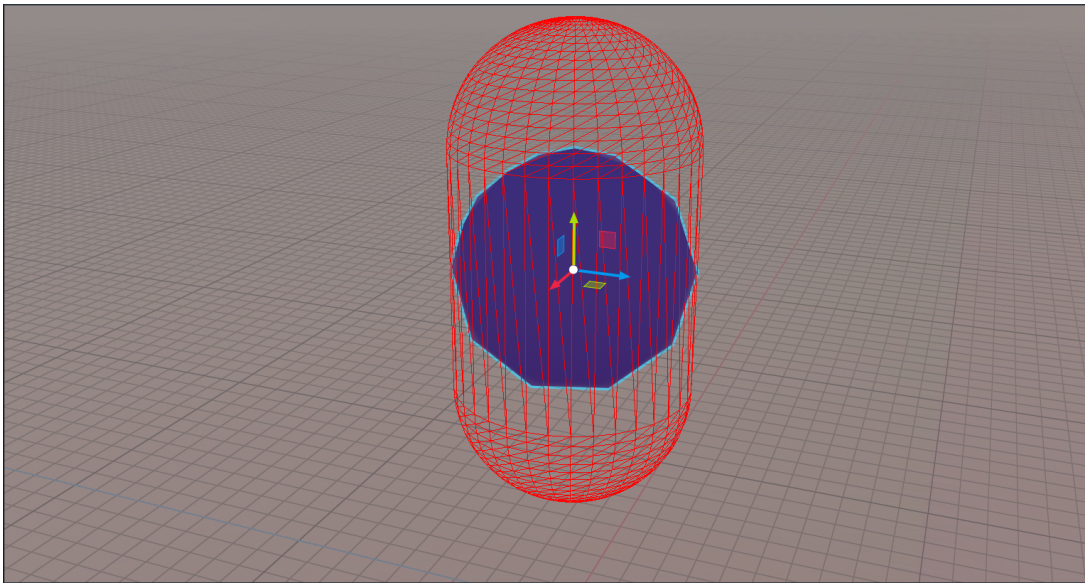


Figure 8.4: Capsule Collider

The job of the colliders above is to detect collisions during runtime. If these colliders are marked as **no contact response** colliders. Then the physics world will not be affected by collisions with these colliders. However, we still keep track of collisions. This is helpful for creating sensors.

8.3.2 Adding Rigid Bodies

When the runtime mode starts, all of the rigid bodies are added to the physics system with their properties. These properties are the **physics material** and **collider** component properties of the entity. We also feed the current origin and orientation of the object so we use **transform component** too. This way, a rigid body is added to the dynamic world.

One thing to mention is that we add a flag called **CUSTOM MATERIAL CALLBACK** which tells the Bullet library that we are using custom callback functions for collisions. These functions are explained in 8.1.

After adding the rigid body to the system. We also return two pointers which are the **motion state** and **rigid body**. Updating the motion state changes the position of the object and we can apply forces, impulses, etc. on the rigid body by using the rigid

body pointer. This is widely used with scripting system.

8.3.3 Deleting Rigid Bodies

Sometimes, we might delete some entities during the runtime. If these entities have **physics material** and **collider** components, then they are deleted. Firstly, their motion state is deleted and secondly, the rigid body is deleted from the dynamics world. Lastly, the collision shape pointer stays as a dangling pointer. We also delete it at the end.

8.4 Clearing Physics World

While changing from runtime mode to editor mode, the physics world is cleared. All rigid bodies and collisions shapes are cleared first. Next, these pointers are deleted in the reverse order of the way they are created:

- Dynamics world
- Solver
- Broadphase
- Dispatcher
- Collision Configuration

CHAPTER 9

OP ENGINE'S SCRIPTING SYSTEM

Scripting is an important pillar of a simulation-game engine's architecture. The logic of the simulations and how entities react to each other are determined with scripting. Everything that is explained in the previous chapters does not contain any custom logic.

After designing a scene, we must have a way to give "life" to the entities. For example, let's consider a scene with:

- An entity with a camera component. The camera should rotate when the mouse is moved according to the move.
- A cube which should move to negative Z direction when **W** button is pressed.

One solution to provide this functionality is directly writing the code in the engine. In the main loop, these functionalities can be provided specifically for the cube and the camera entities. However, this only works for the scene defined above. If we want to load a different scene with entities having different behaviors. We also have to rearrange the entire codebase of the engine. And since simulation and game engines are generally built with C++, we need to compile the engine every time which is a huge waste of time.

This creates a reusability problem. We need a way to define modular behaviors for every entity where we do not need to compile the entire codebase for every scene. Even in the same scene, sometimes we might want to change the behavior of the objects. In order to cope with this problem, engine developers have implemented the **script manager** concept.

Script manager takes **scripts** that are written in a scripting language and matches the functions written in the script with its counterparts in the engine. Normally these scripting languages are used:

- Python
- Lua
- C#

In OP Engine, C# scripts are being used. Thus, in OP Engine's script manager, C# is embedded into C++. For this **mono** [28] is used.

9.1 Mono

Mono is an open-source C# compiler. It has some libraries that simulation engines can use. By integrating these libraries, we can load C# scripts during the runtime of the program and use them whenever they are needed.

Unity Engine [5] also uses Mono in its scripting system. In Unity, GUI is also implemented with C#. So whenever a script is loaded, parts of the GUI are also restarted. In contrast to that, OP Engine's GUI is written in C++, so GUI is separated from the scripting system.

9.1.1 Embedding Mono into C++

This process is being explained in an advanced tutorial of Mono Project about C++ embedding [42]. In this tutorial, it is explained that we can run mono runtime alongside the C++ program. In addition, we can load custom assemblies that contain the scripts that we want to use.

For embedding mono into C++, the first step that has to be taken is determining what C# libraries or modules we want to use in the runtime. We have to include these shared libraries inside a directory named mono to the path where the binary is located so mono runtime has access to them. In OP Engine, these are:

- Mono.Cecil.dll
- Mono.Security.dll
- mscorlib.dll
- System.Configuration.dll
- System.Core.dll
- System.dll
- System.IO.Compression.dll
- System.Numerics.dll
- System.Security.dll
- System.Xml.dll
- System.Xml.Linq.dll

As stated in the tutorial [42], when we open up a mono runtime inside a C++ or a C program, we can load our scripts and make these scripts communicate with the C++ code. The figure below shows this communication:

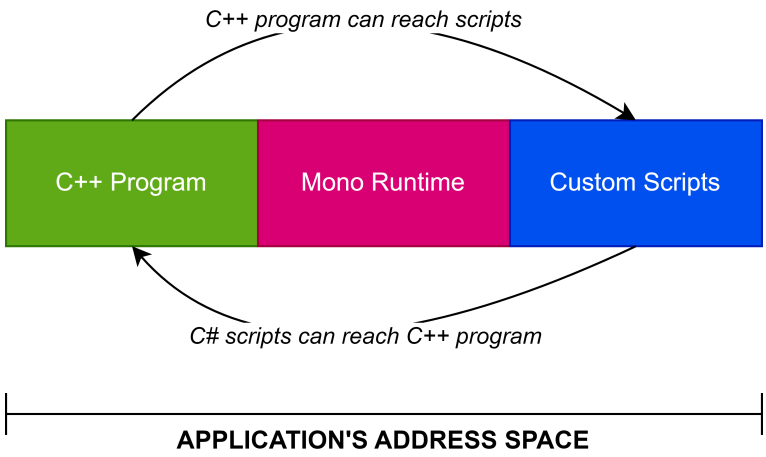


Figure 9.1: Communication between C# and C++

9.2 Script Manager

The backbone of the Scripting System of OP Engine is the Script Manager Class. This class is responsible for calling important functions in the scripts and ensuring the communication between C# scripts and the C++ program. The script manager is initialized as it is discussed in 3.1.

Initialization does three tasks. These are:

- Saving The assembly path given to the initialization function as an argument.
- Initialize Mono
- Load Runtime Assembly

9.2.1 Initialize Mono

In this part, the path of runtime shared libraries (.dlls) that we need is determined. These libraries are explained in 9.1.1. It is determined by invoking the function:

```
mono_set_assemblies_path("mono/lib");
```

Secondly, a pointer to the root domain is acquired by invoking the function:

```
MonoDomain* domain = mono_jit_init("Op");
```

Lastly, we also create an appdomain with a name and set the domain to the appdomain:

```
monoDomain = mono_domain_create_appdomain(name, nullptr);
mono_domain_set(monoDomain, 0);
```

9.2.2 Load Runtime Assembly

Runtime assembly is the custom scripts that are loaded during runtime. There are two types of custom scripts. These are:

- Custom scripts that are provided by the engine.
- Custom scripts written by simulation designers or game developers.

These scripts will be explained in further sections in detail. After these scripts are loaded by using **InitializeAssembly** and their images are acquired, the functions inside these scripts are bound to their C++ counterparts. There are more than 30 functions that are interconnected. If we group them by their purposes. These groups will be:

- Tag component getters and setters
- Input functions controlling key and mouse presses
- Window functions for hiding, showing cursor, or getting mouse position
- Transform component functions. These functions can manipulate transform, scale, and rotations.
- Scene functions. These functions are for manipulating exposure, skybox, checking tone map, or setting the scene.
- Physics material functions. We can manipulate the physical properties of entities such as mass and friction coefficients. We can also apply forces, impulses, and torques to their rigid bodies.
- Camera functions. Manipulating properties of the camera component is being done with these functions.
- Entity functions. Adding, removing entities, or setting their transforms are done with these functions. We can also check whether an entity has a certain component or not.

After loading the assemblies. Initialization of the script manager is completed.

9.2.3 Calling Script Functions

In subsection 9.2.2, it is discussed that there are some C++ functions that we can call from C#. But this is one-way communication. We also need a way to call C#

functions from C++. These functions are the main important functions that are needed to be filled. These functions are:

- **OnCreateEntity:** When an entity is created, this function is called.
- **OnUpdateEntity:** In each time step, this function is invoked.

In addition to these functions, there are functions for detecting collisions. These functions are **OnCollision**, **OnCollisionStarted**, and **OnCollisionEnded** They are explained in 8.1.

9.3 OP Script

As explained in 9.2.2 we need custom scripts provided by the engine. For that, a C# project named OP-Script is implemented. In OP Script the base class that is provided for entities is the **Entity Class**. Every script that is going to be added into entities as a component should have a class derived from this base **Entity Class**.

This class contains the functions about entities which are explained in 9.2.2 and also the ones explained in 9.2.3. The functions which are going to be borrowed from C++ are defined with:

```
[MethodImpl(MethodImplOptions.InternalCall)]
```

in the beginning. And the functions start with **private static extern** keywords. This tells mono that these functions are actually implemented in C++ and it is bound with Internal Call Arranger.

Functions such as **OnUpdate** or **OnCreate** are defined normally in C# and they are called from C++. In the base entity class, these functions are left blank. They are overloaded by the deriving classes.

9.3.1 Structure of OP Script

OP Script project has different components and structs that are and can be used by Entity objects or objects deriving from Entity Class. These components are grouped according to their purposes. They are:

- **Application:** Functions that are for applications are stored in this group. Currently, there is a script called **Window.cs** which invokes C++ functions that are defined in the engine.
- **Input:** Functions for key presses are defined here.
- **Math:** This part contains constants such as **PI** number, struct like **4x4 Matrices**, **Quaternions**, **Vec2s**, **Vec3s** and **Vec4s**. The layout of these structures in the memory should also be explicitly stated in C# in order to match them properly with their C++ counterparts. For that, we put the statement just before the declaration of classes:

  ```
  [StructLayout(LayoutKind.Explicit)]
  ```

  After that, when a member variable is declared its field offset is given. For example, **Quaternions** are struts that have 4 floating point members named **w**, **x**, **y**, and **z**. So, these member variables are declared as:

  ```
  [FieldOffset(0)]  public float w;
  [FieldOffset(4)]  public float x;
  [FieldOffset(8)]  public float y;
  [FieldOffset(12)] public float z;
  ```

- **Scene:** Scene contains components that are connected to the scripting system. These are:
  - Camera Component
  - Physics 3D Material
  - Tag Component
  - Transform Component

This part also contains the base **Entity Class**, **Scene Class**, and another script named **Null Script**. The null script is a special child of **Entity Class**. It does nothing and is automatically added to entities if a script component is added in OP Editor via GUI.

9.4 A Script Example

As an example, in this section, a free first-person camera script is built. This script is then added to the entity which is meant to be the camera, having the necessary components. The name of the script and the class is **FreeFirstPersonCamera**. This script will give the entity ability to change orientation position according to inputs from the keyboard and mouse.

First of all, this class is defined as:

```
internal class FreeFirstPersonCamera : Entity
{
...
}
```

In the above code snippet, it is seen that this class is derived from **Entity Class**. Thus, there are some **virtual** functions that we have to implement in order to get this script working. These will be **OnCreate** and **OnUpdate** functions.

But first, we define the private and public member variables of the class. Private variables are defined as:

```
private TransformComponent Transform;
private (float, float) LastMousePos = (0.0f, 0.0f);
private (float, float) MouseDelta = (0.0f, 0.0f);
private Vec3 UpVec = new Vec3(0.0f, 1.0f, 0.0f);
private bool MoveDisabled = false;
```

The **TransformComponent** is needed to change the transform of the entity. We

also keep track of the last mouse position and mouse delta. UpVec is needed for getting **Right** direction via Cross product with the **ForwardDir** which we get from the **TransformComponent**.

Secondly, the public members are defined as:

```
public float YawSpeed = 0.001f;
public float PitchSpeed = 0.001f;
public float MoveSpeed = 1.0f;
```

These are speed variables and can be manipulated by OP Editor itself since they are public member variables. After defining member variables, **OnCreate** function is implemented. As it is stated in 9.2, this function is called after this entity is created during runtime mode. It is a small function and what it does is:

```
public void OnCreate()
{
    Transform = GetComponent<TransformComponent>();
    LastMousePos = Window.GetMousePos();
}
```

It first acquires the TranformComponent that is attached to the script and the last position of the mouse from the static **Window** object. Secondly, we have to implement **OnUpdate** function. This function is called in every time step. This function is implemented as the following code snippet:

```
public void OnUpdate(float ts)
{

    HandleKeyboardInputs(ts);

    if(!MoveDisabled)
        HandleCameraOrPosInputs(ts);
}
```

In every time step, this function handles keyboard inputs by another private function inside the class. **HandleKeyboardInputs** function has a bunch of **if** statements that are checking whether a certain key is pressed. For example, when **W** is pressed camera should move forward. It is checked as:

```
private void HandleKeyboardInputs(float ts)
{
    ...
    if(!MoveDisabled)
    {
        // Get Forward and Right
        Vec3 ForwardDir = Transform.GetDirection();
        Vec3 RightDir = Vec3.Cross(UpVec, ForwardDir);

        // Forward
        if (Input.IsKeyPressed(KeyCode.W))
        {
            Vec3 pos = Transform.Translation;
            pos += ForwardDir * MoveSpeed * ts;
            Transform.Translation = pos;
        }
    ...
    }
}
```

For handling translation inputs, **HandleKeyboardInputs** first checks whether moving is disabled or not first. Secondly, it acquires forward direction from the **Transform** component and calculates **RightDir** for moving left and right. After that, it checks for keyboard inputs and performs the translation on **Transform** object. For example, when **W** is pressed. The position of the transform is translated along the line of the forward direction with the speed variables. This way, a script can be added to an entity and every kind of behavior can be programmed without changing the code base.

9.5 Reloading Scripts

Finally, there is of course a necessity for reloading the script during runtime. In OP Engine, this is done with a **File Monitor** class that is responsible for checking updates on a file in a separate thread. It checks the modifications on a folder in certain time intervals and changes its member boolean variable called **updated**.

In the layer created by OP Editor. Every time **OP Update** is called, this script file monitor's boolean member variable is checked and if there is a modification, scripts are reloaded. Since it works on a separate thread, it does not cost a lot of performance to implement this feature.

CHAPTER 10

A CASE STUDY: SWARM ROBOTS

In this section, a complete scene with entities that interact with each other with custom scripts is implemented. For this purpose, a scenario of swarm robots roaming inside a maze while avoiding obstacles is being developed. This way, the integrity of every subsystem of OP Engine is tested. In this scene:

- Layer System
- Input and Event Systems
- Entity Component System
- Graphics System
- Resource System
- Physics System
- Scripting System

Parts of OP Engine are tested.

10.1 Ground and Maze

Before starting the implementation of swarm robots, the first thing that is needed to be done is building the **Ground** and the **Maze**. These objects will be obstacles that swarm robots try to avoid.

To work in a more organized manner, the elements which make up **Maze** and **Platform** are grouped together under two distinct entities named **Maze** and **Ground**. **Ground** entity is a cube which is scaled with $(10.0, 0.2, 10.0)$ values for x, y and z-axis. This platform is planned to be a static object, however, we define it to be a dynamic object because we want to build the maze on top of the ground.

Ground entity has also a **Physics 3D Material** component with values:

- **Mass:** 10000000.0
- **Friction Coeff:** 1000000.0
- **Restitution:** 0.5
- **Fixed Rotations:** Rotation around x, y and z-axis are **fixed**

Ground entity also has 4 **GroundPillar** entities which are static objects and support the ground. They are cubes scaled with $(0.2, 5.0, 0.2)$ values and they also have **3D Physics Materials**. Values of the component are as follows:

- **Mass:** 0.0
- **Friction Coeff:** 1000000.0
- **Restitution:** 0.5
- **Fixed Rotations:** Rotation around x, y and z-axis are **fixed**

When **Mass** is equal to 0.0, it tells Bullet Physics Library [11] that the object is a **static object**.

Secondly, every element of the maze consists of standard cubes with physical properties that are equal to the **Ground Entity**. These cubes are placed in certain places on the ground and they form the maze together. To get a sense of depth in the scene, multiple lights can also be added to the scene. At the end, the **Maze** and the **Ground** looks like the following for the scene:

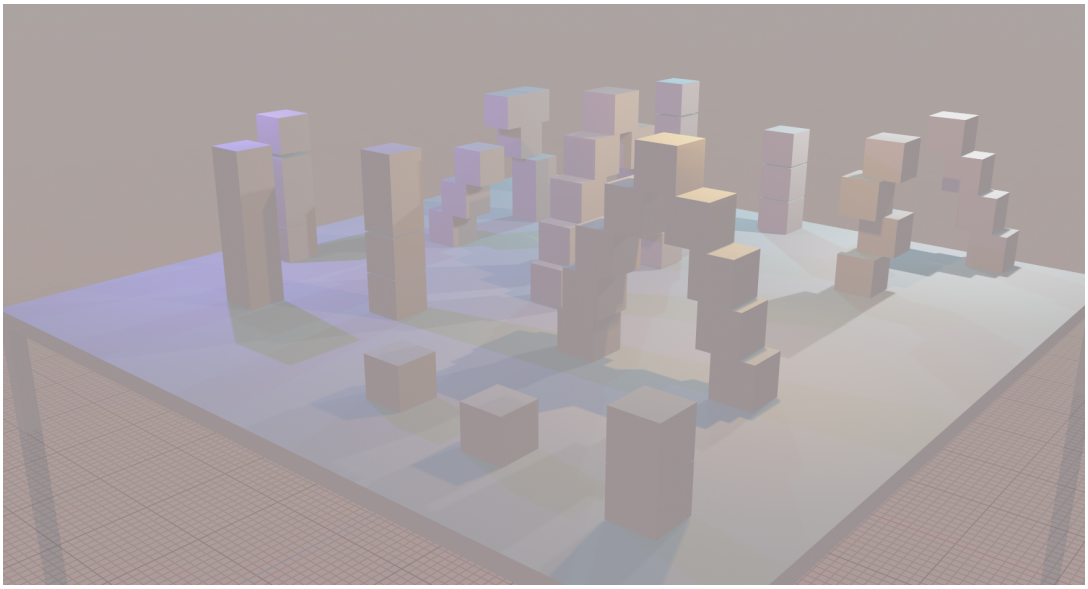


Figure 10.1: Ground and Maze

For the case study, this maze will be used. Finally, every cube inside the maze and the ground have tagged with the number 0. This tag is used by the **Tag Component** and can be reached from scripts. This way, it will be easier to distinguish object types.

10.2 Swarm Robot

Swarm robots are nested entities. On top, they have a mesh which is a **pyramid** shape. They are tagged with the number 1. They also have physics materials and the properties of the 3D Physics material component are as follows:

- **Mass:** 1.0
- **Friction Coeff:** 0.1
- **Restitution:** 0.5
- **Fixed Rotations:** Rotation around x, y and z-axis are **not fixed**

As seen above, **Robot bodies** are dynamic objects both registered in the graphics and physics worlds of the OP Engine. These pyramid shapes also have a **3D collider**

wrapping up the shape of the robot. This 3D collider is a box estimate of the pyramid shape for performance issues. The robot body is shown in the figure below:

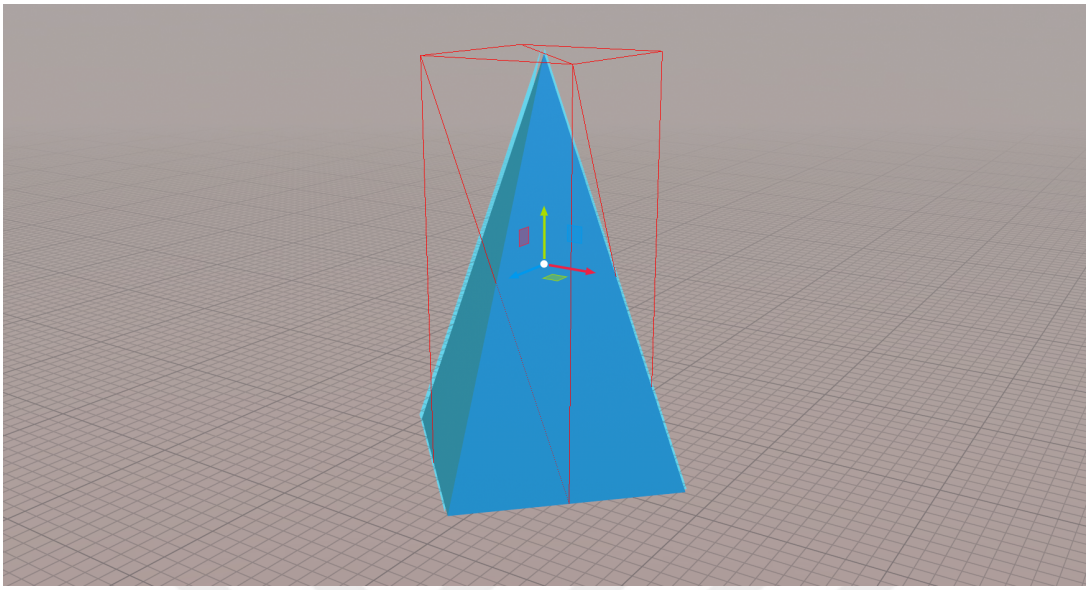


Figure 10.2: Robot body and its collider

Secondly, a robot also has a child entity which is the **Sensor**. This sensor is a **ghost object** in the eyes of the **Graphics System**. It is not rendered but it is there. It has a **Physics Material** and **3D Physics collider**. Since it is the child of **Robot Body**, it also moves accordingly as the robot starts moving. A sensor has the following physical properties:

- **Mass:** 1.0
- **Friction Coeff:** 0.1
- **Restitution:** 0.5
- **Fixed Rotations:** Rotation around x, y and z-axis are **not fixed**
- **Kinematic:** This object is marked as kinematic. This means that it moves independently of the physical world.

In addition, the collider of **Sensor** does not have **Contact Response**. This means that if any object (particularly other robots or obstacles) contacts the sensor of a robot. No

physical force will be generated and only the **Callback Functions** that are explained in 8.1 will be called. This way, collisions with sensors are efficiently checked. The following figure shows the collider of a **Robot**'s **Sensor**:

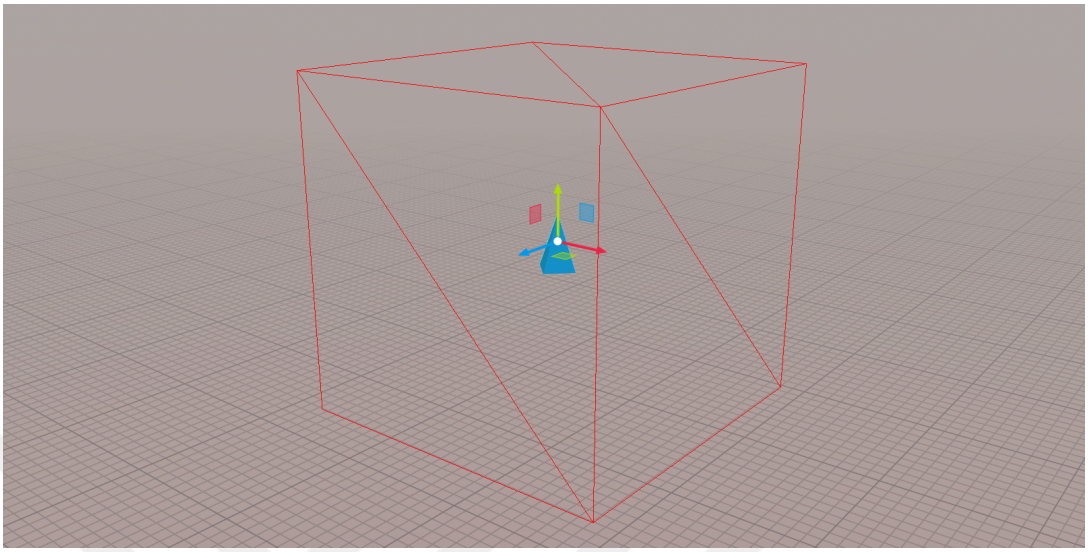


Figure 10.3: Robot's Sensor's Collider

10.3 Scene

In the end, the scene consists:

- 4 Different Lights emitting different colors. Each one of them produces different shadows on the objects.
- Maze Entity. This entity consists of the maze. They are all built from standard cube meshes and connected to the physical world.
- Ground Entity. The ground has a platform and 4 pillars that support the platform. They are also connected to the physical world.
- Robot Manager. Robot manager is the entity that contains all the **Robots** in the scene. A script called **RobotManager** is attached to this entity and some properties of the swarm can be arranged from this script.
- Robots. Each robot also contains a sensor entity and there is also a script called

SwarmRobot that is attached to the **Robot** entity. It is responsible for changing the velocity of the robot according to the input received from the sensor.

In the end, the whole **test scene** looks like the following figure:

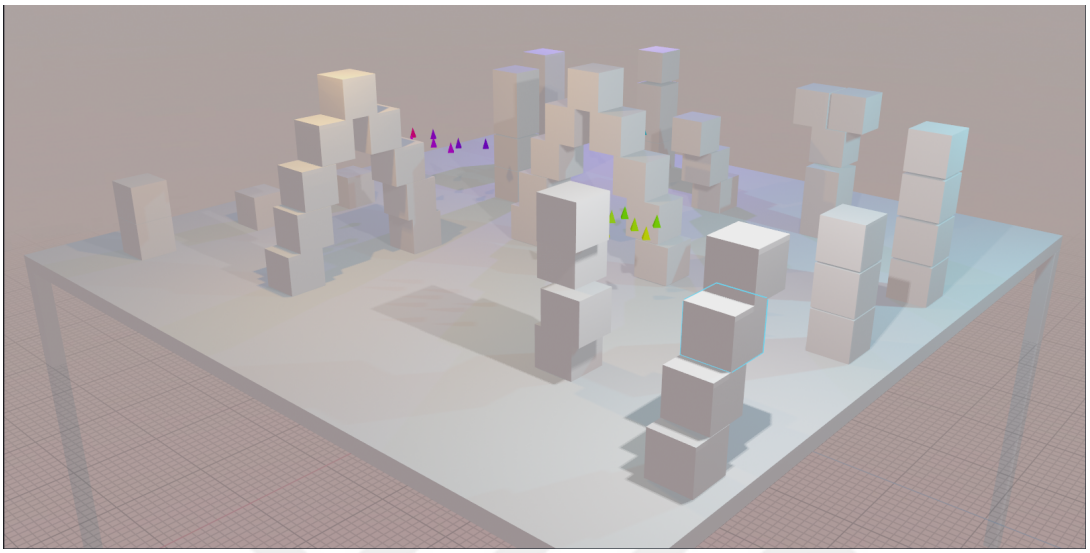


Figure 10.4: Complete Test Scene

10.4 Boids Algorithm

In the previous sections, the structure of the scene has been explained. This case study implements the Boids Algorithm by Craig Reynolds [43]. This algorithm models **flocking** behavior animals such as **fish** and **birds**. It can also be used for swarm robots since they are mimicking the behavior of flocking animals. In boids algorithm, there are 3 main forces that are considered to keep flocking behavior. These are:

- **Cohesion:** Robot senses its environment and gets closer to the average point of its neighboring.
- **Alignment:** Robot senses velocity vectors of its neighbors and tries to align its own velocity vector accordingly.
- **Separation:** If a robot is getting much closer to some of the neighbors, then it will try to avoid them.

In addition to these, an obstacle avoidance force is also being applied to the robots if they are too much close to an obstacle. The algorithm that is used in this case study is shown below:

Algorithm 9 OP Boids Algorithm

1: $currentPosition \leftarrow Position$
2: $cohesion \leftarrow (0.0, 0.0, 0.0)$
3: $separation \leftarrow (0.0, 0.0, 0.0)$
4: $alignment \leftarrow (0.0, 0.0, 0.0)$
5: $obsAvoidance \leftarrow (0.0, 0.0, 0.0)$
6: $avgRobotPos \leftarrow (0.0, 0.0, 0.0)$
7: **for** $robot$ in $neighborRobots$ **do**
8: $robotPos \rightarrow robot.Position$
9: $robotVec \leftarrow robotPos - currentPosition$
10: $robotVec.Normalize()$
11: $currentVelocity \leftarrow Velocity$
12: $currentVelocity.Normalize()$
13: $cosAngle \leftarrow dot(currentVelocity, robotVec)$
14: $cosTreshold \leftarrow \cos(robotManager.SensorAngle * \frac{\pi}{180.0})$
15: **if** $cosAngle \geq cosTreshold$ **then**
16: $robotLinearVelocity \leftarrow robot.Body.LinearVelocity$
17: $avgRobotPos \leftarrow robotPos + avgRobotPos$
18: $alignment \leftarrow robotLinearVelocity + alignment$
19: $dist \leftarrow Distance(robotPos, currentPosition)$
20: **if** $dist \leq robotManager.SeparationDistance$ **then**
21: $distFactor \leftarrow \frac{1.0}{dist^3+0.001}$
22: $dir \leftarrow currentPosition - robotPos$
23: $dir.Normalize()$
24: $separation \leftarrow dir * distFactor + separation$

In the above algorithm, neighboring robots are being sensed and **separation**, **cohesion**, and **alignment** values are calculated. These values will be used to determine the **target velocity** of the robot. In addition to this, obstacle avoidance is also added. The algorithm for obstacle avoidance is shown below. This algorithm is the extension

of the algorithm above:

Algorithm 10 OP Boids Obstacle Avoidance Algorithm

1: **for** $obstacle$ in $Obstacles$ **do**
2: $obstaclePos \leftarrow obstacle.Position$
3: $dist \leftarrow Distance(obstaclePos, currentPosition)$
4: **if** $dist \leq robotManager.ObstacleAvoidanceDistance$ **then**
5: $distFactor \leftarrow \frac{1}{dist^3+0.001}$
6: $dir \leftarrow currentPosition - obstaclePos$
7: $dir.Normalize()$
8: $obstacleAvoidance \leftarrow dir * distFactor + obstacleAvoidance$

With this addition, the obstacle avoidance value is also calculated. This algorithm is directly applied in the **OnUpdate** function of the **SwarmRobot.cs** script that is attached to each robot.

10.5 Scripts

There are 3 scripts that are being used in this case study. These are:

- **RobotManager:** Responsible for keeping the constants that are being used by each robot. It is attached to the parent entity that contains all of the robots. Thus, robots can reach these variables. Some of these variables are:
  - Cohesion = 1.0f;
  - Alignment = 1.0f;
  - Separation = 1.0f;
  - ObstacleAvoidance = 1.0f;
  - SensorAngle = 280.0f;
  - SeparationDistance = 0.5f;
  - ObstacleAvoidanceDistance = 0.5f;
  - Speed = 1.0f;

- **Sensor:** Sensor scripts have blank **OnUpdate** functions but their **OnCollisionStarted** and **OnCollisionEnded** functions are implemented. They hold an array of obstacles and robots that are sensed. Each time a collision callback function is called, this array is updated. This script is attached to the child entity of each robot. So robots can access this array. **SwarmRobot:** This script is attached to every swarm robot in the scene. They take constants from **RobotManager** and sensed objects from **Sensor**. And after that, they use **Boids algorithm** as it is explained in algorithms 9 and 10. Swarm robots also use **Linear Interpolation** to change their velocity.

10.6 Performance Test

In this scenario, we have a maze that is constructed with static cubes, a platform, and its pillars. In this section, the performance of the engine is tested by an increasing number of robots in the scene and how much time is needed for each frame is calculated by profiling.

The specs of the test computer are shown below:

- **Processor:** Intel(R) Core(TM) i7-9700F CPU @ 3.00GHz 3.00 GHz
- **Installed RAM:** 32.0 GB
- **System type:** 64-bit operating system, x64-based processor
- **OS:** Windows 10 Pro
- **GPU:** NVIDIA GeForce RTX 2060

In the table below, the performance of the engine for different numbers of robots is shown. The first column is the number of robots, the second column is how much time is needed to render one frame on average (in milliseconds) and the third column represents the approximate fps value:

Table 10.1: Performance metrics for case study

Number of Robots	Render Time	Fps
20 robots	2.610 ms	383 fps
40 robots	3.790 ms	263 fps
60 robots	5.016 ms	199 fps
80 robots	8.358 ms	119 fps
100 robots	12.048 ms	83 fps
120 robots	17.597 ms	56 fps
140 robots	22.587 ms	44 fps
160 robots	26.201 ms	38 fps
180 robots	37.154 ms	26 fps
200 robots	44.245 ms	22 fps

As is seen in the table, fps starts to drop under 60 after adding more than 100 robots. Each robot has two scripts attached to it and they are invoked at every time step. In addition, each robot and its sensor are also added to the physics and graphics worlds. This increases the load on the scripting and physics systems mainly. The rendering system handles hundreds of meshes well but it is also going to be a burden if massive amounts of meshes are going to be used.

Fps also changes according to cohesion coefficient in **RobotManager** script. When this coefficient is increased, robots start moving closer to each other and as a result, fps drops significantly because the physics system encounters more collisions. For this particular case study, the cohesion coefficient is kept constant.

CHAPTER 11

CONCLUSION & FUTURE WORK

11.1 Conclusion

In this thesis, a fully working **simulation** engine is implemented by observing the software architectures of game engines. It can render realistic **PBR** scenes and deal with **rigid body dynamics** in real-time without performance issues. The software architecture is designed in such a way that different libraries can be easily implemented. So the architecture of the software is flexible. It uses C# scripting like Unity [5], however, the difference is that the GUI of the Engine is not written with C#, and for that, imgui [24] has been used. It currently uses OpenGL [30] inside the rendering system. However, as is discussed, different APIs like Metal [17], Vulkan [16] or DirectX [35] can be added to the structure of the engine. The importance of OP Engine is that it can be used as a **test bed** for different applications and experiments in the future which will grant the developers who understand the architecture full control. Moreover, since the architecture is explained practically, researchers can use this architecture as a reference to build their own simulation engines which deal with rigid body dynamics.

11.2 Future Work

With this study, the bone structure of the simulation engine has been finished. It has all the basic features to build a scene and run physics simulations. However, in the future, new Graphics APIs will be added. In addition, new Graphics techniques will also be implemented. These are **atmospheric rendering**, **terrain generation**,

use of tesselation shaders, **frustum culling**, and implementation of **forward plus rendering**. A rework for shadow maps of directional lights. Font rendering will also be going to be applied. Model loading and skeletal animations will be finished as well. In the future, OP Engine is going to be used for creating cutting-edge and performant algorithms in the field of computer graphics and simulations.

A

REFERENCES

[1] Mathworks, "Simulink - simulation and model based design - matlab & simulink." https://www.mathworks.com/products/simulink.html. Accessed: 2022-10-12.

[2] OpenModelica, "Welcome to openmodelica - openmodelica." https://openmodelica.org/. Accessed: 2022-10-12.

[3] Su2, "Su2 | multiphysics simulation and design software." https://su2code.github.io/. Accessed: 2022-10-12.

[4] E. Games, "The most powerful real-time 3d creation tool - unreal engine." https://www.unrealengine.com/en-US. Accessed: 2022-10-13.

[5] Unity, "Unity real-time development platform | 3d, 2d, vr & ar engine." https://unity.com/. Accessed: 2022-10-13.

[6] Godot, "Godot engine - free and open source 2d and 3d game engine." https://godotengine.org/. Accessed: 2022-10-13.

[7] L. Lindner, "Etengine." https://github.com/Illation/ETEngine. Accessed: 2022-10-13.

[8] Y. Chernikov, "Hazelengine." https://github.com/TheCherno/Hazel. Accessed: 2022-10-13.

[9] Yatiyr, "Opengine." https://github.com/yatiyr/OP-Engine. Accessed: 2022-10-13.

[10] E. Catto, "Box2d." https://box2d.org/. Accessed: 2022-10-13.

[11] erwincoumans, "Bullet3." https://pybullet.org/wordpress/. Accessed: 2022-10-13.

[12] J. Gregory, *Game engine architecture*, pp. 11–12. AK Peters/CRC Press, third ed., 2018.

[13] A. D. Rosario, “Bafta games awards 2021 hades takes home top prize complete winners list.” https://deadline.com/2021/03/bafta-games-awards-2021-hades-takes-home-top-prize-complete-winners-list-1234721128/. Accessed: 2022-10-19.

[14] BetterExplained, “Understanding big and little endian byte order.” https://betterexplained.com/articles/understanding-big-and-little-endian-byte-order/. Accessed: 2022-10-19.

[15] J. Gregory, *Game engine architecture*. AK Peters/CRC Press, third ed., 2018.

[16] K. G. AMD, DICE, “Vulkan | cross platform 3d grahics.” https://www.vulkan.org/. Accessed: 2022-10-23.

[17] Apple, “Metal overview - apple developer.” https://developer.apple.com/metal/. Accessed: 2022-10-23.

[18] M. Stevanovic, *Advanced C and C++ Compiling*. Apress, 2014.

[19] nothings, “stb single-file public domain libraries for c/c++.” https://github.com/nothings/stb/blob/master/stb_image.h. Accessed: 2022-10-26.

[20] gabime, “Fast c++ logging library..” https://github.com/gabime/spdlog. Accessed: 2022-10-26.

[21] syoyo, “Tiny openexr image loader/saver library.” https://github.com/syoyo/tinyexr. Accessed: 2022-10-26.

[22] g truc, “Opengl mathematics (glm).” https://github.com/g-truc/glm. Accessed: 2022-10-26.

[23] skypjack, “Gaming meets modern c++ - a fast and reliable entity component system (ecs) and much more.” https://github.com/skypjack/entt. Accessed: 2022-10-26.

[24] ocornut, “Dear imgui: Bloat-free graphical user interface for c++ with minimal dependencies.” https://github.com/ocornut/imgui. Accessed: 2022-10-26.

[25] CedricGuillemet, “Immediate mode 3d gizmo for scene editing and other controls based on dear imgui.” https://github.com/CedricGuillemet/ImGuizmo. Accessed: 2022-10-26.

[26] assimp, “The official open-asset-importer-library repository. loads 40+ 3d-file-formats into one unified and clean data structure..” https://github.com/assimp/assimp. Accessed: 2022-10-26.

[27] jbeder, “A yaml parser and emitter in c++.” https://github.com/jbeder/yaml-cpp. Accessed: 2022-10-26.

[28] mono, “Mono | cross platform, open source .net framework.” https://www.mono-project.com/. Accessed: 2022-10-26.

[29] Dav1dde, “Webinterface for glad.” https://github.com/dav1dde/glad-web. Accessed: 2022-10-26.

[30] K. Group, “Opengl the indstry standard for high performance graphics.” https://www.opengl.org/. Accessed: 2022-10-26.

[31] glfw, “A multi-platform library for opengl, opengl es, vulkan, window and input.” https://github.com/glfw/glfw. Accessed: 2022-10-26.

[32] google, “A collection of tools, libraries, and tests for vulkan shader compilation..” https://github.com/google/shaderc. Accessed: 2022-10-26.

[33] epezent, “Immediate mode plotting.” https://github.com/epezent/implot. Accessed: 2022-11-18.

[34] google, “Google fonts.” https://fonts.google.com/. Accessed: 2022-11-18.

[35] Microsoft, “Directx graphics and gaming - win32 apps | cross platform 3d microsoft learn.” https://learn.microsoft.com/tr-tr/windows/win32/directx. Accessed: 2022-11-30.

[36] Khronos, “Framebuffer object - opengl wiki.” https://www.khronos.org/opengl/wiki/Framebuffer_Object. Accessed: 2022-12-04.

[37] T. Harada, J. McKee, and J. C. Yang, "Forward+: Bringing deferred lighting to the next level.," in *Eurographics (Short Papers)*, pp. 5–8, 2012.

[38] Y. Saad, *Iterative methods for sparse linear systems*, p. 11. SIAM, 2003.

[39] A. Scholten, "Autumn forest 04." https://polyhaven.com/a/autumn_forest_04. Accessed: 2022-10-13.

[40] B. Karis and E. Games, "Real shading in unreal engine 4," *Proc. Physically Based Shading Theory Practice*, vol. 4, no. 3, p. 1, 2013.

[41] A. Lebedev, "30 free pbr materials." https://gumroad.com/l/ekRhc. Accessed: 2022-10-13.

[42] M. Project, "Embedding mono | mono." https://www.mono-project.com/docs/advanced/embedding/. Accessed: 2022-10-12.

[43] C. W. Reynolds, "Flocks, herds and schools: A distributed behavioral model," in *Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, pp. 25–34, 1987.