目 录

[第一章 绪论 - 1 -](#_Toc133087088)

[1.1 背景和意义 - 1 -](#_Toc133087089)

[1.2 国内外研究现状 - 1 -](#_Toc133087090)

[1.3 本文主要工作 - 2 -](#_Toc133087091)

[1.4 论文组织结构 - 2 -](#_Toc133087092)

[第二章 项目结构设计 - 3 -](#_Toc133087093)

[2.1 包管理模式 - 3 -](#_Toc133087094)

[2.2 层次化结构 - 3 -](#_Toc133087095)

[2.3 模块化结构 - 3 -](#_Toc133087096)

[2.4 编码规范 - 4 -](#_Toc133087097)

[第三章 Core包运行时实现 - 5 -](#_Toc133087098)

[3.1 基础层实现 - 5 -](#_Toc133087099)

[3.1.1 Data（基本数据） - 5 -](#_Toc133087100)

[3.1.2 Reflection（反射） - 5 -](#_Toc133087101)

[3.1.3 Serializer（序列化） - 6 -](#_Toc133087102)

[3.1.4 Object（物体基类） - 6 -](#_Toc133087103)

[3.2 平台层实现 - 8 -](#_Toc133087104)

[3.2.1 Window（窗口） - 8 -](#_Toc133087105)

[3.2.2 GL（图形接口） - 9 -](#_Toc133087106)

[3.3 功能层实现 - 11 -](#_Toc133087107)

[3.3.1 GUI（图形用户接口） - 11 -](#_Toc133087108)

[3.4 框架层实现 - 11 -](#_Toc133087109)

[3.4.1 Actor（参与者） - 12 -](#_Toc133087110)

[3.4.2 Scene（场景） - 12 -](#_Toc133087111)

[第四章 Core包编辑器实现 - 13 -](#_Toc133087112)

[4.1 Assets（素材） - 13 -](#_Toc133087113)

[4.1.1 AssetImporter（素材导入器） - 13 -](#_Toc133087114)

[4.1.2 AssetDatabase（素材数据库） - 13 -](#_Toc133087115)

[4.2 Editor（编辑器） - 13 -](#_Toc133087116)

[4.2.1 Editor（物体编辑面板） - 14 -](#_Toc133087117)

[4.2.2 EditorMenu（编辑器菜单） - 14 -](#_Toc133087118)

[4.2.3 EditorWindow（编辑器窗口） - 14 -](#_Toc133087119)

[第五章 Render包实现 - 15 -](#_Toc133087120)

[5.1.1 ModelImporter（导入模型数据） - 15 -](#_Toc133087121)

[5.1.2 Light，Camera，Renderer（准备运行时数据） - 16 -](#_Toc133087122)

[5.1.3 Render（渲染流程） - 16 -](#_Toc133087123)

[5.1.4 PBR（基于物理的渲染） - 17 -](#_Toc133087124)

[参考文献 - 19 -](#_Toc133087125)

[致谢 - 20 -](#_Toc133087126)

# 绪论

## 背景和意义

PBR是当代最流行的渲染解决方案，多种软件都配备该渲染流程，比如3D建模领域的3DMAX、MAYA，影视制作领域的C4D，游戏引擎领域的Unreal、Unity，可以说PBR已经成为行业中的事实标准。

本着学习的态度，笔者希望能尝试自行实现。目前市面上也有些其他人对该功能的实现，但大多都是在第三方框架的基础上实现的。为了未来不被卡脖子，笔者希望能完全从头自制该效果，包括硬件，操作系统的接入，资源的解析，实现真正的自研。

要说近代最火的计算机技术产物，电子游戏一定名列前茅，这也是PBR需求量相对大的行业，上述说到的建模，影视制作，很多最终也是用在了游戏开发上。所以笔者希望能通过游戏开发的路线来抵达PBR的实现，而这也能为未来从事相关工作打上很好的基础。

## 国内外研究现状

PBR是图形渲染的一种流程方式，游戏引擎是为了便于游戏开发而特意设计的一种软件框架。早在上个世纪，两者便已开始发展。在过去因为发展方向和硬件等问题，两者并无绝对关联，而如今人们越发追求虚拟仿真的概念，不仅是游戏画面的追求，其他领域也因此诞生了诸如“数字孪生”，“元宇宙”等概念。

也因此，PBR，基于物理的渲染，其真实感使其备受欢迎，以至于几乎成为了游戏引擎的标配功能。国内外也有不少自制PBR的同行，有些是在第三方软件框架的基础上，有些是个人开发，而在商业公司基本都是有自己的定制管线。

能学会游戏引擎的开发，对相关工作帮助非常大，不过目前，国内相关的教程资料还是比较少的，很多虽然也是国人制作，但文档和注释基本都以英文为主，而在网络上搜索视频教程，也都是以国外博主为主。

总结起来，该行业目前是相当成熟的，但国内外相比还有些差距，不过也不需要纠结这些，既然别人已经造好了轮子，我们拿来直接用就是了，毕竟自研也不代表一定要从头开始。

## 本文主要工作

上述讨论了一些行业状况，所以可以确定的是本次毕业设计实质就是造轮子，而非什么技术突破，纯粹出于学习，好奇，以及对现状的不满足而推动一次对底层技术的探索，我将会真正的去编写一款以能够实现PBR为目标的游戏引擎，而该论文则是对该引擎的项目的介绍。处于篇幅原因，我没法面面俱到把所有东西都写上，因为里面很多技术细谈起来都足以单独出一篇论文，所以该论文更偏向于概括。我会写出项目的大概结构，以及对部分涉及技术的简述，也希望能为未来的同好，提供一个大致的实现参考。

## 论文组织结构

本论文的组织结构如下：

1. 阐述选择当前项目的背景和目的
2. 分析当前项目所需实现的功能模块，以及大概的项目框架
3. 讲述Core包运行时的代码实现，Core包是开发必备的包，而运行时是打包好程序运行必须的部分
4. 讲述Core包编辑器的代码实现，编辑器主要用于辅助开发，在最终构建项目时可以排除
5. 第五章 讲述Render包的代码实现，Render用于渲染场景画面，包括模型光照等，若没有Render包，则只有Core的GUI功能可以绘制画面，项目便会从3D图形程序退化成普通应用程序

# 项目结构设计

## 包管理模式

引擎是一个很复杂的软件，会涉及到相当多的功能技术，比如数学运算，序列化，渲染等，考虑扩展性未来除了一些核心功能可能还会实现物理，动画，声音等等功能，而这其中光一个核心层实现就已经相当复杂了，若将所有东西都混在一块开发，那项目一定比如从口袋掏出的耳机线还要乱上几千倍。

不仅是为了开发效率，也是为了项目的长线发展，必须想办法减少代码间的耦合性。通过观察可以发现，大部分功能之间往往是没有直接联系的，而且在不同的使用环境下所要用到的功能也略有区别，直接将所有功能代码都放在一起会导致产生没必要的耦合，所以要想办法将他们分离。

观察其他商业引擎，可以发现大多是使用专门的包管理器管理代码，它们将所有功能依据关联性以包为单位进行分割，不同的包提供不同的特有功能，甚至引擎的基本功能也是由多个内置包组成，我们也在此选择同样的管理方案。

目前起步阶段直接创建一个包管理器的成本太大，所以我们选择先用一些简单的类似管理手段。分离源代码，将不同的功能块放在不同的文件夹中，文件夹内应配备说明文件并提供依赖相关的信息，接着用户手动在IDE内通过添加外部项的方式将相关源代码集合编译。

不仅是用户，引擎也是上述包的使用者，不同的引擎需求也不同，渲染引擎注重渲染，物理引擎注重物理模拟，但可能都要用到序列化，空间变换等功能，存在相同功能又略有差异，所以采用组合功能的方式是最佳实现手段。引擎是最终产物，一个可编译项目，而实质上它便是通过上述包管理模式引入功能块，组合形成的源代码集合。

## 层次化结构

不同模块间会存在依赖关系，有些模块会频繁被其他模块引用，比如数学，序列化等底层功能。在依赖关系上要确保单项依赖，这样项目结构才可以呈现出一种强可读性的树状层次结构。

## 模块化结构

接着我们要减少该树结构中的粗细和分叉，遵循设计模式中的单一职责原则和最少知道原则，以高内聚低耦合的原则编写代码。

## 编码规范

引擎基于C++语言开发的，原因是因为C++更接近底层不仅效率高而且接入各种底层功能时适配性很好，对于其编码规范也有几点要求。

#### 优先标准库实现

此处标准库并非只指代C++标准库，包括市面上的一些主流商业或开源库也算在在内，这样可以减少学习和共享成本，便于项目未来的长线发展。具体的外部库引入采用vcpkg实现，以便自动化并减少项目大小。

#### 使用新式语法和函数

目前多种语言间的语法特性都呈现相互融合的趋势，新式功能符合主流发展方向，跟随当代开发者的开发习惯在发展。首先项目需要能跟上时代，另外更重要的是采用新功能编写代码往往更加简便，能大幅减少工作量。

#### 使用模块代替头文件的编译方式(暂时无法实现)

项目长期发展后，代码量将会非常庞大，此时采用头文件方式编译项目非常缓慢。目前C++20提供了模块功能代替头文件，大幅减少了编译时间，这可以加快项目开发速度，对大型项目非常实用。但可惜的是目前主流编辑器对模块功能的支持都不够完善，经常误报语法错误，所以目前暂不考虑。

# Core包运行时实现

## 基础层实现

这一层的功能是整个软件的基础，包括规定公共数据类型，增强开发环境能力。实际上这一层的绝大部分功能，在很多新式语言中甚至是语言本身自带的功能，由于我们是底层开发，使用的是C++语言，其默认没有这些功能所以我们要实现这些。

### Data（基本数据）

没有什么特殊的，主要是标准库数据类型的扩充已经一些常用数据类型，比如向量，颜色这些。这类数据在渲染器实现中非常常用，但可惜标准库并不支持，所以各个软件基本都是自行实现。也因此除了因实用性而定义它们，另一个原因也是为了统一各个第三方库中相关数据类型使用。

### Reflection（反射）

反射功能将允许通过运行时方式访问字段和函数（本项目以字符串的方式访问），同时支持非侵入式的私有成员访问这将便于我们后续编辑器功能的开发，从而方便的对对象进行可视化和编辑。另外可访问的函数包括构造函数，也因此我们得以将类型信息存储，并在后续重新以此创建对象，这便为我们的序列化功能实现提供了条件。

#### Transferer（传输器）

一个专注于变量信息的传输器，可以传输变量的名称、类型、地址等信息，很多功能都会用到该对象，主要用于对象的序列化操作。之所以放在Reflection包中是因为所有可反射对象（继承Reflective的对象），都默认提供Transfer（Transferer&）函数，该函数用于标记所有的序列化字段，借此可以很方便的对对象的字段进行遍历，反射信息的注册也是借此完成的，另外用到该对象的还有如：Serializer，遍历字段进行数据储存恢复；GUITransferer遍历字段进行UI显示，ObjectPtrTransferer：遍历字段判断字段类型，找出所有指针，借此获取对象所使用的其他外部对象。

字段分引用类型和值类型，在传输引用类型的时需要特殊处理：比如如何区分传输指针本身还是指针所指向的对象，如何表示数组的个个元素，若单纯只使用字段名称表示就会出现同名的问题。更具体一些，在Json语法中会使用大括号表示的数据的嵌套关系，那如何表示是进入“下一层数据”还是“下一个数据”，由此可见数据信息是呈现树形结构而非线性的，所以需要一个入栈出栈的操作表示数据路径的变化，这就是Transferer中PushPath和PopPath的由来。

PushPath和PopPath在一些其他情况下也会用到，比如GUITransferer是借助Imgui实现的，Imgui要求所有可交互UI都必须提供一个唯一的事件ID，若要表示一个数组，但数组内的ID仅使用数组名表示的话，那ID就会相同也因此便无法正常使用。但在一些无需表示数据路径的情况中，该功能就会变得没有必要，比如在Bytes序列化中，考虑性能和适用情况，字段名等信息将直接被丢弃，所以这两个函数都为空，同时这也解释了为什么Bytes序列化没有Json序列化那样可读可写。

### Serializer（序列化）

序列化即能将一个对象转换成一段序列，该序列可存储可传递，并后续还可以借助该序列反向重新恢复成该对象。

序列即一段数据流，比如使用字符串或字节流，两者各有优缺点：字符串可以搭配一些主流的数据交换格式，如Json，Xml等，可读性强且可配合版本控制软件（git）；字节流空间占用小速度快，但不便于修改，只能程序化处理。通常对于开发阶段使用的数据文件以字符串储存，便于查错和团队协作，缓存文件或最终的打包文件，选择用字节流，提高性能。

### Object（物体基类）

物体是整个软件中几乎所有类型的基类，它对前文谈到的功能（反射和序列化）进行了封装，除此之外还提供了比如安全指针，生命周期管理等实用基础功能，这也是大多数新语言都会实现的，如在C#，java中都自带Object，是所有类型的基类，也因此这些语言中的对象都自带上述说过的很多功能。

当前项目中的Object除了具有上述功能外，还额外增加了一些功能。

#### Awake（序列化支持）

首先因为序列化的缘故，所以当一个物体被构造函数创建出来时不一定处于准备好的状态，此时需要通过序列化或反射为其填充初始值，然后调用AwakeObject函数进行唤醒，这时的Object才是真正可用的Object。

#### Destroy（主动销毁）

传统的托管内存管理采用引用计数的方式，只有引用为空时，物体才会被销毁，C++不是托管语言，但也提供了shared\_ptr指针实现同类操作。但本项目除了具有引用计数回收的方式外，还额外增加了主动销毁的方式（DestroyObject），因此对于安全指针需要特殊实现。

#### ObjectPtr（专用安全指针）

ObjectPtr指针是Object类型的专用安全指针，分ObjectPtrBase和ObjectPtr<T>两种，ObjectPtrBase是指针功能的真正实现者，而ObjectPtr<T>只是通过模板功能免去使用者类型转换的麻烦。引用计数的实现同shared\_ptr，让相关指针共享一个计数变量即可。而对于Object指针的返回，因Destroy功能的存在，需要特殊处理。

C++中存在野指针的情况，即当一个指针指向的内存被回收时，指针使用者可能并不知道，继续使用该指针就会出现异常。因此我们不能直接返回指针也不能储存指针，我们采用一种叫ObjectID的属性来间接寻址Object。

#### ObjectID（运行时身份标识）

所有的Object对象构造后都将被分配唯一的ObjectID作为它们的身份证明，通过ObjectID我们可用获取到Object的原始指针，以及它的生命周期。使用Object时，先进行Object的生命状态确认，当Object被标记为销毁时，我们便可得知指针已经失效，从而避免了野指针的问题（项目中获取指针和生命状态是合并在FindObjectOfObjectID中实现的）。

#### ObjectRef（存储时身份标识）

ObjectID只在构造函数调用后生成，每次运行时同一物体的ObjectID存在随机性，而Object需要序列化支持，需要序列化引用关系，而无论是存储指针还是ObjectID都将无法无法复原，因为两者都只是运行时有效，所以我们需要使用ObjectRef作为存储时的身份标识。

标识引用最简单的方式就是给被引用物体起个永久唯一的名称，引用者只要记下该名称，下次运行时再通过名称找到对方即可，最符合这一需求的服务就是GUID。但只使用单个GUID无法标识物体间的关联性，也不便于查找，通常很多数据都是成组的出现，比如模型数据，内部可能有多个网格，如何表示引用其中的一个数据？

由于模型是单个文件，因此即使只是需要内部的一个对象，也需要先加载整个文件，然后再从中挑选，所以二级索引是肯定的。可以选择再来一个GUID，但没有必须，项目中使用FileID进行二级索引，这是一个随机整数，因为是单个文件内使用，所以碰撞概率不会很高。

#### ObjectSerialization（Object专用序列化）

这是一个静态工具类，其对序列化功能进行了封装，可自动处理Object的加载保存和复制。Object及其相关Object将会保存一起保存在同一个文件或序列中，通过GUID判断多个Object间是否相关：相同则相关，没有GUID则在其使用者保存时设置为和使用者一样的GUID但FileID不同；当GUID不同时代表是外部引用关系，被引用的物体将被认为是已存储在其他文件中，所以不会跟随序列化。

## 平台层实现

平台即运行环境，常规来讲便是指操作系统的差异，而跨平台就是指要求软件能够横跨在这层差异上运行，使这层差异透明化。平台层的目的便是为了这一目标而搭建的，但受限于工作效率的原因，目前没法实现跨平台，但这不代表平台层名存实亡。对于平台层，可以将其视为运行环境的搭建层，如果说基础层是楼房的地基，那平台层便是搭在地基上的舞台，只有舞台准备好了，演员们才能登台表演。

### Window（窗口）

窗口是整个引擎的躯体，是运动的基本。夸张点说，没有窗口就没有应用程序，甚至有操作系统直接将窗口作为自己的名称（Windows操作系统），可想而知它的重要性非比寻常。窗口是操作系统提供的，不同平台有不同的窗口，为实现跨平台需求并节省工作量，此处直接借助glfw来开发。

#### 提供功能

现在我们需要通过Window模块实现和用户的交互，并让窗口执行我们的代码，以及提供一些基本的软件环境信息，整体需求可以概括为以下几点。

#### 输入输出

Window是我们用于和用户通信的通道，Window应可以接受用户输入，比如键盘，鼠标事件，并提供相应回调，让我们的程序得以处理这些用户操作。处理完后我们应当能将结果反馈给用户，常见的比如在屏幕上显示一行字，但这部分比较复杂，原生窗口也没有直接提供功能，所以这方面的工作要交给另一个模块（GUI）处理，而窗口需要提供给这些模块一个接入点。

#### 循环更新

除去输入输出时的回调，Window还应该提供一个随时间流逝自动执行的回调，这样我们才能让画面得以随时间保持动态更新。窗口本身具有一个绘制事件，因为窗口的画面经常需要重绘，绝大时候该事件都会自动调用且更新频繁，但用户无操作时便会停止，我们需要手动让该事件在空闲时也能持续触发，不过glfw用不到这个，因为他的窗口事件处理是非阻塞的，和原生的Windows操作系统的处理函数不一样。

#### 环境信息

除去事件回调，Window还应提供一些运行环境的信息，比如屏幕大小，运行时间，等等，这些功能分别分布在了Window模块内的其他几个文件中。

#### 关于事件回调的讨论

有多个功能块会用到相同的事件，且它们之间很可能需要相互协作，如何正确触发它们的更新事件，确保按照预期的顺序执行很重要。

* 延迟输入输出结果

不直接在输入输出时触发回调，因为这样会导致程序执行顺序不可控。这里可以仿照其他引擎的做法，遇上输入输出时缓存结果，然后让监听者自行在统一的流程中进行查询，这样还能简化事件相关代码的实现。

* 控制注册事件方式

大部分事件主要还是通过用户注册的方式添加，而事件之间往往存在耦合性，提供一个安全可控的注册方式也是需要考虑的事。目前的注册方式可以归纳为以下两种，都各有优缺点。

* + 通过头文件初始化实现自动注册

优点：自动化，对用户透明

缺点：可控性差，不是主流方法，无法确保支持率，会产生模块依赖

* + 通过手动调用来执行流程函数

优点：可控，安全，大部分第三方库都是通过该方式提供服务，不会产生依赖

缺点：需要手动控制，麻烦（但可以通过封装的方式对引擎使用者屏蔽）

目前引擎内部是使用手动调用的方式，而提供给用户的主要是以头文件的方式。

### GL（图形接口）

GL是用于接入底层图形引擎的模块，用户一般无需直接使用，因为通常上层会有其他模块对其进行进一步封装（GUI，Render包）。

#### 工作方式

在GL中图形渲染管线被首次提及，渲染管线是一道用于图形渲染的流水工序，就如同工厂的流水线一样，准备好材料并安排工人到岗，接着启动机器开始作业，工作期间人员和机器都将不再允许调整以避免流水中断，直到所有产品生产完毕，然后开始准备下一次流水作业。

GL便是以这种工作方式运行的，这也是因为受到了GPU工作方式的影响。从微观上上看GPU是并行作业的，但从宏观的角度，其工作方式更类似于早期批处理操作系统，准备好数据和代码，接着启动，运行期间不能再做调整，然后等待其输出结果后，才能进行下一次操作。也因为这一特殊的工作方式，使得出现了很多渲染优化方案，比如渲染批处理，GPU实例化等，但本项目目前还未加装这些功能，所以不做讨论。

#### 相关资源

上文提到了GL的渲染管线工作方式，而如何配置该渲染管线，准备什么数据，就是该节需要提及的内容。我们将这部分涉及到的物体统称为渲染管线资源，总结起来有以下几种。

* Buffer（缓冲区）

缓冲区是一段数据流，有多种类型，比如顶点类型，索引类型，常量类型，这些数据将被上传到渲染管线，不同的类型会有些不同的处理方式，但最终都将被提交给Shader使用。正常引擎的渲染范围确定都是通过三角形网格实现的，而这一资源就是用于定义网格信息的。另外，有时我们也需要向Shader传递一些数据，比如矩阵，颜色等信息，就可以选择用常量类型的缓冲区实现。

* Texture（纹理）

纹理即图片，通过uv坐标我们可以采样图片上的颜色信息，除了输入进来的纹理，我们也可以输出纹理，因为渲染管线的最终产物本身就是纹理，纹理既可作参照物，也可做画板使用，作为画板使用时，需要额外准备一张纹理，用于记录深度模板数据，这在后续的State中会用到。

* Shader（着色器）

Shader是一份代码文件，它就是跑在GPU上的程序，上述说到的所有资源最终都会作为Shader的参数输入进来，而Shader则会将他们计算成最终的结果，比如确定光栅化的范围，像素的颜色。可以将其视作渲染管线上的工人，由于管线由多个阶段组成，所以Shader也需要提供多份，默认必备是顶点着色器和像素着色器。

* State（管线状态）

State是对渲染管线的一些设置选项，若说管线是一台流水线机器，State就是对该机器功能的一些调整选项，比如：当Shader返回了半透明的像素时如何处理，是和前一个像素叠加还是覆盖；当一个网格挡住了另一个网格，另一个网格是被抛弃还是继续放在前面。所以State就是用于设置渲染管线各功能的工作模式。

#### 底层引擎

对于要封装的底层图像引擎，主流有三种选择：OpenGL，Direct3D，Vulkan。其中Vulkan可以视作OpenGL的现代化版本，Vulkan跨平台性能高支持的Shader语言也很多，比如GLSL，HLSL都可用，其中HLSL类似于C语言，入门门槛低，而OpenGL就不支持这个的，所以说Vulkan是一个相当推荐的图像引擎。Direct3D是Windows独占的图形引擎，也支持HLSL，这也是我当成选择它的原因。

Direct3D是目前项目的底层图像引擎，但不能满足跨平台的需求，未来有机会一定会换掉，目前准备用Vulkan进行替换。

## 功能层实现

上述的基础层和平台层是项目最基本的两个层次，分别在内部和外部两个方向上打好基础，而往后的框架层便是独属于引擎自己的软件结构。在进入功能层前，我们需要对前两场进行进一步完善，封装一些常用功能，以提高给高层进行使用。

### GUI（图形用户接口）

图形化的用户交互功能实现，这是对Window和GL的功能封装，便于构建我们普遍意义上的UI系统。目前主要是Editor层在使用，用于以友好的图形化界面向用户反馈引擎状态和功能调用，而在Runtime层中，引擎本身不会使用的，仅为其提供调用环境，以便用户自行使用。出于工作量的考虑，GUI目前是直接借助第三方库Imgui来实现的。

## 框架层实现

从框架层开始，便正式步入了独属于该引擎的编程环境，框架层字如其名，便是为用户准备编程框架的层次，所有用户代码都需要借助框架层，在框架中实现功能。

很多软件都有独属于自己的框架，比如安卓的Activity和Intent便是其框架中的部件，对于该引擎适配的框架，市面上有三种较为成熟的方案，分别可以对应上Unreal，Unity，Cocos三款商业引擎，三种方案各有区别但也都有类似的地方，比如Unreal的Actor可以对应Unity的GameObject和Cocos的Node。其中Actor是场景中的主体对象，拥有自己的属性和功能且允许用户自行编写代码，自定义自己的Actor，而GameObject则是只保留持有人的功能，将自定义代码的能力交给了自己的Component属性，而Node则是直接抹除了GameObject和Component的差异，将万物都视作Node。

由上可见三种方案，本质上其实比较接近，只是在发展方向上选择了不同的权重值。而本引擎主要选择了基于Unity的GameObject方案作为引擎框架。

### Actor（参与者）

对于所有要使用引擎功能的模块，框架层将其抽象层成了Actor模块，Actor模块提供了多种Actor给用户使用，以适配不同的状况。

#### GameObject（游戏物体）

GameObject延续了其在Unity中的叫法，功能上也基本相似，他是Component的容器，可以将其想象成一台机器，而组件是他可提供的功能。在Unity在GameObject默认自带一个无法删除的组件Transform，用于存储空间信息，得以将GameObject抽象成了一个三维空间中的物体，并且因为允许嵌套，从而能构建一个个子坐标空间，也便于进行物体分类。这一功能对后续无论是图形渲染还是物理模拟都是个非常实用的功能，再加上考虑依赖性和简化代码的问题，Transform组件在该引擎中已被作为GameObject的内置功能而做了合并。

#### Component（组件）

组件是GameObject的所持有功能的抽象，也是用户代码的入口处，用户需要编写自己的Component并挂载在GameObject，通过接收事件加入引擎的更新流程中，从而控制着引擎的运作，定义出独属于自己的功能实现。

在Component中有一个特殊的存在：Behavior，该组件是内置的一款封装后的组件，其对所有Component基本事件都提供了安全的事件回调入口，并额外提供了随时间更新的事件回调，正常来说所有用户功能一般都是通过该类进行实现的。

### Scene（场景）

Scene是所有GameObject的容器，当然除此之外还会存储一些GameObject可能用到的额外信息，比如一些设置信息等。Scene是一个容器，本身不在意自己存储什么，它要做的就是管理好这些所存物体的收纳功能，Scene提供保存和加载的功能，同时作为最上层的容器，它也是整个引擎运行时的入口点，所有引擎物体以Scene为单位加载和卸载，每次只允许一个Scene存在于引擎中。

# Core包编辑器实现

前文说到引擎以Scene为入口点，且以Scene为单位运行，Scene是一个可序列化文件，引擎运行所需的所有数据都在Scene中存储着。若说引擎是一个躯壳，那Scene便是它的灵魂，整个编辑器层实际上都是围绕着编辑Scene目的来实现的。

## Assets（素材）

素材即未加工的原生材料，比如原始的png图片，fbx模型，这些都是外部的文件格式，在实际使用时需要转换成引擎可读的运行时数据，比如fbx模型中网格数据最终要转换成GL可识别的Buffer格式才可使用。所以素材模块就是用于处理这些原始数据加载保存等功能的模块。

### AssetImporter（素材导入器）

如何导入原始素材，对于不同的文件有不同的加载方式，AssetImporter是一个抽象类，用户需要实现其中的导入函数，来支持自己的自定义格式，这些素材最终都会被转换为Object格式，并存在一个主Object，AssetImporter还会为这些Object分配ObjectRef以使其支持序列化的功能，对于这些新数据我们称其为资源。

### AssetDatabase（素材数据库）

加载好资源后，我们需要将这些新数据存储在Scene中，Scene有一个专属的Library文件夹用于储存用到的运行时数据，而AssetDatabase能够自动加载素材并将其产生的资源文件存储其中，使用户无需手动管理Library文件夹。引擎默认将Assets作为素材文件夹，只要将素材放入其中，AssetDatabase就能自动识别加载，并记录信息用于其他管理操作。AssetDatabase还支持反向存储数据，默认以Json进行储存，以便用户进行查阅编辑和版本控制，当然同时它也会在Library文件夹同步存储其对应的Bytes资源文件，以便运行时使用。

## Editor（编辑器）

编辑器模块是一系列编辑器相关的UI接口，用户通过该模块可以定义资源的编辑器界面，编辑器本身的内置界面也基于该模块开发，编辑器自身基于运行时开发，可以说本身就是第一个基于本引擎开发的软件，它可以方便的查阅修改引擎内物体的状态，并提供多个接口进行扩展，不仅是为了用户考虑，也是为了方便后续其他模块的编辑器层实现，可以说该模块就是编辑器层所使用的软件框架。

### Editor（物体编辑面板）

Object是只一个基类，基于Object开发的子类贯穿整个引擎，不同的Object其编辑方式必然会有些区别，因此提供该接口用于用户自定义所用类型的编辑面板。引擎默认会提供一个基于Transferer开发的编辑面板，可以用于修改物体的所有序列化字段，但实际使用中物体的更新还需要通过函数触发，因此该面板一般只做应急使用。

### EditorMenu（编辑器菜单）

编辑器顶端的菜单栏，一般用于作为一些静态函数的入口使用。

### EditorWindow（编辑器窗口）

编辑中基本所有界面都是基于窗口的，该接口允许我们在编辑器中实现一个自己的独立的窗口，以此获得最大程度的自定义需求。

# Render包实现

Render包，即专门用于实现渲染功能的代码库，也即最终实现PBR的地方，因此我们不和Core包一样以项目结构介绍，而是以功能实现的顺序进行介绍，并简要对其中的一些实现注意点进行描述。

### ModelImporter（导入模型数据）

首先最基本要实现的功能就是能导入我们用于渲染的素材，行业通常都会采用模型文件存储这些数据。

#### 模型的组成

，在模型文件中主要由三种资源组成：Transform，Mesh，Material。

#### Transform

Transform即上文提到的GameObject物体，存储着物体的变换信息和父子关系，这种数据结构非常常见，基本所有建模软件都是以该结构规划场景信息的。

#### Mesh，Material

Mesh和Material都是渲染管线中会用到的资源，这并不是底层图像引擎的原生数据类型，而是对其原生数据类型的进一步封装：Mesh封装了顶点索引缓冲区，存储了物体的各种顶点信息，可用于确定物体渲染的范围；Material封装了常量缓冲区，纹理，着色器，管线状态信息，主要用于确认物体如何绘制，配合上诸如法线，uv等顶点数据可以进行一些复杂效果的表示，比如光照效果的实现。

#### 模型格式的选择

模型文件在行业内有多种格式支持，比如fbx，obj，gltf。

最主流的是fbx格式，作为fbx的持有者，老牌数字内容创建公司，Autodesk在行业内有着几乎垄断的优势，这也是fbx之所以流行的原因之一，除去这一层光鲜的外衣，单看fbx这个格式还是存在很多缺点。首先fbx是闭源格式，内部数据不明且非常臃肿，导入fbx必须下载专用的sdk，而光这sdk就要占用掉几百MB，具体导入时也很麻烦，数据还需要自己进一步处理，因为没法直接转换成底层图像引擎可识别的数据格式。该格式曾是本项目的默认模型格式，但在我了解到gltf后，果断抛弃了这一格式。

Gltf格式是新式的开源模型格式，和大名鼎鼎的图像引擎OpenGL是同一推行者。其继承了OpenGL的开源精神，存储数据完全透明，而且采用了当代主流的数据交换格式Json存储数据，只需要一个几百KB的Json解析器即可读写gltf文件。另外gltf专门为图形渲染这一需求做了优化，更符合当前使用者的使用习惯，支持网格压缩，PBR材质参数，最重要的是和底层图形引擎所使用的数据格式完全匹配，基本不需要自己做处理，可以直接将数据提交给图形引擎使用，非常方便。

当初接入fbx导入器时，我花了大半天，但不仅功能很不完善，还导致项目因此膨胀了几百MB。而后来使用gltf时，只花了两个小时，就成功完成了所有导入需求，并且只要用项目里原本就有的Json解析器便能实现了。

不过目前glft也有个很大的缺点，不被主流建模软件支持，比如3dmax这类建模软件，作为竞争对手，这些商业公司肯定是非常抵制的，不过好在另外还有一款开源建模工具Blender支持，这也是我本人很喜欢用的建模软件，因此作为我的个人引擎来说，这点并不致命。

### Light，Camera，Renderer（准备运行时数据）

Light，Camera，Renderer是Render包对Core包框架层Component的实现，三者各为渲染功能提供了不同的基础数据。它们是场景元素，需要用户利用编辑器功能提前放置在场景中，这样后续的渲染流程才知道应渲染什么。

Camera：模拟一台真实相机，并提供一系列相机参数使用，这些参数最终会被转换为两个核心渲染用矩阵，世界到视图空间矩阵和视图到剪辑空间矩阵。

Renderer：指明要渲染的物体，提供物体的本地到世界矩阵，并指明渲染该物体所使用的Mesh和Material。

Light：提供一系列灯光参数，以便在PBR流程中进行光照信息的计算。

### Render（渲染流程）

Render模块通过利用Core包中的事件功能，注册了实现渲染功能所需的基本流程，对该流程，可以进行一下简述。

#### 资源准备

该阶段会统计场景中所有的Light，Camera，Renderer等渲染物体，这些物体指明了基本的渲染数据，划定了渲染的目标，除了统计之外还会对这些物体进行一些预处理，比如灯光要生成光照贴图，Renderer要根据渲染顺序进行排序。

#### 相机渲染

每有一个相机就代表着要进行一次渲染，所以渲染时要遍历每一个相机，将相机参数上传至渲染管线，接着基于该相机可能会进行背景的绘制依以及一些剔除工作处理，接着遍历每一个灯光，在灯光内遍历每一个渲染器，每次都要上传当前渲染器的Mesh和Material，并调用绘制。若连续多个渲染器的Mesh或Material都一样也可以选择不重新上传，这便是批处理。然后遍历灯光的方式，是因为我们使用了向前渲染作为灯光渲染方式，另外还有一种叫延迟渲染的方式，性能更好但也更复杂。

#### 提交结果

每一次绘制都会用到Mesh和Material，其中PBR就存在Material中，这是Shader相关的内容，放到后面再说。总之通过一次又一次的绘制调用，画面逐渐完善，最终绘制完成后，我们需要通知GL交换渲染缓冲区，于是用于绘制的纹理便会被交换到前台，显示在屏幕上。当然因为GUI的缘故，Core层会自动在靠后的时间段内触发该功能，而Render要负责的就是每帧一遍一遍的绘制画面即可。

### PBR（基于物理的渲染）

PBR是一种概念和规范，字面上要求基于物理即渲染应符合现实中的光照规律，但实际到各个软件中都会因业务需求略微调整。PBR并没有定死光照效果的实现方式，所以最重要的还是规范了行业流程，让个软件之间的渲染素材渲染效果得以互通。

PBR又称PBS，其中S是Shader的缩写，这也表明了PBR是一种主要用在Shader上渲染技术，以下讨论的所有内容基本最终都是用在了Shader的编写上。

#### BRDF（双向反射分布函数）

考虑现实中的光照现象，光线通过反射进入人的眼睛，让人得以看到画面，这种反射现象有两种情况：一种是直接反射，比如光线通过镜子反射到眼睛，这样的光没有太多损耗，亮度高且集中；另一种是间接反射，光线经过多次反弹后才进入人的眼睛，光照范围大亮度低，在粗糙的表面上尤为明显。BRDF中的双向反射分布指的就是这两种光照路径，而函数代表了它的形式，它是一种计算公式，能够计算出某一平面的光照效果，而在计算中专门考虑到了两种光照模式的影响。

BRDF不是PBR中独有的概念，在早期就已经出现了基于BRDF的光照模型，比如Blinn-Phong，这是一种经验主义的光照模型，其指出最终光照由三种基本光构成：漫射光，镜射光，环境光。虽然Blinn-Phong是经验主义，但其光照构成和当今的一些PBR光照基本是相同的，只是在计算公式要做些修正。

#### 微表面模型

现实中的物体表面形状多种多样，有些粗糙有些光滑，而且大多数情况单独一个物体的表面就可能存在多种形式，比如一个半上锈的铁板，未上锈的地方保持了金属的光滑感，而上锈的地方则粗糙不平。

为了实现上述情况，我们可以建立微表面模型，将物体表面视作由一个又一个的微表面组成，每一个微表面都有着自己的属性用于描述其特征形状，微表面不需要人为划分，因为图形中的光栅化功能会自动帮我们完成，而对于微表面的属性，我们只需提起准备几张描述属性的纹理，接着直接采样其值就行。

对应微表面的属性的表示，目前PBR主要规范了两种工作方式：金属度+粗糙度；镜面反射+光泽度。其中金属度+粗糙度是主流实现方式，基本上从第三方建模软件导出模型时，都会提供这两个值。

#### 能量守恒

这是现实中光照会遇到的规律，这也能解释为什么漫射光的亮度会比较低，而镜射光的亮度会很高，一个光照亮的范围越大，那它的平均亮度自然也就越低。

#### 菲涅尔效应

从水面正上方观察水体，可以很清楚的看到水底的模样，而从水面平时水体就很难看到，整个水面如同不透明的镜面。这一现象便是菲涅尔效应，它表明了物体表面的反射率会随着观察角度的不同而变化。

#### Gamma矫正

人眼对颜色的敏感度并不是线性的，总体来说暗部是人眼更能观察到的区间，另外早期显示器对颜色的显式也不是线性的，这些都推动了Gamma值的诞生。Gamma函数是一个非常简单的指数函数，而Gamma值就是该指数变量，将颜色的所有分量进行Gamma次方运算并输出，就是被伽马调整后的颜色值。

基本上所有图片存储颜色时都不是线性存储的，因为上述的原因，他们会更多的储存暗部颜色，于是需要调亮颜色，所以存储时的Gamma值一般设为1/2.2，0-1的值乘上0-1次方于是暗部区间得以提高。

但在PBR中要求其处理的颜色空间必须是线性的，这样才符合真实的物理情况，所以使用图片前我们需要进行Gamma矫正，非常简单，反向乘上2.2次方即可。

参考文献（根本不全，完全就是糊弄人）

[1] 王江荣. 【Unity】Asset简介[EB/OL]. (2021-09-30). <https://zhuanlan.zhihu.com/p/411946807>

[2] Unity技术博客. 揭秘Unity的黑盒世界，原生对象和托管对象浅谈[EB/OL]. (2021-09-28). <https://developer.unity.cn/projects/6152e4fbedbc2a0020584027>

[3] 李海辰. Unity引擎资源管理代码分析[EB/OL]. (2017-11-17). [https://cloud.tencent.com/developer/article/1005843#](https://cloud.tencent.com/developer/article/1005843)

[4] 微软文档. HLSL参考[EB/OL]. (2022-09-22). <https://learn.microsoft.com/zh-cn/windows/win32/direct3dhlsl/dx-graphics-hlsl-reference>

[5] 微软文档. Direct3D11图形[EB/OL]. (2022-09-23). <https://learn.microsoft.com/zh-cn/windows/win32/direct3d11/atoc-dx-graphics-direct3d-11>

[6] 翰墨小生. 矩阵-DirectX与OpenGL的不同[EB/OL]. (2012-08-02). <https://www.cnblogs.com/graphics/archive/2012/08/02/2616017.html>

[7] 微软文档. 使用使用Win32API的桌面Windows应用开始[EB/OL]. (2022-09-22). <https://learn.microsoft.com/zh-CN/windows/win32/desktop-programming>

[8] mwdesign. PBR 渲染技术浅谈[EB/OL]. (2022-04-26). <https://zhuanlan.zhihu.com/p/503369650>

[9] Uniy文档. 前向渲染路径[EB/OL]. <https://docs.unity.cn/cn/2019.4/Manual/RenderTech-ForwardRendering.html>

[10] 微软文档. 矩阵转换[EB/OL]. (2022-09-22) <https://learn.microsoft.com/zh-cn/windows/win32/learnwin32/appendix--matrix-transforms>

[11] Unit文档. 事件函数的执行顺序[EB/OL]. <https://docs.unity.cn/cn/2019.4/Manual/ExecutionOrder.html>

[12] Uniy文档. 重要的类 [EB/OL]. <https://docs.unity.cn/cn/2019.4/Manual/ScriptingImportantClasses.html>