

C++

A MASTER PIECE BY

FRANC POUHELA

3D GAME ENGINE DEVELOPMENT

Learn how to Build a Cross-Platform 3D
Game Engine with C++ and OpenGL





3D GAME ENGINE DEVELOPMENT

**Learn how to Build a Cross-Platform 3D
Game Engine with C++ and OpenGL.**

Copyright © 2024 Franc Pouhela Germany
ALL RIGHTS RESERVED.

No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews. For more information on permission to reproduce selections from this book, write to madsycode@gmail.com. Every effort has been made to prepare this book to ensure the accuracy of the information presented. However,

the information contained in this book is sold without warranty, either express or implied. Neither the author nor its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Portions of this book have been generated and edited with the assistance of Large Language Models (LLMs) tools. While Artificial Intelligence (AI) has been employed to enhance the content, including the writing and correction of certain sections, the final output is a collaborative effort involving human oversight and refinement. The book's authorship remains primarily human, and any errors, biases, or inaccuracies are unintentional. The integration of AI technology is intended to contribute to the creative process and improve the overall quality of the content. Readers should be aware that the information provided herein is not solely the product of human authorship, and the book's content may reflect the capabilities and limitations of AI language models.

E-book created March 30, 2024

ISBN:

Contents

Acknowledgement

Foreword

- 1 [Piracy](#)
- 2 [Errata](#)

Preface

- 1 [Expectation](#)
- 2 [Limitation](#)

I Getting Started

1 Introduction

- 1.1 [What is a Game Engine?](#)
 - 1.1.1 [Popular Game Engines](#)
- 1.2 [Game Engine Design](#)
 - 1.2.1 [High-Level Architecture](#)
- 1.3 [Development Tools](#)
 - 1.3.1 [Coding Environment](#)
 - 1.3.2 [Build System](#)
 - 1.3.3 [Graphics APIs](#)
 - 1.3.4 [Scripting APIs](#)
 - 1.3.5 [Graphical User Interface](#)
- 1.4 [Summary](#)

2 Environment Setup

- 2.1 Platform Support
- 2.2 Windows Setup
 - 2.2.1 Installing VS Code
 - 2.2.2 Visual C++ Compiler
 - 2.2.3 Installing CMake
 - 2.2.4 Installing Conan
- 2.3 Linux Setup
 - 2.3.1 Installing VS Code
 - 2.3.2 Installing GNU Compiler Tools
 - 2.3.3 Installing CMake
 - 2.3.4 Installing Conan
- 2.4 Project Setup
 - 2.4.1 Project Directory Tree
 - 2.4.2 VS-Code Configurations
 - 2.4.3 Build Scripts
 - 2.4.4 CMakeLists.txt Files
 - 2.4.5 Conan Configuration
- 2.5 First Compilation
 - 2.5.1 Assertions
 - 2.5.2 Inlining
 - 2.5.3 Console Logging
 - 2.5.4 Helpers
 - 2.5.5 Hello World
 - 2.5.6 First Compilation
 - 2.5.7 First Execution
 - 2.5.8 Include Warnings
 - 2.5.9 Debugging

3 Core Engine Design

- 3.1 Core Application
 - 3.1.1 Layers Management
 - 3.1.2 Retrieving Layers
 - 3.1.3 Attaching Layers
 - 3.1.4 Detaching Layers
 - 3.1.5 Creating First Layer
 - 3.1.6 Dispatcher Interface
- 3.2 Window Event Handling
 - 3.2.1 Linking GLFW
 - 3.2.2 Window Inputs
 - 3.2.3 Window Events
 - 3.2.4 Window Abstraction
 - 3.2.5 Showing the Window

II Rendering Graphics

4 Introduction to OpenGL

- 4.1 OpenGL Versions
- 4.2 Why Version 3.3?
- 4.3 Rendering Pipeline
 - 4.3.1 Vertex Specification
 - 4.3.2 Vertex Shader
 - 4.3.3 Geometry Shader
 - 4.3.4 Clipping
 - 4.3.5 Rasterization
 - 4.3.6 Fragment Shader
 - 4.3.7 Per-Fragment Operations
 - 4.3.8 Frame Buffer Output
- 4.4 Future of OpenGL

4.4.1 [OpenGL vs. Vulkan](#)

4.4.2 [Important Note](#)

5 Basic Rendering

5.1 Renderer Components

5.1.1 OpenGL Loader

5.1.2 Initializing OpenGL Context

5.1.3 OpenGL Mathematics

5.1.4 Vertex Buffer

5.1.5 Frame Buffer

5.2 Implementing Shaders

5.2.1 Shader Compilation

5.2.2 GLSL Shaders

5.2.3 Shader Abstraction

5.3 Entity Component System

5.3.1 What is ECS?

5.3.2 Integrating ECS

5.3.3 Scene Abstraction

5.3.4 Rendering First Frame

5.4 Loading 3D Models

5.4.1 Assimp Library

5.4.2 Model Class

6 Advanced Rendering

6.1 Phong Reflection Model

6.1.1 Ambient Component

6.1.2 Diffuse Component

6.1.3 Specular Component

6.2 Physically Based Rendering

- 6.2.1 Basic Shader
- 6.2.2 Basic Material
- 6.2.3 PBR Microfacets
- 6.3 Scene Illumination
 - 6.3.1 Point Lights
 - 6.3.2 Directional lights
 - 6.3.3 Spotlights
- 6.4 Rendering Textures
 - 6.4.1 Material Textures
 - 6.4.2 Texture Mapping
 - 6.4.3 Normal Mapping
- 6.5 Global Illumination
 - 6.5.1 Scene Skybox
 - 6.5.2 Diffuse Irradiance
 - 6.5.3 Specular Irradiance
 - 6.5.4 HDR Tone Mapping
- 6.6 Rendering Shadows
 - 6.6.1 Shadow Mapping Technique
 - 6.6.2 Applying Shadow Mapping
 - 6.6.3 Improving Shadow Quality
- 6.7 Bloom Effect
 - 6.7.1 Implementing Bloom Effect

7 Skeletal Animation

- 7.1 Important Concepts
 - 7.1.1 Animation Keyframe
 - 7.1.2 Keyframe Interpolation
 - 7.1.3 Joint Transformation
- 7.2 Implementing Animation

- 7.2.1 Animation Data
- 7.2.2 Skeletal Mesh
- 7.2.3 Model Animator
- 7.2.4 Loading Animation

III Physics & Scripting

8 Physics Simulation

- 8.1 Integrating NVIDIA PhysX
 - 8.1.1 Getting Started with NVIDIA PhysX
 - 8.1.2 Initialization and Configuration
 - 8.1.3 Rigid Bodies and Colliders
 - 8.1.4 Handling Physics Interactions
 - 8.1.5 Integration with Application

9 Runtime Scripting

- 9.1 Scripting with Lua
 - 9.1.1 Integrating Lua Dependencies
 - 9.1.2 Initialization and Configuration
 - 9.1.3 Script Handle
 - 9.1.4 Script Context
 - 9.1.5 Introduction to Lua
 - 9.1.6 Implementing Script Modules
 - 9.1.7 Integration with Application

IV Assets & Serialization

10 Asset Management

- 10.1 Assets Definition
 - 10.1.1 Material Asset

- 10.1.2 Texture Asset
- 10.1.3 Skybox Asset
- 10.1.4 Model, Script, Scene, Asset
- 10.2 Asset Registry
 - 10.2.1 Registry Class
 - 10.2.2 Adding Assets
- 10.3 Integrating Assets
 - 10.3.1 Updating Components
- 10.4 Updating Pipelines
 - 10.4.1 Loading Assets

11 Serialization

- 11.1 Integrating YAML
- 11.2 YAML-CPP
 - 11.2.1 yaml-cpp Syntax
 - 11.2.2 YAML File Layout
 - 11.2.3 Library Setup
 - 11.2.4 yaml-cpp Helpers
- 11.3 Scene Serialization
 - 11.3.1 Serializing Entities
 - 11.3.2 Deserializing Entities
- 11.4 Assets Serialization
 - 11.4.1 Serializing Assets
 - 11.4.2 Deserializing Assets
- 11.5 Integrating in Application
 - 11.5.1 Testing Serialization
 - 11.5.2 Serialization Results
 - 11.5.3 Loading Scene from File

v Editor Interface

12 Scene Editor

12.1 Integrating ImGui

 12.1.1 How to Use Dear ImGui

12.2 Creating a GUI Context

 12.2.1 Helpers and Configurations

 12.2.2 Event Types

 12.2.3 Input Fields

 12.2.4 Widget Interface

 12.2.5 Context Interface

12.3 Creating Windows

12.4 Viewport Window

 12.4.1 Rendering to Buffer

 12.4.2 Creating the Viewport

 12.4.3 Showing the Viewport

 12.4.4 Scroll, Drag and Resize

12.5 Hierarchy Window

12.6 MenuBar Window

12.7 Inspector Window

 12.7.1 Control Interface

 12.7.2 Transform Control

 12.7.3 Camera Control

 12.7.4 Info Control

 12.7.5 Directional Light Control

12.8 Resource Window

13 Game Executable

VI Fazit

14 Conclusion

14.1 Perspective

14.2 Outlook

Bibliography

Biography

Acknowledgement

With heartfelt humility and profound gratitude, I wish to express my profound appreciation to my family, who wholeheartedly supported me in bringing this remarkable book to fruition.

Additionally, I extend my sincere thanks to you, as your desire to learn has played an integral role in bringing this project to fruition. Finally, I extend my gratitude to God for bestowing upon me life, inspiration, and unwavering strength.

Foreword

To maximize your learning experience with this book, we have prepared a dedicated GitHub repository for you. This repository houses all the code and additional materials featured within the book. If you encounter any challenges while progressing through the various parts of this book, you can easily clone specific versions of the source code from the repository's distinct branches.

1 Piracy

Copyright piracy is an ongoing problem. We take intellectual property protection and licensing very seriously. If you come across any illegal copies of this book, in any form, on the Internet, please contact us as soon as possible using the email address below. We appreciate your support in safeguarding our writers' safety and our ability to supply you with crucial information.

2 Errata

Despite our best efforts to ensure the accuracy of our material, errors occasionally occur. If you notice a mistake in this book, whether it is in the text or in the code, we would appreciate it if you could let us know.

By doing so, you save other readers time and effort while also helping us improve future editions of this work. Send us an email at madsycode@gmail.com if you find any errors.

Preface

This book is not meant for individuals who are completely novices in C++ programming or any other programming language in general. To fully benefit from the material provided in this book, you should possess a grasp of fundamental coding concepts, including object-oriented programming, inheritance, template functions, pointers, and related topics. Rest assured, no prior experience with OpenGL is required, but it is highly recommended.

1 Expectation

You are embarking on a journey to build a cross-platform 3D game engine from the ground up using C++ and OpenGL. The adventure begins with setting up a versatile development environment and a robust build system, laying the foundation for the challenges that lie ahead. We dive into core functionalities such as message logging, window events input handling, etc. As the journey progresses, we venture into more advanced terrain, tackling the implementation of critical features such as graphics rendering, physics, scripting, serialization, etc. Finally, it culminates with the implementation of a graphical user interface to improve interaction with the engine's features and game creation.

2 Limitation

It is essential to emphasize that this book is not asserting that the game engine built within its pages is production-ready. Instead, our aim is to lay down solid foundations for you to shape according to your aspirations. Approach the knowledge presented here with a discerning eye, recognizing it as a guiding light rather than absolute truth. This book stands as a testament to the extent of my own odyssey in learning computer graphics and programming, a testament to the ever-evolving quest for knowledge in this dynamic field.

M.Eng. Franc Pouhela, Germany, March 30, 2024

Part I

Getting Started

1 Introduction

All hard work brings a profit, but mere talk leads only to poverty. (King Solomon)

I have always been fascinated by how games come together. The sheer magic of objects moving, animations unfolding, and effects blending seamlessly always amazed me. I wanted to dive into that world, understand it, and eventually create it myself. So, I began by learning the basics of programming. Then, I took the plunge into creating simple games, one step at a time. After many years of learning and experimenting, it finally clicked for me. I realized that I had to share this knowledge with others who might be going through the same experience.

Developing a custom game engine in today's landscape might raise eyebrows among many, as there's an abundance of really good free and open-source alternatives readily accessible. To challenge this prevailing wisdom might appear unconventional at best. However, I firmly believe that there are compelling reasons for you to explore the intricate world of game engine development. This pursuit not only fuels the industry's motion but also provides a remarkable avenue for expanding your technological proficiency. Building a custom game engine allows you,

as a developer, to tailor the engine to your specific needs and preferences. This gives you greater control over the game development process and allows you to create games or applications that are unique and differentiated from those made with commonly used engines.

Now, let me be clear from the outset: Creating a game engine from scratch is not a task for the fainthearted. It's a pursuit that demands a solid grasp of computers, programming, mathematics, and physics. It beckons you to embark on a journey where you will unravel intricate concepts and then transmute them into lines of code capable of breathing life into a vast array of virtual experience ideas. Once you emerge from the tunnel's far end, you will possess a solid foundation for designing and developing high-performance game engines. With that being said, I would like to encourage you not to give up.

1.1 What is a Game Engine?

A game engine is a piece of software or a platform that provides the core functionalities and tools required to develop, run, and manage video games. It serves as the technological backbone for game development, offering a set of prebuilt modules, libraries, and systems that streamline the creation of interactive digital experiences. Game engines are also versatile

tools used not only for video game development but also for simulations, architectural visualization, training, and interactive media across various industries. Their continuous evolution and improvement empower game developers to focus on creativity and gameplay while leveraging the engine's technological capabilities.

Over the past two decades, the development of game engines has undergone a remarkable transformation. Advancements in hardware and software have led to the creation of highly sophisticated engines, enabling breathtaking graphics, physics, and interactivity. Open-source engines such as Godot and Unreal have democratized game development, allowing a wider audience to create immersive experiences. Additionally, mobile games have exploded, demanding lightweight engines for portable devices. The industry has also seen a shift toward real-time ray tracing and Virtual Reality (VR) technology. In summary, the last 20 years have witnessed a rapid evolution in game engine technology, making game development more accessible and pushing the boundaries of what's possible in gaming.

1.1.1 Popular Game Engines

Numerous acclaimed game engines are being embraced worldwide. Here is a list of some of the prominent ones.

- **Unity:** A cross-platform game engine popular among independent developers and small studios, valued for its user-friendly interface and broad platform support, including PC, console, mobile, and web.
- **Unreal:** A powerful and widely used game engine developed by Epic Games, known for creating various game genres, including first-person shooters, action games, and Role-Playing Games (RPGs).
- **Godot:** Developed as an open-source game engine, Godot stands out for its user-friendly interface, flexibility, and community-driven development. It offers a node-based scene system that simplifies game design and enables both 2D and 3D game development
- **GameMaker:** A favorite among hobbyists and beginner game developers, GameMaker's simplicity and drag-and-drop interface facilitate 2D game development without extensive programming.
- **CryEngine:** Developed by Crytek, CryEngine stands out for its high-quality graphics and extensive platform compatibility, making it suitable for first-person shooters and action games.

- **Lumberyard**: Developed by Amazon and based on CryEngine, Amazon Lumberyard is a free cross-platform game engine used for various genres of games, including Massively Multiplayer Onlines (MMOs), RPGs and open-world games.
-

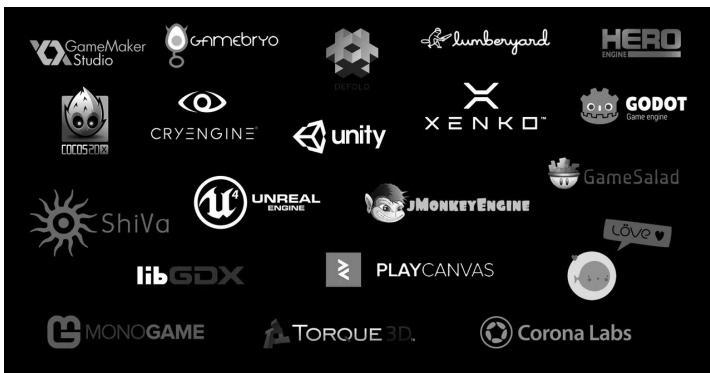


Figure 1.1: Popular Engines
Ayinla

These are merely a handful of notable game engines. Numerous others exist, each possessing its own distinctive set of features and capabilities. In addition, there are independent game engines being developed by content creators on platforms such as YouTube, such as the Hazel Engine [Yan Chernikov](#) and the Cave Engine [Teres](#).

1.2 Game Engine Design

There are many different approaches to the design of game engines, and the specific design of a game engine will depend on the needs and goals of the game to which it is used. Quite often, we have the tendency to associate the term "game engine" with the idea of a user interface for creating games, but this perception is somewhat misconceived. A game engine does not necessarily need to encompass all the tools commonly found in existing popular ones. However, some common features of game engines include:

- **Rendering Engine:** Responsible for generating the visual output of the game, including 3D models, textures, and special effects. It utilizes graphics Application Programming Interface (API) such as OpenGL, Vulkan, or Direct3D to draw the game world on the screen.
- **Physics Engine:** Simulates the physical properties of objects in the game world, such as gravity, mass, and friction. It enables realistic interactions between objects and the environment. Software Development Kits (SDKs) such as NVIDIA PhysX [NVIDIA](#), Bullet Physics [development team](#) are commonly used for this purpose.
- **Sound Engine:** manages the audio aspects of the game, including sound effects, music, and dialogue. It may utilize libraries such as SDL_Mixer, OpenAL, or FMOD to play sound files and control

audio properties such as volume, pitch, and spatialization.

- **Scripting Engine:** The core of the game engine is responsible for managing the game state, updating the game world, and handling user input. It can employ programming languages such as C#, Lua, or Python to implement game mechanics and AI.
- **Assets Management:** In charge of managing the game's assets, such as 3D models, textures, fonts, and audio files. This system may include tools for importing, exporting, and organizing assets.
- **Scene Editor:** The interface through which developers interact with the engine's features to make their game ideas come true.

1.2.1 High-Level Architecture

In contrast to many other industries, the gaming sector operates without a governing body that prescribes a definitive blueprint for constructing a proper architecture. This duality has both advantages and disadvantages. On the positive side, it grants developers the freedom to experiment with their own innovative ideas. However, on the downside, the absence of standardized guidelines makes it challenging to gauge the effectiveness of one's approach objectively. Naturally, best practices exist and should be considered to avoid redundant mistakes.

Game engines are intentionally designed to be flexible, creating a fertile ground for nurturing creativity and enabling artists to bring their unique visions to life. This diversity aligns with the broad spectrum of purposes, requirements, and objectives that each game serves, underscoring the distinctive character of every development effort.

(Figure 1.2) describes the high-level architecture of what we are aiming at this book. The engine's fundamental capabilities are encapsulated within a shared library in Windows or a shared object in Linux. In computer science, a shared library, is a file designed to be simultaneously employed by multiple programs. During runtime, the library's modules are dynamically loaded into the memory space of an executable, such as **Game.exe** and **Empy.exe**, facilitating access to the engine's defined functionalities. **Game.exe** represents the polished, distributable product, which allows others to enjoy the game made using the engine, while **Empy.exe** serves as a versatile tool for editing game scenes and preparing it for distribution.

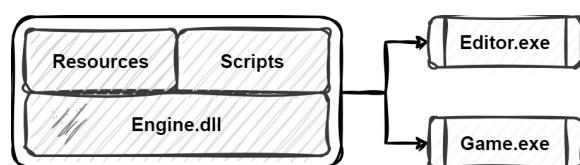


Figure 1.2:

High-Level Architecture

In order to interact with the engine's functionalities, we leverage scripting. This involves crafting supplementary code on top of the existing engine to amplify its capabilities and interface with its features. More insights into this process will be provided in a subsequent chapter.

1.3 Development Tools

A widely accepted consensus is that C++ has firmly established itself as the preferred programming language for game engines, primarily due to its performance optimization capabilities, cross-platform compatibility, and ability to interface with low-level APIs. Its support for object-oriented design aids in structuring complex engine components. The vast C++ ecosystem, coupled with the language's use in established engines like Unreal Engine, has cemented its role in the industry. While C++ dominates, the development of game engines also sees contributions from languages such as C#, Rust, and Python, each chosen based on project-specific needs and goals. This book employs C++ because of the reasons mentioned.

1.3.1 Coding Environment

Navigating the landscape of Integrated Development Environments (IDEs) in the realm of C++ development can be a daunting journey. Developers often encounter a multitude of options, each with its own strengths and complexities. Selecting the right IDE, configuring it, and ensuring compatibility with the C++ development ecosystem can pose significant challenges. However, our choice in this book is Visual Studio Code (VS-Code), a lightweight and versatile code editor developed by Microsoft. VS-Code boasts a rich ecosystem of extensions and plugins that specifically cater to C++ development needs. It combines a user-friendly interface with powerful functionality, making it an excellent choice for both novice and experienced developers. Its flexibility allows us to tailor our development environment to suit the requirements of game engine development seamlessly.

1.3.2 Build System

A build system is a crucial component that automates the process of compiling source code into executable programs or libraries. It encompasses a set of rules and procedures for transforming source files, such as C++ source code and resource files, into machine-readable binary code that can be run on a computer. The build system's primary objectives are to manage dependencies, determine which source files need to be

recompiled based on changes, and ensure that the compilation process is efficient and error-free. **CMake**, which we will utilize in this book, plays a pivotal role in simplifying the often complex and platform-dependent process of building C++ projects, making development more manageable and accessible.

C++ projects often rely on external libraries and packages, each with its own set of dependencies and version requirements. This intricate web of dependencies can lead to compatibility issues, version conflicts, and a significant overhead in configuring and maintaining the development environment. However, **Conan**, a C++ package manager, significantly simplifies this process. Conan simplifies the management of dependencies by providing a centralized repository (<https://conan.io/center>) for C++ packages and ensuring version control and consistency. Automates the retrieval, integration, and building of dependencies, avoiding the tedious and error-prone task of manual dependency management. In this book, we will harness the power of Conan to seamlessly manage and integrate the necessary libraries and packages, allowing us to focus on the core aspects of game engine development without the complexities of dependency wrangling.

1.3.3 Graphics APIs

Graphics APIs are essential software frameworks that enable developers to create visually captivating and interactive applications, particularly in the realm of computer graphics and game development. These APIs serve as intermediaries between the application and the graphics hardware, allowing programmers to harness the full potential of modern Graphics Processing Units (GPUs). Prominent graphics APIs include OpenGL, DirectX, Vulkan, and Metal, each tailored to specific platforms and offering varying degrees of control and performance optimization. The choice of a graphics API often depends on the target platform, project requirements, and the developer's familiarity with a particular framework. These APIs enable developers to render 2D and 3D graphics, implement shaders, manage textures, and achieve real-time visual effects, forming the foundation of immersive gaming experiences and graphical applications on a multitude of devices.

This book employs **OpenGL** [for High Performance Graphics](#) as graphics API for several compelling reasons. OpenGL's cross-platform compatibility ensures that the skills acquired are adaptable to a wide range of operating systems and devices. It is industry standard graphics API widely used in game development, providing readers with information on industry best practices. OpenGL's

openness, performance control, and educational value make it an accessible and versatile choice for learning the foundations of modern graphics programming.

1.3.4 Scripting APIs

Scripting in game engines is the art of imbuing games with interactivity, dynamism, and custom behavior through the use of scripted code. It serves as the bridge between the core engine of the game and the dynamic elements that make each gaming experience unique. Scripting empowers developers to define gameplay mechanics, create complex AI behaviors, orchestrate in-game events, and shape the player's journey. It offers a flexible and efficient way to iterate on game design without delving into the engine's source code, allowing for rapid prototyping and adjustments.

This book incorporates **Lua** [PUC-Rio](#) as its scripting language due to several compelling reasons. Lua's lightweight and embeddable nature adds extensibility to the game engine without introducing complexity. Its simplicity, popularity in game development, and versatility make it an ideal choice for scripting game logic and behavior, and it boasts a supportive community with ample resources. Lua's seamless integration with C/C++ allows us to leverage its

dynamic capabilities while maintaining performance-critical components.

1.3.5 Graphical User Interface

Game engines such as Unity and Unreal allow you to create your games in a Graphical User Interface (GUI) environment and then export the final result as a runtime for specific platforms such as PC, Smartphone, Console, etc. The engine we will be building in this book also needs an editor. (Figure 1.3) shows a preview.



Figure 1.3: EmPy Engine Preview

We will create a game-editor environment using **Dear ImGui** [Omar](#). Dear ImGui is a bloat-free graphical user interface library for C++. It outputs optimized vertex buffers that you can render anytime in your 3D-

pipeline-enabled application. It is fast, portable, renderer-agnostic, and self-contained. Dear ImGui is designed to enable fast iterations and empower programmers to create content creation tools and visualization/debug tools. Dear ImGui, it is particularly suited to integration in games engines, full-screen applications, embedded applications, or any applications on console platforms where operating system features are non-standard.

1.4 Summary

Throughout the course of this book, we'll leverage multiple tools and libraries to enhance our engine with features such as sound effects, 3D model loading, serialization, and more. Each new tool will be introduced as needed, seamlessly integrating them into our development process.

2 Environment Setup

A sluggard's appetite is never filled, but the desires of the diligent are fully satisfied. (King Solomon)

In this chapter, our primary emphasis will be on installing essential tools that are instrumental in facilitating the development of our game engine. If your development environment is exclusively Windows or Linux-based, you can conveniently skip to the relevant sections below that pertain to your specific platform. Platform-specific dependencies will be addressed comprehensively in those sections, ensuring a seamless setup process. Beyond this initial setup, the remainder of the book's content remains consistent across platforms, with only occasional minor adjustments required along the way.

If you are not familiar with the tools presented in this chapter, we recommend that you install the provided versions by following the described steps.

2.1 Platform Support

The code provided in this book is designed to be compatible with Windows, Linux, and MacOS. However, it's important to note that our testing has primarily been focused on Windows and Linux environments. Unfortunately, we were unable to conduct testing on a Mac computer due to the unavailability of such hardware. As a result, our discussions and examples in this book will primarily revolve around Windows and Linux implementations.

Property	Windows	Linux
GPU	GeForce RTX 3052 Ti	Mesa Intel® HD Graphics 4600
CPU	11th Gen Intel® Core™ i7-11800	Intel® Core™ i5-4460
RAM	32.0 GB	16.0 GB

OS

Windows 11 Pro, 64-bit

Ubuntu 22.04.1 LTS, 64-bit

Table 2.1: Platform Specifications

See (Table 2.1) for specific details about the platforms used during the development.

2.2 Windows Setup

We will proceed to install all essential tools for Windows. Feel free to bypass this section if you intend to use a custom IDE.

2.2.1 Installing VS Code

You can obtain VS-Code by visiting the official download page at <https://code.visualstudio.com/download>. On this page, you'll find both 32-bit and 64-bit versions available. Select the appropriate version compatible with your operating system. Not many people still use 32-bit computers nowadays. After downloading, run the ".exe" file and proceed with the default configuration settings. Additionally, ensure that you enable the "Add to PATH" option during the process.

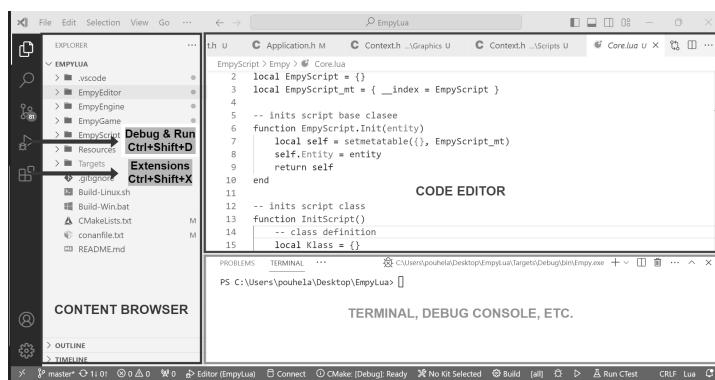


Figure 2.1: Visual Studio Code

The default color theme is "dark", but you can easily change that in the settings or with the shortcuts: **CRTL+K**, **CRTL+T**. (Figure [2.1](#)) shows a preview of the code editor with a description of its main components.

○ **C/C++ Extension Pack Installation** : VS-Code does not natively incorporate C++ development support, which requires the installation of extensions for this purpose. To add extensions to VS-Code, access the Extensions View through either the side menu or by using the keyboard shortcut **Ctrl+Shift+X**. See (Figure [2.1](#)).

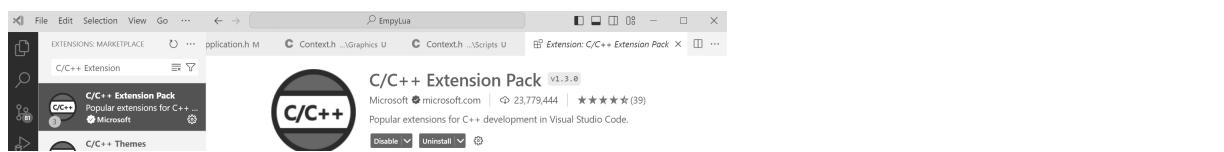


Figure 2.2: C/C++ Extension Pack

This package includes a set of popular extensions for C++ development:

- **C/C++ IntelliSense**: This extension provides robust support for C and C++, such as Debugging, IntelliSense, etc.
- **C/C++ Themes**: These themes enhance the coding experience for C and C++ developers, offering various visual styles for syntax highlighting.
- **CMake**: This extension helps with CMake-based project management and build processes within Visual Studio Code.
- **CMake Tools**: CMake Tools simplifies the configuration and building of CMake projects, streamlining development workflows.

See (Figure [2.2](#)) for visual guidance.

2.2.2 Visual C++ Compiler

In order to compiler our source code on Windows, we need a compiler on the system that can be used by CMake to generate build files. This step is redundant if Visual Studio Development Tools for C++ are already installed on your system. You can download Visual Studio Build Tools, using the following link: <https://visualstudio.microsoft.com/downloads/>.

Scroll down to the "**All Downloads**" section where you can find the "**Microsoft Visual C++ Redistributable for Visual Studio 202x**". Download it and run the installation executable. Select the **Desktop Development with C++** module and click Install to finish (Figure 2.3). This book uses the 2022 version, which is the latest at the time of writing. You can refer to the official documentation using the following link to learn more: <https://code.visualstudio.com/docs/cpp/config-msvc>.

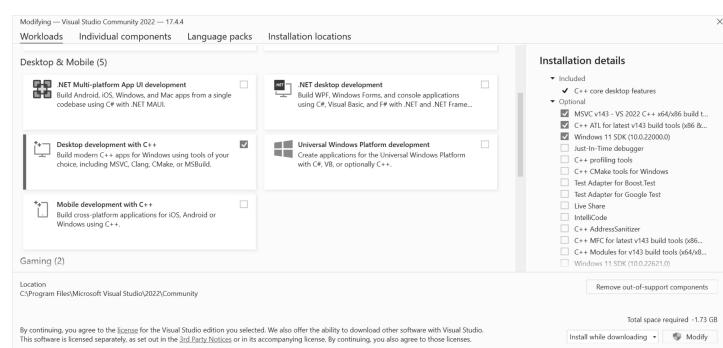


Figure 2.3: Microsoft Visual C++ Toolset

To check whether Microsoft Visual Studio Compiler (MVSC) is installed on your machine, you can open *Developer Command Prompt* from the Windows menu button by simply entering: **Developer Command Prompt**. The following command can be used to check the installed version.

```
c1
```

2.2.3 Installing CMake

CMake is a powerful open source tool designed for cross-platform use. It achieves this by generating native build files tailored to your compiler and platform, using configuration files specific to your compiler and platform. To install CMake on Windows, precompiled binaries are readily available as Microsoft Installer (MSI) packages on the official CMake website: <https://cmake.org/download/>. For 64-bit systems, download the **Windows x64 Installer**, and for 32-bit systems, opt for the **Windows i386 Installer**. Run the downloaded MSI package to install CMake,

ensuring that you select the option to **Add CMake to the system PATH for all users**. This ensures a global installation accessible to third-party software like Vs-Code. The CMake version installed in this book is the "**3.27.2**" but the minimum required version is **3.16.0**. Check your installation using the following command:

```
cmake -version
```

2.2.4 Installing Conan

As mentioned in previously in the introduction, Conan is a package manager for C and C++ which provides a large variety of libraries for the developers. Conan relies on Python [Foundation](#), so you need to have Python installed on your machine.

☞ **Installing Python:** You can download and install python using the following link: <https://www.python.org/downloads/>. During installation, make sure to check the box that says **Add Python X.Y to PATH**, where X.Y is the Python version you are installing. At the time of this writing, **3.11.4** is the latest version.

☞ **Installing Conan:** Open the Command Prompt on your PC. You can do this by searching for "cmd" or "Command Prompt" in the Start menu and run the following command to install Conan **version 1.61.0**:

```
pip install conan==1.61.0
```

Check your installation version using the following command:

```
conan -v
```

2.3 Linux Setup

Now that the Windows setup is done, let us do the same for Linux. The following installation steps are specific to Ubuntu. It's important to note that certain tools mentioned in this section require root access for installation. If you do not intend to use Linux, you may skip this section.

2.3.1 Installing VS Code

Installing VS-Code on Ubuntu is simple. You can simply use the system's built-in software installer, as shown in (Figure 2.4).

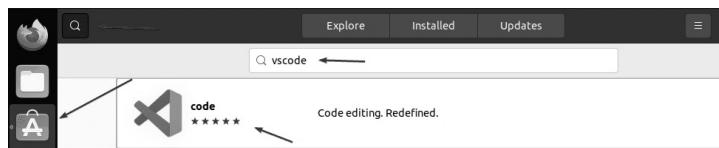


Figure 2.4: VS Code Ubuntu Installation

See (Section 2.2) on VS-Code installation for Windows to add the C/C++ Extension Pack for C++ development.

2.3.2 Installing GNU Compiler Tools

Just as we installed the compiler tools for Windows, a similar procedure is required for Linux. Ubuntu, for instance, doesn't always come with preinstalled GNU compiler tools, so we'll need to perform a manual installation. Use the command below to install all the required compiler tools for Linux:

```
sudo apt-get install build-essential gdb
```

Check the installation using the following command:

```
gdb -v
```

2.3.3 Installing CMake

Just as VS-Code, CMake can also be installed using the system's built-in package installer. See (Figure 2.5). Check the installation using the same command used for Windows:

```
cmake -version
```

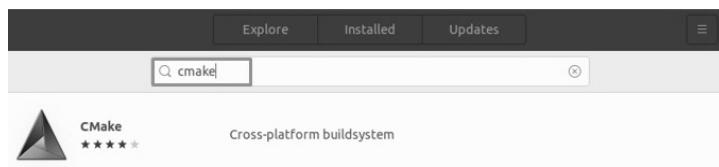


Figure 2.5: CMake
Ubuntu

2.3.4 Installing Conan

Following are the steps to install Conan on your Linux operating system. Under normal circumstances, Ubuntu should already have "python" with most of its tools, such as "pip" or "python3.11-venv" installed in a ready-to-use global virtual environment. But we are considering the worst-case scenario.

☞ **Installing Python:** You can download and install Python and its extensions using the following command:

```
# update packet manager
sudo apt-get update

# install python3 and virtual env. manager
sudo apt-get install python3 python3-pip python3.11-venv

# create python virtual env. called "empy"
sudo python3 -m venv empy
```

☞ **Installing Conan:** Install Conan using the following commands:

```
# activate empy environment
source ~/empy/bin/activate

# install conan 1.61.0
pip install conan==1.61.0
```

2.4 Project Setup

Now that we have completed all the major platform installations, let us proceed with the project setup. This section explores the directory structure and essential files for setting up our project. A well-organized

project structure can make development and collaboration more manageable. It is often a challenge for me to find a good name for my projects. The name I have chosen for the game engine developed in this book is "**Empy**". Feel free to use any other name you like.

2.4.1 Project Directory Tree

The project is organized into several directories, each serving a specific purpose. (Figure [2.6](#)) is an overview of the project directory tree. This helps to separate different aspects of the project, such as the core engine library, editor, and game executables, making it easier to manage and maintain each part individually. Throughout this book, this directory tree will be extended to accommodate our ongoing progress.

Please ensure that you create a similar directory tree structure, including all the specified files and folders, before continuing.

The configuration files in the "**.vscode**" directory will help streamline development with the VS-Code IDE. The project will be built using the provided build scripts for Linux and Windows: "**EmpyWin.bat**", "**EmpyLinux.sh**". The "**CMakeLists.txt**" files define each module that should be build, and the "**conanfile.txt**" specifies external libraries.



Figure 2.6: Project Directory Tree

2.4.2 VS-Code Configurations

To enable the C/C++ extension in VS-Code, which we installed earlier, we need to set up a folder named **".vscode."** Within this folder, we place three essential JavaScript Object Notation (JSON) files: **"c_cpp_properties.json"**, **"launch.json"**, and **"tasks.json"** as presented in the folder tree (Figure 2.6). Now, let us dive into the purpose of each file.

The file depicted in (Listing 2.1) serves the purpose of configuring the C/C++ extension within VS-Code, enhancing our coding experience by providing accurate IntelliSense suggestions, error checking, and syntax

highlighting. It is important to point out the fact that this configuration does not impact the compilation process; its effects are solely focused on improving the coding experience.

Listing 2.1: .vscode/c_cpp_properties.json

```
{  
    "configurations": [  
        {  
            "includePath": [  
                "${workspaceFolder}/EmpyEngine/includes",  
                "${workspaceFolder}/EmpyEditor/src",  
                "${workspaceFolder}/EmpyGame/src",  
            ],  
  
            "configurationProvider": "ms-vscode.cmake-tools",  
            "defines": ["EMPY_EXPORT", "EMPY_ENABLE_LOG"],  
            "intelliSenseMode": "${default}",  
            "cppStandard": "c++17",  
            "cStandard": "c17",  
            "name": "cpp-dev"  
        }  
    ],  
    "version": 4  
}
```

These configurations include paths to directories with essential header files, the configuration provider (e.g., ms-vscode.cmake-tools), IntelliSense mode settings, and language standards (e.g., c/c++17). These settings ensure precise code analysis and compilation within our workspace.

The IntelliSense mode is set to "default" to ensure compatibility with the current platform. However, it's also possible to specify a different mode, such as "clang-x64," if needed. Both preprocessor macros, `EMPY_EXPORT` and `EMPY_ENABLE_LOG` will be introduced later as we define our CMake files. They help with syntax highlighting.

Listing 2.2: .vscode/launch.json

```
{
  "version": "0.2.0",
  "configurations":
  [
    // windows
    {
      "type": "cppvsdbg",
      "request": "launch",
      "name": "Editor Win32",
      "program": "Empy.exe",
      "console": "integratedTerminal",
      "cwd": "${workspaceFolder}/Targets/Debug/bin",
    },
    {
      "type": "cppvsdbg",
      "request": "launch",
      "name": "Game Win32",
      "program": "Game.exe",
      "console": "integratedTerminal",
      "cwd": "${workspaceFolder}/Targets/Debug/bin",
    },
    // Linux
    {
      "type": "cppdbg",
      "request": "launch",
      "name": "Editor Linux",
      "miDebuggerPath": "/usr/bin/gdb",
      "cwd": "${workspaceFolder}/Targets/Debug/bin",
      "program": "${workspaceFolder}/Targets/Debug/bin/Empy",
    },
    {
      "type": "cppdbg",
      "request": "launch",
      "name": "Game Linux",
      "miDebuggerPath": "/usr/bin/gdb",
      "cwd": "${workspaceFolder}/Targets/Debug/bin",
      "program": "${workspaceFolder}/Targets/Debug/bin/Game",
    }
  ]
}
```

The **launch.json** file (Listing 2.2) serves as the configuration for our debugger launcher. It is essential to create separate configurations for each platform. For Windows, two configurations are specified, "**Editor Win32**" and "**Game Win32**". These settings employ the "**cppvsdbg**" debugger that was previously installed with the Microsoft Compiler

Tools (Section [2.4](#)). The debug output is directed to the integrated terminal. The programs to be debugged are "**Empy.exe**" and "**Game.exe**". Similarly, we have two configurations for Linux: "**Editor Linux**" and "**Game Linux**". These configurations use the "**cppdbg**" debugger, which we installed with the GNU Compiler Tools (Section [2.4](#)).

Listing 2.3: .vscode/tasks.json

```
{  
    "version": "2.0.0",  
    "tasks": [  
        {  
            "type": "shell",  
            "label": "Empy-Debug",  
            "problemMatcher": ["$gcc"],  
            "options": { "cwd": "${workspaceFolder}" },  
            "group": { "kind": "build", "isDefault": true },  
  
            "windows": {  
                "args": ["Debug"],  
                "command": "${workspaceFolder}/EmpyWin.bat"  
            },  
  
            "linux": {  
                "sudo": true,  
                "args": ["Debug"],  
                "command": "${workspaceFolder}/EmpyLinux.sh"  
            }  
        },  
        {  
            "type": "shell",  
            "label": "Empy-Release",  
            "problemMatcher": ["$gcc"],  
            "group": { "kind": "build" },  
            "options": { "cwd": "${workspaceFolder}" },  
  
            "windows": {  
                "args": ["Release"],  
                "command": "${workspaceFolder}/EmpyWin.bat"  
            },  
  
            "linux": {  
                "sudo": true,  
                "args": ["Release"],  
                "command": "${workspaceFolder}/EmpyLinux.sh"  
            }  
        }  
    ]  
}
```

```
        }
    }
}
```

The **tasks.json** file (Listing 2.3) is used to configure tasks such as build, clean, etc. Because VS-Code does not know how to compile our project by default, we must instruct it exactly what to do by giving it a command or shell script that will be executed when a build request is triggered. First, the "**Empy-Debug**" task is defined as the default build task. This task uses a shell command. It is designed to match problems using the "**gcc**" problem matcher. On Windows, it is configured to run the "Debug" mode with arguments, invoking the "**EmpyWin.bat**" script from the project's folder. On Linux, it is set to run with elevated privileges "**sudo**" and uses the "**Debug**" arguments, executing the "**EmpyLinux.sh**" script. The second task, "**Empy-Release**", is designed for release builds and shares similar properties with the debug build-configuration.

2.4.3 Build Scripts

As previously stated, our build system relies on two script files, namely, "**EmpyWin.bat**" and "**EmpyLinux.sh**", to perform code compilation on their respective platforms. Now, let's examine their implementations.

Listing 2.4: Root/EmpyWin.bat

```
@echo off

rem output directory
set target="Targets\%1"

rem install conan dependencies
conan install . --install-folder %target% --build missing -s build_type=%1 -c
tools.system.package_manager:mode=install

rem generate cmake build files
cmake -S . -B %target% -DCMAKE_BUILD_TYPE=%1 -
DCMAKE_TOOLCHAIN_FILE="conanbuildinfo.cmake"

rem compile cmake build files
cmake --build %target% --config %1
```

Listing 2.5: Root/EmpyLinux.sh

```
#!/bin/bash

# output directory
target="Targets/$1"

# activate python env
source ~/empy/bin/activate

# install conan dependencies
conan install . --install-folder $target --build missing -s build_type=$1 -c
tools.system.package_manager:sudo=True -c tools.system.package_manager:mode=install

# generate cmake build files
cmake -S . -B $target -DCMAKE_BUILD_TYPE=$1 -
DCMAKE_TOOLCHAIN_FILE="conanbuildinfo.cmake"

# compile cmake build files
cmake --build $target --config $1
```

These scripts (Listing 2.4) and (Listing 2.5) are designed for building and handling dependencies using Conan and CMake. The only difference between the two files is that we need to activate the python virtual environment where Conan was installed in Linux. This is normally not required but if you had to create a virtual environment in (Section 2.3) to install Conan, you are also required to make sure the environment is also activated when using Conan.

Remember to grant execution permission to the 'EmpyLinux.sh' script on your Linux system; Use this command in the directory where your script is located:

```
sudo chmod +x EmpyLinux.sh
```

Both scripts take the build configuration as an argument ("Debug" or "Release") and set the target output directory based on the provided argument, creating a folder in the output directory called "**Targets**". Both scripts use Conan to install project dependencies in the specified target directories, ensuring that any missing dependencies are built. The build type and package manager mode are configured based on the provided argument. The scripts generate CMake build files using the specified

target directory and set the build type and CMake toolchain file based on the argument.

2.4.4 CMakeLists.txt Files

CMake can generate build files by combining multiple CMake configuration files in various locations. In the project directory tree (Figure 2.6), you can see that there is a "**CMakeLists.txt**" file in the root directory, as well as in all three subdirectories "**EmpyEngine**", "**EmpyEditor**", and "**EmpyGame**". These files define how CMake must generate the project's build files for the compiler. CMake will search for all specified **CMakeLists.txt** in the root and subdirectories to collect and aggregate all the necessary information before generating the build files. Let's take a close look at each CMakeLists.txt file.

Listing 2.6: EmPyEngine/CMakeLists.txt

```
project(Engine)

# gather all source files
file(GLOB_RECURSE sources ${CMAKE_CURRENT_SOURCE_DIR}/*.cpp)
file(GLOB_RECURSE headers ${CMAKE_CURRENT_SOURCE_DIR}/*.h)
add_library(${PROJECT_NAME} SHARED ${sources} ${headers})

# export engine symbols
target_compile_definitions(${PROJECT_NAME} PUBLIC
    -DEMPTY_EXPORT
)

# includes directories
target_include_directories(${PROJECT_NAME} PUBLIC
    ${CMAKE_CURRENT_SOURCE_DIR}/includes
    ${CONAN_INCLUDE_DIRS}
)

# link directories
target_link_directories(${PROJECT_NAME} PUBLIC
    ${CONAN_LIB_DIRS}
)

# link libraries
target_link_libraries(${PROJECT_NAME} PUBLIC
```

```
    ${CONAN_LIBS}
)
```

These CMake files must be named: CMakeLists.txt

The **EmPyEngine/CMakeLists.txt** file (Listing 2.6) is responsible for configuring the build process of the engine library. It gathers all source files (both .cpp and .h) from the current directory and its subdirectories to create a shared library. The `target_compile_definitions(...)` line specifies that the compiler should consider the `EMPY_EXPORT` preprocessor macro during the compilation phase. This allows the library to export symbols to a library, which can later be linked to our executables.

Next, we link to the include directory of the project. Since we will be using external libraries from Conan, we also need to add them as a target for our project. The keyword `PUBLIC` is used to tell CMake that any project that links to the engine library may also use the files found in those directories.

Listing 2.7: EmPyEditor/CMakeLists.txt

```
project(EmPy)

# gather source files
file(GLOB_RECURSE sources ${CMAKE_CURRENT_SOURCE_DIR}/*.cpp)
file(GLOB_RECURSE headers ${CMAKE_CURRENT_SOURCE_DIR}/*.h)
add_executable(${PROJECT_NAME} ${sources} ${headers})

# include directories
target_include_directories(${PROJECT_NAME} PRIVATE
    ${CMAKE_CURRENT_SOURCE_DIR}/src
)

# link libraries
target_link_libraries(${PROJECT_NAME} PRIVATE
    Engine
)
```

Listing 2.8: EmPyEditor/CMakeLists.txt

```
project(Game)
```

```

# gather source files
file(GLOB_RECURSE sources ${CMAKE_CURRENT_SOURCE_DIR}/*.cpp)
file(GLOB_RECURSE headers ${CMAKE_CURRENT_SOURCE_DIR}/*.h)
add_executable(${PROJECT_NAME} ${sources} ${headers})

# include directories
target_include_directories(${PROJECT_NAME} PRIVATE
    ${CMAKE_CURRENT_SOURCE_DIR}/src
)

# link libraries
target_link_libraries(${PROJECT_NAME} PRIVATE
    Engine
)

```

Both **EmpyEditor/CMakeLists.txt** (Listing 2.7) and **EmpyGame/CMakeLists.txt** (Listing 2.8) files help configure the build process for the editor and game executables. The editor CMake file, for instance, creates an executable named "**Empy**" using these collected source and header files. It then further specifies the include directories, particularly the "**src**" directory, which allows the executable to locate its own source files. Finally, the engine project is linked, indicating that the executable relies on the engine library to function. The same thing is done in the game's CMake file.

Listing 2.9: Root/CMakeLists.txt

```

cmake_minimum_required(VERSION 3.16 FATAL_ERROR)

# build output directories
set(CMAKE_LIBRARY_OUTPUT_DIRECTORY $<0:>${CMAKE_BINARY_DIR}/bin)
set(CMAKE_RUNTIME_OUTPUT_DIRECTORY $<0:>${CMAKE_BINARY_DIR}/bin)
set(CMAKE_ARCHIVE_OUTPUT_DIRECTORY $<0:>${CMAKE_BINARY_DIR}/lib)
set(CMAKE_PDB_OUTPUT_DIRECTORY $<0:>${CMAKE_BINARY_DIR}/lib)
set(EXECUTABLE_OUTPUT_PATH $<0:>${CMAKE_BINARY_DIR}/bin)

# cpp standard options
set(CMAKE_CONFIGURATION_TYPES Debug Release)
set(CMAKE_CXX_STANDARD_REQUIRED ON)
set(CMAKE_CXX_EXTENSIONS ON)
set(CMAKE_CXX_STANDARD 17)
project(EmpyEngine)

# enable console logging
if(${CMAKE_BUILD_TYPE} MATCHES Debug)

```

```
add_definitions(-DEMPTY_ENABLE_LOG)
endif()

# project subdirectories
add_subdirectory(EmpyEngine)
add_subdirectory(EmpyEditor)
add_subdirectory(EmpyGame)
```

The root CMake file (Listing 2.9) is the glue that is able to combine all sub-projects into one that CMake can use to generate build files. It begins by specifying the minimum required version of CMake, which is set to "3.16.0". It then proceeds to define the build output directories for various types of build artifacts, including libraries, executables, archives, and Program Database (PDB) files. These directories are configured relative to the binary directory using the `CMAKE_BINARY_DIR` environment variable. This environment variable helps us navigate the project directory without having to worry about the folder name. Next, we set options for the C++ standard, specifying that C++17 is required for the project. We also define two configuration types: "Debug" and "Release," which are used for different build configurations.

Conditional compilation is enabled for console logging based on the build type. If the build type is "Debug," the `EMPY_ENABLE_LOG` preprocessor definition is added, indicating that logging functionality should be included. We will use it in our code later.

2.4.5 Conan Configuration

The **conanfile.txt** (Listing 2.10) is a configuration file for the Conan package manager, which is commonly used for managing C and C++ library dependencies in a CMake-based project.

Listing 2.10: Root/conanfile.txt

```
[requires]
spdlog/1.12.0

[generators]
cmake

[options]
```

This file must also be named as given: "conanfile.txt"

Here's an explanation of its sections:

- **[requires]**: In this section, we can specify the external libraries or packages that our project depends on. Right now, we have added **spdlog** but more on that in a bit.
- **[generators]**: Here, we specify the output generators that will be used when generating build files. CMake is specified, indicating that Conan should generate CMake build files for our project.
- **[options]**: This section allows us to specify various options related to our project's dependencies. Depending on the libraries we use and their respective Conan recipes, we may need to specify options here to customize their behavior. For example, we can use this section to tell Conan to link a specific library dynamically rather than statically.

With these foundational configurations in place, we are now poised to delve into the actual coding and compilation of our project.

2.5 First Compilation

Now that we have completed all of the boilerplate, we would like to make a quick compilation of the project for the first time to test if everything we have done so far works. We will also use this opportunity to add basic features such as platform and compiler detection, console logging, function inlining, assertions, etc.

Insert the code depicted in (Listing 2.11) in your **Core.h** header file.

Listing 2.11: EmpyEngine/includes/Common/Core.h

```
#pragma once
#include <queue>
#include <vector>
#include <string>
#include <bitset>
#include <memory>
#include <sstream>
#include <fstream>
#include <assert.h>
#include <algorithm>
```

```

#include <functional>
#include <filesystem>
#include <unordered_map>

// include spdlog
#define FMT_HEADER_ONLY
#define SPDLOG_FMT_EXTERNAL
#include <spdlog/spdlog.h>
#include <spdlog/sinks/stdout_color_sinks.h>

// import, export
#ifndef EMPI_EXPORT
    #ifdef _MSC_VER
        #define EMPI_API __declspec(dllexport)
    #else
        #define EMPI_API __attribute__((visibility("default")))
    #endif
#else
    #ifdef _MSC_VER
        #define EMPI_API __declspec(dllimport)
    #else
        #define EMPI_API
    #endif
#endif

```

This file serves as the foundation for our project, including essential libraries, platform and compiler detection, logging, macros, etc. We start by including various C++ standard headers that are commonly used. If you still recall the CMake file found in "**EmpyEngine**" (Listing 2.6) you might recognize the `EMPI_EXPORT` preprocessor macro. Its purpose is to manage whether we want to export or import a symbol of our engine. This gives us control over what we choose to expose or conceal from external projects. Furthermore, we must consider platform differences, as Windows and Linux handle this process differently. Every time we need a class or a function to be exported, we will just precede its declaration by the keyword `EMPI_API`. We are also including the header files of "**spdlog**" which will be addressed in a bit.

2.5.1 Assertions

Assertions can be used to track down where the program's execution went wrong or even to block compilation from continuing if certain conditions are not met. A static assertion verifies a condition before the code is built,

whereas a runtime assertion fires an event during execution to prohibit your application from continuing. Because assertions are different across compilers, we use the if statement to ensure that the proper assertion call is made for the current compiler used. Add the following code snippet (Listing 2.12) to the bottom of your "Core.h" header file:

Listing 2.12: Common/Core.h

```
// runtime assertion
#define EMPTY_ASSERT assert

// static assertion
#if defined(__clang__) || defined(__gcc__)
    #define EMPTY_STATIC_ASSERT __Static_assert
#else
    #define EMPTY_STATIC_ASSERT static_assert
#endif
```

2.5.2 Inlining

Function in-lining is one of the numerous fascinating C++ features that can be quite handy when it comes to program performance. Whenever a function with the keyword `inline` is encountered during the compilation process, the compiler will try to replace that function's call with a copy of its definition. Inline functions are typically used when the function declarations are short and when they are called several times throughout a program. Once again, add the following code snippet (Listing 2.12) into your "Core.h" header file:

Listing 2.13: Common/Core.h

```
// function inlining
#if defined(__clang__) || defined(__gcc__)
    #define EMPTY_INLINE __attribute__((always_inline)) inline
    #define EMPTY_NOINLINE __attribute__((noinline))
#elif defined(_MSC_VER)
    #define EMPTY_INLINE __forceinline
    #define EMPTY_NOINLINE __declspec(noinline)
#else
    #define EMPTY_INLINE inline
    #define EMPTY_NOINLINE
#endif
```

You'll see that the `inline` keyword differs among compilers, so we need to make sure we're using the right version when building our project with different compilers. Most functions in our project will be preceded by the symbol `EMPY_INLINE` to instruct the compiler to always try to optimize it. This has a significant impact on performance when building the project for distribution.

2.5.3 Console Logging

Console logging is an essential aspect of C/C++ development. Although it can be achieved using the built-in `std::cout` function, we opt for a more sophisticated and efficient approach with "**spdlog**". "spdlog" streamlines the process, offering out-of-the-box support for asynchronous and thread-safe console and file logging, simplifying the heavy lifting required for robust and stylish log management.

In the **conanfile.txt** shown in (Listing 2.10), you will notice that "spdlog" was added as a requirement for our project. Furthermore, we have included "spdlog" header files into our "Core.h" header file. Given that "spdlog" can be utilized as a header-only library, we've opted for this approach by defining specific preprocessor directives before including its header file, as illustrated in (Listing 2.11). This implies that "spdlog" won't be built into a library prior to linking it with our project; instead, CMake will simply expose the include files to our project.

Add the following code (Listing 2.14) to your "**Core.h**" header file.

Listing 2.14: Common/Core.
h

```
// console logging
#ifndef EMPY_ENABLE_LOG
namespace Empy
{
    struct EMPY_API Logger
    {
        using SPDLog = std::shared_ptr<spdlog::logger>;
        EMPY_INLINE Logger()
        {
            m_SPD = spdlog::stdout_color_mt("stdout");
        }
    };
}
```

```

        spdlog::set_level(spdlog::level::trace);
        spdlog::set_pattern("%^[%T]: [%t] %v$");
    }

EMPY_INLINE static SPDLog& Ref()
{
    static Logger logger;
    return logger.m_SPD;
}

private:
    SPDLog m_SPD;
};

}

// logging macros
#define EMPY_TRACE(...) Empy::Logger::Ref()->trace(__VA_ARGS__)
#define EMPY_DEBUG(...) Empy::Logger::Ref()->debug(__VA_ARGS__)
#define EMPY_INFO(...) Empy::Logger::Ref()->info(__VA_ARGS__)
#define EMPY_WARN(...) Empy::Logger::Ref()->warn(__VA_ARGS__)
#define EMPY_ERROR(...) Empy::Logger::Ref()->error(__VA_ARGS__)
#define EMPY_FATAL(...) Empy::Logger::Ref()->critical(__VA_ARGS__)

#else
#define EMPY_TRACE
#define EMPY_DEBUG
#define EMPY_ERROR
#define EMPY_FATAL
#define EMPY_INFO
#define EMPY_WARN
#endif

```

This code snippet defines a log-system using the "spdlog" library. It creates a singleton structure called "Logger" that manages a shared instance of the **"spdlog::logger"** class. The logging level is set to trace, and a custom log message pattern is defined with the following parameters: *time, thread-id, and message*. The nice part of "spdlog" is its ability to print messages using different colors for each level.

We also includes logging macros such as `EMPY_TRACE` or `EMPY_DEBUG`, which utilize the logger for various log levels (trace, debug, info, warn, error, critical). These macros allow developers to log messages with different severity levels easily. If `EMPY_ENABLE_LOG` is not defined, the macros are defined as empty, ensuring that log messages are effectively disabled when not needed. E.g. Release Build. You remember the `EMPY_ENABLE_LOG`

which is defined in the root "**CMakeLists.txt**" file, when the project is build with debug configuration.

You may have noticed that the logger structure's name is preceded by the `EMPY_API` symbol, indicating our intention to expose it. It's worth noting that this export directive is not strictly necessary for classes or functions defined in a header file because they are already accessible through header inclusion.

2.5.4 Helpers

Finally, we also want to add some utilities that will help facilitate our development in the near future. Add the code in (Listing 2.15) to your **Core.h** file.

Listing 2.15: Common/Core.
h

```
// typeid
namespace Empy
{
    template <typename T>
    EMPY_INLINE constexpr uint32_tTypeID()
    {
        return static_cast<uint32_t>(reinterpret_cast<std::uintptr_t>(&typeid(T)));
    }
}

// free allocated memory
#define EMPY_DELETE(ptr) if (ptr != nullptr) { delete (ptr); ptr = nullptr; }
```

The `TypeID()` template function allows us to obtain unique identifiers for any type provided when calling it, which can be helpful for various tasks, such as type-based dispatch. This will help us later identify different types of events in our application. The `EMPY_DELETE` macro ensures safe memory management by deleting dynamically allocated objects and setting the corresponding pointers to `nullptr`, preventing memory leaks.

The entire code of the **Core.h** header file is presented in (Listing 2.16).

Listing 2.16: Common/Core.
h

```
#pragma once

#include <queue>
#include <vector>
#include <string>
#include <bitset>
#include <memory>
#include <sstream>
#include <fstream>
#include <assert.h>
#include <algorithm>
#include <functional>
#include <filesystem>
#include <unordered_map>

// include spdlog
#define FMT_HEADER_ONLY
#define SPDLOG_FMT_EXTERNAL
#include <spdlog/spdlog.h>
#include <spdlog/sinks/stdout_color_sinks.h>

// import, export
#ifndef EMPIY_EXPORT
    #ifdef _MSC_VER
        #define EMPIY_API __declspec(dllexport)
    #else
        #define EMPIY_API __attribute__((visibility("default")))
    #endif
#else
    #ifdef _MSC_VER
        #define EMPIY_API __declspec(dllimport)
    #else
        #define EMPIY_API
    #endif
#endif

// runtime assertion
#define EMPIY_ASSERT assert

// compile assertion
#if defined(__clang__) || defined(__gcc__)
    #define EMPIY_STATIC_ASSERT _Static_assert
#else
    #define EMPIY_STATIC_ASSERT static_assert
#endif

// function inlining
#if defined(__clang__) || defined(__gcc__)
    #define EMPIY_INLINE __attribute__((always_inline)) inline
```

```

#define EMPY_NOINLINE __attribute__((noinline))
#elif defined(_MSC_VER)
#define EMPY_INLINE __forceinline
#define EMPY_NOINLINE __declspec(noinline)
#else
#define EMPY_INLINE inline
#define EMPY_NOINLINE
#endif

// core features
namespace Empy
{
    // runtime type
    template <typename T>
    EMPY_INLINE constexpr uint32_tTypeID()
    {
        return static_cast<uint32_t>(reinterpret_cast<std::uintptr_t>(&typeid(T)));
    }

    // console logging
    struct EMPY_API Logger
    {
        using SPDLog = std::shared_ptr<spdlog::logger>;
        EMPY_INLINE Logger()
        {
            m_SPD = spdlog::stdout_color_mt("stdout");
            spdlog::set_level(spdlog::level::trace);
            spdlog::set_pattern("%^[%T]: [#%t] %v%$");
        }

        EMPY_INLINE static SPDLog& Ref()
        {
            static Logger logger;
            return logger.m_SPD;
        }

        private:
            SPDLog m_SPD;
    };
}

// logging macros
#ifndef EMPY_ENABLE_LOG
#define EMPY_TRACE(...) Empy::Logger::Ref()->trace(__VA_ARGS__)
#define EMPY_DEBUG(...) Empy::Logger::Ref()->debug(__VA_ARGS__)
#define EMPY_INFO(...) Empy::Logger::Ref()->info(__VA_ARGS__)
#define EMPY_WARN(...) Empy::Logger::Ref()->warn(__VA_ARGS__)

```

```

#define EMPY_ERROR(...) EmPy::Logger::Ref()->error(__VA_ARGS__)
#define EMPY_FATAL(...) EmPy::Logger::Ref()->critical(__VA_ARGS__)
#else
#define EMPY_TRACE
#define EMPY_DEBUG
#define EMPY_ERROR
#define EMPY_FATAL
#define EMPY_INFO
#define EMPY_WARN
#endif

// free allocated memory
#define EMPY_DELETE(ptr) if (ptr != nullptr) { delete (ptr); ptr = nullptr; }

```

2.5.5 Hello World

All we need to do in both **EmPy.h** and **EmPy.cpp** for now is to include the appropriate header files as shown in the following code snippets:

Listing 2.17: EmPyEngine/includes/EmPy.h

```

#pragma once
#include "Common/Core.h"

```

Listing 2.18: EmPyEngine/src/EmPy.cpp

```
#include "EmPy.h"
```

Following are the code snippets to define a hello world application for both the editor and the game executables:

Listing 2.19: EmPyEditor/src/Editor.cpp

```

#include <EmPy.h>

int32_t main(int32_t argc, char** argv)
{
    EMPY_INFO("Editor Started!");
    return 0;
}

```

Listing 2.20: EmPyGame/src/Game.cp

```
#include <EmPy.h>

int32_t main(int32_t argc, char** argv)
{
    EMPY_DEBUG("Game Started!");
    return 0;
}
```

2.5.6 First Compilation

Here's where the **tasks.json** ([Listing 2.3](#)). file we created earlier helps VS-Code in the compilation process. In VS-Code, there are at least two ways to compile a C++ project. The first and simplest way is to use the key combination **Ctrl + Shift + B**. The second approach involves using the editor's main menu by navigating to **Terminal -> Run Task**. This approach gives us the option to choose which build configuration to use as defined in tasks.json. See ([Figure 2.7](#)).

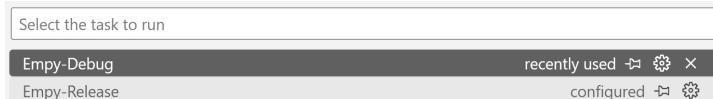


Figure 2.7: Build Tasks Menu

2.5.7 First Execution

Once the project is done compiling, if you open the build directory "**Root/Targets/Debug/bin**", you will find three files called: "**EmPy.exe**", "**EmPy.exe**", and "**EmPy.exe**" as we described earlier in ([Section 2.4](#)).

You can run the editor using either **F5** or by selecting the target program using the VS-Code Debug-View. See ([Figure 2.8](#)). This is possible because of the **launch.json** ([Listing 2.2](#)) file created in ([Section 2.4](#)).

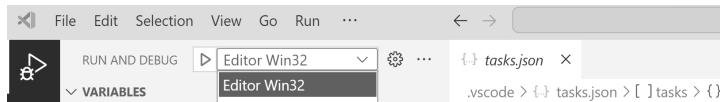


Figure 2.8: Run and Debug Menu

By clicking on the play button near the combo, you can finally run your selected executable. (Figure 2.9) shows the console output of the editor.

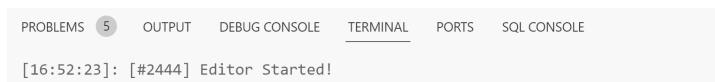


Figure 2.9: Editor's console output

2.5.8 Include Warnings

You probably wonder why spdlog's include statements are highlighted in red, as if there is an error in the code. Well, it is because VS-Code is not able to locate the spdlog include directory. We have not specified that in the `c_cpp_properties.json` as we did for our project source directories.

This is not required for your application to work, but it is quite helpful for your development experience within VS-Code, as this will enable VS-Code to properly highlight the syntax of spdlog.

To solve this, you can simply open the file located in your build folder under **Targets/Debug/conanbuildinfo.txt** where you will find the section **[includedirs]**. You will see that the path to the spdlog's include directory is listed. See (Figure 2.10).



Figure 2.10: Conan CMake Build Info

Copy both the "fmt" and the "spdlog" include path into your **c_cpp_properties.json**'s **includePath** section using double quotes as shown below (Listing 2.21).

Listing 2.21: .vscode/c_cpp_properties.json

```
{  
    "configurations":  
    [  
        {  
            "includePath": [  
  
                "C:/Users/pouhela/.conan/data/spdlog/1.12.0/_/_/package/7a4b7dbecf529  
c983055e2bf9c7700819db975cc/include",  
  
                "C:/Users/pouhela/.conan/data/fmt/10.0.0/_/_/package/2c52a23dc25833dd  
4323e8144302393f0061b96b/include",  
  
                "${workspaceFolder}/EmPyEngine/includes",  
                "${workspaceFolder}/EmPyEditor/src",  
                "${workspaceFolder}/EmPyGame/src"  
            ],  
  
            /* ... same as before ... */  
        }  
    ]
```

This process can be repeated every time a new library is added via Conan.

2.5.9 Debugging

Now that we can execute our program, let us take a quick look at how we can utilize the debugger to trace out any errors in our code.

☞ **Adding Break-Points** : You can add a debug break point using **F9** or by simply clicking on the line number on the side. See (Figure 2.11)

```
3 int32_t main(int32_t argc, char** argv)
4 {
5     EMPLY_INFO("Editor Started!\n\n");
6     return 0;
7 }
```

Figure 2.11: Debug Break Point

☞ **Debugger Actions** : You can hover over variables to see their current value. **F5** will continue your execution, **F10** will step over, **F11** will step into a function call, **Shift + F11** will step out, and **Shift + F5** will stop the debugging process.

The little menu on top of your code editor (Figure 2.12) can also be used to achieve the same results. You can use your mouse to hover over each button to see its meaning and shortcut.



Figure 2.12: Debugger Actions

Code not working? clone the branch "getting-started" on the GitHub repository and grant execution rights to the "EmpyLinux.sh" script on Linux.

<https://github.com/Madsycode/book-empy-engine>.

```
git clone -branch <branch-name> <repository-link>
```

3 Core Engine Design

A slack hand causes poverty, but the hand of the diligent makes rich (King Solomon)

The development of a complex project, such as a game engine, presents a significant number of challenges. One is how to effectively write and organizing the source code. Over time, numerous solutions have arisen to streamline this process for developers. However, C++ often carries the reputation of being complex and not easy to organize.

Throughout my journey working with C++, I've tried different software designs tailored precisely to specific requirements and applications. In this book, we'll leverage my own approach named Context Interface Layers (CIL).

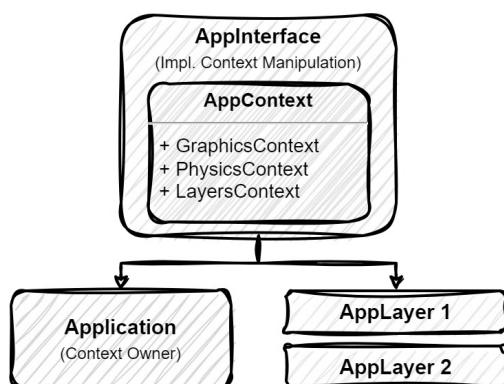


Figure 3.1:

Context Interface Layers Design

As depicted in (Figure 3.1), the application comprises a "**AppContext**", an "**AppInterface**", and potentially multiple "**AppLayers**". The "**AppContext**" serves as the repository for all application data, encompassing configurations, event management, and more.

Meanwhile, the "**AppInterface**" is responsible for implementing functions that facilitate interaction with the data housed within the context. Layers function as extensions, enabling the incorporation of new functionalities into the application. For instance, the game editor can be viewed as a layer enhancing the engine, providing users with a means to engage with the underlying engine features.

This approach offers scalability and flexibility, enabling code organization that is both extendable and versatile. The power of this approach lies in the separation of concerns. The data is separated from the logic, making it easier to maintain and extend. In addition, the methods implemented in the interface are reflected in both the application itself and all layers, eliminating the need to implement them separately.

3.1 Core Application

Let's dive in and apply the CIL approach we just talked about to build our engine core application. Go ahead and set up the folders and files as displayed in (Figure 3.2) within your project.

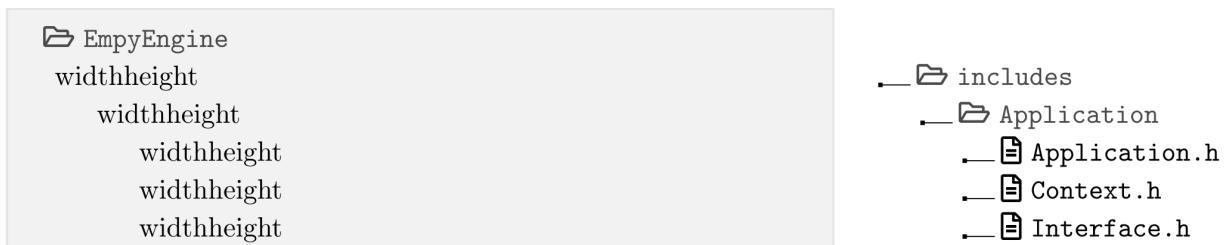


Figure 3.2: Application Directory

At the moment, the "AppContext" structure in (Listing 3.1) only has an array containing all the layers attached to the application. We are leveraging forward declaration of the interface to make possible for us to define the layer context.

Listing 3.1: Application/Context.h

```
#pragma once
#include "Common/Core.h"

namespace EmPy
{
    // forward declaration
    struct AppInterface;
```

```

// application context
struct ApplicationContext
{
    EMPLY_INLINE ~ApplicationContext()
    {
        for(auto& layer : Layers)
        {
            EMPLY_DELETE(layer);
        }
    }

    std::vector<AppInterface*> Layers;
};

}

```

The application interface (Listing 3.2), includes a member variable that points to an application context. It also contains some virtual member functions that any class inheriting from it can implement. The initial function that this interface defines is the one responsible for terminating the application loop. It is crucial to note that the interface neither creates nor destroys the context object; it merely references it.

Listing 3.2: Application/Interface.h

```

#pragma once
#include "Context.h"

namespace EmPy
{
    struct AppInterface

```

```

{
    EMPY_INLINE virtual ~AppInterface() = default;

protected:
    EMPY_INLINE virtual void OnUpdate() {}
    EMPY_INLINE virtual void OnStart() {}

private:
    friend struct Application;
    ApplicationContext* m_Context;
    uint32_t m_LayerID;
};

}

```

The `m_LayerID` serves to distinguish layers within the application. It is instrumental for our event system, helping it track event origins and registrations by differentiating who is posting an event and who's signing up to listen to a specific event within our application. Since layers can also be detached from the application, we need to be able to clear all potential callback function registrations from the event system, and this unique identifier enables that.

Listing 3.3: Application/Application.h

```

#pragma once
#include "Interface.h"

namespace Empy
{
    struct Application : AppInterface

```

```

{
    EMPTY_INLINE Application()
    {
        m_LayerID =TypeID<Application>();
        m_Context = new ApplicationContext();
    }

    EMPTY_INLINE ~Application()
    {
        EMPTY_DELETE(m_Context);
    }

    EMPTY_INLINE void RunContext()
    {
        while(true)
        {
            for(auto layer : m_Context->Layers)
            {
                layer->OnUpdate();
            }
        }
    }
}

```

The "**Application**" class so far initializes the context object and sets up a basic function to kick-start the main loop, as shown in (Listing 3.3). To prevent direct access by layers to the context information present in the application interface, from which all layers must inherit, we made it a private member. Given that the "**Application**" class is the rightful owner of the context, we've established a friendship between the

"AppInterface" and **"Application"**. This allows the application class to create, modify, and even destroy the context when needed. An additional advantage of this approach is that we can decide which methods can be called by the layers or the application.

3.1.1 Layers Management

Given our established design, it's time to implement functions responsible for attaching, detaching, and retrieving layers within the application. Leveraging the versatility of template functions in C++, we aim to streamline these operations seamlessly. This approach offers flexibility in managing layers, accommodating various definitions, and ensuring an efficient process for handling them within the application.

3.1.2 Retrieving Layers

The code snippet in (Listing 3.4) defines a templated function `GetLayer()`, aiming to retrieve a specific layer object from the context layer vector. It enforces that the specified type `Layer` derives from `AppInterface` using a static assertion. The function searches for a layer in the collection based on a unique identifier `m_LayerID` associated with each layer type. If found, it returns a pointer to the located layer; otherwise, it

returns `nullptr`. Add this function to your interface implementation.

Listing 3.4: Application/Interface.h

```
/* ... same as before ... */

template<typename Layer>
EMPTY_INLINE Layer* GetLayer()
{
    EMPTY_STATIC_ASSERT(std::is_base_of<AppInterface,
Layer>::value);
    auto itr = std::find_if(m_Context->Layers.begin(),
m_Context->Layers.end(), [this] (auto layer)
{
    return (layer->m_LayerID == typeid<Layer>());
});
if(itr != m_Context->Layers.end())
{
    return static_cast<Layer*>(*itr);
}
return nullptr;
}

//... same as before
```

3.1.3 Attaching Layers

The next function defined in (Listing 3.5) is designed to attach a new layer to the application. It ensures type safety by verifying that the layer type provided

derives from `AppInterface`. If a layer of the same type already exists, it logs an error and returns `nullptr`. Otherwise, it creates a new layer, adds it to the collection, initializes its properties, and returns a pointer to the attached layer. You can also see that we are leveraging the templated function `TypeID()` to make sure each layer type has a unique id since each type is also unique. This removes the need to manage those identifiers ourselves.

Listing 3.5: Application/Interface. h

```
/* ... same as before ... */

template<typename Layer, typename... Args>
EMPTY_INLINE Layer* AttachLayer(Args&&... args)
{
    // check layer's type compliance
    EMPTY_STATIC_ASSERT(std::is_base_of<AppInterface,
Layer>::value);

    // check if layer allready exist
    if(GetLayer<Layer>() != nullptr)
    {
        EMPTY_ERROR("Layer allready attached!");
        return nullptr;
    }

    // create layer and add to collection
    auto layer = new Layer(std::forward<Args>(args)...);
    m_Context->Layers.push_back(layer);
    layer->m_LayerID = TypeID<Layer>();
```

```
layer->m_Context = m_Context;  
layer->OnStart();  
return layer;  
}  
  
//... same as before
```

An essential aspect to note in the layer attachment function is that we are providing the context pointer to the layer. This ensures that the layer points to the appropriate app context, accurately reflecting the necessary behaviors. You should also note that we are leveraging a variable template to make it possible to attach a layer that has a constructor with multiple arguments.

3.1.4 Detaching Layers

The natural progression would involve implementing a function to detach a layer from the application. However, it's not as straightforward for a critical reason: imagine you're currently executing the update function of a layer, and within that process, you attempt to detach the same layer. This action would inevitably lead to a crash since you would be deleting memory that is actively in use. To address this constraint, we need to introduce an event handler to our application. This entity can oversee the completion

of the layer's update and handle its destruction after the current frame or loop call.

Event Dispatcher

An event dispatcher is an entity responsible for managing and distributing events or messages within a software system. Receive events from various sources and dispatch them to the appropriate handlers or listeners. Essentially, it acts as a central hub, facilitating communication and coordination between different parts of an application by routing and delivering events to their designated recipients.

While there are existing event dispatchers available, such as Boost.Signals2, JUCE's Listener and Broadcaster, etc., we've opted to develop our custom event dispatcher for several reasons. One significant advantage is the increased control we gain over its behavior, allowing us to fine-tune its functionality and seamlessly integrate it into the engine.

Create the file shown in (Listing 3.6) in your "Common" directory:

Listing 3.6: Common/Event.h

```
#pragma once
#include "Core.h"

namespace EmPy
{
    template <typename Event>
    struct EventListener
    {
        using CallbackFn = std::function<void(const Event&)>;
        EMPY_INLINE EventListener(CallbackFn&& callback,
        uint32_t listnrid) :
            Callback(std::move(callback)), ID(listnrid)
        { }

        CallbackFn Callback;
        uint32_t ID;
    };

    // Event Registry

    template <typename Event>
    struct EventRegistry
    {
        using Listener =
        std::unique_ptr<EventListener<Event>>;
        std::queue<std::unique_ptr<Event>> Queue;
        std::vector<Listener> Listeners;
    };

    // Event Dispatcher

    struct EventDispatcher
    {
        EMPY_INLINE ~EventDispatcher()
    };
}
```

```

    {

        for(auto& [_, ptr] : m_Registry)
        {
            auto registry = CastRegistry<char>(ptr);
            EMPLY_DELETE(registry);
        }
    }

    template <typename Event, typename Callback>
    EMPLY_INLINE void AttachCallback(Callback&& callback,
    uint32_t listnrid)
    {
        auto listener =
        std::make_unique<EventListener<Event>>(std::move(callback),
        listnrid);
        GetRegistry<Event>()-
    >Listeners.push_back(std::move(listener));
    }

    template <typename Event>
    EMPLY_INLINE void DetachCallback(uint32_t listnrid)
    {
        auto& listeners = GetRegistry<Event>()-
    >Listeners;
        listeners.erase(std::remove_if(listeners.begin(),
        listeners.end(), [&] (auto& listener)
        {
            return (listener->ID == listnrid);
        }), listeners.end());
    }

    EMPLY_INLINE void EraseListener(uint32_t listnrid)
    {
        for(auto& [_, registry] : m_Registry)
        {

```

```

        auto& listeners = CastRegistry<int8_t>
(registry)->Listeners;

        listeners.erase(std::remove_if(listeners.begin(
),
        listeners.end(), [&] (auto& listener)
{
        return (listener->ID == listnrid);
}),
        listeners.end());
}

}

template <typename Event, typename... Args>
EMPTY_INLINE void PostEvent(Args&&...args)
{
    auto registry = GetRegistry<Event>();
    if(registry->Listeners.empty()) { return; }
    registry->Queue.push(std::make_unique<Event>
(std::forward<Args>(args)...));
}

template <typename Task>
EMPTY_INLINE void PostTask(Task&& task)
{
    m_Tasks.push(std::move(task));
}

EMPTY_INLINE void PollEvents()
{
    // persistent callbacks
    for(auto& [_, pointer] : m_Registry)
    {
        auto registry = CastRegistry<char>(pointer);

        while(!registry->Queue.empty())

```

```

    {
        for(auto& listener : registry->Listeners)
        {
            listener->Callback(*registry-
>Queue.front());
        }
        registry->Queue.pop();
    }

    // frame callbacks
    while(!m_Tasks.empty())
    {
        m_Tasks.front();
        m_Tasks.pop();
    }
}

private:
    template <typename Event>
    EMPTY_INLINE EventRegistry<Event>* CastRegistry(void*
p)
    {
        return static_cast<EventRegistry<Event>*>(p);
    }

    template <typename Event>
    EMPTY_INLINE EventRegistry<Event>* GetRegistry()
    {
        auto it = m_Registry.find(TypeID<Event>());
        if(it != m_Registry.end())
        {
            return CastRegistry<Event>(it->second);
        }
        auto registry = new EventRegistry<Event>();
        m_Registry[TypeID<Event>()] = registry;
    }
}

```

```

        return registry;
    }

private:
    std::unordered_map<uint32_t, void*> m_Registry;
    std::queue<std::function<void()>> m_Tasks;
};

}

```

This code defines an event system comprising three key components: `EventListener`, `EventRegistry`, and `EventDispatcher`. `EventListener` represents a listener for specific events, storing a callback function and an the listener ID. `EventRegistry` manages listeners and a queue for a particular type of event. The core, `EventDispatcher`, handles event attachment, detachment, posting events, and executing tasks. It uses a map to store registries for different types of events and a queue for tasks. Functions within the context or dispatcher, such as `AttachCallback()`, `DetachCallback()`, `PostEvent()`, `PostTask()`, and `PollEvents()`, facilitate event management, enabling attaching callbacks, posting events, and executing tasks. It ensures event execution by iterating through registered listeners and dispatching queued events to respective listeners while also handling queued tasks.

This structure provides flexibility and control over event handling within the application. It is crucial to distinguish between a task and an event in this

context. Unlike an event, a task is not associated with an event object; instead, it is simply a function passed to the dispatcher for execution without arguments. The `PostTask()` function serves this purpose perfectly and is precisely what we require to detach a layer from the application. With our event dispatcher now implemented, let us integrate it into our application context. This integration will enable its utilization within the interface, consequently making it accessible for layers and the entire application. Update the context by adding the dispatcher as shown in (Listing 3.7).

Listing 3.7: Application/Context. h

```
#pragma once
#include "Common/Event.h"

namespace EmPy
{
    // forward declaration
    struct AppInterface;

    // application context
    struct ApplicationContext
    {
        EMPY_INLINE ~ApplicationContext()
        {
            for (auto& layer : Layers)
            {
                EMPY_DELETE(layer);
            }
        }
    };
}
```

```

        }

    }

    std::vector<AppInterface*> Layers;
    EventDispatcher Dispatcher;
};

}

```

Let us implement the function to detach a layer from the application.

Listing 3.8: Application/Interface.h

```

/* ... same as before ... */

template<typename Layer>
EMPY_INLINE void DetachLayer()
{
    EMPY_STATIC_ASSERT(std::is_base_of<AppInterface,
Layer>::value);
    m_Context->Dispatcher.PostTask([this]
    {
        m_Context->Layers.erase(std::remove_if(m_Context-
>Layers.begin(),
        m_Context->Layers.end(), [this] (auto& layer)
        {
            if(layer->m_LayerID ==TypeID<Layer>())
            {
                m_Context->Dispatcher.EraseListener(layer-
>m_LayerID);
                EMPY_DELETE(layer);
                return true;
            }
            return false;
        })
    });
}

```

```
        } ,
        m_Context->Layers.end());
    });
}

/* ... same as before ... */
```

The function `DetachLayer()` is designed to detach a layer from the application. It starts with a static assertion, ensuring that the provided "Layer" type is derived from `AppInterface`. The function uses the event dispatcher to post a task. This task involves erasing the layer from the layer array. We employ `std::remove_if()` to find the specific layer based on its type identifier. When the layer of the specified type is found, we remove it from the layer vector. Additionally, we call the `EraseListener()` function from the dispatcher to clean up the listener associated with this layer type. Finally, it deletes the layer object and ensures its complete removal from the layer vector by using the `erase()` function. This delayed execution within a posted task ensures safe detachment of the layer without interfering with potential ongoing processes involving that layer.

3.1.5 Creating First Layer

Let us apply this to the `Empy.exe` application. First, make sure to update the `Empy.h` header file, as shown

in the following code snippet:

Listing 3.9: includes/Empty.h

```
#pragma once
#include "Application/Application.h"
```

Next, update your Application.h header file by adding the code to update all the attached layers as shown in (Listing 3.10).

Listing 3.10: Application/Application.h

```
#pragma once
#include "Interface.h"

/* ... same as before ... */

EMPY_INLINE void RunContext()
{
    while(true)
    {
        m_Context->Dispatcher.PollEvents(); // <- handle
events

        for(auto layer : m_Context->Layers) // <- update
layers
        {
            layer->OnUpdate();
        }
    }
}
```

```
/* ... same as before ... */
```

Finally, create a layer in your project `EmPy.exe` and attach it to your newly created application as shown in (Listing [3.11](#)).

Listing 3.11: EmPyEditor/src/Editor.cpp

```
#include <EmPy.h>
using namespace EmPy;

struct MyLayer : AppInterface
{
    EMPY_INLINE void OnUpdate()
    {
        EMPY_INFO("OnUpdate()");
    }

    EMPY_INLINE void OnStart()
    {
        EMPY_TRACE("OnStart()");
    }
};

int32_t main(int32_t argc, char** argv)
{
    auto app = new Application();
    app->AttachLayer<MyLayer>();
    app->RunContext();
    return 0;
}
```

When you compile the project and run `Empy.exe`, you will observe the layer initializing on start and updating consistently frame by frame. An essential advantage of this approach lies in its flexibility: you can easily add, remove, or access layers within each other. This seamless interaction is possible due to their shared inheritance from a common interface and their access to the same application context. Moreover, this setup empowers layers to detach themselves, providing self-contained functionality within the system.

Make sure to also test if retrieving and detaching a layer also works. Take some time to digest, if necessary, but I believe this should be straightforward.

3.1.6 Dispatcher Interface

Another aspect we aim to achieve is allowing our application and its layers to post events. Additionally, they should have the ability to attach callbacks to listen for specific events when triggered. To accomplish this, we need to implement specific methods in the application interface. Add the methods defined in (Listing 3.12) to your application interface.

Listing 3.12: Application/Interface.
h

```
/* ... same as before ... */

// attach event callback
template <typename Event, typename Callback>
EMPY_INLINE void AttachCallback(Callback&& callback)
{
    m_Context->Dispatcher.AttachCallback<Event>
(std::move(callback), m_LayerID);
}

// post event
template <typename Event, typename... Args>
EMPY_INLINE void PostEvent(Args&&...args)
{
    m_Context->Dispatcher.PostEvent<Event>(std::forward<Args>
(args)...);
}

// post task event
template <typename Task>
EMPY_INLINE void PostTask(Task&& task)
{
    m_Context->Dispatcher.PostTask(std::move(task));
}

// detach callback
template <typename Event>
EMPY_INLINE void DetachCallback()
{
    m_Context->Dispatcher.DetachCallback<Event>(m_LayerID);
}

// ... same as before
```

These methods serve as wrappers for the dispatcher functions. There's no need for concern regarding potential performance loss because the compiler optimizes these calls, utilizing the `EMPTY_INLINE` keyword to condense them into single calls, especially when optimization is enabled, such as in a release build. These functions will be critical for the editor's user interface as they will bridge interactions between the engine internals and GUI-elements.

You can test these functions in `Empy.exe` as shown in (Listing 3.13)

Listing 3.13: EmPyEditor/src/Editor.cp
p

```
#include <Empy.h>
using namespace Empy;

struct TestEvent
{
    int32_t Data = 0;
};

struct MyLayer : AppInterface
{
    EMPTY_INLINE void OnStart()
    {
        AttachCallback<TestEvent>([this] (auto event)
        {
            EMPTY_TRACE("TestEvent: {}", event.Data);
            DetachLayer<MyLayer>();
        });
    }
};
```

```
}

EMPTY_INLINE void OnUpdate()
{
    EMPTY_TRACE("OnUpdate()");

    if (++m_Counter == 10)
    {
        PostEvent<TestEvent>();
    }
}

private:
    uint8_t m_Counter = 0;
};

/* ... same as before ... */
```

This code defines a `TestEvent` structure containing an integer data member. Within the layer's `OnStart()` method, we attach a callback for the `TestEvent`, logging its data, and subsequently detaching the layer itself. In the `OnUpdate()` method, we increment the counter member variable. When the counter reaches 10, the layer posts a `TestEvent`.

Let your imagination run wild and explore the endless possibilities with this feature! I can almost see a light bulb flashing above your head. Next, we will implement the window and handle its inputs and events using the dispatcher we just created.

3.2 Window Event Handling

We've opted to leverage the Graphics Library Framework (GLFW) to craft a cross-platform window for our application. This choice ensures compatibility across various operating systems and lays the foundation for rendering our engine's output.

3.2.1 Linking GLFW

GLFW is a lightweight utility library designed to complement OpenGL. Developed and maintained by GLFW Development Team, it facilitates window creation, OpenGL context management, and input handling for keyboards, mouse, and joysticks. Being open-source under the zlib/libpng license, it grants easy access to its source code for customization. Alongside supporting multiple video modes and monitors, GLFW proves immensely valuable for OpenGL application development without slowing the system.

Listing 3.14: Root/conanfile.txt

```
[requires]
spdlog/1.12.0
glfw/3.3.8

[generators]
```

```
cmake  
  
[options]  
glfw:shared=False
```

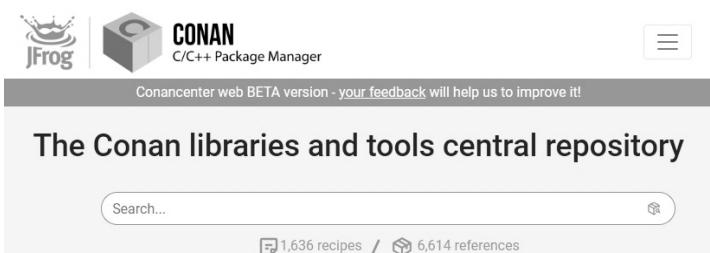


Figure 3.3: Conan Central Repository

Update your "**conanfile.txt**" as shown in (Listing 3.14). This enables our project to link GLFW as a static library. We could also link dynamically just by setting the shared option to "True". At the time of this writing, the latest version of GLFW is "3.3.8". Browse to the conan center using this link <https://conan.io/center/>. Once there, you can search for "glfw" using the input field. See (Figure 3.3)

Extend your project's directory tree as shown in (Figure 3.4).

```
EmptyEngine
widthheight
widthheight
widthheight
widthheight
widthheight
```

```
includes
└── Window
    ├── Events.h
    ├── Inputs.h
    └── Window.h
```

Figure 3.4: Window Directory

3.2.2 Window Inputs

Include the following code snippet (Listing 3.15) in your **"Input.h"** header file. It simply defines a structure that stores the window's input information. We have data for the keyboard and the mouse. You will also notice the inclusion of GLFW.

Listing 3.15: Window/Events.h

```
#pragma once
#include "Common/Event.h" // <-- this inclusion order is
important for later (opengl)!

#include <GLFW/glfw3.h>

namespace Empty
{
    struct WindowInputs
    {
        std::bitset<GLFW_MOUSE_BUTTON_LAST> Mouse;
        std::bitset<GLFW_KEY_LAST> Keys;
    };
}
```

```
    double MouseX = 0.0;
    double MouseY = 0.0;
};

}
```

3.2.3 Window Events

Proceed by adding the following code (Listing 3.16) in the `Events.h` header file. This code defines custom window's event types tailored for mouse, keyboard, etc. These events will play a crucial role in transmitting the data of triggered events from GLFW to our application and its subsequent layers using the event dispatcher created previously.

Listing 3.16: Window/Events. h

```
#pragma once
#include "Common/Event.h"

namespace Empy
{
    // Window Events

    struct WindowMaximizeEvent {};

    struct WindowIconifyEvent {};

    struct WindowRestoreEvent {};

    struct WindowCloseEvent {};
}
```

```
struct WindowResizeEvent
{
    WindowResizeEvent(int32_t w, int32_t h):
        Width(w), Height(h) {}
    int32_t Height = 0;
    int32_t Width = 0;
};

// Key Events

struct KeyReleaseEvent
{
    KeyReleaseEvent(int32_t key): Key(key) {}
    int32_t Key = -1;
};

struct KeyPressEvent
{
    KeyPressEvent(int32_t key): Key(key) {}
    int32_t Key = -1;
};

struct KeyRepeatEvent
{
    KeyRepeatEvent(int32_t key): Key(key) {}
    int32_t Key = -1;
};

// Mouse Events

struct MouseReleaseEvent
{
    MouseReleaseEvent(int32_t b): Button(b) {}
    int32_t Button = -1;
};
```

```

struct MouseDownEvent
{
    MouseDownEvent(int32_t b) : Button(b) { }
    int32_t Button = -1;
};

struct MouseDragEvent
{
    MouseDragEvent(double dx, double dy) :
        DeltaX(dx), DeltaY(dy) { }
    double DeltaX = 0, DeltaY = 0;
};

struct MouseMotionEvent
{
    MouseMotionEvent(double x, double y) :
        TargetX(x), TargetY(y) { }
    double TargetX = 0, TargetY = 0;
};

struct MouseWheelEvent
{
    MouseWheelEvent(double sx, double sy) :
        ScrollX(sx), ScrollY(sy) { }
    double ScrollX = 0, ScrollY = 0;
};
}

```

3.2.4 Window Abstraction

To harness the capabilities of our custom event dispatcher in handling all window events, we must

craft a window abstraction. This entails developing a custom window type modeled after GLFW's.

This specialized window should adeptly capture its events and efficiently broadcast them using the event dispatcher. Open the "**Window.h**" header file and add a window class as showned in ([Listing 3.17](#)).

Listing 3.17: Window/Window.h

```
#pragma once
#include "Events.h"

namespace EmPy
{
    struct AppWindow
    {
        EMPY_INLINE AppWindow(EventDispatcher* dispatcher,
int32_t width, int32_t height, const char* title)
        {
            // coming next
        }

        EMPY_INLINE ~AppWindow()
        {
            glfwDestroyWindow(m_Handle);
            glfwTerminate();
        }

        EMPY_INLINE GLFWwindow* Handle()
        {
            return m_Handle;
        }
}
```

```

EMPTY_INLINE bool PollEvents()
{
    glfwPollEvents();
    m_Dispatcher->PollEvents();
    glfwSwapBuffers(m_Handle);
    return (!glfwWindowShouldClose(m_Handle));
}

EMPTY_INLINE bool IsKey(int32_t key)
{
    if(key >= 0 && key <= GLFW_KEY_LAST)
        return m_Inputs.Keys.test(key);
    return false;
}

EMPTY_INLINE bool IsMouse(int32_t button)
{
    if(button >= 0 && button <=
GLFW_MOUSE_BUTTON_LAST)
        return m_Inputs.Mouse.test(button);
    return false;
}

private:
    EventDispatcher* m_Dispatcher;
    WindowInputs m_Inputs;
    GLFWwindow* m_Handle;
};

}

```

The code above in ([Listing 3.17](#)) defines the `AppWindow` structure. The `PollEvents()` method swaps the window buffers, polls for any pending events, and

returns a boolean indicating whether the window should remain open or close based on the `glfwWindowShouldClose()` function. We also have some member functions to test the state of a key or a mouse button. We are yet to implement the logic to set their value when they are pushed or pressed.

☞ **Buffer Swapping** : in GLFW, buffer swapping is the process of displaying the content of the back buffer on the screen. In double-buffered rendering (common in modern graphics programming), there are two buffers: the front buffer (what the user sees) and the back buffer (where rendering occurs). When rendering is complete, the back buffer needs to be swapped with the front buffer to avoid displaying incomplete or partially rendered images. GLFW's `glfwSwapBuffers()` function performs this swap, making the updated content of the back buffer visible on the screen.

GLFW Callbacks

To effectively engage with GLFW, we are required to create custom callbacks functions that GLFW can invoke upon event triggers. Given that GLFW is written in C and doesn't support class or structure member functions as callbacks, we are then require to implement static member functions within the

`AppWindow`. These static functions must conform to specific argument list mandated by the GLFW's API for them to qualify as valid callback implementations.

A challenge arising from implementing static member functions in `AppWindow` is the inability to access the non-static member variables. However, GLFW offers a solution by allowing us to attach custom user data to any window. These user data can be accessed within the callback functions, as each triggered event includes the associated window.

The subsequent static function implementation (Listing 3.18) accepts a GLFW window as an argument and retrieves its user data pointer, which is subsequently cast into an `AppWindow` object. How this user data pointer can be established will be addressed shortly upon the implementation of the constructor. It's noteworthy that we've chosen to utilize our `AppWindow` type itself as the GLFW's user data. Add the function to your `AppWindow` class as a private member function.

Listing 3.18: `Window/Window.h`

```
/* ... same as before ... */

EMPY_INLINE static AppWindow* GetUserData(GLFWwindow* window)
{
```

```

        return static_cast<AppWindow*>
(glfwGetWindowUserPointer(window)) ;
}

/* ... same as before ... */

```

With the ability to retrieve the window user data pointer in place, we are now set to implement our callback functions.

☞ **Key Callback Function** : This function will be called by GLFW every time a key is pressed, released, or repeated. The code snippet provided below ([Listing 3.19](#)) defines the `OnKey()` static function, primarily handling key events triggered by GLFW.

Listing 3.19: Window/Window.h

```

/* ... same as before ... */

EMPY_INLINE static void OnKey(GLFWwindow* window, int32_t
key, int32_t, int32_t action, int32_t)
{
    Window* self = GetUserData(window);

    if(key >= 0 && key <= GLFW_KEY_LAST)
    {
        switch (action)
        {
            case GLFW_RELEASE:
                self->m_Dispatcher->PostEvent<KeyReleaseEvent>

```

```

(key) ;
    self->m_Inputs.Keys.reset(key) ;
break;

case GLFW_PRESS:
    self->m_Dispatcher->PostEvent<KeyPressEvent>
(key) ;
    self->m_Inputs.Keys.set(key) ;
break;

case GLFW_REPEAT:
    self->m_Dispatcher->PostEvent<KeyRepeatEvent>
(key) ;
    self->m_Inputs.Keys.set(key) ;
break;
}

return;
}

EMPTY_ERROR("invalid key code detected: [{}]", key);
}

/* ... same as before ... */

```

Upon receiving a key event within a GLFW window, this function retrieves the associated user data pointer through the `GetUserData(window)` function. It then proceeds to process the event. If the key falls within a valid range (`0` to `GLFW_KEY_LAST`), it distinguishes between `GLFW_RELEASE`, `GLFW_PRESS`, and `GLFW_REPEAT` actions. For each action, it leverages the event dispatcher within the `Window` to post the corresponding `KeyReleaseEvent`, `KeyPressEvent`, or `KeyRepeatEvent`.

☞ **More Callback Functions** : The same thing can be done for other event types, as shown in (Listing 3.20).

Listing 3.20: Window/Window.h

```
/* ... same as before ... */

EMPTY_INLINE static void OnMouse(GLFWwindow* window, int32_t
button, int32_t action, int32_t)
{
    Window* self = GetUserData(window);

    if(button >= 0 && button <= GLFW_MOUSE_BUTTON_LAST)
    {
        switch (action)
        {
            case GLFW_PRESS:
                self->m_Dispatcher->PostEvent<MouseDownEvent>
(button);
                self->m_Inputs.Mouse.set(button);
                break;

            case GLFW_RELEASE:
                self->m_Dispatcher-
>PostEvent<MouseReleaseEvent>(button);
                self->m_Inputs.Mouse.reset(button);
                break;
        }
        return;
    }
    EMPTY_WARN("Invalid key code detected: [{}]", button);
}
```

```
EMPTY_INLINE static void OnResize(GLFWwindow* window, int32_t width, int32_t height)
{
    GetUserData(window)->m_Dispatcher->PostEvent<WindowResizeEvent>(width, height);
}

EMPTY_INLINE static void OnMotion(GLFWwindow* window, double x, double y)
{
    Window* self = GetUserData(window);
    self->m_Dispatcher->PostEvent<MouseMotionEvent>(x, y);

    if (self->m_Inputs.Mouse.test(GLFW_MOUSE_BUTTON_LEFT))
    {
        self->m_Dispatcher->PostEvent<MouseDragEvent>(
            (self->m_Inputs.MouseX - x),
            (self->m_Inputs.MouseY - y)
        );
    }

    self->m_Inputs.MouseX = x;
    self->m_Inputs.MouseY = y;
}

EMPTY_INLINE static void OnWheel(GLFWwindow* window, double x, double y)
{
    GetUserData(window)->m_Dispatcher->PostEvent<MouseWheelEvent>(x, y);
}

EMPTY_INLINE static void OnMaximize(GLFWwindow* window, int32_t action)
{
    if (action)
```

```
{  
    GetUserData(window)->m_Dispatcher-  
>PostEvent<WindowMaximizeEvent>();  
}  
  
else  
{  
    GetUserData(window)->m_Dispatcher-  
>PostEvent<WindowRestoreEvent>();  
}  
}  
  
EMPTY_INLINE static void OnIconify(GLFWwindow* window, int32_t  
action)  
{  
    if (action)  
    {  
        GetUserData(window)->m_Dispatcher-  
>PostEvent<WindowIconifyEvent>();  
    }  
  
else  
{  
    GetUserData(window)->m_Dispatcher-  
>PostEvent<WindowRestoreEvent>();  
}  
}  
  
EMPTY_INLINE static void OnError(int32_t code, const char*  
msg)  
{  
    EMPTY_ERROR("[GLFW]: [{}][{}]", code, msg);  
}  
  
EMPTY_INLINE static void OnClose(GLFWwindow* window)  
{  
    GetUserData(window)->m_Dispatcher-  
>PostEvent<WindowCloseEvent>();  
}
```

```
}

/* ... same as before ... */
```

Window Constructor

Creating a GLFW window is quite simple. You can see how this is done in the code snippet provided in (Listing 3.21).

Listing 3.21: Window/Window.h

```
/* ... same as before ... */

EMPTY_INLINE AppWindow(EventDispatcher* dispatcher, int32_t
width, int32_t height, const char* title):
m_Dispatcher(dispatcher)
{
    if(glfwInit() != GLFW_TRUE)
    {
        EMPTY_FATAL("glfwInit() failed!");
        exit(EXIT_FAILURE);
    }

    glfwWindowHint(GLFW_OPENGL_CORE_PROFILE, GLFW_TRUE);
    glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
    glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);

    auto display = glfwGetVideoMode(glfwGetPrimaryMonitor());
    glfwWindowHint(GLFW_REFRESH_RATE, display->refreshRate);
    glfwWindowHint(GLFW_GREEN_BITS, display->greenBits);
    glfwWindowHint(GLFW_BLUE_BITS, display->blueBits);
    glfwWindowHint(GLFW_RED_BITS, display->redBits);
```

```
glfwWindowHint(GLFW_SAMPLES, 4);

glfwWindowHint(GLFW_MAXIMIZED, GLFW_FALSE);
glfwWindowHint(GLFW_RESIZABLE, GLFW_TRUE);

m_Handle = glfwCreateWindow(width, height, title, NULL,
NULL);
if(m_Handle == NULL)
{
    EMPLY_FATAL("failed to initialize app-window!");
    exit(EXIT_FAILURE);
}

// set user data
glfwSetWindowUserPointer(m_Handle, this);

// binding callbacks
glfwSetWindowMaximizeCallback(m_Handle, OnMaximize);
glfwSetFramebufferSizeCallback(m_Handle, OnResize);
glfwSetWindowIconifyCallback(m_Handle, OnIconify);
glfwSetMouseButtonCallback(m_Handle, OnMouse);
glfwSetWindowCloseCallback(m_Handle, OnClose);
glfwSetCursorPosCallback(m_Handle, OnMotion);
glfwSetScrollCallback(m_Handle, OnWheel);
glfwSetKeyCallback(m_Handle, OnKey);
glfwSetErrorCallback(OnError);

// create opengl context
glfwMakeContextCurrent(m_Handle);

// buffer swap interval
glfwSwapInterval(1);
}

/* ... same as before ... */
```

The constructor starts by initializing GLFW. If the initialization fails, it logs a fatal error and exits the program. Then it sets various hints for the GLFW window, specifying the OpenGL version, context profile, refresh rate, color bits, and other display-related properties. Using the function

`glfwCreateWindow()`, it creates a window with specific dimensions and a title. If window creation fails, it logs an error and exits the program. The function

`glfwSetWindowUserPointer()` associates the `Window` object itself with the "GLFWwindow". This association aids in retrieving the `Window` object itself from within our callback functions. It registers all callbacks implemented earlier to handle different types of window events triggered by GLFW. It then establishes the OpenGL context for the window using

`glfwMakeContextCurrent()`.

3.2.5 Showing the Window

Let's integrate the window in the application context.

Listing 3.22: Application/Context.
`h`

```
#pragma once
#include "Window/Window.h"

namespace EmPy
{
```

```

// forward declaration
struct AppInterface;

// application context
struct ApplicationContext
{
    EMPY_INLINE ApplicationContext()
    {
        Window = std::make_unique<AppWindow>(&Dispatcher,
1280, 720, "Empty Engine");
    }

    EMPY_INLINE ~ApplicationContext()
    {
        for(auto& layer : Layers)
        {
            EMPY_DELETE(layer);
        }
    }

    std::vector<AppInterface*> Layers;
    std::unique_ptr<AppWindow> Window;
    EventDispatcher Dispatcher;
};

}

```

Update your application's `RunContext()` function, as shown below.

Listing 3.23: Application/Application.h

```

EMPY_INLINE void RunContext()
{
    while (m_Context->Window->PollEvents())

```

```

    {
        for(auto layer : m_Context->Layers)
        {
            layer->OnUpdate();
        }
    }
}

```

Finally, implement some callbacks in **Editor.cpp** to test if the window events are triggered correctly. See ([Listing 3.24](#)).

Listing 3.24: EmPyEditor/src/Editor.cpp

```

#include <EmPy.h>
using namespace EmPy;

struct MyLayer : AppInterface
{
    EMPY_INLINE void OnStart()
    {
        AttachCallback<MouseEvent>([this] (auto e)
        {
            EMPY_TRACE("Mouse x:{} y:{}",
e.TargetX,
e.TargetY);
        });
    }
};

int32_t main(int32_t argc, char** argv)
{
    auto app = new Application();

    app->AttachCallback<KeyPressEvent>([] (auto e) {

```

```
    EMPLY_TRACE("Key: {}", e.Key);  
});  
  
app->AttachLayer<MyLayer>();  
app->RunContext();  
return 0;  
}
```



Figure 3.5: Application Window (Windows)



Figure 3.6: Application Window (Linux)

Code not working? clone the branch "window-events" on the GitHub repository and grant execution rights to the "EmpyLinux.sh" script on Linux. <https://github.com/Madsycode/book-empy-engine>.

```
git clone -branch <branch-name> <repository-link>
```

Part II

Rendering Graphics

4 Introduction to OpenGL

You shall eat the fruit of the labor of your hands.
(King David)

OpenGL, or Open Graphics Library, stands as a powerful cross-platform API used to render 2D and 3D graphics. Originally developed by Silicon Graphics Inc. (SGI) in 1992, OpenGL has since evolved into an industry standard, offering a versatile and efficient means to interact with graphics hardware.

Designed to be platform-independent, OpenGL functions across various operating systems, making it widely accessible for developers creating applications, games, simulations, and graphical interfaces. Its core principles revolve around providing a set of commands that allow developers to describe the objects and operations needed to produce high-quality graphical output.

The architecture of OpenGL is structured around a state machine model, where developers manipulate various states to define the rendering pipeline. This pipeline processes geometric and image data to generate the final visual output on the screen. By utilizing programmable shaders, developers can implement custom algorithms for vertex and fragment

processing, enabling a high degree of flexibility and creativity in rendering scenes.

The flexibility and extensibility of OpenGL have led to its continuous adaptation and integration with modern graphics technologies. Its capabilities have expanded to support advanced features like tessellation, geometry shaders, and compute shaders, empowering developers to create visually stunning and computationally intensive graphics.

4.1 OpenGL Versions

OpenGL has undergone significant evolution through its various versions, each representing a milestone in advancing graphical capabilities. The initial iteration, OpenGL 1.x, laid the groundwork for rendering 2D and early 3D graphics. The advent of OpenGL 2.x marked a turning point with the introduction of programmable shaders, enabling developers to create more intricate and realistic visuals. Subsequently, OpenGL 3.x brought about a fundamental shift in architecture, emphasizing the deprecation of older features in favor of a more streamlined and efficient system. Within this series, OpenGL 3.3 stands as a pivotal version, striking a balance between modern advancements and broader compatibility. It maintains relevance by offering enhanced capabilities while ensuring compatibility with a range of hardware, making it a popular choice for

developers aiming to leverage newer features while catering to diverse systems.

OpenGL 4.x represents a significant leap forward in graphics capabilities compared to its predecessors. Building upon the foundation laid by OpenGL 3.x, version 4.x introduced several advanced features and enhancements. It focused on pushing the boundaries of graphical realism and performance, offering technologies such as tessellation, which allows for smoother curved surfaces by dynamically subdividing geometry. Additionally, OpenGL 4.x expanded support for compute shaders, enabling more general-purpose parallel computations on the GPU beyond traditional graphics tasks. This version also introduced features like shader storage buffer objects and improved texture handling, further empowering developers to create more immersive and visually stunning applications. OpenGL 4.x stands as a testament to the continuous evolution of the API, providing cutting-edge tools and capabilities for pushing the boundaries of graphical rendering in modern applications.

4.2 Why Version 3.3?

Using OpenGL 3.3 as the chosen version in this book is a practical decision. It strikes a balance between leveraging modern rendering capabilities and ensuring compatibility across a broad range of systems, making

it accessible and relevant for a larger audience. With this version, you can learn the essentials of modern OpenGL programming while having a good foundation for understanding newer versions or adapting to different environments.

You can expect to explore programmable shaders, modern rendering techniques, and an updated approach to graphics programming, all within a framework that maintains compatibility with a reasonable spectrum of hardware. This ensures that the knowledge gained from the book remains valuable and applicable in various scenarios.

4.3 Rendering Pipeline

The OpenGL rendering pipeline is a sequence of stages through which 3D graphics data is processed to generate a 2D image on the screen. It's composed of several stages, each responsible for specific transformations and operations on the input data.

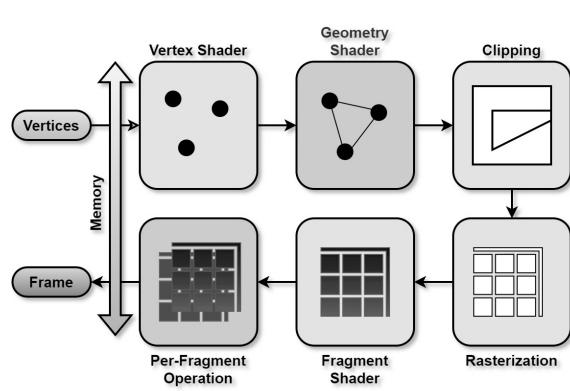


Figure 4.1: OpenGL Rendering Pipeline

(Figure [4.1](#)) shows an overview of the OpenGL rendering pipeline. It is important to note that this representation is not 100% accurate, as many underlying steps are not represented because of their complexity. The steps with gray boxes are optional.

Let us look into each step in a little more detail.

4.3.1 Vertex Specification

A vertex represents a point in 3D space and serves as the basic building block for creating shapes and objects in OpenGL. It typically consists of attributes such as position, color, texture coordinates, and normal. For example, a vertex might contain information about its position (x, y, z), color (r, g, b), and texture coordinates (u, v). Multiple vertices combined create geometry (mesh), defining the structure of objects rendered on the screen.

The rendering process begins with the provision of vertex data, defining the fundamental building blocks of shapes in 3D space. These vertices contain information like positions, colors, and texture

coordinates, specifying the geometry of objects in their local or object space.

```
simple vertex with position and color attributes
struct Vertex
{
    float x, y, z;      // position
    float r, g, b;      // color
};

// defining simple mesh vertices
std::vector<Vertex> vertices = { /* ... */ };

// generating and binding a vertex buffer
unsigned int VBO; // buffer id
glGenBuffers(1, &VBO); // generate buffer
glBindBuffer(GL_ARRAY_BUFFER, VBO); // bind buffer
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices,
GL_STATIC_DRAW); // transfer data to GPU
```

4.3.2 Vertex Shader

Shaders are small programs executed on the GPU that manipulate vertices and pixels during the rendering process. They are written in languages like OpenGL Shading Language (GLSL) and enable sophisticated rendering techniques by controlling how objects appear, move, and interact with light.

```
An example of a simple GLS vertex shader
#version 330 core

// vertex attribute
layout (location = 0) in vec3 a_position;
```

```

void main()
{
    // set vertex position output
    gl_Position = vec4(a_position, 1.0);
}

```

Once the vertex data is provided, it undergoes transformation via the vertex shader. This shader manipulates each vertex individually, applying operations such as translations, rotations, and scaling. Additionally, it projects these vertices from the object space into the clip space using projection matrices, which ultimately leads to the representation of Normalized Device Coordinates (NDC).

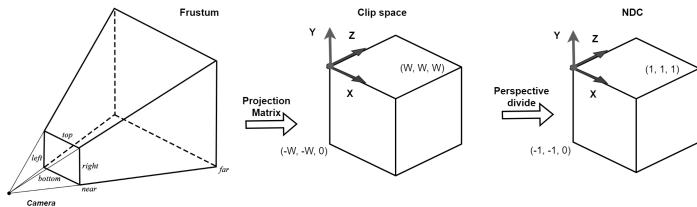


Figure 4.2: Normalized Device Coordinates

NDCs are a crucial part of the rendering pipeline. After projection, the vertices are in NDC, a normalized space where all visible coordinates fit within a cube ranging from -1 to 1 along each axis. See (Figure 4.2). This

space simplifies calculations and ensures consistency across different viewports and aspect ratios.

The transformed vertices are assembled into primitives such as points, lines, or triangles based on the specified rendering mode. These primitives are the basic elements used to construct the visible geometry within the scene.

4.3.3 Geometry Shader

In some cases, a geometry shader might be utilized, allowing further processing at the primitive level. This shader offers powerful capabilities for manipulating geometry data. Unlike the vertex and fragment shaders, which respectively handle individual vertices and fragments, the geometry shader operates on entire primitives.

One primary use of the geometry shader is to transform the incoming primitives. For instance, you can amplify or reduce the number of primitives by emitting multiple output primitives for each input primitive. This process is called “geometry amplification.” For instance, take a single input triangle and output multiple triangles to create effects like tessellation or adaptive subdivision.

Another application involves creating particle systems or instances of objects. Using a single primitive (such as a point or quad), it can generate complex geometry by replicating the same object multiple times, each with variations in position, size, or orientation. This technique is particularly useful for rendering large quantities of objects efficiently.

Despite its versatility, this shader can impact performance due to its nature of working on entire primitives. Overusing it or emitting a significantly higher number of primitives than the input can lead to performance bottlenecks. Thus, it is crucial to use it judiciously and optimize its usage for better performance.

4.3.4 Clipping

Primitives are then assessed against the view frustum to determine their visibility. Portions of primitives that fall outside the view frustum are clipped, ensuring only visible portions proceed through the pipeline.

The remaining primitives are then transformed from NDC to screen-space coordinates, accounting for the dimensions of the viewport. This transformation prepares them for rasterization, aligning them with the display dimensions.

4.3.5 Rasterization

During this stage, primitives are broken down into fragments, or pixels, based on their coverage of the screen. Each fragment represents a portion of the primitive visible on the screen and serves as the basis for further processing.

Attributes, such as colors, texture coordinates, normals, etc., associated with the vertices of the primitive are interpolated across the fragments generated. This interpolation ensures smooth transitions between vertices, giving a visually cohesive appearance to the rendered objects.

Once fragments are generated, they undergo depth testing. The depth values (usually derived from the Z-coordinate of vertices) of fragments are compared against the values stored in the depth buffer. Fragments that pass the depth test are considered visible and contribute to the final image.

Rasterization also involves handling aliasing issues that arise from displaying diagonal or curved lines on a grid of pixels. Techniques like multi-sampling or other anti-aliasing methods can be employed during rasterization to reduce jaggies or pixelation, resulting in smoother edges and improved visual quality.

4.3.6 Fragment Shader

These fragments undergo operations in the fragment shader. The fragment shader determines the final color, texture, depth, and other attributes of each fragment based on various calculations and inputs. This shader significantly influences the appearance of the rendered objects.

4.3.7 Per-Fragment Operations

Fragments go through additional tests and operations. Depth testing ensures that the depth values of the fragments are compared against the depth buffer to determine visibility, while blending combines fragment colors with the frame buffer based on alpha values and blending settings, allowing for transparent and complex visual effects.

4.3.8 Frame Buffer Output

Finally, the processed fragments are written to the frame buffer. A frame buffer in OpenGL is a collection of buffers that store pixel data for the rendered image. It includes color buffers for the final image, depth buffers for depth information, and sometimes stencil buffers for additional information. Frame buffers allow OpenGL to render a scene off-screen before displaying

it on the screen, enabling post-processing effects. The contents of the frame buffer are then displayed on the screen, presenting the culmination of the rendering pipeline as the visible output to the user.

Understanding these steps is crucial for developers to optimize performance, implement advanced rendering techniques, and create visually stunning graphics in OpenGL.

4.4 Future of OpenGL

OpenGL, while powerful, comes with limitations that have become more apparent in the context of modern graphics demands. Its higher-level abstraction introduces notable CPU overhead as a result of its stateful nature and reliance on driver management. This abstraction limits control over the graphics pipeline, hindering fine-tuning and optimization, especially in multi-threaded scenarios. The API's synchronization, often implicit between Central Processing Unit (CPU) and GPU, can lead to inefficiencies and stalls in resource usage. Moreover, its predominantly single-threaded design restricts the effective utilization of multi-core CPUs.

4.4.1 OpenGL vs. Vulkan

In contrast, Vulkan, as a more modern API, addresses these limitations by offering lower CPU overhead through a more explicit and streamlined approach. It provides granular control over the pipeline, empowering developers to optimize resource usage and rendering techniques. Vulkan's explicit synchronization and support for asynchronous operations offer improved performance by reducing stalls and allowing better utilization of modern hardware capabilities.

4.4.2 Important Note

Despite its limitations, OpenGL remains relevant for several reasons. First, its widespread adoption and legacy support across multiple platforms make it a go-to choice for developers working on diverse systems. Many existing applications and frameworks rely on OpenGL, and its familiarity within the development community continues to support its relevance.

Additionally, while newer APIs like Vulkan offer significant advances, OpenGL remains accessible for developers seeking a more straightforward, higher-level abstraction for graphics programming. Its ease of use, especially for beginners or those who do not require ultra-low-level control, makes it a viable option for various projects.

Moreover, OpenGL continues to serve as a foundation for learning computer graphics concepts. Its concepts and principles are transferable to newer APIs, enabling developers to grasp fundamental graphics programming concepts before diving into more complex and lower-level APIs like Vulkan or DirectX.

Lastly, OpenGL's stability and maturity, despite its limitations, ensure ongoing support and compatibility across a wide range of hardware and software environments. This reliability is crucial for applications where compatibility and broad reach are priorities.

5 Basic Rendering

Those who work their land will have abundant food, but those who chase fantasies have no sense. (King Solomon)

Rendering objects with OpenGL, as highlighted in the previous chapter, is not a straightforward task because of the initial setup complexities. Components such as vertex buffers, frame buffers, cameras, and shaders are imperative for this purpose. However, OpenGL's low-level nature provides only fundamental GPU interaction functions. Moreover, developing a renderer to oversee the intricate rendering process poses its own challenges, and we have yet to delve into creating game objects, assets, etc. Evidently, there's considerable groundwork ahead. Prepare yourself, as we are about to embark on this adventure.

5.1 Renderer Components

OpenGL relies on multiple types of buffers to render objects, among which the vertex buffer and frame buffer are widely utilized. These two buffers are key components for our rendering engine. Let us implement them in our project.

Expand your project's directory tree as illustrated in (Figure 5.1).

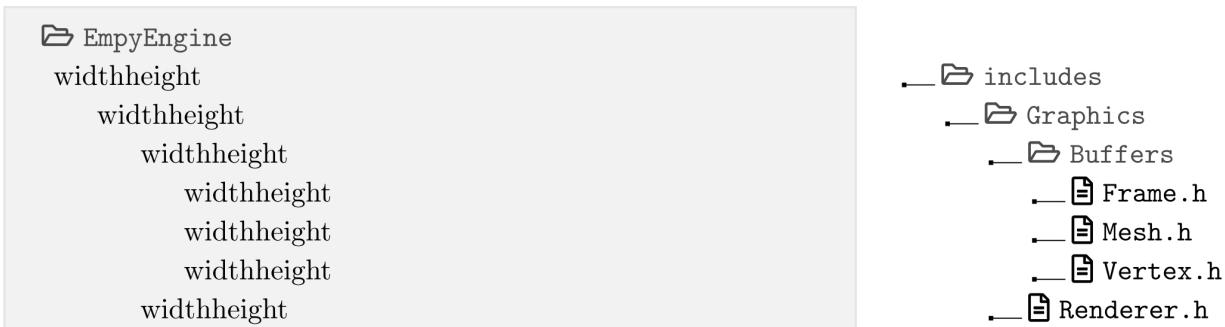


Figure 5.1: Graphics Folder Structure

5.1.1 OpenGL Loader

To interact with the GPU and create our buffers using OpenGL, it's necessary to load the specific GPU's implementation of the API. This task is typically accomplished using tools such as OpenGL Extension Wrangler Library (GLEW) and OpenGL Utility Toolkit (GLUT). These tools play a vital role in OpenGL programming by handling OpenGL extensions across various hardware and operating systems. In this book, we will focus on utilizing GLEW specifically.

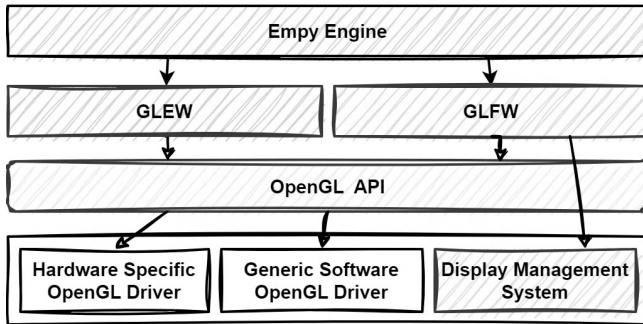


Figure 5.2: Engine, OpenGL and OS interaction

GLEW serves as a critical intermediary between our application and the underlying OpenGL drivers. It ensures that our code can effortlessly access the expanded capabilities offered by OpenGL extensions, facilitating smoother integration and utilization of advanced graphic functionalities. See (Figure [5.2](#)).

Conan effortlessly streamlines the integration of OpenGL and GLEW, As you can see in ([Listing 5.1](#)), we have added GLEW as a static library just as we did for GLFW.

Listing 5.1: Root/conanfile.txt

```
[requires]
spdlog/1.12.0
opengl/system
glfw/3.3.8
glew/2.2.0
```

```
[generators]
cmake

[options]
glew:shared=False
glfw:shared=False
```

We are required to define the `GLEW_STATIC` preprocessor in our code before including GLEW or in the "**CMakeLists.txt**" using the `target_compile_definitions()` function, as we did for the `EMPY_EXPORT` symbol. OpenGL is linked from the local system, which means that Conan will search for OpenGL on the local machine.

Listing 5.2: Common/Core.
 h

```
#pragma once
#define GLEW_STATIC
#include <GL/glew.h>
```

Include GLEW to your project as shown above in (Listing [5.2](#)).

Enable syntax highlighting for GLEW in your project, by setting its include directory in your `c_cpp_properties.json` file, as we did previously for "spdlog" in ([Section 2.5.8](#)).

5.1.2 Initializing OpenGL Context

For OpenGL to render content within our window, we are required to create and initialize an "OpenGL Context". This context acts as the interface, allowing OpenGL to interact with the rendering system of our application and enabling OpenGL to leverage the resources and capabilities of the underlying system, ensuring smooth and efficient rendering of graphics within the designated window.

We have already created this context in the `AppWindow` constructor in (Section [3.2](#)). If you look into it, you will find this function call `glfwMakeContextCurrent()` which helps create the context. All we now have to do is use GLEW to initialize OpenGL.

Add the code depicted in ([Listing 5.3](#)) to your **"Renderer.h"** header file. This ensures that OpenGL is initialized and ready to work.

Listing 5.3: Graphics/Renderer. h

```
#pragma once
#include "Events.h"

namespace EmPy
{
    struct GraphicsRenderer
    {
```

```
EMPTY_INLINE GraphicsRenderer(int32_t width, int32_t
height)
{
    // initialize opengl
    if(glewInit() != GLEW_OK)
    {
        EMPTY_FATAL("failed to init glew!");
        exit(EXIT_FAILURE);
    }
    glewExperimental = GL_TRUE;
}
};
```

To create a `GraphicsRenderer` instance without encountering errors, it is essential to have an existing context before invoking the `glewInit()` function. This means that the application window has to be created before we can initialize OpenGL.

5.1.3 OpenGL Mathematics

Graphics rendering rely on a massive amount of mathematical operations. Mathematics forms the backbone of OpenGL, facilitating transformations, projections, and geometric manipulations in 3D graphics. In this book, we will heavily rely on the OpenGL Mathematics (GLM) library, a robust math toolkit specifically designed for graphics programming. GLM provides a rich set of functionalities for vectors,

matrices, transformations, and geometric calculations, enabling precise control and manipulation of graphics elements within the OpenGL environment.

Link GLM to your project as shown in (Listing 5.4).

Listing 5.4: Root/conanfile.txt

```
[requires]
glm/cci.20230113
opengl/system
spdlog/1.12.0
glfw/3.3.8
glew/2.2.0

[generators]
cmake

[options]
glew:shared=False
glfw:shared=False
```

Unlike GLFW and GLEW, GLM is added as a header-only library similar to what we did with "spdlog".

Listing 5.5: Common/Core.h

```
#pragma once
#include <glm/glm.hpp>
#include <glm/gtc/type_ptr.hpp>
#include <glm/gtx/quaternion.hpp>
#include <glm/gtx/matrix_decompose.hpp>
```

```
#include <glm/ext/matrix_transform.hpp>
#include <glm/ext/matrix_clip_space.hpp>
```

Make sure you include GLM to your project as shown in (Listing 5.5).

5.1.4 Vertex Buffer

Now that we have the prerequisites in place, let us start with the vertex buffer. As introduced in (Section 4.3.1), a vertex buffer is an OpenGL feature that provides methods for uploading vertex data such (position, texture coordinates, normal, color, etc.) to GPU for rendering. We first need to define what a vertex is before we can create a buffer from it. Go ahead and add the following code to your project.

Listing 5.6: Graphics/Buffers/Vertex.
h

```
#pragma once
#include "Common/Core.h"

namespace EmPy
{
    // quad vertex
    struct QuadVertex
    {
        float Data[4] = {0.0f, 0.0f, 0.0f, 0.0f};
    };

    // flat vertex
```

```

struct FlatVertex
{
    glm::vec3 Position = glm::vec3(0.0f);
    glm::vec4 Color = glm::vec4(0.0f);
};

// shading vertex
struct ShadedVertex
{
    glm::vec3 Position = glm::vec3(0.0f);
    glm::vec3 Normal = glm::vec3(0.0f);
    glm::vec2 UVs = glm::vec2(0.0f);
};
}

```

We have defined multiple types of vertex, each serving a different purpose. We could just combine all the attributes into a single vertex type, but that would be a waste of memory and performance. Of course, it makes it a bit more complex to maintain, but the effort is worth it. Here, we provide an overview of each vertex type.

- **Quad Vertex:** In graphics rendering, triangles are generally used as primitive for object composition. The reason is that if you take three random points in space, you will always be able to form a triangle surface by connecting them. The fact that it has a surface is crucial because it gives the GPU the ability to play around with the surface's look to generate nice looking graphics. See ([Figure 5.3](#)).

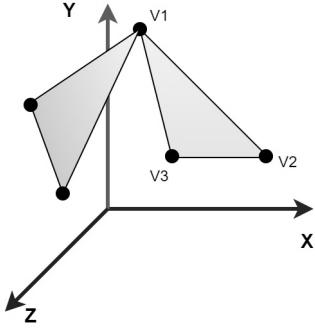


Figure 5.3: Triangle Primitive

A quad in OpenGL is also a primitive, which is basically made from combining two triangles, as shown in (Figure 5.4). Quads find extensive use in graphics programming for tasks like rendering simple 2D sprites, GUI elements, and basic shapes. Historically, they were widely employed in earlier versions of OpenGL for rendering surfaces or basic textures due to their simplicity and ease of use.

We will use quads later in this book to render our final scene after all the post-processing steps. The quad in (Listing 5.6) is defined using an array with four floats. The two first floats represent the position in space on the X- and Y-axes. We don't use the Z-axes because our scene is rendered on a 2D plan, which does not require depth. The two next floats represent the texture coordinates, also known as "UV". They instruct the GPU on how to place a texture on the quad's surface. UV mapping

refers to the process of applying a 2D texture image to a 3D surface. The term "UV" stands for the two coordinates (U and V) used to map a 2D texture onto a 3D surface.

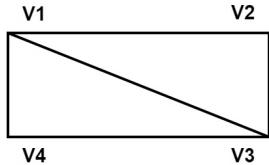


Figure 5.4: Quad Primitive

UV coordinates represent a flattened 2D representation of the surface of a 3D object. The U-coordinate corresponds to the horizontal axis, while the V-coordinate corresponds to the vertical axis of the texture image. These coordinates define how the texture wraps around and adheres to the surface geometry of the 3D model.

- **Flat Vertex:** The flat vertex only has position and color attributes. Objects made with this type of vertex are not affected by light or any shading effect.
- **Shaded Vertex:** The last vertex type is the one we will mostly use to represent our game objects. As you can see, it has attributes that allow texture mapping as well as shading.

There are many more vertex types that can be added to this list; it all depends on what we want to achieve. For instance, an animated object would also have different vertex types. It is important to remember that a vertex is the most primitive element that is used to compose objects.

Buffer Data

With our vertex types now established, our next step involves creating a data structure to store them before transmission to GPU for subsequent processing. In (Listing 5.7), you'll find a templated structure named `MeshData`, tailored to accommodate various vertex types by using the vertex type as a template argument. A vertex buffer can also be called "Mesh". The versatility of this structure allows seamless interaction with multiple vertex types while ensuring code maintainability.

Listing 5.7: Graphics/Buffers/Vertex. h

```
#pragma once
#include "Common/Core.h"

namespace EmPy
{
    /* ... same as before ... */
}
```

```

// vertex buffer data
template <typename Vertex>
struct MeshData
{
    std::vector<uint32_t> Indices;
    std::vector<Vertex> Vertices;
};

}

```

You will also notice that `MeshData` has another member variable called `Indices`. In OpenGL, mesh indices play a crucial role in optimizing the rendering of 3D models. They are used in combination with vertex data to define the geometry of 3D objects efficiently. A mesh index buffer refers to a list of integers that represent the order in which vertices are connected to form primitives, defining the faces of the mesh.

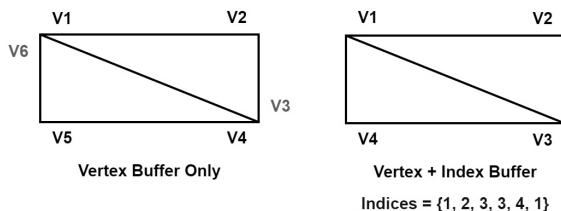


Figure 5.5: Impact of Index Buffers

Using indices allows one to reuse vertices. Instead of duplicating vertices that are shared among multiple triangles (faces), indices reference these shared vertices. This reduces memory usage by storing unique

vertices once and referencing them through indices, which is especially valuable when dealing with complex models with many shared vertices. See (Figure 5.5).

Mesh Class

The code in (Listing 5.8) defines a templated structure `Mesh` that is used to manage OpenGL vertex buffers for different types of vertices. The constructor takes a `MeshData`. It first checks if vertices are provided and initializes variables for the number of vertices and indices accordingly. Then, it generates and binds Vertex Array Object (VAO) and Vertex Buffer Object (VBO) for the vertices, along with Element Buffer Object (EBO) if indices are present.

Listing 5.8: Graphics/Buffers/Mesh.
h

```
#pragma once
#include "Vertex.h"

namespace EmPy
{
    template <typename Vertex>
    struct Mesh
    {
        EMPY_INLINE Mesh(const MeshData<Vertex>& data)
        {
            if(data.Vertices.empty())
            {
```

```

        EMPTY_ERROR("Mesh()! empty mesh data");
        return;
    }

    // number of vertices and indices
    m_NbrVertex = data.Vertices.size();
    m_NbrIndex = data.Indices.size();

    // generate vertex buffer array
    glGenVertexArrays(1, &m_BufferID);

    // active/bind vertex array
    glBindVertexArray(m_BufferID);

    // create vertex buffer
    uint32_t VBO = 0u;
    glGenBuffers(1, &VBO);
    glBindBuffer(GL_ARRAY_BUFFER, VBO);
    glBufferData(GL_ARRAY_BUFFER, m_NbrVertex *
        sizeof(Vertex), data.Vertices.data(),
    GL_STATIC_DRAW);

    // create element buffer
    if(m_NbrIndex != 0u)
    {
        uint32_t EBO = 0u;
        glGenBuffers(1, &EBO);
        glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, EBO);
        glBufferData(GL_ELEMENT_ARRAY_BUFFER,
        m_NbrIndex *
            sizeof(uint32_t), data.Indices.data(),
    GL_STATIC_DRAW);
    }

    // handle vertex types
    if (TypeID<Vertex>() == TypeID<ShadedVertex>())

```

```

        {
            SetAttribute(0, 3,
(void*)offsetof(ShadedVertex, Position));
            SetAttribute(1, 3,
(void*)offsetof(ShadedVertex, Normal));
            SetAttribute(2, 2,
(void*)offsetof(ShadedVertex, UVs));
        }
        else if (TypeID<Vertex>() == TypeID<FlatVertex>
())
{
    SetAttribute(0, 3,
(void*)offsetof(FlatVertex, Position));
    SetAttribute(1, 4,
(void*)offsetof(FlatVertex, Color));
}
else if (TypeID<Vertex>() == TypeID<QuadVertex>
())
{
    SetAttribute(0, 4,
(void*)offsetof(QuadVertex, Data));
}
else
{
    EMFY_ERROR(false && "invalid vertex type!");
}

// unbind vertex array
glBindVertexArray(0);
}

EMFY_INLINE void Draw(uint32_t mode)
{
    glBindVertexArray(m_BufferID);
    if(m_NbrIndex != 0u)
{

```

```

                glDrawElements(mode, m_NbrIndex,
GL_UNSIGNED_INT, 0);
                glBindVertexArray(0);
                return;
}
glDrawArrays(mode, 0, m_NbrVertex);
glBindVertexArray(0);
}

EMPTY_INLINE ~Mesh()
{
    glDeleteVertexArrays(1, &m_BufferID);
}

private:
    EMPTY_INLINE void SetAttribute(uint32_t index, int32_t
size, const void* value)
    {
        glEnableVertexAttribArray(index);
        glVertexAttribPointer(index, size, GL_FLOAT,
GL_FALSE, sizeof(Vertex), value);
    }

private:
    uint32_t m_NbrVertex = 0u;
    uint32_t m_NbrIndex = 0u;
    uint32_t m_BufferID = 0u;
};

// 3d mesh
using ShadedMesh = Mesh<ShadedVertex>;
using Mesh3D = std::shared_ptr<ShadedMesh>;
}
```

The code distinguishes between different types of vertex using conditional checks. For each type, it sets vertex attributes such as position, normal, color, or UV coordinates using `SetAttribute()`, ensuring that OpenGL understands how to interpret the vertex data. The `Draw()` function binds the VAO and issues a draw call using either `glDrawElements()` (if indices exist) or `glDrawArrays()` (if indices are absent) based on the rendering mode provided.

Let us take a more detailed look at the newly introduced concepts such as: VAO, VBO, and EBO.

Vertex Array Object

In OpenGL, a VAO serves as a container or state container that stores the configuration of vertex attribute pointers. It encapsulates the setup for vertex attribute arrays and their associated data so that it can be quickly reactivated when needed during rendering.

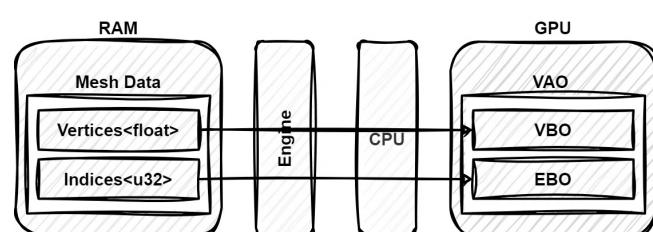


Figure 5.6: Vertex Array Object

The primary purpose of a VAO is to streamline the rendering pipeline by organizing and managing vertex data efficiently. It keeps track of how the attributes of the vertex are organized in memory, allowing OpenGL to understand how to interpret the vertex data stored in the VBO during rendering.

It stores configurations for multiple vertex attribute arrays, including details like attribute locations, data types, offsets, and strides). When binding a VAO as we did in the constructor after creating it, OpenGL stores the configurations of all the vertex attributes, so when the VAO is bound again later in the rendering process as we did in the `Draw()` function, these configurations are quickly applied, saving time and resources in setting up the vertex data.

Using a VAO simplifies the process of rendering multiple objects or using different vertex formats within a scene. Instead of setting up the vertex attributes repeatedly for each draw call, a VAO allows us to configure the attributes once and reuse them when rendering different objects.

Vertex Buffer Object

A VBO is basically where the vertices are actually stored in the GPU memory. It serves as a container for information such as vertex positions, colors, normals, texture coordinates, and other attributes that define the geometry of 3D objects. (Figure 5.7)

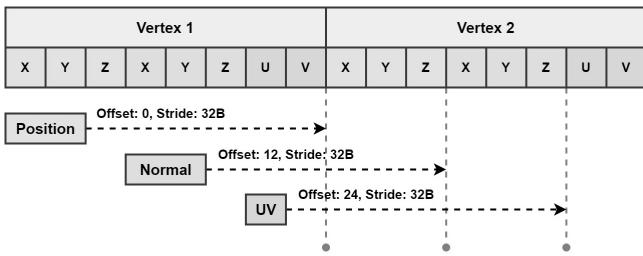


Figure 5.7: Vertex Buffer Layout

When creating a VBO, data is uploaded from the CPU's memory to the GPU's memory, where it can be directly accessed by shaders during rendering as depicted in (Figure 5.6). This process optimizes performance by reducing the need to transfer data between CPU and GPU for each frame, improving rendering efficiency.

VBOs can be used in various ways, including storing static data that remains constant throughout rendering, dynamic data that changes frequently, or streaming data for continual updates. They provide flexibility in managing and organizing vertex information for different rendering scenarios and are a fundamental component in modern OpenGL.

applications for efficient and optimized rendering of 3D geometry.

Element Buffer Object

An EBO also known as an Index Buffer Object (IBO), is a buffer that is used to store indices that reference vertices in VBOs. The primary purpose of an EBO is to optimize memory usage and rendering performance by allowing the reuse of vertices. Instead of duplicate shared vertices for each face of an object, EBOs refers to these vertices through indices.

Vertex Attributes

Vertex attributes define the structure of vertex data and how it should be interpreted during rendering. They describe the layout and organization of data within a VBO, specifying the type, size, and location of each attribute within a vertex. These attributes are associated with specific locations known as attribute indices within the vertex shader. When rendering, OpenGL uses these indices to access the corresponding vertex attributes defined in the vertex shader program. This is depicted in the vertex shader example code below.

You can see that these attributes are simply a reflection of our `ShadedVertex` type we created earlier. The attributes are set using functions such as

`glVertexAttribPointer()` and

`glEnableVertexAttribArray()`.

`glVertexAttribPointer()` configures the data format for a specific attribute, defining its size, type, and stride (the offset in byte between consecutive attribute entries. See (Figure 5.7), and pointer (the offset of the first component of the attribute within the vertex data). Enabling a vertex attribute using

`glEnableVertexAttribArray()` activates it for rendering, indicating that it should be used during the rendering process.

Listing 5.9: Vertex Attributes in a Vertex Shader

```
#version 330 core

// vertex attributes
layout (location = 0) in vec3 a_position;
layout (location = 1) in vec3 a_normal;
layout (location = 2) in vec2 aUvs;

void main()
{
    gl_Position = vec4(a_position, 1.0);
}
```

Creating Meshes

The aim of this chapter is to render an object into our previously created window. For this, we'll generate a basic quad mesh that serves this specific purpose.

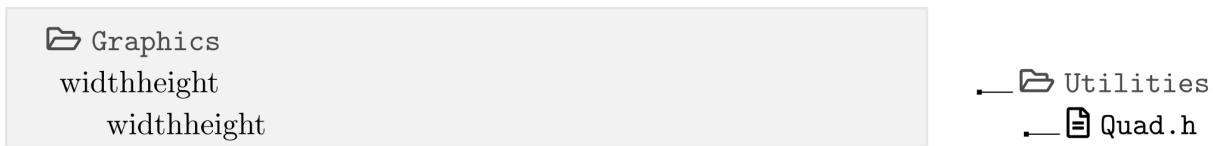


Figure 5.8: Graphics Utilities Folder

Extend your project directory tree as depicted in (Figure 5.8).

Listing 5.10: Graphics/Utilities/Quad.h

```
#pragma once
#include "../Buffers/Mesh.h"

namespace EmPy
{
    using Quad3D = std::unique_ptr<Mesh<ShadedVertex>>;
    using Quad2D = std::unique_ptr<Mesh<QuadVertex>>;

    // creates 2d quad mesh
    EMPY_INLINE Quad2D CreateQuad2D()
    {
        MeshData<QuadVertex> data;

        data.Vertices =
    {
```

```

        {-1.0f, -1.0f, 0.0f, 0.0f },
        {-1.0f, 1.0f, 0.0f, 1.0f },
        { 1.0f, 1.0f, 1.0f, 1.0f },
        { 1.0f, -1.0f, 1.0f, 0.0f }
    } ;

    data.Indices =
{
    0, 1, 2,
    0, 2, 3
} ;

return std::make_unique<Mesh<QuadVertex>>
(std::move(data));
}

// creates 3d quad mesh
EMPTY_INLINE Quad3D CreateQuad3D()
{
    MeshData<ShadedVertex> data;

    // Define vertices for the quad
    ShadedVertex v0, v1, v2, v3;

    v0.Position = glm::vec3(-0.5f, -0.5f, 0.0f); // Bottom-left
    v1.Position = glm::vec3(0.5f, -0.5f, 0.0f); // Bottom-right
    v2.Position = glm::vec3(0.5f, 0.5f, 0.0f); // Top-right
    v3.Position = glm::vec3(-0.5f, 0.5f, 0.0f); // Top-left

    // Define normals (in this case, all normals point
    // in the same direction)
    v0.Normal = glm::vec3(0.0f, 0.0f, 1.0f);
}

```

```

    v1.Normal = glm::vec3(0.0f, 0.0f, 1.0f);
    v2.Normal = glm::vec3(0.0f, 0.0f, 1.0f);
    v3.Normal = glm::vec3(0.0f, 0.0f, 1.0f);

    // Define UVs (texture coordinates)
    v0.UVs = glm::vec2(0.0f, 0.0f);
    v1.UVs = glm::vec2(1.0f, 0.0f);
    v2.UVs = glm::vec2(1.0f, 1.0f);
    v3.UVs = glm::vec2(0.0f, 1.0f);

    // Add vertices to the MeshData structure
    data.Vertices.push_back(v0);
    data.Vertices.push_back(v1);
    data.Vertices.push_back(v2);
    data.Vertices.push_back(v3);

    // Define indices to form two triangles (quad)
    data.Indices.push_back(0); // Triangle 1: v0, v1, v2
    data.Indices.push_back(1);
    data.Indices.push_back(2);
    data.Indices.push_back(0); // Triangle 2: v0, v2, v3
    data.Indices.push_back(2);
    data.Indices.push_back(3);

    return std::make_unique<Mesh<ShadedVertex>>
(std::move(data));
}
}

```

The provided code snippet in (Listing 5.10) comprises two functions dedicated to generating distinct quad meshes. The initial function creates a 2D quad using our defined `QuadVertex`, while the subsequent one creates a 3D quad utilizing the `ShadedVertex`. The 2D

quad serves the purpose of rendering the final scene image onto the screen, while the 3D quad stands as our initial object to be rendered. Let us create a frame buffer that will help render this quad mesh.

5.1.5 Frame Buffer

Frame buffers in OpenGL are also called Frame Buffer Objects (FBOs). A FBO is a specialized object that is used to render to off-screen buffers instead of the default frame buffer (which usually represents the window or screen). The primary purpose of an FBO is to facilitate advanced rendering techniques by allowing the rendering of images or scenes to textures or render buffers that are not directly displayed on the screen. This off-screen rendering is useful for implementing various effects, shadows, bloom, reflections, and more. See ([Figure 5.9](#))

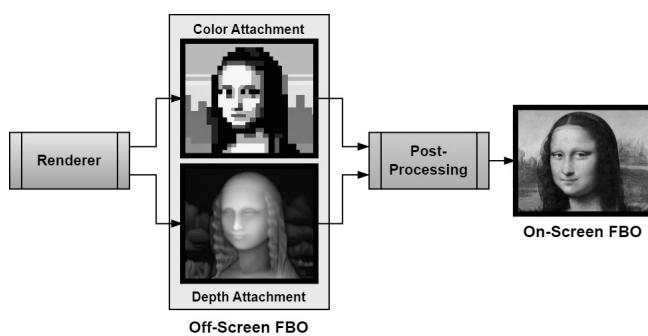


Figure 5.9: Rendering with Frame Buffers

FBOs contain attachments, which are the render targets where the rendering output is directed. These attachments can be textures or render buffers that store color information, depth data, stencil data, or a combination of these. By attaching these buffers to the FBO, OpenGL renders the scene or objects directly into these off-screen buffers. Once the rendering process is complete, the content of the attached render targets in the FBO can be used as textures in subsequent rendering passes or can be displayed on the screen by drawing a textured quad. See (Figure [5.9](#)).

Render Buffer Object

A Render Buffer Object (RBO) is a specialized buffer used specifically to store various types of renderable image data. Unlike textures, which can be sampled and used for various purposes, render buffers are primarily used as render targets in FBOs for off-screen rendering.

The primary purpose of a RBO is to serve as a storage medium for specific renderable data, such as depth information, stencil information, or color information that doesn't require sampling. Render buffers are typically used when data do not need to be accessed directly as a texture, making them more efficient for specific rendering tasks.

Render buffers are often used as attachments to FBOs, providing dedicated storage for rendering operations. They are created and configured similarly to textures, but their main distinction lies in their use case: render buffers are optimized for rendering tasks and are typically faster for operations that don't require texture sampling.

Frame Buffer Class

The code in (Listing 5.11) defines a `FrameBuffer` structure, serving as a wrapper around an OpenGL FBO. The `FrameBuffer` constructor generates the FBO using `glGenFramebuffers()` and binds it to the OpenGL context using `glBindFramebuffer()`.

OpenGL is a stateful API. The stateful nature refers to its internal storage of configuration settings that persist until explicitly modified. When you set a state, like binding a vertex or a frame buffer, these settings remain active until altered again.

This design offers flexibility but requires careful management to avoid unintended side effects. We need to be mindful of the current OpenGL state to ensure desired rendering outcomes, emphasizing explicit state changes and minimizing unnecessary

modifications for efficient and predictable rendering operations.

Listing 5.11: Graphics/Buffers/Frame.h

```
#pragma once
#include "Common/Core.h"

namespace EmPy
{
    struct FrameBuffer
    {
        EMPY_INLINE FrameBuffer(int32_t width, int32_t height):
            m_Width(width), m_Height(height)
        {
            glGenFramebuffers(1, &m_BufferID);
            glBindFramebuffer(GL_FRAMEBUFFER, m_BufferID);

            CreateColorAttachment();
            CreateRenderBuffer();

            // Attachment Targets
            uint32_t attachments[1] =
            {
                GL_COLOR_ATTACHMENT0
            };

            glDrawBuffers(1, attachments);

            // check frame buffer
            if (glCheckFramebufferStatus(GL_FRAMEBUFFER) != GL_FRAMEBUFFER_COMPLETE)
            {

```

```

        EMPTY_ERROR("glCheckFramebufferStatus()
Failed!"); }

        // unbind frame buffer
        glBindFramebuffer(GL_FRAMEBUFFER, 0);
    }

EMPTY_INLINE ~FrameBuffer()
{
    glDeleteTextures(1, &m_Color);
    glDeleteRenderbuffers(1, &m_Render);
    glDeleteFramebuffers(1, &m_BufferID);
}

EMPTY_INLINE float Ratio()
{
    return (float)m_Width/(float)m_Height;
}

EMPTY_INLINE void Resize(int32_t width, int32_t
height)
{
    // update size
    m_Width = width;
    m_Height = height;

    // Resize Color Attachment
    glBindTexture(GL_TEXTURE_2D, m_Color);
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA16F,
m_Width, m_Height, 0, GL_RGBA, GL_FLOAT, NULL);
    glBindTexture(GL_TEXTURE_2D, 0);

    // Resize Depth Attachment
    glBindRenderbuffer(GL_RENDERBUFFER, m_Render);
    glRenderbufferStorage(GL_RENDERBUFFER,

```

```

GL_DEPTH_COMPONENT24, m_Width, m_Height);
glBindRenderbuffer(GL_RENDERBUFFER, 0);
}

EMPTY_INLINE uint32_t GetTexture()
{
    return m_Color;
}

EMPTY_INLINE void Begin()
{
    glBindFramebuffer(GL_FRAMEBUFFER, m_BufferID);
    glViewport(0, 0, m_Width, m_Height);
    glClearColor(0, 0, 0, 1);

    glClear(GL_COLOR_BUFFER_BIT |
GL_DEPTH_BUFFER_BIT);
    glEnable(GL_DEPTH_TEST);
    glEnable(GL_SAMPLES);
}

EMPTY_INLINE void End()
{
    glDisable(GL_SAMPLES);
    glDisable(GL_DEPTH_TEST);
    glBindFramebuffer(GL_FRAMEBUFFER, 0);
}

private:
EMPTY_INLINE void CreateColorAttachment()
{
    glGenTextures(1, &m_Color);
    glBindTexture(GL_TEXTURE_2D, m_Color);
    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_S,
GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T,
GL_CLAMP_TO_EDGE);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA16F,
m_Width, m_Height, 0, GL_RGBA, GL_FLOAT, NULL);
glFramebufferTexture2D(GL_FRAMEBUFFER,
GL_COLOR_ATTACHMENT0, GL_TEXTURE_2D, m_Color, 0);
}

EMPTY_INLINE void CreateRenderBuffer()
{
    glGenRenderbuffers(1, &m_Render);
    glBindRenderbuffer(GL_RENDERBUFFER, m_Render);
    glRenderbufferStorage(GL_RENDERBUFFER,
GL_DEPTH_COMPONENT24, m_Width, m_Height);
    glFramebufferRenderbuffer(GL_FRAMEBUFFER,
GL_DEPTH_ATTACHMENT, GL_RENDERBUFFER, m_Render);
}

private:
    uint32_t m_BufferID = 0u;
    uint32_t m_Render = 0u;
    uint32_t m_Color = 0u;
    int32_t m_Height = 0;
    int32_t m_Width = 0;
};
}

```

This structure manages two key attachments to the frame buffer: a color attachment and a render buffer attachment for depth information. The `CreateColorAttachment()` method generates and

configures a texture `m_Color` as the color attachment. It sets texture parameters and attaches it to the frame buffer as a color buffer using

```
glFramebufferTexture2D().
```

The `CreateRenderBuffer()` method generates and configures a render buffer `m_Render` to store depth information. This render buffer is attached to the frame buffer as a depth buffer using

```
glFramebufferRenderbuffer().
```

It then includes methods for resizing the frame buffer `Resize()`, obtaining the frame buffer texture `GetTexture`, preparing the frame buffer for rendering `Begin()`, and finalizing rendering operations `End()`. The `Begin` method binds the frame buffer, sets the viewport, clears buffers, and enables depth testing and samples. The `End()` method disables depth testing and samples and unbinds the frame buffer.

Depth Buffer

In OpenGL, the depth buffer, also known as the Z-buffer, plays a critical role in rendering scenes with realistic depth perception and handling occlusions accurately. It's a specialized buffer that stores the depth information for each pixel in the frame buffer.

The depth buffer keeps track of the depth or distance of objects from the viewer's perspective in the scene. When rendering, for each pixel on the screen, the depth buffer stores the depth value of the closest object at that pixel's location. During subsequent rendering passes, when a new pixel's depth value is computed, it's compared against the existing depth value in the buffer. If the new pixel's depth is closer to the viewer, it's written to the buffer, replacing the existing value. Otherwise, it's discarded.

This mechanism is crucial for handling the rendering order of objects in 3D scenes. It enables proper occlusion handling by ensuring that only the closest visible pixels are rendered, preventing farther objects from obscuring nearer ones. This process significantly enhances the realism of 3D scenes, as it mimics how objects obscure each other based on their position relative to the viewer.

In OpenGL, enabling and managing the depth buffer using functions like `glEnable(GL_DEPTH_TEST)` and `glDepthFunc()` allows for proper depth testing and comparison operations. Without a depth buffer, rendering complex scenes in correct depth order would be extremely challenging, potentially leading to incorrect visual outcomes with objects improperly occluding each other based solely on their drawing order, rather than their actual depth in the scene.

Multi-Sampling

Multi-sampling in OpenGL is a technique used to improve the visual quality of rendered images by reducing aliasing or jagged edges (also known as “jaggies”) that occur in computer graphics, especially along geometric edges or high-contrast areas. It’s a form of anti-aliasing that enhances image smoothness.

The concept involves sampling multiple points within a pixel and averaging their values to determine the final color of that pixel. Instead of just considering one sample (as in standard rendering), multi-sampling evaluates several samples within each pixel’s area, typically at sub-pixel locations. This technique mitigates the effect of sliding downstairs, resulting in smoother edges and more accurate representation of fine details.

In OpenGL, multi-sampling is enabled using `glEnable(GL_MULTISAMPLE)` and configured with parameters such as the number of samples per pixel `glSampleCoverage()`. In fact, we used GLFW to set the number of sampling steps in the constructor `AppWindow'` using the function `glfwWindowHint(GLFW_SAMPLES, 4)`. This functionality is commonly used in rendering contexts where visual quality is a priority, such as

gaming, high-fidelity graphics, or applications requiring smooth, high-resolution images.

Texture Parameters

The frame buffer code provided in (Listing 5.11) involves configuring texture parameters within the `CreateColorAttachment()` method. Here's a brief explanation of these parameters:

- ☞ **GL_TEXTURE_MIN_FILTER:** This parameter represents the texture minification filter, which determines how the texture is sampled when it needs to be displayed at a size smaller than its original resolution. `GL_LINEAR` specifies that linear interpolation should be used when sampling the texture down, resulting in smoother transitions between texels but potentially causing blurring. We could also use `GL_NEAREST` to sample from the nearest surrounding texel.
- ☞ **GL_TEXTURE_MAG_FILTER :** This parameter represents the texture's magnification filter, defining how the texture is sampled when it needs to be displayed at a size larger than its original resolution. Again, `GL_LINEAR` specifies linear interpolation for smoother magnification.

- ☞ **GL_TEXTURE_WRAP_S:** This parameter sets the texture wrapping behavior along the S-axis (horizontal direction). `GL_CLAMP_TO_EDGE` specifies that the texture coordinates outside the range [0, 1] will be clamped to the edge texels of the texture. This avoids texture-wrapping artifacts at the edges by using the color of the texture's edge pixels for coordinates outside the [0, 1] range.
- ☞ **GL_TEXTURE_WRAP_T:** Similarly, this parameter sets the texture's wrapping behavior along the T-axis (vertical direction) to `GL_CLAMP_TO_EDGE`, preventing wrapping artifacts along the texture's vertical edges.

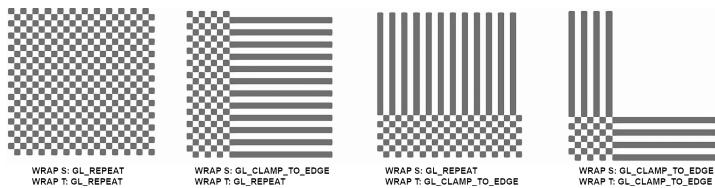


Figure 5.10: Texture Parameters

(Figure 5.10) provides a summary of the texture wrapping parameters and their visual output. These parameters affect how the texture is sampled and how it behaves when the UV coordinates extend beyond the texture's boundaries. The settings used here aim to produce smooth interpolation when scaling the texture and avoid artifacts caused by texture wrapping.

along the edges. Adjusting these parameters can significantly impact the visual quality and behavior of textures in OpenGL rendering.

5.2 Implementing Shaders

Shaders, as we already know, are programs used in modern rendering pipelines to manipulate and process graphical data, ultimately shaping the visual appearance of 3D objects displayed on a screen. These programs run on GPUs and are designed to perform specific tasks in the rendering process. Shaders are divided into various types, such as vertex shaders, fragment shaders, geometry shaders, and compute shaders, each specializing in different aspects of rendering. Vertex shaders handle transformations of 3D vertices, while fragment shaders manage pixel color computations. They allow for sophisticated effects, from simulating lighting and shadows to creating intricate textures and visual effects.

5.2.1 Shader Compilation

Shaders require a compilation step before they can be utilized within an OpenGL program. These shader programs, typically written in GLSL, undergo a compilation process that translates the high-level shader code into a format that GPU can understand

and execute efficiently. Compilation of shaders involves translating the human-readable code into machine-understandable form, performing syntax checks, and optimizing the code for execution on the graphics hardware.

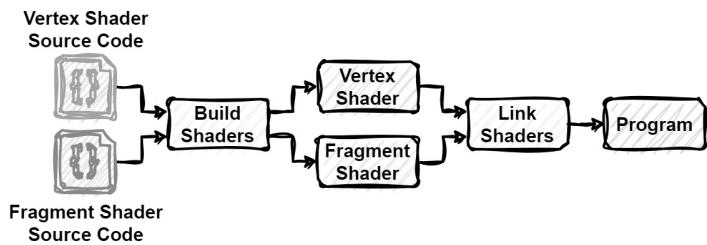


Figure 5.11: Shaders Compilation

When compiling shaders, each type of shader (vertex, fragment, geometry, etc.) is compiled separately. This compilation process involves validating the syntax and semantics of the shader code to ensure correctness and efficiency. Any errors in the code, such as typos, invalid syntax, or unsupported operations, are flagged during this phase, resulting in compilation errors that need to be addressed before the shader can be successfully used for rendering.

Once compiled, the shaders are linked together to form a complete shader program that defines the rendering pipeline. This linkage process ensures that the vertex, fragment, and other shaders work together

cohesively to produce the desired graphical output. During this phase, the shader program is created and can then be utilized within an OpenGL context to render 3D scenes. See (Figure [5.11](#)).

Contrary to the compilation process depicted in (Figure [5.11](#)), where each shader type is implemented in a separate file, we will implement both the vertex and fragment shaders in single files to simplify their maintenance and loading processes.

5.2.2 GLSL Shaders

GLSL serves as the dedicated language for writing shaders in OpenGL. It's a high-level language designed specifically for GPU programming, facilitating the creation of vertex, fragment, geometry, and compute shaders to manipulate graphics and perform complex calculations on the GPU. GLSL's syntax is simple to understand and quite similar to that of the programming language "C".

Extend your project directory tree to accommodate (Figure [5.12](#)).

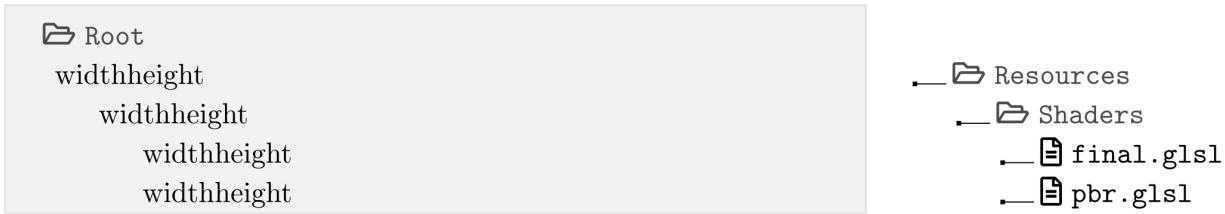


Figure 5.12: Resources Directory

Within the folder tree depicted in (Figure 5.12), we have segregated two files for distinct types of shaders. The initial one, "**final.glsL**", serves as a shader for rendering the conclusive scene onto the screen after all post-processing stages. On the other hand, "**pbr.glsL**" is the shader employed to render the scene in our previously established frame buffer. This separation allows us to tailor each shader specifically to its intended rendering purpose within our engine.

PBR Shaders

Physically Based Rendering (PBR) is a shading and rendering technique in computer graphics that aims to simulate real-world material properties and lighting interactions more accurately. It uses physically derived models to represent how light interacts with surfaces, considering factors like surface roughness, reflectivity, and microfacet behavior, resulting in more realistic and

consistent renderings across different lighting conditions and materials.

The PBR shader for now will not offer advanced features such as lighting, material, shadows, etc. The first step is to ensure that our engine loads, compiles it correctly as well as renders a flat quad mesh. We will dive into the intricacies of PBR shading later in a subsequent section.

Listing 5.12: Resources/Shaders/pbr.gls

```
#version 330 core
layout (location = 0) in vec3 a_position;
layout (location = 1) in vec3 a_normal;
layout (location = 2) in vec2 aUvs;

uniform mat4 u_model;
uniform mat4 u_proj;
uniform mat4 u_view;

void main()
{
    gl_Position = u_proj * u_view * u_model * vec4(a_position,
1.0);
}

++VERTEX++

#version 330 core
layout (location = 0) out vec4 out_fragment;

void main()
```

```
{  
    out_fragment = vec4(0.6, 0.5, 0.7, 1.0);  
}  
  
++FRAGMENT++
```

You can see in the shader code depicted in (Listing 5.12) that the fragment shader, for the moment, straightforwardly outputs a basic color without additional operations. The intricacy primarily resides within the vertex shader.

As previously discussed, OpenGL's rendering process involves a series of operations. In this context, the vertex shader undertakes intricate matrix and vector computations to determine the current vertex's position in the world. This operation specifically revolves around three vital matrices that warrant further explanation.

In computer graphics, Model-View-Projection (MVP) matrices play a pivotal role in transforming 3D objects from their local object space to the final 2D screen space for rendering. These matrices, Model, View, and Projection, form the backbone of the transformation pipeline, allowing precise positioning, orientation, and projection of objects in a 3D scene.

- **Model Matrix:** The model matrix encapsulates the object's local transformations, such as translation,

rotation, and scaling. It transforms vertices from the object's local coordinate system to a world-space coordinate system. It's represented as $\text{Model} = T \times R \times S$, where T is the translation matrix, R is the rotation matrix, and S is the scaling matrix.

- **View Matrix:** The "View" matrix represents the camera's position and orientation in the scene. It transforms the world-space coordinates into camera (or eye) space, aligning the scene according to the viewpoint. This matrix typically involves translating and rotating the entire world in the reverse direction of the camera's movement and rotation. It's given by $\text{View} = \text{lookAt}(\text{eye}, \text{center}, \text{up})$, where 'eye' is the camera's position, 'center' is the point the camera is looking at, and 'up' defines the camera's upward direction.
- **Projection Matrix:** The Projection matrix deals with the conversion of 3D coordinates to 2D screen space, defining the perspective or orthographic projection. It transforms the camera space coordinates to NDC, which are later mapped to the screen space. Common types include perspective ($\text{Perspective} = \text{perspective}(\text{fovy}, \text{aspect}, \text{zNear}, \text{zFar})$) and orthographic projections ($\text{Ortho} = \text{ortho}(\text{left}, \text{right}, \text{bottom}, \text{top}, \text{zNear}, \text{zFar})$).

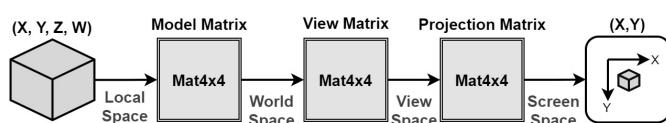


Figure 5.13:

Model View Projection

The MVP matrix is obtained by multiplying these matrices: $\text{MVP} = \text{Projection} \times \text{View} \times \text{Model}$. This combined matrix efficiently transforms object vertices into screen space, considering their local transformations, their positioning concerning the camera, and the perspective or orthographic projection. See ([Figure 5.13](#))

You will also notice that the MVP matrices in the vertex shader are defined using the keyword “uniform”. In OpenGL, uniforms are a type of variable used in shaders that remain constant during the rendering of a particular primitive or across multiple rendering calls. They are explicitly set by the CPU and remain consistent for all vertices or fragments processed by the shader for a specific draw call.

Uniforms are crucial for sending data from the CPU (host application) to the GPU (shaders) and are commonly used for values that don’t change frequently during rendering, such as transformation matrices (like model, view, projection), light properties, material attributes, or any other global data needed across the entire shader.

These variables are declared in shaders using the "uniform" keyword and are set from the CPU or through our engine using functions like

`glUniformMatrix4fv()` in OpenGL. Uniforms offer a way to communicate and synchronize data between the CPU and GPU, enabling dynamic control over various parameters in shaders without needing to recompile the shader program for every change.

`gl_Position` is a built-in variable in OpenGL shaders, specifically used in the vertex shader. It represents the transformed position of a vertex in homogeneous clip coordinates. When computing `gl_Position` within the vertex shader, you're determining the final position of each vertex in a 3D space that's projected onto the 2D screen. This variable holds the vertex's position after being transformed by the MVP matrices and is in homogeneous coordinates, meaning it has four components (x, y, z, w), where 'w' is usually 1.0. The transformed vertex position in `gl_Position` is essential for subsequent steps in the rendering pipeline. Once computed, this position is further processed through perspective division and viewport transformation, ultimately mapping the vertex's position from 3D world space to 2D screen space. The resulting 2D coordinates determine where the vertex appears on the screen.

Final Shader

The shader code depicted in (Listing 5.13) is notably simple compare to the PBR code in (Listing 5.12). Within the vertex shader, the output variable, labeled as `uvs`, contains the texture coordinates relayed from the `QuadVertex`. These specific texture coordinates, subsequently transmitted to the fragment shader, serve the purpose of sampling the scene texture stored in the frame buffer. This process facilitates the rendering of the final scene onto the default or on-screen frame buffer.

Listing 5.13: Resources/Shaders/final.gls
|

```
#version 330 core
layout (location = 0) in vec4 a_quad;
out vec2 uvs;

void main()
{
    uvs = vec2(a_quad.z, a_quad.w);
    gl_Position = vec4(a_quad.x, a_quad.y, 0.0, 1.0);
}

++VERTEX++

#version 330 core
out vec4 out_fragment;
in vec2 uvs;
```

```
uniform sampler2D u_map;

void main()
{
    out_fragment = texture(u_map, uvs);
}

++FRAGMENT++
```

In GLSL, the use of "in" and "out" keywords defines the input and output variables, ensuring their shared access across shaders. Moreover, in GLSL, the equivalent of OpenGL's **TEXTURE_2D** is referred to as `sampler2D`.

5.2.3 Shader Abstraction

Now that we have our GLSL shaders implemented and ready, let us implement a way to load and use them in our engine. Include the directory and files shown in (Figure 5.14) in your project directory tree.

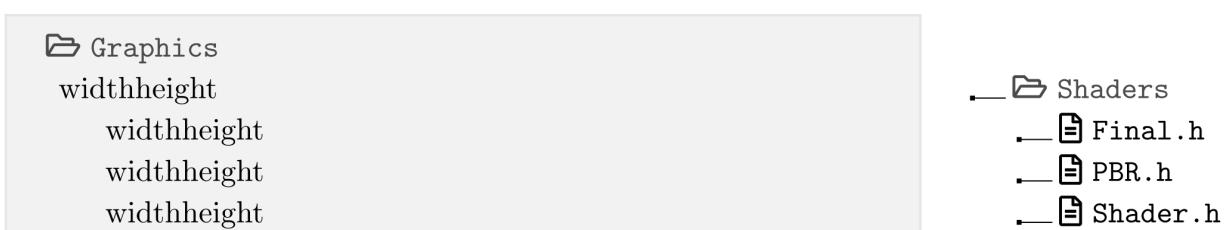


Figure 5.14: Shaders Directory

"Shader.h" will be used to introduce an abstract shader acting as the base for all shader classes, providing essential methods to load, build, and link various shader types. Additionally, you'll notice corresponding shader header files, mirroring those implemented in GLSL.

Listing 5.14: Graphics/Shaders/Shader.h

```
#pragma once
#include "Common/Core.h"

namespace EmPy
{
    struct Shader
    {
        EMPY_INLINE Shader(const std::string& filename)
        {
            m_ShaderID = Load(filename);
        }

        EMPY_INLINE virtual ~Shader()
        {
            glDeleteProgram(m_ShaderID);
        }

        EMPY_INLINE void Unbind()
        {
            glUseProgram(0);
        }

        EMPY_INLINE void Bind()
        {
```

```
        glUseProgram(m_ShaderID);

    }

private:

    EMPLY_INLINE uint32_t Build(const char* src, uint32_t type)
    {
        uint32_t shaderID = glCreateShader(type);
        glShaderSource(shaderID, 1, &src, NULL);
        glCompileShader(shaderID);

        char error[512];
        int32_t status = 0;
        glGetShaderiv(shaderID, GL_COMPILE_STATUS,
&status);

        if (!status) {
            glGetShaderInfoLog(shaderID, 512, NULL,
error);
            throw std::runtime_error(error);
            glDeleteShader(shaderID);
            shaderID = 0u;
        }

        return shaderID;
    }

    EMPLY_INLINE uint32_t Link(uint32_t vert, uint32_t frag)
    {
        uint32_t programID = glCreateProgram();
        glAttachShader(programID, vert);
        glAttachShader(programID, frag);
        glLinkProgram(programID);

        char error[512];
```

```
        int32_t status = 0;
        glGetProgramiv(programID, GL_LINK_STATUS,
&status);

        if (!status) {
            glGetProgramInfoLog(programID, 512, NULL,
error);
            throw std::runtime_error(error);
            glDeleteProgram(programID);
        }

        glDeleteShader(vert);
        glDeleteShader(frag);

        return programID;
    }

EMPTY_INLINE uint32_t Load(const std::string&
filename)
{
    std::ifstream fs;
    fs.exceptions(std::ifstream::failbit |
std::ifstream::badbit);
    try
    {
        bool loading_vtx_source = true;
        fs.open(filename);

        std::string line;
        std::string vtxSource;
        std::string fragSource;

        // load vtx & frag source
        while(getline(fs, line))
        {
            if(loading_vtx_source)
```

```

    {
        if (line.compare("++VERTEX++"))
        {
            vtxSource.append(line + "\n");
            continue;
        }
        loading_vtx_source = false;
        continue;
    }
    else
    {
        if (!line.compare("++FRAGMENT++"))
        {
            break;
        }
        fragSource.append(line + "\n");
    }
    fs.close();
}

uint32_t vtxShader = Build(vtxSource.c_str(),
GL_VERTEX_SHADER);
uint32_t fragShader =
Build(fragSource.c_str(), GL_FRAGMENT_SHADER);
return Link(vtxShader, fragShader);
}
catch (const std::exception& e)
{
    EMPTY_ERROR("Load('{})' Failed: {}", filename,
e.what());
}
return 0;
}

protected:
uint32_t m_ShaderID = 0u;

```

```
};  
}
```

The code in (Listing 5.14) defines an abstract shader class. The constructor starts by reading a file that separates vertex and fragment shader source code sections delineated by the markers "**++VERTEX++**" and "**++FRAGMENT++**". The loaded source code is then compiled into vertex and fragment shaders using `Build()` and linked together into a shader program through `Link()`.

The structure also includes methods to bind and unbind the shader program using `Bind()` and `Unbind()`, respectively, utilizing `glUseProgram()` to activate or deactivate the shader. To use a shader in OpenGL, you need to explicitly activate it. It is also important to deactivate it when you are done using it.

The `Load()` method orchestrates the shader loading process, reading the shader source code file, separating it into vertex and fragment sections, compiling each section into respective shaders, and linking them into a complete shader program. Exception handling ensures robust error management during file operations or shader compilation, displaying appropriate error messages in case of failure.

PBR Shader Abstraction

The code depicted in (Listing 5.15) defines our `PbrShader`, which inherits from the abstract shader that we created earlier. Within its constructor, it retrieves the locations of uniform variables named `u_model`, `u_view`, and `u_proj` from the shader program using `glGetUniformLocation()`. Each uniform has a static location that can be requested on the CPU side to set its value.

Listing 5.15: Graphics/Shaders/PBR.gls

|

```
#pragma once
#include "Shader.h"

namespace EmPy
{
    struct PbrShader : Shader
    {
        EMPY_INLINE PbrShader(const std::string& filename) :
        Shader(filename)
        {
            u_Model = glGetUniformLocation(m_ShaderID,
"u_model");
            u_View = glGetUniformLocation(m_ShaderID,
"u_view");
            u_Proj = glGetUniformLocation(m_ShaderID,
"u_proj");
        }

        private:
        uint32_t u_Model = 0u;
        uint32_t u_View = 0u;
```

```
    uint32_t u_Proj = 0u;  
};  
}
```

These uniform locations are stored in the private member variables `u_Model`, `u_View`, and `u_Proj`. These locations will later be used to pass transformation matrices (model, view, and projection) from CPU to the shader to correctly render objects in their respective spaces. This class will evolve with every new requirement that will arise in upcoming sections and chapters.

Final Shader Abstraction

This code defines a structure `FinalShader` that also inherits from our base shader structure. In its constructor, it fetches the location of a uniform variable named `u_map` from the shader program. This uniform location is stored in the private member variable `u_Map`. Additionally, it creates a 'Quad2D' object using the `CreateQuad2D()` function.

Listing 5.16: Graphics/Shaders/Final.gls

```
#pragma once  
#include "Shader.h"  
#include "../Utilities/Quad.h"
```

```

namespace Empy
{
    struct FinalShader : Shader
    {
        EMPY_INLINE FinalShader(const std::string& filename) :
        Shader(filename)
        {
            u_Map = glGetUniformLocation(m_ShaderID,
            "u_map");
            m_Quad = CreateQuad2D();
        }

        EMPY_INLINE void SetSceneMap(uint32_t map)
        {
            glActiveTexture(GL_TEXTURE0);
            glBindTexture(GL_TEXTURE_2D, map);
            glUniform1i(u_Map, 0);
        }

        EMPY_INLINE void Show(uint32_t map)
        {
            glBindFramebuffer(GL_FRAMEBUFFER, 0);
            glClearColor(0, 0, 0, 1);
            glClear(GL_COLOR_BUFFER_BIT);

            glUseProgram(m_ShaderID);
            glActiveTexture(GL_TEXTURE0);
            glBindTexture(GL_TEXTURE_2D, map);
            glUniform1i(u_Map, 0);
            m_Quad->Draw(GL_TRIANGLES);
            glUseProgram(0);
        }

    private:
        uint32_t u_Map = 0u;
        Quad2D m_Quad;
}

```

```
};  
}
```

It also contains two additional methods:

`SetSceneMap(uint32_t)` activates a texture unit, binds the provided texture to `GL_TEXTURE_2D`, and sets the uniform variable `u_Map` to the corresponding texture unit index. `Show(uint32_t)` configures OpenGL to render the final scene in the default frame buffer. It clears the color buffer, set the shader program, activate and bind the provided texture, set the corresponding uniform, and draw the 2D quad using the shader program.

5.3 Entity Component System

We are close to rendering our first object, but we still need to implement some additional components before that can be done. Hang in there. A "scene" in game development is a fundamental component that organizes and represents a specific part of a game's world or environment. It is essentially a collection of various elements, such as objects, characters, settings, lighting, and more, that come together to form a playable section or area within the game.

A challenging aspect of developing games is finding the proper way to represent scenes and organize objects efficiently. Over several decades, developers

have experimented with various methodologies to address this challenge. Among the most renowned techniques is the Object-Oriented Programming (OOP). Traditionally, OOP has represented objects through classes using inheritance, identifying them with properties and behaviors. However, the emergence of the Entity Component System (ECS) design has brought a revolution to the field.

5.3.1 What is ECS?

ECS is a software design pattern that separates data from behaviour to promote code reuse. Components (data) are usually stored in a cache-friendly manner, which enhances performance and memory usage. The cache memory is a chip-based computer component that improves the efficiency with which data is retrieved from the computer's storage. It serves as a temporary storage area from which the computer's processor may easily retrieve data. This temporary storage area, known as a cache, is available to the processor more easily and quickly than the computer's main work storage.

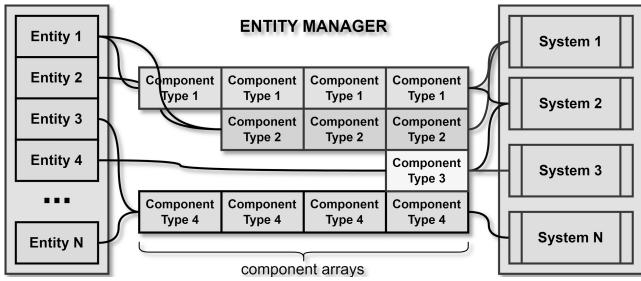


Figure 5.15: Entity Component System Design

ECS breaks down game objects into entities (empty containers or unique identifiers), components (entity's behaviour), and systems (behaviour's logic), allowing for a more modular, flexible, and efficient representation. Components define specific aspects of an object's behavior, like rendering, physics, or AI, while entities assemble these components to form functional game entities. ECS's significance lies in its modularity, enabling reusable components across diverse entities, optimizing performance through parallel processing, and offering scalability, flexibility, and clearer entity design, which collectively streamline game development processes and enhance collaboration among different teams involved in creating intricate game worlds.

ECS Alternatives

To comprehend why ECS stands out as a superior choice for building our game engine, a comparison among various design methodologies is essential. Our exploration will encompass Object-Oriented Design (OOD), Component-Oriented Design (COD), and Component-Based Design (CBD), focusing on critical categories for evaluation such as performance, maintenance, CPU cache efficiency, and memory usage.

- **Object-Oriented Design:** Initially appealing due to its portrayal of games as interactions between objects within a scene, OOP seems to fit the creation of games. However, despite this surface-level suitability, numerous inherent flaws have led many game engines to discard this approach.

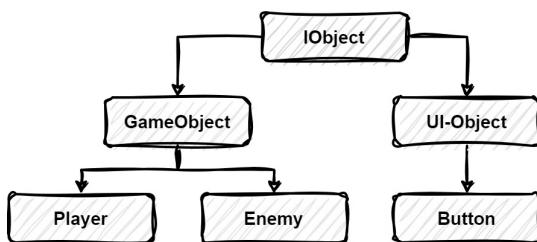


Figure 5.16: Object-Oriented Design

In typical game design, the framework often entails an object interface that houses essential virtual functions such as `update()`, `destroy()`, and `render()`. These functions must be implemented by

all subsequent child classes. The inheritance hierarchy illustrated in (Figure 5.16), exhibits multiple levels, corresponding to a chain of virtual lookup tables. This setup poses a potential performance bottleneck as the processor navigates these tables to locate the appropriate function to execute.

Consider the scenario presented by the classes "Player" and "Enemy." If a requirement arises that an enemy entity also possess player attributes or functionalities, the OOD falters. Accomplishing this necessitates resorting to multiple inheritances, which not only complicate maintenance but also pose challenges to preserving code clarity and integrity.

Furthermore, adherence to the object interface in this paradigm requires memory allocation on the heap for all object types. Regrettably, this approach disrupts memory consistency. When the computer retrieves data from a specific memory address, it not only retrieves the requested data but also pulls surrounding data into the cache memory. This inconsistent memory storage adversely impacts CPU-cache usage, leading to frequent cache misses and consequently undermining the performance.

- **Component-Oriented Design:** The illustration in (Figure 5.17) depicts the structure of a COD, where each game object encompasses a collection of components delineating its behavior. Essential components such as Transform and Sprite, among others, inherit from an abstract class named Component. While this technique does not inherently resolve the memory and CPU-cache concerns present in the OOD, it does mitigate the need for a virtual lookup table for function calls.

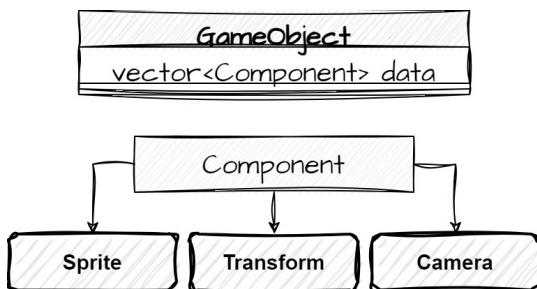


Figure 5.17: Component-Oriented Design

An intriguing advantage of this approach lies in its flexibility: It enables the creation of entities that embody multiple roles, such as an enemy entity also functioning as a player. This versatility stems from the simple addition or removal of components, allowing entities to assume different behaviors dynamically. Consequently, this method outperforms the traditional OOD as it circumvents the overhead of extensive inheritance chains and

virtual function lookups, thereby enhancing overall performance.

- **Data-Oriented Design:** The allure of DOD becomes apparent due to its inherent advantages: a contiguous memory layout, the absence of a virtual lookup table, and optimized CPU-cache utilization. Often referred to as the entity-component design, it stands as an efficient paradigm in game development. The accompanying illustration in (Figure 5.18) serves as a pivotal representation of this concept.

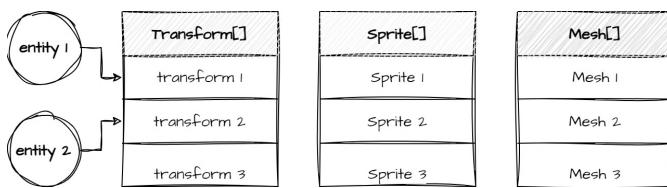


Figure 5.18: Data-Oriented Design

In this approach, each game object is denoted by a specific number, serving as a reference to identify the associated components. In particular, components of identical types across all entities are organized in a cohesive, contiguous manner in memory. This deliberate arrangement significantly enhances CPU-cache efficiency, a pivotal factor in optimizing performance. It is no wonder that numerous game engines embrace this design,

leveraging its capacity to deliver consistent and robust performance while also proving relatively straightforward to maintain.

One of the main differences between DOD and ECS is just the fact that DOD allows components to implement their behaviour's logic themselves instead of using systems. This is a minor difference that does not really effect the overall performance drastically.

5.3.2 Integrating ECS

"**Entt**" stands out as a lightweight, header-only C++ ECS-library. It serves as a powerful framework, facilitating the management of entities, components, and systems with remarkable flexibility and efficiency. While one could opt to develop a custom ECS library tailored for a specific project, leveraging "Entt" proves advantageous. The creation of a custom implementation ECS would consume significant time and resources, which could lead to inefficiencies and maintenance challenges.

Entt's established presence in the industry reinforces its reliability and performance. Notably, it has been adopted by prominent projects like Minecraft, underlining its robustness and suitability for handling

complex entity-based systems in real-world, large-scale applications. This adoption by major projects underscores Entt's capability to handle diverse use cases and scale effectively, making it a trusted choice for managing entities, components, and systems within C++ projects.

To integrate "Entt" into our project, we can simply add it to Conan, as shown below:

Listing 5.17: Root/conanfile.txt

```
[requires]
glm/cci.20230113
opengl/system
spdlog/1.12.0
entt/3.12.2
glfw/3.3.8
glew/2.2.0

[generators]
cmake

[options]
glew:shared=False
glfw:shared=False
```

Here is a simple example of how "EnTT" can be used:

Listing 5.18: EnTT Example

```
#include <entt/entt.hpp>

struct Position
{
    float x, y;
};

struct Velocity
{
    float dx, dy;
};

int main()
{
    entt::registry registry;

    // create entities and attach components
    auto entity = registry.create();
    registry.assign<Position>(entity, 0.0f, 0.0f);
    registry.assign<Velocity>(entity, 1.0f, 1.0f);

    // iterate systems
    while (true)
    {
        registry.view<Position, Velocity>().each([] (auto&
pos, auto& vel)
        {
            // Update positions based on velocities
            pos.x += vel.dx;
            pos.y += vel.dy;
        });
    }

    return 0;
}
```

This example showcases a simple scenario where entities have position and velocity components, and a system updates their positions based on their velocities.

Transform, Camera and Meshes

To render objects effectively, we want to be able to place it in a 3D space, see its physical shape from a camera perspective. These essential elements are encapsulated by three distinct components: the "Transform", "Camera", and "Mesh".

- ☞ **Transform:** The transform defines the object's spatial attributes, including its position, rotation, and scale within the 3D environment. It serves as the foundational component that situates the object accurately within the scene.
- ☞ **Camera:** The camera, on the other hand, specifies the viewpoint and properties of the camera observing the scene. This component dictates how the scene is visualized from a particular perspective, influencing aspects like field of view, projection type, and view matrix.
- ☞ **Mesh:** Lastly, the mesh shapes the object's appearance by defining its geometry, material

properties, and possibly other visual attributes. This component essentially gives the object its visible form within the rendered scene.

Update your project directory tree by adding the "Data.h" header file as shown in (Figure 5.19).

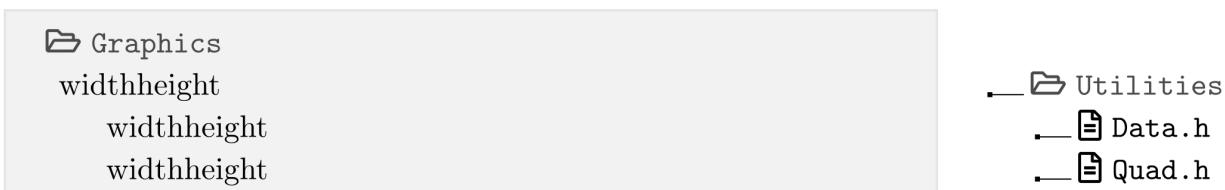


Figure 5.19: Utilities Data

The code snippet in (Listing 5.19) defines essential data for handling 3D graphics in our rendering engine. `shadedMesh` and `Mesh3D` are alias declarations that simplify the usage of a `Mesh` template specialized for `ShadedVertex` type and a unique pointer to this specialized mesh.

Listing 5.19: Graphics/Utilities/Data.h

```
#pragma once
#include "../Buffers/Mesh.h"

namespace EmPy
{
    // transform
```

```

struct Transform3D
{
    EMPLY_INLINE glm::mat4 Matrix() const
    {
        return (glm::translate(glm::mat4(1.0f),
Translate) *
            glm::toMat4(glm::quat(glm::radians(Rotation))) *
            glm::scale(glm::mat4(1.0f), Scale));
    }

    glm::vec3 Translate = glm::vec3(0.0f);
    glm::vec3 Rotation = glm::vec3(0.0f);
    glm::vec3 Scale = glm::vec3(1.0f);
};

// camera
struct Camera3D
{
    EMPLY_INLINE glm::mat4 Frustum(const Transform3D&
transform, float ratio) const
    {
        return Projection(ratio) * View(transform);
    }

    EMPLY_INLINE glm::mat4 View(const Transform3D&
transform) const
    {
        return glm::lookAt(transform.Translate,
(transform.Translate + glm::vec3(0, 0, -1)),
        glm::vec3(0, 1, 0)) *
        glm::toMat4(glm::quat(glm::radians(transform.Rotation)));
    }

    EMPLY_INLINE glm::mat4 Projection(float ratio) const
    {
        return glm::perspective(FOV, ratio, NearPlane,

```

```

        FarPlane) ;
    }

    float NearPlane = 0.3000f;
    float FarPlane = 1000.0f;
    float FOV = 45.0f;
};

}

```

The `Transform3D` represents the transformation attributes of a 3D object. It contains methods to compute a transformation matrix based on translation, rotation, and scaling values. The `Matrix()` method combines these attributes using GLM functions to produce a final transformation matrix.

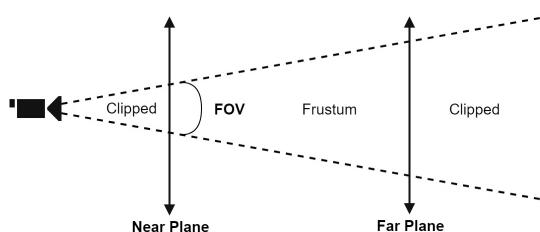


Figure 5.20: Perspective Camera

`Camera3D` models a 3D camera. It offers methods to compute the different matrices used in the camera's view and projection transformations. The `Frustum()` method combines the projection and view matrices; `View()` computes the view matrix; and `Projection()`

generates the projection matrix based on specified parameters such as Field of View (FOV), near and far planes. GLM helps creates all matrices by simply leveraging its built-in functions `glm::view()` and `glm::projection()`.

Update your project's directory tree as depicted below (Figure 5.21).

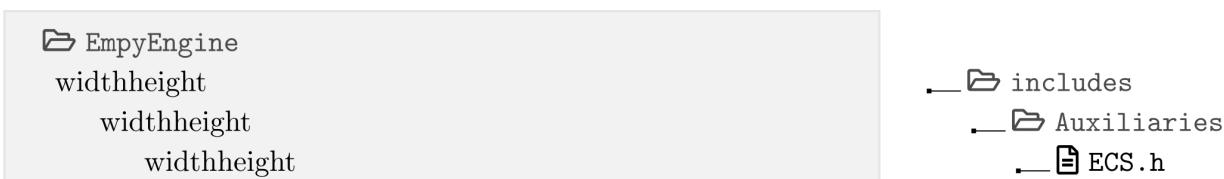


Figure 5.21: EnTT
Wrapper

Component Types

Include EnTT in the "**ECS.h**" header file as shown in (Listing 5.20). We are simply creating type aliases for EnTT to make our interaction with it more convenient. We also define the transform, camera, and mesh components and as you can see, each component type has a public member data member variable.

Listing 5.20: Auxiliaries/ECS. h

```
#pragma once
#include <entt/entt.hpp>
#include "Graphics/Utilities/Data.h"

namespace EmPy
{
    // typedefs
    using EntityID = entt::entity;
    using EntityRegistry = entt::registry;
    constexpr EntityID NENTT = entt::null; // null entity

    // transform component
    struct TransformComponent
    {
        EMPY_INLINE TransformComponent(const TransformComponent&) = default;
        EMPY_INLINE TransformComponent() = default;
        Transform3D Transform;
    };

    // camera component
    struct CameraComponent
    {
        EMPY_INLINE CameraComponent(const CameraComponent&) = default;
        EMPY_INLINE CameraComponent() = default;
        Camera3D Camera;
    };

    // common component
    struct EnttComponent
    {
```

```

    EMPTY_INLINE EnttComponent(const EnttComponent&) =  

default;  

    EMPTY_INLINE EnttComponent() = default;  

    std::string Name = "Untitled";  

};  
  

// mesh component  

struct MeshComponent  

{  

    EMPTY_INLINE MeshComponent(const MeshComponent&) =  

default;  

    EMPTY_INLINE MeshComponent() = default;  

    Mesh3D Mesh;  

};  

}

```

Entity Class

An entity is essentially denoted by a 32-bit unsigned integer identifier. This identifier enables the attachment, detachment, or retrieval of components within the registry for entities. However, to streamline our interaction with game objects, direct manipulation of this entity identifier is not practical. Instead, we opt to create a wrapper entity structure around it, allowing for a more simplified and intuitive handling of game objects. This wrapper provides a layer of abstraction, shielding the direct use of raw identifiers and enhancing the clarity and ease of managing game objects.

Add the following code in your "**ECS.h**" header file.

Listing 5.21: Auxiliaries/ECS.
 h

```
#pragma once
#include <entt/entt.hpp>
#include "Graphics/Utilities/Data.h"

namespace EmPy
{
    /* ... same as before ... */

    struct Entity
    {
        EMPY_INLINE Entity(EntityRegistry* registry, EntityID entity) :
            m_Registry(registry), m_EnttID(entity)
        { }

        EMPY_INLINE Entity(EntityRegistry* registry) :
            m_Registry(registry)
        {
            m_EnttID = m_Registry->create();
        }

        EMPY_INLINE virtual ~Entity() = default;
        EMPY_INLINE Entity() = default;

        EMPY_INLINE operator EntityID () 
        {
            return m_EnttID;
        }

        EMPY_INLINE operator bool()
    };
}
```

```

    {

        return m_Registry != nullptr &&
        m_Registry->valid(m_EnttID);
    }

EMPTY_INLINE EntityID ID()
{
    return m_EnttID;
}

// ++

template<typename T, typename... Args>
EMPTY_INLINE T& Attach(Args&&... args)
{
    return m_Registry->get_or_emplace<T>(m_EnttID,
std::forward<Args>(args)...);
}

template<typename T>
EMPTY_INLINE void Detach()
{
    m_Registry->remove<T>(m_EnttID);
}

EMPTY_INLINE void Destroy()
{
    if(m_Registry) { m_Registry->destroy(m_EnttID); }
}

template<typename T>
EMPTY_INLINE bool Has()
{
    return m_Registry != nullptr &&
    m_Registry->all_of<T>(m_EnttID);
}

```

```

template<typename T>
EMPY_INLINE T& Get()
{
    return m_Registry->get<T>(m_EnttID);
}

protected:
EntityRegistry* m_Registry = nullptr;
EntityID m_EnttID = NENTT;
};

}

```

The `Entity` class acts as a comprehensive wrapper for entities, encapsulating their identifiers and offering efficient functionalities for attaching, detaching, and checking components' existence. It includes constructors to initialize entities within an `EntityRegistry` and conversion operators to ensure entity validity. Notably, the virtual destructor allows for inheritance, enabling the creation of specialized entity types with custom functionalities.

Shader Class Update

Update your PBR shader by implementing the new functions present in the code snippet depicted in (Listing 5.22).

Listing 5.22: Graphics/Shaders/PBR.
h

```

#pragma once
#include "Shader.h"
#include "../Utilities/Data.h"

namespace EmPy
{
    struct PbrShader : Shader
    {
        EMPY_INLINE PbrShader(const std::string& filename) :
        Shader(filename)
        {
            u_Model = glGetUniformLocation(m_ShaderID,
"u_model");
            u_View = glGetUniformLocation(m_ShaderID,
"u_view");
            u_Proj = glGetUniformLocation(m_ShaderID,
"u_proj");
        }

        EMPY_INLINE void SetCamera(Camera3D& camera,
Transform3D& transform, float ratio)
        {
            glUniformMatrix4fv(u_Proj, 1, GL_FALSE,
glm::value_ptr(camera.Projection(ratio)));
            glUniformMatrix4fv(u_View, 1, GL_FALSE,
glm::value_ptr(camera.View(transform)));
        }

        EMPY_INLINE void Draw(Mesh3D& mesh, Transform3D&
transform)
        {
            glUniformMatrix4fv(u_Model, 1, GL_FALSE,
glm::value_ptr(transform.Matrix()));
            mesh->Draw(GL_TRIANGLES);
        }
    };
}

```

```

private:
    uint32_t u_Model = 0u;
    uint32_t u_View = 0u;
    uint32_t u_Proj = 0u;
};

}

```

The function `SetCamera()` helps set the "view" and "projection" matrices and the `Draw()` function for now simply sets the "model" matrix and calls the `Draw()` function of the mesh object.

Renderer Class Update

Be encouraged; we are almost there! Update your renderer class as shown in (Listing 5.23).

Listing 5.23: Graphics/Renderer.h

```

#pragma once
#include "Buffers/Frame.h"
#include "Shaders/PBR.h"
#include "Shaders/Final.h"

namespace EmPy
{
    struct GraphicsRenderer
    {
        EMPY_INLINE GraphicsRenderer(int32_t width, int32_t
height)

```

```

    {

        // initialize opengl
        if(glewInit() != GLEW_OK)
        {
            EMPLY_FATAL("failed to init glew!");
            exit(EXIT_FAILURE);
        }
        glewExperimental = GL_TRUE;

        m_Final = std::make_unique<FinalShader>
("Resources/Shaders/final.glsl");
        m_Pbr = std::make_unique<PbrShader>
("Resources/Shaders/pbr.glsl");
        m_Frame = std::make_unique<FrameBuffer>(width,
height);
    }

    EMPLY_INLINE void SetCamera(Camera3D& camera,
Transform3D& transform)
{
    m_Pbr->SetCamera(camera, transform, m_Frame-
>Ratio());
}

    EMPLY_INLINE void Draw(Mesh3D& mesh, Transform3D&
transform)
{
    m_Pbr->Draw(mesh, transform);
}

    EMPLY_INLINE void Resize(int32_t width, int32_t
height)
{
    m_Frame->Resize(width, height);
}

```

```

EMPTY_INLINE uint32_t GetFrame()
{
    return m_Frame->GetTexture();
}

EMPTY_INLINE void NewFrame()
{
    m_Frame->Begin();
    m_Pbr->Bind();
}

EMPTY_INLINE void EndFrame()
{
    m_Pbr->Unbind();
    m_Frame->End();
}

EMPTY_INLINE void ShowFrame()
{
    m_Final->Show(m_Frame->GetTexture());
}

private:
    std::unique_ptr<FrameBuffer> m_Frame;
    std::unique_ptr<FinalShader> m_Final;
    std::unique_ptr<PbrShader> m_Pbr;
};

}

```

The `GraphicsRenderer` structure creates unique instances of shaders (`PbrShader` and `FinalShader`) and a frame buffer to manage rendering processes. Methods within this structure facilitate rendering tasks: `SetCamera()` configures the camera for rendering,

utilizing the `PbrShader` and the frame buffer's aspect ratio. `Draw()` renders a provided 3D mesh with a specified transformation using the PBR shader. `Resize()` updates the dimensions of the frame buffer. `GetFrame()` retrieves the texture associated with the frame buffer for further processing or display. `BeginFrame()`, `EndFrame()`, and `ShowFrame()` manage the rendering pipeline, starting and finalizing frame processing while displaying the rendered frame using the `FinalShader`. The private members consist of unique pointers to manage the frame buffer and shaders, encapsulate essential rendering components, and ensure proper resource management.

5.3.3 Scene Abstraction

Let's integrate a scene into our engine's context, facilitating the addition or removal of entities. Proceed by incorporating the scene into your application context, following the example depicted in `ctxt-scene.h`.

Listing 5.24: Application/Context. h

```
#pragma once
#include "Window/Window.h"
#include "Auxiliaries/ECS.h"
#include "Graphics/Renderer.h"

namespace EmPy
```

```

{
    // forward declaration
    struct AppInterface;

    // application context
    struct ApplicationContext
    {
        EMPY_INLINE ApplicationContext()
        {
            Window = std::make_unique<AppWindow>(&Dispatcher,
1280, 720, "Empty Engine");
            Renderer = std::make_unique<GraphicsRenderer>
(1280, 720);
        }

        EMPY_INLINE ~ApplicationContext()
        {
            for(auto layer : Layers)
            {
                EMPY_DELETE(layer);
            }
        }

        std::unique_ptr<GraphicsRenderer> Renderer;
        std::vector<AppInterface*> Layers;
        std::unique_ptr<AppWindow> Window;
        EventDispatcher Dispatcher;
        EntityRegistry Scene;
    };
}

```

The scene is simply an entity registry from "EnTT". It is responsible for managing all entities and their corresponding components. While we could create a

wrapper around this registry to enhance its usability, we have opted to utilize the application interface directly for interaction. This choice aligns with our goal of reflecting these interactions across various layers, ensuring that the ability to add, remove, and modify entities remains integrated within the editor. This approach maintains consistency throughout the application, enabling seamless entity management across different components and layers of our application. You will notice the `GraphicsRenderer` we have added as well.

Scene Interface Functions

As previously highlighted, our aim is to enable every layer within the application to seamlessly interact with the scene, allowing for entity creation, destruction, and modification. Achieving this goal is straightforward by implementing dedicated member functions within the application interface class shown in (Listing 5.25).

Listing 5.25: Application/Interface. h

```
#pragma once
#include "Context.h"

namespace EmPy
{
    struct AppInterface
```

```

{
    /* ... same as before ... */

    // create entity
    template <typename Entt, typename... Args>
    EMPY_INLINE Entt CreateEntt(Args&&... args)
    {
        EMPY_STATIC_ASSERT(std::is_base_of<Entity,
Entt>::value);
        return std::move(Entt(&m_Context->Scene,
std::forward<Args>(args)...));
    }

    // convert id to entity
    template<typename Entt>
    EMPY_INLINE Entt ToEntt(EntityID entity)
    {
        EMPY_STATIC_ASSERT(std::is_base_of<Entity,
Entt>::value);
        return std::move(Entt(&m_Context->Scene,
entity));
    }

    // loop through entities
    template<typename Entt, typename Comp, typename Task>
    EMPY_INLINE void EnttView(Task&& task)
    {
        EMPY_STATIC_ASSERT(std::is_base_of<Entity,
Entt>::value);
        m_Context->Scene.view<Comp>().each([this, &task]
(auto entity, auto& comp)
{
    task(std::move(Entt(&m_Context->Scene,
entity)), comp);
});
    }
}

```

```
    /* ... same as before ... */  
}  
}
```

The member functions showcased utilize template arguments to offer versatility. For example, the function `CreateEntt()` accepts the entity class type as a template argument, facilitating the creation of custom entity types. On the other hand, `EnttView()` helps to iterate through all entities possessing the specified component type. The function `ToEntt` is self-explanatory, it simply converts a given entity identifier into an entity object defined by the template type provided.

5.3.4 Rendering First Frame

Now is the time to finally see the result of your hard work. Go ahead and update your "Application.h" header file as shown in ([Listing 5.26](#)). This will basically cover all that is left to render our first frame.

Listing 5.26: Application/Application.h

```
struct Application : AppInterface  
{  
    EMPTY_INLINE Application()  
    {  
        // initialize application context
```

```

m_LayerID =TypeID<Application>();
m_Context = new ApplicationContext();

// attach window resize event callback
AttachCallback<WindowResizeEvent>([this] (auto e)
{
    m_Context->Renderer->Resize(e.Width, e.Height);
});

EMPTY_INLINE ~Application()
{
    EMPTY_DELETE(m_Context);
}

EMPTY_INLINE void RunContext()
{
    // create scene camera
    auto camera = CreateEntt<Entity>();
    camera.Attach<TransformComponent>
().Transform.Translate.z = 2.0f;
    camera.Attach<CameraComponent>();

    // create quad entity
    auto quad = CreateEntt<Entity>();
    quad.Attach<MeshComponent>().Mesh = CreateQuad3D();
    quad.Attach<TransformComponent>();

    while(m_Context->Window->PollEvents())
    {
        // render new frame
        m_Context->Renderer->NewFrame();

        // set shader camera
        EnttView<Entity, CameraComponent>([this] (auto
entity, auto& comp)

```

```

    {
        auto& transform = entity.template
Get<TransformComponent>().Transform;
        m_Context->Renderer->SetCamera(comp.Camera,
transform);
    });

    // render models
    EnttView<Entity, MeshComponent>([this] (auto
entity, auto& comp)
{
    auto& transform = entity.template
Get<TransformComponent>().Transform;
    m_Context->Renderer->Draw(comp.Mesh,
transform);
});

m_Context->Renderer->EndFrame();

    // update layers
    for(auto layer : m_Context->Layers)
{
    layer->OnUpdate();
}

    // show frame to screen
    m_Context->Renderer->ShowFrame();
}
}
};


```

You will notice in the application's constructor that we have attached an event callback function for when a `WindowResizeEvent` is triggered. This helps update the

size of the frame buffer to always match the current size of the window. `RunContext()` as you know, defines the core engine loop. It starts by creating a scene camera entity and a quad entity used for rendering. The loop then proceeds to manage window events, update various layers within the engine context, and render a new frame continuously. Within the rendering phase, it sets the camera position and renders models present in the scene by iterating through entities possessing specific components: `CameraComponent` and `MeshComponent`. The camera's position is adjusted according to its transformation, and models are drawn based on their mesh and transformation data. Finally, the rendered frame is displayed on the screen, ensuring continuous update and rendering of the scene in response to events and updates within the engine context.

Final Touch

Add the following code ([Listing 5.27](#)) in the `"EmptyEngine/CMakeLists.txt"` to copy the `"Root/Resources"` folder into the build directory post compilation. Do not worry about the copy slowing down your compilation process, it only happens when the files within the folder are modified and it only copies the modified ones. This makes sure that the

source codes of our shaders are available to the executables at the time of execution.

Listing 5.27: EmpyEngine/CMakeLists.txt

```
# ... same as before

# copy resources
if(EXISTS ${CMAKE_SOURCE_DIR}/Resources)
    add_custom_command(TARGET ${PROJECT_NAME} POST_BUILD
COMMAND
    ${CMAKE_COMMAND} -E copy_directory
    ${CMAKE_SOURCE_DIR}/Resources
    ${EXECUTABLE_OUTPUT_PATH}/Resources
)
else()
    message(WARNING "[WARNING] no resource directory!")
endif()
```

With your **"Editor.cpp"** code looking like the one in (Listing 5.28), you can compile and run your application to see the magic happening live in front of your eyes.

Listing 5.28: EmpyEditor/src/Editor.cpp

```
#include <Empy.h>
using namespace Empy;

struct Editor : AppInterface
{
    EMPY_INLINE void OnUpdate() { }
```

```
EMPTY_INLINE void OnStart() { }

};

int32_t main(int32_t argc, char** argv)
{
    auto app = new Application();
    app->AttachLayer<Editor>();
    app->RunContext();
    return 0;
}
```

This will produce the result depicted in (Figure 5.22). Try resizing the window and see how the frame buffer and the viewport of the camera will also adjust themselves accordingly.

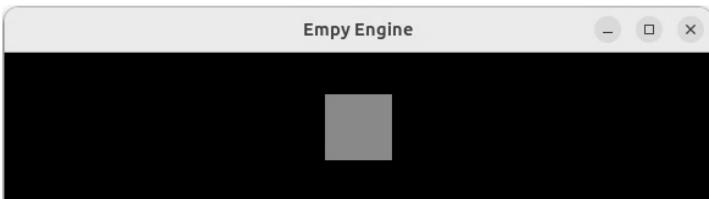


Figure 5.22: Renderer Output (Linux)

5.4 Loading 3D Models

Now that our renderer is able to produce some visible results, Let us load models from files and render them as well. To achieve that, we will take advantage of a well-known library called "**Assimp**".

5.4.1 Assimp Library

The **Assimp (Open Asset Import Library)** is a powerful open-source library designed to simplify the process of importing and handling various 3D model formats within applications. It serves as a bridge between different 3D model formats, providing a unified interface for accessing and processing data from these models.

Developed in C++, Assimp supports a wide range of 3D file formats used in the industry, including popular ones like OBJ, FBX, COLLADA, glTF, and many more. It offers functionalities to access mesh data, textures, materials, and scene hierarchy from 3D models. It abstracts the complexities involved in parsing different file formats, providing a consistent interface for accessing and manipulating model data. This simplifies tasks such as rendering, physics simulations, and other operations involving 3D models within applications.

The library is widely used in the game development, computer graphics, and visualization domains, offering a convenient solution for dealing with diverse 3D model formats, thereby saving developers valuable time and effort in handling file format intricacies.

Listing 5.29: Root/conanfile.tx
t

```
[requires]
glm/cci.20230113
opengl/system
spdlog/1.12.0
assimp/5.2.2
entt/3.12.2
glfw/3.3.8
glew/2.2.0

[generators]
cmake

[options]
assimp:shared=False
glew:shared=False
glfw:shared=False
```

Update your "**conanfile.txt**" as depicted in (Listing [5.29](#)) and extend your project's directory tree as shown in (Figure [5.23](#)).

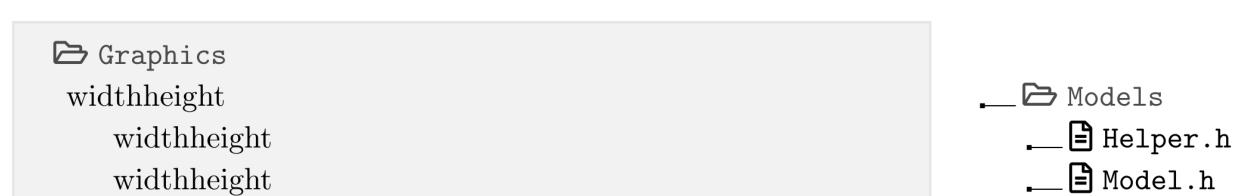


Figure 5.23: Models Directory

Assimp Helpers

The functions showcased in (Listing 5.30) play a pivotal role in converting specific Assimp data types into GLM data types. We need this because our internal Vertex type uses GLM to represent its attributes.

Listing 5.30: Graphics/Models/Helper.h

```
#pragma once
#include "../Buffers/Mesh.h"
#include <assimp/quaternion.h>
#include <assimp/matrix4x4.h>
#include <assimp/color4.h>

namespace EmPy
{
    EMPY_INLINE static glm::vec3 AssimpToVec3(const
aiVector3D& v)
    {
        return glm::vec3(v.x, v.y, v.z);
    }

    EMPY_INLINE static glm::vec4 AssimpToVec4(const
aiColor4D& c)
    {
        return glm::vec4(c.r, c.g, c.b, c.a);
    }

    EMPY_INLINE static glm::quat AssimpToQuat(const
aiQuaternion& q)
    {
        return glm::quat(q.w, q.x, q.y, q.z);
    }
}
```

```

    EMPY_INLINE static glm::mat4 AssimpToMat4(const
aiMatrix4x4& from)
{
    glm::mat4 to;
    //the a, b, c, d in assimp is the row; the 1, 2, 3,
4 is the column
    to[0][0] = from.a1; to[1][0] = from.a2; to[2][0] =
from.a3; to[3][0] = from.a4;
    to[0][1] = from.b1; to[1][1] = from.b2; to[2][1] =
from.b3; to[3][1] = from.b4;
    to[0][2] = from.c1; to[1][2] = from.c2; to[2][2] =
from.c3; to[3][2] = from.c4;
    to[0][3] = from.d1; to[1][3] = from.d2; to[2][3] =
from.d3; to[3][3] = from.d4;
    return to;
}
}

```

Make sure this code is added to your project before you proceed.

5.4.2 Model Class

The code showcased in ([Listing 5.31](#)) defines a structure named `Model`. The `Model` serves as a container to load 3D models using the Assimp library, providing methods to parse these models and draw them using OpenGL. The `Load()` function initializes an Assimp importer, reads the specified model file, applies various processing flags for optimization, and then parses the loaded scene's meshes.

Listing 5.31: Graphics/Models/Model.h

```
#pragma once

#include <assimp/postprocess.h>
#include <assimp/quaternion.h>
#include <assimp/Importer.hpp>
#include <assimp/scene.h>
#include "Helper.h"

namespace EmPy
{
    struct Model
    {
        EMPTY_INLINE Model() = default;

        EMPTY_INLINE Model(const std::string& filename)
        {
            Load(filename);
        }

        EMPTY_INLINE void Load(const std::string& filename)
        {
            Assimp::Importer importer;
            uint32_t flags = aiProcess_Triangulate |
aiProcess_GenSmoothNormals | aiProcess_CalcTangentSpace |
                aiProcess_OptimizeMeshes |
aiProcess_OptimizeGraph | aiProcess_ValidateDataStructure |
                    aiProcess_ImproveCacheLocality |
aiProcess_FixInfacingNormals |
                aiProcess_GenUVCoords | aiProcess_FlipUVs;

            const aiScene* ai_scene =
importer.ReadFile(filename, flags);
        }
    };
}
```

```

        if (!ai_scene || ai_scene->mFlags ==
AI_SCENE_FLAGS_INCOMPLETE || !ai_scene->m rootNode) {
            EMPTY_ERROR("failed to load model: {}",
importer.GetErrorString());
            return;
        }

        // parse all meshes
        ParseNode(ai_scene, ai_scene->m rootNode);
    }

EMPTY_INLINE void Draw(uint32_t mode)
{
    for(auto& mesh : m_Meshes)
    {
        mesh->Draw(mode);
    }
}

private:
    EMPTY_INLINE void ParseNode(const aiScene* ai_scene,
aiNode* ai_node)
    {
        for (uint32_t i = 0; i < ai_node->mNumMeshes;
i++)
        {
            ParseMesh(ai_scene->mMeshes[ai_node-
>mMeshes[i]]);
        }

        for (uint32_t i = 0; i < ai_node->mNumChildren;
i++)
        {
            ParseNode(ai_scene, ai_node->mChildren[i]);
        }
    }
}

```

```
EMPTY_INLINE void ParseMesh(aiMesh* ai_mesh)
{
    MeshData<ShadedVertex> data;

    // vertices
    for (uint32_t i = 0; i < ai_mesh->mNumVertices;
i++)
    {
        ShadedVertex vertex;

        // positions
        vertex.Position = AssimpToVec3(ai_mesh-
>mVertices[i]);

        // normals
        vertex.Normal = AssimpToVec3(ai_mesh-
>mNormals[i]);

        // texcoords
        vertex.UVs.x = ai_mesh->mTextureCoords[0]
[i].x;
        vertex.UVs.y = ai_mesh->mTextureCoords[0]
[i].y;

        // push to array
        data.Vertices.push_back(vertex);
    }

    // indices
    for (uint32_t i = 0; i < ai_mesh->mNumFaces;
i++)
    {
        for (uint32_t k = 0; k < ai_mesh-
>mFaces[i].mNumIndices; k++)
        {
```

```

        data.Indices.push_back(ai_mesh-
>mFaces[i].mIndices[k]);
    }
}

// create new mesh instance
auto mesh = std::make_unique<ShadedMesh>(data);
m_Meshes.push_back(std::move(mesh));
}

private:
    std::vector<Mesh3D> m_Meshes;
};

// 3d model typedef
using Model3D = std::shared_ptr<Model>;
}

```

The `Draw()` function iterates through the loaded meshes and draws them accordingly. The private functions, `ParseNode()` and `ParseMesh()`, recursively traverse the scene graph, extracting vertex data, normals, texture coordinates, and indices from each mesh. This extracted data is stored in a custom `MeshData` structure, which is used to create instances of `ShadedMesh`. Ultimately, these `ShadedMesh` instances are stored in the `m_Meshes` vector, encapsulating the loaded 3D model's geometry for rendering purposes. Additionally, a `Model3D` type definition is introduced for convenience, representing a shared pointer to a `Model` instance.

Model Component

Add the model component in "**ECS.h**" as shown in ([Listing 5.32](#))

Listing 5.32: Graphics/Models/Helper.h

```
/* ... same as before ... */

struct ModelComponent
{
    EMPY_INLINE ModelComponent(const ModelComponent&) = default;
    EMPY_INLINE ModelComponent() = default;
    Model3D Model;
};

/* ... same as before ... */
```

PBR Shader Class Update

Go ahead and add the following function to the "**PBR.h**" header file so that it can render models as we did with meshes.

Listing 5.33: Graphics/Shaders/PBR.h

```
/* ... same as before ... */

EMPY_INLINE void Draw(Model3D& model, Transform3D& transform)
```

```
{  
    glUniformMatrix4fv(u_Model, 1, GL_FALSE,  
glm::value_ptr(transform.Matrix()));  
    model->Draw(GL_TRIANGLES);  
}  
  
/* ... same as before ... */
```

Renderer Class Update

Go ahead and add the following function to the **"Renderer.h"** header so that it can render models use the PBR shader `Draw()` function we implemented above.

Listing 5.34: Graphics/Renderer.h

```
/* ... same as before ... */  
  
EMPY_INLINE void Draw(Model3D& model, Transform3D& transform)  
{  
    m_Pbr->Draw(model, transform);  
}  
  
/* ... same as before ... */
```

Rendering Models

Open your **"Application.h"** header file and add a new entity with a `ModelComponent` by updating the

`RunContext()` function to render all entities with a `ModelComponent`. See ([Listing 5.35](#)).

Listing 5.35: Application/Application. h

```
/* ... same as before ... */

EMPY_INLINE void RunContext()
{
    // create scene camera
    auto camera = CreateEntt<Entity>();
    camera.Attach<TransformComponent>().Transform.Translate.z
= 2.0f;
    camera.Attach<CameraComponent>();

    // create cube entity
    auto model = std::make_shared<Model>
("Resources/Models/cube.fbx");
    auto cube = CreateEntt<Entity>();
    cube.Attach<TransformComponent>().Transform.Rotation.y =
30.0f;
    cube.Attach<ModelComponent>().Model = model;

    while (m_Context->Window->PollEvents())
    {
        // start frame
        m_Context->Renderer->NewFrame();

        // set shader camera
        EnttView<Entity, CameraComponent>([this] (auto
entity, auto& comp)
        {
            auto& transform = entity.template
Get<TransformComponent>().Transform;
```

```

        m_Context->Renderer->SetCamera(comp.Camera,
transform);
    });

    // render models
    EnttView<Entity, ModelComponent>([this] (auto entity,
auto& comp)
{
    auto& transform = entity.template
Get<TransformComponent>().Transform;
    m_Context->Renderer->Draw(comp.Model, transform);
});

    // end frame
    m_Context->Renderer->EndFrame();

    // update layers
    for(auto layer : m_Context->Layers)
{
    layer->OnUpdate();
}

    // show frame to screen
    m_Context->Renderer->ShowFrame();
}

/*
 ... same as before ...
*/

```

You will notice that the model is a cube loaded from the "Resources" folder and is an FBX file. The model loaded in this code is present in the GitHub repository. But you can also download additional models online (Google). Link: <https://free3d.com/3d-models/fbx>.

Compiling and running this code will produce the following result:

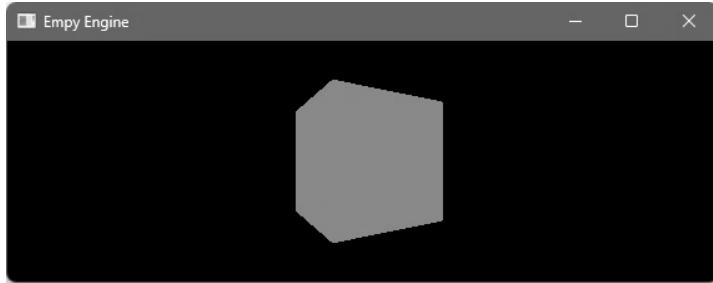


Figure 5.24: Rendering Loaded Model

Keep in mind that Conan must first download and compile Assimp. So, it will take some time to compile the project.

You might have noticed that our model looks like a flat colored object. Well, let us fix that by adding some lighting to the scene in the next chapter. We are now going to jump into advanced rendering techniques.

Code not working? clone the branch "basic-rendering" on the GitHub repository and grant execution rights to the "EmpyLinux.sh" script on Linux. <https://github.com/Madsycode/book-empy-engine>.

```
git clone -branch <branch-name> <repository-link>
```

6 Advanced Rendering

Diligent hands will rule, but laziness ends in forced labor. (King Solomon)

In the world of computer graphics, lighting computations rely on simplified models that approximate real-world lighting phenomena while being computationally feasible. Real-world lighting is a vastly intricate process, influenced by numerous factors beyond our computational capabilities. Rooted in our understanding of light physics, these lighting models aim to emulate reality by working around the limitations of computational power.

6.1 Phong Reflection Model

One simple and well establish approach is the Phong [Wikipedia \[b\]](#) reflection model. It was developed by Bui Tuong Phong at the University of Utah, who published it in his 1975 Ph.D. dissertation. It was published in conjunction with a method for interpolating the calculation for each individual pixel that is rasterized from a polygonal surface model; the interpolation technique is known as Phong shading, even when it is used with a reflection model other than Phong's. Phong's methods were considered radical at the time of their introduction but have since become

the de facto baseline shading method for many rendering applications. Phong's methods have proven popular due to their generally efficient use of computation time per rendered pixel.

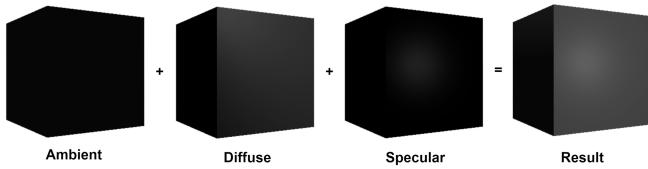


Figure 6.1: Phong Lighting

Phong approximates lighting effects using simpler mathematical equations that involve ambient, diffuse, and specular components (Figure 6.1). It doesn't fully account for real-world physics but has been widely used due to its simplicity and efficiency in rendering.

6.1.1 Ambient Component

The ambient component represents the uniform illumination present in a scene, regardless of light sources. Set the base color of objects when no direct light hits them. The PBR ambient light formula can be represented using the following equation.

$$Ambient = K_a * I_a * L_r$$

- ☞ **Ka** (Ambient Reflection Coefficient): This coefficient defines how much of the ambient light contributes to the object's color. Higher values of Ka result in a more pronounced influence of ambient light on the object's appearance.
- ☞ **Ia** (Ambient Light Intensity): It represents the strength or intensity of the ambient light source. Higher Ia values mean brighter ambient light, affecting the overall brightness of the scene.
- ☞ **Lr** (Ambient Light Radiance): It represents the color or the radiance of the ambient light source.

6.1.2 Diffuse Component

Mimicking the scattered reflection of light on surfaces, diffuse lighting captures how light interacts with surfaces equally in all directions, creating the appearance of a matte surface. The PBR diffuse light formula calculates the diffuse light contribution using the following equation:

$$\text{Diffuse} = K_d * I_d * \cos(\theta) * L_r$$

- ☞ **Kd** (Diffuse Reflection): It characterizes how light scatters or diffuses on a surface, creating a softer, matte appearance. This component accounts for light reflecting equally in all directions from a surface.

☞ **Kd** (Diffuse Light Intensity): This coefficient controls the influence of diffuse light on the object's color. Higher values of Kd result in a more significant contribution of diffuse light to the object's appearance.

☞ θ (Angle between Surface Normal and Light Direction): The cosine term measures the angle between the surface normal (a vector perpendicular to the surface) and the direction from the surface to the light source. It quantifies how much of the light is directly hitting the surface, affecting the intensity of the diffuse reflection. See (Figure 6.2).

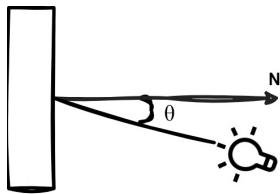


Figure 6.2: Diffusion Angle

6.1.3 Specular Component

Emulating the shiny highlights caused by light reflecting off surfaces, specular lighting focuses on the directional reflection of light, typically seen on glossy or reflective surfaces. The PBR specular light formula

calculates the specular light contribution using the following equation:

$$\text{Specular} = K_s * I_s * \cos(\theta)^n * L_r$$

☞ **K_s (Specular Reflection Coefficient):** Controls the strength of the specular reflection on the object's appearance. Higher values of 'K_s' result in more pronounced and intense specular highlights.

☞ **I_s (Specular Light Intensity):** Represents the strength or intensity of the specular light source. Higher *I_s* values mean a brighter specular light source, affecting the overall brightness and intensity of specular highlights.

☞ **θ (Angle between Reflected Light and Viewer):** Measures the angle between the reflected light direction and the direction towards the viewer or camera. It determines how much of the specular highlight is visible to the viewer based on their viewing angle. See ([Figure 6.3](#))

☞ **n (Specular Exponent):** The specular exponent controls the focus and sharpness of the specular highlight. Higher values of *n* create a more focused and concentrated highlight.

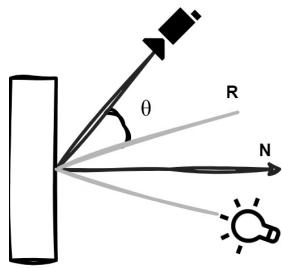


Figure 6.3: Specular Parameters

While Phong shading offers simplicity, its limitations fall short of our engine's aspirations for realism. Our aim is to approach reality as closely as possible, which is why we've opted for a PBR approach.

6.2 Physically Based Rendering

PBR [Wikipedia \[a\]](#) is a rendering approach that strives to simulate the behavior of light and materials in a physically accurate manner, yielding more realistic and consistent results. At its core, PBR relies on a few fundamental principles and equations to model how light interacts with surfaces.

Unlike Phong, PBR aims for greater realism by adhering more closely to physical principles governing how light interacts with materials. PBR incorporates complex models such as the rendering equation and microfacet theory, considering intricate details like surface roughness, energy conservation, and Fresnel

effects. By accurately modeling how light bounces off surfaces, PBR can produce more lifelike and consistent results across various lighting conditions and material types.

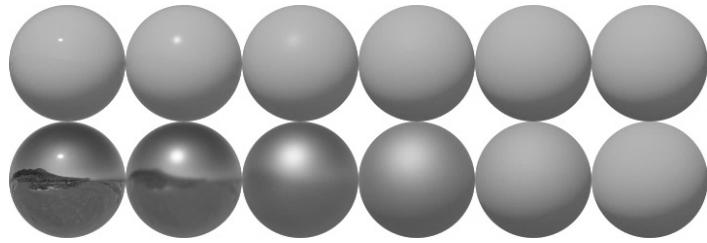


Figure 6.4: PBR Materials
Systèmes

The rendering equation, a cornerstone of PBR, describes the outgoing radiance L_o from a point on a surface in a particular direction as the sum of emitted radiance L_e and reflected radiance from incoming light:

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

Here: - L_o is the outgoing radiance. - L_e represents emitted radiance. - f_r is the Bidirectional Reflectance Distribution Function (BRDF) defining how light scatters at a point based on incoming and outgoing angles. - L_i denotes incoming radiance. - ω_i and ω_o are

incoming and outgoing light directions. - \mathbf{n} is the surface normal.

The BRDF f_r encapsulates the surface's reflective properties and plays a pivotal role in PBR. It often comprises multiple components such as Lambertian diffuse [cratch a Pixel](#) reflection and Cook-Torrance [Compendium](#) microfacet specular reflection models:

$$f_r(\mathbf{x}, \omega_i, \omega_o) = k_d \cdot \frac{c}{\pi} + k_s \cdot \frac{D(\mathbf{h}) \cdot F(\mathbf{v}, \mathbf{h}) \cdot G(\mathbf{v}, \mathbf{l})}{4 \cdot (\mathbf{n} \cdot \mathbf{l}) \cdot (\mathbf{n} \cdot \mathbf{v})}$$

Here: - k_d represents the diffuse reflectance coefficient. - k_s denotes the specular reflectance coefficient. - $D(\mathbf{h})$ is the microfacet distribution function. - $F(\mathbf{v}, \mathbf{h})$ represents the Fresnel term. - $G(\mathbf{v}, \mathbf{l})$ is the geometric attenuation factor. - \mathbf{v} is the view direction, \mathbf{l} is the light direction, and \mathbf{h} is the half-vector.

Understanding these equations is not mandatory; they simply offer technical insights into the principles behind PBR.

PBR's essence lies in accurately modeling these equations and principles, leveraging physically plausible materials and lighting interactions to produce

more lifelike and believable renderings in computer graphics.

6.2.1 Basic Shader

The code in (Listing 6.1) is the updated version of our PBR shader. This simplified version demonstrates a basic PBR model. It computes diffuse and specular lighting components based on material properties such as "albedo" (base color), "metallic" factor, and "roughness".

Listing 6.1: Resources/Shaders/pbr.gls

```
#version 330 core
layout (location = 0) in vec3 a_position;
layout (location = 1) in vec3 a_normal;
layout (location = 2) in vec2 a_uvs;

out Vertex
{
    vec3 Position;
    vec3 Normal;
} vertex;

uniform mat4 u_model;
uniform mat4 u_proj;
uniform mat4 u_view;

void main()
{
    vertex.Normal = mat3(u_model) * a_normal;
```

```
vertex.Position = vec3(u_model * vec4(a_position, 1.0));
gl_Position = u_proj * u_view * u_model * vec4(a_position,
1.0);
}

++VERTEX++

#version 330 core
layout (location = 0) out vec4 out_fragment;

// input vertex
in Vertex
{
    vec3 Position;
    vec3 Normal;
} vertex;

void main()
{
    vec3 lightPos = vec3(-0.5, 0.0, 1.0); // light Position
    vec3 lightRad = vec3(1.0, 1.0, 1.0); // light radiance
    vec3 viewPos = vec3(0.0, 0.0, 2.0); // camera position

    // Material properties (you can customize these)
    vec3 albedo = vec3(0.3, 0.8, 0.8); // base color
    float roughness = 0.5; // roughness factor (0-
1)
    float metallic = 0.2; // metallic factor (0-
1)

    // Diffuse (Lambertian) lighting
    vec3 N = normalize(vertex.Normal);
    vec3 lightDir = normalize(lightPos - vertex.Position);
    float diff = max(dot(N, lightDir), 0.0);
    vec3 diffuse = diff * lightRad;
```

```

// Dpecular (Cook-Torrance) lighting
vec3 viewDir = normalize(viewPos - vertex.Position);
vec3 halfwayDir = normalize(lightDir + viewDir);
float NdotH = max(dot(N, halfwayDir), 0.0);
float roughnessSq = roughness * roughness;
float denom = (NdotH * NdotH) * (roughnessSq - 1.0) +
1.0;
float D = roughnessSq / (3.1415 * denom * denom);
float kS = metallic;
vec3 specular = lightRad * (kS * D);

// Final color calculation
vec3 color = (diffuse + specular) * albedo;

out_fragment = vec4(color, 1.0);
}
++FRAGMENT++

```

As you can see in the shader code, we have defined a structure named *Vertex* in both the vertex and fragment shaders, prefaced with the 'out' and 'in' keywords, respectively. This signifies our intent to pass its information from the vertex shader to the fragment shader for further processing. The *Vertex* type presently includes the vertex's position and surface normal.

This example doesn't cover all aspects of PBR (such as Fresnel effects, multiple light sources, environment maps, etc.), but it provides a starting point for understanding the basic principles of PBR shading.

This shader produces the result depicted in (Figure 6.5).

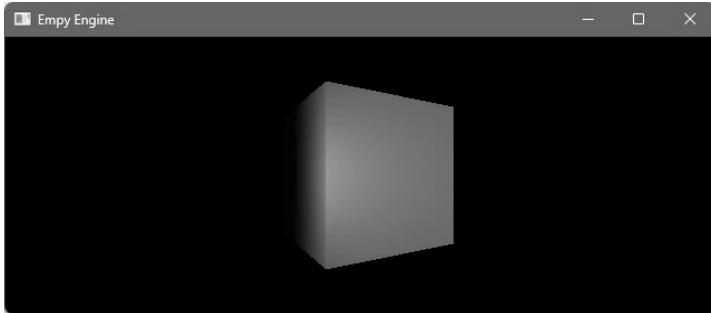


Figure 6.5: Basic PBR Rendering

6.2.2 Basic Material

You may have noticed that several crucial parameters defining the material of the rendered objects are hard-coded. This results in a static rendering pipeline, limiting our ability to render diverse objects with varied looks. To address this, we need to be able to create and control these parameters on the CPU side using GLSL-uniforms.

Materials in computer graphic are fundamental elements defining how surfaces appear when illuminated and viewed in a virtual environment. They encapsulate visual properties like color, reflectivity, transparency, and how surfaces interact with light.

Each material type defines how light interacts with the surface, affecting its visual characteristics.

In PBR, a material is defined by several components that collectively govern how light interacts with a surface.

☞ **Albedo (ρ):** Represents the base color of the material, determining its intrinsic color when lit by diffuse light. Typically denoted by the symbol ρ , it influences the perceived color of the surface under uniform lighting conditions. For instance, in the shader, 'albedo' is the RGB color of the material without any lighting effects applied.

☞ **Metallic (m):** Defines how metallic or non-metallic the material appears. A metallic surface exhibits strong reflections resembling metals like copper or iron, while non-metallic surfaces tend to have diffuse reflections like plastic or wood. Often a value between 0 and 1, where 0 represents a non-metallic (dielectric) material and 1 represents a fully metallic surface. In the shader, 'metallic' determines the balance between specular and diffuse reflections.

☞ **Roughness (a):** Represents the micro-surface irregularities or smoothness of the material. A lower roughness value results in a smoother surface with more focused and sharp reflections, akin to polished

surfaces. Conversely, higher roughness leads to scattered and diffused reflections, resembling rough or matte surfaces. Usually represented by a value between 0 (perfectly smooth) and 1 (very rough). In the shader, 'roughness' governs the blurriness or sharpness of specular reflections.

Adjusting these parameters allows for the creation of diverse and convincing material appearances, mimicking various real-world surfaces and lighting conditions. (Figure [6.4](#)).

Material Class

Implement the material class as depicted in (Listing [6.2](#)).

Listing 6.2: Graphics/Utilities/Data.
h

```
struct PbrMaterial
{
    glm::vec3 Albedo = glm::vec3(0.3f, 0.8f, 0.8f);
    float Roughness = 0.5f;
    float Metallic = 0.2f;
};
```

The *PbrMaterial* class straightforwardly encapsulates the previously introduced parameters.

PBR Shader Update

Before rendering a mesh in the frame buffer, its material must be sent to the shader from CPU. Additionally, equivalent material uniforms must be defined on the GPU side to receive these values. The code in (Listing 6.3) demonstrates the updated version of the PBR shader.

Listing 6.3: Resources/Shaders/pbr.gls

```
#version 330 core
layout (location = 0) in vec3 a_position;
layout (location = 1) in vec3 a_normal;
layout (location = 2) in vec2 aUvs;

out Vertex
{
    vec3 Position;
    vec3 Normal;
} vertex;

uniform mat4 u_model;
uniform mat4 u_proj;
uniform mat4 u_view;

void main()
{
    vertex.Normal = mat3(u_model) * a_normal;
    vertex.Position = vec3(u_model * vec4(a_position, 1.0));
    gl_Position = u_proj * u_view * u_model * vec4(a_position,
1.0);
```

```
}

++VERTEX++

#version 330 core
layout (location = 0) out vec4 out_fragment;

// input vertex
in Vertex
{
    vec3 Position;
    vec3 Normal;
} vertex;

// material type
struct Material
{
    float Roughness;
    float Metallic;
    vec3 Albedo;
};

// material uniforms
uniform Material u_material;

void main()
{
    vec3 lightPos = vec3(-0.5, 0.0, 1.0); // light Position
    vec3 lightRad = vec3(1.0, 1.0, 1.0); // light radiance
    vec3 viewPos = vec3(0.0, 0.0, 2.0); // camera position

    // diffuse (Lambertian) lighting
    vec3 N = normalize(vertex.Normal);
    vec3 lightDir = normalize(lightPos - vertex.Position);
    float diff = max(dot(N, lightDir), 0.0);
    vec3 diffuse = diff * lightRad * u_material.Albedo;
```

```

// specular (Cook-Torrance) lighting
vec3 viewDir = normalize(viewPos - vertex.Position);
vec3 halfwayDir = normalize(lightDir + viewDir);
float NdotH = max(dot(N, halfwayDir), 0.0);
float roughnessSq = u_material.Roughness *
u_material.Roughness;
float denom = (NdotH * NdotH) * (roughnessSq - 1.0) +
1.0;
float D = roughnessSq / (3.1415 * denom * denom);
vec3 specular = lightRad * (u_material.Metallic * D);

// final color calculation
out_fragment = vec4((diffuse + specular), 1.0);
}

++FRAGMENT++

```

The shader defines a structure similar to the one declared in C++. This structure is then used to create a uniform called `u_material`. In the remaining fragment shader code, we simply replace the "albedo", "roughness" and "metallic" accordingly.

Shader Class Update

We now need to update our C++ shader so that it can set the parameters of the material uniform while rendering the model. The code in ([Listing 6.4](#)) provides insights into how this is done. As you can see, we have added new member variables to store the location of all parameters found in the shader's material. If you

look closely at the `Draw()` function, you will see that it takes the material as extra argument to render the mesh.

Listing 6.4: Graphics/Shaders/PBR.
h

```
#pragma once
#include "Shader.h"
#include "../Utilities/Data.h"

namespace EmPy
{
    struct PbrShader : Shader
    {
        EMPY_INLINE PbrShader(const std::string& filename) :
        Shader(filename)
        {
            u_Roughness = glGetUniformLocation(m_ShaderID,
"u_material.Roughness");
            u_Metallic = glGetUniformLocation(m_ShaderID,
"u_material.Metallic");
            u_Albedo = glGetUniformLocation(m_ShaderID,
"u_material.Albedo");

            u_Model = glGetUniformLocation(m_ShaderID,
"u_model");
            u_View = glGetUniformLocation(m_ShaderID,
"u_view");
            u_Proj = glGetUniformLocation(m_ShaderID,
"u_proj");
        }

        EMPY_INLINE void SetCamera(Camera3D& camera,
Transform3D& transform, float ratio)
```

```

    {

        glUniformMatrix4fv(u_Proj, 1, GL_FALSE,
glm::value_ptr(camera.Projection(ratio)));
        glUniformMatrix4fv(u_View, 1, GL_FALSE,
glm::value_ptr(camera.View(transform)));
    }

    EMPTY_INLINE void Draw(Model3D& model, PbrMaterial&
material, Transform3D& transform)
{
    glUniformMatrix4fv(u_Model, 1, GL_FALSE,
glm::value_ptr(transform.Matrix()));
    glUniform3fv(u_Albedo, 1, &material.Albedo.x);
    glUniform1f(u_Roughness, material.Roughness);
    glUniform1f(u_Metallic, material.Metallic);
    model->Draw(GL_TRIANGLES);
}

private:
    uint32_t u_Roughness = 0u;
    uint32_t u_Metallic = 0u;
    uint32_t u_Albedo = 0u;

    uint32_t u_Model = 0u;
    uint32_t u_View = 0u;
    uint32_t u_Proj = 0u;
};

}

```

Renderer Class Update

We are also required to update the renderer to make sure that the `Draw()` function takes the material as

argument as well. See ([Listing 6.5](#)).

Listing 6.5: Graphics/Renderer.h

```
EMPTY_INLINE void Draw(Model3D& model, PbrMaterial& material,
Transform3D& transform)
{
    m_Pbr->Draw(model, material, transform);
}
```

Application Class Update

Now, open you "**Application.h**" and ensure proper rendering call as provided in ([Listing 6.6](#)).

Listing 6.6: Application/Application.h

```
// render with the material as argument
EnttView<Entity, ModelComponent>([this] (auto entity, auto&
comp)
{
    auto& transform = entity.template Get<TransformComponent>()
        .Transform;
    m_Context->Renderer->Draw(comp.Model, comp.Material,
        transform);
});
```

Compiling and running this version of the code will produce a similar result to that in ([Figure 6.5](#)).

Camera Position Uniform

If you look closely at the fragment part of the PBR shader, you will see that the camera view position (`viewPos`) is still hard-coded. We want to be able to define its value from the entity's transform component. Let us define a uniform in the shader to capture its value. You can see this in the updated version of the PBR shader code depicted in (Listing 6.7).

Listing 6.7: Resources/Shaders/pbr.gls

```
/* ... same as before ... */

// uniforms
uniform Material u_material;
uniform vec3 u_viewPos; // ----- here the uniform

void main()
{
    vec3 lightPos = vec3(-0.5, 0.0, 1.0); // light Position
    vec3 lightRad = vec3(1.0, 1.0, 1.0); // light radiance

    // Diffuse (Lambertian) lighting
    vec3 N = normalize(vertex.Normal);
    vec3 lightDir = normalize(lightPos - vertex.Position);
    float diff = max(dot(N, lightDir), 0.0);
    vec3 diffuse = diff * lightRad * u_material.Albedo;

    // Dpecular (Cook-Torrance) lighting
    vec3 viewDir = normalize(u_viewPos - vertex.Position); // 
<-- changed
```

```

vec3 halfwayDir = normalize(lightDir + viewDir);
float NdotH = max(dot(N, halfwayDir), 0.0);
float roughnessSq = u_material.Roughness *
u_material.Roughness;
float denom = (NdotH * NdotH) * (roughnessSq - 1.0) +
1.0;
float D = roughnessSq / (3.1415 * denom * denom);
vec3 specular = lightRad * (u_material.Metallic * D);

// Final color calculation
out_fragment = vec4((diffuse + specular), 1.0);
}

++FRAGMENT++

```

Accordingly, we will also update the C++ code of the PBR shader to accommodate the change.

See ([Listing 6.8](#))

Listing 6.8: Graphics/Shaders/PBR. h

```

#pragma once
#include "Shader.h"
#include "../Utilities/Data.h"

namespace EmPy
{
    struct PbrShader : Shader
    {
        EMPY_INLINE PbrShader(const std::string& filename) :
        Shader(filename)
        {
            u_ViewPos = glGetUniformLocation(m_ShaderID,
            "u_viewPos");
        }
    };
}

```

```

        u_Roughness = glGetUniformLocation(m_ShaderID,
"u_material.Roughness");
        u_Metallic = glGetUniformLocation(m_ShaderID,
"u_material.Metallic");
        u_Albedo = glGetUniformLocation(m_ShaderID,
"u_material.Albedo");

        u_Model = glGetUniformLocation(m_ShaderID,
"u_model");
        u_View = glGetUniformLocation(m_ShaderID,
"u_view");
        u_Proj = glGetUniformLocation(m_ShaderID,
"u_proj");
    }

    EMPLY_INLINE void SetCamera(Camera3D& camera,
Transform3D& transform, float ratio)
{
    glUniformMatrix4fv(u_Proj, 1, GL_FALSE,
glm::value_ptr(camera.Projection(ratio)));
    glUniformMatrix4fv(u_View, 1, GL_FALSE,
glm::value_ptr(camera.View(transform)));
    glUniform3fv(u_ViewPos, 1,
&transform.Translate.x); // <-- uniform set here
}

/* ... same as before ... */

private:
    uint32_t u_Roughness = 0u;
    uint32_t u_Metallic = 0u;
    uint32_t u_Albedo = 0u;

    uint32_t u_ViewPos = 0u;
    uint32_t u_Model = 0u;
    uint32_t u_View = 0u;

```

```
    uint32_t u_Proj = 0u;
};

}
```

To be able to see how this is working, you can make the camera rotate every frame by just increasing its y-coordinate in the application `RunContext()` function. See ([Listing 6.9](#)).

Listing 6.9: Application/Application.h

```
EnttView<Entity, CameraComponent>([this] (auto entity, auto&
comp)
{
    auto& transform = entity.template Get<TransformComponent>()
        .Transform;
    m_Context->Renderer->SetCamera(comp.Camera, transform);
    transform.Rotation.y += 0.1f; // ----- here the rotation
});
```

This should cause the camera to slowly rotate and present the scene from different angles.

6.2.3 PBR Microfacets

Before we proceed to integrate light sources to our scene, we first want to introduce some important concept of PBR. these concepts help capture the microfacet of the model to produce realistic visual effects. We are going to introduce these concepts and

subsequently include them in our shader to extend its capability.

Fresnel Equation

The Fresnel effect is a critical component in PBR that describes how light reflects off surfaces based on the viewing angle and the material's properties. It determines the ratio of reflected light to transmitted light at the interface between two mediums, such as air and a material.

The Fresnel effect plays a crucial role in understanding how much light is reflected off a surface relative to the viewing angle. This effect varies based on the material's properties, particularly its reflectivity at different angles of incidence.

The Fresnel equation is often approximated using Schlick's approximation, which provides a simplified yet accurate way to calculate the Fresnel reflectance. Schlick's approximation models the Fresnel effect using a blend between the material's base color and its reflectivity at glancing angles. The equation for Fresnel-Schlick approximation is:

$$F(\cos(\theta)) = F_0 + (1 - F_0)(1 - \cos(\theta))^5$$

Here: - $R(\cos(\theta))$ represents the Fresnel reflectance at the given angle. - F_0 denotes the material's base (or Fresnel) reflectivity at normal incidence. - $\cos(\theta)$ signifies the cosine of the angle between the incident light and the surface normal.

Key points about the Fresnel equation:

- ☞ **Angle Dependency:** Fresnel reflectance varies with the viewing angle. At normal incidence ($\theta = 0^\circ$), the material reflects light according to its base reflectivity (F_0), and as the angle increases, the reflectance increases.
- ☞ **Material Properties:** Different materials exhibit varying Fresnel reflectance behaviors. Metals typically have a higher reflectivity at grazing angles compared to dielectrics.
- ☞ **Impact on Appearance:** The Fresnel effect influences the appearance of surfaces. For instance, it contributes to the glossy reflections seen on smooth surfaces and affects the perceived material properties.

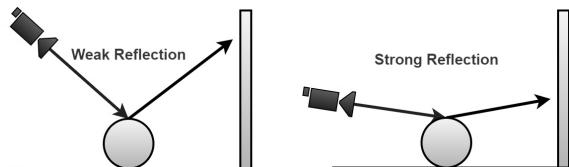


Figure 6.6: Fresnel Reflection Effect

(Figure [6.6](#)) provides a visual representation of the principle, and the code depicted in (Listing [6.10](#)) is the GLSL implementation. You can find a much more detailed explanation of this concept here [Dorian](#).

Listing 6.10: Fresnel Function

```
vec3 FresnelSchlick(float cosTheta, vec3 F0)
{
    return F0 + (1.0 - F0) * pow(1.0 - cosTheta, 5.0);
}
```

Smith, Schlick GGX

The Geometry Smith and Schlick functions are crucial components used in the Cook-Torrance shading model to compute specular reflections accurately. Here are their formulas:

- ☞ **Smith GGX:** Computes the visibility term that accounts for self-shadowing and masking effects due to microfacet distribution.

$$G_1 = \frac{2N_{\text{dot}V}}{N_{\text{dot}V} + \sqrt{r^2 + (1 - r^2) \cdot (N_{\text{dot}V}^2)}}$$

$$G_2 = \frac{2N_{\text{dot}L}}{N_{\text{dot}L} + \sqrt{r^2 + (1 - r^2) \cdot (N_{\text{dot}L}^2)}}$$

$$G_{\text{smith-GGX}} = G_1 \cdot G_2$$

Where $N_{\text{dot}V}$ and $N_{\text{dot}L}$ are the dot products of the surface normal with the view and light directions, respectively. r represents the surface roughness parameter.

☞ **Schlick GGX:** This function approximates the Fresnel factor for the microfacet distribution model, governing the material's reflectivity based on the viewing angle.

$$F_{\text{Schlick-GGX}}(\cos(\theta), r) = \frac{\cos(\theta)}{\cos(\theta) \cdot (1 - r) + r}$$

Where: $\cos(\theta)$ is the cosine of the angle between the view direction and the microfacet normal. r represents the surface roughness parameter.

These functions are essential in PBR shaders to accurately model the behavior of specular reflections on surfaces with varying roughness. They allow for precise calculations of visibility and Fresnel effects, contributing to the realism of rendered materials by accounting for microfacet distribution and reflectivity.

characteristics. (Listing 6.11) provides an implementation of these equations in GLSL. You can find extended explanations of these concepts here: [JoeyDeVries HABLE Joe](#).

Listing 6.11: Smith and Schlick GGX

```
float GeometrySchlickGGX(float NdotV, float roughness)
{
    float r = (roughness + 1.0);
    float k = (r * r) / 8.0;
    float num = NdotV;
    float denom = NdotV * (1.0 - k) + k;
    return num / denom;
}

float GeometrySmithGGX(float NdotV, float NdotL, float
roughness)
{
    float ggx1 = GeometrySchlickGGX(NdotV, roughness);
    float ggx2 = GeometrySchlickGGX(NdotL, roughness);
    return ggx1 * ggx2;
}
```

Distribution GGX

The Normal Distribution function, often referred to as the Distribution GGX (Trowbridge-Reitz), describes the distribution of microfacets over a surface, influencing the probability of various surface orientations for microfacets.

The DistributionGGX computes the probability distribution of surface normals for microfacets based on the surface roughness:

$$D_{\text{GGX}}(N, H, r) = \frac{r^2}{\pi \cdot [(N \cdot H)^2 \cdot (r^2 - 1) + 1]^2}$$

Where N is the surface normal, H is the half-vector, halfway between the view direction and the light direction. r is the surface roughness parameter. The formula generates a distribution curve that characterizes the likelihood of microfacet orientations. At higher roughness values, the distribution spreads, resulting in a broader range of surface orientations, whereas lower roughness values concentrate the orientations toward the normal direction. (Listing 6.12) shows the implementation of the Normal Distribution GGX. Learn more about it here: [JoeyDeVries](#).

Listing 6.12: Distribution
GGX

```
float DistributionGGX(vec3 N, vec3 H, float roughness)
{
    float a = roughness * roughness;
    float a2 = a * a;
    float NdotH = max(dot(N, H), 0.0);
    float NdotH2 = NdotH * NdotH;
    float num = a2;
    float denom = (NdotH2 * (a2 - 1.0) + 1.0);
    denom = PI * denom * denom;
```

```
    return num / denom;  
}
```

These functions will help us to better organize our PBR shader source code, and produce better visual results when combined with lighting.

6.3 Scene Illumination

So far, the lighting is quite static, as there is only one source and type of light. In computer graphics, various types of lighting play fundamental roles in shaping the visual appearance of 3D scenes. Each type of light brings unique characteristics and functionalities to rendering. Point lights add depth and ambiance, spotlights offer focused illumination, and directional lights provide consistent and widespread lighting, collectively contributing to the creation of realistic and visually compelling 3D environments.

6.3.1 Point Lights

Point lights emit light uniformly in all directions from a single point in space. They simulate sources like light bulbs and create smooth illumination, radiating outward in spherical patterns.

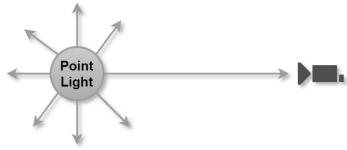


Figure 6.7: Point Light Source

Point Light class

Our shader so far has been using a point light as a light source. We now want to organize that to enable our engine to create a light source on the CPU and send their properties over to the GPU.

Listing 6.13: Graphics/Utilities/Data.h

```
struct PointLight
{
    EMPY_INLINE PointLight(const PointLight&) = default;
    EMPY_INLINE PointLight() = default;

    glm::vec3 Radiance = glm::vec3(1.0f);
    float Intensity = 1.0f;
};
```

Add this definition of the point light (Listing [6.13](#)) to your project.

Point Light Component

Since a point light is an entity, we are required to also create a component to house the data of the point light source. See (Listing [6.14](#)).

Listing 6.14: Auxiliaries/ECS. h

```
struct PointLightComponent
{
    EMPLY_INLINE PointLightComponent(const
PointLightComponent&) = default;
    EMPLY_INLINE PointLightComponent() = default;
    PointLight Light;
};
```

PBR Shader Update

We now need to also prepare the PBR shader to receive the properties of all point lights that must contribute to the final result of the rendering process. See (Listing [6.15](#)).

Listing 6.15: Resources/Shaders/pbr.gls l

```
#version 330 core
layout (location = 0) out vec4 out_fragment;

// pi constant
#define PI 3.14159265358979323846
```

```
// max lights
#define MAX_LIGHTS 10

// input vertex
in Vertex
{
    vec3 Position;
    vec3 Normal;
} vertex;

// material type
struct Material
{
    float Roughness;
    float Metallic;
    vec3 Albedo;
};

// point light type
struct PointLight
{
    float Intensity;
    vec3 Radiance;
    vec3 Position;
};

// point light uniforms
uniform PointLight u_pointLights[MAX_LIGHTS];
uniform int u_nbrPointLight;

uniform Material u_material;
uniform vec3 u_viewPos;

// computes fresnel reflectivity
vec3 FresnelSchlick(float cosTheta, vec3 F0)
```

```

{
    return F0 + (1.0 - F0) * pow(1.0 - cosTheta, 5.0);
}

// computes geometry schlick ggx
float GeometrySchlickGGX(float NdotV, float roughness)
{
    float r = (roughness + 1.0);
    float k = (r * r) / 8.0;
    float num = NdotV;
    float denom = NdotV * (1.0 - k) + k;
    return num / denom;
}

// computes distribution ggx
float DistributionGGX(vec3 N, vec3 H, float roughness)
{
    float a = roughness * roughness;
    float a2 = a * a;
    float NdotH = max(dot(N, H), 0.0);
    float NdotH2 = NdotH * NdotH;
    float num = a2;
    float denom = (NdotH2 * (a2 - 1.0) + 1.0);
    denom = PI * denom * denom;
    return num / denom;
}

// computes geometry smith ggx
float GeometrySmithGGX(float NdotV, float NdotL, float
roughness)
{
    float ggx1 = GeometrySchlickGGX(NdotV, roughness);
    float ggx2 = GeometrySchlickGGX(NdotL, roughness);
    return ggx1 * ggx2;
}

```

```

// compute point lights
vec3 ComputePointLights(vec3 N, vec3 V, vec3 F0)
{
    vec3 result = vec3(0.0);

    for (int i = 0; i < u_nbrPointLight; ++i)
    {
        // compute parameters
        vec3 L = normalize(u_pointLights[i].Position -
vertex.Position);
        vec3 H = normalize(L + V);
        float NdotL = max(dot(N, L), 0.0);
        float NdotV = max(dot(N, V), 0.0);

        // Cook-Torrance (BRDF)
        float NDF = DistributionGGX(N, H, u_material.Roughness);
        vec3 FS = FresnelSchlick(clamp(dot(H, V), 0.0, 1.0),
F0);
        float GS = GeometrySmithGGX(NdotV, NdotL,
u_material.Roughness);

        // diffuse light
        vec3 diffuse = (vec3(1.0) - FS) * (1.0 -
u_material.Metallic);

        // specular light
        vec3 specular = (NDF * GS * FS) / max(4.0 * NdotV *
NdotL, 0.0001);

        // light attenuation
        float distance = length(u_pointLights[i].Position -
vertex.Position);
        float attenuation = u_pointLights[i].Intensity/(distance *
distance);

        // combine components
    }
}

```

```

        result += (diffuse * u_material.Albedo / PI + specular)
    *
        u_pointLights[i].Radiance * attenuation * NdotL;

        // break if max light
        if (i >= MAX_LIGHTS - 1) { break; }
    }

    return result;
}

// main function
void main()
{
    // normalized surface normal
    vec3 N = normalize(vertex.Normal);

    // camera view direction
    vec3 V = normalize(u_viewPos - vertex.Position);

    // fresnel base reflectivity
    vec3 F0 = mix(vec3(0.04), u_material.Albedo,
u_material.Metallic);

    // point lights contribution
    vec3 result = ComputePointLights(N, V, F0);

    // final color calculation
    out_fragment = vec4(result, 1.0);
}

++FRAGMENT++

```

The vertex shader remains unchanged. In the fragment shader, we have added some new uniforms

for point lights. We have a uniform that tells how many point lights where sent to the shader (`u_nbrPointLight`) and the other uniform called (`u_pointLights`) which is an array that stores all point light sources. We have additionally added a separate function `ComputePointLights()` to perform the computation of the contribution of all point lights in the scene.

Shader Class Update

We are required to reflect these changes in the C++ shader to make sure that our engine's shader class and the GLSL shader both speak the same language.

Listing 6.16: Graphics/Shaders/PBR.h

```
#pragma once
#include "Shader.h"
#include "../Utilities/Data.h"

namespace EmPy
{
    struct PbrShader : Shader
    {
        EMPY_INLINE PbrShader(const std::string& filename) :
        Shader(filename)
        {
            u_NbrPointLight =
glGetUniformLocation(m_ShaderID, "u_nbrPointLight");
        }
    };
}
```

```

        u_ViewPos = glGetUniformLocation(m_ShaderID,
"u_viewPos");

        u_Roughness = glGetUniformLocation(m_ShaderID,
"u_material.Roughness");
        u_Metallic = glGetUniformLocation(m_ShaderID,
"u_material.Metallic");
        u_Albedo = glGetUniformLocation(m_ShaderID,
"u_material.Albedo");

        u_Model = glGetUniformLocation(m_ShaderID,
"u_model");
        u_View = glGetUniformLocation(m_ShaderID,
"u_view");
        u_Proj = glGetUniformLocation(m_ShaderID,
"u_proj");
    }

    EMPY_INLINE void SetPointLight(PointLight& light,
Transform3D& transform, int32_t index)
{
    std::string intensity = "u_pointLights[" +
std::to_string(index) + "].Intensity";
    std::string radiance = "u_pointLights[" +
std::to_string(index) + "].Radiance";
    std::string position = "u_pointLights[" +
std::to_string(index) + "].Position";

    uint32_t u_intensity =
glGetUniformLocation(m_ShaderID, intensity.c_str());
    uint32_t u_radiance =
glGetUniformLocation(m_ShaderID, radiance.c_str());
    uint32_t u_position =
glGetUniformLocation(m_ShaderID, position.c_str());

    glUniform3fv(u_position, 1,

```

```

&transform.Translate.x);

    glUniform3fv(u_radiance, 1, &light.Radiance.x);
    glUniform1f(u_intensity, light.Intensity);
}

EMPTY_INLINE void SetPointLightCount(int32_t count)
{
    glUniform1i(u_NbrPointLight, count);
}

EMPTY_INLINE void SetCamera(Camera3D& camera,
Transform3D& transform, float ratio)
{
    glUniformMatrix4fv(u_Proj, 1, GL_FALSE,
glm::value_ptr(camera.Projection(ratio)));
    glUniformMatrix4fv(u_View, 1, GL_FALSE,
glm::value_ptr(camera.View(transform)));
    glUniform3fv(u_ViewPos, 1,
&transform.Translate.x);
}

EMPTY_INLINE void Draw(Model3D& model, PbrMaterial&
material, Transform3D& transform)
{
    glUniformMatrix4fv(u_Model, 1, GL_FALSE,
glm::value_ptr(transform.Matrix()));
    glUniform3fv(u_Albedo, 1, &material.Albedo.x);
    glUniform1f(u_Roughness, material.Roughness);
    glUniform1f(u_Metallic, material.Metallic);
    model->Draw(GL_TRIANGLES);
}

private:
    uint32_t u_NbrPointLight = 0u;
    uint32_t u_ViewPos = 0u;

```

```

        uint32_t u_Roughness = 0u;
        uint32_t u_Metallic = 0u;
        uint32_t u_Albedo = 0u;

        uint32_t u_Model = 0u;
        uint32_t u_View = 0u;
        uint32_t u_Proj = 0u;
    };
}

```

Important to note here ([Listing 6.16](#)) are the two functions, `SetPointLight()` and `SetPointLightCount()`, along with the minor adjustments made in both the constructor and the member variables. We are not caching the uniform locations of the point light sources because we do not know how many will be provided at the time of rendering. We can still do that if want, considering there is a limit to the number of point light sources we can typically send to the shader for computation. Perhaps this optimization can be explored later.

Renderer Class Update

The `GraphicsRenderer` class has to implement these functions so that our application can interact with the underlying implementation in the shader class.

Listing 6.17: `Graphics/Renderer.h`

```

// sets point light source uniforms
EMPY_INLINE void SetPointLight(PointLight& light,
Transform3D& transform, uint32_t index)
{
    m_Pbr->SetPointLight(light, transform, index);
}

// sets total number of point lights
EMPY_INLINE void SetPointLightCount(int32_t count)
{
    m_Pbr->SetPointLightCount(count);
}

```

Application Class Update

Finally, we can test this in the application class by adding two new entities with point light components. See ([Listing 6.18](#))

Listing 6.18: Application/Application.h

```

EMPY_INLINE void RunContext()
{
    auto model1 = std::make_shared<Model>
("Resources/Models/cube.fbx");
    auto model2 = std::make_shared<Model>
("Resources/Models/sphere.fbx");

    // create scene camera
    auto camera = CreateEntt<Entity>();
    camera.Attach<TransformComponent>().Transform.Translate.z
= 2.0f;

```

```

camera.Attach<CameraComponent>();

// create point light 1           <----- added
auto plight1 = CreateEntt<Entity>();
plight1.Attach<PointLightComponent>().Light.Radiance.b =
0.0f;
auto& tp1 = plight1.Attach<TransformComponent>
().Transform;
tp1.Translate = glm::vec3(-0.5f, 0.0f, 2.0f);

// create point light 2           <----- added
auto plight2 = CreateEntt<Entity>();
plight2.Attach<PointLightComponent>().Light.Radiance.g =
0.0f;
auto& tp2 = plight2.Attach<TransformComponent>
().Transform;
tp2.Translate = glm::vec3(0.5f, 0.0f, 2.0f);

// create cube entity
auto cube = CreateEntt<Entity>();
cube.Attach<ModelComponent>().Model = model2;
auto& tc = cube.Attach<TransformComponent>().Transform;
tc.Scale = glm::vec3(1.5f);

while (m_Context->Window->PollEvents())
{
    // render new frame
    m_Context->Renderer->NewFrame();

    // set shader point lights <----- added
    int32_t lightCounter = 0u;

    EnttView<Entity, PointLightComponent>([this,
&lightCounter] (auto entity, auto& comp)
    {
        auto& transform = entity.template

```

```

Get<TransformComponent>().Transform;
    m_Context->Renderer->SetPointLight(comp.Light,
transform, lightCounter);
    lightCounter++;
} );

// set number of point lights
m_Context->Renderer->SetPointLightCount(lightCounter);

/* ... same as before ... */
}
}

```

Most of the code is similar to what we had previously. We have added two point light entities and the according "View" in the main loop to send the properties of our point lights to the shader. This time, we are rendering the sphere model rather than a cube. See ([Figure 6.8](#)).

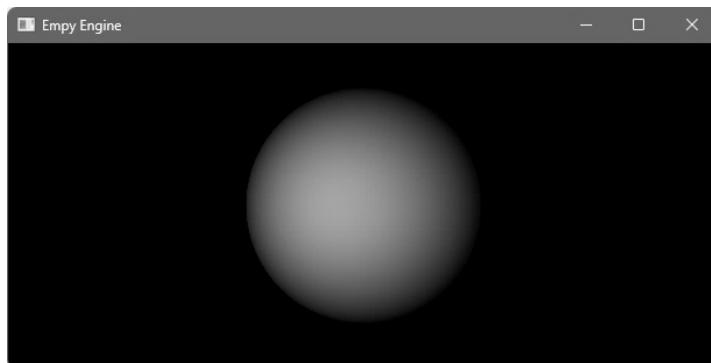


Figure 6.8: Multiple Point Lights

6.3.2 Directional lights

Unlike point lights, directional lights do not have a specific location, but are characterized by their direction. They emit light in a constant direction throughout the entire scene, often mimicking natural sources like the sun. They provide uniform lighting from a distant source, casting parallel rays across the scene. (Figure 6.9) provides a visual representation.



Figure 6.9: Directional Light Source

Directional Light Class

Following the same steps as earlier for point lights, we can also integrate directional light in our rendering pipeline.

Listing 6.19: Graphics/Utilities/Data.
h

```
struct DirectLight
{
    EMPL_INLINE DirectLight(const DirectLight&) = default;
```

```
EMPTY_INLINE DirectLight() = default;

glm::vec3 Radiance = glm::vec3(1.0f);
float Intensity = 2.0f;
};
```

Add this directional light class to your project's **Data.h** file.

Directional Light Component

Add the directional light component as well.

Listing 6.20: Auxiliaries/ECS. h

```
struct DirectLightComponent
{
    EMPTY_INLINE DirectLightComponent(const
DirectLightComponent&) = default;
    EMPTY_INLINE DirectLightComponent() = default;
    DirectLight Light;
};
```

Shader Class Update

We are also required to implement additional functions in our PBR shader class to set the uniforms for directional light sources.

Listing 6.21: Graphics/Shaders/PBR. h

```
#pragma once
#include "Shader.h"
#include "../Utilities/Data.h"

namespace EmPy
{
    struct PbrShader : Shader
    {
        EMPY_INLINE PbrShader(const std::string& filename) :
        Shader(filename)
        {
            u_NbrDirectLight =
glGetUniformLocation(m_ShaderID, "u_nbrDirectLight");
            u_NbrPointLight =
glGetUniformLocation(m_ShaderID, "u_nbrPointLight");
            u_ViewPos = glGetUniformLocation(m_ShaderID,
"u_viewPos");

            u_Roughness = glGetUniformLocation(m_ShaderID,
"u_material.Roughness");
            u_Metallic = glGetUniformLocation(m_ShaderID,
"u_material.Metallic");
            u_Albedo = glGetUniformLocation(m_ShaderID,
"u_material.Albedo");

            u_Model = glGetUniformLocation(m_ShaderID,
"u_model");
            u_View = glGetUniformLocation(m_ShaderID,
"u_view");
            u_Proj = glGetUniformLocation(m_ShaderID,
"u_proj");
        }
    };
}
```

```

    EMPY_INLINE void SetDirectLight(DirectLight& light,
Transform3D& transform, int32_t index)
{
    std::string intensity = "u_directLights[" + std::to_string(index) + "].Intensity";
    std::string direction = "u_directLights[" + std::to_string(index) + "].Direction";
    std::string radiance = "u_directLights[" + std::to_string(index) + "].Radiance";

    uint32_t u_intensity =
glGetUniformLocation(m_ShaderID, intensity.c_str());
    uint32_t u_position =
glGetUniformLocation(m_ShaderID, direction.c_str());
    uint32_t u_radiance =
glGetUniformLocation(m_ShaderID, radiance.c_str());

    glUniform3fv(u_position, 1,
&transform.Rotation.x);
    glUniform3fv(u_radiance, 1, &light.Radiance.x);
    glUniform1f(u_intensity, light.Intensity);
}

EMPY_INLINE void SetDirectLightCount(int32_t count)
{
    glUniform1i(u_NbrDirectLight, count);
}

/* ... same as before ... */

private:
    uint32_t u_NbrDirectLight = 0u;
    uint32_t u_NbrPointLight = 0u;
    uint32_t u_ViewPos = 0u;

```

```

        uint32_t u_Roughness = 0u;
        uint32_t u_Metallic = 0u;
        uint32_t u_Albedo = 0u;

        uint32_t u_Model = 0u;
        uint32_t u_View = 0u;
        uint32_t u_Proj = 0u;
    };
}

```

You will note how the direction of the light source is represented by the rotation component of the transform rather than the position.

Renderer Class Update

Update the *Graphicsrenderer* class by adding the following functions.

Listing 6.22: Graphics/Renderer.h

```

EMPTY_INLINE void SetDirectLight(DirectLight& light,
Transform3D& transform, uint32_t index)
{
    m_Pbr->SetDirectLight(light, transform, index);
}

EMPTY_INLINE void SetDirectLightCount(int32_t count)
{
    m_Pbr->SetDirectLightCount(count);
}

```

PBR Shader Update

Update the shader by making sure that the uniforms for directional light sources are defined as well as the function to compute their contribution.

Listing 6.23: Resource/Shaders/pbr.gls

```
/* ... same as before ... */

// direct light type
struct DirectLight
{
    float Intensity;
    vec3 Direction;
    vec3 Radiance;
};

// point light uniforms
uniform DirectLight u_directLights[MAX_LIGHTS];
uniform int u_nbrDirectLight;

/* ... same as before ... */

// compute direct lights
vec3 ComputeDirectLights(vec3 N, vec3 V, vec3 F0)
{
    vec3 result = vec3(0.0);

    for (int i = 0; i < u_nbrDirectLight; ++i)
    {
        // compute parameters
        vec3 L = -normalize(u_directLights[i].Direction);
```

```

    float NdotL = max(dot(N, L), 0.0);
    float NdotV = max(dot(N, V), 0.0);
    vec3 H = normalize(L + V);

    // Cook-Torrance (BRDF)
    float NDF = DistributionGGX(N, H, u_material.Roughness);
    vec3 FS = FresnelSchlick(clamp(dot(H, V), 0.0, 1.0),
F0);
    float GS = GeometrySmithGGX(NdotV, NdotL,
u_material.Roughness);

    // diffuse light
    vec3 diffuse = (vec3(1.0) - FS);
    diffuse *= (1.0 - u_material.Metallic);

    // specular light
    vec3 specular = (NDF * GS * FS) / max(4.0 * NdotV *
NdotL, 0.0001);

    // combine components
    result += (diffuse * u_material.Albedo / PI + specular)
*
        u_directLights[i].Radiance * NdotL *
u_directLights[i].Intensity;

    // break if max light
    if (i >= MAX_LIGHTS - 1) { break; }

}

return result;
}

// main function
void main()
{
    // normalized surface normal

```

```

vec3 N = normalize(vertex.Normal);

// camera view direction
vec3 V = normalize(u_viewPos - vertex.Position);

// fresnel base reflectivity
vec3 F0 = mix(vec3(0.04), u_material.Albedo,
u_material.Metallic);

// point lights contribution
vec3 result = ComputePointLights(N, V, F0);
result += ComputeDirectLights(N, V, F0);

// final color calculation
out_fragment = vec4(result, 1.0);
}

++FRAGMENT++

```

If you look closely in the `ComputeDirectLits()` function, you will observe that we have not computed the attenuation. The sun light for instance is quite uniformly distributed.

Application Class Update

Finally, replace the point light in the `RunContext()` function with a directional light entity and make sure to also add the logic to upload their value to the shader as shown below.

Listing 6.24: Application/Application. h

```
EMPTY_INLINE void RunContext()
{
    auto model1 = std::make_shared<Model>
("Resources/Models/cube.fbx");
    auto model2 = std::make_shared<Model>
("Resources/Models/sphere.fbx");

    // create scene camera
    auto camera = CreateEntt<Entity>();
    camera.Attach<TransformComponent>().Transform.Translate.z
= 2.0f;
    camera.Attach<CameraComponent>();

    // create point light 1
    auto dlight1 = CreateEntt<Entity>();
    dlight1.Attach<DirectLightComponent>();
    auto& tp1 = dlight1.Attach<TransformComponent>
().Transform;
    tp1.Rotation.x = 0.0f;
    tp1.Rotation.y = -0.1f;
    tp1.Rotation.z = -0.5f;

    // create cube entity
    auto cube = CreateEntt<Entity>();
    cube.Attach<ModelComponent>().Model = model2;
    cube.Attach<TransformComponent>().Transform.Scale =
glm::vec3(1.5f);

    // application main loop
    while(m_Context->Window->PollEvents())
    {
        // start new frame
    }
}
```

```

m_Context->Renderer->NewFrame();

// set shader point lights
int32_t lightCounter = 0u;

EnttView<Entity, PointLightComponent>([this,
&lightCounter] (auto entity, auto& comp)
{
    auto& transform = entity.template
Get<TransformComponent>().Transform;
    m_Context->Renderer->SetPointLight(comp.Light,
transform, lightCounter);
    lightCounter++;
});

// set number of point lights
m_Context->Renderer->SetPointLightCount(lightCounter);

// set shader direct. lights
lightCounter = 0;

EnttView<Entity, DirectLightComponent>([this,
&lightCounter] (auto entity, auto& comp)
{
    auto& transform = entity.template
Get<TransformComponent>().Transform;
    m_Context->Renderer->SetDirectLight(comp.Light,
transform, lightCounter);
    lightCounter++;
});

// set number of point lights
m_Context->Renderer-
>SetDirectLightCount(lightCounter);

/* ... same as before ... */

```

```
    }  
}
```

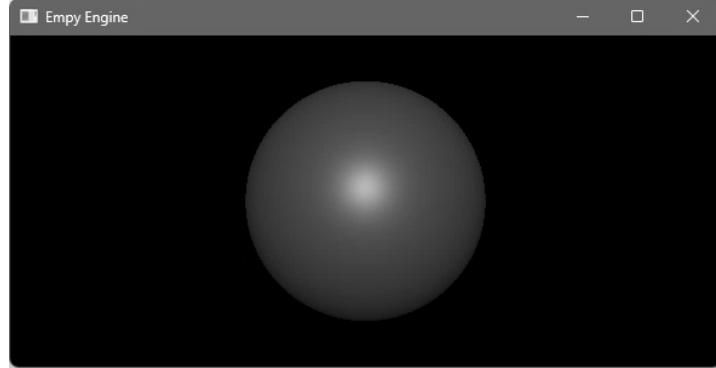


Figure 6.10: Single Directional Light

6.3.3 Spotlights

Spotlights are directional lights that emit light within a specified cone angle, akin to focused beams. They are versatile for highlighting specific areas or objects within a scene, allowing for controlled illumination and casting well-defined shadows. In addition to the position and direction, spotlights have "Cutoff and Falloff Angles".

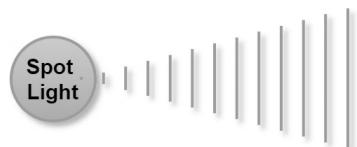


Figure 6.11:

Spotlight Source

The cutoff angle determines the angle within which the spotlight emits full intensity. Within the cone defined by the cutoff angle, the light intensity is at its maximum, creating a focused beam. Beyond the cutoff angle, the light intensity gradually diminishes until it reaches zero at the outer angle, or falloff angle. This area outside the cutoff angle is where the light intensity attenuates, creating a smooth transition from full intensity to zero. The formula to calculate the intensity of light from a spotlight at a specific point in space can involve various factors, including attenuation due to distance and falloff based on the angle from the spotlight's direction.

Attenuation and Falloff Formula

Given a point in space P and a spotlight with parameters:

- I_{\max} = Maximum intensity of the spotlight. d = distance between the spotlight's position and point P . a = attenuation factor. θ = Angle between the spotlight's direction and the vector from the spotlight to point P . ϕ = cutoff angle of the spotlight cone. $f(\theta)$ = a falloff function that describes how the intensity diminishes

with angle. The intensity I at point P due to the spotlight can be calculated as:

$$I = \begin{cases} I_{\max} \cdot \text{attenuation}(d) \cdot f(\theta), & \text{if } \theta \leq \phi \\ 0, & \text{if } \theta > \phi \end{cases}$$

The attenuation function, which reduces intensity based on distance, can vary. For example, a common attenuation formula is:

$$\text{attenuation}(d) = \frac{1}{a + b \cdot d + c \cdot d^2}$$

Where a , b , and c are constants determining the rate of attenuation with distance.

The falloff function $f(\theta)$ determines how the intensity diminishes with the angle θ from the spotlight's direction. For a simple linear falloff:

$$f(\theta) = 1 - \frac{\theta}{\phi}$$

This linear falloff reduces intensity linearly from 1 at the spotlight's direction to 0 at the cutoff angle ϕ .

These formulas illustrate how the intensity of light from a spotlight can be calculated, incorporating factors such as distance attenuation and angle-based

falloff to simulate the behavior of real-world spotlights in a rendering or lighting system. Adjusting the spotlight's parameters alters the distribution and reach of emitted light in the scene.

Spotlight Component

I assume you already know how it goes. Add both the spot light class and component to your project.

Listing 6.25: Graphics/Utilities/Data.
h

```
struct SpotLight
{
    EMPY_INLINE SpotLight(const SpotLight&) = default;
    EMPY_INLINE SpotLight() = default;

    glm::vec3 Radiance = glm::vec3(1.0f);
    float Intensity = 3.5f;
    float FallOff = 60.5f;
    float CutOff = 20.0f;
};
```

Listing 6.26: Auxiliaries/ECS.
h

```
struct SpotLightComponent
{
    EMPY_INLINE SpotLightComponent(const SpotLightComponent&)
= default;
    EMPY_INLINE SpotLightComponent() = default;
```

```
    SpotLight Light;  
};
```

Shader Class Update

Similar to both point lights and directional lights we also have to add additional functions to our shader abstraction.

Listing 6.27: Graphics/Shaders/PBR.
h

```
#pragma once  
#include "Shader.h"  
#include "../Utilities/Data.h"  
  
namespace EmPy  
{  
    struct PbrShader : Shader  
    {  
        EMPY_INLINE PbrShader(const std::string& filename) :  
        Shader(filename)  
        {  
            u_NbrDirectLight =  
                glGetUniformLocation(m_ShaderID, "u_nbrDirectLight");  
            u_NbrPointLight =  
                glGetUniformLocation(m_ShaderID, "u_nbrPointLight");  
            u_NbrSpotLight = glGetUniformLocation(m_ShaderID,  
                "u_nbrSpotLight");  
  
            /* ... same as before ... */  
        }  
}
```

```

EMPTY_INLINE void SetSpotLight(SpotLight& light,
Transform3D& transform, int32_t index)
{
    std::string intensity = "u_spotLights[" +
std::to_string(index) + "].Intensity";
    std::string direction = "u_spotLights[" +
std::to_string(index) + "].Direction";
    std::string radiance = "u_spotLights[" +
std::to_string(index) + "].Radiance";
    std::string position = "u_spotLights[" +
std::to_string(index) + "].Position";
    std::string falloff = "u_spotLights[" +
std::to_string(index) + "].Falloff";
    std::string cutoff = "u_spotLights[" +
std::to_string(index) + "].Cutoff";

    uint32_t u_intensity =
glGetUniformLocation(m_ShaderID, intensity.c_str());
    uint32_t u_direction =
glGetUniformLocation(m_ShaderID, direction.c_str());
    uint32_t u_radiance =
glGetUniformLocation(m_ShaderID, radiance.c_str());
    uint32_t u_position =
glGetUniformLocation(m_ShaderID, position.c_str());
    uint32_t u_falloff =
glGetUniformLocation(m_ShaderID, falloff.c_str());
    uint32_t u_cutoff =
glGetUniformLocation(m_ShaderID, cutoff.c_str());

    glUniform3fv(u_direction, 1,
&transform.Rotation.x);
    glUniform3fv(u_position, 1,
&transform.Translate.x);
    glUniform1f(u_falloff,
glm::radians(light.FallOff));
    glUniform1f(u_cutoff, glm::radians(light.CutOff));
}

```

```

        glUniform3fv(u_radiance, 1, &light.Radiance.x);
        glUniform1f(u_intensity, light.Intensity);
    }

EMPTY_INLINE void SetSpotLightCount(int32_t count)
{
    glUniform1i(u_NbrSpotLight, count);
}

/* ... same as before ... */

private:
    uint32_t u_NbrDirectLight = 0u;
    uint32_t u_NbrPointLight = 0u;
    uint32_t u_NbrSpotLight = 0u;
    uint32_t u_ViewPos = 0u;

    uint32_t u_Roughness = 0u;
    uint32_t u_Metallic = 0u;
    uint32_t u_Albedo = 0u;

    uint32_t u_Model = 0u;
    uint32_t u_View = 0u;
    uint32_t u_Proj = 0u;
};

}

```

An important note in this code (Listing 6.27) is the `glm::radians()` function used to convert both the "Falloff" and the "Cutoff" angles from degree to radian. Not doing this will result in our shader not showing anything.

PBR Shader Update

Similar to the previous light sources, we will also implement a function to compute the contribution of spotlight sources in the scene. See (Listing [6.28](#)).

Listing 6.28: Resources/Shaders/pbr.gls

```
/* ... same as before ... */

// spot light type
struct SpotLight
{
    float Intensity;
    vec3 Direction;
    vec3 Position;
    vec3 Radiance;
    float Falloff;
    float CutOff;
};

// spot light uniforms
uniform SpotLight u_spotLights[MAX_LIGHTS];
uniform int u_nbrSpotLight;

// compute spot lights
vec3 ComputeSpotLights(vec3 N, vec3 V, vec3 F0)
{
    vec3 result = vec3(0.0);

    for (int i = 0; i < u_nbrSpotLight; ++i)
    {
        // compute parameters
```

```

    vec3 L = normalize(u_spotLights[i].Position -
vertex.Position);

    float NdotL = max(dot(N, L), 0.0);
    float NdotV = max(dot(N, V), 0.0);
    vec3 H = normalize(L + V);

    // Cook-Torrance (BRDF)
    float NDF = DistributionGGX(N, H, u_material.Roughness);
    vec3 FS = FresnelSchlick(clamp(dot(H, V), 0.0, 1.0),
F0);

    float GS = GeometrySmithGGX(NdotV, NdotL,
u_material.Roughness);

    // diffuse light
    vec3 diffuse = (vec3(1.0) - FS) * (1.0 -
u_material.Metallic);

    // specular light
    vec3 specular = (NDF * GS * FS) / max(4.0 * NdotV *
NdotL, 0.0001);

    // compute spot
    float theta = dot(L, normalize(-
u_spotLights[i].Direction));
    float epsilon = (u_spotLights[i].FallOff -
u_spotLights[i].CutOff);
    float spotFactor = clamp((theta -
u_spotLights[i].CutOff)/epsilon, 0.0, 1.0);

    // light attenuation
    float distance = length(u_spotLights[i].Position -
vertex.Position);
    float attenuation = u_spotLights[i].Intensity / (distance *
distance);

    // combine components

```

```

        result += (diffuse * u_material.Albedo / PI + specular)
    *
        u_spotLights[i].Radiance * u_spotLights[i].Intensity *
        attenuation * NdotL * spotFactor;

        // break if max light
        if (i >= MAX_LIGHTS - 1) { break; }
    }

    return result;
}

// main function
void main()
{
    // normalized surface normal
    vec3 N = normalize(vertex.Normal);

    // camera view direction
    vec3 V = normalize(u_viewPos - vertex.Position);

    // fresnel base reflectivity
    vec3 F0 = mix(vec3(0.04), u_material.Albedo,
    u_material.Metallic);

    // point lights contribution
    vec3 result = ComputeDirectLights(N, V, F0);
    result += ComputePointLights(N, V, F0);
    result += ComputeSpotLights(N, V, F0);

    // final color calculation
    out_fragment = vec4(result, 1.0);
}
++FRAGMENT++

```

Once again, the vertex shader remains unchanged. You can notice inside the `ComputeSpotLights()` function how we are able to compute the spot-factor using both the Cutoff and the Falloff angles. The spot-factor is then used to multiply the final result.

Renderer Class Update

Add the following wrapper functions to your renderer to set spotlight parameters from outside.

Listing 6.29: Auxiliaries/ECS. h

```
// set single spotlight uniforms
EMPY_INLINE void SetSpotLight(SpotLight& light, Transform3D&
transform, uint32_t index)
{
    m_Pbr->SetSpotLight(light, transform, index);
}

// set total number of spotlight
EMPY_INLINE void SetSpotLightCount(int32_t count)
{
    m_Pbr->SetSpotLightCount(count);
}
```

Application Class Update

Now create a spotlight in your application and add an entity view for spotlights.

Listing 6.30: Application/Application. h

```
EMPTY_INLINE void RunContext()
{
    auto model1 = std::make_shared<Model>
("Resources/Models/cube.fbx");
    auto model2 = std::make_shared<Model>
("Resources/Models/sphere.fbx");

    // create scene camera
    auto camera = CreateEntt<Entity>();
    camera.Attach<TransformComponent>().Transform.Translate.z
= 3.0f;
    camera.Attach<CameraComponent>();

    // create point light 1
    auto slight = CreateEntt<Entity>();
    slight.Attach<SpotLightComponent>();
    auto& stp = slight.Attach<TransformComponent>
().Transform;
    stp.Rotation = glm::vec3(0.0f, 0.0f, -1.0f);
    stp.Translate.z = 1.0f;

    // create cube entity
    auto cube = CreateEntt<Entity>();
    cube.Attach<ModelComponent>().Model = model1;
    auto& t = cube.Attach<TransformComponent>().Transform;
    t.Scale = glm::vec3(3.0f, 3.0f, 1.0f);
    t.Translate.z = -1.0f;

    // application main loop
    while(m_Context->Window->PollEvents())
    {
        // start new frame
```

```

m_Context->Renderer->NewFrame();

    // set shader point lights
    int32_t lightCounter = 0;
    EnttView<Entity, PointLightComponent>([this,
&lightCounter] (auto entity, auto& comp)
{
    auto& transform = entity.template
Get<TransformComponent>().Transform;
    m_Context->Renderer->SetPointLight(comp.Light,
transform, lightCounter);
    lightCounter++;
});

    // set number of point lights
m_Context->Renderer->SetPointLightCount(lightCounter);

    // set shader direct. lights
    lightCounter = 0;
    EnttView<Entity, DirectLightComponent>([this,
&lightCounter] (auto entity, auto& comp)
{
    auto& transform = entity.template
Get<TransformComponent>().Transform;
    m_Context->Renderer->SetDirectLight(comp.Light,
transform, lightCounter);
    lightCounter++;
});

    // set number of direct lights
m_Context->Renderer-
>SetDirectLightCount(lightCounter);

    // set shader spot lights
    lightCounter = 0;
    EnttView<Entity, SpotLightComponent>([this,
&lightCounter] (auto entity, auto& comp)
{

```

```
        auto& transform = entity.template
Get<TransformComponent>().Transform;
        m_Context->Renderer->SetSpotLight(comp.Light,
transform, lightCounter);
        lightCounter++;
    });
// set number of spot lights
m_Context->Renderer->SetSpotLightCount(lightCounter);

// the rest ramins unchanged
}
}
```

Compiling and running this will result in (Figure 6.12). You can see how the light is spotted at the center of the cube. We will be able to conveniently play around with these light sources with the addition of the GUI. More on that later.

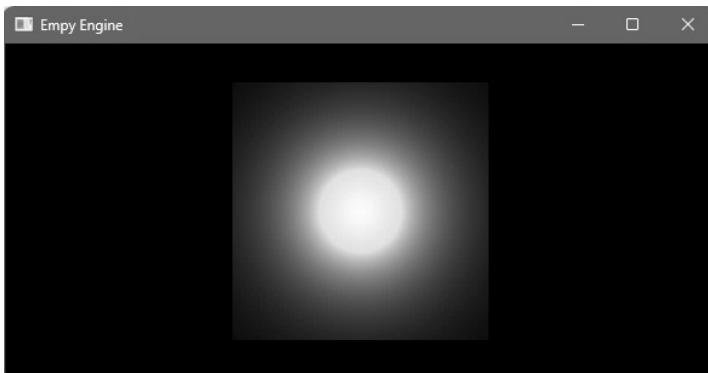


Figure 6.12: Single Spotlight Source

6.4 Rendering Textures

Texture mapping is the process of applying texture images to 3D models to enhance their visual appearance and realism. These textures contain information such as color (albedo), roughness, metallic properties, normals, and more, allowing for high-level surface representations and accurate material rendering.

Here's an overview of how textures are typically used in OpenGL:

- ☞ **Creating a Texture:** To use a texture, you first need to generate and configure it. This involves loading an image into memory and then creating a texture object in OpenGL using functions like `glGenTextures()` and `glBindTexture()`.
- ☞ **Loading Data into the Texture:** Once the texture object is created, you'll load your image data into it. This can be done using functions like `glTexImage2D()` or `glTexSubImage2D()`.
- ☞ **Setting Texture Parameters:** You can set various parameters for how the texture behaves, such as its wrapping behavior (`GL_REPEAT`, `GL_CLAMP_TO_EDGE`, etc.), filtering methods (`GL_LINEAR`, `GL_NEAREST`), mip-

mapping, and more using functions like

`glTexParameteri()`.

☞ **Using the Texture in Shaders:** To actually use the texture in your rendering process, you'll pass it to your shaders using uniform variables. In the shader code, you'd sample the texture using the `texture2D()` function (in older versions of GLSL, like for OpenGL ES 2.0 or earlier) or the `texture()` function in more recent versions.

Texture units in OpenGL are essential for managing multiple textures in a rendering pipeline. These units allow you to bind multiple textures simultaneously and use them in shaders during rendering. Each GPU has a limited number of texture units available (which can vary between different devices). Commonly, modern hardware supports a minimum of 16 texture units, and more advanced GPU may support even more.

Here's an overview of how texture units are typically used:

☞ **Active Texture Units:** To work with a texture unit, you need to make it active using `glActiveTexture()`. This function takes an argument specifying which texture unit to activate (`GL_TEXTURE0`, `GL_TEXTURE1`, etc.).

☞ **Binding Textures to Units:** After making a texture unit active, you can bind a texture to it using `glBindTexture()`. This associates the currently bound texture with the active texture unit. Each texture unit can have its own texture bound to it.

☞ **Using Texture Units in Shaders:** In the shader code you'd use a uniform variable (like in (Listing 6.31)) to specify which texture unit to sample from. For instance, if you're using `GL_TEXTURE0`, you'd have a uniform variable associated with that unit in the shader. Here's a simple example of how texture sampling works in a fragment shader:

Listing 6.31: Texture Sampling Example

```
#version 330 core

in vec2 TexCoord; // texture coordinates passed from the
vertex shader
uniform sampler2D textureUnit; // The texture unit

out vec4 FragColor; // output color

void main()
{
    FragColor = texture(textureUnit, TexCoord);
}
```

This shader takes the texture coordinates `TexCoord` and uses the `texture()` function to fetch the color

from the specified texture `textureUnit`. This code is quite similar to that of the output shader found in **final.glsl**.

6.4.1 Material Textures

PBR materials rely on textures to accurately represent surface properties. Following are some common texture types used in PBR materials for rendering:

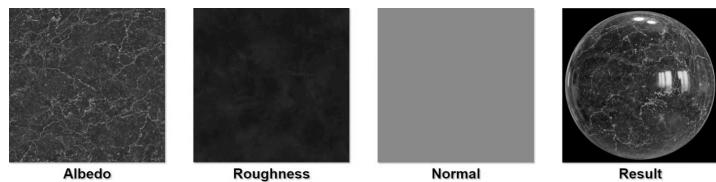


Figure 6.13: PBR Textures
[Lennart](#)

- ☞ **Albedo Texture:** This texture represents the base color of the material. It defines the surface's inherent color without any lighting.
- ☞ **Roughness Texture:** Roughness determines the microsurface of the material. A black pixel represents a perfectly smooth surface, while white denotes a very rough surface. This texture controls how light scatters on the surface, affecting its perceived smoothness.

☞ **Metallic Texture:** This texture defines which parts of the surface are metallic and which are dielectric (non-metallic). Values closer to black represent dielectric surfaces (e.g., wood, plastic), while values closer to white represent metallic surfaces (e.g., metals).

☞ **Normal Map Texture:** Normal maps alter the surface's normals, affecting how light interacts with the object's geometry without adding more geometry. They provide the illusion of surface detail and depth.

☞ **Ambient Occlusion Texture:** Ambient Occlusion (AO) maps define areas where light is occluded or blocked, usually occurring in corners, crevices, or areas where objects meet. These areas tend to be darker, contributing to a more realistic look by simulating shadowing.

☞ **Height/Displacement Map:** These maps affect the surface's elevation, creating the illusion of depth. They can displace vertices or modify surface normals for added realism.

☞ **Emissive Texture:** Used to simulate materials that emit light, like screens, Light-Emitting Diodes (LEDs), or glowing surfaces.

PBR textures are often used in game engines like Unreal Engine or Unity to create highly realistic materials. When applied in a rendering engine or shader, these textures contribute to how light interacts with the surface, affecting the final appearance of rendered objects. The combination and accurate representation of these textures lead to more believable and visually appealing 3D scenes.

6.4.2 Texture Mapping

Let's integrate textures into our rendering pipeline. To start, we'll establish the texture type or class within our project and proceed by loading pixel data from image files like PNG or JPG. Integrating this data into our material class will enable us to seamlessly apply these textures when rendering our models.

Add the "**Textures**" directory to your project as depicted below (Figure [6.14](#)).

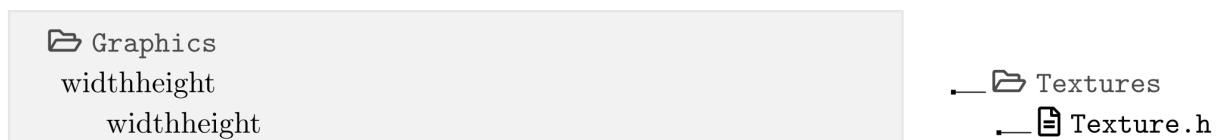


Figure 6.14: Textures Directory

Texture Loader

STB-Image is a popular single-header library written in C for loading various image file formats. It provides a straightforward and lightweight solution for adding image loading capabilities to a C or C++ projects without relying on larger libraries like FreeImage or DevIL.

We can simply add it to our project using Conan, as depicted below ([Listing 6.32](#)):

Listing 6.32: Root/conanfile.txt

```
[requires]
glm/cci.20230113
stb/cci.20230920
opengl/system
spdlog/1.12.0
assimp/5.2.2
entt/3.12.2
glfw/3.3.8
glew/2.2.0

[generators]
cmake

[options]
assimp:shared=False
glew:shared=False
glfw:shared=False
```

Add the include path of STB-Image in the `c_cpp_properties.json` file if you want syntax highlighting for its functions and types. This is not required!

Texture2D Class

The code snippet in (Listing 6.33) defines a `Texture2D` struct, representing an OpenGL 2D texture. This structure encapsulates functionalities to manage textures. The constructor initializes default parameters, and an overloaded constructor facilitates loading a texture from an image file specified by the 'filename' parameter.

Listing 6.33: Graphics/Textures/Texture.h

```
#pragma once
#include "Common/Core.h"
#include <stb_image.h>

namespace EmPy
{
    struct Texture2D
    {
        EMPY_INLINE Texture2D() = default;

        EMPY_INLINE Texture2D(const std::string& filename)
        {
            Load(filename);
        }
    };
}
```

```
}

EMPTY_INLINE ~Texture2D()
{
    glDeleteTextures(1, &m_ID);
}

EMPTY_INLINE bool Load(const std::string& filename)
{
    // flip y axis (common)
    stbi_set_flip_vertically_on_load(true);

    // load texture data
    uint8_t* pixels = stbi_load(filename.c_str(),
        &m_Width, &m_Height, nullptr, 4);

    // check pixels
    if(pixels == nullptr)
    {
        EMPTY_ERROR("failed to load texture!");
        return false;
    }

    // create texture
    glGenTextures(1, &m_ID);
    glBindTexture(GL_TEXTURE_2D, m_ID);

    // load texture to gpu
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, m_Width,
        m_Height, 0, GL_RGBA, GL_UNSIGNED_BYTE, pixels);

    // free allocated memory
    stbi_image_free(pixels);

    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
GL_LINEAR);
```

```

        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
GL_LINEAR);

        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S,
GL_CLAMP_TO_EDGE);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T,
GL_CLAMP_TO_EDGE);
        glGenerateMipmap(GL_TEXTURE_2D);

        glBindTexture(GL_TEXTURE_2D, 0);
        return true;
    }

EMPTY_INLINE void Use(uint32_t uniform, int32_t unit)
{
    glActiveTexture(GL_TEXTURE0 + unit);
    glBindTexture(GL_TEXTURE_2D, m_ID);
    glUniform1i(uniform, unit);
}

EMPTY_INLINE operator uint32_t() const { return m_ID;
}

EMPTY_INLINE int32_t Height() const { return
m_Height; }

EMPTY_INLINE int32_t Width() const { return m_Width;
}

EMPTY_INLINE uint32_t ID() const { return m_ID; }

EMPTY_INLINE void Bind() {
glBindTexture(GL_TEXTURE_2D, m_ID); }
EMPTY_INLINE void Unbind() {
glBindTexture(GL_TEXTURE_2D, 0); }

private:

```

```
    int32_t m_Height = 0;
    int32_t m_Width = 0;
    uint32_t m_ID = 0u;
}
}

// type definition for shared pointer
using Texture = std::shared_ptr<Texture2D>;
```

The `Load()` function handles the loading process. It employs the `stb_image` library added earlier to load image data from the provided file path. The function sets an option to flip the image along the y-axis (common for many image formats). After successfully loading the image data, it generates a texture object (`m_ID`) in OpenGL using `glGenTextures()` and binds it using `glBindTexture()`. It proceeds to load the image data into the GPU memory using `glTexImage2D()`, specifying the texture's format as `GL_RGBA`. It then sets various parameters for the texture, such as filtering and wrapping modes, via `glTexParameter()` calls.

Additionally, it generates mipmaps for the texture using `glGenerateMipmap()` for improved rendering quality at different distances. Mipmaps are pre-generated, lower-resolution versions of a texture. They're used to optimize rendering performance and improve visual quality, especially when textures are viewed at different distances or scales. For example, if the original texture is 360x360 pixels, the next mipmap

level will be 160x160, followed by 80x80, and so on, until a 1x1 pixel texture. See ([Figure 6.15](#)).

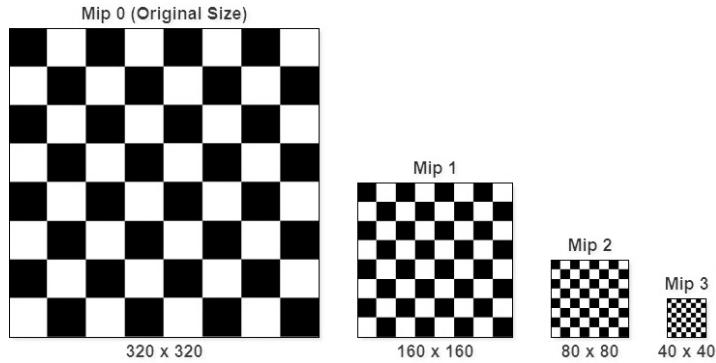


Figure 6.15: Texture Mip-Map Levels

The benefit of mipmaps lies in their usage during rendering. When a textured object appears smaller on the screen than its original size, the GPU can switch to a lower-resolution mipmap level. This process reduces visual artifacts like aliasing and improves performance by sampling from a smaller texture, which requires fewer computations. Additionally, mipmaps help in reducing texture aliasing known as "moire patterns" that can occur when textures are displayed at oblique angles or when they're down-scaled.

The structure provides utility functions to retrieve the texture's properties, such as width, height, etc., and it has methods to bind (`Bind()`) and unbind (`Unbind()`) the texture in the OpenGL context. The `Use()` method

sets the active texture unit and binds the texture to a specified uniform in the shader for rendering.

Material Class Update

You can update your material by adding additional fields, as depicted in (Listing 6.34). These fields are simply representing our PBR textures.

Listing 6.34: Graphics/Utilities/Data. h

```
struct PbrMaterial
{
    EMPY_INLINE PbrMaterial(const PbrMaterial&) = default;
    EMPY_INLINE PbrMaterial() = default;

    glm::vec3 Emissive = glm::vec3(0.0f);
    glm::vec3 Albedo = glm::vec3(1.0f);
    float Occlusion = 1.0f;
    float Roughness = 0.4f;
    float Metallic = 0.5f;

    Texture OcclusionMap;
    Texture RoughnessMap;
    Texture MetallicMap;
    Texture EmissiveMap;
    Texture AlbedoMap;
    Texture NormalMap;
};
```

PBR Shader Update

Since these textures must be sent to the shader for further processing and rendering, we also need to update the material structure in the fragment shader and update the main as well as the compute light functions accordingly. See (Listing 6.35). For now, we are only doing that for the albedo map. The vertex shader remains unchanged.

Listing 6.35: Resources/Shaders/pbr.gls

```
// material type
struct Material
{
    sampler2D OcclusionMap;
    sampler2D RoughnessMap;
    sampler2D EmissiveMap;
    sampler2D MetallicMap;
    sampler2D AlbedoMap;
    sampler2D NormalMap;

    float Roughness;
    float Occlusion;
    float Metallic;
    vec3 Emissive;
    vec3 Albedo;
};

/* ... same as before ... */

// compute direct lights
vec3 ComputeDirectLights(vec3 N, vec3 V, vec3 F0, vec3
albedo)
{
```

```

vec3 result = vec3(0.0);

for (int i = 0; i < u_nbrDirectLight; ++i)
{
    // compute parameters
    vec3 L = -normalize(u_directLights[i].Direction);
    float NdotL = max(dot(N, L), 0.0);
    float NdotV = max(dot(N, V), 0.0);
    vec3 H = normalize(L + V);

    // Cook-Torrance (BRDF)
    float NDF = DistributionGGX(N, H, u_material.Roughness);
    vec3 FS = FresnelSchlick(clamp(dot(H, V), 0.0, 1.0),
F0);
    float GS = GeometrySmithGGX(NdotV, NdotL,
u_material.Roughness);

    // diffuse light
    vec3 diffuse = (vec3(1.0) - FS) * (1.0 -
u_material.Metallic);

    // specular light
    vec3 specular = (NDF * GS * FS) / max(4.0 * NdotV *
NdotL, 0.0001);

    // combine components
    result += (diffuse * albedo / PI + specular) *
u_directLights[i].Radiance *
NdotL * u_directLights[i].Intensity;

    // break if max light
    if (i >= MAX_LIGHTS - 1) { break; }
}

return result;
}

```

```

// compute point lights
vec3 ComputePointLights(vec3 N, vec3 V, vec3 F0, vec3
albedo)
{
    vec3 result = vec3(0.0);

    for (int i = 0; i < u_nbrSpotLight; ++i)
    {
        // compute parameters
        vec3 L = normalize(u_pointLights[i].Position -
vertex.Position);
        float NdotL = max(dot(N, L), 0.0);
        float NdotV = max(dot(N, V), 0.0);
        vec3 H = normalize(L + V);

        // Cook-Torrance (BRDF)
        float NDF = DistributionGGX(N, H, u_material.Roughness);
        vec3 FS = FresnelSchlick(clamp(dot(H, V), 0.0, 1.0),
F0);
        float GS = GeometrySmithGGX(NdotV, NdotL,
u_material.Roughness);

        // diffuse light
        vec3 diffuse = (vec3(1.0) - FS) * (1.0 -
u_material.Metallic);

        // specular light
        vec3 specular = (NDF * GS * FS) / max(4.0 * NdotV *
NdotL, 0.0001);

        // light attenuation
        float distance = length(u_pointLights[i].Position -
vertex.Position);
        float attenuation = u_pointLights[i].Intensity /
(distance * distance);
    }
}

```

```

    // combine components
    result += (diffuse * albedo / PI + specular) *
u_pointLights[i].Radiance *
    attenuation * NdotL;

    // break if max light
    if (i >= MAX_LIGHTS - 1) { break; }
}

return result;
}

// compute spot lights
vec3 ComputeSpotLights(vec3 N, vec3 V, vec3 F0, vec3 albedo)
{
    vec3 result = vec3(0.0);

    for (int i = 0; i < u_nbrSpotLight; ++i)
    {
        // compute parameters
        vec3 L = normalize(u_spotLights[i].Position -
vertex.Position);
        float NdotL = max(dot(N, L), 0.0);
        float NdotV = max(dot(N, V), 0.0);
        vec3 H = normalize(L + V);

        // Cook-Torrance (BRDF)
        float NDF = DistributionGGX(N, H, u_material.Roughness);
        vec3 FS = FresnelSchlick(clamp(dot(H, V), 0.0, 1.0),
F0);
        float GS = GeometrySmithGGX(NdotV, NdotL,
u_material.Roughness);

        // diffuse light
        vec3 diffuse = (vec3(1.0) - FS) * (1.0 -

```

```

u_material.Metallic);

    // specular light
    vec3 specular = (NDF * GS * FS) / max(4.0 * NdotV *
NdotL, 0.0001);

    // compute spot
    float theta = dot(L, normalize(-
u_spotLights[i].Direction));
    float epsilon = (u_spotLights[i].Falloff -
u_spotLights[i].CutOff);
    float spotFactor = clamp((theta -
u_spotLights[i].CutOff)/epsilon, 0.0, 1.0);

    // light attenuation
    float distance = length(u_spotLights[i].Position -
vertex.Position);
    float attenuation = u_spotLights[i].Intensity / (distance *
distance);

    // combine components
    result += (diffuse * albedo / PI + specular) *
u_spotLights[i].Radiance *
    u_spotLights[i].Intensity * attenuation * NdotL *
spotFactor;

    // break if max light
    if (i >= MAX_LIGHTS - 1) { break; }
}

return result;
}

// main function
void main()
{

```

```

// normalized surface normal
vec3 N = normalize(vertex.Normal);

// camera view direction
vec3 V = normalize(u_viewPos - vertex.Position);

// material albedo + albedo-map <----- here the only
change
vec3 albedo = (u_material.Albedo +
texture(u_material.AlbedoMap, vertex.UVs).rgb);

// fresnel base reflectivity
vec3 F0 = mix(vec3(0.04), albedo, u_material.Metallic);

// point lights contribution
vec3 result = ComputeDirectLights(N, V, F0, albedo);
result += ComputePointLights(N, V, F0, albedo);
result += ComputeSpotLights(N, V, F0, albedo);

// final color calculation
out_fragment = vec4(result, 1.0);
}

++FRAGMENT++

```

You will notice in the `Material` structure that the texture type is `sampler2D`. As introduced earlier, it represents the unit or slot where the texture is located on the GPU. You can also see in the main function how the albedo component of the material is now a combination of color (`Albedo`) and texture (`AlbedoMap`).

The result is then provided to our compute light functions. What we have simply done in the compute light functions is just replace the `u_material.Albedo` with the newly function parameter. The same thing will be done for normal, metallic, and roughness as well.

Shader Class Update

We now need to load these texture uniform locations in the C++ shader class so we can properly apply them before rendering the model. You can see how that is done in the shader class constructor in ([Listing 6.36](#)).

Additionally, we use these uniform locations in the `Draw()` function to set the texture unit. The texture unit is represented by an integer, which is incremented for every subsequent texture used. The rest of this class remains unchanged.

Listing 6.36: Graphics/Shaders/PBR. h

```
namespace Empy
{
    struct PbrShader : Shader
    {
        EMPY_INLINE PbrShader(const std::string& filename) :
        Shader(filename)
        {
    
```

```

        u_RoughnessMap = glGetUniformLocation(m_ShaderID,
"u_material.RoughnessMap");
        u_OcclusionMap = glGetUniformLocation(m_ShaderID,
"u_material.OcclusionMap");
        u_EmissiveMap = glGetUniformLocation(m_ShaderID,
"u_material.EmissiveMap");
        u_MetallicMap = glGetUniformLocation(m_ShaderID,
"u_material.MetallicMap");
        u_AlbedoMap = glGetUniformLocation(m_ShaderID,
"u_material.AlbedoMap");
        u_NormalMap = glGetUniformLocation(m_ShaderID,
"u_material.NormalMap");

        u_Roughness = glGetUniformLocation(m_ShaderID,
"u_material.Roughness");
        u_Occlusion = glGetUniformLocation(m_ShaderID,
"u_material.Occlusion");
        u_Emissive = glGetUniformLocation(m_ShaderID,
"u_material.Emissive");
        u_Metallic = glGetUniformLocation(m_ShaderID,
"u_material.Metallic");
        u_Albedo = glGetUniformLocation(m_ShaderID,
"u_material.Albedo");

        //... same as before
    }

EMPTY_INLINE void Draw(Model3D& model, PbrMaterial&
material, Transform3D& transform)
{
    glUniformMatrix4fv(u_Model, 1, GL_FALSE,
glm::value_ptr(transform.Matrix()));

    glUniform3fv(u_Emissive, 1,
&material.Emissive.x);
    glUniform3fv(u_Albedo, 1, &material.Albedo.x);
}

```

```

        glUniform1f(u_Roughness, material.Roughness);
        glUniform1f(u_Metallic, material.Occlusion);
        glUniform1f(u_Metallic, material.Metallic);

        // material maps
        int32_t unit = 0;

        if(material.AlbedoMap) { material.AlbedoMap-
>Use(u_AlbedoMap, unit++); }
        if(material.NormalMap) { material.NormalMap-
>Use(u_NormalMap, unit++); }
        if(material.MetallicMap) { material.MetallicMap-
>Use(u_MetallicMap, unit++); }
        if(material.EmissiveMap) { material.EmissiveMap-
>Use(u_EmissiveMap, unit++); }
        if(material.OcclusionMap) {
material.OcclusionMap->Use(u_OcclusionMap, unit++); }
        if(material.RoughnessMap) {
material.RoughnessMap->Use(u_RoughnessMap, unit++); }

        // render model
        model->Draw(GL_TRIANGLES);
    }

    // ...

private:
    uint32_t u_NbrDirectLight = 0u;
    uint32_t u_NbrPointLight = 0u;
    uint32_t u_NbrSpotLight = 0u;
    // --
    uint32_t u_RoughnessMap = 0u;
    uint32_t u_OcclusionMap = 0u;
    uint32_t u_EmissiveMap = 0u;
    uint32_t u_MetallicMap = 0u;
    uint32_t u_AlbedoMap = 0u;

```

```

        uint32_t u_NormalMap = 0u;
        // --
        uint32_t u_Roughness = 0u;
        uint32_t u_Occlusion = 0u;
        uint32_t u_Emissive = 0u;
        uint32_t u_Metallic = 0u;
        uint32_t u_Albedo = 0u;
        //--
        uint32_t u_ViewPos = 0u;
        uint32_t u_Model = 0u;
        uint32_t u_View = 0u;
        uint32_t u_Proj = 0u;
    };
}

```

Material Albedo

Let us now try this in the `Application` class. Load a texture from any image file, create a new entity with a model component, and make sure to set the material albedo map as depicted below.

Listing 6.37: Applications/Applications.h

```

EMPY_INLINE void RunContext()
{
    // load textures
    auto roughness = std::make_shared<Texture2D>
("Resources/Textures/Marble/Roughness.jpg");
    auto albedo = std::make_shared<Texture2D>
("Resources/Textures/Marble/Albedo.jpg");
    auto normal = std::make_shared<Texture2D>

```

```
("Resources/Textures/Marble/Normal.jpg");

    // load models
    auto sphereModel = std::make_shared<Model>
("Resources/Models/sphere.fbx");
    auto cubeModel = std::make_shared<Model>
("Resources/Models/cube.fbx");

    // create scene camera
    auto camera = CreateEntt<Entity>();
    camera.Attach<TransformComponent>().Transform.Translate.z
= 3.0f;
    camera.Attach<CameraComponent>();

    // create point light 1
    auto slight = CreateEntt<Entity>();
    slight.Attach<DirectLightComponent>();
    auto& stp = slight.Attach<TransformComponent>
().Transform;
    stp.Rotation = glm::vec3(0.0f, 0.0f, -1.0f);
    stp.Translate.z = 1.0f;

    // create cube entity
    auto cube = CreateEntt<Entity>();
    cube.Attach<TransformComponent>().Transform.Scale *=
2.5f;
    auto& mod = cube.Attach<ModelComponent>();
    mod.Model = sphereModel;

    mod.Material.Albedo = glm::vec3(0.0f);
    mod.Material.Metallic = 0.25f;
    mod.Material.Roughness = 0.1f;

    mod.Material.RoughnessMap = roughness; // <--- roughness
map
    mod.Material.AlbedoMap = albedo;           // <--- albedo
```

```
map
    mod.Material.NormalMap = normal;           // <--- normal
map

    // the rest remains unchanged!
}
```

We are focusing on the albedo map only, which requires only one texture. However, to prepare for upcoming changes involving other material properties, we are also loading additional textures preemptively. You will also notice that we set the albedo color of the material to zero, because this will be added to the color sample from the texture, which will not look the same. We also have to do the same thing for the roughness and metallic properties when their textures are added.

You will find these textures in the GitHub repository, or if needed, you can access free PBR textures from this site. [Lennart](#).

(Figure [6.16](#)) is the result of the addition of the albedo map. You can play with the roughness and metallic properties to change the look of the object.

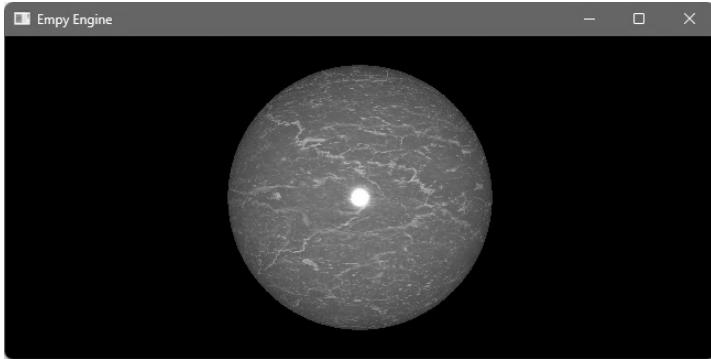


Figure 6.16: Material Albedo

Material Roughness

Since we have already loaded the roughness map, we can now simply update our shader to use it for rendering. In the fragment shader in (Listing 6.38) we have added a new 'roughness' argument to the compute light functions, which is calculated in the main function.

Listing 6.38: Resource/Shaders/pbr.gls
|

```
/* ... same as before ... */

vec3 ComputeDirectLights(vec3 N, vec3 V, vec3 F0, vec3
albedo, float roughness)
{
    // ---> replace all "u_material.Roughness" with
    "roughness"
```

```

}

vec3 ComputePointLights(vec3 N, vec3 V, vec3 F0, vec3
albedo, float roughness)
{
    // ---> replace all "u_material.Roughness" with
"roughness"
}

vec3 ComputeSpotLights(vec3 N, vec3 V, vec3 F0, vec3 albedo,
float roughness)
{
    // ---> replace all "u_material.Roughness" with
"roughness"
}

// main function
void main()
{
    // normalized surface normal
    vec3 N = normalize(vertex.Normal);

    // camera view direction
    vec3 V = normalize(u_viewPos - vertex.Position);

    // material albedo + albedo-map
    vec3 albedo = (u_material.Albedo +
texture(u_material.AlbedoMap, vertex.UVs).rgb);

    // material roughness + roughness-map <----- Here is the
change
    float roughness = (u_material.Roughness +
texture(u_material.RoughnessMap, vertex.UVs).r);

    // fresnel base reflectivity
    vec3 F0 = mix(vec3(0.04), albedo, u_material.Metallic);
}

```

```
// point lights contribution
vec3 result = ComputeDirectLights(N, V, F0, albedo,
roughness);
result += ComputePointLights(N, V, F0, albedo, roughness);
result += ComputeSpotLights(N, V, F0, albedo, roughness);

// final color calculation
out_fragment = vec4(result, 1.0);
}

++FRAGMENT++
```

If you look closely, you will notice that we are only using the "r" component of the color sampled from the roughness map. Roughness maps are monochromatic maps, providing only values between black and white.

Adding the roughness factor with the sampled roughness value is dangerous because this will most likely end up with a roughness values outside of the range [0, 1]. Do not worry, this will be fixed later.

(Figure [6.17](#)) is the result of the addition of the roughness map.

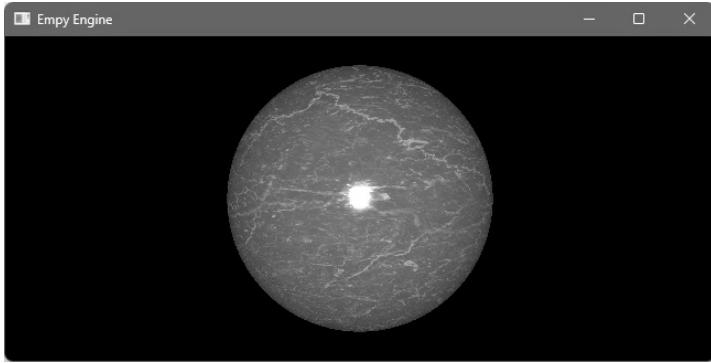


Figure 6.17: Material Roughness Map

Material Metallic

The metallic property just as the roughness property can be sample from a metallic map and provided to the functions to compute lighting effects as depicted in the following code snippet:

Listing 6.39: Resource/Shaders/pbr.gls

```
/* ... same as before ... */

vec3 ComputeDirectLights(vec3 N, vec3 V, vec3 F0, vec3
albedo, float roughness, float metallic)
{
    // ---> replace all "u_material.Metallic" with
"metallic"
}
```

```
vec3 ComputePointLights(vec3 N, vec3 V, vec3 F0, vec3
albedo, float roughness, float metallic)
{
    // ---> replace all "u_material.Metallic" with
"metallic"
}

vec3 ComputeSpotLights(vec3 N, vec3 V, vec3 F0, vec3 albedo,
float roughness, float metallic)
{
    // ---> replace all "u_material.Metallic" with
"metallic"
}

// main function
void main()
{
    // normalized surface normal
    vec3 N = normalize(vertex.Normal);

    // camera view direction
    vec3 V = normalize(u_viewPos - vertex.Position);

    // material albedo
    vec3 albedo = (u_material.Albedo +
texture(u_material.AlbedoMap, vertex.UVs).rgb);

    // material metallness [0, 1]
    float metallic = (u_material.Metallic +
texture(u_material.MetallicMap, vertex.UVs).r);

    // material roughness [0, 1]
    float roughness = (u_material.Roughness +
texture(u_material.RoughnessMap, vertex.UVs).r);

    // fresnel base reflectivity
    vec3 F0 = mix(vec3(0.04), albedo, metallic); // <---
```

```

metallic used

    // point lights contribution
    vec3 result = ComputeDirectLights(N, V, F0, albedo,
roughness, metallic);
    result += ComputePointLights(N, V, F0, albedo, roughness,
metallic);
    result += ComputeSpotLights(N, V, F0, albedo, roughness,
metallic);

    // final color calculation
    out_fragment = vec4(result, 1.0);
}

++FRAGMENT++

```

(Figure [6.18](#)) is the result you get with a metallic value equal "0.8". you can set different values to see how this affects the output.

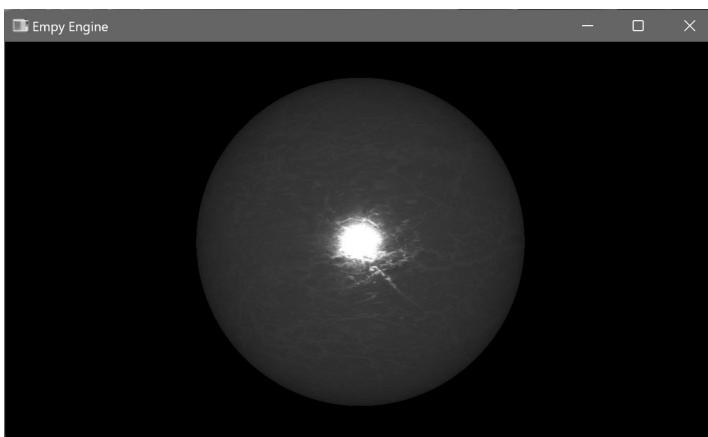


Figure 6.18: Material Metallic ($M = 0.8f$)

Material Emissive

We stated earlier, while describing the different texture types involved in a PBR material, that the emissive map describes the ability of an object to emit light. To add this to our pipeline, we simply need to add the sampled emissive color or use the "Emissive" field of the material to the computed fragment result. This is depicted in the following code:

Listing 6.40: Resource/Shaders/pbr.gls
|

```
/* ... same as before ... */

// main function
void main()
{
    // normalized surface normal
    vec3 N = normalize(vertex.Normal);

    // camera view direction
    vec3 V = normalize(u_viewPos - vertex.Position);

    // material albedo
    vec3 albedo = (u_material.Albedo +
texture(u_material.AlbedoMap, vertex.UVs).rgb);

    // material metallness [0, 1]
    float metallic = (u_material.Metallic +
texture(u_material.MetallicMap, vertex.UVs).r);

    // material roughness [0, 1]
```

```

float roughness = (u_material.Roughness +
texture(u_material.RoughnessMap, vertex.UVs).r);

// material emissive
vec3 emissive = (u_material.Emissive +
texture(u_material.EmissiveMap, vertex.UVs).rgb);

// fresnel base reflectivity
vec3 F0 = mix(vec3(0.04), albedo, metallic);

// point lights contribution
vec3 result = ComputeDirectLights(N, V, F0, albedo,
roughness, metallic);
result += ComputePointLights(N, V, F0, albedo, roughness,
metallic);
result += ComputeSpotLights(N, V, F0, albedo, roughness,
metallic);

// adding emissive value
result += emissive;

// final color calculation
out_fragment = vec4(result, 1.0);
}

++FRAGMENT++

```

This will be crucial later when implementing glowing or bloom effect. The marble material we are using does not have an emissive map but you can use the emissive color instead. You will get a result similar to the one depicted in (Figure [6.19](#)).

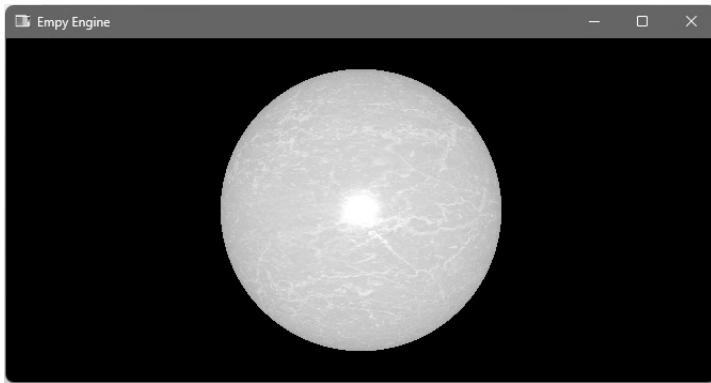


Figure 6.19: Material Emissive Color

Material Occlusion

certain models and certain parts of models are often more occluded than others. To capture this behaviour, PBR uses an ambient occlusion map. this can be added in the pipeline by just multiplying the computed fragment color with the value sampled from the occlusion map. This is done in the following code.

Listing 6.41: Resource/Shaders/pbr.gls
|

```
/* ... same as before ... */

// main function
void main()
{
    // normalized surface normal
    vec3 N = normalize(vertex.Normal);
```

```
// camera view direction
vec3 V = normalize(u_viewPos - vertex.Position);

// material albedo
vec3 albedo = (u_material.Albedo +
texture(u_material.AlbedoMap, vertex.UVs).rgb);

// material emissivness
vec3 emissive = (u_material.Emissive +
texture(u_material.EmissiveMap, vertex.UVs).rgb);

// material metallness [0, 1]
float metallic = (u_material.Metallic +
texture(u_material.MettalicMap, vertex.UVs).r);

// material roughness [0, 1]
float roughness = (u_material.Roughness +
texture(u_material.RoughnessMap, vertex.UVs).r);

// material occlusion [0, 1]
float occlusion = (u_material.Occlusion +
texture(u_material.OcclusionMap, vertex.UVs).r);

// fresnel base reflectivity
vec3 F0 = mix(vec3(0.04), albedo, metallic);

// point lights contribution
vec3 result = ComputeDirectLights(N, V, F0, albedo,
roughness, metallic);
    result += ComputePointLights(N, V, F0, albedo, roughness,
metallic);
    result += ComputeSpotLights(N, V, F0, albedo, roughness,
metallic);

// occlusion and emissive
```

```
    result = (result * occlusion) + emissive;

    // final color calculation
    out_fragment = vec4(result, 1.0);
}

++FRAGMENT++
```

Set the value of the occlusion factor in your material to any value within the range [0, 1] and you will see how your sphere will be occluded or darkened. You can also download additional PBR materials with ambient occlusion map from [Lennart](#) to see the effect on the result.

6.4.3 Normal Mapping

Similar to the roughness and the albedo maps, the normal map is also already loaded and ready to use. But sampling from a normal map is a bit more complicated than the other texture types. There are certain things we need to address before we can see the result.

Tangent and Bitangent

The tangent (T) and bitangent (B) vectors, along with the surface normal (N), form the basis of the Tangent Space Normal Mapping technique, which allows for

more detailed and accurate representations of surfaces in 3D rendering. See (Figure [6.20](#))

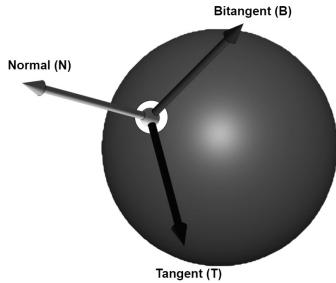


Figure 6.20:

Normal, Tangent and Bitangent

- ☞ **Surface Normal (N):** The surface normal is a vector perpendicular to the surface at each point. It represents the direction the surface is facing.
- ☞ **Tangent (T):** This vector lies in the tangent plane of the surface, aligned with the direction of the texture coordinates.
- ☞ **Bitangent (B):** This vector also lies in the tangent plane but is perpendicular to both the surface normal and the tangent. It's aligned with the V direction of the texture coordinates.

The combination of these three vectors (Tangent, Bitangent, and Normal) forms a local coordinate system known as the "**Tangent Space**". When using

normal maps, which encode surface normals in textures, the tangent and bitangent vectors are crucial in transforming the sampled normal from the texture space (where the normals are encoded) to the world or view space (where lighting calculations occur). This transformation ensures that the sampled normals align correctly with the geometry of the 3D model, even when the model is deformed or animated.

Steps to Use Tangent Space

Here's an overview of how tangent and bitangent vectors are used in normal mapping:

- ☞ **Vertex Attributes:** Tangent and bitangent vectors are often calculated and stored as vertex attributes along with the surface normals. These attributes are typically computed when the 3D model is loaded or generated in the shader.
- ☞ **Texture Space to Tangent Space:** When sampling the normal map texture in the shader, we use the tangent and bitangent vectors to transform the sampled color values from texture space to the tangent space of the surface at each fragment or pixel.
- ☞ **Transforming to World/View Space:** Once the normal is in tangent space, it's transformed into world

or view space using the tangent, bitangent, and normal vectors calculated for the specific point on the surface. This transformation ensures that the normal is correctly aligned with the rest of the surface geometry, allowing for accurate lighting calculations.

Let us add this to our rendering pipeline. You can use the following references if you are interested in understanding the mathematical and technical details behind this concept: [de Vries \[c\] opengl tutorial](#).

Vertex Class Update

We need to add the tangent and the bitangent attributes to the `ShadedVertex` structure so we can set their value when loading the model with Assimp.

Listing 6.42: Graphics/Buffers/Vertex.h

```
struct ShadedVertex
{
    glm::vec3 Position = glm::vec3(0.0f);
    glm::vec3 Normal = glm::vec3(0.0f);
    glm::vec2 UVs = glm::vec2(0.0f);

    glm::vec3 Tangent = glm::vec3(0.0f);    // <--- added
    glm::vec3 Bitangent = glm::vec3(0.0f); // <--- added
};
```

We could actually only use the *Tangent* attribute. Because the *Tangent*, the *Bitangent* and the *Normal* are orthogonal, we only need two to compute the third. But let us keep it simple.

Mesh Class Update

We then have to make sure the mesh class is updated accordingly. The only change required is found in the constructor of the mesh class.

Listing 6.43: Graphics/Buffers/Mesh.h

```
template <typename Vertex> struct Mesh
{
    EMPY_INLINE Mesh(const MeshData<Vertex>& data)
    {
        // ... remain unchanged

        // handle vertex types
        if (TypeID<Vertex>() == TypeID<ShadedVertex>())
        {
            SetAttribute(0, 3, (void*)offsetof(ShadedVertex,
Position));
            SetAttribute(1, 3, (void*)offsetof(ShadedVertex,
Normal));
            SetAttribute(2, 2, (void*)offsetof(ShadedVertex,
UVs));
            // tangent and bitangent
            SetAttribute(3, 3, (void*)offsetof(ShadedVertex,
Tangent));
            SetAttribute(4, 3, (void*)offsetof(ShadedVertex,
```

```

Bitangent));
}

// ... remain unchanged
}

// ... remain unchanged
}

```

Model Class Update

We then have to make sure that these attributes are set when loading the model. As you can see in the following code snippet, we only need to update the `ParseMesh()` function to make that possible.

Listing 6.44: Graphics/Models/Model.h

```

EMPY_INLINE void ParseMesh(aiMesh* ai_mesh)
{
    // mesh data
    MeshData<ShadedVertex> data;

    // vertices
    for (uint32_t i = 0; i < ai_mesh->mNumVertices; i++)
    {
        ShadedVertex vertex;
        // positions
        vertex.Position = AssimpToVec3(ai_mesh-
>mVertices[i]);
        // normals
        vertex.Normal = AssimpToVec3(ai_mesh->mNormals[i]);

```

```

    // texcoords
    vertex.UVs.x = ai_mesh->mTextureCoords[0][i].x;
    vertex.UVs.y = ai_mesh->mTextureCoords[0][i].y;

    // bi-tangent <--- added
    vertex.Bitangent =
glm::normalize(AssimpToVec3(ai_mesh->mBitangents[i]));
    vertex.Tangent = glm::normalize(AssimpToVec3(ai_mesh-
>mTangents[i]));

    // push vertex
    data.Vertices.push_back(vertex);
}

// indices
for (uint32_t i = 0; i < ai_mesh->mNumFaces; i++)
{
    for (uint32_t k = 0; k < ai_mesh-
>mFaces[i].mNumIndices; k++)
    {
        data.Indices.push_back(ai_mesh-
>mFaces[i].mIndices[k]);
    }
}

// create new mesh instance
m_Meshes.push_back(std::make_unique<ShadedMesh>(data));
}

```

If you compile and run your code, you should get the same result as before.

PBR Shader Update

We also want to change the PBR vertex shader to use this newly added attributes. (Listing 6.45) show the updated version of the vertex shader. As you can see we have added the new attributes to the layout. We have also added a new variable to the output vertex called `TBN`. This matrix will help transform our normals from texture space to world space in the fragment shader. This matrix is computed using the tangent, bitangent and the normal and the model matrix to ensure consistency even when the object is moved or animated.

Listing 6.45: Resources/Shaders/pbr.gls
|

```
#version 330 core

layout (location = 0) in vec3 a_position;
layout (location = 1) in vec3 a_normal;
layout (location = 2) in vec2 a_uvs;
layout (location = 3) in vec3 a_tangent; // <-- added
layout (location = 4) in vec3 a_bitangent; // <-- added

out Vertex
{
    vec3 Position;
    vec3 Normal;
    mat3 TBN; // <-- added
    vec2 UVs;
} vertex;

uniform mat4 u_model;
uniform mat4 u_proj;
uniform mat4 u_view;
```

```

void main()
{
    vertex.UVs = a_uvs;
    vertex.Normal = (u_model * vec4(a_normal, 1.0)).xyz;
    vertex.Position = (u_model * vec4(a_position, 1.0)).xyz;
    gl_Position = u_proj * u_view * u_model * vec4(a_position,
1.0);
    vertex.TBN = mat3(u_model) * mat3(a_tangent, a_bitangent,
a_normal);
}
++VERTEX++

```

The fragment shader (Listing 6.46), on the other hand, has a couple of changes we need to address. First of all, we have the `Material` that now offers some additional variables to tell if the material maps should be used or not. This is important because we don't want to always manually set the albedo color to zero every time we want to use the albedo map instead.

Listing 6.46: Resources/Shaders/pbr.gls

```

#version 330 core

/* ... same as before ... */

// material type
struct Material
{
    sampler2D RoughnessMap;
    bool UseRoughnessMap;
}

```

```
sampler2D OcclusionMap;
bool UseOcclusionMap;

sampler2D EmissiveMap;
bool UseEmissiveMap;

sampler2D MetallicMap;
bool UseMetallicMap;

sampler2D NormalMap;
bool UseNormalMap;

sampler2D AlbedoMap;
bool UseAlbedoMap;

float Occlusion;
float Roughness;
float Metallic;
vec3 Albedo;
};

// vertex input
in Vertex
{
    vec3 Position;
    vec3 Normal;
    mat3 TBN; // <-- added
    vec2 UVs;
} vertex;

/* ... same as before ... */

// main function
void main()
{
    // camera view direction
```

```
vec3 V = normalize(u_viewPos - vertex.Position);

// surface normal <--- added
vec3 N = normalize(vertex.Normal);
if(u_material.UseNormalMap)
{
    // convert from [0,1] range to [-1, 1] range
    N = 2.0 * texture(u_material.NormalMap, vertex.UVs).rgb
- 1.0;
    N = normalize(vertex.TBN * N);
}

// material roughness
float roughness = u_material.Roughness;
if(u_material.UseRoughnessMap)
{
    roughness = texture(u_material.RoughnessMap,
vertex.UVs).r;
}

// material occlusion
float occlusion = u_material.Occlusion;
if(u_material.UseOcclusionMap)
{
    occlusion = texture(u_material.OcclusionMap,
vertex.UVs).r;
}

// material metallic
float metallic = u_material.Metallic;
if(u_material.UseMetallicMap)
{
    metallic = texture(u_material.MetallicMap, vertex.UVs).r;
}

// material emissivness
```

```
vec3 emissive = u_material.Emissive;
if(u_material.UseEmissiveMap)
{
    emissive = texture(u_material.EmissiveMap,
vertex.UVs).rgb;
}

// material albedo
vec3 albedo = u_material.Albedo;
if(u_material.UseAlbedoMap)
{
    albedo = texture(u_material.AlbedoMap, vertex.UVs).rgb;
}

// fresnel reflectivity
vec3 F0 = mix(vec3(0.04), albedo, metallic);

// lights contribution
vec3 result = ComputeDirectLights(N, V, F0, albedo,
roughness, metallic);
result += ComputePointLights(N, V, F0, albedo, roughness,
metallic);
result += ComputeSpotLights(N, V, F0, albedo, roughness,
metallic);

// occlusion and emissive
result = (result * occlusion) + emissive;

// final color calculation
out_fragment = vec4(result, 1.0);
}

++FRAGMENT++
```

We will simply check if the material has an albedo map for example and set the `UseAlbedoMap` uniform to true. Secondly, you will notice the `Vertex` input that also defines the `TBN` matrix introduced previously in the vertex shader. Finally, we wrap things up in the main function where we simply check if each map is available so we can use it to compute the material property as well as the normal vector.

Shader Class Update

We then require to get the uniform location of the materials field from the shader in our C++ shader class and use them accordingly.

Listing 6.47: Graphics/Shaders/PBR.
h

```
struct PbrShader : Shader
{
    EMPLY_INLINE PbrShader(const std::string& filename) :
    Shader(filename)
    {
        u_UseRoughnessMap = glGetUniformLocation(m_ShaderID,
"u_material.UseRoughnessMap");
        u_UseOcclusionMap = glGetUniformLocation(m_ShaderID,
"u_material.UseOcclusionMap");
        u_UseEmissiveMap = glGetUniformLocation(m_ShaderID,
"u_material.UseEmissiveMap");
        u_UseMetallicMap = glGetUniformLocation(m_ShaderID,
"u_material.UseMetallicMap");
    }
}
```

```

        u_UseAlbedoMap = glGetUniformLocation(m_ShaderID,
"u_material.UseAlbedoMap");
        u_UseNormalMap = glGetUniformLocation(m_ShaderID,
"u_material.UseNormalMap");

        /* ... same as before ... */
    }

EMPY_INLINE void Draw(Model3D& model, PbrMaterial&
material, Transform3D& transform)
{
    glUniformMatrix4fv(u_Model, 1, GL_FALSE,
glm::value_ptr(transform.Matrix()));

    glUniform3fv(u_Albedo, 1, &material.Albedo.x);
    glUniform1f(u_Roughness, material.Roughness);
    glUniform1f(u_Metallic, material.Metallic);

    // material maps

    int32_t unit = 0;
    bool useMap = false;

    // albedo map
    useMap = material.AlbedoMap != nullptr;
    glUniform1i(u_UseAlbedoMap, useMap);
    if(useMap) { material.AlbedoMap->Use(u_AlbedoMap,
unit++); }

    // normal map
    useMap = material.NormalMap != nullptr;
    glUniform1i(u_UseNormalMap, useMap);
    if(useMap) { material.NormalMap->Use(u_NormalMap,
unit++); }

    // mettallic map
}

```

```

        useMap = material.MetallicMap != nullptr;
        glUniform1i(u_UseMetallicMap, useMap);
        if(useMap) { material.MetallicMap->Use(u_MetallicMap,
unit++); }

        // emissive map
        useMap = material.EmissiveMap != nullptr;
        glUniform1i(u_UseEmissiveMap, useMap);
        if(useMap) { material.EmissiveMap->Use(u_EmissiveMap,
unit++); }

        // occlusion map
        useMap = material.OcclusionMap != nullptr;
        glUniform1i(u_UseOcclusionMap, useMap);
        if(useMap) { material.OcclusionMap-
>Use(u_OcclusionMap, unit++); }

        // roughness map
        useMap = material.RoughnessMap != nullptr;
        glUniform1i(u_UseRoughnessMap, useMap);
        if(useMap) { material.RoughnessMap->Use(u_RoughnessMap,
unit++); }

        // render model
        model->Draw(GL_TRIANGLES);
    }

private:
    uint32_t u_UseRoughnessMap = 0u;
    uint32_t u_UseOcclusionMap = 0u;
    uint32_t u_UseEmissiveMap = 0u;
    uint32_t u_UseMetallicMap = 0u;
    uint32_t u_UseAlbedoMap = 0u;
    uint32_t u_UseNormalMap = 0u;
    // --
    uint32_t u_RoughnessMap = 0u;

```

```
uint32_t u_OcclusionMap = 0u;
uint32_t u_EmissiveMap = 0u;
uint32_t u_MetallicMap = 0u;
uint32_t u_AlbedoMap = 0u;
uint32_t u_NormalMap = 0u;
// --
uint32_t u_Roughness = 0u;
uint32_t u_Occlusion = 0u;
uint32_t u_Emissive = 0u;
uint32_t u_Metallic = 0u;
uint32_t u_Albedo = 0u;

/* ... same as before ... */
}
```

If you compile and run this version of the code you will get the following result:

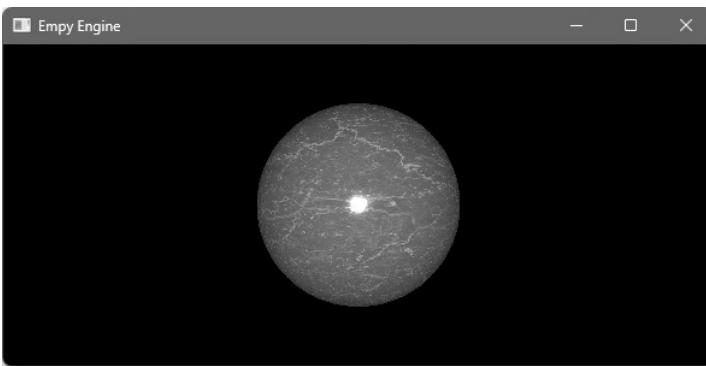


Figure 6.21: Albedo, Roughness and Normal Map

There are other additional PBR-material textures such as the ambient occlusion map, the height or displacement map, the emissive map for bloom effect

that can be used to add more layers of realism to the scene but we will just stick with these now.

6.5 Global Illumination

The interaction between an object and its environment profoundly influences its appearance and believability within a scene. This dynamic behavior creates a sense of cohesion and realism, crucial for immersive experiences in computer graphics and rendering. When an object is placed within a specific environment, various environmental factors influence its appearance. Light bounces off surfaces, reflects from nearby objects, and gets absorbed or scattered based on the material properties. Shadows, reflections, and ambient light contribute to the object's visual context. Accurate representation of an object's interaction with its environment leads to enhanced realism and immersion (Figure [6.22](#)).



Figure 6.22: Global Illumination

Environment lighting refers to the use of the surrounding environment (like the skybox or other environmental maps) to simulate how light interacts with objects. In particular, it's used for global illumination techniques in rendering, such as Image-Based Lighting (IBL). Much like the various light sources discussed earlier, environment lighting also consists of both diffuse and specular components. The diffuse aspect is attained through the creation of a **"Diffuse Irradiance"** map generated from the environment cube map. See ([Figure 6.23](#)).

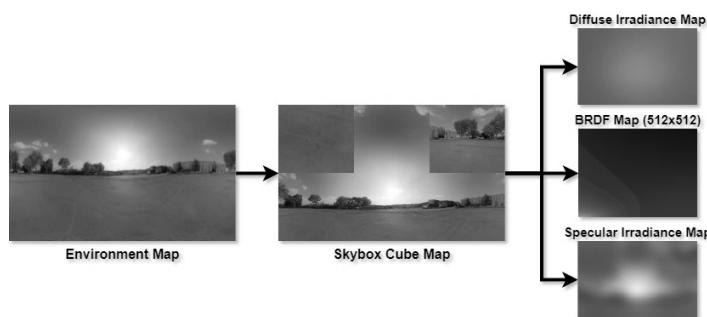


Figure 6.23:

Image-Based Lighting Components

This figure provides a high-level description of the steps we will undertake to implement environment lighting. The first step involve generating a skybox from a High Dynamic Range (HDR) environment. Then we need to generate a diffuse and specular irradiance and BRDF maps from the environment cube map

which are then used in the PBR shader to compute ambient lighting.

6.5.1 Scene Skybox

A skybox is a technique used in computer graphics to simulate distant scenery or create an illusion of a distant background. It's typically implemented by rendering a cube around the scene, where each face of the cube has a texture representing different parts of the surrounding environment (sky, distant mountains, etc.) (Figure [6.24](#)). A skybox is often rendered separately from the rest of the scene, and it doesn't interact with the objects in the scene. It's usually drawn first and serves as a backdrop to the 3D environment, providing a sense of depth and atmosphere.

There is a specific type of texture known as "**cube map**" that are used to render skyboxes. A cube map is a texture used in computer graphics to represent an environment or surrounding scenery, typically in 3D applications. Instead of a flat 2D image, a cube map comprises six 2D images (or faces) arranged to form the six sides of a cube (Figure [6.24](#)), representing different perspectives of an environment as if it's viewed from a specific point.

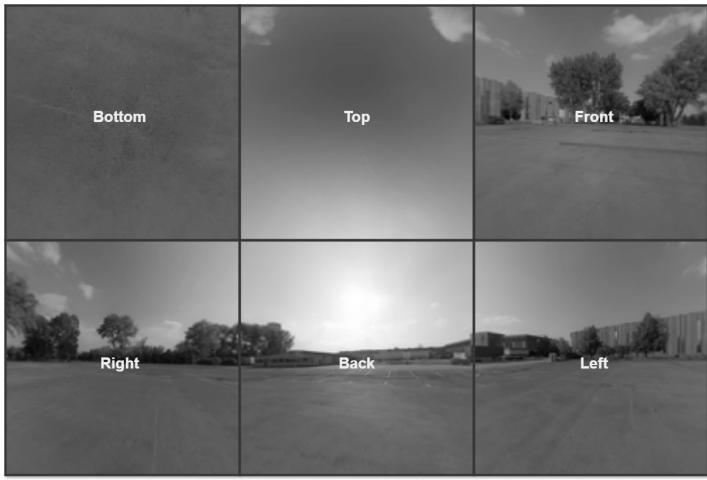


Figure 6.24: Cube Map Faces

The six faces of a cube map represent the view along the positive and negative directions of the x, y, and z axes. Each face corresponds to a specific direction: Positive X (right), Negative X (left), Positive Y (up), Negative Y (down), Positive Z (forward or front), Negative Z (backward or back).

In OpenGL or other graphics APIs, cube maps are created by assembling these six images into a specific texture format. They are then used in shaders to simulate reflections, environment lighting, or as background scenery by mapping different faces of the cube map to specific directions in the 3D space.

Skybox Mesh

In order to add a skybox to our scene, we need to have a cube or a box on which the texture of the cube map will be rendered. Therefore, we need a new type of vertex that will facilitate the creation of the box. We could also use a simple cube mesh to achieve this but it is not as efficient as having a dedicated mesh type.

Add the following (Listing [6.48](#)) vertex type to your **"Vertex.h"** header file.

Listing 6.48: Graphics/Buffers/Vertex.h

```
// ... same vertex

struct SkyboxVertex
{
    glm::vec3 Position = glm::vec3(0.0f);
};
```

The *SkyboxVertex* structure only has the position attribute. Because a skybox is simply a box rendered around the scene, it does not require additional attributes. Following (Listing [6.49](#)) is the updated version constructor of the mesh class with the addition of the skybox vertex.

Listing 6.49: Graphics/Buffers/Mesh.h

```
EMPTY_INLINE Mesh(const MeshData<Vertex>& data)
{
```

```

/* ... same as before ... */

// handle vertex types
if (TypeID<Vertex>() == TypeID<ShadedVertex>())
{
    SetAttribute(0, 3, (void*)offsetof(ShadedVertex,
Position));
    SetAttribute(1, 3, (void*)offsetof(ShadedVertex,
Normal));
    SetAttribute(2, 2, (void*)offsetof(ShadedVertex,
UVs));
    SetAttribute(3, 3, (void*)offsetof(ShadedVertex,
Tangent));
    SetAttribute(4, 3, (void*)offsetof(ShadedVertex,
Bitangent));
}
else if (TypeID<Vertex>() == TypeID<FlatVertex>())
{
    SetAttribute(0, 3, (void*)offsetof(FlatVertex,
Position));
    SetAttribute(1, 4, (void*)offsetof(FlatVertex,
Color));
}
else if (TypeID<Vertex>() == TypeID<QuadVertex>())
{
    SetAttribute(0, 4, (void*)offsetof(QuadVertex,
Data));
}
else if (TypeID<Vertex>() == TypeID<SkyboxVertex>()) // 
<-- skybox vertex
{
    SetAttribute(0, 3, (void*)offsetof(SkyboxVertex,
Position));
}
else
{

```

```

    EMPTY_ERROR("invalid vertex type!");
}

// unbind vertex array
glBindVertexArray(0);
}

// ...

```

Additionally, we have added the header file **"Skybox.h"** depicted in (Listing [6.50](#)) to implement helper functions to create and render a skybox mesh.

Listing 6.50: Graphics/Utilities/Skybox.h

```

#pragma once
#include "../Buffers/Mesh.h"

namespace EmPy
{
    // skybox mesh type definition
    using SkyboxMesh = std::unique_ptr<Mesh<SkyboxVertex>>;

    // helps render a skybox mesh
    EMPTY_INLINE void RenderSkyboxMesh(SkyboxMesh& mesh)
    {
        // Enable back face culling
        glEnable(GL_CULL_FACE);
        glCullFace(GL_BACK);

        // Disable writing to the depth buffer
        glDepthMask(GL_FALSE);

        // Set the depth function to LEQUAL (less than or

```

```
equal)
    glDepthFunc(GL_LESS) ;

    // Draw the skybox mesh using triangles
    mesh->Draw(GL_TRIANGLES) ;

    // Reset the depth function to LESS (default)
    glDepthFunc(GL_LESS) ;

    // Enable writing to the depth buffer again
    glDepthMask(GL_TRUE) ;

    // Disable face culling
    glDisable(GL_CULL_FACE) ;
}

// helps create a skybox mesh
EMPTY_INLINE SkyboxMesh CreateSkyboxMesh()
{
    // cube vertices
    std::vector<glm::vec3> positions =
    {
        {-1.0f,  1.0f, -1.0f},
        {-1.0f, -1.0f, -1.0f},
        {1.0f, -1.0f, -1.0f},
        {1.0f, -1.0f, -1.0f},
        {1.0f,  1.0f, -1.0f},
        {-1.0f,  1.0f, -1.0f},
        {-1.0f, -1.0f,  1.0f},
        {-1.0f, -1.0f, -1.0f},
        {-1.0f,  1.0f, -1.0f},
        {-1.0f,  1.0f, -1.0f},
        {-1.0f,  1.0f,  1.0f},
        {-1.0f, -1.0f,  1.0f},
```

```

        {1.0f, -1.0f, -1.0f},
        {1.0f, -1.0f, 1.0f},
        {1.0f, 1.0f, 1.0f},
        {1.0f, 1.0f, 1.0f},
        {1.0f, 1.0f, -1.0f},
        {1.0f, -1.0f, -1.0f},

        {-1.0f, -1.0f, 1.0f},
        {-1.0f, 1.0f, 1.0f},
        {1.0f, 1.0f, 1.0f},
        {1.0f, 1.0f, 1.0f},
        {1.0f, -1.0f, 1.0f},
        {-1.0f, -1.0f, 1.0f},

        {-1.0f, 1.0f, -1.0f},
        {1.0f, 1.0f, -1.0f},
        {1.0f, 1.0f, 1.0f},
        {1.0f, 1.0f, 1.0f},
        {-1.0f, 1.0f, 1.0f},
        {-1.0f, 1.0f, -1.0f},

        {-1.0f, -1.0f, -1.0f},
        {-1.0f, -1.0f, 1.0f},
        {1.0f, -1.0f, -1.0f},
        {1.0f, -1.0f, -1.0f},
        {-1.0f, -1.0f, 1.0f},
        {1.0f, -1.0f, 1.0f}
    };

    // set mesh data
    MeshData<SkyboxVertex> data;
    for (uint32_t i = 0; i < positions.size(); ++i)
    {
        data.Vertices.push_back({ positions[i] });
    }
}

```

```
    // create and return skybox mesh
    return std::make_unique<Mesh<SkyboxVertex>>(data);
}
}
```

The function `RenderSkyboxMesh()` is responsible for rendering the skybox mesh. Let's break down the function's actions step by step:

- ☞ **Back Face Culling:** `glEnable(GL_CULL_FACE)` enables face culling, discarding triangles facing away from the camera. `glCullFace(GL_BACK)` sets back-face culling, meaning triangles facing away from the view (opposite to their defined normal) won't be rendered.
- ☞ **Depth Buffer Handling:** `glDepthMask(GL_FALSE)` disables writing to the depth buffer, preventing the depth buffer from being updated. `glDepthFunc(GL_EQUAL)` sets the depth function to `GL_EQUAL`, allowing fragments with depths equal to or less than the current depth value to pass the depth test.
- ☞ **Drawing the Skybox:** `Draw(GL_TRIANGLES)` instructs the skybox mesh to be drawn using triangle primitives. This renders the skybox geometry.
- ☞ **Resetting Depth Buffer Settings:** `glDepthFunc(GL_LESS)` resets the depth function to the

default value of `GL_LESS`. This change is essential to restore the typical depth testing behavior for subsequent rendering operations.

Skybox Cube Map

The next step involves loading the environment from an environment map as shown in (Figure 6.23), and creating a skybox cube map similar to the example illustrated in (Figure 6.24). The environment map utilizes spherical coordinates to encapsulate the captured scene. These scenes map each pixel of the environment using spherical coordinates. We will develop a custom shader responsible for generating a cube map by converting pixel coordinates from spherical space into equirectangular space.

The shader code is depicted in (Listing 6.51). This shader is written in a new file called "**skymap.glsl**". Make sure you add the file to your project as well. The vertex shader receives a position attribute for each vertex and transforms it into world space, computing the final position in clip space by combining the view and projection matrices. Additionally, it normalizes the world position and passes it to the fragment shader as an output.

Listing 6.51: Resources/Shaders/skymap.gls

```
#version 330 core
layout (location = 0) in vec3 a_position;

out vec3 world_position;

uniform mat4 u_view;
uniform mat4 u_proj;

void main()
{
    gl_Position = u_proj * u_view * vec4(a_position, 1.0);
    world_position = normalize(a_position);
}

++VERTEX++

#version 330 core
out vec4 out_fragment;
in vec3 world_position;

uniform sampler2D u_map;

vec2 GetSphericalUVs(vec3 v)
{
    vec2 uv = vec2(atan(v.z, v.x), asin(v.y));
    uv *= vec2(0.1591, 0.3183);
    uv += 0.5;
    return uv;
}

void main()
{
```

```
    vec2 uv = GetSphericalUVs(world_position);
    vec3 color = texture(u_map, uv).rgb;
    out_fragment = vec4(color, 1.0f);
}

++FRAGMENT++
```

Moving to the fragment shader, it receives the normalized world position computed in the vertex shader. Using this position, it converts the 3D spherical coordinates into 2D texture coordinates. This transformation involves the `atan()` and `asin()` functions, followed by scaling and offset adjustments to map the spherical coordinates to a 2D space. The resulting texture coordinates are then used to sample a 2D texture (*u*

width.3em *map*), retrieving the color value from the spherical texture. Finally, the resulting color is assigned to the fragment output.

This shader's C++ abstraction is depicted in the following code snippet (Listing 6.52). Similarly to what we have been doing in the PBR shader, we start by retrieving the locations of the uniforms from the shader in the constructor. The function `Generate()` is the one that does the job of creating a cube map and filling its faces with the values sampled from the environment map provided as an argument.

The function begins by defining all the different view directions where the camera will face when rendering the faces (left, right, top, bottom, etc.). This is crucial to render all faces of the cube map. It then proceeds to generate a new texture, which is then bound as a `GL_TEXTURE_CUBE_MAP`. This is why OpenGL differentiates between 2D textures and cube maps. The following for loop initializes the size of the faces.

In the subsequent part, we generate a frame buffer and a render buffer object which serve as rendering targets. The second for loop will subsequently render each face of the cube map using the skybox mesh. The function then ends by deleting all the allocated buffers and returning the cube map buffer ID.

Listing 6.52: Graphics/Shaders/SkyMap.h

```
#pragma once
#include "Shader.h"
#include "../Utilities/Skybox.h"

namespace EmPy
{
    struct SkyMapShader : Shader
    {
        EMPY_INLINE SkyMapShader(const std::string& path) :
        Shader(path)
        {
            u_View = glGetUniformLocation(m_ShaderID,
            "u_view");
        }
    };
}
```

```

        u_Proj = glGetUniformLocation(m_ShaderID,
"u_proj");
        u_Map = glGetUniformLocation(m_ShaderID,
"u_map");
    }

    EMPY_INLINE uint32_t Generate(Texture& texture,
SkyboxMesh& mesh, int32_t size)
{
    // view matrices
    glm::mat4 views[] =
    {
        glm::lookAt(glm::vec3(0.0f),
glm::vec3(1.0f, 0.0f, 0.0f), glm::vec3(0.0f, -1.0f, 0.0f)),
        glm::lookAt(glm::vec3(0.0f),
glm::vec3(-1.0f, 0.0f, 0.0f), glm::vec3(0.0f,
-1.0f, 0.0f)),
        glm::lookAt(glm::vec3(0.0f),
glm::vec3(0.0f, 1.0f, 0.0f), glm::vec3(0.0f, 0.0f, 1.0f)),
        glm::lookAt(glm::vec3(0.0f), glm::vec3(0.0f,
-1.0f, 0.0f), glm::vec3(0.0f, 0.0f, -1.0f)),
        glm::lookAt(glm::vec3(0.0f),
glm::vec3(0.0f, 0.0f, 1.0f), glm::vec3(0.0f, -1.0f, 0.0f)),
        glm::lookAt(glm::vec3(0.0f),
glm::vec3(0.0f, -1.0f, 0.0f), glm::vec3(0.0f, 0.0f, -1.0f))
    };

    // projection matrix
    glm::mat4 proj =
glm::perspective(glm::radians(90.0f), 1.0f, 0.1f, 10.0f);

    // bind shader
    glUseProgram(m_ShaderID);

    // set projection matrix
    glUniformMatrix4fv(u_Proj, 1, GL_FALSE,

```

```

glm::value_ptr(proj));

        // set texture source
texture->Use(u_Map, 0);

        // generate cube map
uint32_t cubeMap = 0u;
glGenTextures(1, &cubeMap);
glBindTexture(GL_TEXTURE_CUBE_MAP, cubeMap);

        // set cube map faces size
for (uint32_t i = 0; i < 6; ++i)
{
    glTexImage2D(GL_TEXTURE_CUBE_MAP_POSITIVE_X +
i, 0,
                GL_RGB16F, size, size, 0, GL_RGB, GL_FLOAT,
NULL);
}

        // set cube map sampling parameters
glTexParameteri(GL_TEXTURE_CUBE_MAP,
GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_CUBE_MAP,
GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_CUBE_MAP,
GL_TEXTURE_WRAP_R, GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_CUBE_MAP,
GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_CUBE_MAP,
GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_LINEAR);

        // target frame and render buffer
uint32_t FBO = 0u, RBO = 0u;
glGenFramebuffers(1, &FBO);
glGenRenderbuffers(1, &RBO);

```

```

        glBindFramebuffer(GL_FRAMEBUFFER, FBO);
        glBindRenderbuffer(GL_RENDERBUFFER, RBO);
        glRenderbufferStorage(GL_RENDERBUFFER,
GL_DEPTH_COMPONENT24, size, size);
        glFramebufferRenderbuffer(GL_FRAMEBUFFER,
GL_DEPTH_ATTACHMENT, GL_RENDERBUFFER, RBO);

        // set view port
        glViewport(0, 0, size, size);

        for (uint32_t i = 0; i < 6; ++i)
        {
            // set current face view matrix
            glUniformMatrix4fv(u_View, 1, GL_FALSE,
glm::value_ptr(views[i]));

            // set cube map current face
            glFramebufferTexture2D(GL_FRAMEBUFFER,
GL_COLOR_ATTACHMENT0,
                GL_TEXTURE_CUBE_MAP_POSITIVE_X + i, cubeMap,
0);

            // clear frame buffer
            glClear(GL_COLOR_BUFFER_BIT |
GL_DEPTH_BUFFER_BIT);

            // render face
            RenderSkyboxMesh(mesh);
        }

        // generate cubemap mipmap levels
        glGenerateMipmap(GL_TEXTURE_CUBE_MAP);

        // unbind shader, buffer
        glBindTexture(GL_TEXTURE_CUBE_MAP, 0);
        glBindFramebuffer(GL_FRAMEBUFFER, 0);

```

```

        glUseProgram(0);

        // delete buffers
        glDeleteRenderbuffers(1, &RBO);
        glDeleteFramebuffers(1, &FBO);
        return cubeMap;
    }

private:
    uint32_t u_View = 0u;
    uint32_t u_Proj = 0u;
    uint32_t u_Map = 0u;
};

}

```

Skybox GLSL Shader

Now that we have the tools to create, render a skybox mesh and generate a cube map from an environment texture, we want to also implement a shader to render the skybox in the scene. (Listing 6.53) shows the GLSL code of the skybox renderer shader.

Listing 6.53: Resources/Shaders/skybox.gls

```

#version 330 core
layout (location = 0) in vec3 a_position;

out vec3 world_position;
uniform mat4 u_model;
uniform mat4 u_view;
uniform mat4 u_proj;

```

```

void main()
{
    //mat4(mat3(u_view))
    vec4 position = u_proj * mat4(mat3(u_view)) *
    u_model * vec4(a_position, 1.0f);

    gl_Position = position.xyww;
    world_position = a_position;
}

++VERTEX++

#version 330 core
out vec4 out_fragment;

in vec3 world_position;
uniform samplerCube u_map;

void main()
{
    out_fragment = vec4(texture(u_map, world_position).rgb,
1.0);
}

++FRAGMENT++

```

The only thing worth mentioning here is the fact that we are canceling the translation of the camera in the view matrix by first making it a `mat3` and then `mat4`. This assure that the skybox is rendered around the scene and not far away from it. You can later remove the `mat4(mat3(u_view))` and simply use the view matrix to see this effect. You also note in the fragment

shader that the texture type is no longer `sampler2D` but `samplerCube` which is a cube map.

Skybox C++ Shader

Similar to the cube map generator shader we also need a C++ abstraction of the skybox renderer shader. This is done in the following code.

Listing 6.54: Graphics/Shaders/Skybox.h

```
#pragma once
#include "Shader.h"
#include "../Utilities/Skybox.h"

namespace EmPy
{
    struct SkyboxShader : Shader
    {
        EMPY_INLINE SkyboxShader(const std::string& path) :
        Shader(path)
        {
            u_Model = glGetUniformLocation(m_ShaderID,
"u_model");
            u_View = glGetUniformLocation(m_ShaderID,
"u_view");
            u_Proj = glGetUniformLocation(m_ShaderID,
"u_proj");
            u_Map = glGetUniformLocation(m_ShaderID,
"u_map");
        }
    };
}
```

```

    EMPY_INLINE void SetCamera(Camera3D& camera,
Transform3D& transform, float ratio)
{
    glUseProgram(m_ShaderID);
    glUniformMatrix4fv(u_Proj, 1, GL_FALSE,
glm::value_ptr(camera.Projection(ratio)));
    glUniformMatrix4fv(u_View, 1, GL_FALSE,
glm::value_ptr(camera.View(transform)));
}

    EMPY_INLINE void Draw(SkyboxMesh& mesh, uint32_t
cubeMap, Transform3D& transform)
{
    glm::mat4 model =
glm::toMat4(glm::quat(glm::radians(transform.Rotation)));
    glUseProgram(m_ShaderID);
    glUniformMatrix4fv(u_Model, 1, GL_FALSE,
glm::value_ptr(model));
    glActiveTexture(GL_TEXTURE0);
    glBindTexture(GL_TEXTURE_CUBE_MAP, cubeMap);
    glUniform1i(u_Map, 0);
    RenderSkyboxMesh(mesh);
}

private:
    uint32_t u_Model = 0u;
    uint32_t u_View = 0u;
    uint32_t u_Proj = 0u;
    uint32_t u_Map = 0u;
};

}

```

If you look closely in the `Draw()` function you will remark how the model matrix is set using only the

rotation component of the transform. Again the skybox can not move or be scaled but we can rotate it to set the proper face we want to be visible.

Skybox Component

Since want the skybox to an entity as well, we need to create a data structure and a component to house its properties. This is done in the following code snippets.

Listing 6.55: Graphics/Utilities/Data.
h

```
/* ... same as before ... */

// skybox data
struct Skybox
{
    EMPI_INLINE Skybox(const Skybox&) = default;
    EMPI_INLINE Skybox() = default;
    uint32_t CubeMap = 0u;
};
```

The skybox data will be extended later. Here the skybox component:

Listing 6.56: Auxiliaries/ECS.
h

```
/* ... same as before ... */

// skybox component
```

```

struct SkyboxComponent
{
    EMPY_INLINE SkyboxComponent(const SkyboxComponent&) = default;
    EMPY_INLINE SkyboxComponent() = default;
    Skybox Sky;
};

```

Texture Class Update

If you are using the same environment texture as depicted here, you have likely noticed that it is an HDR file. This indicates that the values stored in it are not within the range of [0, 255], as they are in a PNG file, for example. Consequently, we need to update the texture class to account for this difference.

Listing 6.57: Graphics/Textures/Texture.h

```

#pragma once
#include "Common/Core.h"
#include <stb_image.h>

namespace Empy
{
    struct Texture2D
    {
        EMPY_INLINE Texture2D(const std::string& path, bool
isHDR, bool flipY)
        {
            Load(path, isHDR, flipY);
        }
    };
}

```

```
EMPTY_INLINE Texture2D(const std::string& path) {
    Load(path);
}

EMPTY_INLINE ~Texture2D() { glDeleteTextures(1, &m_ID); }

EMPTY_INLINE Texture2D() = default;

EMPTY_INLINE bool Load(const std::string& path, bool isHDR = false, bool flipY = true)
{
    // flip y axis (common)
    stbi_set_flip_vertically_on_load(flipY);
    void* pixels = nullptr;

    // load texture data
    if (isHDR)
    {
        int32_t channels;
        pixels = stbi_loadf(path.c_str(), &m_Width,
&m_Height, &channels, 0);
    }
    else
    {
        pixels = stbi_load(path.c_str(), &m_Width,
&m_Height, nullptr, 4);
    }

    // check pixels
    if(pixels == nullptr)
    {
        EMPTY_ERROR("failed to load texture!");
        return false;
    }

    // create texture
    glGenTextures(1, &m_ID);
```

```

        glBindTexture(GL_TEXTURE_2D, m_ID);

        // load texture to gpu
        if (isHDR)
        {
            glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB16F, m_Width,
                         m_Height, 0, GL_RGB, GL_FLOAT,
                         (float*)pixels);
        }
        else
        {
            glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, m_Width,
                         m_Height,
                         0, GL_RGBA, GL_UNSIGNED_BYTE,
                         (uint32_t*)pixels);
        }

        // free allocated memory
        stbi_image_free(pixels);

        glTexParameteri(GL_TEXTURE_2D,
                        GL_TEXTURE_MAG_FILTER, GL_LINEAR);
        glTexParameteri(GL_TEXTURE_2D,
                        GL_TEXTURE_MIN_FILTER, GL_LINEAR);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S,
                        GL_CLAMP_TO_EDGE);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T,
                        GL_CLAMP_TO_EDGE);
        glGenerateMipmap(GL_TEXTURE_2D);

        glBindTexture(GL_TEXTURE_2D, 0);
        return true;
    }

    /* ... same as before ... */

```

```

private:
    int32_t m_Height = 0;
    int32_t m_Width = 0;
    uint32_t m_ID = 0u;
};

using Texture = std::shared_ptr<Texture2D>;
}

```

All we have done here is add some if statements to check whether the source texture is an HDR file or not. We have also added a constructor accordingly. In addition to that, everything remains unchanged.

This would work regardless of the texture type (PNG, JPG, etc.). The advantage of HDR textures is first the quality and the fact that we can apply post-processing effects such as gamma correction to make it look more realistic. More on that later.

Renderer Class Update

Let us now create these new introduced shaders in the engine's renderer class. This is done in the code depicted in (Listing 6.58).

Listing 6.58: Graphics/Renderer.h

```

struct GraphicsRenderer
{
    EMPY_INLINE GraphicsRenderer(int32_t width, int32_t
height)
    {
        // initialize opengl
        if(glewInit() != GLEW_OK)
        {
            EMPY_FATAL("failed to init glew!");
            exit(EXIT_FAILURE);
        }
        glewExperimental = GL_TRUE;

        // make the skybox edges look smooth
        glEnable(GL_TEXTURE_CUBE_MAP_SEAMLESS);

        // render shaders
        m_Final = std::make_unique<FinalShader>
("Resources/Shaders/final.glsl");
        m_Pbr = std::make_unique<PbrShader>
("Resources/Shaders/pbr.glsl");

        // skybox shaders
        m_SkyMap = std::make_unique<SkyMapShader>
("Resources/Shaders/skymap.glsl");
        m_Skybox = std::make_unique<SkyShader>
("Resources/Shaders/skybox.glsl");

        m_Frame = std::make_unique<FrameBuffer>(width,
height);
        m_SkyboxMesh = CreateSkyboxMesh(); //<-- create mesh
    }

    /* ... same as before ... */
}

```

```

    // initializes skybox
    EMPY_INLINE void InitSkybox(Skybox& sky, Texture&
texture, int32_t size)
    {
        sky.CubeMap = m_SkyMap->Generate(texture,
m_SkyboxMesh, size);
    }

    // renders skybox
    EMPY_INLINE void DrawSkybox(Skybox& sky, Transform3D&
transform)
    {
        m_Skybox->Draw(m_SkyboxMesh, sky.CubeMap, transform);
    }

    EMPY_INLINE void SetCamera(Camera3D& camera, Transform3D&
transform)
    {
        float aspect = m_Frame->Ratio();
        m_Pbr->SetCamera(camera, transform, aspect);

        // binds skybox shader ad set mvp
        m_Skybox->SetCamera(camera, transform, aspect);

        // rebind pbr shader again
        m_Pbr->Bind();
    }

private:
    std::unique_ptr<SkyMapShader> m_SkyMap; // <-- added
    std::unique_ptr<SkyboxShader> m_Skybox; // <-- added
    std::unique_ptr<FrameBuffer> m_Frame;
    std::unique_ptr<FinalShader> m_Final;
    std::unique_ptr<PbrShader> m_Pbr;

```

```
SkyboxMesh m_SkyboxMesh; // <-- added  
};
```

Everything done is self-explanatory. the `InitSkybox()` function takes a texture and a skybox as arguments to generate a cube map for the skybox. The `DrawSkybox()` function simply renders the skybox to the bounded frame buffer.

Application Class Update

Now let us add a skybox entity in the application context to see the result of our hard labor. This is done in the following code snippet.

Listing 6.59: Application/Application.h

```
/* ... same as before ... */  
  
EMPY_INLINE void RunContext()  
{  
    // load environment map  
    auto skymap = std::make_shared<Texture2D>  
("Resources/Textures/HDRs/Sky.hdr", true, true);  
  
    // load textures  
    auto roughness = std::make_shared<Texture2D>  
("Resources/Textures/Marble/Roughness.png");  
    auto albedo = std::make_shared<Texture2D>  
("Resources/Textures/Marble/Albedo.png");  
    auto normal = std::make_shared<Texture2D>
```

```
("Resources/Textures/Marble/Normal.png");

    // load models
    auto sphereModel = std::make_shared<Model>
("Resources/Models/sphere.fbx");

    // create scene camera
    auto camera = CreateEntt<Entity>();
    camera.Attach<TransformComponent>().Transform.Translate.z
= 3.0f;
    camera.Attach<CameraComponent>();

    // create skybox entity <----- note
    auto skybox = CreateEntt<Entity>();
    skybox.Attach<TransformComponent>();
    skybox.Attach<SkyboxComponent>();

    // create point light 1
    auto slight = CreateEntt<Entity>();
    slight.Attach<DirectLightComponent>().Light.Intensity =
5.0f;
    auto& stp = slight.Attach<TransformComponent>
().Transform;
    stp.Rotation = glm::vec3(0.0f, 0.0f, -2.0f);
    stp.Translate.z = 1.0f;

    // create cube entity
    auto cube = CreateEntt<Entity>();
    cube.Attach<TransformComponent>().Transform.Scale *=
2.0f;
    auto& mod = cube.Attach<ModelComponent>();
    mod.Model = sphereModel;
    mod.Material.Albedo = glm::vec3(0.0f);
    mod.Material.Roughness = 0.1f;
    mod.Material.Metallic = 0.25f;
    mod.Material.RoughnessMap = roughness;
```

```

    mod.Material.AlbedoMap = albedo;
    mod.Material.NormalMap = normal;

    // generate cube map <----- note!
    EnttView<Entity, SkyboxComponent>([this, &skymap] (auto
entity, auto& comp)
{
    m_Context->Renderer->InitSkybox(comp.Sky, skymap,
2048);
});

    // application main loop
    while(m_Context->Window->PollEvents())
{
    // start new frame
    m_Context->Renderer->NewFrame();

    // set shader camera
    EnttView<Entity, CameraComponent>([this] (auto
entity, auto& comp)
{
    auto& transform = entity.template
Get<TransformComponent>().Transform;
    m_Context->Renderer->SetCamera(comp.Camera,
transform);
});

/* ... same as before ... */

    // render skybox <----- note
    EnttView<Entity, SkyboxComponent>([this] (auto
entity, auto& comp)
{
    auto& transform = entity.template
Get<TransformComponent>().Transform;
    m_Context->Renderer->DrawSkybox(comp.Sky,

```

```

        transform);
        transform.Rotation.y += 0.16f;
    });

    // end frame
    m_Context->Renderer->EndFrame();

    // update layers
    for(auto layer : m_Context->Layers)
    {
        layer->OnUpdate();
    }

    // show frame to screen
    m_Context->Renderer->ShowFrame();
}
}

```

Compile and run the code to see the following result (Figure [6.25](#)). You will note how both the sphere and the skybox are incoherent in the scene. That is because the sphere is not affected by the presence of the skybox. This is where environment lighting comes in to solve this issue. Another thing to note here is the darkness of the skybox texture. We need to apply HDR Tone and Gamma correction to get the proper color and lighting range. We will deal with that later. But just to prove my point, you can load the texture with the argument `isHDR` set to `false` and you will see that the result is much clearer.



Figure 6.25: Scene Skybox

6.5.2 Diffuse Irradiance

In PBR, lighting of the environment is essential for accurate material rendering. IBL techniques often involve using HDR environment maps to capture lighting information from real environments. These maps contain information about lighting conditions, reflections, and the overall environment, allowing for more realistic rendering by accurately representing how light interacts with surfaces based on their material properties.

Diffuse irradiance refers to the amount of light energy diffusely reflected and scattered across a surface. It's a crucial component in global illumination and PBR, representing the indirect lighting that results from light

bouncing off surfaces and illuminating nearby areas. When light hits a surface, part of it gets absorbed, and the rest is reflected. Diffuse surfaces scatter reflected light uniformly in all directions, rather than reflecting it in a specific direction like a mirror. Diffuse irradiance describes the total amount of light energy per unit area arriving at a surface from all directions due to indirect illumination. It is a key factor in determining how objects appear when illuminated by indirect light sources, providing soft and even lighting across surfaces.

The diffuse irradiance at a point on a surface is commonly derived from Lambert's Cosine Law [GoPhotonics](#). It simplifies the calculation of diffuse reflection by assuming that the amount of reflected light is proportional to the cosine of the angle between the incoming light direction and the surface normal.

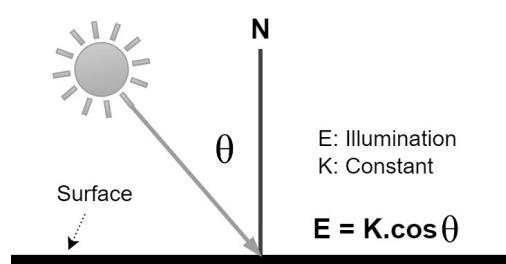


Figure 6.26: Lambert's Cosine Law

Lambert's cosine law establishes a direct relationship between the radiant intensity from an ideally diffusely reflecting surface and the cosine of the angle θ formed between the incident light direction and the surface normal. This relationship is depicted in (Figure [6.26](#)).

GLSL Shader Code

In order to apply the environment ambient light to the sphere or any other object in the scene, we need to generate the irradiance map from the skybox cube map we create earlier. This irradiance map is simply a convoluted version of the that cube map or in basic terms a blurred version. This is depicted in (Figure [6.23](#)). We therefore need to implement a new shader to do that for us.

Listing 6.60: Resource/Shaders/irradiance.gls

```
#version 330 core
layout (location = 0) in vec3 a_position;

out vec3 world_position;

uniform mat4 u_view;
uniform mat4 u_proj;

void main()
{
    gl_Position = u_proj * u_view * vec4(a_position, 1.0f);
```

```

    world_position = a_position;
}

++VERTEX++

#version 330 core
out vec4 out_fragment;
in vec3 world_position;

#define PI 3.14159265358979323846

uniform samplerCube u_cubemap;

void main()
{
    vec3 N = normalize(world_position);
    vec3 irradiance = vec3(0.0);

    // irradiance
    vec3 up = vec3(0.0, 1.0, 0.0);
    vec3 right = cross(up, N);
    up = cross(N, right);

    int nrSamples = 0;
    float sampleDelta = 0.025;

    for(float phi = 0.0; phi < 2.0 * PI; phi +=
sampleDelta)
    {
        for(float theta = 0.0; theta < 0.5 * PI; theta +=
sampleDelta)
        {
            // spherical to cartesian (in tangent space)
            vec3 tangentSample = vec3(sin(theta) *
cos(phi), sin(theta) * sin(phi), cos(theta));

```

```

        // tangent space to world
        vec3 sampleVec = tangentSample.x * right +
tangentSample.y * up + tangentSample.z * N;

        // compute irradiance value
        irradiance += texture(u_cubemap, sampleVec).rgb *
cos(theta) * sin(theta);

        // increment sample count
        nrSamples++;
    }

}

irradiance = PI * irradiance * (1.0 / float(nrSamples));
out_fragment = vec4(irradiance, 1.0);
}

++FRAGMENT++

```

This shader simply generates a blurred version of the skybox cube map.

Shader C++ Abstraction

This shader abstraction is similar to the **"SkyMapShader"**. It also has a `Generate()` function which does almost the same thing.

Listing 6.61: Graphics/Shaders/Irradiance.h

```
#pragma once
#include "Shader.h"
```

```

#include "../Utilities/Skybox.h"

namespace EmPy
{
    struct IrradianceShader : Shader
    {
        EMPY_INLINE IrradianceShader(const std::string& path) : Shader(path)
        {
            u_CubeMap = glGetUniformLocation(m_ShaderID,
"u_cubemap");
            u_View = glGetUniformLocation(m_ShaderID,
"u_view");
            u_Proj = glGetUniformLocation(m_ShaderID,
"u_proj");
        }

        EMPY_INLINE uint32_t Generate(uint32_t skyCubMap,
SkyboxMesh& mesh, int32_t size)
        {
            // view matrices
            glm::mat4 views[] =
            {
                glm::lookAt(glm::vec3(0.0f),
glm::vec3(1.0f, 0.0f, 0.0f), glm::vec3(0.0f, -1.0f, 0.0f)),
                glm::lookAt(glm::vec3(0.0f),
glm::vec3(-1.0f, 0.0f, 0.0f), glm::vec3(0.0f,
-1.0f, 0.0f)),
                glm::lookAt(glm::vec3(0.0f),
glm::vec3(0.0f, 1.0f, 0.0f), glm::vec3(0.0f, 0.0f, 1.0f)),
                glm::lookAt(glm::vec3(0.0f), glm::vec3(0.0f,
-1.0f, 0.0f), glm::vec3(0.0f, 0.0f, -1.0f)),
                glm::lookAt(glm::vec3(0.0f),
glm::vec3(0.0f, 0.0f, 1.0f), glm::vec3(0.0f, -1.0f, 0.0f)),
                glm::lookAt(glm::vec3(0.0f),
glm::vec3(0.0f, -1.0f, 0.0f), glm::vec3(0.0f, 1.0f, 0.0f))
            };
        }
    };
}

```

```

    };

    // projection
    glm::mat4 projection =
glm::perspective(glm::radians(90.0f), 1.0f, 0.1f, 10.0f);

    // generate cube map
    uint32_t irradMap = 0u;
    glGenTextures(1, &irradMap);
    glBindTexture(GL_TEXTURE_CUBE_MAP, irradMap);

    // init size
    for (uint32_t i = 0; i < 6; ++i)
    {
        glTexImage2D(GL_TEXTURE_CUBE_MAP_POSITIVE_X +
i, 0,
                    GL_RGB16F, size, size, 0, GL_RGB, GL_FLOAT,
NULL);
    }

    glTexParameteri(GL_TEXTURE_CUBE_MAP,
GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
    glTexParameteri(GL_TEXTURE_CUBE_MAP,
GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
    glTexParameteri(GL_TEXTURE_CUBE_MAP,
GL_TEXTURE_WRAP_R, GL_CLAMP_TO_EDGE);
    glTexParameteri(GL_TEXTURE_CUBE_MAP,
GL_TEXTURE_MIN_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_CUBE_MAP,
GL_TEXTURE_MAG_FILTER, GL_LINEAR);

    glUseProgram(m_ShaderID);
    glUniformMatrix4fv(u_Proj, 1, GL_FALSE,
glm::value_ptr(projection));

    // bind skybox cube map

```

```

        glActiveTexture(GL_TEXTURE0);
        glBindTexture(GL_TEXTURE_CUBE_MAP, skyCubMap);
        glUniform1i(u_CubeMap, 0);

        uint32_t FBO, RBO = 0u;
        glGenFramebuffers(1, &FBO);
        glGenRenderbuffers(1, &RBO);
        glBindFramebuffer(GL_FRAMEBUFFER, FBO);
        glBindRenderbuffer(GL_RENDERBUFFER, RBO);
        glRenderbufferStorage(GL_RENDERBUFFER,
GL_DEPTH_COMPONENT24, size, size);

        glViewport(0, 0, size, size);
        glBindFramebuffer(GL_FRAMEBUFFER, FBO);

        for (uint32_t i = 0; i < 6; ++i)
        {
            glUniformMatrix4fv(u_View, 1, GL_FALSE,
glm::value_ptr(views[i]));
            glFramebufferTexture2D(GL_FRAMEBUFFER,
GL_COLOR_ATTACHMENT0,
GL_TEXTURE_CUBE_MAP_POSITIVE_X + i, irradMap,
0);
            glClear(GL_COLOR_BUFFER_BIT |
GL_DEPTH_BUFFER_BIT);
            RenderSkyboxMesh(mesh);
        }

        // unbind shader, buffer
        glBindTexture(GL_TEXTURE_CUBE_MAP, 0);
        glBindFramebuffer(GL_FRAMEBUFFER, 0);
        glUseProgram(0);

        // delete fbo & rbo
        glDeleteRenderbuffers(1, &RBO);
        glDeleteFramebuffers(1, &FBO);

```

```

        return irradMap;
    }

private:
    uint32_t u_CubeMap = 0u;
    uint32_t u_View = 0u;
    uint32_t u_Proj = 0u;
};

}

```

PBR Shader Update

Let us update the PBR shader to use the irradiance map to compute the ambient light. You can see in the following code snippet how that is done.

Listing 6.62: Resources/Shaders/pbr.gls

```

/* ... same as before ... */

// enviroment maps <--- added
uniform samplerCube u_irradMap;

// compute ambient light <--- added
vec3 ComputeAmbientLight(vec3 N, vec3 V, vec3 F0, vec3
albedo, float roughness, float metallic)
{
    // angle between surface normal & light direction.
    float cosTheta = max(0.0, dot(N, V));

    // sample diffuse irradiance
    vec3 F = FresnelSchlick(cosTheta, F0);

```

```

    // get diffuse contribution factor
    vec3 kd = mix(vec3(1.0) - F, vec3(0.0), metallic);

    // irradiance map contains exitant radiance
    vec3 diffuse = kd * albedo * texture(u_irradMap, N).rgb;

    return diffuse;
}

// main function
void main()
{
    // lights contribution <--- update
    vec3 result = ComputeAmbientLight(N, V, F0, albedo,
roughness, metallic);
    result += ComputeDirectLights(N, V, F0, albedo, roughness,
metallic);
    result += ComputePointLights(N, V, F0, albedo, roughness,
metallic);
    result += ComputeSpotLights(N, V, F0, albedo, roughness,
metallic);
}

++FRAGMENT++

```

The newly added function `ComputeAmbientLight()` takes care of that and returns the computed value which used in the main function to calculate the combined effect of all light sources.

Shader Class Update

Simply add a member variable to the uniform location of the irradiance map and make sure to set the value in the constructor.

Listing 6.63: Graphics/Shaders/PBR.
h

```
/* ... same as before ... */

struct PbrShader : Shader
{
    EMPTY_INLINE PbrShader(const std::string& filename) :
    Shader(filename)
    {
        /* ... same as before ... */

        u_IrradMap = glGetUniformLocation(m_ShaderID,
"u_irradMap");
    }

    EMPTY_INLINE void SetEnvMaps(uint32_t irrad)
    {
        glUseProgram(m_ShaderID);

        // irradiance map
        glActiveTexture(GL_TEXTURE0); // <-- texture unit 0
        glBindTexture(GL_TEXTURE_CUBE_MAP, irrad);
        glUniform1i(u_IrradMap, 0);

        // more is coming
    }

    EMPTY_INLINE void Draw(Model3D& model, PbrMaterial&
material, Transform3D& transform)
    {
```

```

        glUniformMatrix4fv(u_Model, 1, GL_FALSE,
glm::value_ptr(transform.Matrix()));

        glUniform3fv(u_Albedo, 1, &material.Albedo.x);
        glUniform1f(u_Roughness, material.Roughness);
        glUniform1f(u_Metallic, material.Metallic);

        int32_t unit = 1 // <-- starts at unit 1 !!

        /* ... same as before ... */
    }

    /* ... same as before ... */

private:
    /* ... same as before ... */

    uint32_t u_IrradMap = 0u;
}

```

Skybox Data Update

Update the skybox structure as shown in the following code snippet:

Listing 6.64: Graphics/Utilities/Data. h

```

struct Skybox
{
    EMPY_INLINE Skybox(const Skybox&) = default;
    EMPY_INLINE Skybox() = default;
    uint32_t IrradMap = 0u;
}

```

```
    uint32_t CubeMap = 0u;  
};
```

We have the addition of the irradiance cube map. This will be sent to the PBR shader to compute the diffuse light of the sky.

Renderer Class Update

If you are not sure whether the irradiance map is generated properly, you can use it as a skybox cube map. The result will look like the one in (Figure [6.27](#)).

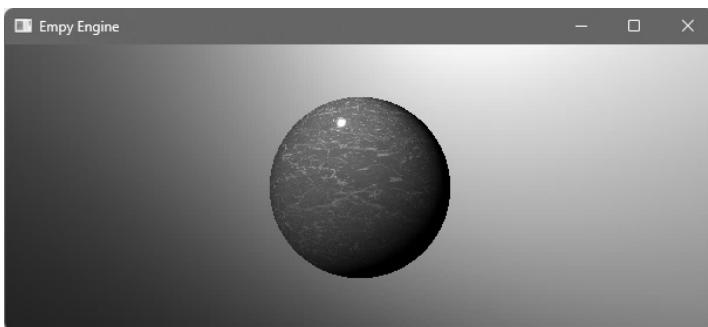


Figure 6.27: Irradiance Map

Now create the new shader in the renderer and add it to the rendering pipeline as depicted below.

Listing 6.65: Graphics/Renderer.h

```

struct GraphicsRenderer
{
    EMPTY_INLINE void InitSkybox(Skybox& sky, Texture&
texture, int32_t size)
    {
        // generate skybox cube map
        sky.CubeMap = m_SkyMap->Generate(texture,
m_SkyboxMesh, size);

        // generate irradiance map
        sky.IrradMap = m_Irrad->Generate(sky.CubeMap,
m_SkyboxMesh, 32);
    }
};

EMPTY_INLINE void DrawSkybox(Skybox& sky, Transform3D&
transform)
{
    m_Skybox->Draw(m_SkyboxMesh, sky.CubeMap, transform);

    // set pbr shader irradiance map
    m_Pbr->SetEnvMaps(sky.IrradMap);
}

```

If you compile and run your code without a light source, or simply set the directional light intensity to zero, you will get the result depicted in (Figure [6.28](#)). You can see how the environment is affecting the sphere even if there is not direct light in the scene.



Figure 6.28: Ambient Diffuse Light

6.5.3 Specular Irradiance

Indirect diffuse lighting is quite good because it creates some coherence in the scene, but we are also seeking surfaces that gleam. This implies the necessity of incorporating specular lighting. This concept revolves around the idea that rough surfaces will reflect less light than smooth surfaces. This is possible by computing the so-called "**Radiance Integral**".

Radiance Integral

The radiance integral is a fundamental equation in rendering that describes how light interacts with surfaces. It's typically represented by the rendering equation, formulated by Jim Kajiya in computer

graphics. The rendering equation describes the total outgoing light (radiance) at a point in a given direction as the sum of emitted light and reflected light from all other directions. The general form of the rendering equation is:

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

Where: - $L_o(\mathbf{x}, \omega_o)$ is the outgoing radiance from point \mathbf{x} in the direction ω_o . - $L_e(\mathbf{x}, \omega_o)$ is the emitted radiance from the surface at point \mathbf{x} in the direction ω_o . - $f_r(\mathbf{x}, \omega_i, \omega_o)$ is the Bidirectional Reflectance Distribution Function (BRDF) which describes how light is reflected from the surface at point \mathbf{x} from direction ω_i to direction ω_o . - $L_i(\mathbf{x}, \omega_i)$ is the incoming radiance from direction ω_i at point \mathbf{x} . - ω_i and ω_o are incoming and outgoing directions respectively. - \mathbf{n} is the surface normal. - Ω represents the hemisphere of all possible incoming directions.

The integral represents the summation of reflected light from all directions ω_i weighted by the incoming radiance and the BRDF at that point on the surface. This equation provides the foundation for PBR and describes the light transport within a scene, accounting for reflection, refraction, absorption, and scattering of light. In reality, this formula is quite heavy to compute

in realtime but thanks to "Epic Games" approach describe in this paper [Brian Karis](#), this can be achieved with less computation. Here are the different steps involved in the process.

☞ **Prefiltered Environment Map:** The first step evolves generating a prefiltered map of the environment with multiple Mip-Level which also match roughness values between 0 and 1. See (Figure [6.29](#)).

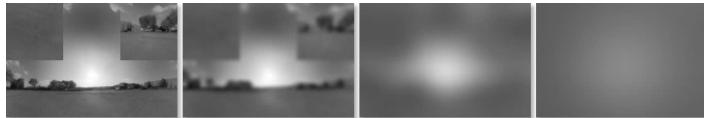


Figure 6.29: Prefiltered Environment Map

As we transition between Mip-Levels, we'll be blurring the images through convolution, influenced by the surface roughness. As the roughness grows, we'll utilize a greater number of samples, resulting in a more widespread and pronounced effect.

☞ **BRDF Integration Map:** In order to properly match the object roughness to the according Mip-Level of the prefiltered environment map, we need to create the "BRDF Integration Map" using the BRDF Integration. This integration takes two inputs: the roughness, ranging between 0 and 1, and the dot

product of the normal and the light direction ($N \cdot L$, read as N dot L), also ranging between 0 and 1. Here is how it looks:



Figure 6.30: BRDF Integration Map

These explanations provided in this book are quite fundamental. If you would like a deeper understanding, I recommend referring to the Epic Games paper [Brian Karis](#). I personally don't fully grasp it, but it works.

Let us implement this in the engine.

Prefiltered and BRDF Shaders

Following are the code of both the prefiltered map generator and the BRDF map generator shaders. These shader source codes are mostly inspired by the

code described in the "**LearnOpenGL**" website and if you are interested in the technical details you can refer to it here: [de Vries \[a\]](#). As I said earlier, I don't understand most of it but it does the job xD.

Listing 6.66: Resources/Shaders/prefiltered.gls

```
#version 330 core
layout (location = 0) in vec3 a_position;

out vec3 world_position;

uniform mat4 u_view;
uniform mat4 u_proj;

void main()
{
    gl_Position = u_proj * u_view * vec4(a_position, 1.0f);
    world_position = a_position;
}

++VERTEX++

#version 330 core
out vec4 out_fragment;
in vec3 world_position;

const float PI = 3.14159265358979323846;

uniform float u_roughness;
uniform samplerCube u_cubemap;

float RadicalInverseVdC(uint bits)
{
```

```

        bits = (bits << 16u) | (bits >> 16u);
        bits = ((bits & 0x55555555u) << 1u) | ((bits &
0xAAAAAAAU) >> 1u);
        bits = ((bits & 0x33333333u) << 2u) | ((bits &
0xCCCCCCCCu) >> 2u);
        bits = ((bits & 0xF0F0F0Fu) << 4u) | ((bits &
0xF0F0F0F0u) >> 4u);
        bits = ((bits & 0x00FF00FFu) << 8u) | ((bits &
0xFF00FF00u) >> 8u);
        return float(bits) * 2.3283064365386963e-10; // /
0x100000000
    }

vec3 ImportanceSampleGGX(vec2 Xi, vec3 N, float roughness)
{
    float a = roughness*roughness;

    float phi = 2.0 * PI * Xi.x;
    float cosTheta = sqrt((1.0 - Xi.y) / (1.0 + (a*a - 1.0)
* Xi.y));
    float sinTheta = sqrt(1.0 - cosTheta*cosTheta);

    // from spherical coordinates to cartesian coordinates
    vec3 H;
    H.x = cos(phi) * sinTheta;
    H.y = sin(phi) * sinTheta;
    H.z = cosTheta;

    // from tangent-space vector to world-space sample
    vector
        vec3 up          = abs(N.z) < 0.999 ? vec3(0.0, 0.0, 1.0)
: vec3(1.0, 0.0, 0.0);
        vec3 tangent     = normalize(cross(up, N));
        vec3 bitangent   = cross(N, tangent);

    vec3 sampleVec = tangent * H.x + bitangent * H.y + N *

```

```

H.z;

    return normalize(sampleVec);
}

vec2 Hammersley(uint i, uint N)
{
    return vec2(float(i)/float(N), RadicalInverseVdC(i));
}

void main()
{
    vec3 N = normalize(world_position);
    vec3 R = N;
    vec3 V = R;

    const uint SAMPLE_COUNT = 1024u;
    float totalWeight = 0.0;
    vec3 result = vec3(0.0);

    for(uint i = 0u; i < SAMPLE_COUNT; ++i)
    {
        vec2 Xi = Hammersley(i, SAMPLE_COUNT);
        vec3 H = ImportanceSampleGGX(Xi, N, u_roughness);
        vec3 L = normalize(2.0 * dot(V, H) * H - V);

        float NdotL = max(dot(N, L), 0.0);
        if(NdotL > 0.0)
        {
            result += texture(u_cubemap, L).rgb * NdotL;
            totalWeight += NdotL;
        }
    }

    result = result / totalWeight;
    out_fragment = vec4(result, 1.0);
}

```

```
++FRAGMENT++
```

Here the BRDF map generator shader code.

Listing 6.67: Resources/Shaders/brdf.gls

```
#version 330 core
layout (location = 0) in vec4 a_quad;

out vec2 uvs;

void main()
{
    gl_Position = vec4(a_quad.x, a_quad.y, 0.0, 1.0);
    uvs = vec2(a_quad.z, a_quad.w);
}

++VERTEX++

#version 330 core
out vec4 out_fragment;
in vec2 uvs;

const float PI = 3.14159265358979323846;

float RadicalInverseVdC(uint bits)
{
    bits = (bits << 16u) | (bits >> 16u);
    bits = ((bits & 0x55555555u) << 1u) | ((bits &
0xAAAAAAAAu) >> 1u);
    bits = ((bits & 0x33333333u) << 2u) | ((bits &
0xCCCCCCCCu) >> 2u);
```

```

        bits = ((bits & 0x0F0F0FFu) << 4u) | ((bits &
0xF0F0F0Fu) >> 4u);
        bits = ((bits & 0x00FF00FFu) << 8u) | ((bits &
0xFF00FF00u) >> 8u);
        return float(bits) * 2.3283064365386963e-10; // /
0x100000000
    }

vec2 Hammersley(uint i, uint N)
{
    return vec2(float(i)/float(N), RadicalInverseVdC(i));
}

vec3 ImportanceSampleGGX(vec2 Xi, vec3 N, float roughness)
{
    float a = roughness*roughness;

    float phi = 2.0 * PI * Xi.x;
    float cosTheta = sqrt((1.0 - Xi.y) / (1.0 + (a*a - 1.0)
* Xi.y));
    float sinTheta = sqrt(1.0 - cosTheta*cosTheta);

    // from spherical coordinates to cartesian coordinates -
halfway vector
    vec3 H;
    H.x = cos(phi) * sinTheta;
    H.y = sin(phi) * sinTheta;
    H.z = cosTheta;

    // from tangent-space H vector to world-space sample
vector
    vec3 up = abs(N.z) < 0.999 ? vec3(0.0, 0.0,
1.0) : vec3(1.0, 0.0, 0.0);
    vec3 tangent = normalize(cross(up, N));
    vec3 bitangent = cross(N, tangent);
}

```

```

    vec3 sampleVec = tangent * H.x + bitangent * H.y + N *
H.z;
    return normalize(sampleVec);
}

float GeometrySchlickGGX(float NdotV, float roughness)
{
    // note that we use a different k for IBL
    float a = roughness;
    float k = (a * a) / 2.0;

    float nom    = NdotV;
    float denom = NdotV * (1.0 - k) + k;

    return nom / denom;
}

float GeometrySmith(vec3 N, vec3 V, vec3 L, float roughness)
{
    float NdotV = max(dot(N, V), 0.0);
    float NdotL = max(dot(N, L), 0.0);
    float ggx2 = GeometrySchlickGGX(NdotV, roughness);
    float ggx1 = GeometrySchlickGGX(NdotL, roughness);

    return ggx1 * ggx2;
}

vec2 IntegrateBRDF(float NdotV, float roughness)
{
    vec3 V;
    V.x = sqrt(1.0 - NdotV*NdotV);
    V.y = 0.0;
    V.z = NdotV;

    float A = 0.0;
    float B = 0.0;
}

```

```

vec3 N = vec3(0.0, 0.0, 1.0);

const uint SAMPLE_COUNT = 1024u;
for(uint i = 0u; i < SAMPLE_COUNT; ++i)
{
    // generates a sample vector that's biased towards
the
    // preferred alignment direction (importance
sampling).

    vec2 Xi = Hammersley(i, SAMPLE_COUNT);
    vec3 H = ImportanceSampleGGX(Xi, N, roughness);
    vec3 L = normalize(2.0 * dot(V, H) * H - V);

    float NdotL = max(L.z, 0.0);
    float NdotH = max(H.z, 0.0);
    float VdotH = max(dot(V, H), 0.0);

    if(NdotL > 0.0)
    {
        float G = GeometrySmith(N, V, L, roughness);
        float G_Vis = (G * VdotH) / (NdotH * NdotV);
        float Fc = pow(1.0 - VdotH, 5.0);

        A += (1.0 - Fc) * G_Vis;
        B += Fc * G_Vis;
    }
}

A /= float(SAMPLE_COUNT);
B /= float(SAMPLE_COUNT);
return vec2(A, B);
}

void main()
{
    out_fragment = vec4(IntegrateBRDF(uvs.x, uvs.y), 0.0,

```

```

1.0);
}

++FRAGMENT++

```

Shaders C++ Abstraction

Again we need to also define C++ class to implement abstractions to these shaders. Following is the prefiltered map generator shader abstraction:

Listing 6.68: Graphics/Shaders/Prefiltered.h

```

#pragma once
#include "Shader.h"
#include "../Utilities/Skybox.h"

namespace EmPy
{
    struct PrefilteredShader : Shader
    {
        EMPY_INLINE PrefilteredShader(const std::string&
path) : Shader(path)
        {
            u_Roughness = glGetUniformLocation(m_ShaderID,
"u_roughness");
            u_CubeMap = glGetUniformLocation(m_ShaderID,
"u_cubemap");
            u_View = glGetUniformLocation(m_ShaderID,
"u_view");
            u_Proj = glGetUniformLocation(m_ShaderID,
"u_proj");
        }
}

```

```

    EMPTY_INLINE uint32_t Generate(uint32_t skyCubMap,
SkyboxMesh& mesh, int32_t size)
{
    // view matrices
    glm::mat4 views[] =
    {
        glm::lookAt(glm::vec3(0.0f),
glm::vec3(1.0f, 0.0f, 0.0f), glm::vec3(0.0f, -1.0f, 0.0f)),
        glm::lookAt(glm::vec3(0.0f),
glm::vec3(-1.0f, 0.0f, 0.0f), glm::vec3(0.0f,
-1.0f, 0.0f)),
        glm::lookAt(glm::vec3(0.0f),
glm::vec3(0.0f, 1.0f, 0.0f), glm::vec3(0.0f, 0.0f, 1.0f)),
        glm::lookAt(glm::vec3(0.0f), glm::vec3(0.0f,
-1.0f, 0.0f), glm::vec3(0.0f, 0.0f, -1.0f)),
        glm::lookAt(glm::vec3(0.0f),
glm::vec3(0.0f, 0.0f, 1.0f), glm::vec3(0.0f, -1.0f, 0.0f)),
        glm::lookAt(glm::vec3(0.0f),
glm::vec3(0.0f, -1.0f, 0.0f), glm::vec3(0.0f, 0.0f, -1.0f))
    };

    // projection
    glm::mat4 projection =
glm::perspective(glm::radians(90.0f), 1.0f, 0.1f, 10.0f);

    // generate cube map
    uint32_t prefilteredMap = 0u;
    glGenTextures(1, &prefilteredMap);
    glBindTexture(GL_TEXTURE_CUBE_MAP,
prefilteredMap);

    // init size
    for (uint32_t i = 0; i < 6; ++i)
    {
        glTexImage2D(GL_TEXTURE_CUBE_MAP_POSITIVE_X +

```

```

i, 0,
        GL_RGB16F, size, size, 0, GL_RGB, GL_FLOAT,
NULL);
}

        glTexParameteri(GL_TEXTURE_CUBE_MAP,
GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
        glTexParameteri(GL_TEXTURE_CUBE_MAP,
GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
        glTexParameteri(GL_TEXTURE_CUBE_MAP,
GL_TEXTURE_WRAP_R, GL_CLAMP_TO_EDGE);
        glTexParameteri(GL_TEXTURE_CUBE_MAP,
GL_TEXTURE_MIN_FILTER, GL_LINEAR);
        glTexParameteri(GL_TEXTURE_CUBE_MAP,
GL_TEXTURE_MAG_FILTER, GL_LINEAR);

        glUseProgram(m_ShaderID);
        glUniformMatrix4fv(u_Proj, 1, GL_FALSE,
glm::value_ptr(projection));

        // bind skybox cube map
        glBindTexture(GL_TEXTURE0);
        glBindTexture(GL_TEXTURE_CUBE_MAP, skyCubMap);
        glUniform1i(u_CubeMap, 0);

uint32_t FBO, RBO = 0u;
 glGenFramebuffers(1, &FBO);
 glGenRenderbuffers(1, &RBO);
 glBindFramebuffer(GL_FRAMEBUFFER, FBO);
 glBindRenderbuffer(GL_RENDERBUFFER, RBO);
 glRenderbufferStorage(GL_RENDERBUFFER,
GL_DEPTH_COMPONENT24, size, size);

        // number of mip levels
        uint32_t nbrMipLevels = 5;

```

```

        // loop for each mip level
        for (uint32_t mip = 0; mip < nbrMipLevels;
++mip)
    {
        // reisze framebuffer according to mip-level.
        int32_t mipWidth = (int)(size * std::pow(0.5,
mip));
        int32_t mipHeight = (int)(size *
std::pow(0.5, mip));

        glBindRenderbuffer(GL_RENDERBUFFER, RBO);
        glRenderbufferStorage(GL_RENDERBUFFER,
GL_DEPTH_COMPONENT24, mipWidth, mipHeight);
        glViewport(0, 0, mipWidth, mipHeight);

        float roughness = (float)mip / (float)
(nbrMipLevels - 1);
        glUniform1f(u_Roughness, roughness);

        for (uint32_t i = 0; i < 6; ++i)
        {
            glUniformMatrix4fv(u_View, 1, GL_FALSE,
glm::value_ptr(views[i]));
            glFramebufferTexture2D(GL_FRAMEBUFFER,
GL_COLOR_ATTACHMENT0,
                GL_TEXTURE_CUBE_MAP_POSITIVE_X + i,
prefilteredMap, mip);
            glClear(GL_COLOR_BUFFER_BIT |
GL_DEPTH_BUFFER_BIT);
            RenderSkyboxMesh(mesh);
        }
    }

    // unbind shader, buffer
    glBindTexture(GL_TEXTURE_CUBE_MAP, 0);

```

```

        glBindFramebuffer(GL_FRAMEBUFFER, 0);
        glUseProgram(0);

        // delete fbo & rbo
        glDeleteRenderbuffers(1, &RBO);
        glDeleteFramebuffers(1, &FBO);
        return prefilteredMap;
    }

private:
    uint32_t u_Roughness = 0u;
    uint32_t u_CubeMap = 0u;
    uint32_t u_View = 0u;
    uint32_t u_Proj = 0u;
};

}

```

This code is similar to that of the irradiance shader class. The main difference here is that we have added another loop on top of what was there before to compute the different Mip-Levels of the cube map. We have only chosen 5 Mip-Levels. You can increase it if you want more details, but 5 is largely enough.

Here is the same thing for the BRDF map generator shader.

Listing 6.69: Graphics/Shaders/BRDF.h

```

#pragma once
#include "Shader.h"
#include "../Utilities/Quad.h"

```

```

namespace Empty
{
    struct BrdfShader : Shader
    {
        EMPTY_INLINE BrdfShader(const std::string& path) :
            Shader(path)
        { }

        EMPTY_INLINE uint32_t Generate(int32_t size)
        {
            uint32_t brdfMap = 0u;
            glGenTextures(1, &brdfMap);
            glBindTexture(GL_TEXTURE_2D, brdfMap);
            glTexImage2D(GL_TEXTURE_2D, 0, GL_RG16F, size,
size, 0, GL_RG, GL_FLOAT, 0);
            glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S,
GL_CLAMP_TO_EDGE);
            glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T,
GL_CLAMP_TO_EDGE);
            glTexParameteri(GL_TEXTURE_2D,
GL_TEXTURE_MIN_FILTER, GL_LINEAR);
            glTexParameteri(GL_TEXTURE_2D,
GL_TEXTURE_MAG_FILTER, GL_LINEAR);

            glUseProgram(m_ShaderID);

            uint32_t FBO, RBO = 0;
            glGenFramebuffers(1, &FBO);
            glGenRenderbuffers(1, &RBO);

            glBindFramebuffer(GL_FRAMEBUFFER, FBO);
            glBindRenderbuffer(GL_RENDERBUFFER, RBO);
            glRenderbufferStorage(GL_RENDERBUFFER,
GL_DEPTH_COMPONENT24, size, size);
            glFramebufferTexture2D(GL_FRAMEBUFFER,

```

```

GL_COLOR_ATTACHMENT0, GL_TEXTURE_2D, brdfMap, 0);

        glViewport(0, 0, size, size);
        glClear(GL_COLOR_BUFFER_BIT |
GL_DEPTH_BUFFER_BIT);
        CreateQuad2D()->Draw(GL_TRIANGLES);

        glBindFramebuffer(GL_FRAMEBUFFER, 0);
        glBindTexture(GL_TEXTURE_2D, 0);
        glUseProgram(0);

        // delete fbo & rbo
        glDeleteRenderbuffers(1, &RBO);
        glDeleteFramebuffers(1, &FBO);
        return brdfMap;
    }
};

}

```

This shader does not have any uniforms. It only generates a texture and then uses the shader to compute the BRDF pixels. It uses a quad because it is a 2D texture.

Skybox Class Update

Update the Skybox structure by adding these two additional maps.

Listing 6.70: Graphics/Utilities/Data.
h

```

// skybox data
struct Skybox
{
    EMPY_INLINE Skybox(const Skybox&) = default;
    EMPY_INLINE Skybox() = default;

    uint32_t PrefilMap = 0u; // prefiltered map
    uint32_t IrradMap = 0u;
    uint32_t CubeMap = 0u;
    uint32_t BrdfMap = 0u; // brdf map
};

```

Renderer Class Update

You can now add these shaders to the renderer and update the *InitSkybox()* function as shown in (Listing 6.71).

Listing 6.71: Graphics/Renderer.h

```

EMPY_INLINE void InitSkybox(Skybox& sky, Texture& texture,
int32_t size)
{
    // generate brdf map
    sky.BrdfMap = m_Brdf->Generate(512);

    // generate environment cube map
    sky.CubeMap = m_SkyMap->Generate(texture, m_SkyboxMesh,
size);

    // generate diffuse irradiance map
    sky.IrradMap = m_Irrad->Generate(sky.CubeMap,

```

```

m_SkyboxMesh, 32);

    // generate specular irradiance map
    sky.PrefilMap = m_Prefil->Generate(sky.CubeMap,
m_SkyboxMesh, 128);
}

EMPTY_INLINE void DrawSkybox(Skybox& sky, Transform3D&
transform)
{
    // render skybox
    m_Skybox->Draw(m_SkyboxMesh, sky.CubeMap, transform);

    // set all environment maps
    m_Pbr->SetEnvMaps(sky.IrradMap, sky.PrefilMap,
sky.BrdfMap);
}

```

PBR Shader Update

Lastly, we need to use these textures in the PBR shader to compute the specular contribution.

Listing 6.72: Resources/Shaders/pbr.gls

```

//... same as before

// enviroment maps
uniform samplerCube u_prefilMap;
uniform samplerCube u_irradMap;
uniform sampler2D u_brdfMap;

// compute ambient lights

```

```

vec3 ComputeAmbientLight(vec3 N, vec3 V, vec3 F0, vec3
albedo, float roughness, float metallic)
{
    // angle between surface normal and light direction.
    float cosTheta = max(0.0, dot(N, V));

    // get diffuse contribution factor
    vec3 F = FresnelSchlick(cosTheta, F0);
    vec3 kd = mix(vec3(1.0) - F, vec3(0.0), metallic);

    // irradiance map contains exitant radiance
    vec3 diffuseIBL = kd * albedo * texture(u_irradMap,
N).rgb;

    // sample pre-filtered map at correct mipmap level.
    int mipLevels = 5;
    vec3 Lr = 2.0 * cosTheta * N - V;
    vec3 Ks = textureLod(u_prefilMap, Lr, roughness *
mipLevels).rgb;

    // split-sum approx.factors for Cook-Torrance specular
    BRDF.
    vec2 brdf = texture(u_brdfMap, vec2(cosTheta,
roughness)).rg;
    vec3 specularIBL = (F0 * brdf.x + brdf.y) * Ks;

    return (diffuseIBL + specularIBL);
}

```

The function `ComputeAmbientLight()` now also computes the specular contribution using both the prefiltered and the BRDF maps. The final result is the combination of the diffuse and specular components.

PBR Shader Class Update

Update the PBR shader as depicted in the following code snippet. As you can see, we have added additional arguments to the `SetEnvMaps()` to match our requirement to also set the prefiltered map and the BRDF map.

Listing 6.73: Graphics/Renderer.h

```
struct PbrShader : Shader
{
    EMPY_INLINE PbrShader(const std::string& filename) :
    Shader(filename)
    {
        /* ... same as before ... */
        u_PrefilMap = glGetUniformLocation(m_ShaderID,
"u_prefilMap");
        u_IrradMap = glGetUniformLocation(m_ShaderID,
"u_irradMap");
        u_BrdfMap = glGetUniformLocation(m_ShaderID,
"u_brdfMap");
    }

    EMPY_INLINE void Draw(Model3D& model, PbrMaterial&
material, Transform3D& transform)
    {
        glUniformMatrix4fv(u_Model, 1, GL_FALSE,
glm::value_ptr(transform.Matrix()));
        glUniform3fv(u_Albedo, 1, &material.Albedo.x);
        glUniform1f(u_Roughness, material.Roughness);
        glUniform1f(u_Metallic, material.Metallic);
    }
}
```

```

        int32_t unit = 3; // <-- starts at unit 3 !!!!
        bool useMap = false;

        /* ... same as before ... */
    }

EMPTY_INLINE void SetEnvMaps(uint32_t irrad, uint32_t
prefil, uint32_t brdf)
{
    glUseProgram(m_ShaderID);

    // irradiance map
    glBindTexture(GL_TEXTURE0); // <--- texture unit 0
    glBindTexture(GL_TEXTURE_CUBE_MAP, irrad);
    glUniform1i(u_IrradMap, 0);

    // prefiltered map
    glBindTexture(GL_TEXTURE1); // <--- texture unit 1
    glBindTexture(GL_TEXTURE_CUBE_MAP, prefil);
    glUniform1i(u_PrefilMap, 1);

    // BRDF Map
    glBindTexture(GL_TEXTURE2); // <--- texture unit 2
    glBindTexture(GL_TEXTURE_2D, brdf);
    glUniform1i(u_BrdfMap, 2);
}

/* ... same as before ... */

private:
/* ... same as before ... */

uint32_t u_PrefilMap = 0u;
uint32_t u_IrradMap = 0u;

```

```
    uint32_t u_BrdfMap = 0u;  
}
```

Now that our environment maps are using the first three texture units [0, 2], we also have to make sure that the `Draw()` function is updated accordingly to use texture units starting from three [3, N].

If you compile and run without a light source, you get the following result:



Figure 6.31: Diffuse and Specular (No Gamma Correction)

6.5.4 HDR Tone Mapping

You might have noticed that the scene rendered in (Figure 6.31) is quite dark, considering the environment has daily light. Well, it is because we

need to apply some "Gamma correction.". Gamma correction is a crucial concept in computer graphics that addresses the nonlinear relationship between the intensity of light emitted by a display and the corresponding brightness perceived by the human eye. The gamma correction process is implemented to compensate for this non-linearity and ensure that displayed images appear visually consistent.

Mathematical Equations

In mathematical terms, the relationship between the input voltage (or digital intensity value) and the perceived brightness on a monitor is often modeled using the power-law function:

$$I_{\text{out}} = I_{\text{in}}^{\gamma}$$

Here, I_{out} is the output intensity (brightness), I_{in} is the input intensity (original color value), and γ is the gamma value. The typical gamma value is approximately 2.2 for most display systems.

Gamma correction is performed by either encoding or decoding the intensity values using the reciprocal of the gamma value. When encoding, the formula becomes:

$$I_{\text{out}} = I_{\text{in}}^{1/\gamma}$$

And when decoding, it is:

$$I_{\text{out}} = I_{\text{in}}^{\gamma}$$

This correction helps in achieving perceptually uniform brightness levels across different shades, as the human visual system is more sensitive to changes in darker areas than brighter ones.

Final Shader Update

This is done in the following code snippet.
See ([Listing 6.74](#)).

Listing 6.74: Resources/Shaders/final.gls

```
#version 330 core
layout (location = 0) in vec4 a_quad;
out vec2 uvs;

void main()
{
    gl_Position = vec4(a_quad.x, a_quad.y, 0.0, 1.0);
    uvs = vec2(a_quad.z, a_quad.w);
}

++VERTEX++
```

```

#version 330 core
out vec4 out_fragment;
in vec2 uvs;

const float GAMMA = 2.5;
const float EXPOSURE = 10.0;
uniform sampler2D u_map;

void main()
{
    // sample color from map
    vec3 result = pow(texture(u_map, uvs).rgb, vec3(GAMMA));

    // process exposure
    result = vec3(1.0) - exp(-result * EXPOSURE);

    // gamma correction
    result = pow(result, vec3(1.0 / max(GAMMA, 0.000001)));

    // fragment color
    out_fragment = vec4(result, 1.0);
}

++FRAGMENT++

```

(Figure [6.32](#)) is the result you will get with this shader code. You will notice that the skybox and the sphere are brighter. This is still without a light source. You can modify the `GAMMA` and `EXPOSURE` values to see how they affect the final result.



Figure 6.32: Gamma Correction Applied

6.6 Rendering Shadows

Shadows play a crucial role in enhancing the realism and visual depth of scenes in computer graphics. They contribute to a more immersive and believable environment by providing important visual cues related to lighting and spatial relationships.

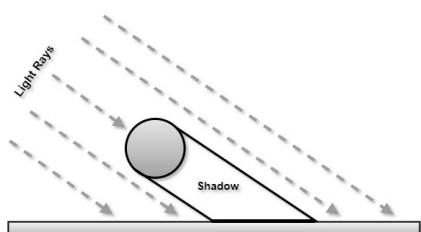


Figure 6.33: Shadow Projection

6.6.1 Shadow Mapping Technique

"Shadow Mapping" is a widely used technique for rendering realistic shadows in computer graphics. It involves creating a depth map, known as the shadow map, from the perspective of the light source to determine which areas of the scene are in shadow. The basic idea is to compare the depth values between the shadow map and the actual scene during rendering to determine if a point is in shadow or not. Here is a description of the steps involved in the implementation of shadow mapping with OpenGL:

- **Render the Scene from Light Perspective:** We start by setting up a separate frame buffer to hold depth values. We then use an orthographic projection to render the scene from the light's viewpoint and store the depth values in the frame buffer depth attachment. An orthographic projection is a type of graphical projection used in computer graphics where objects are projected onto a 2D plane without accounting for perspective. In an orthographic projection, parallel lines remain parallel, and the size of objects does not change with distance. This projection is commonly employed for technical drawings and architectural illustrations, as it preserves accurate measurements and proportions. In contrast,

perspective projection aims to simulate how objects appear in the real world by accounting for the effects of distance and convergence. Objects that are farther away appear smaller, and parallel lines converge towards a vanishing point.

- **Render the Scene from Camera Perspective:** We proceed by rendering the scene from the camera's perspective as usual, but now we also pass the light's space matrix to the shader. This light's space matrix is simply the multiplication of the projection and the view matrix used to render depth values in the previous step.
- **Shadow Map Comparison in the Fragment Shader:** Then we calculate the position of the fragment in light space using the light's space matrix in the fragment shader, and we continue by sampling the corresponding depth value from the shadow map, also known as the depth map. Compare the sampled depth value with the value calculated with the light's space matrix. If the calculated depth is greater, the fragment is in shadow; otherwise, it is lit.
- **Handling Shadow Acne:** Shadow acne, also known as self-shadowing artifacts or Peter Panning, manifests as undesired high-frequency noise or flickering in rendered shadows. It primarily results from floating-point precision errors in the depth values used during the shadow map comparison.

To address shadow acne, we can apply a small bias to the calculated depth before comparison. `depth = depth - bias.`

- **Implement Soft Shadows (Optional):** The last stage involves achieving softer shadows that are more appealing to the eyes. Techniques such as Percentage-Closer Filtering (PCF) [Meiri \[a\]](#) can be used for this purpose. PCF involves sampling multiple points within the shadow map and averaging the results. This makes the resulting shadows have less jaggy edges.

This guide provides a concise overview of the shadow mapping process, covering key steps from rendering the shadow map to implementing the fragment shader logic. Let us now implement this feature in the engine.

6.6.2 Applying Shadow Mapping

As mentioned above, we need to create a texture attachment, also known as a depth attachment, in an additional frame buffer to store our scene's depth values, and we also have to create a new shader to render the depth values in the frame buffer attachment. We could easily create both the shader and the frame buffer in separate files, but we will instead combine the two in the shader file to keep

things organized and simple. Let us start with the shader's GLSL code, as it is quite simple.

GLSL Shader

You can see from the code in (Listing 6.75) that the shadow shader is quite simple. All the work is done in the vertex shader. The uniform `u_lightSpace` is the matrix we mentioned earlier, which is basically the multiplication of the orthographic projection matrix with the view matrix from the light perspective.

Listing 6.75: Resources/Shaders/shadow.gls

```
#version 330 core
layout (location = 0) in vec3 a_position;

uniform mat4 u_lightSpace;
uniform mat4 u_model;

void main()
{
    gl_Position = u_lightSpace * u_model * vec4(a_position,
1.0f);
}

++VERTEX++

#version 330 core

void main()
```

```
{  
    // No need to do anything  
}  
  
++FRAGMENT++
```

The model matrix, as you know, represents the transformation of the object we are trying to render the depth value of. You will see that the fragment shader has no code. That is because the current depth value will be rendered in the frame buffer's depth attachment automatically and does not require us to do that manually. This will be defined in the frame buffer in a bit.

Shader Abstraction

As usual, we need to create a C++ shader class to properly interact with the actual GLSL shader. This is done in the following code snippet ([Listing 6.76](#)).

Listing 6.76: Graphics/Shaders/Shadow.
h

```
#pragma once  
#include "Shader.h"  
  
namespace EmPy  
{  
    struct ShadowShader : Shader  
    {
```

```

EMPTY_INLINE ShadowShader(const std::string& path):
Shader(path)
{
    u_LightSpace = glGetUniformLocation(m_ShaderID,
"u_lightSpace");
    u_Model = glGetUniformLocation(m_ShaderID,
"u_model");

    // create depth texture
    glGenTextures(1, &m_DepthMap);
    glBindTexture(GL_TEXTURE_2D, m_DepthMap);
    glTexImage2D(GL_TEXTURE_2D, 0,
GL_DEPTH_COMPONENT,
MapSize, MapSize, 0, GL_DEPTH_COMPONENT,
GL_FLOAT, NULL);

    // set texture parameters
    glTexParameteri(GL_TEXTURE_2D,
GL_TEXTURE_MAG_FILTER, GL_NEAREST);
    glTexParameteri(GL_TEXTURE_2D,
GL_TEXTURE_MIN_FILTER, GL_NEAREST);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S,
GL_REPEAT);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T,
GL_REPEAT);

    // create frame buffer
    glGenFramebuffers(1, &m_FrameBuffer);
    glBindFramebuffer(GL_FRAMEBUFFER, m_FrameBuffer);

    // attach to frame buffer
    glFramebufferTexture2D(GL_FRAMEBUFFER,
GL_DEPTH_ATTACHMENT, GL_TEXTURE_2D, m_DepthMap, 0);

    // no drawing nor reading
    glDrawBuffer(GL_NONE);
}

```

```

        glReadBuffer(GL_NONE);

        // check frame buffer
        if (glCheckFramebufferStatus(GL_FRAMEBUFFER) != GL_FRAMEBUFFER_COMPLETE)
        {
            EMPTY_ERROR("CreateDepthBuffer() Failed!");
        }

        // unbind frame buffer
        glBindFramebuffer(GL_FRAMEBUFFER, 0);
    }

    EMPTY_INLINE void Draw(Model3D& model, Transform3D&
transform)
    {
        glUniformMatrix4fv(u_Model, 1, GL_FALSE,
glm::value_ptr(transform.Matrix()));
        model->Draw(GL_TRIANGLES);
    }

    EMPTY_INLINE void BeginFrame(const glm::mat4&
lightSpaceMtx)
    {
        // bind shadow shader
        glUseProgram(m_ShaderID);

        // set view projection matrix
        glUniformMatrix4fv(u_LightSpace, 1, GL_FALSE,
glm::value_ptr(lightSpaceMtx));

        // bind target frame buffer
        glBindFramebuffer(GL_FRAMEBUFFER, m_FrameBuffer);

        // set viewport a clear buffer
        glViewport(0, 0, MapSize, MapSize);
    }
}

```

```

        glClear(GL_DEPTH_BUFFER_BIT);
        glEnable(GL_DEPTH_TEST);
    }

EMPTY_INLINE uint32_t GetDepthMap()
{
    return m_DepthMap;
}

EMPTY_INLINE void EndFrame()
{
    glDisable(GL_DEPTH_TEST);
    glBindFramebuffer(GL_FRAMEBUFFER, 0);
    glUseProgram(0);
}

EMPTY_INLINE ~ShadowShader()
{
    glDeleteFramebuffers(1, &m_FrameBuffer);
    glDeleteTextures(1, &m_DepthMap);
}

private:
    uint32_t m_FrameBuffer = 0u;
    uint32_t m_DepthMap = 0u;
    int32_t MapSize = 1024;

    uint32_t u_LightSpace = 0u;
    uint32_t u_Model = 0u;
};

}

```

Starting in the constructor, you observe how we are gathering the uniform locations, and then we proceed by creating a frame buffer with a depth attachment,

which is simply a 2D texture, as we did for the color buffer. You will note how we are explicitly using the `glDrawBuffer(GL_NONE)` and `glReadBuffer(GL_NONE)` to tell OpenGL that we are not planning to read nor write to this frame buffer from C++. This is not required, but it is good for performance.

Following the constructor is the `Draw()` function, which is similar to the one in the PBR shader. The `BeginFrame()` function begins the rendering process of the depth values. We also make sure we destroy the frame buffer as well as the depth map in the class destructor. There is really nothing new in this code. Lastly, in the member variable section, you will find `m_MapSize`, which defines the size of the depth map and other common variables.

PBR Shader Update

Assuming for now that the depth exists, we can update the PBR shader to leverage the depth map as well as the light space matrix to compute the shadowing effect of fragments. This is done in the code snippet depicted in (Listing 6.77).

Listing 6.77: Resources/Shaders/pbr.gls
|

```
/* ... same as before ... */

// shadow mapping uniforms
uniform sampler2D u_depthMap;
uniform mat4 u_lightSpace;

// ... same sa before

float ComputeShadow()
{
    // compute light space fragment position
    vec4 position = u_lightSpace * vec4(vertex.Position, 1.0);

    // normalize projection coordinates
    vec3 uvs = (position.xyz / position.w) * 0.5 + 0.5;

    // sample depth value from shadow map
    float depth = texture(u_depthMap, uvs.xy).r;

    // compare to light space depth value
    return position.z > depth ? 1.0 : 0.0;
}

// ... same sa before

// main function
void main()
{
    /* ... same as before ... */

    // lights contribution
    vec3 result = ComputeAmbientLight(N, V, F0, albedo,
roughness, metallic);
    result += ComputeDirectLights(N, V, F0, albedo, roughness,
metallic);
}
```

```

    result += ComputePointLights(N, V, F0, albedo, roughness,
metallic);
    result += ComputeSpotLights(N, V, F0, albedo, roughness,
metallic);

    // occlusion and emissive
    result = (result * occlusion) + emissive;

    // compute shadow value <----- added
    result *= (1.0 - ComputeShadow());
}


```

There are two uniforms added: `u_depthMap`, and `u_lightSpace` which we have already introduced. The newly added function called `ComputeShadow()` calculates whether or not the current fragment is shadowed or lit.

PBR Shader Class Update

We are also required to update the PBR shader class to reflect the changes made in the GLSL shader.

Listing 6.78: Graphics/Shaders/PBR. h

```

struct PbrShader : Shader
{
    EMPTY_INLINE PbrShader(const std::string& filename) :

```

```
Shader(filename)
{
    /* ... same as before ... */

    u_LightSpace = glGetUniformLocation(m_ShaderID,
"u_lightSpace");
    u_DepthMap = glGetUniformLocation(m_ShaderID,
"u_depthMap");
}

EMPTY_INLINE void Draw(Model3D& model, PbrMaterial&
material, Transform3D& transform)
{
    // ...

    int32_t unit = 4; // <-- starts at unit 4

    // ...
}

EMPTY_INLINE void SetEnvMaps(uint32_t irrad, uint32_t
prefil, uint32_t brdf, uint32_t depthMap)
{
    glUseProgram(m_ShaderID);

    // irradiance map
    glBindTexture(GL_TEXTURE0);
    glUniform1i(u_IrradMap, 0);

    // prefiltered map
    glBindTexture(GL_TEXTURE1);
    glUniform1i(u_PrefilMap, 1);

    // BRDF Map
```

```

        glActiveTexture(GL_TEXTURE2);
        glBindTexture(GL_TEXTURE_2D, brdf);
        glUniform1i(u_BrdfMap, 2);

        // Depth Map      <---- added
        glActiveTexture(GL_TEXTURE3);
        glBindTexture(GL_TEXTURE_2D, depthMap);
        glUniform1i(u_DepthMap, 3);
    }

EMPTY_INLINE void SetLightSpaceMatrix(const glm::mat4&
lightSpaceMtx)
{
    // set view projection matrix
    glUniformMatrix4fv(u_LightSpace, 1, GL_FALSE,
glm::value_ptr(lightSpaceMtx));
}

private:
    uint32_t u_LightSpace = 0u;
    uint32_t u_DepthMap = 0u;
}

```

The two major changes made here are found in the functions `SetEnvMaps()` and `SetLightSpaceMatrix()`. As the name of the latest suggests, its role is to set the light space matrix uniform found in the GLSL code. In the `SetEnvMaps()` we have added another argument along with a new section to set the texture unit of the depth map, which is "3". This requires us to update the `Draw()` function to make sure that the material texture unit starts from unit "4".

Make sure your texture units are set accordingly, as depicted in (Listing 6.78), or you will run into a lot of troubles.

Renderer Class Update

We can simply create this shader in the renderer class to perform the depth mapping for us. So, go ahead and update your renderer class as depicted in the following code snippet (Listing 6.79).

Listing 6.79: Graphics/Renderer.h

```
struct GraphicsRenderer
{
    EMPLY_INLINE GraphicsRenderer(int32_t width, int32_t height)
    {
        /* ... same as before ... */
        m_Shadow = std::make_unique<ShadowShader>
("Resources/Shaders/shadow.glsl");
    }

    // render the skybox and set env. maps and depth map
    EMPLY_INLINE void DrawSkybox(Skybox& sky, Transform3D& transform)
    {
        // render skybox
        m_Skybox->Draw(m_SkyboxMesh, sky.CubeMap, transform);

        // set env. map along with depth map
        m_Pbr->SetEnvMaps(sky.IrradMap, sky.PrefilMap,
```

```

    sky.BrdfMap, m_Shadow->GetDepthMap()); // <--- added
}

// helps render depth value
EMPTY_INLINE void DrawDepth(Model3D& model, Transform3D&
transform)
{
    m_Shadow->Draw(model, transform);
}

// starts the depth rendering process
EMPTY_INLINE void BeginShadowPass(const glm::vec3&
LightDir)
{
    // prepare projection and view mtx
    float nearPlane = 1.0f;
    float farPlane = 10.0f;
    static auto proj = glm::ortho(-10.0f, 10.0f, -10.0f,
10.0f, nearPlane, farPlane);
    auto view = glm::lookAt(LightDir, glm::vec3(0.0f,
0.0f, 0.0f),
glm::vec3(0.0f, 1.0f, 0.0f));

    // calculate light space matrix
    auto lightSpaceMtx = (proj * view);

    // set pbr shader light space mtx
    m_Pbr->Bind();
    m_Pbr->SetLightSpaceMatrix(lightSpaceMtx);

    // begin depth rendering
    m_Shadow->BeginFrame(lightSpaceMtx);
}

// ends the depath rendering process
EMPTY_INLINE void EndShadowPass()

```

```

{
    m_Shadow->EndFrame();
}

private:
/* ... same as before ... */

std::unique_ptr<ShadowShader> m_Shadow; // <--- added
}

```

By just looking at the code in (Listing 6.79), one could easily understand the changes applied here. You will note in the `BeginShadowPass()` function how the light's space matrix is computed and sent to the GPU. The code snippet `glm::ortho(-10.0f, 10.0f, -10.0f, 10.0f, nearPlane, farPlane)` is using the GLM library to create an orthographic projection matrix. The provided parameters specify the dimensions of the orthographic projection box. In this case, the left, right, bottom, and top values are set to -10.0f, 10.0f, -10.0f, and 10.0f, respectively. These values define the extent of the box along the x and y axes. The `nearPlane` and `farPlane` parameters determine the distance to the near and far clipping planes, respectively.

The resulting orthographic projection matrix, denoted below as **M**, can be expressed as:

$$\mathbf{M} = \begin{bmatrix} \frac{2}{\text{right-left}} & 0 & 0 & -\frac{\text{right+left}}{\text{right-left}} \\ 0 & \frac{2}{\text{top-bottom}} & 0 & -\frac{\text{top+bottom}}{\text{top-bottom}} \\ 0 & 0 & -\frac{2}{\text{far-near}} & -\frac{\text{far+near}}{\text{far-near}} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The view matrix simply tells the shader which part of the screen to focus on in order to capture depth values. We will only cast shadows for directional light, but the same process can be applied for all sorts of light sources. Another reason for this is that shadow mapping for multiple light sources can be quite expensive. The issue lies in the fact that each light source is supposed to render the depth value of each object in the scene from its perspective. There are definitely strategies to mitigate this limitation, but these are a bit complex to address in this book and are not really our main focus as this book is mainly about building a game engine, not just a graphic renderer. Like the sun in the daytime or the moon in the night, there is usually only one directional light in a scene, so it makes sense to cast shadows for every object utilizing this one light source. The performance will not drop, and the outcome will still be of high quality.

Application Class Update

Now, go to your "Application.h" header file and add the following changes depicted in (Listing [6.80](#)) to test

this feature.

Listing 6.80: Application/Application.h

```
EMPTY_INLINE Application()
{
    EMPTY_INLINE void RunContext()
    {
        // load environment hdr texture
        auto skymap = std::make_shared<Texture2D>
("Resources/Textures/HDRs/Sky.hdr", true, true);

        // load models
        auto sphereModel = std::make_shared<Model>
("Resources/Models/sphere.fbx");
        auto cubeModel = std::make_shared<Model>
("Resources/Models/cube.fbx");

        // create scene camera
        auto camera = CreateEntt<Entity>();
        auto& tf = camera.Attach<TransformComponent>
().Transform;
        tf.Translate.y = 4.5f;
        tf.Translate.z = 20.0f;
        camera.Attach<CameraComponent>();

        // create skybox entity
        auto skybox = CreateEntt<Entity>();
        skybox.Attach<TransformComponent>();
        skybox.Attach<SkyboxComponent>();

        // create point light 1
        auto slight = CreateEntt<Entity>();
        slight.Attach<DirectLightComponent>().Light.Intensity
```

```

= 0.0f;

        auto& stp = slight.Attach<TransformComponent>
() .Transform;
        //stp.Rotation = glm::vec3(0.0f, 0.5f, -1.0f);
        stp.Rotation = glm::vec3(0.0f, 1.5f, -2.0f);

        // create sphere entity
        auto sphere = CreateEntt<Entity>();
        sphere.Attach<ModelComponent>().Model = sphereModel;
        auto& ts = sphere.Attach<TransformComponent>
() .Transform;
        ts.Translate.y = 6.0f;
        ts.Translate.z = -0.65f;
        ts.Scale *= 6.0f;

        // create plane entity
        auto cube = CreateEntt<Entity>();
        cube.Attach<ModelComponent>().Model = cubeModel;
        auto& tc = cube.Attach<TransformComponent>
() .Transform;
        tc.Scale.x = 20.0f;
        tc.Scale.z = 20.0f;
        tc.Scale.y = 0.5f;

        // generate enviroment maps
        EnttView<Entity, SkyboxComponent>([this, &skymap]
(auto entity, auto& comp)
{
    m_Context->Renderer->InitSkybox(comp.Sky, skymap,
2048);
});

        // application main loop
while(m_Context->Window->PollEvents())
{
    // render depth map
}

```

```

        RenderSceneDepth(); // ----- added

        // start new frame
        m_Context->Renderer->NewFrame();

        // ... remains unchanged

        m_Context->Renderer->EndFrame();

        // update layers
        for(auto layer : m_Context->Layers)
        {
            layer->OnUpdate();
        }

        // show frame to screen
        m_Context->Renderer->ShowFrame();
    }

}

private:
    // render the scene depth map
    EMPTY_INLINE void RenderSceneDepth()
    {
        EnttView<Entity, DirectLightComponent>([this] (auto
light, auto&)
        {
            // light direction
            auto& lightDir = light.template
Get<TransformComponent>().Transform.Rotation;

            // begin rendering
            m_Context->Renderer->BeginShadowPass(lightDir);

            // render depth
            EnttView<Entity, ModelComponent>([this,

```

```

&lightDir] (auto entity, auto& comp)
{
    auto& transform = entity.template
Get<TransformComponent>().Transform;
    m_Context->Renderer->DrawDepth(comp.Model,
transform);
}

// finalize frame
m_Context->Renderer->EndShadowPass();
}

}
}

```

Important to note here is the newly added `RenderSceneDepth()` function, which renders the depth value for each entity in the scene. This function is called in the main loop before the proper rendering begins.

Here is the result you should get when compiling and running the project:



Figure 6.34:

Shadow Mapping Result

You should see a dark dot at the bottom of your sphere.

**You can quickly check the content of the depth map by simply rendering it to the screen.
Following are optional steps to achieve that:**

Showing Depth Map (Optional)

Use the subsequent steps to show your depth map content to the screen as depicted in (Figure [6.35](#)).

Listing 6.81: Graphics/Renderer.
h

```
EMPTY_INLINE void ShowFrame()
{
    //m_Final->Show(m_Frame->GetTexture());
    m_Final->Show(m_Shadow->GetDepthMap());
}
```

And you should also modify the final shader fragment output color accordingly:

Listing 6.82: Resources/Shaders/final.gls
|

```
/* ... same as before ... */

void main()
{
    // get depth value
    float depth = texture(u_map, uvs).r;

    // set output color
    out_fragment = vec4(depth, depth, depth, 1.0);
}
```

You will get the following result:

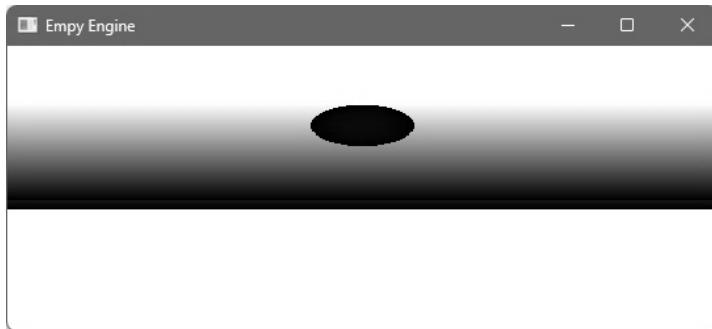


Figure 6.35: Scene Depth Map

You can use key input to move the sphere and see how the shadow is affected.

Reset the final shader main function back to its original state before continuing. This was just for testing.

6.6.3 Improving Shadow Quality

We discussed handling shadow acne by introducing a bias factor when describing the steps for achieving shadow mapping. In the image depicted in (Figure 6.34), there is no visible acne to see, but moving the light source can result in situations such as that depicted in (Figure 6.36). We stated that this issue came from the comma precision of float values. Another reason for shadow acne is the discreteness of shadow maps, which are composed of samples while surfaces are continuous.



Figure 6.36: Shadow Acne

Solving Shadow Acne

You can add a bias to the `ComputeShadow()` function as depicted below:

Listing 6.83: Resources/Shaders/pbr.gls

```
float ComputeShadow()
{
    float bias = 0.005;
    vec4 position = u_lightSpace * vec4(vertex.Position, 1.0);
    vec3 uvs = (position.xyz / position.w) * 0.5 + 0.5;
    float depth = texture(u_depthMap, uvs.xy).r;
    return (position.z - bias) > depth ? 1.0 : 0.0;
}
```

While this is sufficient to solve the problem, it is not very dynamic to have a hard-coded number like that for all circumstances. One option to improve this is to add an additional uniform to adjust the bias value from the CPU, which is relatively straightforward to do.

Peter Panning Effect

Shadow Peter Panning, also known as depth biasing or simply "Peter Panning," is an artifact in shadow mapping that manifests as a shadow hovering slightly above the surface, creating an unrealistic visual effect reminiscent of the character Peter Pan's shadow. This phenomenon occurs when a bias is applied to the depth values during shadow map comparisons. If the

bias is too aggressive, it can cause shadows to detach from the underlying surfaces.

This artifact is solved by simply culling (Figure 6.37) the front face of the object when rendering it. As we only require depth values, the choice between extracting depth from the front faces or back faces of solid objects is inconsequential. Utilizing the depths from the back faces does not yield inaccuracies since visibility inside objects is not possible regardless.

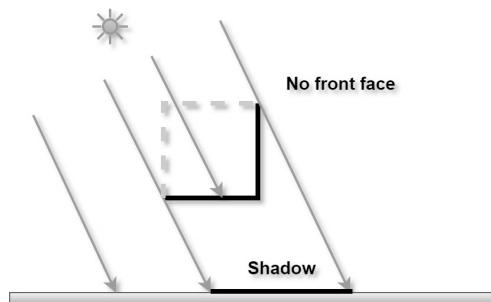


Figure 6.37: Face Culling

This is achieved in the following code:

Listing 6.84: Graphics/Shaders/Shadow.h

```
EMPTY_INLINE void Draw(Model3D& model, Transform3D& transform)
{
    glUniformMatrix4fv(u_Model, 1, GL_FALSE,
glm::value_ptr(transform.Matrix()));
```

```
    glCullFace(GL_FRONT);
    model->Draw(GL_TRIANGLES);
    glCullFace(GL_BACK);
}
```

Percentage-Closer Filtering

If you zoom in enough or use a lower-resolution depth map, you will notice that the edges of the shadow are not smooth but quite jaggy. This can be improved using a technique called PCF. As introduced earlier, this technique helps compute the average of the depth values around a sampled depth value to get smooth shadow results.

Following is a depiction of how this is done:

Listing 6.85: Resources/Shaders/pbr.gls

```
float ComputeShadow()
{
    vec4 position = u_lightSpace * vec4(vertex.Position, 1.0);
    vec3 coords = (position.xyz / position.w) * 0.5 + 0.5;

    // pixel size (1024 -> map size)
    float pixelSize = 1.0/1024;
    float shadow = 0.0;
    float bias = 0.005;

    // compute average pcf
    for(int x = -1; x <= 1; ++x)
```

```

{
    for(int y = -1; y <= 1; ++y)
    {
        float depth = texture(u_depthMap, coords.xy + vec2(x,
y) * pixelSize).r;
        shadow += (position.z - bias) > depth ? 1.0 : 0.0;
    }
}
shadow /= 9.0;

// no shadow if outside of orthographic plane
if(coords.z > 1.0)
    shadow = 0.0;

return shadow;
}

```

There are still small additional changes that can be applied to improve the look of the shadow but that's yours to discover. You can find additional resources about **Shadow Mapping** here: [de Vries \[b\]](#) [Meiri \[b\]](#).

6.7 Bloom Effect

The Bloom effect in computer graphics is a post-processing technique designed to simulate the visual phenomenon of light scattering and blooming around bright areas in an image. It enhances the perception of brightness and gives a visually appealing glow to intense light sources or high-contrast regions.



Figure 6.38: No Bloom vs. With Bloom

Achieving the Bloom effect typically involves the following steps:

- ☞ **Render Scene to a Frame buffer:** We begin by rendering the scene to the frame buffer as we have been doing. One attachment stores the original scene's colors, and additional attachments are used to store the scene's brightness.
- ☞ **Extract Bright Areas:** While rendering objects, we will check whether their color is above a specific brightness threshold and store it in the brightness attachment.
- ☞ **Blur Bright Areas:** Then we perform a horizontal and vertical blur on the bright areas using a separable convolution kernel. This step enhances the glow effect by spreading the bright pixels. Implementing this blur can be done through multiple rendering passes with

frame buffers or by employing specific blur algorithms, such as Gaussian blur.

☞ **Combine Blurred Image with Original Scene:**

Finally, we combine the blurred image with the original scene image in the final stage of rendering, giving more emphasis to the bright areas. This combination is often performed using an additive-blending operation.

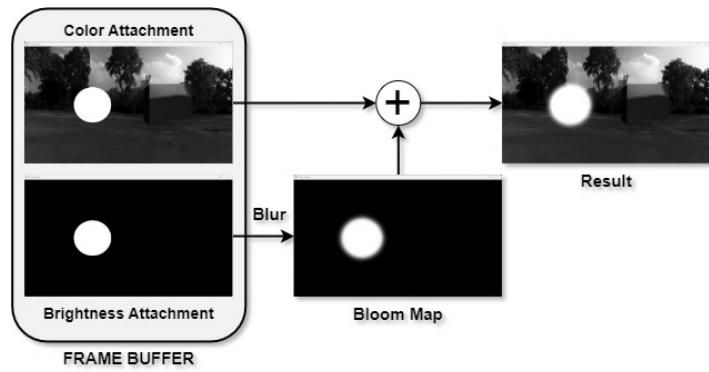


Figure 6.39: Bloom Flow Diagram

(Figure 6.39) is a visual representation of these steps. Let us implement this in the rendering pipeline.

6.7.1 Implementing Bloom Effect

Similar to what we did for shadow mapping, we will also create a shader to generate a blurred version of

the brightness map. We also have to extend the frame buffer by adding a brightness attachment.

Extending the Frame Buffer

Looking at the code snippet depicted below in (Listing 6.86), you will note how the brightness attachment is added to the frame buffer. You will see in the constructor that we now have two target attachments. The function `CreateBrightnessAttachment` helps create the new attachment and is quite similar to the `CreateColorAttachment()`.

Listing 6.86: Graphics/Buffers/Frame.
h

```
/* ... same as before ... */

struct FrameBuffer
{
    EMPLY_INLINE FrameBuffer(int32_t width, int32_t height) :
m_Width(width), m_Height(height)
{
    glGenFramebuffers(1, &m_BufferID);
    glBindFramebuffer(GL_FRAMEBUFFER, m_BufferID);

    CreateColorAttachment();
    CreateBrightnessAttachment(); // <--- create
attachment
    CreateRenderBuffer();

    // Attachment Targets
```

```

    uint32_t attachments[2] =
    {
        GL_COLOR_ATTACHMENT0,
        GL_COLOR_ATTACHMENT1, // <-- add attachment
    };

    glDrawBuffers(2, attachments);

    // check frame buffer
    if (glCheckFramebufferStatus(GL_FRAMEBUFFER) !=
GL_FRAMEBUFFER_COMPLETE)
    {
        EMPLY_ERROR("glCheckFramebufferStatus() Failed!");
    }

    // unbind frame buffer
    glBindFramebuffer(GL_FRAMEBUFFER, 0);
}

EMPLY_INLINE void Resize(int32_t width, int32_t height)
{
    // update size
    m_Width = width;
    m_Height = height;

    // Resize Color Buffer
    glBindTexture(GL_TEXTURE_2D, m_Color);
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA16F,
m_Width, m_Height, 0, GL_RGBA, GL_FLOAT, NULL);

    // Resize Brightness Buffer // <--- resize
attachment
    glBindTexture(GL_TEXTURE_2D, m_Brightness);
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA16F,
m_Width, m_Height, 0, GL_RGBA, GL_FLOAT, NULL);
}

```

```

        // Resize Render Buffer
        glBindRenderbuffer(GL_RENDERBUFFER, m_Render);
        glRenderbufferStorage(GL_RENDERBUFFER,
GL_DEPTH_COMPONENT24, m_Width, m_Height);

        glBindRenderbuffer(GL_RENDERBUFFER, 0);
        glBindTexture(GL_TEXTURE_2D, 0);
    }

EMPTY_INLINE uint32_t GetBrightnessMap()
{
    return m_Brightness;
}

EMPTY_INLINE int32_t Height() { return m_Height; }

EMPTY_INLINE int32_t Width() { return m_Width; }

EMPTY_INLINE ~FrameBuffer()
{
    glDeleteTextures(1, &m_Color);
    glDeleteTextures(1, &m_Brightness);
    glDeleteRenderbuffers(1, &m_Render);
    glDeleteFramebuffers(1, &m_BufferID);
}

/* ... same as before ... */

private:
    // creates the attachment
EMPTY_INLINE void CreateBrightnessAttachment()
{
    glGenTextures(1, &m_Brightness);
    glBindTexture(GL_TEXTURE_2D, m_Brightness);
    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_S,
GL_CLAMP_TO_EDGE);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T,
GL_CLAMP_TO_EDGE);
        glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA16F, m_Width,
m_Height, 0, GL_RGBA, GL_FLOAT, NULL);
        glFramebufferTexture2D(GL_FRAMEBUFFER,
GL_COLOR_ATTACHMENT1, GL_TEXTURE_2D, m_Brightness, 0);
    }

private:
    uint32_t m_Brightness = 0u; // <-- attachment id
    uint32_t m_BufferID = 0u;
    uint32_t m_Render = 0u;
    uint32_t m_Color = 0u;
    int32_t m_Height = 0;
    int32_t m_Width = 0;
}

```

We also want to make sure the new attachment is also resized whenever the resize function is called. The same thing is done in the destructor to make sure that the memory is released when the frame buffer object is destroyed. That is all we need to do in the frame buffer.

Extending the PBR Shader

Now that the frame buffer has multiple render targets, we need to make the PBR shader aware of that by

making sure that each fragment is rendered to the proper target. This is done in the following code snippet:

Listing 6.87: Resources/Shaders/pbr.gls

```
/* ... same as before ... */

++VERTEX++

#version 330 core
layout (location = 0) out vec4 out_fragment; // attachment
0
layout (location = 1) out vec4 out_brightness; //attachment
1

// constants
const vec3 BLOOM_THRESHOLD = vec3(0.2126, 0.7152, 0.0722);
#define PI 3.14159265358979323846
#define MAX_LIGHTS 10

/* ... same as before ... */

// main function
void main()
{
    /* ... same as before ... */

    // occlusion and emissive
    result = (result * occlusion) + emissive;

    // compute shadow value
    result *= (1.0 - ComputeShadow());
```

```

// output brightness <----- output brightness
if(dot(result, BLOOM_THRESHOLD) > 1.0)
{
    out_brightness = vec4(result, 1.0);
}
else
{
    out_brightness = vec4(0.0, 0.0, 0.0, 1.0);
}

// output fragment <----- output color
out_fragment = vec4(result, 1.0);
}

++FRAGMENT++

```

The vertex shader remains unchanged. Directly above the fragment shader, an additional output variable, `out_brightness`, is introduced to represent the second attachment added to the frame buffer. To output to these attachments, the output is set accordingly. For capturing bright spots in the scene, a threshold is essential, denoted by the constant `BLOOM_THRESHOLD`. In the main function, the threshold is employed to compute a dot product with the computed color. If the result exceeds one, indicating a bright spot, the color is output to the brightness map.

Gaussian Blur Shader

We then have to create the shader to blur the brightness map so we can use it to get the final result. This is done in the code depicted in (Listing 6.88). The vertex shader is quite simple and does not require us to say much about it. All we need to render a texture is a quad. That is the reason why the buffer layout in the vertex shader is for the `Quad2D` we have been using in the final shader. Moving into the fragment shader, you will not see a couple of interesting things.

Listing 6.88: Resources/Shaders/bloom.gls

```
#version 330 core
layout(location = 0) in vec4 a_quad;
out vec2 uvs;

void main()
{
    gl_Position = vec4(a_quad.x, a_quad.y, 0.0, 1.0);
    uvs = vec2(a_quad.z, a_quad.w);
}

++VERTEX++

#version 330 core
out vec4 out_fragment;
in vec2 uvs;

const float WEIGHTS[5] = float[] (0.2270270270,
0.1945945946, 0.1216216216, 0.0540540541, 0.0162162162);

uniform sampler2D u_brightnessMap;
```

```

uniform bool u_horizontalPass;
uniform int u_frameHeight;
uniform int u_frameWidth;

void main()
{
    vec3 color = texture(u_brightnessMap, uvs).rgb *
WEIGHTS[0];

    // vec2 texelSize = 1.0 / textureSize(u_brightnessMap,
0); with opengl 4.x
    vec2 texelSize = 1.0 / vec2(u_frameWidth,
u_frameHeight);

    if(u_horizontalPass)
    {
        for(int i = 1; i < 5; ++i)
        {
            color += texture(u_brightnessMap, uvs +
vec2(texelSize.x * i, 0.0)).rgb * WEIGHTS[i];
            color += texture(u_brightnessMap, uvs -
vec2(texelSize.x * i, 0.0)).rgb * WEIGHTS[i];
        }
    }
    else
    {
        for(int i = 1; i < 5; ++i)
        {
            color += texture(u_brightnessMap, uvs + vec2(0.0,
texelSize.y * i)).rgb * WEIGHTS[i];
            color += texture(u_brightnessMap, uvs - vec2(0.0,
texelSize.y * i)).rgb * WEIGHTS[i];
        }
    }
    out_fragment = vec4(color, 1.0);
}

```

```
++FRAGMENT++
```

This shader is a separable Gaussian blur shader commonly used for image processing to achieve a blurring effect. It operates on a brightness map texture specified by the uniform variable `u_brightnessMap`. The blur can be applied either horizontally or vertically based on the boolean uniform variable `u_horizontalPass`. The size of the frame is determined by the uniform variables `u_frameWidth` and `u_frameHeight`. The shader utilizes a 5-tap Gaussian kernel defined by the array `WEIGHTS` to compute the weighted average of neighboring texels.

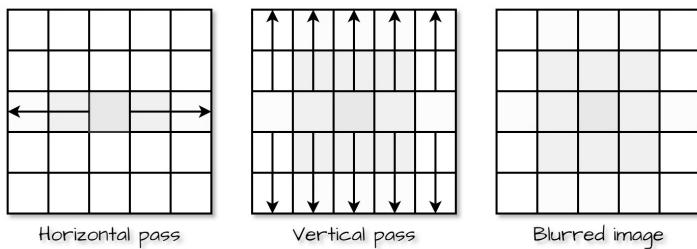


Figure 6.40: Gaussian Blur

The formula for a Gaussian function is given by:

$$G(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2}$$

Where μ is the mean (center) of the distribution and σ is the standard deviation, controlling the spread of the distribution.

The weights are pre-defined constants with values. The main function processes each texel of the input texture `u_brightnessMap` by convolving it with the Gaussian kernel. The convolution is performed in a loop, where the neighboring texels are sampled based on the direction of the blur (horizontal or vertical) and multiplied by the corresponding weights. The resulting color is accumulated and assigned to the output variable `out_fragment`. The final output is a blurred version of the input brightness map.

Gaussian Shader Abstraction

The following code depicted in (Listing 6.89) is the C++ abstraction of the bloom shader we created above. This shader uses two distinct frame buffers to perform both the horizontal and vertical passes of the Gaussian blur algorithm. You will note that we are also scaling down the size of the frame by a factor of "5". This is done to make the bloom spread outside of the shape of the object, giving the impression that the object is glowing.

Listing 6.89: Graphics/Shaders/Bloom.h

```
#pragma once
#include "Shader.h"
#include "../Utilities/Quad.h"

namespace EmPy
{
    struct BloomShader : Shader
    {
        EMPY_INLINE BloomShader(const std::string& path,
int32_t width, int32_t height) :
            Shader(path), m_Width(width), m_Height(height)
        {
            u_HorizontalPass =
glGetUniformLocation(m_ShaderID, "u_horizontalPass");
            u_BrightnessMap =
glGetUniformLocation(m_ShaderID, "u_brightnessMap");
            u_FrameHeight = glGetUniformLocation(m_ShaderID,
"u_frameHeight");
            u_FrameWidth = glGetUniformLocation(m_ShaderID,
"u_frameWidth");
            m_Quad = CreateQuad2D();

            // -----
            // scale down size
            m_Height = (height / m_Scale);
            m_Width = (width / m_Scale);

            // create frame buffer
            glGenFramebuffers(2, m_FrameBuffer);
        }
}
```

```

        // create horizontal texture
        glGenTextures(2, m_PingPongMaps);

        for(auto i = 0; i < 2; i++)
        {
            // bind target frame buffer
            glBindFramebuffer(GL_FRAMEBUFFER,
m_FrameBuffer[i]);

            // bind current texture
            glBindTexture(GL_TEXTURE_2D,
m_PingPongMaps[i]);
            glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA16F,
m_Width, m_Height, 0, GL_RGBA, GL_FLOAT,
NULL);

            // set texture parameters
            glTexParameteri(GL_TEXTURE_2D,
GL_TEXTURE_MAG_FILTER, GL_LINEAR);
            glTexParameteri(GL_TEXTURE_2D,
GL_TEXTURE_MIN_FILTER, GL_LINEAR);
            glTexParameteri(GL_TEXTURE_2D,
GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
            glTexParameteri(GL_TEXTURE_2D,
GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);

            // attach to frame buffer
            glFramebufferTexture2D(GL_FRAMEBUFFER,
GL_COLOR_ATTACHMENT0,
GL_TEXTURE_2D, m_PingPongMaps[i], 0);

            // check frame buffer
            if (glCheckFramebufferStatus(GL_FRAMEBUFFER)
!= GL_FRAMEBUFFER_COMPLETE)
            {
                EMPLY_ERROR("BloomShader() Failed!");
            }
        }
    }
}

```

```
        }

    }

    // unbind frame buffer
    glBindFramebuffer(GL_FRAMEBUFFER, 0);
}

EMPTY_INLINE void Compute(uint32_t brightnessMap,
uint32_t stepCount)
{
    // bind shader program
    glUseProgram(m_ShaderID);

    // set brightness map
    glActiveTexture(GL_TEXTURE0);
    glBindTexture(GL_TEXTURE_2D, brightnessMap);
    glUniform1i(u_BrightnessMap, 0);

    // set frame size
    glUniform1i(u_FrameHeight, m_Height);
    glUniform1i(u_FrameWidth, m_Width);

    // set viewport a clear buffer
    glViewport(0, 0, m_Width, m_Height);
    glClear(GL_COLOR_BUFFER_BIT);

    bool horizontal = true;

    for (uint32_t i = 0u; i < stepCount; i++)
    {
        glBindFramebuffer(GL_FRAMEBUFFER,
m_FrameBuffer[horizontal]);
        glUniform1i(u_HorizontalPass, horizontal);

        if (i > 0)
```

```

    {
        glActiveTexture(GL_TEXTURE0);
        glBindTexture(GL_TEXTURE_2D,
m_PingPongMaps[!horizontal]);
        glUniform1i(u_BrightnessMap,
0);
    }

    m_Quad->Draw(GL_TRIANGLES);
    horizontal = !horizontal;
}

glBindFramebuffer(GL_FRAMEBUFFER, 0);
glBindTexture(GL_TEXTURE_2D, 0);
glUseProgram(0);
}

EMPTY_INLINE void Resize(int32_t width, int32_t
height)
{
    m_Height = (height / m_Scale);
    m_Width = (width / m_Scale);

    for(auto i = 0; i < 2; i++)
    {
        glBindTexture(GL_TEXTURE_2D,
m_PingPongMaps[i]);
        glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA16F,
m_Width,
m_Height, 0, GL_RGBA, GL_FLOAT, NULL);
    }

    glBindTexture(GL_TEXTURE_2D, 0);
}

EMPTY_INLINE uint32_t GetMap()

```

```

    {
        return m_PingPongMaps[0];
    }

EMPTY_INLINE ~BloomShader()
{
    glDeleteFramebuffers(2, m_FrameBuffer);
    glDeleteTextures(2, m_PingPongMaps);
}

private:
    uint32_t u_HorizontalPass = 0u;
    uint32_t u_BrightnessMap = 0u;
    uint32_t u_FrameHeight = 0u;
    uint32_t u_FrameWidth = 0u;

    uint32_t m_PingPongMaps[2];
    uint32_t m_FrameBuffer[2];

    int32_t m_Height = 0;
    int32_t m_Width = 0;
    int32_t m_Scale = 5;

    Quad2D m_Quad;
};

}

```

The function `Compute()` takes the brightness map and uses it to generate a blurred version that will be used to render the finale scene. We also have the ability to specify how many steps we want the algorithm to take. More steps means a better result, but more computation time.

Updating the Final Shader

As already stated, the output of the bloom shader is meant to be used in the final shader to show the scene with bloom effect. this means we have to update the final shader as well. this is done in (Listing 6.90).

Listing 6.90: Resources/Shaders/final.gls

```
// vertex shader remains unchanged

++VERTEX++

#version 330 core
out vec4 out_fragment;
in vec2 uvs;

const float GAMMA = 2.5;
const float EXPOSURE = 10.0;
const float MIN_GAMMA = 0.000001;

uniform sampler2D u_map;
uniform sampler2D u_bloom;

void main()
{
    // combine the actual color and the bloom color
    vec3 result = texture(u_map, uvs).rgb + texture(u_bloom,
uvs).rgb;

    // sample color from map
    result = pow(result, vec3(GAMMA));
```

```

// process exposure
result = vec3(1.0) - exp(-result * EXPOSURE);

// gamma correction
result = pow(result, vec3(1.0 / max(GAMMA, MIN_GAMMA))) ;

// fragment color
out_fragment = vec4(result, 1.0);
}

++FRAGMENT++

```

The only changes made here are: The new uniform `u_bloom` which represents the blurred version of the brightness map and the result which is now computed by adding the color sampled from both the color map and the bloom map.

Final Shader Abstraction

We also need to quickly add these changes to the final shader abstraction to make sure that both the C++ version and GLSL version are speaking the same language. See ([Listing 6.91](#))

Listing 6.91: Graphics/Shaders/Final.
h

```

#pragma once
#include "Shader.h"
#include "../Utilities/Quad.h"

```

```

namespace Empy
{
    struct FinalShader : Shader
    {
        EMPY_INLINE FinalShader(const std::string& filename) :
        Shader(filename)
        {
            u_Bloom = glGetUniformLocation(m_ShaderID,
"u_bloom");
            u_Map = glGetUniformLocation(m_ShaderID,
"u_map");
            m_Quad = CreateQuad2D();
        }

        EMPY_INLINE void Show(uint32_t map, uint32_t bloom)
        {
            glBindFramebuffer(GL_FRAMEBUFFER, 0);
            glClear(GL_COLOR_BUFFER_BIT);
            glClearColor(0, 0, 0, 1);
            glUseProgram(m_ShaderID);

            // set color map
            glActiveTexture(GL_TEXTURE0);
            glBindTexture(GL_TEXTURE_2D, map);
            glUniform1i(u_Map, 0);

            // set bloom map
            glActiveTexture(GL_TEXTURE1);
            glBindTexture(GL_TEXTURE_2D, bloom);
            glUniform1i(u_Bloom, 1);

            // render quad
            m_Quad->Draw(GL_TRIANGLES);
            glUseProgram(0);
        }
}

```

```

    private:
        uint32_t u_Bloom = 0u;
        uint32_t u_Map = 0u;
        Quad2D m_Quad;
    };
}

```

The change here lies in the newly added argument to the `Show()` function. It takes the blurred version of the brightness map and binds using the texture unit "1".

Updating Renderer Class

With all these components in place, we need to put them together in the renderer class to make produce the desire result. Following are the minor changes you need to apply to get it to run.

Listing 6.92: Graphics/Renderer.h

```

struct GraphicsRenderer
{
    EMPLY_INLINE GraphicsRenderer(int32_t width, int32_t
height)
    {
        /* ... same as before ... */

        m_Bloom = std::make_unique<BloomShader>
("Resources/Shaders/bloom.glsl", width, height);
    }
}

```

```

EMPTY_INLINE void ShowFrame()
{
    glViewport(0, 0, m_Frame->Width(), m_Frame-
>Height());
    m_Final->Show(m_Frame->GetTexture(), m_Bloom-
>GetMap());
}

EMPTY_INLINE void EndFrame()
{
    m_Pbr->Unbind();
    m_Frame->End();

    // post-processing
    m_Bloom->Compute(m_Frame->GetBrightnessMap(), 10);
}

private:
    std::unique_ptr<BloomShader> m_Bloom;
    /* ... same as before ... */
};

```

Starting with the constructor, all we need to do is to create the shader by providing the frame size. Due to the fact that the bloom shader uses a scaled down viewport to render the bloom map, we need to reset the viewport back to its original state when showing the scene to the screen. This is done in the `ShowFrame()`. In the `EndFrame()` function, you will note how the bloom map is computed after all the rendering is done. This is why it is called post-processing.

Testing the Bloom Effect

You can add an entities in your application context as depicted below to get the result in (Figure [6.41](#)).

Listing 6.93: Application/Application.h

```
// load textures
auto skymap = std::make_shared<Texture2D>
("Resources/Textures/HDRs/Sky.hdr", true, true);

// load models
auto sphereModel = std::make_shared<Model>
("Resources/Models/sphere.fbx");
auto cubeModel = std::make_shared<Model>
("Resources/Models/cube.fbx");

// create scene camera
auto camera = CreateEntt<Entity>();
camera.Attach<TransformComponent>().Transform.Translate.z =
3.0f;
camera.Attach<CameraComponent>();

// skybox entity
auto skybox = CreateEntt<Entity>();
skybox.Attach<TransformComponent>();
skybox.Attach<SkyboxComponent>();

// create point light 1
auto slight = CreateEntt<Entity>();
slight.Attach<DirectLightComponent>().Light.Intensity = 5.0f;
auto& stp = slight.Attach<TransformComponent>().Transform;
stp.Rotation = glm::vec3(0.0f, 0.0f, -1.0f);
```

```
// create sphere entity
auto sphere = CreateEntt<Entity>();
auto& mod = sphere.Attach<ModelComponent>();
mod.Model = sphereModel;
mod.Material.Emissive = glm::vec3(1.0f);
mod.Material.Albedo = glm::vec3(0.8f, 0.1f, 0.8f);
sphere.Attach<TransformComponent>().Transform.Translate.x =
-1.0f;

// create cube entity
auto cube = CreateEntt<Entity>();
auto& mod1 = cube.Attach<ModelComponent>();
mod1.Model = cubeModel;
mod1.Material.Albedo = glm::vec3(0.1f, 0.0f, 0.5f);
cube.Attach<TransformComponent>().Transform.Translate.x =
1.0f;
```



Figure 6.41: Bloom Effect Result

Additional post-processing effects could enhance this renderer, but we won't delve into those details in this book, as our focus lies elsewhere. We have other

topics, such as animations, physics, scripting, and more, to explore. Let us transition to a different subject. I hope you found this journey enjoyable, and as always, if you encountered any challenges in implementing specific sections of this chapter, refer to the notes below for guidance. Let us now tackle skeletal animations.

Code not working? clone the branch "advanced-rendering" on the GitHub repository and grant execution rights to the "EmPyLinux.sh" script on Linux. <https://github.com/Madsycode/book-empy-engine>.

```
git clone -branch <branch-name> <repository-link>
```

7 Skeletal Animation

In all toil there is profit, but mere talk tends only to poverty (King Solomon)

Skeletal animation, also known as rigging, is a technique widely used in computer graphics and animation to bring life and movement to characters. It involves the use of a hierarchical structure of interconnected bones or joints, forming a skeleton, which is then associated with a 3D model.

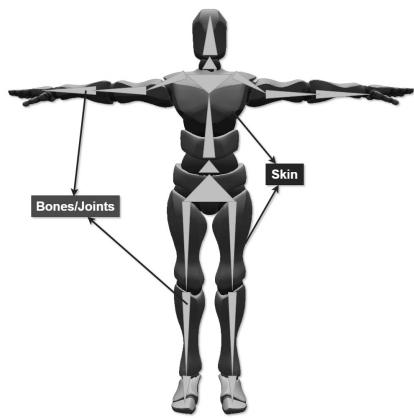


Figure 7.1: Skeletal Animation Joints
[Mixamo](#)

Each bone represents a segment of the model, and their arrangement simulates the underlying structure of a character.

7.1 Important Concepts

The primary purpose of skeletal animation is to create realistic and dynamic movements for characters in animations or video games. Instead of animating each individual vertex of a 3D model, which can be computationally expensive and time-consuming, skeletal animation works by manipulating the bones of the skeleton. The movement of the bones is then translated to the attached vertices, deforming the character model accordingly.

Key components of skeletal animation include:

- **Bones/Joints:** The structural elements of the skeleton, representing different parts of the character. Each bone has a local transformation, and they are organized hierarchically.
See ([Figure 7.2](#))

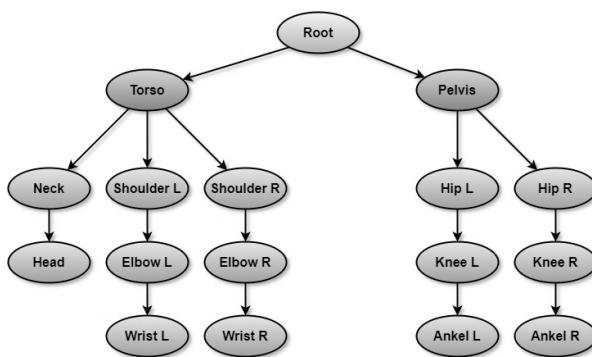


Figure 7.2: Joints Hierarchy

- **Skin/Vertices:** The points that make up the 3D model. These vertices are associated with specific bones through a process known as vertex weighting, which determines how much influence each bone has on a particular vertex.
- **Animations:** Skeletal animation involves creating keyframes and defining how the bones move over time. Interpolation between keyframes produces smooth and realistic motion.
- **Inverse Kinematics:** In some cases, inverse kinematics is used to control the end effector of a bone (like a hand or foot) directly, allowing for more natural and intuitive animation.

Skeletal animation offers several advantages, including efficient storage of animations, the ability to reuse animations on different characters with similar skeletons, and a more intuitive workflow for animators. It is widely employed in industries such as gaming, film, and virtual reality to create lifelike character movements.

7.1.1 Animation Keyframe

A keyframe in animation serves as a significant marker within the timeline, representing a specific point in time where key changes or poses occur. It is essentially a frame that contains essential information

about the state of an object or character at that particular moment. The keyframe is crucial in defining the overall motion and appearance of the animated element. Here are the components typically found in a keyframe:

- **Position:** The position component of a keyframe specifies the location of an object or a character in the 3D space at that specific moment. In skeletal animation, this often refers to the 3D coordinates (x, y, z) of a bone or joint.
- **Rotation:** Rotation information in a keyframe defines the orientation or angular positioning of an object or character. It specifies how much the object or character is rotated around each axis (roll, pitch, yaw). In skeletal animation, this is often represented using quaternions to avoid gimbal lock issues.
- **Scale:** The scale component determines the size or scaling factor of an object or character at the keyframe. It defines how much the object or character is stretched or compressed along each axis (x, y, z). Scale is important for controlling the size variation of elements over time.
- **Timestamp:** Each keyframe is associated with a specific point in time, represented by a timestamp or frame number. This information indicates when the keyframe occurs within the overall animation.

timeline. The timing of keyframes is crucial for controlling the pacing and flow of the animation.

- **Interpolation Information:** Interpolation information specifies how the animated element transitions from one keyframe to the next. It includes details about the type of interpolation used for position, rotation, and scale.
- **Additional Parameters:** Depending on the animation system, keyframes may include additional parameters specific to the animation requirements. For example, in complex systems, keyframes might include information related to deformations, constraints, or specific attributes of the animated elements.

By defining keyframes strategically along the timeline, we can create a sequence of frames that, when played in succession, result in a smooth and coherent animated motion.

7.1.2 Keyframe Interpolation

Keyframe interpolation in skeletal animation involves generating intermediate frames between two keyframes to achieve smooth and natural motion. The process ensures that the transition from one pose to another is visually pleasing and realistic. Here is an

explanation of how keyframe interpolation is typically done:

- **keyframes:** Animators define keyframes to represent significant poses or positions in the animation timeline. Each keyframe contains information about the position, rotation, and scale of each bone in the skeleton at that particular moment.
- **Interpolation Types:** Animators choose an interpolation type for each parameter (position, rotation, scale) between keyframes. The two common types are linear interpolation and spline interpolation.
- **Linear Interpolation:** Linear interpolation is the simplest form of interpolation. It creates a straight line between two keyframes. For example, for position interpolation, the new position at any given frame is calculated based on a linear progression between the positions of the two keyframes.

$$P(t) = (1 - t) \cdot P_{\text{start}} + t \cdot P_{\text{end}}$$

where: - $P(t)$ is the interpolated position at time t . - P_{start} is the position at the start keyframe. - P_{end} is the position at the end keyframe. - t varies from 0 to 1, representing the interpolation progress.

- **Spline Interpolation:** Spline interpolation, such as cubic splines, provides smoother transitions between keyframes by using mathematical curves. Instead of a straight line, the interpolation function is a curve that passes through both keyframes, resulting in more organic motion.

$$P(t) = (2t^3 - 3t^2 + 1) \cdot P_{\text{start}} + (t^3 - 2t^2 + t) \cdot T_{\text{start}} + \\ (-2t^3 + 3t^2) \cdot P_{\text{end}} + (t^3 - t^2) \cdot T_{\text{end}}$$

where: - $P(t)$ is the interpolated position at time t . - P_{start} and P_{end} are the positions at the start and end keyframes. - T_{start} and T_{end} are tangent vectors representing the direction of the curve at the start and end keyframes.

- **Quaternion Slerp (Spherical Linear Interpolation):** For rotational interpolation, quaternion slerp is often used to ensure smooth rotation between keyframes while avoiding gimbal lock issues.

$$Q(t) = \text{slerp}(Q_{\text{start}}, Q_{\text{end}}, t)$$

where: - $Q(t)$ is the interpolated quaternion at time t . - Q_{start} and Q_{end} are the quaternions representing rotations at the start and end keyframes. - t varies from 0 to 1.

By applying these interpolation techniques to each parameter (position, rotation, scale) of each bone in the skeleton, a smooth and continuous animation is achieved between keyframes. The choice between linear and spline interpolation depends on the desired style of motion and the artistic goals of the animation. We will be using linear interpolation for positions and scales and the quaternion slerp for the rotations.

7.1.3 Joint Transformation

In skeletal animation, transforming joints from local space to global space involves applying a hierarchical transformation that considers the parent-child relationships between joints. Each joint has its own local transformation, representing its position, orientation, and scale relative to its parent joint. The goal is to compute the global transformation for each joint, representing its position in the overall skeleton.

The typical process for transforming joints from local to global space involves the following steps:

- **Traversal of the Skeleton Hierarchy:** Starting from the root joint, we traverse the skeleton hierarchy in a depth-first manner. This ensures that transformations are propagated from parent-to-child joints.

- **Local Transformations:** At each joint, we calculate its local transformation matrix. This matrix incorporates the joint's local translation, rotation, and scaling information. It represents the joint's transformation relative to its parent.
- **Global Transformations:** We then accumulate the local transformations as we traverse up the hierarchy, from child to parent joints. This accumulation results in a global transformation for each joint, representing its position and orientation in the entire skeleton.
- **Transformation Composition:** Lastly, the global transformation matrix for each joint is computed by multiplying its local transformation matrix with the accumulated global transformation matrix of its parent. This composition allows for the correct propagation of transformations through the hierarchy.

The transformation process can be expressed as follows:

$$GT_{\text{joint}} = PGT_{\text{joint}} \times LT_{\text{joint}}$$

Where: GT_{joint} is the global transformation matrix of the current joint. PGT_{joint} is the global transformation matrix of its parent. LT_{joint} is the local transformation matrix of the current joint. This ensures that each joint's transformation is

correctly represented in the global coordinate system, accounting for its position relative to both its parent and the overall skeleton.

7.2 Implementing Animation

There is a lot we can say about skeletal animation, but I don't think that is the reason you are reading this book. Let us move forward and write some code. The first thing we have to do is update the directory tree of our project, as depicted below in (Figure 7.3).

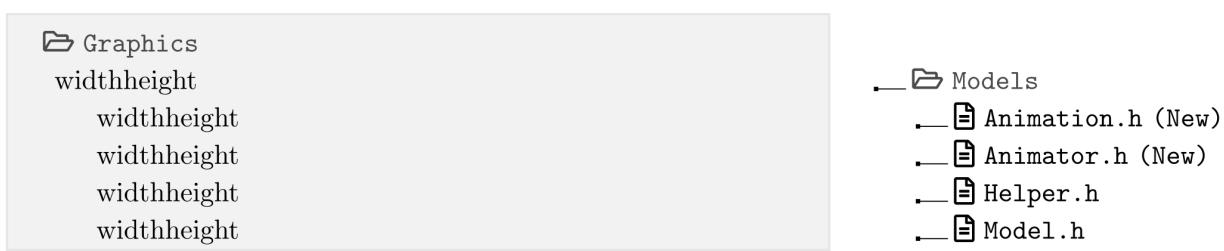


Figure 7.3: Models directory

7.2.1 Animation Data

We need to encapsulate all important information about animations in a way that will make it simple for us to use them in the animator. Let us start with the animation, the keyframe, and the joints. The code depicted in (Listing 7.1) has three important structures we need to address. The first

structure, `Animation`, encapsulates information related to an animation. It includes parameters such as the duration in seconds, a speed factor to control the playback speed, and a name for easy reference.

Listing 7.1: Graphics/Models/Animation. h

```
#pragma once
#include "Helper.h"

namespace EmPy
{
    struct Animation
    {
        float Duration = 0.0f;
        float Speed = 1.0f;
        std::string Name;
    };

    struct KeyFrame
    {
        glm::vec3 Position = glm::vec3(0.0f);
        glm::quat Rotation = glm::vec3(0.0f);
        glm::vec3 Scale = glm::vec3(1.0f);
        float TimeStamp = 0.0f;
    };

    struct Joint
    {
        std::vector<Joint> Children;
        std::vector<KeyFrame> Keys;
        std::string Name;
        glm::mat4 Offset; // <- inverse transform mtx
```

```
    int32_t Index;  
};  
}
```

The second structure, `KeyFrame` It represents a specific moment in time and contains transformation data for a joint or bone. This includes the 3D position, represented as a vector, the rotation stored as a quaternion, the scaling information, and a timestamp indicating when the keyframe occurs during the animation. The third structure, `Joint`, represents a joint in the skeletal hierarchy. A joint has child joints, a collection of keyframes defining its animation over time, a name identifier, and a transformation offset represented by a 4x4 matrix. The offset matrix is used to transform the joint from local space global space (more on that in a bit). An index for referencing its position in the overall skeletal structure.

7.2.2 Skeletal Mesh

Before we can attempt to load animation from model files, we must first create a new vertex type along with a mesh type tailored for objects that involve animations.

Skeletal Vertex

In the following code snippet, you can observe that the skeletal vertex closely resembles the shaded vertex but

incorporates additional attributes to accommodate animation. The initial property is a `Vector4 Joints` that serves as a storage for joint identifiers influencing the vertex.

Listing 7.2: Graphics/Buffers/Vertex. h

```
namespace EmPy
{
    struct SkeletalVertex
    {
        glm::vec3 Position = glm::vec3(0.0f);
        glm::vec3 Normal = glm::vec3(0.0f);
        glm::vec2 UVs = glm::vec2(0.0f);

        // for lighting
        glm::vec3 Tangent = glm::vec3(0.0f);
        glm::vec3 Bitangent = glm::vec3(0.0f);

        // for animation
        glm::ivec4 Joints = glm::ivec4(-1);
        glm::vec4 Weights = glm::vec4(0.0f);
    };

    // ...
}
```

The second attribute `Weights` quantifies how much the joints in the first attribute influence this vertex. Higher values signify a more substantial impact, while smaller values indicate a more subtle effect. This behavior mirrors the real-world scenario where, for instance, turning your arm affects certain parts of your body more than others.

Skeletal Mesh

We then have to use this skeletal vertex in the mesh class to make sure it is used when creating a skeletal meshes. The following code snippet depicts how this is done.

Listing 7.3: Graphics/Buffers/Vertex.h

```
template <typename Vertex> struct Mesh
{
    EMPLY_INLINE Mesh(const MeshData<Vertex>& data)
    {
        /* ... same as before ... */
        else if (TypeID<Vertex>() == TypeID<SkeletalVertex>())
        {
            SetAttribute(0, 3,
(void*)offsetof(SkeletalVertex, Position));
            SetAttribute(1, 3,
(void*)offsetof(SkeletalVertex, Normal));
            SetAttribute(2, 2,
(void*)offsetof(SkeletalVertex, UVs));
            SetAttribute(3, 3,
(void*)offsetof(SkeletalVertex, Tangent));
            SetAttribute(4, 3,
(void*)offsetof(SkeletalVertex, Bitangent));
            SetAttribute(5, 4,
(void*)offsetof(SkeletalVertex, Joints));
            SetAttribute(6, 4,
(void*)offsetof(SkeletalVertex, Weights));
        }
        /* ... same as before ... */
    }
}
```

```

        // unbind vertex array
        glBindVertexArray(0);
    }

private:
    uint32_t m_NbrVertex = 0u;
    uint32_t m_NbrIndex = 0u;
    uint32_t m_BufferID = 0u;
}

```

7.2.3 Model Animator

Because the animator is required inside the model, we want to implement it first to keep things simple. The following code shows the implementation of the model animator.

Listing 7.4: Graphics/Models/Animator.h

```

#pragma once
#include "Animation.h"

namespace EmPy
{
    struct Animator
    {
        EMPY_INLINE auto& Animate(float deltaTime)
        {
            if (m_Sequence < m_Animations.size())
            {
                m_Time += m_Animations[m_Sequence].Speed *
deltaTime;
                m_Time = fmod(m_Time,
m_Animations[m_Sequence].Duration);
            }
        }
    };
}

```

```

        UpdateJoints(m_Root, glm::identity<glm::mat4>
() );
    }

    return m_Transforms;
}

private:
    EMPLY_INLINE glm::mat4 Interpolate(const KeyFrame&
prev, const KeyFrame& next, float progression)
{
    return (glm::translate(glm::mat4(1.0f),
glm::mix(prev.Position, next.Position, progression)) *

        glm::toMat4(glm::normalize(glm::slerp(prev.Rotation
, next.Rotation, progression))) *
        glm::scale(glm::mat4(1.0f), glm::mix(prev.Scale,
next.Scale, progression)));
}

    EMPLY_INLINE void GetPreviousAndNextFrames(Joint&
joint, KeyFrame& prev, KeyFrame& next)
{
    for (uint32_t i = 1u; i < joint.Keys.size();
i++)
    {
        if (m_Time < joint.Keys[i].TimeStamp)
        {
            prev = joint.Keys[i - 1];
            next = joint.Keys[i];
            return;
        }
    }
}

    EMPLY_INLINE void UpdateJoints(Joint& joint, const
glm::mat4& parentTransform)

```

```

    {

        KeyFrame prev, next;

        // get previous and next frames
        GetPreviousAndNextFrames(joint, prev, next);

        // compute interpolation factor
        float progression = (m_Time - prev.TimeStamp)
/ (next.TimeStamp - prev.TimeStamp);

        // compute joint new transform
        glm::mat4 transform = parentTransform *
Interpolate(prev, next, progression);

        // update joint transform
        m_Transforms[joint.Index] = (transform *
m_GlobalTransform * joint.Offset);

        // update children joints
        for (auto& child : joint.Children) {
UpdateJoints(child, transform); }

    }

private:
    std::vector<Animation> m_Animations;
    std::vector<glm::mat4> m_Transforms;
    glm::mat4 m_GlobalTransform;
    friend struct SkeletalModel; //<-- friend class
    int32_t m_Sequence = 0;
    float m_Time = 0.0f;
    Joint m_Root;
};

}

```

Contrary to what one would expect, there isn't really much going on in the animator class. The function named `Animate()` takes a `deltaTime` parameter, representing the time elapsed since the last frame. Within the function, there is a conditional check to ensure that the current animation sequence index `m_Sequence` is within the valid range of available animations. If so, the animation time `m_Time` is updated based on the animation's speed and the elapsed time. We then ensures that the animation time is within the duration of the current animation sequence by using the modulus operator. Finally, We call the `UpdateJoints()` with the root joint and an identity matrix, to update the transformations of all joints in the skeletal structure. The function returns a reference to the transform array, which will be sent to the PBR by the renderer.

The private section includes additional helper functions. The `Interpolate()` function computes an interpolated transformation matrix between two keyframes, considering translation, rotation, and scaling. The `GetPreviousAndNextFrames()` function determines the keyframes preceding and succeeding the current animation time for a given joint. The `UpdateJoints()` function recursively updates the transformations of joints based on the current animation time, parent transformations, and interpolation factors. The `m_GlobalTransform` is a matrix that helps us transform the joint from local meaning relative to parent to global meaning relative to the root joint. This joint

hierarchy was depicted in (Figure 7.2). Note how the skeletal model is a friend to the animator class. This ensures that the model is able to access private variables and functions.

7.2.4 Loading Animation

To incorporate animations into our system, we must extend our model loading process to handle animation data. Fortunately, the Assimp library, employed for loading models, is versatile and can also handle animation information.

Static Model

We are going to change the structure of the model class in a way that both models with and without animation inherit from the same base model class. This will keep us from having to change our rendering pipeline to incorporate skeletal models.

Listing 7.5: Graphics/Models/Model.h

```
#pragma once
#include <assimp/postprocess.h>
#include <assimp/quaternion.h>
#include <assimp/Importer.hpp>
#include <assimp/scene.h>
#include "Animator.h"

namespace Empr
```

```

{
    // abstract model
    struct Model
    {
        EMPTY_INLINE virtual bool HasJoint() { return false; }

        EMPTY_INLINE virtual void Load(const std::string&) {}
        EMPTY_INLINE virtual void Draw(uint32_t mode) {}

    };

    // static model
    struct StaticModel : Model
    {
        EMPTY_INLINE StaticModel() = default;

        EMPTY_INLINE StaticModel(const std::string& path)
        {
            Load(path);
        }

        EMPTY_INLINE void Load(const std::string& path)
override final
        {
            uint32_t flags = aiProcess_Triangulate |
aiProcess_GenSmoothNormals | aiProcess_CalcTangentSpace |
                aiProcess_OptimizeMeshes |
aiProcess_OptimizeGraph | aiProcess_ValidateDataStructure |
                    aiProcess_ImproveCacheLocality |
aiProcess_FixInfacingNormals |
                aiProcess_GenUVCoords | aiProcess_FlipUVs;

            Assimp::Importer importer;
            const aiScene* ai_scene = importer.ReadFile(path,
flags);

            if (!ai_scene || ai_scene->mFlags ==

```

```

AI_SCENE_FLAGS_INCOMPLETE || !ai_scene->mRootNode)
{
    EMPTY_ERROR("failed to load model-{}", importer.GetErrorString());
    return;
}

// parse all meshes
ParseNode(ai_scene, ai_scene->mRootNode);
}

EMPTY_INLINE void Draw(uint32_t mode) override final
{
    for(auto& mesh : m_Meshes)
    {
        mesh->Draw(mode);
    }
}

private:
/* ... same as before ... */

private:
std::vector<std::unique_ptr<ShadedMesh>> m_Meshes;
};

}

```

The code of the model depicted above in (Listing 7.5) is the same code we have been using so far, with the only difference that the name has changed to `StaticModel` and that it now inherits from the newly added abstract `Model` structure which has some virtual functions that can be overridden in a derived classes. This should be simple to understand.

Skeletal Model

We then want to do the same thing, but this time for skeletal models. The following code snippet (Listing 7.6) shows the implementation of the skeletal model. This code is written straight after the static model defined above. There is a bit more going on in this code. Let us explain what role each function plays in the loading process. Starting with the `Load()` function, you will note how similar it is to that found in the static model class, with the addition of the animator being created. We are getting the global transform matrix from the `Root` joint. We have already stated how this matrix is important to transform joint transformations from the local space to the global space. The `ParseNode()` function now has an additional parameter to temporarily hold the joints for further parsing.

Listing 7.6: Graphics/Models/Model.h

```
/* ... same as before ... */

// struct StaticModel : Model {...}

// skeletal model
struct SkeletalModel : Model
{
    using JointMap = std::unordered_map<std::string, Joint>

    EMPTY_INLINE SkeletalModel(const std::string& path)
```

```

    {

        Load(path);

    }

EMPTY_INLINE SkeletalModel() = default;

EMPTY_INLINE bool HasJoint() override final { return
m_JointCount; }

EMPTY_INLINE void Load(const std::string& path) override
final
{
    uint32_t flags = aiProcess_Triangulate |
aiProcess_GenSmoothNormals | aiProcess_CalcTangentSpace |
        aiProcess_OptimizeMeshes |
aiProcess_OptimizeGraph | aiProcess_ValidateDataStructure |
            aiProcess_ImproveCacheLocality |
aiProcess_FixInfacingNormals |
                aiProcess_SortByPType |
aiProcess_JoinIdenticalVertices |
                    aiProcess_FlipUVs | aiProcess_GenUVCoords |
            aiProcess_LimitBoneWeights;

    Assimp::Importer importer;
    const aiScene* ai_scene = importer.ReadFile(path,
flags);
    if (!ai_scene || ai_scene->mFlags ==
AI_SCENE_FLAGS_INCOMPLETE || !ai_scene->mRootNode)
    {
        EMPTY_ERROR("failed to load model: '{}',",
importer.GetErrorString());
        return;
    }

    m_Animator = std::make_shared();
    m_Animator->m_GlobalTransform =

```

```

glm::inverse(AssimpToMat4(ai_scene->mRootNode-
>mTransformation));

        // temp joints map
        JointMap jointMap = { };

        // parse all meshes
        ParseNode(ai_scene, ai_scene->mRootNode, jointMap);

        // parse animations
        ParseAnimations(ai_scene, jointMap);
    }

EMPTY_INLINE void Draw(uint32_t mode) override final
{
    for(auto& mesh : m_Meshes)
    {
        mesh->Draw(mode);
    }
}

EMPTY_INLINE auto GetAnimator()
{
    return m_Animator;
}

private:
    EMPTY_INLINE void ParseNode(const aiScene* ai_scene,
aiNode* ai_node, JointMap& jointMap)
    {
        for (uint32_t i = 0; i < ai_node->mNumMeshes; i++)
        {
            ParseMesh(ai_scene->mMeshes[ai_node->mMeshes[i]], jointMap);
        }
    }
}

```

```

        for (uint32_t i = 0; i < ai_node->mNumChildren; i++)
    {
        ParseNode(ai_scene, ai_node->mChildren[i],
jointMap);
    }
}

EMPTY_INLINE void SetVertexJoint(SkeletalVertex& vertex,
int32_t id, float weight)
{
    for (uint32_t i = 0; i < 4; i++)
    {
        if (vertex.Joints[i] < 0)
        {
            vertex.Weights[i] = weight;
            vertex.Joints[i] = id;
            return;
        }
    }
}

EMPTY_INLINE void ParseHierarchy(aiNode* ai_node, Joint&
joint, JointMap& jointMap)
{
    std::string jointName(ai_node->mName.C_Str());

    if(jointMap.count(jointName))
    {
        joint = jointMap[jointName];

        for (uint32_t i = 0; i < ai_node->mNumChildren;
i++)
        {
            Joint child;
            ParseHierarchy(ai_node->mChildren[i], child,
jointMap);
        }
    }
}

```

```

                joint.Children.push_back(std::move(child));
            }
        }
        else
        {
            for (uint32_t i = 0; i < ai_node->mNumChildren;
i++)
            {
                ParseHierarchy(ai_node->mChildren[i], joint,
jointMap);
            }
        }
    }

EMPTY_INLINE void ParseAnimations(const aiScene* ai_scene,
JointMap& jointMap)
{
    // parse animation data
    for (uint32_t i = 0; i < ai_scene->mNumAnimations;
i++)
    {
        auto ai_anim = ai_scene->mAnimations[i];

        Animation animation;
        animation.Name = ai_anim->mName.C_Str();
        animation.Duration = ai_anim->mDuration;
        animation.Speed = ai_anim->mTicksPerSecond;
        m_Animator->m_Animations.push_back(animation);

        // parse animation keys
        for (uint32_t j = 0; j < ai_anim->mNumChannels;
j++)
        {
            aiNodeAnim* ai_channel = ai_anim-
>mChannels[j];
            auto jointName(ai_channel->mNodeName.C_Str());

```

```

        if (!jointMap.count(jointName)) { continue; }

        for (uint32_t k = 0; k < ai_channel-
>mNumPositionKeys; k++)
        {
            KeyFrame key;
            key.Position = AssimpToVec3(ai_channel-
>mPositionKeys[k].mValue);
            key.Rotation = AssimpToQuat(ai_channel-
>mRotationKeys[k].mValue);
            key.Scale = AssimpToVec3(ai_channel-
>mScalingKeys[k].mValue);
            key.TimeStamp = ai_channel-
>mPositionKeys[k].mTime;
            jointMap[jointName].Keys.push_back(key);
        }
    }

    // parse jointMap hierarchy
    ParseHierarchy(ai_scene->mRootNode, m_Animator-
>m_Root, jointMap);

    // initialize animator
    m_Animator->m_Transforms.resize(m_JointCount);
}

EMPTY_INLINE void ParseMesh(const aiMesh* ai_mesh,
JointMap& jointMap)
{
    // mesh data
    MeshData<SkeletalVertex> data;

    // vertices
    for (uint32_t i = 0; i < ai_mesh->mNumVertices; i++)
    {

```

```

        SkeletalVertex vertex;
        // positions
        vertex.Position = AssimpToVec3(ai_mesh-
>mVertices[i]);
        // normals
        vertex.Normal = AssimpToVec3(ai_mesh-
>mNormals[i]);
        // texcoords
        vertex.UVs.x = ai_mesh->mTextureCoords[0][i].x;
        vertex.UVs.y = ai_mesh->mTextureCoords[0][i].y;
        // push vertex
        data.Vertices.push_back(std::move(vertex));
    }

    // indices
    for (uint32_t i = 0; i < ai_mesh->mNumFaces; i++)
    {
        for (uint32_t k = 0; k < ai_mesh-
>mFaces[i].mNumIndices; k++)
        {
            data.Indices.push_back(ai_mesh-
>mFaces[i].mIndices[k]);
        }
    }

    // joints
    for (uint32_t i = 0; i < ai_mesh->mNumBones; i++)
    {
        aiBone* ai_bone = ai_mesh->mBones[i];

        // get joint name
        std::string jointName(ai_bone->mName.C_Str());

        // add joint if not found
        if(jointMap.count(jointName) == 0)
        {

```

```

                jointMap[jointName].Offset =
AssimpToMat4(ai_bone->mOffsetMatrix);
                jointMap[jointName].Index = m_JointCount++;
                jointMap[jointName].Name = jointName;
            }

            // set vertex joint weights
            for (uint32_t j = 0; j < ai_bone->mNumWeights;
j++)
{
            SetVertexJoint(data.Vertices[ai_bone-
>mWeights[j].mVertexId],
                jointMap[jointName].Index, ai_bone-
>mWeights[j].mWeight);
        }
    }

    // create new mesh instance
    m_Meshes.push_back(std::make_unique<SkeletalMesh>
(data));
}

private:
    std::vector<std::unique_ptr<SkeletalMesh>> m_Meshes;
    std::shared_ptr<Animator> m_Animator;
    uint32_t m_JointCount = 0;
};

// helper type definitions
using Animator3D = std::shared_ptr<Animator>;
using Model3D = std::shared_ptr<Model>;

```

The `ParseHierarchy()` function is responsible for parsing a hierarchy of joints from an Assimp node. It begins by extracting the name of the current Assimp node and checks

if this name already exists in the `jointMap`. If the joint name is present, indicating that a corresponding joint structure has been previously created, the code retrieves and assigns this joint structure to the current joint. For each child of the current Assimp node, the function recursively calls itself, parsing the hierarchy and constructing the joint structure for each child. If the joint name is not found in the `jointMap`, the recursion continues without modifying the current joint.

The `ParseAnimations` function is crucial for extracting animation properties and keyframes associated with each joint in the model. The initial loop iterates through the animations present in the Assimp scene, constructing `Animation` structures for each animation by extracting relevant information such as the name, duration, and ticks per second. These animations are then added to the `m_Animations` vector within the associated `Animator` structure. Within the animation loop, there is a nested loop responsible for iterating through animation channels `aiNodeAnim` instances associated with each joint.

For every channel, the function retrieves the joint's name and checks its existence in the `jointMap`. If the joint is present, the function proceeds to parse keyframes for position, rotation, and scaling. Each keyframe is constructed from the corresponding Assimp data and appended to the `keys` vector of the respective joint in the `jointMap`. This process ensures that the animation data, including

keyframes, is correctly associated with the corresponding joints. Following the animation parsing loop, the function calls the `ParseHierarchy()` to parse the hierarchy as explained previously.

In the `ParseMesh()` function, the most important part is found in the loop to parse the joints. This loop iterates through the bones present in the Assimp mesh, extracting information such as the joint name, offset matrix, and weights associated with each bone. Let's break down its functionality:

The loop begins by iterating through the bones of the Assimp mesh using a 'for' loop. For each bone `aIBone`, we extract the bone/joint's name `jointName` and check whether it is already present in the `jointMap`. If the joint is not found, indicating that it hasn't been encountered before, a new entry is created in the map. The information extracted includes the index and name of the joint as well as its offset matrix, also known as the "Bind Pose Matrix" or the "Inverse Bind Matrix," which represents the initial transformation of a joint in its "bind pose" or "rest pose," which is the pose the joint has in the original, undeformed state of the model.

Following the joint initialization, we proceed to set the vertex joint weights. It iterates through the weights associated with the current bone, obtaining the vertex ID and weight for each. The `SetVertexJoint()` function is then called, which

associates the joint index and weight with the corresponding vertex in the mesh data. Finally, we have some type definitions to simplify our interaction with the `Animator` class. The `Model3D` type definition has been around since the first time the model class was created. With this, both classes can be rendered using the same `Draw()` function in the PBR shader.

Updating the Shader

We now have the animation loading mechanism in place, but we still can't use it. Update your PBR vertex shader as depicted in the following code.

Listing 7.7: Resources/Shaders/pbr.gls

```
#version 330 core
layout (location = 0) in vec3 a_position;
layout (location = 1) in vec3 a_normal;
layout (location = 2) in vec2 a_uvs;
layout (location = 3) in vec3 a_tangent;
layout (location = 4) in vec3 a_bitangent;
layout (location = 5) in ivec4 a_joints;
layout (location = 6) in vec4 a_weights;

#define MAX_WEIGHTS 4
#define MAX_JOINTS 100

out Vertex
{
```

```
vec3 Position;
vec3 Normal;
mat3 TBN;
vec2 UVs;
} vertex;

uniform mat4 u_model;
uniform mat4 u_proj;
uniform mat4 u_view;

uniform mat4 u_joints[MAX_JOINTS];
uniform bool u_hasJoints = false;

void main()
{
    mat4 transform = mat4(1.0);

    if(u_hasJoints)
    {
        transform = mat4(0.0);

        for(int i = 0; i < MAX_WEIGHTS && a_joints[i] > -1;
i++)
        {
            transform += u_joints[a_joints[i]] * a_weights[i];
        }
    }

    vertex.UVs = a_uvs;
    transform = u_model * transform;
    vertex.Normal = mat3(transform) * a_normal;
    vertex.Position = (transform * vec4(a_position, 1.0)).xyz;
    gl_Position = u_proj * u_view * transform *
    vec4(a_position, 1.0);
    vertex.TBN = mat3(transform) * mat3(a_tangent, a_bitangent,
a_normal);
```

```
}
```

```
++VERTEX++
```

```
// ... fragment remains unchanged
```

In the vertex attribute section above, you'll observe the introduction of new attributes corresponding to joint indices and their weights, accompanied by a constant denoting the maximum allowable number of joints per model and the maximum number of weights or joints that can influence a vertex. This restriction is in place because we specifically utilize four joints per vertex.

Moving to the uniform variables, two additional uniforms are essential for animation rendering. The first uniform, denoted as `u_joints[MAX_JOINTS]`, comprises an array of matrices that encapsulates the transformations of all joints sent from the engine. The second uniform, `u_hasJoints`, serves the purpose of indicating whether the current model has joints to animate. It ensures the flexibility to render both animated and non-animated objects.

Within the `main()` function, the cumulative transformation of the vertex is computed by multiplying each transformation matrix by the corresponding weight. This resultant transformation is then applied in conjunction with the model matrix. To maintain consistent lighting across the model, special attention is given to animating normals, ensuring

their appropriate adjustment throughout the animation process.

Updating Shader Abstraction

Let us update the PBR shader abstraction class accordingly.

Listing 7.8: Graphics/Shaders/PBR.h

```
struct PbrShader : Shader
{
    EMPY_INLINE PbrShader(const std::string& filename) :
    Shader(filename)
    {
        u_HasJoints = glGetUniformLocation(m_ShaderID,
"u_hasJoints");

        // ...
    }

    EMPY_INLINE void Draw(Model3D& model, PbrMaterial&
material, Transform3D& transform)
    {
        glUniform1i(u_HasJoints, model->HasJoint());

        // ...
    }

    EMPY_INLINE void SetJoints(std::vector<glm::mat4>&
transforms)
    {
        for (size_t i = 0; i < transforms.size() && i <
```

```

100; ++i)
{
    std::string uniform = "u_joints[" +
std::to_string(i) + "]";
    uint32_t u_joint =
glGetUniformLocation(m_ShaderID, uniform.c_str());
    glUniformMatrix4fv(u_joint, 1, GL_FALSE,
glm::value_ptr(transforms[i]));
}
}

// ...

private:
    uint32_t u_HasJoints = 0u;
// ...
}

```

Updating Renderer Class

This newly added function must also be added to the renderer so that we can call it in the application context loop when rendering an animated model.

Listing 7.9: Graphics/Renderer.h

```

struct GraphicsRenderer
{
    EMPTY_INLINE void SetJoints(std::vector<glm::mat4>&
transforms)
{
    m_Pbr->SetJoints(transforms);
}

```

```
    }

    // ...
}
```

Animator Component

Since we are rendering all models using the same function from the shader, we need an additional component for the animator. This component is the key to distinguishing between a static model and a skeletal model during the rendering process.

Listing 7.10: Auxiliairies/ECS. h

```
// model animator
struct AnimatorComponent
{
    EMPLY_INLINE AnimatorComponent(const AnimatorComponent&) = default;
    EMPLY_INLINE AnimatorComponent() = default;
    Animator3D Animator;
};
```

This component holds a pointer to the animator found in a skeletal model.

Showing the Animation

We are now ready to see our first animation. Open your "Application.h" header file and add the following code:

Listing 7.11: Application/Application.h

```
EMPTY_INLINE void RunContext()
{
    // create dir light, skybox

    // load model along with animation
    auto walking =
        std::make_shared<SkeletalModel>("Resources/Models/Walking.fbx");

    // create animated entity
    auto robot = CreateEntt<Entity>();
    robot.Attach<ModelComponent>().Model = walking;
    auto& tr = robot.Attach<TransformComponent>().Transform;
    tr.Translate = glm::vec3(1.5f, -2.5f, -5.0f);
    tr.Scale = glm::vec3(0.03f);
    robot.Attach<AnimatorComponent>().Animator = walking-
>GetAnimator();

    // ...

    while (m_Context->Window->PollEvents())
    {
        // ...

        // render models
        EnttView<Entity, ModelComponent>([this] (auto entity,
auto& comp)
        {
            // check if has animator & compute keyframes
            if(entity.Has<AnimatorComponent>())

```

```

    {
        auto& animator =
entity.Get<AnimatorComponent>().Animator;
        auto& transforms = animator-
>Animate(0.016666f);
        m_Context->Renderer->SetJoints(transforms);
    }

        auto& transform = entity.template
Get<TransformComponent>().Transform;
        m_Context->Renderer->Draw(comp.Model,
comp.Material, transform);
    });

// ...
}
}

```

You can see in the main loop how we are able to check whether the entity has an animator component so we can compute its keyframes before sending the transformation matrices to the shader using the renderer. You should also consider how the entity is created with an animator component. The following model (Figure 7.4) downloaded from Mixamo [Mixamo](#) will be used for testing.



Figure 7.4: Animated Model

With this, you can compile and run your project. You should get the result depicted in (Figure 7.5). Consider how, even when animated, the model can still fit within the scene's lighting. This is all there is to do for animating objects. You can play around with different models and animations to test if everything works fine.



Figure 7.5: Rendering Animated Models

Code not working? clone the branch "skeletal-animation" on the GitHub repository and grant execution rights to the "EmPyLinux.sh" script on Linux. <https://github.com/Madsycode/book-empy-engine>.

```
git clone -branch <branch-name> <repository-link>
```

Part III

Physics & Scripting

8 Physics Simulation

Whatever your hand finds to do, do it with your might.
(King Solomon)

In the realm of gaming, physics plays a pivotal role in enhancing the overall realism, interactivity, and immersive experience for players. The incorporation of physics simulations allows game developers to model the behavior of objects and environments more authentically, creating a dynamic and responsive virtual world. Physics simulations are instrumental in handling collision detection and response. Accurate collision models prevent objects from passing through each other, ensuring that interactions between characters, vehicles, projectiles, and the environment are plausible. This is crucial for creating challenging and enjoyable gameplay experiences where the laws of physics dictate the outcomes of in-game events.

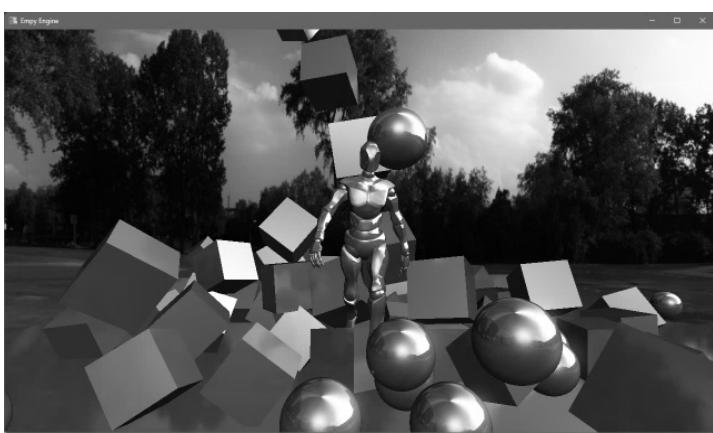
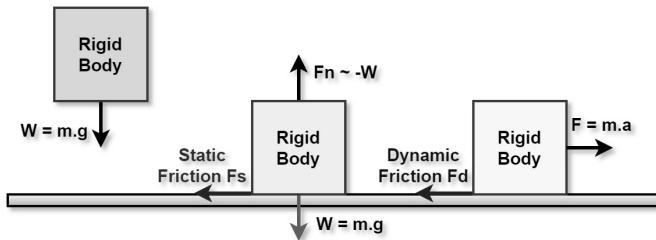


Figure 8.1:

Rigid Body Simulation

Here is an overview of some of the commonly applied simulation:

- **Rigid Body Dynamics:** Rigid body dynamics simulates the motion and interactions of solid objects without deformation. Objects are treated as rigid bodies with mass and can undergo translation and rotation. They are used for simulating the movement and collisions of physical objects in games and simulations, including vehicles, characters, and projectiles. Here are some key concepts and mathematical formulas commonly used in rigid body simulation:
-

**Figure 8.2:**

Rigid Body Kinematics

- ☞ **Newton's Second Law of Motion:** $F = ma$.

Newton's second law relates the force applied to an object (F) to its mass (m) and resulting acceleration (a). In rigid body simulation, this law is used to calculate the

linear acceleration of an object based on the applied forces.

☞ **Torque and Angular Acceleration:** $\tau = Ia$. Torque (τ) is the rotational equivalent of force. It is related to the angular acceleration (a) of an object by the moment of inertia (I). This formula is essential for simulating the rotational motion of rigid bodies.

☞ **Linear Motion Equations:** These equations describe the linear motion of an object. s is the displacement, u is the initial velocity, a is acceleration, t is time, and v is the final velocity.

$$\text{Translation: } s = ut + \frac{1}{2}at^2$$

$$\text{Velocity: } v = u + at$$

☞ **Angular Motion Equations:** These equations describe the angular motion of an object. θ is angular displacement, ω is angular velocity, a is angular acceleration, ω_0 is initial angular velocity, and t is time.

$$\text{Angular Displacement: } \theta = \omega t + \frac{1}{2}at^2$$

$$\text{Angular Velocity: } \omega = \omega_0 + at$$

☞ **Collision Response:** In collision response, the impulse (J) applied to an object is equal to the change in momentum (Δp). This theorem helps calculate the

effects of collisions on linear motion. Impulse-Momentum Theorem: $J = \Delta p$

☞ **Rotational Inertia:** The rotational inertia (I) of an object is the sum of the products of the square of the distance from each mass element (r) to the axis of rotation and the mass (dm) of that element. $I = \int r^2 dm$

These formulas, when integrated into a physics engine, allow for the accurate simulation of rigid body dynamics, enabling realistic interactions and animations in virtual environments. The combination of these equations forms the foundation for rigid body simulation in various applications. The good news is that we do not have to integrate these equations ourselves because we are going to a library that does it for us.

- **Collision Detection:** Collision detection identifies when two or more objects intersect in the virtual world. It is a crucial aspect of physics simulations to handle interactions accurately. This is essential for preventing objects from passing through each other, triggering events on collisions, and calculating realistic responses to impacts.
- **Soft Body Dynamics:** Soft body dynamics simulate the deformation and elasticity of objects. Unlike rigid bodies, soft bodies can bend, stretch, and deform realistically. They are commonly used to simulate cloth, rubber, fluids, and other deformable objects. Provides a more

realistic representation of materials and their responses to forces.

- **Fluid Dynamics:** Fluid dynamics simulates the behavior of liquids and gases. It models the flow, viscosity, and interactions of fluids in the virtual environment. Used for realistic representation of water, smoke, fire, and other fluid-based effects in games, movies, and simulations.
- **Particle Systems:** Particle systems simulate the behavior of a large number of individual particles. Each particle can have properties such as position, velocity, mass, and lifespan. They are widely used for effects like explosions, fire, rain, and snow in games and simulations. Provides a way to model complex behaviors with a large number of entities.
- **Joint and Constraint Systems:** Joint and constraint systems define how objects are connected or restricted in their movement. Joints simulate physical connections between objects. They are used to model complex mechanical systems, such as joints in robotic arms or constraints in a physics-based puzzle game.
- **Force and Torque Simulation:** Physics engines apply forces and torques to objects based on user input, environmental conditions, or scripted events. Used to model the effects of gravity, user controls, wind, and other forces that influence the motion and behavior of objects.

This book integrates physics simulations using NVIDIA's PhysX SDK library. We do not expand on many of the simulations mentioned above because it would be difficult to properly describe them in the context of this book. Gracefully, PhysX has great documentation that provides good guidance on how to perform specific simulations. You can find the link to the documentation for the PhysX version used in this book here: [PhysX \[a\]](#).

8.1 Integrating NVIDIA PhysX

In this section, we embark on the integration of physics simulation capabilities into our engine. NVIDIA PhysX is a widely adopted physics engine known for its efficiency and versatility in simulating a wide range of physical interactions, from rigid body dynamics to complex soft body simulations. It originated with Ageia Technologies, founded in 2002, with a vision to revolutionize gaming physics through dedicated hardware called the PhysX Processing Unit (PPU). In 2005, Ageia introduced the PPU and its proprietary SDK, aiming to enhance physics simulations in games. NVIDIA later acquired Ageia in 2008, integrating PhysX into its GPU architecture. The PhysX library has since evolved into a widely adopted physics engine, offering developers a comprehensive toolset for realistic simulations in virtual worlds.

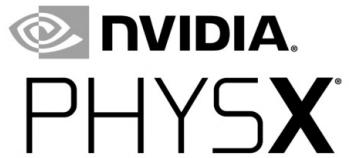


Figure 8.3: NVIDIA PhysX Logo

8.1.1 Getting Started with NVIDIA PhysX

Before diving into the implementation, it is essential to set up the necessary dependencies. We need to ensure that we have the NVIDIA PhysX libraries linked to our project, making them accessible within our development environment. Thankfully, Conan does all of that for us. You can see in the following code how PhysX is added to the list of our requirements.

Listing 8.1: Root/conanfile.txt

```
[requires]
glm/cci.20230113
stb/cci.20230920
opengl/system
spdlog/1.12.0
assimp/5.2.2
entt/3.12.2
physx/4.1.1
```

```

glfw/3.3.8
glew/2.2.0

[generators]
cmake

[options]
physx:shared=True
glew:shared=False
glfw:shared=False
assimp:shared=False

```

Important to note here is the line *physx : shared = True* in the "option" section. This tells Conan to use PhysX as a shared library. It is possible to use PhysX as a static library, but I could not, on top of my head, find out how to do this. So I settled for the shared version. This means we need to copy all the ".dll" files of PhysX into the build folder for our application to use them at runtime. Do not worry; you don't have to do that manually. We can automate this using CMake's post-build command. This is done in the following code snippet.

Listing 8.2: EmpyEngine/CMakeLists.txt

```

# copy .dll files
if (MSVC)
    foreach(directory ${CONAN_BIN_DIRS})
        file(GLOB dllfiles ${directory}/*.dll)
        foreach(dllfile ${dllfiles})
            if (EXISTS ${dllfile})
                add_custom_command(TARGET ${PROJECT_NAME}

```

```
POST_BUILD
    COMMAND ${CMAKE_COMMAND} -E
copy_if_different
    ${dllfile} ${EXECUTABLE_OUTPUT_PATH})
endif()
endforeach()
endforeach()
endif(MSVC)
```

Copy this code snippet and add it to your engine's "CMakeLists.txt" file. As you can see, the command only runs if we are currently on a Windows operating system. Linux operating systems use shared objects (.so), which do not require additional runtime files. This command will be executed after the compilation is completed. If you look closely, you will notice that this command does not copy the ".dll" files of the PhysX but all dynamically linked libraries. This has the effect that every time a new shared library is added to "conanfile.txt", CMake will also copy its ".dll" files into the build folder.

8.1.2 Initialization and Configuration

We need to initialize the PhysX SDK before we can create simulations with it. This involves setting up the necessary components, such as the physics scene, materials, and other parameters. Proper initialization ensures a stable and consistent physics environment for our simulations. Start by extending your project directory tree as depicted in (Figure 8.4).

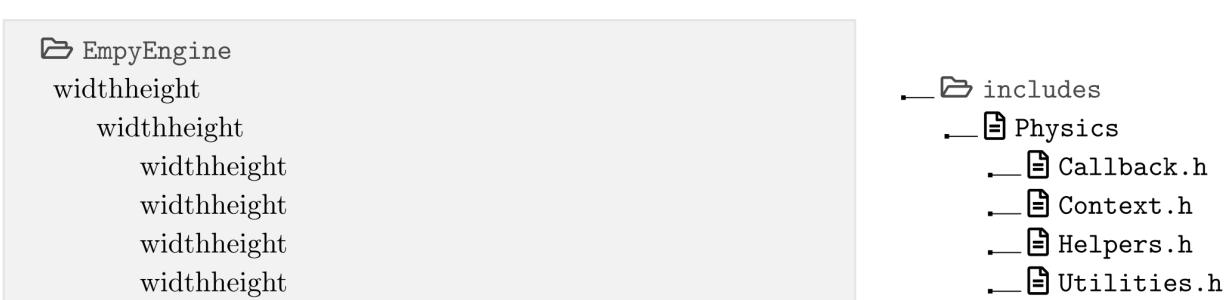


Figure 8.4: Physics Directory

Physics Helper Functions

Because PhysX uses custom vector types that are different from GLM's, we need to define some helper functions to convert the value from one context to another. The code snippet provided in (Listing 8.3) below defines two helper functions for this purpose. The first function, `PxToVec3()`, is used to convert a PhysX `PxVec3` to a GLM `glm::vec3`. The second function, `ToPxVec3()`, performs the reverse conversion.

Listing 8.3: Physics/Helpers.

```
#pragma once  
#include "Common/Core.h"  
#include <PxPhysicsAPI.h>  
  
using namespace physx;
```

```

namespace EmPy
{
    // helper function to convert PhysX Vec3 to GLM vec3
    EMPY_INLINE glm::vec3 PxToVec3(const PxVec3& physxVec)
    {
        return glm::vec3(physxVec.x, physxVec.y, physxVec.z);
    }

    // helper function to convert GLM vec3 to PhysX Vec3
    EMPY_INLINE PxVec3 ToPxVec3(const glm::vec3& glmVec)
    {
        return PxVec3(glmVec.x, glmVec.y, glmVec.z);
    }
}

```

You can also see that the "**PxPhysicsAPI.h**" header is included, which contains all the necessary declarations for PhysX functionality. To make our experience with PhysX more pleasant, we forwardly define its namespace so we never have to use it when creating variables or calling functions.

Custom Event Callback

The event callback type defined in the code provided in (Listing [8.4](#)), represents a custom event callback type which is meant to be the point of exchange between PhysX and our engine. This event callback must derived from the PhysX event callback interface `PxSimulationEventCallback` which gives us the ability to override virtual function to capture collision events in the PhysX simulation.

Listing 8.4: Physics/Callback.h

```
#pragma once
#include "Helpers.h"
#include "Auxiliaries/ECS.h"

namespace Empty
{
    enum class PxEvent
    {
        UNKNOWN = 0,
        TRIGGER,
        CONTACT,
        SLEEP,
        WAKE
    };

    struct PxPayload
    {
        EntityID Entity1 = NENTT;
        EntityID Entity2 = NENTT;
        PxEvent Event = PxEvent::UNKNOWN;
    };

    using PxCallbackFunction = std::function<void(const
PxPayload&)>;
}

struct PxEventCallback : public PxSimulationEventCallback
{
    // override the onContact callback
    EMPTY_INLINE void onContact(const PxContactPairHeader&
header, const
PxContactPair* pairs, PxU32 nbPairs) override
{
```

```

        // collision actors
        auto actor1 = header.actors[0];
        auto actor2 = header.actors[1];

        // check if actors
        if (m_Callback && actor1 && actor2)
        {
            PxPayload event;
            event.Event = PxEvent::CONTACT;
            event.Entity1 = *static_cast<EntityID*>
(actor1->userData);
            event.Entity2 = *static_cast<EntityID*>
(actor2->userData);
            m_Callback(event);
        }

        EMPLY_DEBUG("onContact Event!");
    }

    // override the onTrigger callback
    EMPLY_INLINE void onTrigger(PxTriggerPair* pairs,
PxU32 nbPairs) override
{
    for (PxU32 i = 0; m_Callback && i < nbPairs;
++i)
    {
        // Access actor pointers from the trigger
pair
        const PxRigidActor* actor0 =
pairs[i].triggerActor;
        const PxRigidActor* actor1 =
pairs[i].otherActor;

        // Check if actors are valid before using
        if (actor0 && actor1)
        {

```

```

        PxPayload event;
        event.Event = PxEvent::TRIGGER;
        event.Entity1 = *static_cast<EntityID*>
(actor1->userData);
        event.Entity2 = *static_cast<EntityID*>
(actor0->userData);
        m_Callback(event);
    }
}

EMPTY_DEBUG("onTrigger Event!");

}

EMPTY_INLINE void onAdvance(const PxRigidBody*const*
bodyBuffer,
    const PxTransform* poseBuffer, const PxU32 count)
override { }

EMPTY_INLINE void onSleep(PxActor** actors, PxU32
count)
override { EMPTY_DEBUG("onSleep Event!"); }

EMPTY_INLINE void onWake(PxActor** actors, PxU32
count)
override { EMPTY_DEBUG("onWake Event!"); }

EMPTY_INLINE void onConstraintBreak(PxConstraintInfo*
constraints, PxU32 count) override { }

private:
PxCallbackFunction m_Callback;
friend struct PhysicsContext;
};

}

```

Starting at the top, you will see the enumerator `PxEvent` we have created to list different types of events followed by the payload data type `PxPayload`. Every time a collision event will be triggered, this payload will be set along with the event type. It will be used by the member callback function `m_Callback` to notify our engine about the event.

The event callback structure overrides all the virtual functions from the PhysX event callback interface, especially `onContact()` and `onTrigger()`, which are called when contact (collision with simulation) and trigger (collision without simulation) events occur, respectively. These functions extract relevant information from PhysX callback parameters, create `PxPayload` instances, and invoke the callback function `m_Callback`. This callback function will be implemented later in the next chapter when scripting will be applied. Additionally, the structure provides empty implementations for other PhysX event-related callbacks, such as `onAdvance()`, `onSleep()`, `onWake()`, and `onConstraintBreak()`. This must be done because these are pure virtual functions.

Physics Context Class

In order to keep things organized, we need to create a class that holds all the necessary information about PhysX in our engine. This is done in the code depicted in ([Listing 8.5](#))

below. The class `PhysicsContext` serves as a wrapper for managing the initialization and cleanup of the NVIDIA PhysX library. In the constructor, the code initializes the PhysX SDK by creating a foundation, a physics instance, setting up worker threads, and creating a physics scene. A foundation in PhysX is an essential component that serves as the fundamental building block for the entire physics simulation framework. The PhysX foundation provides a platform-independent base for the initialization and operation of the physics engine. It encapsulates fundamental functionalities such as memory allocation, error handling, and basic system-level configuration. The dispatcher here is initialized with two workers. You can perform simulations in the main thread by providing zero workers. However, remember that your application will not be as scalable as you would like it to be, as the calculation of physics simulations for thousands of objects can be quite expensive.

Listing 8.5: Physics/Context.
h

```
#pragma once
#include "Callback.h"
#include "Utilities.h"

namespace Empy
{
    struct PhysicsContext
    {
        EMPY_INLINE PhysicsContext()
        {

```

```

        // sinitialize physX SDK
        m_Foundation =
PxCreateFoundation(PX_PHYSICS_VERSION, m_AllocatorCallback,
m_ErrorCallback);
        if (!m_Foundation)
{
    EMFY_ERROR("Error initializing PhysX
m_Foundation");
    return;
}

        // create context instance
        m_Physics = PxCreatePhysics(PX_PHYSICS_VERSION,
*m_Foundation, PxTolerancesScale());
        if (!m_Physics)
{
    EMFY_ERROR("Error initializing PhysX
m_Physics");
    m_Foundation->release();
    return;
}

        // create worker threads
        m_Dispatcher = PxDefaultCpuDispatcherCreate(2);

        // create a scene desciption
        PxSceneDesc sceneDesc(m_Physics-
>getTolerancesScale());
        sceneDesc.simulationEventCallback =
&m_EventCallback;
        sceneDesc.gravity = PxVec3(0.0f, -9.81f, 0.0f);
        sceneDesc.filterShader = CustomFilterShader;
        sceneDesc.cpuDispatcher = m_Dispatcher;

        // create scene instance
        m_Scene = m_Physics->createScene(sceneDesc);

```

```

    if (!m_Scene)
    {
        EMPTY_ERROR("Error creating PhysX m_Scene");
        m_Physics->release();
        m_Foundation->release();
        return;
    }
}

EMPTY_INLINE ~PhysicsContext()
{
    if (m_Scene) { m_Scene->release(); }
    if (m_Physics) { m_Physics->release(); }
    if (m_Dispatcher) { m_Dispatcher->release(); }
    if (m_Foundation) { m_Foundation->release(); }
}

private:
    // custom collision filter shader callback
    static PxFilterFlags CustomFilterShader
    (
        PxFilterObjectAttributes attributes0,
        PxFilterData filterData0,
        PxFilterObjectAttributes attributes1,
        PxFilterData filterData1,
        PxPairFlags& pairFlags, const void*
        constantBlock, PxU32 constantBlockSize
    )
    {
        // generate contacts and triggers for actors
        pairFlags |= PxPairFlag::eCONTACT_DEFAULT |
            PxPairFlag::eTRIGGER_DEFAULT;

        return PxFilterFlag::eDEFAULT;
    }
}

```

```
private:  
    PxDefaultErrorCallback m_ErrorCallback;  
    PxDefaultAllocator m_AllocatorCallback;  
    PxDefaultCpuDispatcher* m_Dispatcher;  
    PxEventCallback m_EventCallback;  
    PxFoundation* m_Foundation;  
    PxPhysics* m_Physics;  
    PxScene* m_Scene;  
};  
}
```

The static function `CustomFilterShader(...)` defines a custom collision filter shader callback function. In PhysX, collision filtering is crucial for controlling interactions between different types of objects in a physics simulation. The custom filter shader is a user-defined function responsible for specifying how collisions and triggers are handled between pairs of objects. The purpose of the function is to determine how collisions and triggers should be handled between the two objects based on their attributes and filter data. Just as we needed shaders to render objects, we also need a shaders in PhysX to properly handle collisions between actors.

The flags `eCONTACT_DEFAULT` and `eTRIGGER_DEFAULT` are added to `pairFlags` to indicate that default collision contacts and triggers should be generated for the pair of objects. PhysX does not notify us by default when collisions happen, which is why we need to explicitly tell it what to do when they happen. The function returns `eDEFAULT`, indicating that the

default collision filtering behavior should be applied. This typically means that PhysX should use its default criteria for determining whether two objects should collide or trigger based on their attributes and filter data. Read more about collision filtering here: [PhysX \[b\]](#).

8.1.3 Rigid Bodies and Colliders

With the ability to initialize PhysX in place, we can proceed to create rigid bodies and colliders to represent the physical entities in the physics simulations. Rigid bodies define the mass, motion, and orientation of objects, while colliders determine their shapes for collision detection.

Data Structure

We want to incorporate PhysX with our already established ECS framework. That's why we will create `RigidBody` and `Collider` components so we can properly manipulate their states during both the physics simulation and the rendering.

In the code snippet depicted in (Listing 8.6) below, the `RigidBody3D` structure encapsulates properties related to a 3D rigid body. It contains a pointer to a `PxRigidActor` found in the PhysX context. Additionally, it includes a `Density` property to specify the body's density and a `Type` enumerator, allowing the designation of whether the rigid

body is dynamic or static. In other words, whether the rigid body actor should be affected by gravity during the simulation or not.

Listing 8.6: Physics/Utilities. h

```
#pragma once
#include "Helpers.h"

namespace EmPy
{
    struct RigidBody3D
    {
        EMPY_INLINE RigidBody3D(const RigidBody3D&) = default;
        EMPY_INLINE RigidBody3D() = default;

        // body actor pointer
        PxRigidActor* Actor = nullptr;

        // body density
        float Density = 1.0f;

        // rigidbody type
        enum {
            DYNAMIC = 0,
            STATIC,
        } Type;
    };

    struct Collider3D
    {
        EMPY_INLINE Collider3D(const Collider3D&) = default;
        EMPY_INLINE Collider3D() = default;
    };
}
```

```
// collider material pointer
PxMaterial* Material = nullptr;

// collider material data
float DynamicFriction = 0.5f;
float StaticFriction = 0.0f;
float Restitution = 0.1f;

// mesh for custom shape
PxConvexMeshGeometry Mesh;

// collider geometry shape
PxShape* Shape = nullptr;

// collider shape type
enum {
    BOX = 0,
    SPHERE,
    MESH
} Type;
};

}
```

On the other hand, the `Collider3D` structure focuses on properties related to 3D colliders. It holds a pointer to a `PxMaterial`, allowing customization of physical properties like dynamic friction, static friction, and restitution. The structure also includes a pointer to a `PxShape` representing the geometry shape of the collider and a `Type` enumerator, indicating whether the collider shape is a box, sphere, or a custom mesh.

Creating Rigid Bodies

Let us now go back to the `PhysicsContext` class we created earlier to implement functions to add rigid bodies. The function `AddRigidBody()` depicted in (Listing 8.7) below is responsible for adding a rigid body into the PhysX simulation context. It takes an `Entity` object as argument and initializes the pose or the transformation of the rigid body within the PhysX world. The rotation of the rigid body is then computed using the rotation from the transform component.

Listing 8.7: Physics/Context.
h

```
#pragma once
#include "Callback.h"
#include "Utilities.h"

namespace Empy
{
    struct PhysicsContext
    {
        /* ... same as before ... */

        EMPY_INLINE void AddRigidBody(Entity& entity)
        {
            auto& transform = entity.template
Get<TransformComponent>().Transform;
            auto& body = entity.template
Get<RigidBodyComponent>().RigidBody;
            bool hasCollider = entity.template
Has<ColliderComponent>();
        }
}
```

```

        // create rigidbody transformation
        PxTransform pose(ToPxVec3(transform.Translate));
        glm::quat rot(transform.Rotation);
        pose.q = PxQuat(rot.x, rot.y, rot.z, rot.w);

        // create a rigid body actor
        if(entity.template Has<ColliderComponent>())
        {
            // create collider shape

            auto& collider = entity.template
Get<ColliderComponent>().Collider;

            // create collider material
            collider.Material = m_Physics-
>createMaterial(collider.StaticFriction,
                collider.DynamicFriction,
collider.Restitution
            );
        }

        if(collider.Type == Collider3D::BOX)
        {
            PxBoxGeometry
box(ToPxVec3(transform.Scale/2.0f));
            collider.Shape = m_Physics-
>createShape(box, *collider.Material);
        }
        else if(collider.Type == Collider3D::SPHERE)
        {
            PxSphereGeometry
sphere(transform.Scale.x/2.0f);
            collider.Shape = m_Physics-
>createShape(sphere, *collider.Material);
        }
        else if(collider.Type == Collider3D::MESH)

```

```

    {
        // coming next!
    }
    else
    {
        EMPTY_ERROR("Error creating collider
invalid type provided");
        return;
    }

    // create actor instance

    if(body.Type == RigidBody3D::DYNAMIC)
    {
        body.Actor = PxCreateDynamic(*m_Physics,
pose, *collider.Shape, body.Density);
        body.Actor-
>setActorFlag(PxActorFlag::eSEND_SLEEP_NOTIFIES, true);
    }
    else if(body.Type == RigidBody3D::STATIC)
    {
        body.Actor = PxCreateStatic(*m_Physics,
pose, *collider.Shape);
    }
}
else
{
    if(body.Type == RigidBody3D::DYNAMIC)
    {
        body.Actor = m_Physics-
>createRigidDynamic(pose);
    }
    else if(body.Type == RigidBody3D::STATIC)
    {
        body.Actor = m_Physics-
>createRigidStatic(pose);
}

```

```

        }

    }

    // check actor
    if (!body.Actor)
    {
        EMPTY_ERROR("Error creating dynamic actor");
        return;
    }

    // set user data to entt id
    body.Actor->userData = new EntityID(entity.ID());

    // add actor to the m_Scene
    m_Scene->addActor(*body.Actor);
}

};

}

```

This function begins by extracting the necessary components from the entity, such as the transformation and the rigid body data. It also checks if a collider is present in the entity by using. Next, if a collider is present, the function proceeds to create the collider shape and material. The type of collider (box or sphere) is determined, and the appropriate PhysX geometry is created accordingly. After creating the collider shape and material, the function creates a PhysX actor based on the rigid body type (dynamic or static). For dynamic bodies, it sets additional properties such as density and sleep notification flags (this is necessary if we want the object to notify its state). If no collider is present, the function directly

creates a dynamic or static actor based on the rigid body type.

If the actor is created successfully, we set the user data of the PhysX actor to the entity's ID, allowing for association between the physics simulation and the entity within our ECS framework. Finally, the actor is added to the physics scene. The user data here is crucial. Without it, we wouldn't have a way to know which entity has which rigid body in the physics simulation. This will also prevent us from publishing the event with the proper entities involved in the collision.

Mesh Colliders

In certain situations, you might want to create a custom shape for your colliders for specific reasons. This can be done with PhysX as well. The following member function of `PhysicsContext` provides the means for you to achieve that.

Listing 8.8: Physics/Context. h

```
/* ... same as before ... */

EMPY_INLINE PxConvexMeshGeometry CookMesh(const
MeshData<ShadedVertex>& data)
{
    // px vertex container
    std::vector<PxVec3> vertices;
```

```

    // convert position attributes
    for(auto& vertex : data.Vertices)
    {
        vertices.push_back(ToPxVec3(vertex.Position));
    }

    PxConvexMeshDesc meshDesc;
    // vertices
    meshDesc.points.data = vertices.data();
    meshDesc.points.stride = sizeof(PxVec3);
    meshDesc.points.count = vertices.size();
    // indices
    meshDesc.indices.data = data.Indices.data();
    meshDesc.indices.count = data.Indices.size();
    // flags
    meshDesc.flags = PxConvexFlag::eCOMPUTE_CONVEX;

    // cooking the mesh
    PxCookingParams cookingParams =
    PxCookingParams(PxTolerancesScale());
    PxCooking* cooking = PxCreateCooking(PX_PHYSICS_VERSION,
*m_Foundation, cookingParams);
    PxConvexMeshCookingResult::Enum result;
    PxConvexMesh* convexMesh = cooking-
>createConvexMesh(meshDesc,
    m_Physics->getPhysicsInsertionCallback(), &result);
    PxConvexMeshGeometry convexMeshGeometry(convexMesh);

    cooking->release();
    return convexMeshGeometry;
}

```

You can see how the function takes a `Meshdata<ShadedVertex>` parameter similar to the mesh class we created in the previous part of the book. The function

starts by converting the vertices in a way that PhysX will be able to understand them. Then, it defines all the configurations of the mesh, which is finally created, resulting in a `PxConvexMeshGeometry`. This process is called "**Cooking**" in PhysX. This function can then be called in the `AddRigidBody()` method to create a custom geometry shape for rigid actors with a mesh collider.

8.1.4 Handling Physics Interactions

Physics simulations come to life through the interactions between objects. We need to use physics to perform all the simulations so that we can later use the results to render the entities in our scene. This is quite simple to do with PhysX. The following code snippet in (Listing 8.9) provides the implementation of the `Simulate()` function, which runs all the simulations using the workers defined in the dispatcher.

Listing 8.9: Physics/Context. h

```
/* ... same as before ... */

EMPY_INLINE void Simulate(uint32_t step, float dt)
{
    for (int i = 0; i < step; ++i)
    {
        // simulate m_Physics for a time step
        m_Scene->simulate(dt);

        // block until simulation is complete
```

```
    m_Scene->fetchResults(true);  
}  
}
```

This function iterates through the specified number of simulation steps. In each iteration, it simulates the physics scene for a time step of a second, which is simply the delta-time (more about this in a bit). A higher step count means faster simulations, and vice versa. The `fetchResults()` method is then called with the parameter set to `true`, ensuring that the function blocks until the simulation for the current time step is complete and the results are fetched. This is needed because we have multiple asynchronous workers performing the computation concurrently. Here is the entire code of the `PhysicsContext` class:

Listing 8.10: Physics/Context.h

```
#pragma once  
  
#include "Callback.h"  
#include "Utilities.h"  
  
namespace EmPy  
{  
    struct PhysicsContext  
    {  
        EMPY_INLINE PhysicsContext()  
        {  
            // sinitialize physX SDK  
            m_Foundation =  
PxCreateFoundation(PX_PHYSICS_VERSION, m_AllocatorCallback,  
m_ErrorCallback);  
    };  
};
```

```

    if (!m_Foundation)
    {
        EMPIY_ERROR("Error initializing PhysX
m_Foundation");
        return;
    }

    // create context instance
    m_Physics = PxCreatePhysics(PX_PHYSICS_VERSION,
*m_Foundation, PxTolerancesScale());
    if (!m_Physics)
    {
        EMPIY_ERROR("Error initializing PhysX
m_Physics");
        m_Foundation->release();
        return;
    }

    // create worker threads
    m_Dispatcher = PxDefaultCpuDispatcherCreate(2);

    // create a scene desciption
    PxSceneDesc sceneDesc(m_Physics-
>getTolerancesScale());
    sceneDesc.simulationEventCallback =
&m_EventCallback;
    sceneDesc.gravity = PxVec3(0.0f, -9.81f, 0.0f);
    sceneDesc.filterShader = CustomFilterShader;
    sceneDesc.cpuDispatcher = m_Dispatcher;

    // create scene instance
    m_Scene = m_Physics->createScene(sceneDesc);
    if (!m_Scene)
    {
        EMPIY_ERROR("Error creating PhysX m_Scene");
        m_Physics->release();
    }
}

```

```

        m_Foundation->release();
        return;
    }

}

EMPTY_INLINE ~PhysicsContext()
{
    if (m_Scene) { m_Scene->release(); }
    if (m_Physics) { m_Physics->release(); }
    if (m_Dispatcher) { m_Dispatcher->release(); }
    if (m_Foundation) { m_Foundation->release(); }
}

EMPTY_INLINE void AddRigidBody(Entity& entity)
{
    auto& transform = entity.template
Get<TransformComponent>().Transform;
    auto& body = entity.template
Get<RigidBodyComponent>().RigidBody;
    bool hasCollider = entity.template
Has<ColliderComponent>();

    // create rigidbody transformation
    PxTransform pose(ToPxVec3(transform.Translate));
    glm::quat rot(transform.Rotation);
    pose.q = PxQuat(rot.x, rot.y, rot.z, rot.w);

    // create a rigid body actor
    if(entity.template Has<ColliderComponent>())
    {
        // create collider shape

        auto& collider = entity.template
Get<ColliderComponent>().Collider;

        // create collider material
    }
}

```

```

        collider.Material = m_Physics-
>createMaterial(collider.StaticFriction,
                  collider.DynamicFriction,
collider.Restitution
    );

    if(collider.Type == Collider3D::BOX)
    {
        PxBoxGeometry
box(ToPxVec3(transform.Scale/2.0f));
        collider.Shape = m_Physics-
>createShape(box, *collider.Material);
    }
    else if(collider.Type == Collider3D::SPHERE)
    {
        PxSphereGeometry
sphere(transform.Scale.x/2.0f);
        collider.Shape = m_Physics-
>createShape(sphere, *collider.Material);
    }
    else
    {
        EMY_ERROR("Error creating collider
invalid type provided");
        return;
    }

    // create actor instance

    if(body.Type == RigidBody3D::DYNAMIC)
    {
        body.Actor = PxCreateDynamic(*m_Physics,
pose, *collider.Shape, body.Density);
        body.Actor-
>setActorFlag(PxActorFlag::eSEND_SLEEP_NOTIFIES, true);
    }

```

```

        else if(body.Type == RigidBody3D::STATIC)
        {
            body.Actor = PxCreateStatic(*m_Physics,
pose, *collider.Shape);
        }
    }
else
{
    if(body.Type == RigidBody3D::DYNAMIC)
    {
        body.Actor = m_Physics-
>createRigidDynamic(pose);
    }
    else if(body.Type == RigidBody3D::STATIC)
    {
        body.Actor = m_Physics-
>createRigidStatic(pose);
    }
}

// check actor
if (!body.Actor)
{
    EMPTY_ERROR("Error creating dynamic actor");
    return;
}

// set user data to entt id
body.Actor->userData = new EntityID(entity.ID());

// add actor to the m_Scene
m_Scene->addActor(*body.Actor);
}

EMPTY_INLINE void Simulate(uint32_t step, float dt)
{

```

```

        for (int i = 0; i < step; ++i)
        {
            // simulate m_Physics for a time step
            m_Scene->simulate(dt);
            // block until simulation is complete
            m_Scene->fetchResults(true);
        }
    }

EMPTY_INLINE void
SetEventCallback(PxCallbackFunction&& callback)
{
    m_EventCallback.m_Callback = callback;
}

EMPTY_INLINE PxConvexMeshGeometry CookMesh(const
MeshData<ShadedVertex>& data)
{
    // px vertex container
    std::vector<PxVec3> vertices;

    // convert position attributes
    for(auto& vertex : data.Vertices)
    {
        vertices.push_back(ToPxVec3(vertex.Position));
    }

    PxConvexMeshDesc meshDesc;
    // vertices
    meshDesc.points.data = vertices.data();
    meshDesc.points.stride = sizeof(PxVec3);
    meshDesc.points.count = vertices.size();
    // indices
    meshDesc.indices.data = data.Indices.data();
    meshDesc.indices.count = data.Indices.size();
    // flags
}

```

```

    meshDesc.flags = PxConvexFlag::eCOMPUTE_CONVEX;

        // cooking the mesh
        PxCookingParams cookingParams =
PxCookingParams(PxTolerancesScale());
        PxCooking* cooking =
PxCreateCooking(PX_PHYSICS_VERSION, *m_Foundation,
cookingParams);
        PxConvexMeshCookingResult::Enum result;
        PxConvexMesh* convexMesh = cooking-
>createConvexMesh(meshDesc,
        m_Physics->getPhysicsInsertionCallback(),
&result);
        PxConvexMeshGeometry
convexMeshGeometry(convexMesh);

        cooking->release();
        return convexMeshGeometry;
    }

private:
    // custom collision filter shader callback
    static PxFilterFlags CustomFilterShader
    (
        PxFilterObjectAttributes attributes0,
PxFilterData filterData0,
        PxFilterObjectAttributes attributes1,
PxFilterData filterData1,
        PxPairFlags& pairFlags, const void*
constantBlock, PxU32 constantBlockSize
    )
    {
        // generate contacts and triggers for actors
        pairFlags |= PxPairFlag::eCONTACT_DEFAULT |
PxPairFlag::eTRIGGER_DEFAULT;

```

```
        return PxFilterFlag::eDEFAULT;
    }

private:
    PxDefaultErrorCallback m_ErrorCallback;
    PxDefaultAllocator m_AllocatorCallback;
    PxDefaultCpuDispatcher* m_Dispatcher;
    PxEventCallback m_EventCallback;
    PxFoundation* m_Foundation;
    PxPhysics* m_Physics;
    PxScene* m_Scene;
};

}
```

8.1.5 Integration with Application

To achieve seamless integration between physics and the rest of the application, we must synchronize the physics simulation with our rendering pipeline. This involves updating the positions and orientations of objects based on their PhysX state.

Integrating Physics Context

Let us add the `PhysicsContext` to the application context so we can use it in the application main loop to perform simulations when needed. Update your `AppContext` as depicted in the following code:

Listing 8.11: Application/Context.h

```
#pragma once

#include "Window/Window.h"
#include "Physics/Context.h"
#include "Graphics/Renderer.h"

namespace EmPy
{
    // forward declaration
    struct AppInterface;

    // application context
    struct ApplicationContext
    {
        EMPY_INLINE ApplicationContext()
        {
            Window = std::make_unique<AppWindow>(&Dispatcher,
1280, 720, "EmPy Engine");
            Renderer = std::make_unique<GraphicsRenderer>
(1280, 720);
            Physics = std::make_unique<PhysicsContext>();
        }

        EMPY_INLINE ~ApplicationContext()
        {
            for(auto layer : Layers)
            {
                EMPY_DELETE(layer);
            }
        }

        std::unique_ptr<GraphicsRenderer> Renderer;
        std::unique_ptr<PhysicsContext> Physics;
    };
}
```

```

        std::vector<AppInterface*> Layers;
        std::unique_ptr<AppWindow> Window;
        EventDispatcher Dispatcher;
        EntityRegistry Scene;
    } ;
}

```

So far, everything looks good.

Defining Physics Components

Let us create components to store the actual properties of rigid actors. See code below in ([Listing 8.12](#)).

Listing 8.12: Auxiliaries/ECS. h

```

// rigid body component
struct RigidBodyComponent
{
    EMPI_INLINE RigidBodyComponent(const RigidBodyComponent&)
= default;
    EMPI_INLINE RigidBodyComponent() = default;
    RigidBody3D RigidBody;
};

// collider component
struct ColliderComponent
{
    EMPI_INLINE ColliderComponent(const ColliderComponent&) =
default;
    EMPI_INLINE ColliderComponent() = default;
    Collider3D Collider;
};

```

```
};

/* ... same as before ... */
```

Running the Simulation

You might have noticed that the physics simulation is not running on the actual entities but on an actor created in the simulation context. This means we need to get the simulated results from PhysX to update the actual entity in our scene. But we want things to be a little more organized moving forward. The following code in (Listing 8.13) shows how the Application() class has been reorganized.

Listing 8.13: Application/Application.h

```
/* ... same as before ... */

struct Application : AppInterface
{
    // creates application context
    EMPTY_INLINE Application()
    {
        // initialize app context
        m_LayerID = TypeID<Application>();
        m_Context = new ApplicationContext();

        // register event callbacks
        RegisterEventCallbacks();

        // create scene entities
    }
};
```

```
    CreateSceneEntities();

    // create physics actors
    CreatePhysicsActors();

    // create environm. maps
    CreateSkyboxEnvMaps();
}

// destroy application context
EMPTY_INLINE ~Application()
{
    // release physics actors
    DestroyPhysicsActors();
    EMPTY_DELETE(m_Context);
}

// runs application main loop
EMPTY_INLINE void RunContext()
{
    // application main loop
    while(m_Context->Window->PollEvents())
    {
        // compute delta time value
        ComputeFrameDeltaTime();

        // run physics simulation
        RunPhysicsSimulation();

        // render scene shadow map
        RenderSceneDepthMap();

        // render scene to buffer
        RenderSceneToFBO();

        // update all layers
    }
}
```

```

        UpdateAppLayers();
    }

}

private:
    EMPY_INLINE void RegisterEventCallbacks() /* ... */
    EMPY_INLINE void ComputeFrameDeltaTime() /* ... */
    EMPY_INLINE void RunPhysicsSimulation() /* ... */
    EMPY_INLINE void DestroyPhysicsActors() /* ... */
    EMPY_INLINE void CreateSkyboxEnvMaps() /* ... */
    EMPY_INLINE void RenderSceneDepthMap() /* ... */
    EMPY_INLINE void CreatePhysicsActors() /* ... */
    EMPY_INLINE void CreateSceneEntities() /* ... */
    EMPY_INLINE void RenderSceneToFBO() /* ... */
    EMPY_INLINE void UpdateAppLayers() /* ... */
}

```

All we have done here is rearrange everything that was in this class previously to add more clarity to what is going on. You will note the functions `CreatePhysicsActors()`, `RunPhysicsSimulation()`, `DestroyPhysicsActors()`, etc., which are added to handle the creation, simulation, and destruction of all rigid actors, as well as much more. Let us take a look at the implementation of all these functions. Keep in mind that most functions found in this code introduce anything to what was there before. It is just reorganized.

Adding the Delta Time

In the realms of game programming and computer graphics, the term "delta time" signifies a crucial concept aimed at

maintaining a hardware-independent pace for processes within a game loop. It denotes the time variance between the preceding frame and the presently rendered frame. The introduction of the delta time concept addressed a challenge observed in early computer games. These games exhibited varied speeds based on the processor's speed, occasionally rendering them unplayable on faster processors than initially intended by the game developer.

We need to integrate this to make animations and physics simulations time-based and not frame rate-based. The following function can be added to the application class to perform the delta time computation:

Listing 8.14: Application/Application.h

```
/* ... same as before ... */

// computes delta time value
EMPY_INLINE void ComputeFrameDeltaTime()
{
    static double sLastTime = glfwGetTime();
    double currentTime = glfwGetTime();
    m_Context->DeltaTime = (currentTime - sLastTime);
    sLastTime = currentTime;
}
```

We are then required to add the *DeltaTime* member variable to the application context as depicted below.

Listing 8.15: Application/Context.h

```
/* ... same as before ... */

struct ApplicationContext
{
    EMPTY_INLINE ApplicationContext()
    {
        Window = std::make_unique<AppWindow>(&Dispatcher,
1280, 720, "Empty Engine");
        Renderer = std::make_unique<GraphicsRenderer>(1280,
720);
        Physics = std::make_unique<PhysicsContext>();
        DeltaTime = 0.0;
    }

    /* ... same as before ... */

    std::unique_ptr<GraphicsRenderer> Renderer;
    std::unique_ptr<PhysicsContext> Physics;
    std::vector<AppInterface*> Layers;
    std::unique_ptr<AppWindow> Window;
    EventDispatcher Dispatcher;
    EntityRegistry Scene;
    double DeltaTime; // <-- deltatime
};
```

Functions Implementation

We can then provide the implementation of all functions depicted in the application main loop found in ([Listing 8.13](#)).

☞ **RegisterEventCallbacks()**: This function helps register all event callbacks. For now, the physics event callback function does not do anything, but this will be addressed in the next chapter when scripting is added. The goal is to notify the scripts about the collision so they can take action.

Listing 8.16: Application/Application.
h

```
/* ... same as before ... */

EMPTY_INLINE void RegisterEventCallbacks()
{
    // set physics event callback
    m_Context->Physics->SetEventCallback([this] (auto e)
    {
        // coming later with scripting
    });

    // attach window resize event callback
    AttachCallback<WindowResizeEvent>([this] (auto e)
    {
        m_Context->Renderer->Resize(e.Width, e.Height);
    });
}
```

☞ **RunPhysicsSimulation()**: This function runs all the rigid body simulations and fetches the result back to the scene for proper rendering. You will note the `DeltaTime` variable used to run the simulation.

Listing 8.17: Application/Application.
h

```

/* ... same as before ... */

// runs physics and fetches physics
EMPY_INLINE void RunPhysicsSimulation()
{
    // compute physx
    m_Context->Physics->Simulate(1, m_Context->DeltaTime);

    // start physics
    EnttView<Entity, RigidBodyComponent>([this] (auto entity,
auto& comp)
    {
        auto& transform = entity.template
Get<TransformComponent>().Transform;
        auto pose = comp.RigidBody.Actor->getGlobalPose();
        glm::quat rot(pose.q.x, pose.q.y, pose.q.z,
pose.q.w);
        transform.Rotation =
glm::degrees(glm::eulerAngles(rot));
        transform.Translate = PxToVec3(pose.p);
    });
}

```

☞ **DestroyPhysicsActors():** This function is there to make sure that when the simulation is done, all the actors, materials, and colliders are destroyed from the physics context.

Listing 8.18: Application/Application.h

```

/* ... same as before ... */

// destroys all actors and colliders

```

```

EMPTY_INLINE void DestroyPhysicsActors()
{
    EnttView<Entity, RigidBodyComponent>([this] (auto entity,
auto& comp)
    {
        if(entity.template Has<ColliderComponent>())
        {
            auto& collider = entity.template
Get<ColliderComponent>().Collider;
            collider.Material->release();
            collider.Shape->release();
        }

        // destroy actor user data
        EntityID* owner = static_cast<EntityID*>
(comp.RigidBody.Actor->userData);
        EMPTY_DELETE(owner);

        // destroy actor instance
        comp.RigidBody.Actor->release();
    });
}

```

☞ **RenderSceneDepthMap():** This function simply renders the depth map for shadow mapping.

Listing 8.19: Application/Application.h

```

/* ... same as before ... */

EnttView<Entity, DirectLightComponent>([this] (auto light,
auto&)
{
    // light direction

```

```

        auto& lightDir = light.template Get<TransformComponent>()
().Transform.Rotation;

        // begin rendering
m_Context->Renderer->BeginShadowPass(lightDir);

        // render depth
EnttView<Entity, ModelComponent>([this, &lightDir] (auto
entity, auto& comp)
{
    auto& transform = entity.template
Get<TransformComponent>().Transform;
    m_Context->Renderer->DrawDepth(comp.Model, transform);
});

        // finalize frame
m_Context->Renderer->EndShadowPass();
});

```

☞ **CreatePhysicsActors()**: This function creates all the rigid body actors for simulation before the beginning of the main loop. It does so by calling the previously created functions for entities with and without colliders.

Listing 8.20: Application/Application. h

```

/* ... same as before ... */

// creates all actors and colliders
EMPY_INLINE void CreatePhysicsActors()
{
    EnttView<Entity, RigidBodyComponent>([this] (auto entity,
auto& comp)

```

```

    {
        m_Context->Physics->AddRigidBody(entity);
    });
}

```

- ☞ **CreateSkyboxEnvMaps()**: This function helps create the environment maps of the skybox.

Listing 8.21: Application/Application.h

```

/* ... same as before ... */

// creates maps for ambient lighting
EMPY_INLINE void CreateSkyboxEnvMaps()
{
    // load environment map
    auto skymap = std::make_shared<Texture2D>
("Resources/Textures/HDRs/Sky.hdr", true, true);

    // generate enviroment maps
    EnttView<Entity, SkyboxComponent>([this, &skymap] (auto
entity, auto& comp)
    {
        m_Context->Renderer->InitSkybox(comp.Sky, skymap,
2048);
    });
}

```

- ☞ **RenderSceneToFBO()**: This function renders all the entities in the frame buffer.

Listing 8.22: Application/Application.h

```

/* ... same as before ... */

// renders scene to the frame buffer
EMPY_INLINE void RenderSceneToFBO()
{
    // start new frame
    m_Context->Renderer->NewFrame();

    // set shader camera
    EnttView<Entity, CameraComponent>([this] (auto entity,
auto& comp)
    {
        auto& transform = entity.template
Get<TransformComponent>().Transform;
        m_Context->Renderer->SetCamera(comp.Camera,
transform);
    });

    // set shader point lights
    int32_t lightCounter = 0u;
    EnttView<Entity, PointLightComponent>([this,
&lightCounter] (auto entity, auto& comp)
    {
        auto& transform = entity.template
Get<TransformComponent>().Transform;
        m_Context->Renderer->SetPointLight(comp.Light,
transform, lightCounter);
        lightCounter++;
    });
    // set number of point lights
    m_Context->Renderer->SetPointLightCount(lightCounter);

    // set shader direct. lights
    lightCounter = 0;
    EnttView<Entity, DirectLightComponent>([this,

```

```

&lightCounter] (auto entity, auto& comp)
{
    auto& transform = entity.template
Get<TransformComponent>().Transform;
    m_Context->Renderer->SetDirectLight(comp.Light,
transform, lightCounter);
    lightCounter++;
});
// set number of direct lights
m_Context->Renderer->SetDirectLightCount(lightCounter);

// set shader spot lights
lightCounter = 0;
EnttView<Entity, SpotLightComponent>([this,
&lightCounter] (auto entity, auto& comp)
{
    auto& transform = entity.template
Get<TransformComponent>().Transform;
    m_Context->Renderer->SetSpotLight(comp.Light,
transform, lightCounter);
    lightCounter++;
});
// set number of spot lights
m_Context->Renderer->SetSpotLightCount(lightCounter);

// render models
EnttView<Entity, ModelComponent>([this] (auto entity,
auto& comp)
{
    // compute key frames
    if(entity.template Has<AnimatorComponent>())
    {
        auto& animator = entity.template
Get<AnimatorComponent>().Animator;
        auto& transforms = animator->Animate(m_Context-
>DeltaTime);
    }
}

```

```

        m_Context->Renderer->SetJoints(transforms);
    }

        auto& transform = entity.template
Get<TransformComponent>().Transform;
        m_Context->Renderer->Draw(comp.Model, comp.Material,
transform);
    });

    // render skybox
    EnttView<Entity, SkyboxComponent>([this] (auto entity,
auto& comp)
{
    auto& transform = entity.template
Get<TransformComponent>().Transform;
    m_Context->Renderer->DrawSkybox(comp.Sky, transform);
});

    // end frame
    m_Context->Renderer->EndFrame();
}

```

☞ **UpdateAppLayers()**: This function simply updates all application layers and renders the final scene to the on-screen frame buffer so it can be seen by the user.

Listing 8.23: Application/Application.h

```

/* ... same as before ... */

// updates all application layers
EMPY_INLINE void UpdateAppLayers()
{
    for(auto layer : m_Context->Layers)

```

```

    {
        layer->OnUpdate();
    }

    // show scene to screen
    m_Context->Renderer->ShowFrame();
}

```

☞ **CreateSceneEntities()**: This function helps create all entities. It starts by loading all the necessary assets and proceeds by creating the camera, the directional light, the skybox, and our animated model from earlier. In the subsequent part, we create a plane that is going to be a static actor upon which all dynamic actors will be falling to test the collision. We then proceed to create a couple of spheres and cube entities, each with both a rigid body and a collider component.

Listing 8.24: Application/Application. h

```

/* ... same as before ... */

// creates entities with components
EMPY_INLINE void CreateSceneEntities()
{
    // load models
    auto walking = std::make_shared<SkeletalModel>
("Resources/Models/Walking.fbx");
    auto sphereModel = std::make_shared<StaticModel>
("Resources/Models/sphere.fbx");
    auto cubeModel = std::make_shared<StaticModel>
("Resources/Models/cube.fbx");
}

```

```
// create scene camera
auto camera = CreateEntt<Entity>();
camera.Attach<TransformComponent>().Transform.Translate.z
= 20.0f;
camera.Attach<CameraComponent>();

// create directional light
auto light = CreateEntt<Entity>();
light.Attach<DirectLightComponent>().Light.Intensity =
1.0f;
auto& td = light.Attach<TransformComponent>().Transform;
td.Rotation = glm::vec3(0.0f, 2.0f, -1.0f);

// create skybox entity
auto skybox = CreateEntt<Entity>();
skybox.Attach<TransformComponent>();
skybox.Attach<SkyboxComponent>();

// create robot entity
auto robot = CreateEntt<Entity>();
robot.Attach<ModelComponent>().Model = walking;
auto& tr = robot.Attach<TransformComponent>().Transform;
tr.Translate = glm::vec3(0.0f, -10.0f, -10.0f);
tr.Scale = glm::vec3(0.1f);
robot.Attach<AnimatorComponent>().Animator = walking-
>GetAnimator();

// create plane entity (ground)
auto plane = CreateEntt<Entity>();
plane.Attach<RigidBodyComponent>().RigidBody.Type =
RigidBody3D::STATIC;
plane.Attach<ColliderComponent>().Collider.Type =
Collider3D::BOX;
plane.Attach<ModelComponent>().Model = cubeModel;
auto& tp = plane.Attach<TransformComponent>().Transform;
```

```
    tp.Scale = glm::vec3(100.0f, 1.0f, 100.0f);
    tp.Translate.z = -50.0f;
    tp.Translate.y = -15.0f;

    // create sphere rigid bodies
    for(uint32_t i = 0; i < 100; i++)
    {
        Entity sphere = CreateEntt<Entity>();
        sphere.Attach<RigidBodyComponent>().RigidBody.Type =
RigidBody3D::DYNAMIC;
        sphere.Attach<ColliderComponent>().Collider.Type =
Collider3D::SPHERE;
        sphere.Attach<ModelComponent>().Model = sphereModel;
        auto& ts = sphere.Attach<TransformComponent>();
        ts.Transform.Translate.y = i*10.0f;
        ts.Transform.Translate.z = -10.0f;
        ts.Transform.Translate.x = 3.0f;
        ts.Transform.Scale *= 5.0f;
    }

    // create cube rigid bodies
    for(uint32_t i = 0; i < 100; i++)
    {
        Entity cube = CreateEntt<Entity>();
        cube.Attach<RigidBodyComponent>().RigidBody.Type =
RigidBody3D::DYNAMIC;
        cube.Attach<ColliderComponent>().Collider.Type =
Collider3D::BOX;
        cube.Attach<ModelComponent>().Model = cubeModel;
        auto& tc = cube.Attach<TransformComponent>();
        tc.Transform.Translate.y = i*10.0f;
        tc.Transform.Translate.z = -15.0f;
        tc.Transform.Rotation.x = i*10.0f;
        tc.Transform.Scale *= 5.0f;
```

```
    }  
}
```

Running the Application

Note that Conan will take some time to download and compile the PhysX SDK. You should consider going for a cup of tea.

Compile and run your project to get a result similar to (Figure 8.5).



Figure 8.5: Rigid Body Simulation Result

We haven't even scratched the surface of what can be done with PhysX. We will add some additional stuff later, with the addition of scripting. I already mentioned the fact that PhysX has a pretty good Wiki [NVIDIA](#) that can help you navigate all

the intricacies of physics simulation with PhysX. Feel free to play around with it.

Code not working? clone the branch "physics-simulation" on the GitHub repository and grant execution rights to the "EmPyLinux.sh" script on Linux. <https://github.com/Madsycode/book-empy-engine>.

```
git clone -branch <branch-name> <repository-link>
```

9 Runtime Scripting

You will definitely enjoy what you have worked hard for — you will be happy, and things will go well for you.
(King David)

Runtime scripting in game engines involves the execution of scripts or code during the actual runtime of a game. This allows for dynamic and on-the-fly changes to the game's behavior without the need to recompile the entire codebase. In the context of game engines, scripting languages like C#, Lua, or Python are commonly used for this purpose. The primary advantage of runtime scripting is its flexibility in modifying game logic without interrupting gameplay. Game developers can implement features, tweak parameters, or even fix bugs without the need for a full recompilation. This is particularly useful in iterative development and for supporting modding communities. (Figure 9.1) depicts a general approach to runtime scripting.

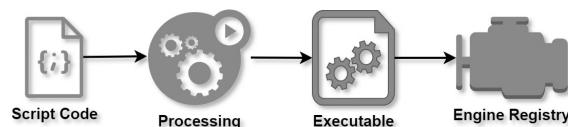


Figure 9.1: From Script to behaviour

In Unreal Engine and Unity, runtime scripting is accomplished through the integration of scripting languages and the use of respective APIs. Unreal Engine utilizes Blueprints as a visual scripting language, allowing developers to create gameplay elements through a node-based system. Blueprints facilitate script creation without traditional coding, making it accessible to designers and artists. For more traditional coding in Unreal Engine, C++ is the language of choice. Developers can use C++ to write game logic, and Unreal Engine provides a robust API for interacting with the engine's systems during runtime. In Unity, the scripting language used is C#. Unity allows developers to attach scripts to game objects, defining their behavior during runtime. The *MonoBehaviour* class in Unity provides essential functions, such as Start and Update, executed at specific points during the game's execution. Both engines support hot-reloading, enabling developers to make script changes and see the results immediately without restarting the entire game. This facilitates rapid iteration and experimentation during development.

9.1 Scripting with Lua

Lua [PUC-Rio](#) is a lightweight, high-level scripting language designed for embedded systems and general-purpose programming. Developed in Brazil in the early 1990s, Lua prioritizes simplicity, efficiency, and extensibility. Its name, "Lua," means "moon" in Portuguese, emphasizing its small

size and versatility. Primarily known for its use as an embedded scripting language in various applications, Lua has gained widespread adoption in the gaming industry, notably in game engines like Unreal Engine and Unity. Its syntax is clean and straightforward, resembling a mix of C and Python, making it accessible for developers with diverse programming backgrounds. Lua's key features include a powerful and flexible data description syntax called tables, first-class functions, and automatic memory management through garbage collection. It lacks certain complex features found in larger languages but compensates with a minimalistic design that allows for easy integration into existing systems.

9.1.1 Integrating Lua Dependencies

Instead of developing a custom API for Lua integration into our engine, we opt to utilize an existing and well-established library named "SOL." You can find a link to the online documentation here: [ThePhD](#). Sol2 facilitates scripting with Lua by providing a C++ wrapper that simplifies the integration of Lua scripts into C++ codebases. It serves as a bridge between C++ and Lua, offering a user-friendly interface for invoking Lua functions, manipulating Lua objects, and handling Lua variables within C++ programs. The core concept of Sol2 lies in its ability to seamlessly bind C++ functions and classes to Lua scripts. Developers can expose C++ functions and data to Lua, allowing Lua scripts

to interact with and utilize C++ functionality. This two-way communication enables the incorporation of Lua scripts for game logic, artificial intelligence, or other dynamic behaviors in C++-based applications.

You can see in the following code how "Lua" and "SOL" are added to the list of Conan requirements. These two libraries are header-only libraries.

Listing 9.1: Root/conanfile.txt

```
[requires]
glm/cci.20230113
stb/cci.20230920
opengl/system
spdlog/1.12.0
assimp/5.2.2
entt/3.12.2
physx/4.1.1
glfw/3.3.8
glew/2.2.0
sol2/3.3.1
lua/5.4.6

[generators]
cmake

[options]
physx:shared=True
glew:shared=False
glfw:shared=False
assimp:shared=False
```

9.1.2 Initialization and Configuration

In order to initialize Lua and SOL in our project, we need to create some additional files and folders to keep our code clean and organized. Add the files and folders depicted in (Figure 9.2) to your project directory tree. You can also proceed to include Lua and SOL in your project, as depicted in (Listing 9.2).

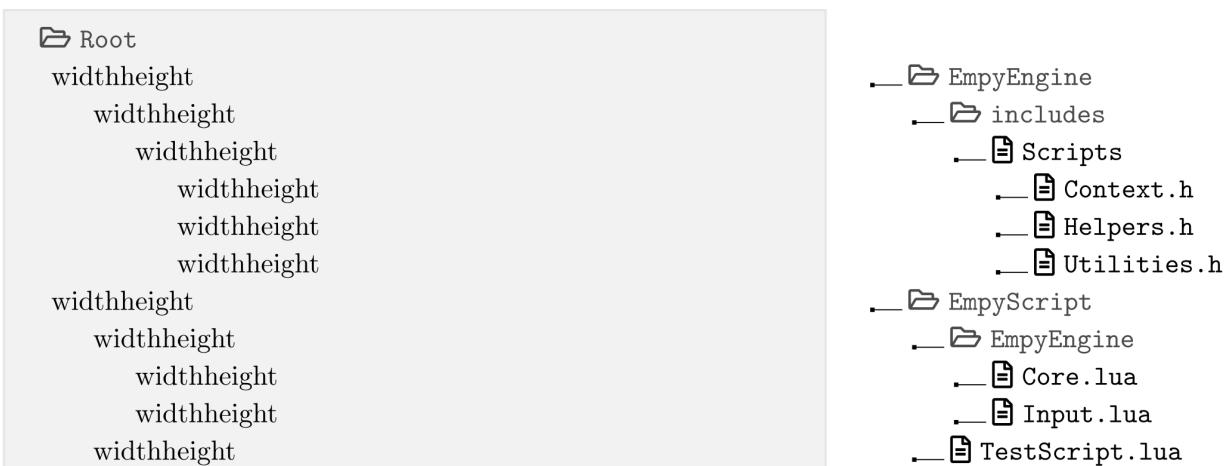


Figure 9.2: Physics Directory

Listing 9.2: Scripts/Helpers.h

```
#pragma once
extern "C"
{
    #include "lua.h"
    #include "lualib.h"
```

```
#include "lauxlib.h"
}
// include sol
#include <sol/sol.hpp>
#include "Common/Core.h"
```

The use of "extern \"C"" is necessary when including headers from a C library within a C++ program. It informs the compiler to use C-style linkage for the enclosed code block, preventing C++ name-mangling. The folder "**EmPyScript**" is where all our Lua scripts will be implemented. "Core.lua" and "Input.lua" will help us define the foundation for all scripts. "TestScript.lua" is going to be the first script to test all the features in the script context.

9.1.3 Script Handle

The code in (Listing 9.3) defines a structure called `Script`. This structure represents a script handle and includes various methods for handling events like window resizing, mouse interactions, collisions, key presses, updates, entity destruction, and script initialization. It utilizes a Lua table (`sol::table`) to encapsulate the interface between Lua and our engine. The `sol::table` type represents a Lua table, which is a fundamental data structure in Lua used to store key-value pairs.

Listing 9.3: Scripts/Utilities.
h

```
#pragma once
#include "Helpers.h"

namespace Empty
{
    struct Script
    {
        // script handle constructor
        EMPTY_INLINE Script(sol::table handle, const std::string& name) :
            m_Handle(handle), m_Name(name)
        { }

        // callback for window resize event
        EMPTY_INLINE void OnResize(int32_t width, int32_t height)
        {
            if(m_Handle.valid())
            {
                m_Handle["OnResize"](m_Handle, width, height);
            }
        }

        // callback for mouse down event
        EMPTY_INLINE void OnMouseDown(int32_t button)
        {
            if(m_Handle.valid())
            {
                m_Handle["OnMouseDown"](m_Handle, button);
            }
        }

        // callback for rigidbody collision
        EMPTY_INLINE void OnCollision(EntityID other)
```

```

    {

        if (m_Handle.valid())
        {
            m_Handle["OnCollision"] (m_Handle, other);
        }
    }

    // callback for key down event
EMPTY_INLINE void OnKeyDown(int32_t key)
{
    if (m_Handle.valid())
    {
        m_Handle["OnKeyDown"] (m_Handle, key);
    }
}

// callback to update script
EMPTY_INLINE void OnUpdate(float dt)
{
    if (m_Handle.valid())
    {
        m_Handle["OnUpdate"] (m_Handle, dt);
    }
}

// callback to destroy entity
EMPTY_INLINE void OnDestroy()
{
    if (m_Handle.valid())
    {
        m_Handle["OnDestroy"] (m_Handle);
    }
}

// callback sto tart script
EMPTY_INLINE void OnStart()

```

```

    {

        if (m_Handle.valid())
        {
            m_Handle["OnStart"] (m_Handle);
        }
    }

    // returns script name
EMPTY_INLINE const std::string& Name()
{
    return m_Name;
}

// checks if valid
EMPTY_INLINE bool Valid()
{
    return m_Handle.valid();
}

private:
    friend struct ScriptContext;
    sol::table m_Handle;
    std::string m_Name;
};

using LuaScript = std::unique_ptr<Script>;
}

```

In the functions of the script, the handle (`m_Handle`) is also passed as an argument to give the Lua script access to the script instance within itself for further actions. More on that in a bit. We additionally define a type `LuaScript` to encapsulate a unique pointer to the script. The function `OnCollision()` provides the entity identifier of the other

entity involved in the contact. This can be important in shooting games where you want to apply damage to the entity or maybe destroy it.

9.1.4 Script Context

Below (Listing 9.4) is the code of the script context class, which will help us handle our script by providing functions to load, create instances, attach to and entity, etc. As you can see, its functions are not implemented yet. Let us take a look at each function implementation.

Listing 9.4: Scripts/Context.h

```
#pragma once
#include "Window/Window.h"
#include "Auxiliaries/ECS.h"

namespace Empy
{
    struct ScriptContext
    {
        EMPY_INLINE ScriptContext(EntityRegistry* scene,
AppWindow* window)
        { /* ... */ }

        // creates instance of existing script
        EMPY_INLINE bool AttachScript(Entity& entity, const
std::string& name)
        { /* ... */ }

        // loads script into lua state
    };
}
```

```

    EMPTY_INLINE std::string LoadScript(const std::string&
filepath)
    { /* ... */ }

private:
    // registers functions to interact with entity
    EMPTY_INLINE void SetApiFunctions(EntityRegistry*
scene, AppWindow* window)
    { /* ... */ }

private:
    // lua state container
    sol::state m_Lua;
};

}

```

Context Constructor

The constructor helps us initialize all important components within the script context. The `open_libraries()` function is used to import the base Lua libraries. Subsequently, we load the two core scripts, "Input.lua" and "Core.lua," as modules into our Lua state. This ensures that everything we need to create a script instance is available. because we want to be able to request entity's components from our Lua scripts, we need to established an agreement between Lua and our application about the type identifier. That is the reason why the `TRANSFORM` value is added to the Lua state. this will be used within the Lua code tell which component to request from our ECS framework.

Listing 9.5: Scripts/Context.h

```
/* ... same as before ... */

EMPY_INLINE ScriptContext(EntityRegistry* scene, AppWindow*
window)
{
    // import lua libraries
    m_Lua.open_libraries(sol::lib::base);

    // load empy api module
    m_Lua.require_file("Inputs",
"Resources/Scripts/EmpyEngine/Input.lua");
    m_Lua.require_file("Core",
"Resources/Scripts/EmpyEngine/Core.lua");

    // runtime type identifiers
    m_Lua["TRANSFORM"] =TypeID<TransformComponent>();

    // add transform data type
    m_Lua.new_usertype<Transform3D>("Transform3D",
        "Translate", &Transform3D::Translate,
        "Rotation", &Transform3D::Rotation,
        "Scale", &Transform3D::Scale
    );

    // add vec3 data type
    m_Lua.new_usertype<glm::vec3>("Vec3",
        sol::constructors<glm::vec3(), glm::vec3(float,
float, float)>(),
        "x", &glm::vec3::x,
        "y", &glm::vec3::y,
        "z", &glm::vec3::z
    );
}
```

```
// register window inputs callbacks  
SetApiFunctions(scene, window);  
}
```

The subsequent code defines user types for *Transform* and *Vec3*. This makes it possible for us to use the transform and the vector3 type in Lua just as we would in C++. Finally, the function *SetApiFunctions()* will help us define all the function that will allow us to interact with the engine from Lua.

Loading the Script

In order to execute the code of a script in our engine, we first need to load it in the script context. This is done by the function *LoadScript()*. This function return the name of the script so that it can be used to create instances of the script.

Listing 9.6: Scripts/Context.h

```
/* ... same as before ... */  
  
// loads script into lua state  
EMPY_INLINE std::string LoadScript(const std::string&  
filepath)  
{  
    auto Initializer = m_Lua["Initializer"];  
  
    // check if modules are loaded  
    if(!Initializer.valid())  
    {
```

```

        EMPTY_ERROR("failed to load script! core not
initialized!");
        return "";
    }

    // check if script file exists
    std::filesystem::path path(filepath);
    if(!std::filesystem::exists(path))
    {
        EMPTY_ERROR("failed to load script! invalid file
path");
        return "";
    }

    // create script class handle
    auto scriptName = path.stem().string();
    m_Lua[scriptName] = Initializer();
    m_Lua.script_file(filepath);
    return scriptName;
}

```

This function starts by collecting an object called `Initialazer` from the Lua state. It was added when loading the "Core.lua" file in the constructor. We will see that in a bit. This object is simply a function that initializes the script by creating a "class" that represents it in the Lua state so that it can be instantiated every time a new script instance has to be created. We use the `std::filesystem` to retrieve the filename which is also supposed to be the same name used in the file for the script type. This is not a requirement of Lua but of our engine. Every script filename must also be the class name used within the script code. more on that in a

bit. Using the function `script_file()`, we are able to load the script code into our Lua state.

Script API Functions

As mentioned earlier, we need to be able to interact with the engine from Lua. This can be to retrieve, update, destroy objects. This means we have to implement some API functions to handle this for us. This is done in the function `SetApiFunctions()`. The name of the function added below are: `ApiGetTransform()`, `ApiMouseDown()`, `ApiApplyForce()`, `ApiKeyDown()`, and `ApiDestroy()`. What these functions do can easily be deduced from their names.

Listing 9.7: Scripts/Context. h

```
/* ... same as before ... */

// registers functions to interact with entity
EMPY_INLINE void SetApiFunctions(EntityRegistry* scene,
AppWindow* window)
{
    // api function to get entity transform
    m_Lua.set_function("ApiGetTransform", [this, scene]
(EntityID entity)
    {
        return std::ref(scene->get<TransformComponent>
(entity).Transform);
    });
}
```

```

    // api function to check if mouse down
    m_Lua.set_function("ApiMouseDown", [this, window]
(int32_t button)
{
    return window->IsMouse(button);
} );

    // api function to apply force on actor
    m_Lua.set_function("ApiApplyForce", [this, scene]
(EntityID entity, const glm::vec3& force)
{
    // return reference to transform
    if(scene->all_of<RigidBodyComponent>(entity))
    {
        auto& body = scene->get<RigidBodyComponent>
(entity).RigidBody;
        if(body.Type == body.DYNAMIC)
        {
            auto actor = static_cast<PxRigidDynamic*>
(body.Actor);
            actor->addForce(ToPxVec3(force),
PxForceMode::eACCELERATION);
            return true;
        }
        EMPLY_ERROR("trying to apply force to static
body!");
        return false;
    }
    EMPLY_ERROR("trying to apply force to non physics
entity!");
    return false;
} );

    // api function to check if key pressed
    m_Lua.set_function("ApiKeyDown", [this, window] (int32_t
key)

```

```

    {

        return window->IsKey(key);
    });

    // api function to destroy entity
    m_Lua.set_function("ApiDestroy", [this, scene] (EntityID
entity)
    {
        // return if entity is dead!
        if(scene->valid(entity) == false) { return; }

        // entity has rigidbody component
        if(scene->all_of<RigidBodyComponent>(entity))
        {
            // destroy actor user data
            auto& body = scene->get<RigidBodyComponent>
(entity).RigidBody;
            auto userData = static_cast<EntityID*>
(body.Actor->userData);
            body.Actor->release();
            EMPTY_DELETE(userData);
        }

        // destroy collider material & shape
        if(scene->all_of<ColliderComponent>(entity))
        {
            // destroy actor user data
            auto& collider = scene->get<ColliderComponent>
(entity).Collider;
            collider.Material->release();
            collider.Shape->release();
        }

        // entity has script component
        if(scene->all_of<ScriptComponent>(entity))
        {
    }
}

```

```

        // call on destroy function
        scene->get<ScriptComponent>(entity).Instance-
>OnDestroy();
    }

    // destroy entity from scene
    scene->destroy(entity);
}

```

Important to note in the `ApiDestroy()` is that we are also making sure that the rigid body actor, collider and material are destroyed. This keeps our physics simulation consistent when an entity is destroyed. If you don't do this, PhysX will think that the entity is still alive and thus compute its state, and handle its collisions. You can see how simple it is to add an API functions to the context. SOL encapsulates all the complexity in the function `set_function()`.

Attaching Script to Entity

The next step involves attaching the script to an entity so that it can be referenced in the application main loop and events. The function `AttachScript()` does that for us. Following is its implementation. This function starts by making sure that the script is loaded and present in the Lua state. Each script type as we will see, must have a constructor which helps creates instances of the script type.

If the constructor is present, then we create an instance passing the entity identifier as argument.

Listing 9.8: Scripts/Context. h

```
/* ... same as before ... */

// creates instance of existing script
EMPTY_INLINE bool AttachScript(Entity& entity, const
std::string& name)
{
    // check if handle is correct and has a constructor
    if (!m_Lua[name].valid() && !m_Lua[name]
["Constructor"].valid())
    {
        EMPTY_ERROR("failed to create script: invalid script
name!");
        return false;
    }

    // create instance script
    auto object = m_Lua[name]["Constructor"] (entity.ID());

    // check if obj is valid
    if(!object.valid())
    {
        // handle failure
        sol::error error = object;
        sol::call_status status = object.status();
        EMPTY_ERROR("failed to create script: {}",
error.what());
        return false;
    }
}
```

```

entity.Attach<ScriptComponent>().Instance =
std::make_unique<Script>(object, name);
return true;
}

```

This is important because we want to know who owns this specific instance of the script type. This is similar to what we did in PhysX when providing the entity identifier as the rigid actor's user data. The function proceeds to check whether the script instance is created properly and a unique pointer script handle is created from it and attached to the entity using a script component.

Here is the code of the script component:

Listing 9.9: Auxiliaries/ECS. h

```

/* ... same as before ... */

// include script utilities
#include "Scripts/Utilities.h"

// script component type
struct ScriptComponent
{
    EMPY_INLINE ScriptComponent(const ScriptComponent&) =
default;
    EMPY_INLINE ScriptComponent() = default;
    LuaScript Instance;
};

```

9.1.5 Introduction to Lua

Lua's syntax is minimalistic but powerful. It offers a range of features suitable for a wide array of applications. In Lua, programs are organized into blocks, and the language employs dynamic typing, allowing variables to change types during runtime. Lua uses a simple and flexible syntax that emphasizes readability and ease of use. Statements end with a semicolon, but it is often optional. Code blocks are denoted by the keywords "if," "else," "while," and "for," and indentation is not significant, making Lua forgiving in terms of formatting. Comments can be either single-line (using "--") or multi-line (enclosed by "--[[]]").

Listing 9.10: Lua Syntax

```
-- Single-line comment
--[[ Multi-line comment ]]

local x = 10      -- Variable declaration

if x > 5 then    -- Conditional statement
    print("x is greater than 5")
else
    print("x is less than or equal to 5")
end
```

Data Types

Lua is dynamically typed, meaning that variables do not have predefined types and can hold values of different types

during execution. The basic data types in Lua include nil, boolean, number, string, function, and table. Tables are a fundamental data structure in Lua, serving as arrays, dictionaries, or a combination of both.

Listing 9.11: Lua Data Types

```
local nilValue = nil          -- Nil type
local boolValue = true        -- Boolean type
local intValue = 42           -- Number type
local stringValue = "Hello"   -- String type
local functionValue = function() print("Function") end -- Function type
local tableValue = {1, 2, 3}   -- Table type
```

Functions

Functions play a crucial role in Lua, and they are first-class citizens in the language. Functions can be assigned to variables, passed as arguments to other functions, and returned as values. Anonymous functions, known as "closures," are also supported. Lua supports multiple return values from functions, contributing to its expressive and flexible nature.

Listing 9.12: Lua Functions

```
-- Named function
function add(a, b)
```

```

        return a + b
    end

-- Anonymous function (closure)
local multiply = function(x, y)
    return x * y
end

-- Functions as arguments
function applyOperation(operation, a, b)
    return operation(a, b)
end

print(add(3, 4))           -- 7
print(applyOperation(multiply, 2, 5)) -- 10

```

Meta-tables and Meta-methods

Lua introduces metatables, which allow the customization of tables and define how they behave in certain operations. Meta-methods are special functions associated with metatables, enabling operators like addition, subtraction, and comparison to be customized for user-defined types.

Listing 9.13:

Lua Metatables and Metamethods

```

-- Metatable example
local myTable = { value = 42 }

-- Define metatable
local metaTable = {
    __add = function(a, b)

```

```
        return a.value + b.value
    end
}

setmetatable(myTable, metaTable)

local otherTable = { value = 10 }
print(myTable + otherTable) -- 52 (customized addition)
```

These snippets provide a glimpse into the syntax, data types, functions, etc., in Lua. Keep in mind that Lua's documentation [PUC-Rio](#) has a lot more to say than what we can do here.

9.1.6 Implementing Script Modules

Let us now take a look at the Lua code we have been talking about. The first file we can look into is the "Core.lua" file.

Script Core Module

This file holds the core definition of what a script is as well as the means to creating instances. What we have done in this code is to create a sort of inheritance concept. We are establishing a scripting framework, consisting of a base script class `EmptyScript` and a derived script class `ScriptKlass`. The `EmptyScript` table serves as the base class, employing a metatable `EmptyScript_mt` to define inheritance. The `EmptyScript.Constructor()` function initializes instances of

the base class by creating a local table with the provided entity identifier. The `EmpyScript.Destroy()` function is designed to destroy entities using the previously introduced API function `ApiDestroy()`.

Listing 9.14: Root/EmpyScript/EmpyEngine/Core.lua

```
-- core script class
local EmpyScript = {}
local EmpyScript_mt = { __index = EmpyScript }

-- inits script base classe
function EmpyScript.Constructor(entity)
    local self = setmetatable({}, EmpyScript_mt)
    self.Entity = entity
    return self
end

-- destroy entity with id
function EmpyScript.Destroy(entity)
    ApiDestroy(entity)
end

-- inits script class
function Initializer()
    -- script class
    local ScriptKlass = {}
    local ScriptKlass_mt = { __index = ScriptKlass }
    setmetatable(ScriptKlass, { __index = EmpyScript })

    -- constructor
    function ScriptKlass.Constructor(entity)
        local obj = EmpyScript.Constructor(entity)
```

```

        self = setmetatable(obj, ScriptKlass_mt)
        return self
    end

    -- apply force to rigidbody
    function ScriptKlass:ApplyForce(force)
        ApiApplyForce(self.Entity, force)
    end

    -- destroy self
    function ScriptKlass:Destroy()
        ApiDestroy(self.Entity)
    end

    -- get data
    function ScriptKlass:Get(type)
        if type == TRANSFORM then
            return ApiGetTransform(self.Entity)
        end
        print("invalid type:", type);
        return {}
    end

    -- export class
    return ScriptKlass
end

-- export module
return EmptyScript

```

The `Initializer()` function as previously introduced in the script context class sets up the derived script class `ScriptKlass` which is a kind of template type for the actual script type. This class, represented by the `ScriptKlass` table,

inherits from the base class `EmpyScript` using a metatable `ScriptKlass_mt`. The `ScriptKlass.Constructor()` function initializes instances of the derived class by calling the constructor of the base class and further configuring the metatable. The `ScriptKlass` class introduces additional methods. The `ApplyForce()` method allows instances to apply forces to associated entities through the `ApiApplyForce()` function. The `Get()` method retrieves information based on the provided type, calling the appropriate API function such as `ApiGetTransform()` for the type `TRANSFORM`. In case of an invalid type, it prints an error message and returns an empty table. Finally, both the `ScriptKlass` and `EmpyScript` tables are exported for external use. This Lua script follows an object-oriented paradigm, facilitating entity-related operations through a flexible and extensible framework within our engine.

Script Input Module

As you know, we need to be able to use key and mouse inputs in the script to be able to implement things such as player and camera movements. This is encapsulated in the input module implemented in "Input.lua". Below is a code snippet for this module:

Listing 9.15: Root/EmpyScript/EmpyEngine/Input.lua

```
-- define module table
local EmPyInput = {}

-- check mouse input
function EmPyInput.IsMouse(button)
    return ApiMouseDown(button)
end

-- check key input
function EmPyInput.IsKey(key)
    return ApiKeyDown(key)
end

-- mouse input codes
EmPyInput.MOUSE_LEFT      = 0
EmPyInput.MOUSE_MIDDLE    = 1
EmPyInput.MOUSE_RIGHT     = 2
EmPyInput.MOUSE_BUTTON_4   = 3
EmPyInput.MOUSE_BUTTON_5   = 4
EmPyInput.MOUSE_BUTTON_6   = 5
EmPyInput.MOUSE_BUTTON_7   = 6
EmPyInput.MOUSE_BUTTON_8   = 7

-- key input codes
EmPyInput.KEY_SPACE        = 32
EmPyInput.KEY_APOSTROPHE   = 39
EmPyInput.KEY_COMMA        = 44
EmPyInput.KEY_MINUS         = 45
EmPyInput.KEY_PERIOD        = 46
EmPyInput.KEY_SLASH         = 47
EmPyInput.KEY_0              = 48
EmPyInput.KEY_1              = 49
EmPyInput.KEY_2              = 50
EmPyInput.KEY_3              = 51
EmPyInput.KEY_4              = 52
```

EmPyInput.KEY_5	= 53
EmPyInput.KEY_6	= 54
EmPyInput.KEY_7	= 55
EmPyInput.KEY_8	= 56
EmPyInput.KEY_9	= 57
EmPyInput.KEY_SEMICOLON	= 59
EmPyInput.KEY_EQUAL	= 61
EmPyInput.KEY_A	= 65
EmPyInput.KEY_B	= 66
EmPyInput.KEY_C	= 67
EmPyInput.KEY_D	= 68
EmPyInput.KEY_E	= 69
EmPyInput.KEY_F	= 70
EmPyInput.KEY_G	= 71
EmPyInput.KEY_H	= 72
EmPyInput.KEY_I	= 73
EmPyInput.KEY_J	= 74
EmPyInput.KEY_K	= 75
EmPyInput.KEY_L	= 76
EmPyInput.KEY_M	= 77
EmPyInput.KEY_N	= 78
EmPyInput.KEY_O	= 79
EmPyInput.KEY_P	= 80
EmPyInput.KEY_Q	= 81
EmPyInput.KEY_R	= 82
EmPyInput.KEY_S	= 83
EmPyInput.KEY_T	= 84
EmPyInput.KEY_U	= 85
EmPyInput.KEY_V	= 86
EmPyInput.KEY_W	= 87
EmPyInput.KEY_X	= 88
EmPyInput.KEY_Y	= 89
EmPyInput.KEY_Z	= 90
EmPyInput.KEY_LEFT_BRACKET	= 91
EmPyInput.KEY_BACKSLASH	= 92
EmPyInput.KEY_RIGHT_BRACKET	= 93

```
EmpyInput.KEY_GRAVE_ACCENT      = 96
EmpyInput.KEY_WORLD_1           = 161
EmpyInput.KEY_WORLD_2           = 162

-- Fion keys *--
EmpyInput.KEY_ESCAPE            = 256
EmpyInput.KEY_ENTER             = 257
EmpyInput.KEY_TAB               = 258
EmpyInput.KEY_BACKSPACE         = 259
EmpyInput.KEY_INSERT            = 260
EmpyInput.KEY_DELETE            = 261
EmpyInput.KEY_RIGHT             = 262
EmpyInput.KEY_LEFT              = 263
EmpyInput.KEY_DOWN              = 264
EmpyInput.KEY_UP                = 265
EmpyInput.KEY_PAGE_UP           = 266
EmpyInput.KEY_PAGE_DOWN         = 267
EmpyInput.KEY_HOME              = 268
EmpyInput.KEY_END               = 269
EmpyInput.KEY_CAPS_LOCK         = 280
EmpyInput.KEY_SCROLL_LOCK       = 281
EmpyInput.KEY_NUM_LOCK          = 282
EmpyInput.KEY_PRINT_SCREEN      = 283
EmpyInput.KEY_PAUSE              = 284
EmpyInput.KEY_F1                 = 290
EmpyInput.KEY_F2                 = 291
EmpyInput.KEY_F3                 = 292
EmpyInput.KEY_F4                 = 293
EmpyInput.KEY_F5                 = 294
EmpyInput.KEY_F6                 = 295
EmpyInput.KEY_F7                 = 296
EmpyInput.KEY_F8                 = 297
EmpyInput.KEY_F9                 = 298
EmpyInput.KEY_F10                = 299
EmpyInput.KEY_F11                = 300
EmpyInput.KEY_F12                = 301
```

EmPyInput.KEY_F13	= 302
EmPyInput.KEY_F14	= 303
EmPyInput.KEY_F15	= 304
EmPyInput.KEY_F16	= 305
EmPyInput.KEY_F17	= 306
EmPyInput.KEY_F18	= 307
EmPyInput.KEY_F19	= 308
EmPyInput.KEY_F20	= 309
EmPyInput.KEY_F21	= 310
EmPyInput.KEY_F22	= 311
EmPyInput.KEY_F23	= 312
EmPyInput.KEY_F24	= 313
EmPyInput.KEY_F25	= 314
EmPyInput.KEY_KP_0	= 320
EmPyInput.KEY_KP_1	= 321
EmPyInput.KEY_KP_2	= 322
EmPyInput.KEY_KP_3	= 323
EmPyInput.KEY_KP_4	= 324
EmPyInput.KEY_KP_5	= 325
EmPyInput.KEY_KP_6	= 326
EmPyInput.KEY_KP_7	= 327
EmPyInput.KEY_KP_8	= 328
EmPyInput.KEY_KP_9	= 329
EmPyInput.KEY_KP_DECIMAL	= 330
EmPyInput.KEY_KP_DIVIDE	= 331
EmPyInput.KEY_KP_MULTIPLY	= 332
EmPyInput.KEY_KP_SUBTRACT	= 333
EmPyInput.KEY_KP_ADD	= 334
EmPyInput.KEY_KP_ENTER	= 335
EmPyInput.KEY_KP_EQUAL	= 336
EmPyInput.KEY_LEFT_SHIFT	= 340
EmPyInput.KEY_LEFT_CONTROL	= 341
EmPyInput.KEY_LEFT_ALT	= 342
EmPyInput.KEY_LEFT_SUPER	= 343
EmPyInput.KEY_RIGHT_SHIFT	= 344
EmPyInput.KEY_RIGHT_CONTROL	= 345

```
EmpyInput.KEY_RIGHT_ALT      = 346
EmpyInput.KEY_RIGHT_SUPER    = 347
EmpyInput.KEY_MENU           = 348

-- export module
return EmpyInput
```

This module is quite simple. We start by defining the module table, which will be exported for external use, followed by the functions to query whether the mouse button or a key is down or pressed. In the subsequent part, we simply define all the input codes for both the mouse and the keyboard. The value of these codes came from GLFW. Each mouse button or key on the keyboard has a unique code that can be used to identify it and know when an event was triggered by it.

These two files are all we really need to get started with scripting in our engine. We could, of course, add additional modules for graphics, physics, etc., but I will let you do that yourself. We can now move forward to the test script to run some simple tests.

Test Script Code

Below is the code for the `TestScript.lua` file. The first thing you will notice is that the name of the class is the same as the filename. Again, this is not required, but we want it to be like this, as this will simply be the loading process of scripts.

This script simply implements the functions `OnStart()` and `OnUpdate()` found in the structure `Script`.

Listing 9.16: Root/EmpyScript/TestScript.lu a

```
-- call once to init script
function TestScript.OnStart(self)
    -- retrieve entity transformation
    self.Transform = self:Get(TRANSFORM)

    print("script started!")
end

-- update every frame
function TestScript.OnUpdate(self, dt)
    -- scale object every frame
    self.Transform.Scale.y = self.Transform.Scale.y + dt;

    print("script update dt: ", dt)
end
```

Each function simply prints out a text on the console using the `print()` function. Let us integrate script handling into the application context.

9.1.7 Integration with Application

We now have to integrate everything we have created so far into the application context to see it in action. One of the first things we want to do is make sure that every time the code is compiled, if the script code is modified, it is copied to

the build folder so that our engine can always use the new version of the script.

Automating Script Copy

This is done using CMake, as depicted in the following code. This code simply defines a post-build command to copy everything in the "EmPyScript" folder into the build directory.

Listing 9.17: EmPyEngine/CMakeLists.txt

```
# copy script directory
if(EXISTS ${CMAKE_SOURCE_DIR}/EmPyScript)
    add_custom_command(TARGET ${PROJECT_NAME} POST_BUILD
COMMAND
    ${CMAKE_COMMAND} -E copy_directory
    ${CMAKE_SOURCE_DIR}/EmPyScript
    ${EXECUTABLE_OUTPUT_PATH}/Resources/Scripts
)
else()
    message(WARNING "[WARNING] no script directory!")
endif()
```

Extending Application Context

We need to create an instance of the script context in the application context to make sure that we are able to load and attach scripts to entities. Extend your application context as depicted in the following code:

Listing 9.18: Application/Context.h

```
#pragma once
#include "Scripts/Context.h"
#include "Physics/Context.h"
#include "Graphics/Renderer.h"

namespace EmPy
{
    // forward declaration
    struct AppInterface;

    // application context
    struct ApplicationContext
    {
        EMPI_INLINE ApplicationContext()
        {
            Window = std::make_unique<AppWindow>(&Dispatcher,
1280, 720, "EmPy Engine");
            Scripts = std::make_unique<ScriptContext>(&Scene,
Window.get());
            Renderer = std::make_unique<GraphicsRenderer>
(1280, 720);
            Physics = std::make_unique<PhysicsContext>();
            DeltaTime = 0.0;

            // create and init scripts context
        }

        EMPI_INLINE ~ApplicationContext()
        {
            for(auto layer : Layers)
            {
                EMPI_DELETE(layer);
            }
        }
    };
}
```

```

        }

    }

    std::unique_ptr<GraphicsRenderer> Renderer;
    std::unique_ptr<PhysicsContext> Physics;
    std::unique_ptr<ScriptContext> Scripts; // <-- script
context
    std::vector<AppInterface*> Layers;
    std::unique_ptr<AppWindow> Window;
    EventDispatcher Dispatcher;
    EntityRegistry Scene;
    double DeltaTime;
};

}

```

Creating Entity with Script

Now that we have the script context instance created and ready to use, we need to load the test script, create some entities, and attach the script to them. The following code describes how this is done.

Listing 9.19: Application/Application.h

```

/* ... same as before ... */

// creates entities with components
EMPY_INLINE void CreateSceneEntities()
{
    // load models
    auto walking = std::make_shared<SkeletalModel>
("Resources/Models/Walking.fbx");

```

```

        auto sphereModel = std::make_shared<StaticModel>
("Resources/Models/sphere.fbx");
        auto cubeModel = std::make_shared<StaticModel>
("Resources/Models/cube.fbx");

        // create scene camera
        auto camera = CreateEntt<Entity>();
        camera.Attach<TransformComponent>().Transform.Translate.z
= 20.0f;
        camera.Attach<CameraComponent>();

        // create skybox entity
        auto skybox = CreateEntt<Entity>();
        skybox.Attach<TransformComponent>();
        skybox.Attach<SkyboxComponent>();

        // create directional light
        auto light = CreateEntt<Entity>();
        light.Attach<DirectLightComponent>().Light.Intensity =
1.0f;
        auto& td = light.Attach<TransformComponent>().Transform;
        td.Rotation = glm::vec3(0.0f, 1.0f, -1.0f);

        // create robot entity
        auto robot = CreateEntt<Entity>();
        robot.Attach<ModelComponent>().Model = walking;
        robot.Attach<AnimatorComponent>().Animator = walking-
>GetAnimator();
        auto& tr = robot.Attach<TransformComponent>().Transform;
        tr.Translate = glm::vec3(0.0f, -14.99f, -15.0f);
        tr.Scale = glm::vec3(0.1f);

        // ----- added!

        // load script file
        auto scriptName = m_Context->Scripts->

```

```

LoadScript("Resources/Scripts/TestScript.lua");

    // create plane entity (ground)
    auto plane = CreateEntt<Entity>();
    plane.Attach<RigidBodyComponent>().RigidBody.Type =
RigidBody3D::STATIC;
    plane.Attach<ColliderComponent>().Collider.Type =
Collider3D::BOX;
    plane.Attach<ModelComponent>().Model = cubeModel;
    auto& tp = plane.Attach<TransformComponent>().Transform;
    tp.Translate = glm::vec3(0.0f, -15.0f, -50.0f);
    tp.Scale = glm::vec3(100.0f, 1.0f, 100.0f);

    // create cube entities
    for(int i = 0; i < 5; i++)
    {
        auto cube = CreateEntt<Entity>();
        cube.Attach<ModelComponent>().Model = cubeModel;

        // attach script to entity
        m_Context->Scripts->AttachScript(cube, scriptName);

        cube.Attach<ColliderComponent>().Collider.Type =
Collider3D::BOX;
        cube.Attach<RigidBodyComponent>().RigidBody.Type =
RigidBody3D::DYNAMIC;
        auto& tc = cube.Attach<TransformComponent>()
        .Transform;
        tc.Translate = glm::vec3(0.0f, 6.0f *i, -10.0f);
        tc.Scale *= 5.0f;
    }
}

```

Starting and Updating Scripts

Now that we have some entities with the script component, we need to make sure that the functions we have implemented are called accordingly. Each of the following functions uses a view on all entities with a script component to call both functions: `OnStart()` and `OnUpdate()`. Additionally, the update function also takes the delta time as an argument.

Listing 9.20: Application/Application.
h

```
/* ... same as before ... */

// calls script instances on-start
EMPY_INLINE void StartScriptInstances()
{
    EnttView<Entity, ScriptComponent>([this] (auto entity,
auto& script)
    {
        if(script.Instance)
        {
            script.Instance->OnStart();
        }
    });
}

// calls script instances on-update
EMPY_INLINE void UpdateScriptInstances()
{
    EnttView<Entity, ScriptComponent>([this] (auto entity,
auto& script)
    {
        if(script.Instance)
        {
```

```

        script.Instance->OnUpdate(m_Context->DeltaTime);
    }
} );
}

```

With these functions, you can update your application constructor and main loop as follows to ensure seamless integration.

Listing 9.21: Application/Application.h

```

/* ... same as before ... */

// creates application context
EMPTY_INLINE Application()
{
    m_LayerID =TypeID<Application>();
    m_Context = new AppContext();

    // register event callbacks
    RegisterEventCallbacks();
    // create scene entities
    CreateSceneEntities();
    // create physics actors
    CreatePhysicsActors();
    // create environm. maps
    CreateSkyboxEnvMaps();
    // start script instances
    StartScriptInstances(); // <-- on-start()
}

// runs application main loop
EMPTY_INLINE void RunContext()
{

```

```
// application main loop
while (m_Context->Window->PollEvents())
{
    // compute and update delta time value
    ComputeFrameDeltaTime();

    // call update func script instances
    UpdateScriptInstances(); // <-- on-update()

    // run and fetch physics simulation
    RunPhysicsSimulation();

    // render scene shadow map
    RenderSceneDepthMap();

    // render scene to buffer
    RenderSceneToFBO();

    // update all layers
    UpdateAppLayers();
}

}
```

If you compile and run your code, you will be able to see the following output in the terminal:

```
script started!
script started!
script started!
script started!
script started!
script update dt: 0.016537100076675
script update dt: 0.016537100076675
script update dt: 0.016537100076675
script update dt: 0.016537100076675
```

Applying Force to Rigid Body

We now want to continue down this road to test the input API. The following code shows how this can be done.

Listing 9.22: Root/EmpyScript/TestScript.lu
a

```
-- update every frame
function TestScript.OnUpdate(self, dt)
    -- move entity to the left
    if Inputs.IsKey(Inputs.KEY_A) then
        self:ApplyForce(Vec3.new(-100.0, 0.0, 0.0));
    end

    -- move entity to the right
    if Inputs.IsKey(Inputs.KEY_D) then
        self:ApplyForce(Vec3.new(100.0, 0.0, 0.0));
    end
end
```

This function checks if the keys "A" and "D" are currently pressed and applies a force to the actor accordingly. If you run this code, you will be able to move your boxes using these keys.

Extending Event Callbacks

In the previous chapter, we mentioned the fact that we would handle collision when scripting will be available. As stated, we want the script to be able to know who it collided

with so it can perform additional actions on it. Go back to you application and extend the function

RegisterEventCallbacks() as depicted in the following code:

Listing 9.23: Application/Application. h

```
/* ... same as before ... */

EMPTY_INLINE void RegisterEventCallbacks()
{
    // set physics event callback
    m_Context->Physics->SetEventCallback([this] (auto e)
    {
        // do not report if invalid entities
        if(!m_Context->Scene.valid(e.Entity1) ||
           !m_Context->Scene.valid(e.Entity2))
        { return; }

        PostTask([this, e]
        {
            auto entity1 = ToEntt<Entity>(e.Entity1);
            auto entity2 = ToEntt<Entity>(e.Entity2);

            // entity 1 call script on-collision
            if(entity1.template Has<ScriptComponent>())
            {
                entity1.template Get<ScriptComponent>().
                    Instance->OnCollision(e.Entity2);
            }

            // entity 2 call script on-collision
            if(entity2.template Has<ScriptComponent>())
            {
                entity2.template Get<ScriptComponent>().
```

```

        Instance->OnCollision(e.Entity1);
    }
}

} );

// attach window resize event callback
AttachCallback<WindowResizeEvent>([this] (auto e)
{
    // resire renderer frame buffer
    m_Context->Renderer->Resize(e.Width, e.Height);

    // call scripts resize function
    EnttView<Entity, ScriptComponent>([e] (auto entity,
auto& script)
{
    if(script.Instance)
    {
        script.Instance->OnResize(e.Width, e.Height);
    }
}) ;

} );

// register mouse down callback
AttachCallback<MouseDownEvent>([this] (auto e)
{
    // call scripts mouse down callback
    EnttView<Entity, ScriptComponent>([e] (auto entity,
auto& script)
{
    if(script.Instance)
    {
        script.Instance->OnMouseDown(e.Button);
    }
}) ;

} );

```

```

// register key down callback
AttachCallback<KeyPressEvent>([this] (auto e)
{
    // call scripts mouse down callback
    EnttView<Entity, ScriptComponent>([e] (auto entity,
auto& script)
    {
        if(script.Instance)
        {
            script.Instance->OnKeyDown(e.Key);
        }
    });
});
}

```

You can see how the collision event is captured and provided to the scripts using the `OnCollision()` function. We first check if the entity has a script and proceed by calling the collision function, which provides the identifier of the other entity involved in the collision. Additionally, we have added callback functions to handle key and mouse events.

Listing 9.24: Root/EmptyScript/TestScript.lu a

```

-- ...

-- call when colliding
function TestScript.OnCollision(self, other)
    print("collision:", self.Entity, "->", other)
    self:Destroy(other);
end

-- call when mouse is down

```

```

function TestScript.OnMouseDown(self, button)
    print("mousedown: ", button)
end

-- call when key down
function TestScript.OnKeyDown(self, key)
    print("keydown: ", key)
end

-- call when entity is destroyed
function TestScript.OnDestroy(self)
    print("destroyed:", self.Entity)
end

```

You can add the `OnCollision()` function to your script to destroy the other entity involved in the collision as depicted in the code above. You will also see additional functions to capture key and mouse events as well as the `OnDestroy()` function which is called when an entity with script is going to be destroyed.

Here is the entire code found in the "**TestScript.lua**" file

Listing 9.25: Root/EmpyScript/TestScript.lua

```

-- call once to init script
function TestScript.OnStart(self)
    self.Transform = self:Get(TRANSFORM)
    print("script started!")
end

-- update every frame
function TestScript.OnUpdate(self, dt)

```

```
-- move entity to the left
if Inputs.IsKey(Inputs.KEY_A) then
    self:ApplyForce(Vec3.new(-100.0, 0.0, 0.0));
end

-- move entity to the right
if Inputs.IsKey(Inputs.KEY_D) then
    self:ApplyForce(Vec3.new(100.0, 0.0, 0.0));
end

-- scale object every frame
self.Transform.Scale.y = self.Transform.Scale.y + dt;
end

-- call when key down
function TestScript.OnResize(self, width, height)
    print("resize: ", width, height)
end

-- call when mouse is down
function TestScript.OnMouseDown(self, button)
    print("mousedown: ", button)
end

-- call when colliding
function TestScript.OnCollision(self, other)
    print("collision:", self.Entity, "->", other)
    self:Destroy(other);
end

-- call when key down
function TestScript.OnKeyDown(self, key)
    print("keydown: ", key)
end

-- call when entity is destroyed
```

```
function TestScript.OnDestroy(self)
    print("destroyed:", self.Entity)
end
```

You should feel free to experiment with this framework, adding new features and refining the ones that already exist. We could add more features to this book, but that would be too much for us to handle. You should have the foundation in place to take this to the next level.

Code not working? clone the branch "runtime-scripting" on the GitHub repository and grant execution rights to the "EmPyLinux.sh" script on Linux. <https://github.com/Madsycode/book-empy-engine>.

```
git clone -branch <branch-name> <repository-link>
```

Part IV

Assets & Serialization

10 Asset Management

The hardworking farmer should be the first to receive a share of the crops. (Timothy)

Asset management in game engines involves the systematic organization, storage, retrieval, and manipulation of digital assets used in game development. Digital assets include 3D models, textures, audio files, animations, and other resources essential for creating a game. The primary goal of asset management is to efficiently handle these resources throughout the game development life cycle. In a game engine, assets are often stored in a structured manner, and a database or file system keeps track of their locations and dependencies. This allows developers to easily reference and incorporate assets into the game without manual intervention.

10.1 Assets Definition

We are now going to create each asset type. This is done in the header file depicted in (Listing 10.1) below. This file defines the base asset type, which encompasses shared information for all asset types.

Listing 10.1: Auxiliaries/Assets.
h

```
#pragma once

#include "Scripts/Utilities.h"
#include "Physics/Utilities.h"
#include "Graphics/Models/Model.h"
#include "Graphics/Textures/Texture.h"

namespace EmPy
{
    // define the AssetID type as a 64-bit unsigned integer
    using AssetID = uint64_t;
    const AssetID EMPTY_ASSET = 0u;

    // asset type
    enum class AssetType : uint8_t
    {
        UNKNOWN = 0,
        MATERIAL,
        TEXTURE,
        SKYBOX,
        SCRIPT,
        SCENE,
        MODEL,
    };

    // define the base Asset structure
    struct Asset
    {
        // generate unique asset id
        AssetID UID = EMPTY_ASSET;

        // file path of asset
        std::string Source;

        // name of the asset
        std::string Name;
    };
}
```

```
// asset type
AssetType Type;
};

}
```

All subsequent asset types will inherit from this to implement specific properties. You will note how each asset is supposed to have a unique 64-bit identifier. This will be used to universally identify the asset. This identifier will also provide an interface to interact with assets from runtime scripts. All we need to do is implement interface functions to set specific asset properties in Lua.

10.1.1 Material Asset

The first material type we will create is the material asset. This asset encapsulates all that is needed to store information about a material.

Listing 10.2: Auxiliaries/Assets. h

```
struct MaterialAsset : Asset
{
    AssetID RoughnessMap = EMPTY_ASSET;
    AssetID OcclusionMap = EMPTY_ASSET;
    AssetID MetallicMap = EMPTY_ASSET;
    AssetID EmissiveMap = EMPTY_ASSET;
    AssetID AlbedoMap = EMPTY_ASSET;
    AssetID NormalMap = EMPTY_ASSET;
```

```
    Material Data;  
};
```

You will note how this asset depends on other assets. Since the material has textures, we have to provide a way to know which texture it is without holding the data itself. We have also added a member `Data` which holds an instance of the material.

10.1.2 Texture Asset

The next asset type is a texture. The following code snippet shows how it is defined.

Listing 10.3: Auxiliaries/Assets.h

```
struct TextureAsset : Asset  
{  
    bool IsHDR = false;  
    bool FlipV = true;  
    Texture2D Data;  
};
```

10.1.3 Skybox Asset

Here, the next asset type is a skybox asset. The following code snippet shows how it is defined.

Listing 10.4: Auxiliaries/Assets.h

```

struct SkyboxAsset : Asset
{
    int32_t Size = 2048;
    bool IsHDR = true;
    bool FlipV = true;
    Texture2D EnvMap;
    Skybox Data;
};

```

10.1.4 Model, Script, Scene, Asset

Following that same trend, we can define all the other asset types as shown below.

Listing 10.5: Auxiliaries/Assets. h

```

struct ModelAsset : Asset
{
    bool HasJoints = false;
    Model3D Data;
};

struct ScriptAsset : Asset
{
    /* ... */
};

struct SceneAsset : Asset
{
    /* ... */
};

// custom type definition

```

```
using SharedAsset = std::shared_ptr<Asset>;
using AssetMap = std::unordered_map<AssetID, SharedAsset>;
```

You should also note the type definitions that will help us interact with these assets in the registry.

10.2 Asset Registry

We are then required to create an asset registry to manage these different asset types. The asset registry will help load, store, and provide these assets when requested.

10.2.1 Registry Class

The following code snippet ([Listing 10.6](#)) describes the first iteration of the asset registry class implementation. Starting in the member variable section, you will note how the assets are all stored in a map of maps. The advantage of using maps rather than a contiguous container such as a `std::vector` is that if we were to retrieve an asset from a vector, we would have to potentially search through the entire array to find the specific asset. Knowing the asset identifier, we can simply retrieve the value from the map at a faster rate.

Listing 10.6: Auxiliaries/Assets. h

```
// asset registry to manage the addition and retrieval of
assets
```

```

struct AssetRegistry
{
    EMPTY_INLINE AssetRegistry()
    {
        // add default asset for each type
        AddEmpty<MaterialAsset>();
        AddEmpty<TextureAsset>();
        AddEmpty<SkyboxAsset>();
        AddEmpty<ScriptAsset>();
        AddEmpty<ModelAsset>();
        AddEmpty<SceneAsset>();
    }

    // retrieve asset based on its uid
    template <typename T>
    EMPTY_INLINE T& Get(AssetID uid)
    {
        const uint32_t type =TypeID<T>();
        if (m_Registry[type].count(uid))
            return (T&) (*m_Registry[type][uid]);
        return (T&) (*m_Registry[type][EMPTY_ASSET]);
    }

    // helps loop through all assets
    template <typename F>
    EMPTY_INLINE void View(F&& func)
    {
        for(auto& [_, assetMap] : m_Registry)
        {
            for(auto& [uid, asset] : assetMap)
            {
                if(uid != EMPTY_ASSET)
                {
                    func(asset.get());
                }
            }
        }
    }
}

```

```
        }

    }

    // return collection of asset
    template <typename T>
    EMPY_INLINE auto& GetMap()
    {
        return m_Registry[TypeID<T>()];
    }

    EMPY_INLINE void Clear()
    {
        m_Registry.clear();
    }

private:
    // adds a new asset to the registry
    template <typename T>
    EMPY_INLINE void Add(
        AssetID uid,
        const std::string& source,
        std::shared_ptr<T>& asset
    )
    {
        asset->UID = uid;
        asset->Source = source;
        std::filesystem::path path(source);
        asset->Name = path.stem().string();
        m_Registry[TypeID<T>()][asset->UID] = asset;
    }

    template <typename T>
    EMPY_INLINE void AddEmpty()
    {
        m_Registry[TypeID<T>()][EMPTY_ASSET] =
std::make_shared<T>();
```

```

    }

private:
    std::unordered_map<uint32_t, AssetMap> m_Registry;
};

```

To make sure we never retrieve a `nullptr` asset, we are adding empty assets for each type in the constructor. Whenever the `Get()` function is called with the asset identifier, if the asset is not found, the registry will return the empty asset, which is a kind of default asset. The `View()` function, as you might already know from ECS, helps loop through all assets available in the registry.

10.2.2 Adding Assets

The `Add()` function in the code in (Listing 10.6) as you can see, takes an instance of an asset to store in the targeted storage. Both the name and the file path of the asset are derived from the provided source argument. Let us now implement specific functions to add each asset type. This will be done in additional functions, as depicted below.

Listing 10.7: Auxiliaries/Assets. h

```

/* ... same as before ... */

struct AssetRegistry
{
    EMPI_INLINE auto AddSkybox(AssetID uid, const

```

```

std::string& source, int32_t size, bool isHDR = true, bool
flipV = true)
{
    auto asset = std::make_shared<SkyboxAsset>();
    asset->EnvMap.Load(source, isHDR, flipV);
    asset->Type = AssetType::SKYBOX;
    asset->IsHDR = isHDR;
    asset->FlipV = flipV;
    asset->Size = size;
    Add(uid, source, asset);
    return asset;
}

/* ... same as before ... */
};

```

The first function we will implement is the function to add a skybox asset. You can see how the asset type, properties, and data are all set before the `Add()` does the rest. This process is done for all asset types, as depicted in the following code.

Listing 10.8: Auxiliaries/Assets. h

```

/* ... same as before ... */

EMPY_INLINE auto AddTexture(AssetID uid, const std::string&
source, bool isHDR = false, bool flipV = true)
{
    auto asset = std::make_shared<TextureAsset>();
    asset->Data.Load(source, isHDR, flipV);
    asset->Type = AssetType::TEXTURE;
    asset->FlipV = flipV;
}

```

```

        asset->IsHDR = isHDR;
        Add(uid, source, asset);
        return asset;
    }

EMPTY_INLINE auto AddModel(AssetID uid, const std::string&
source, bool hasJoints = false)
{
    auto asset = std::make_shared<ModelAsset>();
    asset->HasJoints = hasJoints;
    asset->Type = AssetType::MODEL;

    // load model
    if(hasJoints)
        asset->Data = std::make_shared<SkeletalModel>
(source);
    else
        asset->Data = std::make_shared<StaticModel>(source);

    Add(uid, source, asset);
    return asset;
}

EMPTY_INLINE auto AddMaterial(AssetID uid, const std::string&
name)
{
    auto asset = std::make_shared<MaterialAsset>();
    asset->Type = AssetType::MATERIAL;
    Add(uid, name, asset);
    return asset;
}

EMPTY_INLINE auto AddScript(AssetID uid, const std::string&
source)
{
    auto asset = std::make_shared<ScriptAsset>();

```

```

    asset->Type = AssetType::SCRIPT;
    Add(uid, source, asset);
    return asset;
}

EMPY_INLINE auto AddScene(AssetID uid, const std::string&
source)
{
    auto asset = std::make_shared<SceneAsset>();
    asset->Type = AssetType::SCENE;
    Add(uid, source, asset);
    return asset;
}

```

This is basically all we need for the asset registry. We are now required to update the ECS components to incorporate these assets instead of saving the data themselves.

10.3 Integrating Assets

Now that the foundation for asset management is in place, let us integrate this into the application context.

Listing 10.9: Application/Context.h

```

#pragma once
#include "Scripts/Context.h"
#include "Physics/Context.h"
#include "Graphics/Renderer.h"
#include "Auxiliaries/Serializer.h"

namespace Empy
{

```

```
// forward declaration
struct AppInterface;

// application context
struct ApplicationContext
{
    EMPLY_INLINE ApplicationContext()
    {
        Window = std::make_unique<AppWindow>(&Dispatcher,
1280, 720, "Empty Engine");
        Scripts = std::make_unique<ScriptContext>(&Scene,
Window.get());
        Renderer = std::make_unique<GraphicsRenderer>
(1280, 720);
        Physics = std::make_unique<PhysicsContext>();
        Assets = std::make_unique<AssetRegistry>();
        DeltaTime = 0.0;
    }

    EMPLY_INLINE ~ApplicationContext()
    {
        for(auto layer : Layers)
        {
            EMPLY_DELETE(layer);
        }
    }

    std::unique_ptr<GraphicsRenderer> Renderer;
    std::unique_ptr<PhysicsContext> Physics;
    std::unique_ptr<ScriptContext> Scripts;
    std::unique_ptr<AssetRegistry> Assets;
    std::vector<AppInterface*> Layers;
    std::unique_ptr<AppWindow> Window;
    EventDispatcher Dispatcher;
    EntityRegistry Scene;
    double DeltaTime;
```

```
    } ;  
}
```

10.3.1 Updating Components

As stated earlier, we need to update the components to incorporate information about which assets they are using. This is important because when the scene is serialized, you need to know which entity has which asset. This is done in the following code snippet:

Listing 10.10: Auxiliaries/ECS.h

```
#pragma once  
  
#include "Assets.h"  
  
// skybox component  
struct SkyboxComponent  
{  
    EMPLY_INLINE SkyboxComponent(const SkyboxComponent&) =  
default;  
    EMPLY_INLINE SkyboxComponent() = default;  
    AssetID Skybox = EMPTY_ASSET;  
};  
  
// script component  
struct ScriptComponent  
{  
    EMPLY_INLINE ScriptComponent(const ScriptComponent&) =  
default;  
    EMPLY_INLINE ScriptComponent() = default;  
    AssetID Script = EMPTY_ASSET;
```

```

        LuaScript Instance;
    } ;

    // model component
    struct ModelComponent
    {
        EMPTY_INLINE ModelComponent(const ModelComponent&) = default;
        EMPTY_INLINE ModelComponent() = default;
        AssetID Material = EMPTY_ASSET;
        AssetID Model = EMPTY_ASSET;
    } ;

    // common component
    struct InfoComponent
    {
        EMPTY_INLINE InfoComponent(const InfoComponent&) = default;
        EMPTY_INLINE InfoComponent() = default;
        AssetID Parent = EMPTY_ASSET;
        std::string Name = "Entity";
        AssetID UID = RandomU64();
    } ;

```

You will note how all components that use assets now hold the asset identifier instead of the data itself. The `InfoComponent` is a common component for all entities, and this will help encapsulate critical information about an entity, such as its name and its universal unique identifier, which can be used for things such as entity hierarchies or, in other words, parent-child relationships.

Every time an entity with a model component is rendered, we must first retrieve the asset data from the asset registry before we can do so. This seems inefficient, but this is the path we have to take if we want to be able to serialize our game state. We now have to go back to the application and add the code to retrieve the asset before the component is used.

10.4 Updating Pipelines

We can now use these additional features to update the application rendering pipeline. We start by loading the asset and creating entities.

10.4.1 Loading Assets

You can see in the code snippet below how the asset identifier is provided to the component after its loading.

Listing 10.11: Application/Application. h

```
EMPY_INLINE void CreateEntities()
{
    // load assets
    auto skyboxAsset = m_Context->Assets-
>AddSkybox(RandomU64(), "Resources/Textures/HDRs/Sky.hdr",
2048);
    auto robotAsset = m_Context->Assets-
>AddModel(RandomU64(), "Resources/Models/Walking.fbx", true);
    auto scriptAsset = m_Context->Assets-
```

```

>AddScript(RandomU64(), "Resources/Scripts/TestScript.lua");
    auto sphereAsset = m_Context->Assets-
>AddModel(RandomU64(), "Resources/Models/sphere.fbx");
    auto cubeAsset = m_Context->Assets->AddModel(RandomU64(),
"Resources/Models/cube.fbx");
    auto mtlAsset = m_Context->Assets-
>AddMaterial(RandomU64(), "Nimrod");
    mtlAsset->Data.Albedo.x = 0.0f;

    // create scene camera
    auto camera = CreateEntt<Entity>();
    camera.Attach<InfoComponent>();
    camera.Attach<TransformComponent>().Transform.Translate.z
= 20.0f;
    camera.Attach<CameraComponent>();

    // create skybox entity
    auto skybox = CreateEntt<Entity>();
    skybox.Attach<InfoComponent>();
    skybox.Attach<SkyboxComponent>().Skybox = skyboxAsset-
>UID;
    skybox.Attach<TransformComponent>();

    // create robot entity
    auto robot = CreateEntt<Entity>();
    robot.Attach<InfoComponent>();
    auto& robotMod = robot.Attach<ModelComponent>();
    robotMod.Material = mtlAsset->UID;
    robotMod.Model = robotAsset->UID;
    auto& tr = robot.Attach<TransformComponent>().Transform;
    tr.Translate = glm::vec3(0.0f, -14.99f, -15.0f);
    tr.Scale = glm::vec3(0.1f);

    /* ... create more entities ... */
}

```

The `RandomU64()` function helps us generate a unique 64-bit number for every asset. Here is the implementation of this function.

Listing 10.12: Common/Core.h

```
/* ... same as before ... */

// include random
#include <random>

namespace EmPy
{
    // generate random 64-bit
    EMPY_INLINE uint64_t RandomU64()
    {
        static std::random_device device;
        static std::mt19937_64 generator(device());
        static std::uniform_int_distribution<uint64_t>
distribution;
        return distribution(generator);
    }
}

/* ... same as before ... */
```

Here is how the skybox and the script are initialized:

Listing 10.13: Application/Application.h

```
// generate environmental maps
EnttView<Entity, SkyboxComponent>([this] (auto entity, auto&
comp)
```

```

{
    // retrieve skybox asset
    auto& skybox = m_Context->Assets->Get<SkyboxAsset>
(comp.Skybox);

    // generate environment maps
    m_Context->Renderer->InitSkybox(skybox.Data,
skybox.EnvMap, skybox.Size);
};

// load, create and start scripts
EnttView<Entity, ScriptComponent>([this] (auto entity, auto&
comp)
{
    // retrieve script asset
    auto& script = m_Context->Assets->Get<ScriptAsset>
(comp.Script);
    // load script source
    auto name = m_Context->Scripts-
>LoadScript(script.Source);
    // attach to entity
    m_Context->Scripts->AttachScript(entity, name);
});

```

We are also required to do the same when rendering entities, be it for the shadow mapping, environment map, or PBR pipeline. Here is how this is done:

Listing 10.14: Application/Application. h

```

/* ... same as before ... */

// shadow mapping
EnttView<Entity, ModelComponent>([this, &lightDir] (auto

```

```

entity, auto& comp)
{
    auto& transform = entity.template Get<TransformComponent>()
        .Transform;
    auto& model = m_Context->Assets->Get<ModelAsset>(comp.Model);
    m_Context->Renderer->DrawDepth(model.Data, transform);
}

/* ... same as before ... */

// pbr rendering
EnttView<Entity, ModelComponent>([this] (auto entity, auto& comp)
{
    // retrieve assets
    auto& transform = entity.template Get<TransformComponent>()
        .Transform;
    auto& material = m_Context->Assets->Get<MaterialAsset>(comp.Material);
    auto& model = m_Context->Assets->Get<ModelAsset>(comp.Model);

    // render model
    m_Context->Renderer->Animate(model.Data, m_Context->DeltaTime);
    m_Context->Renderer->Draw(model.Data, material.Data, transform);
});

/* ... same as before ... */

// render skybox
EnttView<Entity, SkyboxComponent>([this] (auto entity, auto& comp)
{

```

```
    auto& transform = entity.template Get<TransformComponent>()
().Transform;
    auto& skybox = m_Context->Assets->Get<SkyboxAsset>
(comp.Skybox);
    m_Context->Renderer->DrawSkybox(skybox.Data, transform);
} );
```

You can now implement a function in the script context that helps you change the material or the model of an entity at runtime by just providing the asset identifier. We are now going to implement serialization.

Code not working? clone the branch "assets-serialization" on the GitHub repository and grant execution rights to the "EmPyLinux.sh" script on Linux. <https://github.com/Madsycode/book-empy-engine>.

```
git clone -branch <branch-name> <repository-link>
```

11 Serialization

A little sleep, a little slumber, a little folding of the hands to rest and poverty will come on you like a thief and scarcity like an armed man. (King Solomon)

Serialization in game engines plays a crucial role in the storage, transfer, and recreation of game data. It involves converting complex data structures, such as game objects, scenes, or states, into a format that can be easily saved to a file or transmitted over a network. The primary purposes of serialization in game engines include:

- ☞ **Persistence:** Serialization allows game developers to save the current state of a game or specific game elements, such as characters, levels, or items, to a file. This is essential for implementing features like saving and loading games, enabling players to resume their progress at a later time.
- ☞ **Data Transmission:** Games often need to send data between different systems, such as through client-server communication or multiplayer networking. Serialization ensures that the data can be efficiently converted into a format suitable for transmission over a network and reconstructed on the receiving end.
- ☞ **Interoperability:** Game engines may need to interact with external tools or services, such as level editors or

analytics platforms. Serialization enables the seamless exchange of data between the game engine and these external components, promoting interoperability.

☞ **Asset Management:** Serialized data is often used for storing and managing game assets, including 3D models, textures, and audio files. Serialization allows these assets to be stored in a standardized format, making it easier to organize, version, and distribute them within the game development pipeline.

11.1 Integrating YAML

YAML Ain't Markup Language (YAML) is a human-friendly data serialization language. YAML originally stood for "Yet Another Markup Language," but since 2002, it has been replaced with the recursive acronym "YAML Ain't Markup Language". As a result, YAML is defined as an encoding language that isn't one. Markup languages like XML and HTML are primarily used to store data in a human-readable format. YAML differs from XML in that it uses a simple approach. The main justification for YAML is its simplicity, which includes constraints that arise from XML's broad application spectrum. However, YAML's main competitor is JSON, rather than XML.

11.2 YAML-CPP

yaml-cpp is a C++ library for parsing and emitting YAML files. 'yaml-cpp' provides a convenient way to work with YAML files in C++.

11.2.1 yaml-cpp Syntax

Here's a brief introduction along with a simple code snippet:

Listing 11.1: yaml-cpp sample code

```
// Creating a YAML document
YAML::Emitter emitter;
emitter << YAML::BeginMap;
emitter << YAML::Key << "name" << YAML::Value << "John Doe";
emitter << YAML::Key << "age" << YAML::Value << 25;
emitter << YAML::EndMap;

// Outputting the YAML document
std::cout << "YAML Document:\n" << emitter.c_str() << "\n";

// Parsing a YAML document
YAML::Node config = YAML::Load("{name: Alice, age: 30}");

// Accessing values from the parsed YAML document
std::string name = config["name"].as<std::string>();
int age = config["age"].as<int>();

// Displaying the parsed values
std::cout << "Name: " << name << "\n";
std::cout << "Age: " << age << "\n";
```

In this example, the code creates a YAML document using `YAML :: Emitter` to represent a simple key-value structure, outputs the YAML document as a string, then parses a YAML document using `YAML :: Load()`. It accesses specific values from the parsed YAML document.

11.2.2 YAML File Layout

YAML uses indentation to represent the hierarchy and structure of the data. It is a human-readable format that is commonly used for configuration files and data exchange between different systems.

Listing 11.2: Sample YAML File

```
# Sample YAML file
person:
  name: John Doe
  age: 30
  address:
    city: Anytown
    country: USA
  hobbies:
    - Reading
    - Hiking
    - Coding
```

11.2.3 Library Setup

You can link yaml-cpp to your project using Conan, as depicted in the following code snippet:

Listing 11.3: Root/conanfile.txt

```
[requires]
glm/cci.20230113
stb/cci.20230920
magic_enum/0.9.5
yaml-cpp/0.8.0
opengl/system
spdlog/1.12.0
assimp/5.2.2
entt/3.12.2
physx/4.1.1
glfw/3.3.8
glew/2.2.0
sol2/3.3.1
lua/5.4.6

[generators]
cmake

[options]
yaml-cpp:shared=True
assimp:shared=False
physx:shared=True
glew:shared=False
glfw:shared=False
```

You will note that yaml-cpp is linked dynamically. It is also possible to link it statically by defining certain CMake options.

You can refer to the GitHub repository for more detail on that [Repository](#).

11.2.4 yaml-cpp Helpers

yaml-cpp does not come with the ability to serialize all types of data by default. Data such as `glm::vec3` are not supported. But yaml-cpp does offer ways to implement custom-type serializers and deserializers.

Listing 11.4: Common/YAM L

```
#pragma once
#include "Core.h"
#include <yaml-cpp/yaml.h>

namespace YAML
{
    template <>
    struct convert<glm::vec3>
    {
        EMPY_INLINE static Node encode(const glm::vec3& rhs)
        {
            Node node;
            node.push_back(rhs.x);
            node.push_back(rhs.y);
            node.push_back(rhs.z);
            node.SetStyle(EmitterStyle::Flow);
            return node;
        }

        EMPY_INLINE static bool decode(const Node& node,
```

```

glm::vec3& rhs)
{
    if (!node.IsSequence() || node.size() != 3)
    {
        return false;
    }

    rhs.x = node[0].as<float>();
    rhs.y = node[1].as<float>();
    rhs.z = node[2].as<float>();

    return true;
}

};

// stream operator
EMPTY_INLINE Emitter &operator << (Emitter& emitter,
const glm::vec3& v)
{
    emitter << Flow;
    emitter << BeginSeq << v.x << v.y << v.z << EndSeq;
    return emitter;
}
}

```

This code implements an encoder and a decoder for a *glm :: vec3*. You can use this same pattern to implement custom serializers for different data types.

11.3 Scene Serialization

We can now use this foundation to implement a serializer for our engine. We need to be able to serialize all entities and all

assets. We will start with the serialization of the scene. Here is a skeleton of the serializer class:

Listing 11.5: Auxiliaries/Serializer.h

```
#pragma once
#include "ECS.h"
#include "Common/YAML.h"

namespace Empy
{
    struct DataSerializer
    {
        EMPY_INLINE void Serialize(EntityRegistry& scene,
const std::string& path)
        {
            // serialize scene
        }

        EMPY_INLINE void Serialize(AssetRegistry& registry,
const std::string& path)
        {
            // serialize assets
        }

        EMPY_INLINE void Deserialize(EntityRegistry& scene,
const std::string& path)
        {
            // deserialize scene
        }

        EMPY_INLINE void Deserialize(AssetRegistry& registry,
const std::string& path)
        {
        }
    };
}
```

```
        // deserialize assets
    }
};

}
```

11.3.1 Serializing Entities

Below is the code to serialize the transform component and export it in the file provided as an argument to the function. You can see how we are using yaml-cpp to define the layout of the output file.

Listing 11.6: Auxiliaries/Serializer.h

```
EMPY_INLINE void Serialize(EntityRegistry& scene, const
std::string& path)
{
    YAML::Emitter emitter;

    emitter << YAML::BeginMap;
    {
        emitter << YAML::Key << "ENTITIES" << YAML::Value <<
YAML::BeginSeq;
        {
            scene.each([&] (EntityID entt)
            {
                // convert to entity object
                Entity entity(&scene, entt);

                // serialize component
                emitter << YAML::BeginMap;
                {
                    // serialize transform component
```

```

                if (entity.template
Has<TransformComponent>())
{
    auto& transform = entity.template
Get<TransformComponent>().Transform;
    emitter << YAML::Key <<
"TransformComponent" << YAML::BeginMap;
{
    emitter << YAML::Key <<
"Translate" << YAML::Value << transform.Translate;
    emitter << YAML::Key <<
"Rotation" << YAML::Value << transform.Rotation;
    emitter << YAML::Key << "Scale"
<< YAML::Value << transform.Scale;
}
    emitter << YAML::EndMap;
}
}
emitter << YAML::EndMap;
} );
}

emitter << YAML::EndSeq;
}
emitter << YAML::EndMap;

std::ofstream filepath(path);
filepath << emitter.c_str();
}

```

We iterate over each entity in the scene and proceed to serialize its components. The serialization includes handling the `TransformComponent` if the entity possesses one, capturing properties such as translation, rotation, and scale. The resulting YAML file is structured with an `ENTITIES` key

containing a sequence of mappings, where each mapping corresponds to an entity and its serialized components. Finally, the serialized YAML content is written to the specified file path.

Serializing Components

This function can be extended to support all the other component types as shown in the following code:

Listing 11.7: Auxiliaries/Serializer.h

```
EMPTY_INLINE void Serialize(EntityRegistry& scene, const std::string& path)
{
    YAML::Emitter emitter;

    emitter << YAML::BeginMap;
    {
        emitter << YAML::Key << "ENTITIES" << YAML::Value << YAML::BeginSeq;
        {
            scene.each([&] (EntityID entt)
            {
                // create entity object
                Entity entity(&scene, entt);

                // serialize component
                emitter << YAML::BeginMap;
                {
                    // serialize transform component
```

```

        if (entity.template Has<InfoComponent>())
        {
            auto& info = entity.template
Get<InfoComponent>();
            emitter << YAML::Key <<
"InfoComponent" << YAML::BeginMap;
            {
                emitter << YAML::Key << "UID" <<
YAML::Value << info.UID;
                emitter << YAML::Key << "Name" <<
YAML::Value << info.Name;
                emitter << YAML::Key << "Parent"
<< YAML::Value << info.Parent;
            }
            emitter << YAML::EndMap;
        }

        // serialize transform component
        if (entity.template Has<CameraComponent>
())
        {
            auto& camera = entity.template
Get<CameraComponent>().Camera;
            emitter << YAML::Key <<
"CameraComponent" << YAML::BeginMap;
            {
                emitter << YAML::Key <<
"NearPlane" << YAML::Value << camera.NearPlane;
                emitter << YAML::Key <<
"FarPlane" << YAML::Value << camera.FarPlane;
                emitter << YAML::Key << "FOV" <<
YAML::Value << camera.FOV;
            }
            emitter << YAML::EndMap;
        }
    }
}

```

```

        // serialize transform component
        if (entity.template
Has<TransformComponent>())
    {
        auto& transform = entity.template
Get<TransformComponent>().Transform;
        emitter << YAML::Key <<
"TransformComponent" << YAML::BeginMap;
    {
        emitter << YAML::Key <<
"Translate" << YAML::Value << transform.Translate;
        emitter << YAML::Key <<
"Rotation" << YAML::Value << transform.Rotation;
        emitter << YAML::Key << "Scale"
<< YAML::Value << transform.Scale;
    }
    emitter << YAML::EndMap;
}

        // serialize rigidbody component
        if (entity.template
Has<RigidBodyComponent>())
    {
        auto& body = entity.template
Get<RigidBodyComponent>().RigidBody;
        emitter << YAML::Key <<
"RigidBodyComponent" << YAML::BeginMap;
    {
        emitter << YAML::Key << "Density"
<< YAML::Value << body.Density;
        emitter << YAML::Key << "Dynamic"
<< YAML::Value << body.Dynamic;
    }
    emitter << YAML::EndMap;
}

```

```

        // serialize collider component
        if (entity.template
Has<ColliderComponent>())
{
    auto& collider = entity.template
Get<ColliderComponent>().Collider;
    std::string
type(magic_enum::enum_name(collider.Type));

    emitter << YAML::Key <<
"ColliderComponent" << YAML::BeginMap;
{
    emitter << YAML::Key <<
"DynamicFriction" << YAML::Value << collider.DynamicFriction;
    emitter << YAML::Key <<
"StaticFriction" << YAML::Value << collider.StaticFriction;
    emitter << YAML::Key <<
"Restitution" << YAML::Value << collider.Restitution;
    emitter << YAML::Key << "Type" <<
YAML::Value << type;
}
emitter << YAML::EndMap;
}

        // serialize pointlight component
        if (entity.template
Has<DirectLightComponent>())
{
    auto& light = entity.template
Get<DirectLightComponent>().Light;

    emitter << YAML::Key <<
"DirectLightComponent" << YAML::BeginMap;
{
    emitter << YAML::Key <<
"Intensity" << YAML::Value << light.Intensity;
}

```

```

                emitter << YAML::Key <<
"Radiance" << YAML::Value << light.Radiance;
                emitter << YAML::Key << "Bias" <<
YAML::Value << light.ShadowBias;
            }
            emitter << YAML::EndMap;
        }

        // serialize pointlight component
        if (entity.template
Has<PointLightComponent>())
{
    auto& light = entity.template
Get<PointLightComponent>().Light;

        emitter << YAML::Key <<
"PointLightComponent" << YAML::BeginMap;
    {
        emitter << YAML::Key <<
"Intensity" << YAML::Value << light.Intensity;
        emitter << YAML::Key <<
"Radiance" << YAML::Value << light.Radiance;
    }
    emitter << YAML::EndMap;
}

        // serialize spotlight component
        if (entity.template
Has<SpotLightComponent>())
{
    auto& light = entity.template
Get<SpotLightComponent>().Light;

        emitter << YAML::Key <<
"SpotLightComponent" << YAML::BeginMap;
    {

```

```

                emitter << YAML::Key <<
"Intensity" << YAML::Value << light.Intensity;
                emitter << YAML::Key <<
"Radiance" << YAML::Value << light.Radiance;
                emitter << YAML::Key << "FallOff"
<< YAML::Value << light.FallOff;
                emitter << YAML::Key << "CutOff"
<< YAML::Value << light.CutOff;
}
emitter << YAML::EndMap;
}

// serialize script component
if (entity.template Has<ScriptComponent>
())
{
    emitter << YAML::Key <<
"ScriptComponent" << YAML::BeginMap;
{
    auto& asset = entity.template
Get<ScriptComponent>().Script;
    emitter << YAML::Key << "Script"
<< YAML::Value << asset;
}
emitter << YAML::EndMap;
}

// serialize script component
if (entity.template Has<SkyboxComponent>
())
{
    emitter << YAML::Key <<
"SkyboxComponent" << YAML::BeginMap;
{
    auto& asset = entity.template
Get<SkyboxComponent>().Skybox;
}
```

```

                emitter << YAML::Key << "Skybox"
<< YAML::Value << asset;
}
emitter << YAML::EndMap;
}

// serialize model component
if (entity.template Has<ModelComponent>
())
{
    auto& comp = entity.template
Get<ModelComponent>();
    emitter << YAML::Key <<
"ModelComponent" << YAML::BeginMap;
{
    emitter << YAML::Key <<
"Material" << YAML::Value << comp.Material;
    emitter << YAML::Key << "Model"
<< YAML::Value << comp.Model;
}
emitter << YAML::EndMap;
}
emitter << YAML::EndMap;
} );
}

emitter << YAML::EndSeq;
}
emitter << YAML::EndMap;

std::ofstream filepath(path);
filepath << emitter.c_str();
}

```

11.3.2 Deserializing Entities

Below is the code to deserialize the transform component from the source file provided as an argument to the function.

Listing 11.8: Auxiliaries/Serializer.h

```
EMPTY_INLINE void Deserialize(EntityRegistry& scene, const
std::string& path)
{
    try
    {
        auto root = YAML::LoadFile(path);
        auto& nodes = root["ENTITIES"];
        scene.clear();

        for(auto& node : nodes)
        {
            // deserialize transform
            if (auto& data = node["TransformComponent"])
            {
                auto& transform =
scene.emplace<TransformComponent>(entity).Transform;
                transform.Translate =
data["Translate"].as<glm::vec3>();
                transform.Rotation =
data["Rotation"].as<glm::vec3>();
                transform.Scale = data["Scale"].as<glm::vec3>();
            }
        }
    }
    catch (YAML::ParserException& e)
    {
        EMPTY_ERROR("failed to deserialize scene!");
    }
}
```

```
    }  
}
```

You can see how we are able to cast the type of the field using template types. Even the `glm::vec3` is cast. This is done using the helper we created earlier.

Deserializing Components

The `Deserialize()` function can be extended to support all the other component types as shown in the following code:

Listing 11.9: Auxiliaries/Serializer.h

```
EMPY_INLINE void Deserialize(EntityRegistry& scene, const  
std::string& path)  
{  
    try  
    {  
        auto root = YAML::LoadFile(path);  
        auto& nodes = root["ENTITIES"];  
        scene.clear();  
  
        // deserialize nodes  
        for(auto& node : nodes)  
        {  
            // create entity  
            EntityID entity = scene.create();  
  
            // deserialize entt infos  
            if (auto& data = node["InfoComponent"])
```

```

    {

        auto& info = scene.emplace<InfoComponent>
(entity);

        info.Parent = data["Parent"].as<AssetID>();
        info.Name = data["Name"].as<std::string>();
        info.UID = data["UID"].as<AssetID>();

    }

    // deserialize entt infos

    if (auto& data = node["CameraComponent"])
    {

        auto& camera = scene.emplace<CameraComponent>
(entity).Camera;

        camera.NearPlane =
data["NearPlane"].as<float>();
        camera.FarPlane = data["FarPlane"].as<float>()
();
        camera.FOV = data["FOV"].as<float>();

    }

    // deserialize transform

    if (auto& data = node["TransformComponent"])
    {

        auto& transform =
scene.emplace<TransformComponent>(entity).Transform;
        transform.Translate =
data["Translate"].as<glm::vec3>();
        transform.Rotation =
data["Rotation"].as<glm::vec3>();
        transform.Scale = data["Scale"].as<glm::vec3>()

();;

    }

    // deserialize rigidbody

    if (auto& data = node["RigidBodyComponent"])
    {

```

```

        auto& body =
scene.emplace<RigidBodyComponent>(entity).RigidBody;
        body.Density = data["Density"].as<float>();
        body.Dynamic = data["Dynamic"].as<bool>();
    }

    // deserialize collider
    if (auto& data = node["ColliderComponent"])
    {
        auto& collider =
scene.emplace<ColliderComponent>(entity).Collider;
        collider.DynamicFriction =
data["DynamicFriction"].as<float>();
        collider.StaticFriction =
data["StaticFriction"].as<float>();
        collider.Restitution =
data["Restitution"].as<float>();
        // get type
        const auto name =
data["Type"].as<std::string>();
        auto type =
magic_enum::enum_cast<ColliderType>(name);
        if(type.has_value()) { collider.Type =
type.value(); }
    }

    // deserialize model
    if (auto& data = node["ModelComponent"])
    {
        auto& comp = scene.emplace<ModelComponent>
(entity);
        comp.Material = data["Material"].as<AssetID>
();
        comp.Model = data["Model"].as<AssetID>();
    }

```

```

        // deserialize script
        if (auto& data = node["ScriptComponent"])
        {
            auto& comp = scene.emplace<ScriptComponent>
(entity);
            comp.Script = data["Script"].as<AssetID>();
        }

        // deserialize script
        if (auto& data = node["SkyboxComponent"])
        {
            auto& comp = scene.emplace<SkyboxComponent>
(entity);
            comp.Skybox = data["Skybox"].as<AssetID>();
        }

        // deserialize direct light
        if (auto& data = node["DirectLightComponent"])
        {
            auto& light =
scene.emplace<DirectLightComponent>(entity).Light;
            light.Radiance =
data["Radiance"].as<glm::vec3>();
            light.Intensity = data["Intensity"].as<float>
();
            light.ShadowBias = data["Bias"].as<float>();
        }

        // deserialize point light
        if (auto& data = node["PointLightComponent"])
        {
            auto& light =
scene.emplace<PointLightComponent>(entity).Light;
            light.Radiance =
data["Radiance"].as<glm::vec3>();
            light.Intensity = data["Intensity"].as<float>
();
        }
    }
}

```

```

() ;

}

// deserialize point light
if (auto& data = node["SpotLightComponent"])
{
    auto& light =
scene.emplace<SpotLightComponent>(entity).Light;
    light.Radiance =
data["Radiance"].as<glm::vec3>();
    light.Intensity = data["Intensity"].as<float>()
();
    light.FallOff = data["FallOff"].as<float>();
    light.CutOff = data["CutOff"].as<float>();
}
}

}

catch (YAML::ParserException& e)
{
    EMPLY_ERROR("failed to deserialize scene!");
}
}
}

```

11.4 Assets Serialization

We can do the same thing for assets. Following are the implementations of both functions to serialize and deserialize assets.

11.4.1 Serializing Assets

Below is the code to serialize all assets present in the application.

Listing 11.10: Auxiliaries/Serializer.h

```
EMPTY_INLINE void Serialize(AssetRegistry& registry, const
std::string& path)
{
    YAML::Emitter emitter;

    emitter << YAML::BeginMap;
    {
        emitter << YAML::Key << "ASSETS" << YAML::Value <<
YAML::BeginSeq;
        {
            registry.View([&] (Asset* asset)
            {
                // asset type to string
                std::string
typeName(magic_enum::enum_name(asset->Type));

                // serialize asset item
                emitter << YAML::BeginMap;
                {
                    // asset auxiliaries
                    emitter << YAML::Key << "UID" <<
YAML::Value << asset->UID;
                    emitter << YAML::Key << "Type" <<
YAML::Value << typeName;
                    emitter << YAML::Key << "Name" <<
YAML::Value << asset->Name;
                    emitter << YAML::Key << "Source" <<
YAML::Value << asset->Source;

                    // serialize material
                    if(asset->Type == AssetType::MATERIAL)
                    {

```

```

        auto mtl =
static_cast<MaterialAsset*>(asset);
                emitter << YAML::Key << "Properties"
<< YAML::BeginMap;
{
    // texture assets
    emitter << YAML::Key <<
"RoughnessMap" << YAML::Value << mtl->RoughnessMap;
    emitter << YAML::Key <<
"OcclusionMap" << YAML::Value << mtl->OcclusionMap;
    emitter << YAML::Key <<
"EmissiveMap" << YAML::Value << mtl->EmissiveMap;
    emitter << YAML::Key <<
"MetallicMap" << YAML::Value << mtl->MetallicMap;
    emitter << YAML::Key <<
"AlbedoMap" << YAML::Value << mtl->AlbedoMap;
    emitter << YAML::Key <<
"NormalMap" << YAML::Value << mtl->NormalMap;

    // properties
    emitter << YAML::Key <<
"Roughness" << YAML::Value << mtl->Data.Roughness;
    emitter << YAML::Key <<
"Occlusion" << YAML::Value << mtl->Data.Occlusion;
    emitter << YAML::Key <<
"Metallic" << YAML::Value << mtl->Data.Metallic;
    emitter << YAML::Key <<
"Emissive" << YAML::Value << mtl->Data.Emissive;
    emitter << YAML::Key << "Albedo"
<< YAML::Value << mtl->Data.Albedo;
}
    emitter << YAML::EndMap;
}
// serialize texture
else if(asset->Type ==
AssetType::TEXTURE)

```

```

    {
        auto texture =
static_cast<TextureAsset*>(asset);
        emitter << YAML::Key << "Properties"
<< YAML::BeginMap;
    {
        emitter << YAML::Key << "IsHDR"
<< YAML::Value << texture->IsHDR;
        emitter << YAML::Key << "FlipV"
<< YAML::Value << texture->FlipV;
    }
    emitter << YAML::EndMap;
}
// serialize skybox
else if(asset->Type == AssetType::SKYBOX)
{
    auto skybox =
static_cast<SkyboxAsset*>(asset);
    emitter << YAML::Key << "Properties"
<< YAML::BeginMap;
{
    emitter << YAML::Key << "IsHDR"
<< YAML::Value << skybox->IsHDR;
    emitter << YAML::Key << "FlipV"
<< YAML::Value << skybox->FlipV;
    emitter << YAML::Key << "Size" <<
YAML::Value << skybox->Size;
}
    emitter << YAML::EndMap;
}
// serialize model
else if(asset->Type == AssetType::MODEL)
{
    auto model = static_cast<ModelAsset*>
(asset);
    emitter << YAML::Key << "Properties"
}

```

```

<< YAML::BeginMap;
{
    emitter << YAML::Key <<
"HasJoints" << YAML::Value << model->HasJoints;
}
emitter << YAML::EndMap;
}
emitter << YAML::EndMap;
} );
}
emitter << YAML::EndSeq;
}
emitter << YAML::EndMap;

std::ofstream filepath(path);
filepath << emitter.c_str();
}

```

11.4.2 Deserializing Assets

We need to do the same to deserialize assets from the asset source file.

Listing 11.11: Auxiliaries/Serializer.h

```

EMPTY_INLINE void Deserialize(AssetRegistry& registry, const
std::string& path)
{
    try
    {
        auto root = YAML::LoadFile(path);
        auto& nodes = root["ASSETS"];

```

```

    // deserialize nodes
    for(auto& node : nodes)
    {
        Asset* asset = nullptr;
        auto props = node["Properties"];
        auto uid = node["UID"].as<AssetID>();
        auto name = node["Name"].as<std::string>();
        auto source = node["Source"].as<std::string>();

        // get asset type
        auto typeName = node["Type"].as<std::string>();
        auto opt = magic_enum::enum_cast<AssetType>
(typeName);
        auto type = (opt.has_value()) ? opt.value() :
AssetType::UNKNOWN;

        // create instance
        if(type == AssetType::MATERIAL && props)
        {
            auto mtlAsset = registry.AddMaterial(uid,
source);

            // set material textures
            mtlAsset->RoughnessMap =
props["RoughnessMap"].as<AssetID>();
            mtlAsset->OcclusionMap =
props["OcclusionMap"].as<AssetID>();
            mtlAsset->EmissiveMap =
props["EmissiveMap"].as<AssetID>();
            mtlAsset->MetallicMap =
props["MetallicMap"].as<AssetID>();
            mtlAsset->AlbedoMap =
props["AlbedoMap"].as<AssetID>();
            mtlAsset->NormalMap =
props["NormalMap"].as<AssetID>();

```

```

        // set material props
        mtlAsset->Data.Emissive =
props["Emissive"].as<glm::vec3>();
        mtlAsset->Data.Occlusion =
props["Occlusion"].as<float>();
        mtlAsset->Data.Roughness =
props["Roughness"].as<float>();
        mtlAsset->Data.Metallic =
props["Metallic"].as<float>();
        mtlAsset->Data.Albedo =
props["Albedo"].as<glm::vec3>();

        // cast asset
        asset = (Asset*)mtlAsset.get();
    }
    else if(type == AssetType::TEXTURE && props)
    {
        // set properties
        bool isHDR = props["IsHDR"].as<bool>();
        bool flipV = props["FlipV"].as<bool>();
        asset = (Asset*)registry.AddTexture(uid,
source, isHDR, flipV).get();
    }
    else if(type == AssetType::SKYBOX && props)
    {
        // set properties
        bool isHDR = props["IsHDR"].as<bool>();
        bool flipV = props["FlipV"].as<bool>();
        auto size = props["Size"].as<int32_t>();
        asset = (Asset*)registry.AddSkybox(uid,
source, size, isHDR, flipV).get();
    }
    else if(type == AssetType::MODEL && props)
    {
        bool hasJoints = props["HasJoints"].as<bool>

```

```

() ;

        asset = (Asset*) registry.AddModel(uid,
source, hasJoints).get();
    }
    else if(type == AssetType::SCRIPT)
    {
        asset = (Asset*) registry.AddScript(uid,
source).get();
    }
    else if(type == AssetType::SCENE)
    {
        asset = (Asset*) registry.AddScene(uid,
source).get();
    }
    else
    {
        EMPLY_ERROR("failed to deserialize asset:
invalid type");
        return;
    }

    // set asset properties
    asset->Source = source;
    asset->Name = name;
}

}

catch (YAML::ParserException& e)
{
    EMPLY_ERROR("failed to deserialize assets!");
}
}

```

11.5 Integrating in Application

We can then incorporate this in the application context as shown in the following code snippet:

Listing 11.12: Application/Context.h

```
#pragma once

#include "Scripts/Context.h"
#include "Physics/Context.h"
#include "Graphics/Renderer.h"
#include "Auxiliaries/Serializer.h"

namespace EmPy
{
    // forward declaration
    struct AppInterface;

    // application context
    struct ApplicationContext
    {
        EMPY_INLINE ApplicationContext()
        {
            Window = std::make_unique<AppWindow>(&Dispatcher,
1280, 720, "EmPy Engine");
            Scripts = std::make_unique<ScriptContext>(&Scene,
Window.get());
            Renderer = std::make_unique<GraphicsRenderer>
(1280, 720);
            Serializer = std::make_unique<DataSerializer>();
            Physics = std::make_unique<PhysicsContext>();
            Assets = std::make_unique<AssetRegistry>();
            DeltaTime = 0.0;
        }

        EMPY_INLINE ~ApplicationContext()
```

```

    {
        for (auto layer : Layers)
        {
            EMPTY_DELETE(layer);
        }
    }

    std::unique_ptr<GraphicsRenderer> Renderer;
    std::unique_ptr<DataSerializer> Serializer; // -->
added!
    std::unique_ptr<PhysicsContext> Physics;
    std::unique_ptr<ScriptContext> Scripts;
    std::unique_ptr<AssetRegistry> Assets;
    std::vector<AppInterface*> Layers;
    std::unique_ptr<AppWindow> Window;
    EventDispatcher Dispatcher;
    EntityRegistry Scene;
    double DeltaTime;
};

}

```

11.5.1 Testing Serialization

Moving to the application header file, we can simply call this function after all entities are created and assets are loaded to serialize the data in YAML files. This is done in the following code snippet:

Listing 11.13: Application/Application.h

```

EMPTY_INLINE void CreateSceneEntites()
{
    // load assets
}

```

```

        auto skyboxAsset = m_Context->Assets-
>AddSkybox(RandomU64(), "Resources/Textures/HDRs/Sky.hdr",
2048);

        auto robotAsset = m_Context->Assets-
>AddModel(RandomU64(), "Resources/Models/Walking.fbx", true);

        auto scriptAsset = m_Context->Assets-
>AddScript(RandomU64(), "Resources/Scripts/TestScript.lua");

        auto sphereAsset = m_Context->Assets-
>AddModel(RandomU64(), "Resources/Models/sphere.fbx");

        auto cubeAsset = m_Context->Assets->AddModel(RandomU64(),
"Resources/Models/cube.fbx");

        auto mtlAsset = m_Context->Assets-
>AddMaterial(RandomU64(), "Nimrod");

        mtlAsset->Data.Albedo.x = 0.0f;

        // create scene camera

        auto camera = CreateEntt<Entity>();
        camera.Attach<InfoComponent>();
        camera.Attach<TransformComponent>().Transform.Translate.z
= 20.0f;
        camera.Attach<CameraComponent>();

        // create skybox entity

        auto skybox = CreateEntt<Entity>();
        skybox.Attach<InfoComponent>();
        skybox.Attach<SkyboxComponent>().Skybox = skyboxAsset-
>UID;
        skybox.Attach<TransformComponent>();

        /* ... create more entities ... */

        // serialize all entities and assets!
        m_Context->Serializer->Serialize(*m_Context->Assets,
"Resources/Projects/assets.yaml");
        m_Context->Serializer->Serialize(m_Context->Scene,

```

```
"Resources/Projects/scene.yaml");  
}
```

Create the folder Project in the Resources directory.

11.5.2 Serialization Results

The two listings below show the output files of the serialization.

Assets Output

Listing 11.14: Resources/Projects/assets.yam
|

```
ASSETS:  
- UID: 3203648000287975621  
  Type: MATERIAL  
  Name: Nimrod  
  Source: Nimrod  
  Properties:  
    RoughnessMap: 0  
    OcclusionMap: 0  
    EmissiveMap: 0  
    MetallicMap: 0  
    AlbedoMap: 0  
    NormalMap: 0  
    Roughness: 0.5  
    Occlusion: 1  
    Metallic: 0.600000024  
    Emissive: [0, 0, 0]  
    Albedo: [0, 1, 1]
```

```
- UID: 6859465390222840426
  Type: MODEL
  Name: Walking
  Source: Resources/Models/Walking.fbx
  Properties:
    HasJoints: true
- UID: 10918668268976596116
  Type: MODEL
  Name: sphere
  Source: Resources/Models/sphere.fbx
  Properties:
    HasJoints: false
- UID: 10562925605299858384
  Type: MODEL
  Name: cube
  Source: Resources/Models/cube.fbx
  Properties:
    HasJoints: false
- UID: 12834013311619044530
  Type: SCRIPT
  Name: TestScript
  Source: Resources/Scripts/TestScript.lua
- UID: 12699942177245588509
  Type: SKYBOX
  Name: Sky
  Source: Resources/Textures/HDRs/Sky.hdr
  Properties:
    IsHDR: true
    FlipV: true
    Size: 2048
```

Scene Output

Listing 11.15: Resources/Projects/scene.yam

```
ENTITIES:
- InfoComponent:
  UID: 10685569070467818381
  Name: Entity9
  Parent: 0
  TransformComponent:
    Translate: [0, 54, -10]
    Rotation: [0, 0, 0]
    Scale: [5, 5, 5]
  RigidBodyComponent:
    Density: 1
    Dynamic: true
  ColliderComponent:
    DynamicFriction: 0.5
    StaticFriction: 0.300000012
    Restitution: 0.400000006
    Type: BOX
  ModelComponent:
    Material: 3203648000287975621
    Model: 10562925605299858384
- InfoComponent:
  UID: 2697344576064221484
  Name: Entity8
  Parent: 0
  TransformComponent:
    Translate: [0, 48, -10]
    Rotation: [0, 0, 0]
    Scale: [5, 5, 5]
  RigidBodyComponent:
    Density: 1
    Dynamic: true
  ColliderComponent:
```

```
DynamicFriction: 0.5
StaticFriction: 0.300000012
Restitution: 0.400000006
Type: BOX
ModelComponent:
    Material: 3203648000287975621
    Model: 10562925605299858384
```

11.5.3 Loading Scene from File

You can now remove the `CreateSceneEntites()` and call the deserializer functions to see if the scene and the scene are loaded accordingly.

Listing 11.16: Application/Application. h

```
EMPY_INLINE void CreateSceneEntites()
{
    /* ... comment every thing ... */

    m_Context->Serializer->Deserialize(*m_Context->Assets,
"Resources/Projects/assets.yaml");
    m_Context->Serializer->Deserialize(m_Context->Scene,
"Resources/Projects/scene.yaml");
}
```

This code deserializes both the scene and the assets.

Code not working? clone the branch "assets-serialization" on the GitHub repository and grant execution rights to the "EmpyLinux.sh" script on

Linux. <https://github.com/Madsycode/book-empy-engine>.

```
git clone -branch <branch-name> <repository-link>
```

Part V

Editor Interface

12 Scene Editor

Do not be one who shakes hands in pledge or puts up security for debts (King Solomon)

A game editor provides a comprehensive toolset for designers and developers to create games with ease. At its core, the editor User Interface (UI) typically consists of panels, menus, and various interactive elements that facilitate the manipulation of game assets, scenes, and settings. These components are strategically organized to streamline the development workflow. Central to the UI is the Scene View, allowing creators to visualize and manipulate the game environment in a 3D space. Concurrently, the hierarchy panel provides a structured overview of the game's elements, aiding in efficient management and organization.



Figure 12.1: Editor Components

The Inspector panel plays a pivotal role, offering detailed control over selected game objects. This is where properties, components, and scripts can be fine-tuned, allowing for precise customization. Additionally, the Asset Browser simplifies the import and management of resources, ensuring a smooth integration of graphics, audio, and other elements. See (Figure 12.1)

12.1 Integrating ImGu

ImGui (Dear ImGui) [Omar](#) short for Immediate Mode Graphical User Interface, is a lightweight and powerful GUI library primarily designed for use in game development and real-time applications. It follows an immediate mode paradigm, allowing developers to dynamically create and manipulate UI elements directly within their rendering loop.



Figure 12.2:

Made with ImGui
[Omar](#)

ImGui has been widely adopted in the game development community for creating in-game tools and debug interfaces. It provides a quick and efficient way to visualize and modify variables during runtime. It is also utilized in various visualization tools for data analysis and simulation applications. Its ease of integration and flexibility make it a preferred choice for rapidly prototyping and implementing custom UIs.

12.1.1 How to Use Dear ImGui

Here are the steps to follow to integrate ImGui into a C++ project:

- **Download ImGui:** The first step involves obtaining the ImGui library from the official repository on GitHub or use package managers like Conan or vcpkg.
- **Include ImGui in Your Project:** The next step is to copy the ImGui source and header files to our project. Ensure that the necessary dependencies are included as well.
- **Initialize ImGui:** In your application initialization, set up ImGui by calling the appropriate functions. This typically involves initializing the ImGui context and setting up a rendering backend.

- **Create a ImGui Frame:** Then, we can create ImGui frames within our rendering loop using `ImGui::NewFrame()`.
- **Add UI Elements:** Use ImGui functions to add various UI elements such as buttons, sliders, and text inputs. For example, create a window using `ImGui::Begin()` and add a button with `ImGui::Button()`.
- **Render ImGui Draw Data:** After adding UI elements, end the ImGui frame with `ImGui::Render()`. ImGui will generate draw data that can be incorporated into our rendering pipeline.
- **Render ImGui Draw Lists:** Finally, we can then render the ImGui draw lists or frame through our graphics API (OpenGL, DirectX, etc.) to display the UI on the screen.

Here's (Listing 12.1) a simple code snippet demonstrating the basic setup:

Listing 12.1: ImGui Sample Code

```
#include "imgui.h"

int main()
{
    // initialize ImGui
    ImGui::CreateContext();
    ImGuiIO& io = ImGui::GetIO();

    // setup other configuration options, e.g., rendering
    backend
```

```

while /* application's main loop condition */
{
    // Start a new ImGui frame
    ImGui::NewFrame();

    // start ui window elements
    ImGui::Begin("My Window");
    {
        if (ImGui::Button("Click Me"))
            // handle button click logic
    }
    // end window ui element
    ImGui::End();

    // show demo window of imgui
    ImGui::ShowDemoWindow();

    // render ImGui draw data
    ImGui::Render();

    // render ImGui draw lists using OpenGL
    // Update and render application
}

// cleanup imgui context
ImGui::DestroyContext();

return 0;
}

```

Let us follow these steps to integrate ImGui into our game editor.

Extending Project Tree

Expand your project directory tree according to the structure illustrated in (Figure 12.3). Take notice of the inclusion of the "**Vendors**" folder, designated to accommodate external code not originating from our project. The "**imgui**" folder is reserved for hosting all source files from ImGui. Additionally, the header file "**FA.h**" is meant to contain the definitions of **Font Awesome** icons.

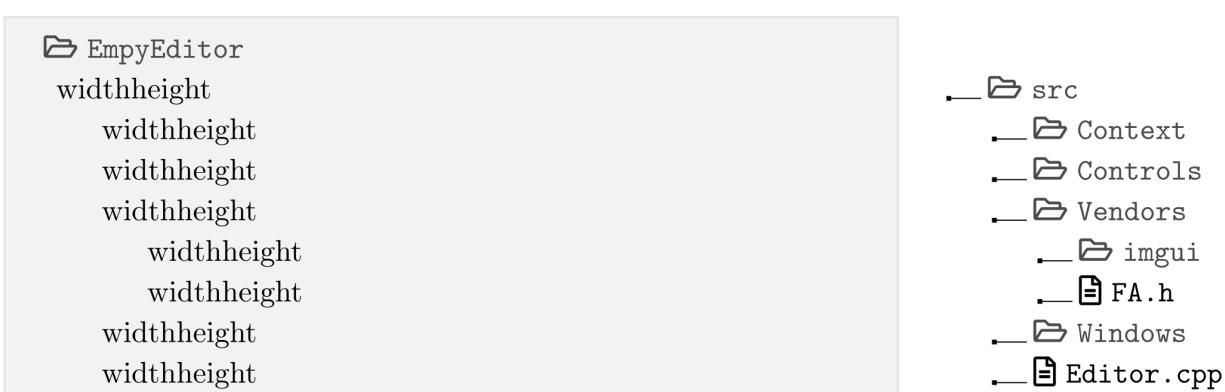


Figure 12.3: Editor Directory

Font Awesome [Team](#) is a popular and comprehensive icon set and toolkit designed for web development and graphic design. It provides a collection of scalable vector icons that can be customized and easily integrated into websites, applications, and various digital projects. The GitHub repository **IconFontCppHeaders** [Foucaut and all.](#) provides C++ header files to incorporate Font Awesome with ImGui.

Downloading Dear ImGui

When incorporating ImGui into a C++ project, there are two primary methods: incorporating the ImGui source files directly into the project or utilizing package managers such as Conan or vcpkg. For our project, we have opted for the former approach. ImGui facilitates the use of various graphics APIs (OpenGL, SDL2, DirectX, etc.), allowing us to choose our preferred backend. Unfortunately, Conan does not offer a straightforward solution for selecting the desired backend API, making the direct inclusion of ImGui source files a more pragmatic choice for our specific project requirements.

☞ **Download ImGui** from GitHub using the following link: <https://github.com/ocornut/imgui>. Ensure that you select the **docking** branch during the download process, as this branch specifically includes support for window docking. See (Figure 12.4).

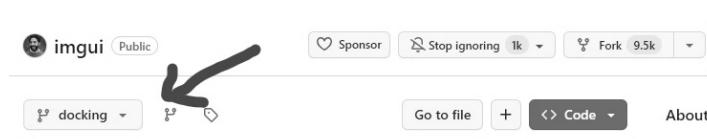


Figure 12.4: Downloading ImGui

- ☞ **Unpack and Copy** all header and source files present at the root level of the ImGui directory (Figure 12.5) you just downloaded. Paste them all in the ImGui folder in your project tree.

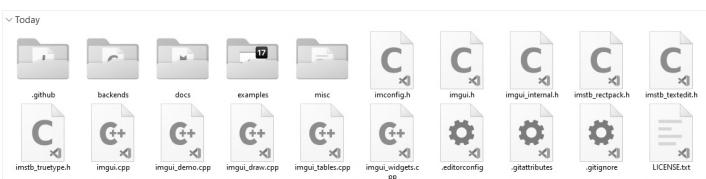


Figure 12.5: ImGui Root Directory

- ☞ **Copy OpenGL Backend** files present in the "backend" folder and paste them in the ImGui folder as well. (Figure 12.6): is how your ImGui folder should look.



Figure 12.6: ImGui Directory

It's noteworthy that ImGui comes bundled with a comprehensive demo file "**imgui_demo.cpp**", showcasing the library's capabilities and providing an interactive exploration of its features. As depicted in (Listing 12.1), you can show this using the `ImGui::ShowDemoWindow()` function. The demo serves as an invaluable resource for users, offering a hands-on experience to understand the various UI elements, customization options, and functionalities provided by ImGui.

Adding Font Awesome

As stated before, we are also going to be using Font Awesome to make our UI look professional. Therefore, we need to download the Font Awesome header file from this link: <https://github.com/juliettef/IconFontCppHeaders>. You can copy the content of the header called "**IconsFontAwesome6.h**" and paste its content in your "FA.h" header file.

Keep in mind that these files are available in the source code provided in our GitHub repository. So if you don't want to deal with all these dependencies, you can just download the provided code.

12.2 Creating a GUI Context

Our initial step toward rendering UI elements involves defining key components to ensure a seamless development experience. The essential files required for this purpose are outlined in the following folder tree. Ensure that you incorporate these files into your project.

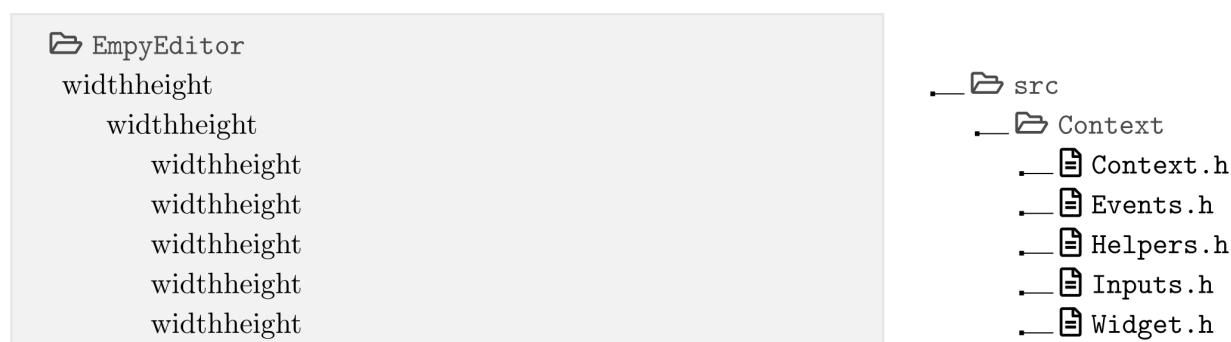


Figure 12.7: Gui Context Directory

12.2.1 Helpers and Configurations

Add the following code to your header file, "Helpers.h":

Listing 12.2: Context/Helpers.h

```
#pragma once
#include <Empty.h>
#include <Vendors/FA.h>
#include <Vendors/imgui/imgui.h>
```

```

#include <Vendors/imgui/imgui_internal.h>
#include <Vendors/imgui/imgui_impl_glfw.h>
#include <Vendors/imgui/imgui_impl_opengl3.h>

using namespace Empy;

#define FONT_FILE "Resources/Fonts/Roboto-Medium.ttf"
#define ICON_FONT "Resources/Fonts/fa-solid-900.ttf"
#define SHADER_VERSION "#version 130"

#define REGULAR_FONT_SIZE 17
#define SMALL_FONT_SIZE 15

#define LABEL_SPACING 130
#define MENU_BAR_HEIGHT 50

#define ASSET_SPACING 10
#define ASSET_SIZE 90

// helper macros
#define LINE_HEIGHT() ImGui::GetTextLineHeight()
#define USE_SMALL_FONT()
ImGui::SetCurrentFont(ImGui::GetIO().Fonts->Fonts[1])
#define USE_REGULAR_FONT()
ImGui::SetCurrentFont(ImGui::GetIO().Fonts->Fonts[0])
#define TEXT_HEIGHT() ImGui::GetTextLineHeight() + (2.0f *
ImGui::GetStyle().FramePadding.y)

```

You can see how ImGui and Font Awesome are added on top. Following this, is the definition of crucial settings, including the path for loading the font file for text and icons. Further down, a set of macros is introduced to facilitate the dynamic selection of the active font and obtain information about line height with and without padding.

12.2.2 Event Types

Moving on to the "**Events.h**" header file, it plays a pivotal role in defining the essential events within the editor. Currently, there is one event type, specifically crafted to inform all listeners about the presently selected entity.

Listing 12.3: Context/Events.h

```
#pragma once
#include "Helpers.h"

struct SelectEvent
{
    EMPLY_INLINE SelectEvent(EntityID entity) :
        EnttID(entity)
    {}

    EntityID EnttID;
};
```

12.2.3 Input Fields

We need to be able to interact with different UI input elements such number inputs, text inputs, combos inputs, etc. The header file "**Inputs.h**", provide a definition for all these different input types.

Listing 12.4: Context/Inputs.h

```
#include "Helpers.h"

// starts input field
EMPY_INLINE void BeginInput(const char* label)
{
    ImGui::PushID(label);
    ImGui::Columns(2, NULL, false);
    ImGui::SetColumnWidth(0, LABEL_SPACING);
    ImGui::AlignTextToFramePadding();
    ImGui::Text("%s", label);
    ImGui::NextColumn();
    ImGui::PushItemWidth(-1);
}

// end input field
EMPY_INLINE void EndInput()
{
    ImGui::EndColumns();
    ImGui::PopID();
}

// -----
// shows bool input field
EMPY_INLINE bool InputBool(const char* label, bool* value)
{
    BeginInput(label);
    bool hasChanged = ImGui::Checkbox("##", value);
    EndInput();
    return hasChanged;
}

// shows float input field
EMPY_INLINE bool InputFloat(const char* label, float* value)
{
```

```

    BeginInput(label);
    bool hasChanged = ImGui::InputFloat("##", value);
    EndInput();
    return hasChanged;
}

// shows vec3 input field
EMPY_INLINE bool InputVec3(const char* label, glm::vec3* value)
{
    BeginInput(label);
    bool hasChanged = ImGui::InputFloat3("##", &value->x);
    EndInput();
    return hasChanged;
}

// shows bool input field
EMPY_INLINE bool InputColor(const char* label, float* value)
{
    BeginInput(label);
    bool hasChanged = ImGui::ColorEdit3("##", value);
    EndInput();
    return hasChanged;
}

// shows button
EMPY_INLINE bool InputButton(const char* label, ImVec2& size
= ImVec2(0,0))
{
    ImGui::PushStyleColor(ImGuiCol_Button, ImVec4(0.1f, 0.0f,
1.0f, 1.0f));
    ImGui::PushStyleVar(ImGuiStyleVar_FrameRounding, 1.0f);
    bool clicked = ImGui::ButtonEx(label, size);
    ImGui::PopStyleColor();
    ImGui::PopStyleVar();
    return clicked;
}

```

```

}

// shows combo input field
EMPY_INLINE bool InputCombo(const char* label, const
std::vector<const char*>& combos)
{
    BeginInput(label);
    bool hasChanged = false;
    static char* preview = "Select";

    if (ImGui::BeginCombo("##", preview))
    {
        for (auto i = 0; i < combos.size(); i++)
        {
            if (ImGui::Selectable(combos[i],
!strcmp(combos[i], preview)))
            {
                strcpy(preview, combos[i]);
                hasChanged = true;
            }
        }
        ImGui::EndCombo();
    }
    EndInput();

    return hasChanged;
}

// shows text input field
EMPY_INLINE bool InputText(const char* label, char* value,
const char* hint = nullptr, size_t size = 32)
{
    BeginInput(label);
    bool hasChanged = ImGui::InputTextEx("##", hint, value,
size, ImVec2(0,0), ImGuiInputTextFlags_EnterReturnsTrue);
    EndInput();
}

```

```
    return hasChanged;  
}
```

Observe that these input elements collectively utilize the `BeginInput()` and `EndInput()` functions. These functions play a crucial role in establishing a consistent layout, featuring a two-column structure with the label on one side and the input field on the other. The visual outcome of this arrangement is illustrated in (Figure [12.8](#)).

Near	0.300
Far	1000.000

Figure 12.8: ImGui Root Directory

You can see how we have the label on one side and the input field on the other. The constant `LABEL_SPACING` defines the spacing between the two columns.

12.2.4 Widget Interface

Our objective is to centralize the behavior of widgets within our application, necessitating the creation of an interface that encompasses all potential widgets. These widgets include elements like windows, panels, and more. The encapsulation of this functionality is achieved through the **"Widget.h"** header file.

Listing 12.5: Context/Widget.h

```
#pragma once
#include "Events.h"
#include "Inputs.h"

// forward decl.
struct GuiContext;

// widget interface
struct IWidget
{
    EMPY_INLINE IWidget(GuiContext*) { };
    EMPY_INLINE virtual ~IWidget() = default;
    EMPY_INLINE virtual void OnSelect(Entity) {};
    EMPY_INLINE virtual void OnShow(GuiContext*) {};
    EMPY_INLINE virtual void SetTitle(const char*) {};
};

// widget type definition
using Widget = std::unique_ptr<IWidget>;
```

The widget interface is straightforward, with a forward declaration of the `GuiContext`. This forward declaration is essential as the widget is intended for use within the `GuiContext`. This approach mitigates potential issues related to recursive file inclusion, preventing compiler confusion and ensuring a smooth integration of the widget functionality within the `GuiContext`. The `OnSelect()` function is meant to be the callback function when the select event is triggered.

12.2.5 Context Interface

The context interface is basically where every thing we just introduced come together to enable a wide range of dynamic behaviours. This class or structure as you can see in the following code snippet inherit from the the `AppInterface` to give us the ability to access all the core functionalities of the engine. The first on being the application window provide by the function `GetWindowHandle()`.

Listing 12.6: Context/Context.h

```
#pragma once
#include "Widget.h"

struct GuiContext : AppInterface
{
    EMPTY_INLINE virtual ~GuiContext()
    {
        ImGui_ImplOpenGL3_Shutdown();
        ImGui_ImplGlfw_Shutdown();
        ImGui::DestroyContext();
        glfwTerminate();
    }

    EMPTY_INLINE virtual void OnGuiStart() { }

    EMPTY_INLINE virtual void OnGuiFrame() { }

    EMPTY_INLINE void OnStart() override final
    {
        // init imgui context
        IMGUI_CHECKVERSION();
        ImGui::CreateContext();
        ImGuiIO& io = ImGui::GetIO(); (void)io;
    }
}
```

```

        io.ConfigWindowsMoveFromTitleBarOnly = true;
        io.ConfigFlags |= ImGuiConfigFlags_DockingEnable;
        io.ConfigFlags |= ImGuiConfigFlags_NavEnableGamepad;
        io.ConfigFlags |= ImGuiConfigFlags_NavEnableKeyboard;

        ImGui_ImplOpenGL3_Init(SHADER_VERSION);
        ImGui_ImplGlfw_InitForOpenGL(this->GetWindowHandle(),
true);

        ImFontConfig fontConfig;
        fontConfig.MergeMode = true;
        fontConfig.PixelSnapH = true;
        static const ImWchar iconRange[] = { ICON_MIN_FA,
ICON_MAX_FA, 0 };

        // regular font and icon
        io.Fnts->AddFontFromFileTTF(FONT_FILE,
REGULAR_FONT_SIZE);
        io.Fnts->AddFontFromFileTTF(ICONT_FONT,
REGULAR_FONT_SIZE, &fontConfig, iconRange);

        // small font and icon
        io.Fnts->AddFontFromFileTTF(FONT_FILE,
SMALL_FONT_SIZE);
        io.Fnts->AddFontFromFileTTF(ICONT_FONT,
SMALL_FONT_SIZE, &fontConfig, iconRange);

        // set imgui style
        ImGui::StyleColorsDark();

        // attach event callback
        AttachCallback<SelectEvent>([this] (auto e)
{
    for(auto& window : m_Windows)
        window->OnSelect(ToEntt<Entity>(e.EnttID));
});

```

```
// interface start
OnGuiStart();
}

EMPTY_INLINE void OnUpdate() override final
{
    // Initialize ImGui for the frame
    ImGui_ImplOpenGL3_NewFrame();
    ImGui_ImplGlfw_NewFrame();
    ImGui::NewFrame();

    // Set up the main viewport
    static auto viewport = ImGui::GetMainViewport();
    ImGui::SetNextWindowViewport(viewport->ID);
    ImGui::SetNextWindowSize(viewport->Size);
    ImGui::SetNextWindowPos(viewport->Pos);

    // Define main window flags
    static auto flags =
ImGuiWindowFlags_NoBringToFrontOnFocus |
    ImGuiWindowFlags_NoTitleBar |
ImGuiWindowFlags_NoCollapse |
    ImGuiWindowFlags_NoResize |
ImGuiWindowFlags_NoNavFocus |
    ImGuiWindowFlags_NoBackground |
ImGuiWindowFlags_NoMove |
    ImGuiWindowFlags_MenuBar;

    // Set window style
    ImGui::PushStyleVar(ImGuiStyleVar_WindowPadding,
ImVec2(0, 0));
    ImGui::PushStyleColor(ImGuiCol_WindowBg, ImVec4(0, 0,
0, 1));
    ImGui::PushStyleVar(ImGuiStyleVar_WindowBorderSize,
0);
}
```

```

ImGui::PushStyleVar(ImGuiStyleVar_WindowRounding, 0);

    // Begin the main window
    ImGui::Begin("MainWindow", NULL, flags);
    {
        // Set up the main dockspace
        ImGui::DockSpace(ImGui::GetID("MainDockspace"),
ImVec2(0, 0),
            ImGuiDockNodeFlags_PassthruCentralNode);
        ImGui::PopStyleColor();
        ImGui::PopStyleVar(3);

        // Iterate through each window in the vector and
call OnShow
        for(auto& window : m_Windows)
        {
            window->OnShow(this);
        }

        // Show ImGui Demo Window for debugging purposes
        ImGui::ShowDemoWindow();

        // Interface update
        OnGuiFrame();
    }
    ImGui::End();

    // Render ImGui draw data
    //glClear(GL_COLOR_BUFFER_BIT);
    ImGui::Render();

    ImGui_ImplOpenGL3_RenderDrawData(ImGui::GetDrawData());
}

template<typename T, typename... Args>
EMPTY_INLINE void AttachWindow(Args&&... args)

```

```

    {

        EMPTY_STATIC_ASSERT(std::is_base_of<IWidget,
T>::value);

        auto window = std::make_unique<T>(this,
std::forward<Args>(args)...);

        m_Windows.push_back(std::move(window));
    }

    template<typename T, typename... Args>
EMPTY_INLINE auto CreateWidget(Args&&... args)
{
    EMPTY_STATIC_ASSERT(std::is_base_of<IWidget,
T>::value);

    auto widget = std::make_unique<T>(this,
std::forward<Args>(args)...);

    return widget;
}

private:
    std::vector<Widget> m_Windows;
};

```

As evident, this GUI context essentially serves as an application layer. Within the `OnStart()` virtual function of the `AppInterface`, our initial step involves `ImGui` initialization with an OpenGL backend. Following this, we proceed to load the font files utilizing the constants defined in the "Helpers.h" header file. Looking at the member variable section, a vector of widgets is established, designed to house all windows created in our editor. Given that these windows are considered widgets, a callback is attached to the

`SelectEvent` to ensure each window receives notification when a select event is triggered.

You will note that the layer also provides additional virtual function that can be implemented by any class deriving from this context. The `OnUpdate()` function just as the `OnStart()` function is overridden from the `AppInterface`. At the beginning of the function, ImGui is initialized for the current frame using `ImGui_ImplOpenGL3_NewFrame()`, `ImGui_ImplGlfw_NewFrame()`, and `ImGui::NewFrame()`.

Subsequently, the code sets up the main viewport and configures the main window based on this viewport's dimensions. Various ImGui window flags are defined to customize the behavior of the main window. Window style settings are adjusted using `ImGui::PushStyleVar()` and `ImGui::PushStyleColor()`, modifying attributes such as padding, background color, border size, and rounding. The main window is then initiated with `ImGui::Begin()` and encapsulates several ImGui operations. The main dockspace is established using `ImGui::DockSpace()` to enable docking capabilities for various windows. The code iterates through a vector of windows (`m_Windows`) and calls the `OnShow()` function for each window, allowing for individualized rendering within the main window.

Additionally, the ImGui Demo Window is displayed using `ImGui::ShowDemoWindow()` for debugging purposes. The

`OnGuiFrame()` function is invoked to handle interface updates specific to the application. Finally, the ImGui draw data is rendered with `ImGui::Render()`, and the OpenGL-specific rendering is performed using

`ImGui_ImplOpenGL3_RenderDrawData()`. This entire process is crucial for updating and rendering the GUI elements within the application.

Window Handle Getter

Make sure the `GetWindowHandle()` function is implemented in your `AppInterface` so that ImGui can get the GLFW window to initialize itself. Here the code:

Listing 12.7: Graphics/Renderer.h

```
/* ... same as before ... */

struct AppInterface
{
    EMPLY_INLINE GLFWwindow* GetWindowHandle()
    {
        return m_Context->Window->Handle();
    }

    EMPLY_INLINE uint32_t GetSceneFrame()
    {
        return m_Context->Renderer->GetFrame();
    }
};
```

You will also note the additional function to retrieve the frame buffer attachment from the renderer. This will help us render the scene to an ImGui window instead of the default on-screen frame buffer.

12.3 Creating Windows

Now, let's proceed to create various windows to construct our editor, commencing with the most prominent one: the Viewport.

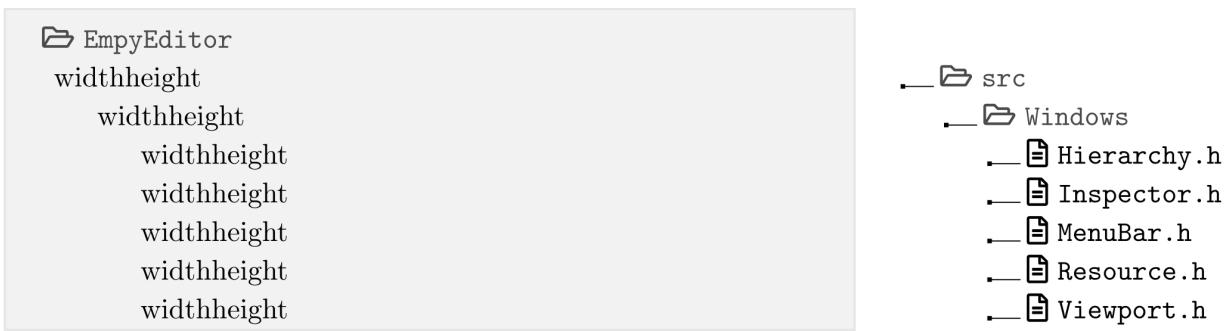


Figure 12.9: Windows Directory

12.4 Viewport Window

The viewport simply show the scene. As you know, we have been rendering the scene in the default frame buffer of the screen. We now want to capture the rendered scene and show it using ImGui instead. (Figure 12.10) is a preview of the viewport.



Figure 12.10: Viewport Window

12.4.1 Rendering to Buffer

Currently, our concluding scene is directly rendered into the default frame buffer. This scene comprises both the color attachment and the bloom attachment. However, an impediment arises: accessing the default frame buffer's texture attachment for rendering with ImGui is unfeasible. Hence, the solution involves establishing an additional frame buffer, dedicated to storing the final scene frame. This new frame buffer incorporates a texture attachment, providing accessibility within our editor. This can be done inside of the final shader class.

Listing 12.8: Graphics/Sahders/Final.h

```
#pragma once
#include "Shader.h"
```

```

#include "../Utilities/Quad.h"

namespace EmPy
{
    struct FinalShader : Shader
    {
        EMPY_INLINE FinalShader(const std::string& filename,
int32_t width, int32_t height) :
            Shader(filename)
        {
            u_Bloom = glGetUniformLocation(m_ShaderID,
"u_bloom");
            u_Map = glGetUniformLocation(m_ShaderID,
"u_map");
            CreateBuffer(width, height);
            m_Quad = CreateQuad2D();
        }

        EMPY_INLINE ~FinalShader()
        {
            glDeleteTextures(1, &m_Final);
            glDeleteFramebuffers(1, &m_FBO);
        }

        EMPY_INLINE void Render(uint32_t map, uint32_t bloom,
bool useFBO)
        {
            glBindFramebuffer(GL_FRAMEBUFFER, (useFBO) ? 0 :
m_FBO);
            glClear(GL_COLOR_BUFFER_BIT);
            glClearColor(0, 0, 0, 1);
            glUseProgram(m_ShaderID);

            // set color map
            glActiveTexture(GL_TEXTURE0);
            glBindTexture(GL_TEXTURE_2D, map);
        }
    };
}

```

```

        glUniform1i(u_Map, 0);

        // set bloom map
        glActiveTexture(GL_TEXTURE1);
        glBindTexture(GL_TEXTURE_2D, bloom);
        glUniform1i(u_Bloom, 1);

        // render quad
        m_Quad->Draw(GL_TRIANGLES);
        glUseProgram(0);

        // bind default fbo
        glBindFramebuffer(GL_FRAMEBUFFER, 0);
    }

EMPTY_INLINE void Resize(int32_t width, int32_t
height)
{
    glBindTexture(GL_TEXTURE_2D, m_Final);
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA16F,
width, height, 0, GL_RGBA, GL_FLOAT, NULL);
    glBindTexture(GL_TEXTURE_2D, 0);
}

EMPTY_INLINE uint32_t GetMap()
{
    return m_Final;
}

private:
EMPTY_INLINE void CreateBuffer(int32_t width, int32_t
height)
{
    // create frame buffer
    glGenFramebuffers(1, &m_FBO);
    glBindFramebuffer(GL_FRAMEBUFFER, m_FBO);
}

```

```

        // create attachment
        glGenTextures(1, &m_Final);
        glBindTexture(GL_TEXTURE_2D, m_Final);
        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_S,
GL_CLAMP_TO_EDGE);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T,
GL_CLAMP_TO_EDGE);
        glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA16F, width,
height, 0, GL_RGBA, GL_FLOAT, NULL);
        glFramebufferTexture2D(GL_FRAMEBUFFER,
GL_COLOR_ATTACHMENT0, GL_TEXTURE_2D, m_Final, 0);

        // check frame buffer
        if (glCheckFramebufferStatus(GL_FRAMEBUFFER) !=
GL_FRAMEBUFFER_COMPLETE)
{
    EMPLY_ERROR("glCheckFramebufferStatus()
Failed!");
}

        // unbind frame buffer
        glBindFramebuffer(GL_FRAMEBUFFER, 0);
}

private:
    uint32_t m_Final = 0u;
    uint32_t m_FBO = 0u;

    uint32_t u_Bloom = 0u;
    uint32_t u_Map = 0u;

```

```
    Quad2D m_Quad;  
};  
}
```

In this code snippet, a new frame buffer (`m_FBO`) has been introduced, featuring a texture attachment named `m_Final`. This frame buffer serves as the designated storage for the final scene frame. The `Render()` function, essentially a renamed version of the `Show()` function, remains largely unchanged. Noteworthy, however, is the addition of a variable determining whether the final scene should be rendered to the screen or the newly established frame buffer. This demarcation enables the distinction between editor mode and game mode. Important here is the `GetMap()` function. It returns the texture identifier of the attachment which can be used in ImGui to render the frame.

Renderer Update

We then have to also apply this to the graphics renderer class. You can see in the following code snippet that the `GetFrame()` returns the Attachment of the frame buffer found in the final shader. The function `ShowFrame()` now also takes a boolean which tells the final shader how and where to render the final frame.

Listing 12.9: Graphics/Renderer.h

```

/* ... same as before ... */

struct GraphicsRenderer
{
    EMPTY_INLINE void ShowFrame(bool useFBO)
    {
        glViewport(0, 0, m_Frame->Width(), m_Frame-
>Height());
        m_Final->Render(m_Frame->GetTexture(), m_Bloom-
>GetMap(), useFBO);
    }

    EMPTY_INLINE uint32_t GetFrame()
    {
        //return m_Frame->GetTexture();
        return m_Final->GetMap();
    }
};

```

Application Update

We then have to also reflect this behaviour in the application main loop. This can be done by providing the function `RunContext()` with an argument which tells if the application is currently in editor mode or game mode.

Listing 12.10: Application/Application.h

```

/* ... same as before ... */

EMPTY_INLINE void RunContext(bool showFrame = false)
{

```

```

// application main loop
while (m_Context->Window->PollEvents ())
{
    /* ... */

    for (auto layer : m_Context->Layers)
    {
        layer->OnUpdate ();
    }

    // show only for game
    m_Context->Renderer->ShowFrame (showFrame);
}
}

```

12.4.2 Creating the Viewport

Having established this framework, creating the viewport window becomes a straightforward task, as exemplified in the subsequent code. Within the `OnShow()` function, we effortlessly generate an ImGui window that encompasses a child window. Subsequently, we utilize the **m_Frame**, obtained from the context in the constructor, to render the scene within the ImGui window. Ensuring an appropriate fit, we retrieve the size of the current ImGui window and employ it for rendering the frame.

Listing 12.11: EmPyEditor/src/Windows/Viewport.h

```

#pragma once
#include "Context/Context.h"

```

```

struct ViewportWindow : IWidget
{
    EMPLY_INLINE ViewportWindow(GuiContext* context) :
IWidget(context)
    {
        m_Frame = (ImTextureID)context->GetSceneFrame();
    }

    EMPLY_INLINE void OnShow(GuiContext* context) override
    {
        if(ImGui::Begin(ICON_FA_IMAGE "\tViewport"))
        {
            ImGui::BeginChild("Frame");
            {
                // show scene frame
                const ImVec2 size =
                ImGui::GetWindowContentRegionMax();
                ImGui::Image(m_Frame, size, ImVec2(0, 1), ImVec2(1,
0));
            }
            ImGui::EndChild();
        }
        ImGui::End();
    }

    EMPLY_INLINE void OnSelect(Entity entity) override
    {
        // nothing done
    }

private:
    ImTextureID m_Frame;
    ImVec2 m_Viewport;
};

```

12.4.3 Showing the Viewport

Now, let's promptly integrate this window into the editor to observe its appearance. In the ensuing code snippet, you can observe how the viewport window is incorporated into the editor. The GUI context's built-in function for adding windows is employed to seamlessly include the viewport window in the editor.

Listing 12.12: EmPyEditor/src/Windows/Viewport.h

```
#include "Windows/Inspector.h"
#include "Windows/Hierarchy.h"
#include "Windows/Resource.h"
#include "Windows/Viewport.h"
#include "Windows/MenuBar.h"

struct Editor : GuiContext
{
    EMPY_INLINE void OnGuiStart()
    {
        AttachWindow<ViewportWindow>();
    }
};

int32_t main(int32_t argc, char** argv)
{
    auto app = new Application();
    app->AttachLayer<Editor>();
    app->RunContext(false);
    EMPY_DELETE(app);
    return 0;
}
```

If you compile and run your code you will be able to see this:



Figure 12.11: Viewport and Demo Window

12.4.4 Scroll, Drag and Resize

If you try to resize the viewport window using the border, you will note that the scene looks weird. This is because the frame buffer is not resized accordingly. We want the backend frame buffer to have the same size and the ImGui window.

Resizing the Frame

Going back to the `ViewportWindow`, we can simply post a `WindowResizeEvent` event whenever the size of the ImGui window changes. This is done by updating the `OnShow()` function as depicted below.

Listing 12.13: EmpyEditor/src/Windows/Viewport.h

```
EMPY_INLINE void OnShow(GuiContext* context) override
{
    if(ImGui::Begin(ICON_FA_IMAGE "\tViewport"))
    {
        ImGui::BeginChild("Frame");
        {
            // show scene frame
            const ImVec2 size =
            ImGui::GetWindowContentRegionMax();
            ImGui::Image(m_Frame, size, ImVec2(0, 1),
            ImVec2(1, 0));

            // handle resizing
            if(m_Viewport.x != size.x || m_Viewport.y != size.y)
            {
                int32_t width = (int32_t)size.x, height =
                (int32_t)size.y;
                context->PostEvent<WindowResizeEvent>(width,
                height);
                m_Viewport = size;
            }
        }
        ImGui::EndChild();
    }
    ImGui::End();
}
```

Leveraging the member variable `m_Viewport`, we can simply compare the current window size with the previous one to discern whether any changes have occurred. In the event of

a change, a resize event is triggered, subsequently captured by the application to appropriately resize the frame buffer. If you now run your application again, you can see how the frame will always match the window's size.

Dragging and Scrolling

An indispensable feature of an editor is the capability to utilize mouse inputs for controlling the view position and angle of the camera. This functionality can be seamlessly incorporated, as demonstrated in the updated `OnShow()` function below:

Listing 12.14: EmPyEditor/src/Windows/Viewport.h

```
EMPY_INLINE void OnShow(GuiContext* context) override
{
    if(ImGui::Begin(ICON_FA_IMAGE "\tViewport"))
    {
        ImGui::BeginChild("Frame");
        {
            // show scene frame
            const ImVec2 size =
                ImGui::GetWindowContentRegionMax();
            ImGui::Image(m_Frame, size, ImVec2(0, 1),
                        ImVec2(1, 0));

            // handle resizing
            if(m_Viewport.x != size.x || m_Viewport.y != size.y)
            {

```

```

                int32_t width = (int32_t)size.x, height =
(int32_t)size.y;
                context->PostEvent<WindowResizeEvent>(width,
height);
                m_Viewport = size;
            }

// handling scrolling and dragging
auto& io = ImGui::GetIO();

if(ImGui::IsWindowHovered())
{
    // handle zoom in/out
    if(io.MouseWheel != 0)
    {
        float sensibility = 50;

context->EnttView<Entity, CameraComponent>
([&] (auto entity, auto& comp)
{
    auto& transform = entity.template
Get<TransformComponent>().Transform;
    transform.Translate.z -=
(io.MouseWheel * ioDeltaTime * sensibility);
} );
    }
}

// handle dragging

if(ImGui::IsMouseDragging(ImGuiMouseButton_Left
))
{
    float sensibility = 10;

context->EnttView<Entity, CameraComponent>
([&] (auto entity, auto& comp)

```

```

        {
            auto& transform = entity.template
Get<TransformComponent>().Transform;
            transform.Rotation.x +=
(io.MouseDelta.y * io.deltaTime * sensibility);
            transform.Rotation.y +=
(io.MouseDelta.x * io.deltaTime * sensibility);
        });
    }

}
ImGui::EndChild();
}

ImGui::End();
}

```

We harness the inherent capabilities of ImGui to retrieve mouse inputs, employing them to dynamically update the properties of the camera. See result in (Figure [12.12](#)).

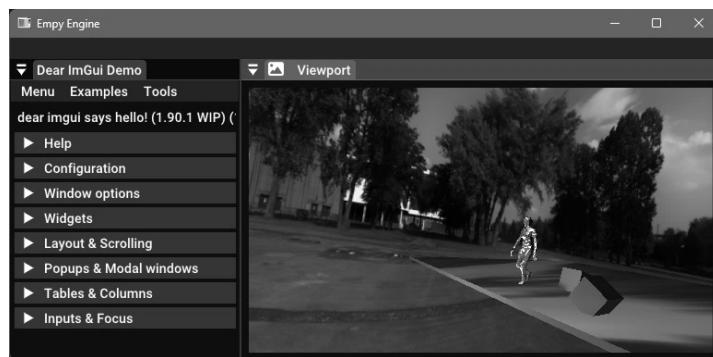


Figure 12.12: Dragging and Scrolling

12.5 Hierarchy Window

The hierarchy window plays a pivotal role in visually representing the relationships between entities within the editor. An entity in some instances can be a child to another entity. This hierarchical organization is instrumental in depicting the structural dependencies and parent-child relationships that exist among various entities within the scene. The following code snippet shows the implementation of this window.

Listing 12.15: EmPyEditor/src/Windows/Hierarchy.h

```
#pragma once
#include "Context/Context.h"

struct HierarchyWindow : IWidget
{
    EMPY_INLINE HierarchyWindow(GuiContext* context) :
    IWidget(context)
    {
        // empty
    }

    EMPY_INLINE void OnShow(GuiContext* context) override
    {
        if(ImGui::Begin(ICON_FA_CUBES "\tHierarchy"))
        {
            context->EnttView<Entity, InfoComponent>([&] (auto
entity, auto& info)
            {
                // check if current entity is selected
            });
        }
    }
}
```

```

    bool isSelected = (m_Selected.ID() == entity.ID());

    // makes each selectable unique
    ImGui::PushID((int32_t)entity.ID());

    // compute selectable label
    std::string label = ICON_FA_CUBE "\t" + info.Name;

    // special label with icons
    if(entity.template Has<DirectLightComponent>())
        label = ICON_FA_SUN "\t" + info.Name;
    if(entity.template Has<CameraComponent>())
        label = ICON_FA_VIDEO "\t" + info.Name;
    if(entity.template Has<RigidBodyComponent>())
        label = ICON_FA_RING "\t" + info.Name;

    // show entity selectable
    if(ImGui::Selectable(label.c_str(), &isSelected))
    {
        context->PostEvent<SelectEvent>(entity);
    }

    // w are required to pop
    ImGui::PopID();
} );
}

ImGui::End();
}

EMPTY_INLINE void OnSelect(Entity entity) override
{
    m_Selected = entity;
}

private:

```

```
    Entity m_Selected;  
};
```

The primary focus in this context is the `OnShow()` function, where we iterate through all the entities present in the scene and render an `ImGui::Selectable` representing each entity. To ensure clarity and prevent confusion when interacting with these elements, we utilize the entity identifier to push an ImGui ID, guaranteeing the uniqueness of each element. When rendering UI elements with ImGui, it's crucial to avoid identical labels or identifiers, as they would be treated as a single entity, leading to potential data loss.

Furthermore, this implementation demonstrates the integration of Font Awesome to enhance the visual appeal of the UI elements by associating them with distinctive icons. Additionally, special labels are generated based on specific components of the entity, such as a light source or a camera. Upon clicking a UI element, a `SelectEvent` is dispatched, captured by the `GuiContext`, and subsequently disseminated to all registered windows. This mechanism ensures seamless communication and synchronization among various windows when entities are selected.

Below (Figure 12.13) is a resulting image:



Figure 12.13:Hierarchy
Window

12.6 MenuBar Window

Additionally, we aim to incorporate a menubar that offers supplementary functionalities within our application. The subsequent code snippet illustrates how this is achieved.

Listing 12.16: EmPyEditor/src/Windows/MenuBar.h

```
#pragma once
#include "Context/Context.h"

struct MenuBarWindow : IWidget
{
    EMPY_INLINE MenuBarWindow(GuiContext* context) :
        IWidget(context)
    {
        // nothing
    }

    EMPY_INLINE void OnShow(GuiContext* context) override
    {
        ImGui::PushStyleVar(ImGuiStyleVar_ItemSpacing, ImVec2(6,
6));
    }

    if (ImGui::BeginMenuBar())
    {
        if (ImGui::BeginMenu("File"))
        {
            if (ImGui::MenuItem(ICON_FA_FILE " Open
```

```

Project", "Ctrl+O")) { }
    if (ImGui::MenuItem(ICON_FA_STORE " Save
Scene", "Ctrl+S")) { }
    if (ImGui::MenuItem(ICON_FA_DOOR_CLOSED " Exit",
"Alt+F4")) { }
    ImGui::EndMenu();
}
if (ImGui::BeginMenu("Scene"))
{
    if (ImGui::MenuItem(ICON_FA_FORWARD " Add
Entity"))
    {
        auto entity = context->CreateEntt<Entity>();
        entity.template Attach<TransformComponent>();
        entity.template Attach<InfoComponent>().Name =
"New
Entity";
    }

    if (ImGui::MenuItem(ICON_FA_FORWARD " Undo",
"CTRL+Z")) { }
    if (ImGui::MenuItem(ICON_FA_BACKWARD " Redo",
"CTRL+Y")) { }
    ImGui::Separator();
    if (ImGui::MenuItem(ICON_FA_SCISSORS " Cut",
"CTRL+X")) { }
    if (ImGui::MenuItem(ICON_FA_CLONE " Copy",
"CTRL+C")) { }
    if (ImGui::MenuItem(ICON_FA_PASTE " Paste",
"CTRL+V")) { }
    if (ImGui::MenuItem(ICON_FA_TRASH " Delete",
"Delete")) { }
    ImGui::EndMenu();
}
if (ImGui::BeginMenu("Settings"))
{
    if (ImGui::MenuItem(ICON_FA_PALETTE " " "
```

```

Theme" )) { }

        ImGui::EndMenu();

    }

    if (ImGui::BeginMenu("Extra"))

    {

        if (ImGui::MenuItem(ICON_FA_INFO " Help",
"Ctrl+H")) {}

        if (ImGui::MenuItem(ICON_FA_QUESTION " About)) {}

        ImGui::EndMenu();

    }

    ImGui::EndMenuBar();

}

ImGui::PopStyleVar();

}

};


```

This code effortlessly utilizes ImGui's built-in functionalities to construct a menu bar for our application. Upon reviewing the `GuiContext::OnUpdate()` function, you'll observe the inclusion of a flag named `ImGuiWindowFlags_MenuBar` among the flags designated for the main window. This specific flag signals ImGui to include a menubar in the viewport. Consequently, you'll notice an allocated space at the top of your window reserved for the menubar. Important to note here is the menu item we have added which helps us create an entity on the go.

Add this menubar to your editor layer to get the following result:



Figure 12.14: Hierarchy Window

Try to create an entity using the "Add Entity" menu item!

12.7 Inspector Window

The primary purpose of the inspector window is to display all the components and properties of the presently selected entity. Consequently, this window necessitates responsiveness to the select event. The following code snippet outlines the implementation of this window.

Listing 12.17: EmpyEditor/src/Windows/Inspector.h

```
#pragma once
#include "Controls/Camera.h"
#include "Controls/EnttInfo.h"
#include "Controls/Transform.h"
#include "Controls/DirectLight.h"

struct InspectorWindow : IWidget
{
    EMPY_INLINE InspectorWindow(GuiContext* context) :
    IWidget(context)
```

```

    {
        m_Widgets.push_back(context->CreateWidget<InfoControl>
());
        m_Widgets.push_back(context-
>CreateWidget<TransformControl>());
        m_Widgets.push_back(context->CreateWidget<CameraControl>
());
        m_Widgets.push_back(context-
>CreateWidget<DirectLightControl>());
    }

    EMPLY_INLINE void OnShow(GuiContext* context) override
{
    if(ImGui::Begin(ICON_FA_CIRCLE_INFO "\tInspector"))
    {
        for(auto& widget : m_Widgets)
            widget->OnShow(context);
    }
    ImGui::End();
}

EMPLY_INLINE void OnSelect(Entity entity) override
{
    for(auto& widget : m_Widgets)
        widget->OnSelect(entity);
}

private:
    std::vector<Widget> m_Widgets;
};

```

It is evident that there are several components in this code that have not yet been created, specifically the controls. In our engine, controls refer to UI elements or widgets

designed for components. These controls empower us to interact with the properties of components in real-time. This leads us to introduce the "Controls" directory (Figure 12.15).

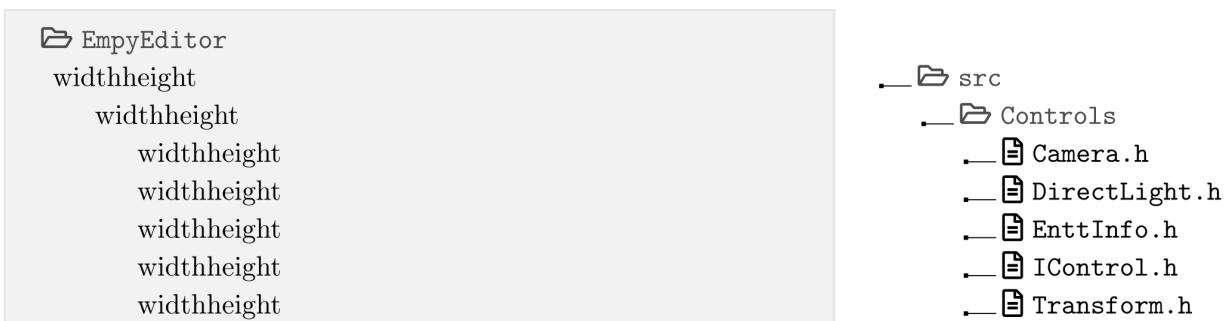


Figure 12.15: Controls Directory

Contrary to windows, controls the directly inherit from widget. We need to implement an in-between interface to make these controls more dynamic. this is done within the "**IControl.h**" header file.

12.7.1 Control Interface

Following is the code snippet which provides the implementation of the interface for all the controls:

Listing 12.18: EmpyEditor/src/Controls/IControl.h

```
#pragma once
#include "../Context/Context.h"
```

```

template <typename Component>
struct IControl : public IWidget
{
    EMPY_INLINE virtual ~IControl() = default;

    EMPY_INLINE IControl(GuiContext* context) :
        IWidget(context)
    {}

    EMPY_INLINE void OnSelect(Entity entity) override
    {
        m_Entity = entity;
    }

    EMPY_INLINE void SetTitle(const char* title) override
    {
        m_Title = title;
    }

    EMPY_INLINE void OnShow(GuiContext* context) override
    {
        if (!m_Entity.template Has<Component>()) { return; }

        ImGui::PushID(m_Title);
        static const auto flags =
            ImGuiTreeNodeFlags_DefaultOpen |
            ImGuiTreeNodeFlags_OpenOnArrow |
            ImGuiTreeNodeFlags_OpenOnDoubleClick |
            ImGuiTreeNodeFlags_SpanFullWidth |
            ImGuiTreeNodeFlags_AllowItemOverlap;

        ImGui::PushStyleVar(ImGuiStyleVar_FramePadding,
                           ImVec2(5.0f, 4.0f));
        bool isCollapsed = ImGui::CollapsingHeader(m_Title,
                                                 flags);
        ImGui::PopStyleVar();
    }
};

```

```
// activate small font
USE_SMALL_FONT();

ImGui::SameLine();
ImGui::SetCursorPosY(ImGui::GetCursorPosY() + 2.0f);
ImGui::SetCursorPosX(ImGui::GetWindowWidth() -
30.0f);

if (InputButton(ICON_FA_GEAR))
{
    ImGui::OpenPopup(m_Title);
}

// Popup
if (ImGui::BeginPopup(m_Title))
{
    ImGui::Text(m_Title);
    ImGui::Separator();
    OnMenu(context, m_Entity);
    ImGui::EndPopup();
}

// Content
if (isCollapsed)
{
    ImGui::Spacing();
    ImGui::Indent(5.0f);
    OnBody(context, m_Entity);
    ImGui::Spacing();
    ImGui::Unindent(5.0f);
}

ImGui::PopID();

// switch back to regular font
```

```

        USE_REGULAR_FONT();
    };

protected:
    EMPLY_INLINE void virtual OnMenu(GuiContext*, Entity&) { }
    EMPLY_INLINE void virtual OnBody(GuiContext*, Entity&) { }

private:
    const char* m_Title;
    Entity m_Entity;
};

```

Again the main focus is on the `OnShow()` function. This function is basically creating an ImGui collapsing header widget with a button on the side to provide additional actions as depicted in (Figure 12.16).

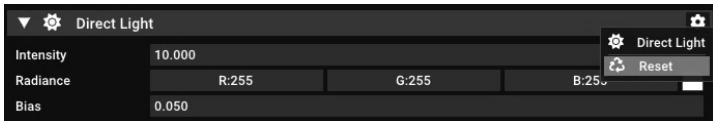


Figure 12.16: Example Control

The button on the side triggers the generation of a popup menu with additional actions. Implementing these actions involves overriding the `OnMenu()` virtual member function. The `OnBody()` function is where the actual content of the control will be implemented in the subsequent control class. Notably, this interface incorporates a template type, representing the component type. This template approach

facilitates swift verification of whether the currently selected entity possesses the specified component. If not, the `OnShow()` function promptly returns without displaying anything.

Let us now implement the control widgets for our components.

12.7.2 Transform Control

True to its name, the transform control is a widget that empowers us to visualize and modify the properties of an entity's transform component.

Listing 12.19: EmPyEditor/src/Controls/Transform.h

```
#pragma once
#include "IControl.h"

struct TransformControl : IControl<TransformComponent>
{
    EMPY_INLINE TransformControl(GuiContext* context) :
        IControl(context)
    {
        SetTitle(ICON_FA_LOCATION_ARROW "\tTransform");
    }

    EMPY_INLINE void OnBody(GuiContext* context, Entity&
entity)
    {
        auto& data = entity.template Get<TransformComponent>()
            .Transform;
```

```

        InputVec3("Translate", &data.Translate);
        InputVec3("Rotation", &data.Rotation);
        InputVec3("Scale", &data.Scale);
    }

    EMPLY_INLINE void OnMenu(GuiContext* context, Entity&
entity)
{
    if (ImGui::Selectable(ICON_FA_RECYCLE "\tReset"))
    {
        entity.template Get<TransformComponent>() = {};
    }
}
};


```

The implementation of the `OnMenu()` virtual function is evident, featuring a reset action that restores the transform's values to their default settings. On the other hand, the `OnBody()` function straightforwardly establishes input fields for each property of the transformation. See (Figure 12.17) for visual result.

	0.000	0.000	0.000
Translate	0.000	0.000	0.000
Rotation	0.000	1.000	-1.000
Scale	1.000	1.000	1.000

Figure 12.17: Example Control

12.7.3 Camera Control

Adhering to a similar pattern, we can create the camera control, as demonstrated in the following code snippet.

Listing 12.20: EmPyEditor/src/Controls/Camera.h

```
#pragma once

#include "IControl.h"

struct CameraControl : IControl<CameraComponent>
{
    EMPY_INLINE CameraControl(GuiContext* context) :
        IControl(context)
    {
        SetTitle(ICON_FA_VIDEO "\tCamera");
    }

    EMPY_INLINE void OnBody(GuiContext* context, Entity& entity)
    {
        auto& data = entity.template Get<CameraComponent>()
            .Camera;
        InputFloat("Near", &data.NearPlane);
        InputFloat("Far", &data.FarPlane);
        InputFloat("FOV", &data.FOV);
    }

    EMPY_INLINE void OnMenu(GuiContext* context, Entity& entity)
    {
        if (ImGui::Selectable(ICON_FA_RECYCLE "\tReset"))
        {
            entity.template Get<CameraComponent>() = {};
        }
    }
}
```

```
    }  
};
```

See (Figure 12.18) for visual result.

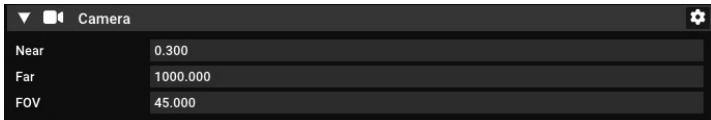


Figure 12.18: Example Control

12.7.4 Info Control

We can also do the same for the info control as shown in the following code snippet.

Listing 12.21: EmPyEditor/src/Controls/EnttInfo.h

```
#pragma once  
  
#include "IControl.h"  
  
struct InfoControl : IControl<InfoComponent>  
{  
    EMPY_INLINE InfoControl(GuiContext* context) :  
        IControl(context)  
    {  
        SetTitle(ICON_FA_INFO "\tEntity");  
    }  
  
    EMPY_INLINE void OnBody(GuiContext* context, Entity&
```

```

entity)
{
    auto& data = entity.template Get<InfoComponent>();
    InputText("Name", data.Name.data(), "Untitled", 64);
}

EMPTY_INLINE void OnMenu(GuiContext* context, Entity&
entity)
{
    if (ImGui::Selectable(ICON_FA_RECYCLE "\tReset"))
    {
        entity.template Get<CameraComponent>() = {};
    }
}
};

```

See (Figure [12.19](#)) for visual result.

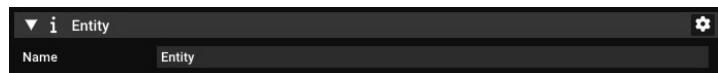


Figure 12.19: Example Control

12.7.5 Directional Light Control

and finally, the same can be done for the directional light. Here the code:

Listing 12.22: EmpyEditor/src/Controls/DirectLight.h

```

#pragma once
#include "IControl.h"

struct DirectLightControl : IControl<DirectLightComponent>
{
    EMPY_INLINE DirectLightControl(GuiContext* context) :
    IControl(context)
    {
        SetTitle(ICON_FA_SUN "\tDirect Light");
    }

    EMPY_INLINE void OnBody(GuiContext* context, Entity& entity)
    {
        auto& data = entity.template
Get<DirectLightComponent>().Light;
        InputFloat("Intensity", &data.Intensity);
        InputColor("Radiance", &data.Radiance.x);
        InputFloat("Bias", &data.ShadowBias);
    }

    EMPY_INLINE void OnMenu(GuiContext* context, Entity& entity)
    {
        if (ImGui::Selectable(ICON_FA_RECYCLE "\tReset"))
        {
            entity.template Get<DirectLightComponent>() = {};
        }
    }
};

```

I hope you were able to get the flow. It's up to you to do the rest.

12.8 Resource Window

A quite common feature in many engines is a content browser. It provides the ability to interact with all assets available in the project. we aim to implement a similar functionality to visualize all assets present. The code snippet below outlines the implementation to achieve this.

Listing 12.23: EmPyEditor/src/Windows/Resource.h

```
#pragma once
#include "Context/Context.h"

struct ResourceWindow : IWidget
{
    EMPY_INLINE ResourceWindow(GuiContext* context) :
    IWidget(context)
    {
        m_IconImage.Load("Resources/Icons/asset.png");
        m_Icon = (ImTextureID)m_IconImage.ID();
    }

    EMPY_INLINE void OnShow(GuiContext* context) override
    {
        if(ImGui::Begin(ICON_FA_FOLDER_OPEN "\tResources"))
        {
            int nbrColumn =
            (ImGui::GetContentRegionAvail().x/ASSET_SIZE) + 1;
            int columnCounter = 1;
            int rowCounter = 1;

            context->AssetView([&] (auto* asset)
            {
```

```

    if(columnCounter++ <= rowCounter * nbrColumn)
        ImGui::SameLine();
    else
        rowCounter++;

    // show asset icon
    bool isClicked = ImGui::ImageButtonEx(asset->UID,
m_Icon, ImVec2(ASSET_SIZE, ASSET_SIZE), ImVec2(0, 1),
ImVec2(1, 0), ImVec4(0, 0, 0, 1), ImVec4(1, 1, 1,
1));
}

ImGui::End();
}

private:
Texture2D m_IconImage;
ImTextureID m_Icon;
AssetID m_Selected;
};

```

It's noteworthy that we are loading a file icon using our previously created texture type. This texture is then utilized within the **OnShow()** function to render each asset available in the engine using an ImGui image button. Here is a resulting figure:



Figure 12.20: Resource Window

Here is the final code of the "Editor.cpp" file:

Listing 12.24: EmPyEditor/src/Editor.cpp

```
#include "Windows/Inspector.h"
#include "Windows/Hierarchy.h"
#include "Windows/Resource.h"
#include "Windows/Viewport.h"
#include "Windows/MenuBar.h"

struct Editor : GuiContext
{
    EMPTY_INLINE void OnGuiStart()
    {
        PostEvent<SelectEvent>((EntityID)4);

        AttachWindow<HierarchyWindow>();
        AttachWindow<InspectorWindow>();
        AttachWindow<ResourceWindow>();
        AttachWindow<ViewportWindow>();
        AttachWindow<MenuBarWindow>();
    }

    EMPTY_INLINE void OnGuiFrame()
    {
        // nothing
    }
};

int32_t main(int32_t argc, char** argv)
{
    auto app = new Application();
    app->AttachLayer<Editor>();
    app->RunContext(false);
```

```
EMPTY_DELETE(app);  
return 0;  
}
```



Figure 12.21: Final Scene Editor

(Figure 12.21) shows the final editor with all windows.

13 Game Executable

The final step is to implement the game executable (Game.exe), allowing the deployment of the final product created using the editor. The process is straightforward, considering the foundation is already in place.

Listing 13.1: EmPyGame/src/Game.cpp

```
#include <EmPy.h>

int32_t main(int32_t argc, char** argv)
{
    using namespace EmPy;
    auto app = new Application();
    app->RunContext(true);
    EMPI_DELETE(app);
    return 0;
}
```

As depicted in the code snippet above, creating an application instance and running the context with the `showFrame` parameter set to "true" is all it takes. Setting this parameter to "true" ensures that the final shader renders the frame using the default on-screen frame buffer, resulting in the scene being displayed on the screen. All other features will continue to function as expected.



Figure 13.1: Game Executable

Code not working? clone the branch "scene-editor" on the GitHub repository and grant execution rights to the "EmPyLinux.sh" script on Linux.

<https://github.com/Madsycode/book-empy-engine>

```
git clone -branch <branch-name> <repository-link>
```

Part VI

Fazit

14 Conclusion

When pride comes, then comes disgrace, but with the humble is wisdom (King Solomon)

In conclusion, this book has laid the groundwork for the ambitious task of building a 3D game engine from scratch. While the journey ahead is extensive, the foundation we have established serves as a crucial stepping stone. Acknowledging the complexities that lie ahead, it's clear that this endeavor is a continuous evolution. The principles and insights shared in this book provide a sturdy framework upon which further development and refinement can be built.

14.1 Perspective

As you may have noticed, there is still a substantial amount of work to cover. My hope is that you are now equipped to tackle the remaining tasks independently. It's essential to recognize that this is a journey, and the most crucial aspect is not merely reaching the goal but learning throughout the process. Always keep in mind not to give up and to embrace the journey of continuous learning.

14.2 Outlook

As you venture into the intricate world of game or game engine development, may this foundation guide and inspire your ongoing efforts. Your comprehension of the subject discussed in this book should now be enhanced. This

knowledge is versatile and applicable across various facets of software development. Always keep a humble spirit. Your rewards will be abundant, and your joy will overflow. Best wishes on your journey, and may you be blessed with good health.

Bibliography

Oluwaseye Ayinla. 10 criteria for adopting a game engine.

URL <https://www.linkedin.com/pulse/10-criteria-adopting-game-engine-oluwaseye-ayinla/>.

Epic Games Brian Karis. Real shading in unreal engine 4.

URL https://blog.selfshadow.com/publications/s2013-shading-course/karis/s2013_pbs_epic_notes_v2.pdf.

Graphics Compendium. Chapter 47: Physically-based rendering cook-torrance reflectance model. URL <https://graphicscompendium.com/gamedev/15-pbr>.

cratch a Pixel. Introduction to shading. URL

<https://www.scratchapixel.com/lessons/3d-basic-rendering/introduction-to-shading/diffuse-lambertian-shading.html>.

Joey de Vries. Specular ibl, a. URL

<https://learnopengl.com/PBR/IBL/Specular-IBL>.

Joey de Vries. Shadow mapping, b. URL

<https://learnopengl.com/Advanced-Lighting/Shadows/Shadow-Mapping>.

Joey de Vries. Normal mapping, c. URL

<https://learnopengl.com/Advanced-Lighting/Normal-Mapping>.

Bullet Physics development team. Bullet physics sdk. URL
<https://github.com/bulletphysics/bullet3>.

Dorian. Understanding the fresnel effect. URL
<https://www.dorian-iten.com/fresnel/>.

The Industry's Foundation for High Performance Graphics.
Opengl library. URL <https://www.opengl.org/>.

Juliette Foucaut and all. Iconfontcppheaders. URL
<https://github.com/juliettef/IconFontCppHeaders>.

Python Software Foundation. What is python? executive summary. URL
<https://www.python.org/doc/essays/blurb/>.

Editorial Team GoPhotonics. What is lambert's cosine law?
URL <https://www.gophotonics.com/community/what-is-lambert-s-cosine-law>.

JOHN HABLE. Optimizing ggx shaders with dot(l,h). URL
<http://filmicworlds.com/blog/optimizing-ggx-shaders-with-dotl/>.

Schutte Joe. Importance sampling techniques for ggx with smith masking-shadowing. URL
https://schuttejoe.github.io/post/ggximportancesampling_part1/.

JoeyDeVries. Physically based rendering. URL
<https://learnopengl.com/PBR/Theory>.

Lennart. Your public domain 3d foundation. URL
<https://ambientcg.com/>.

Etay Meiri. Tutorial 42: Percentage closer filtering, a. URL
<https://ogldev.org/www/tutorial42/tutorial42.html>.

Etay Meiri. Tutorial 16 : Shadow mapping, b. URL
<https://www.opengl-tutorial.org/intermediate-tutorials/tutorial-16-shadow-mapping/>.

Adobe Mixamo. Animate 3d characters with mixamo. URL
<https://www.mixamo.com/\#/?page=1&type=Motion%2CMotionPack>.

NVIDIA. Physx. URL <https://developer.nvidia.com/physx-sdk>.

Ocornut Omar. Dear imgui. URL
<https://github.com/ocornut/imgui>.

opengl tutorial. Tutorial 13 : Normal mapping. URL
<https://www.opengl-tutorial.org/intermediate-tutorials/tutorial-13-normal-mapping/>.

NVIDIA PhysX. Physx 4.1 sdk guide, a. URL
<https://frameworksdocs.nvidia.com/PhysX/4.1/documentation/physxguide/Manual/Startup.html>.

NVIDIA PhysX. Collision filtering, b. URL
<https://frameworksdocs.nvidia.com/PhysX/4.1/documentation/physxguide/Manual/RigidBodyCollision.html>.

PUC-Rio. Lua. URL <https://www.lua.org/>.

GitHub Repository. yaml-cpp. URL

<https://github.com/jbeder/yaml-cpp>.

Dassault Systèmes. Enterprise pbr shading model. URL

https://dassaultsystemes-technology.github.io/EnterprisePBRShadingModel/user_guide.md.html.

Font Awesome Team. Font awesome website. URL

<https://fontawesome.com/>.

Guilherme Teres. Cave-engine. URL

<https://unidaystudio.github.io/CaveEngine-Docs/>.

ThePhD. sol 3.2.3 documentation. URL

<https://sol2.readthedocs.io/en/latest/>.

the free encyclopedia Wikipedia. Physically based rendering, a. URL

https://en.wikipedia.org/wiki/Physically_based_rendering

.

the free encyclopedia Wikipedia. Phong reflection model, b. URL

https://en.wikipedia.org/wiki/Phong_reflection_model.

TheCherno Yan Chernikov. Hazel-engine. URL

<https://hazelengine.com/>.

- **Biography**

Thanks, on behalf of all prospective readers, for gaining clarity on whether this book aligns with their needs. A favorable review extends the book's reach, while constructive criticism enhances the quality of subsequent works.

—Franc Pouhela