**项目总体结构**

# **Hazel —— 引擎核心代码，编译为静态库，链接到Hazel-Editor**

* Core，共用的一些核心功能，重要内容包括Application（应用程序单例），Log（日志系统），Layer和LayerStack（层和层栈，管理事件传递和游戏更新），Window（窗口抽象类）。
* Event，事件系统。
* ImGui，定义UI层的行为。
* Math，数学工具。
* Renderer，渲染器（重要），涉及到图形API的封装，渲染合批等。
* Scene，场景系统，涉及到实体、组件、序列化等。
* Scripting，脚本系统，与Mono配合。
* Utils，提供平台无关的API。
* Platform，平台层，为每种图形API和每个平台提供具体实现，目前只支持OpenGL和Windows系统。
* vendor，引擎核心代码用到的第三方库，大部分是直接使用源码，与项目一起编译，编译为静态库后链接到Hazel静态库。

# **Hazel-Editor —— 引擎编辑器，编译为可执行程序。**

* SandboxProject，C#项目，内部有用户自定义的脚本，这些脚本会被编译为动态库，编辑器在运行游戏时会加载这些脚本中的内容，执行用户的自定义逻辑。
* src，编辑器界面的代码以及编辑器层EditorLayer，启动时会加入LayerStack，执行编辑器的更新逻辑和UI渲染。HazelEditorApp继承Application，创建唯一的应用程序单例。

# **Hazel-ScriptCore —— 引擎脚本系统，调用引擎的C++代码，并提供C#API，供Editor层的脚本使用。**

C#项目，类似于Unity为用户提供的C#接口，内部有Component，Input，Vector等攻击，这些脚本会被编译为动态库，随程序运行时加载。SandboxProject会引用这个项目，使得用户自定义的逻辑可以使用核心脚本中的API，类似于在Unity中编写自定义脚本需要Using UnityEngine。

**功能梳理**

# **程序入口（EntryPoint）与测试沙盒（Sandbox）**

引擎相关代码写在Hazel项目中，测试代码写在Sandbox项目中（Sandbox项目最后弃用，使用Hazel-Editor项目充当该角色），Sandbox项目为启动项目，相当于引擎的前端。程序入口点（EntryPoint.h，内含main函数）为一个头文件，位于Hazel项目中，Sandbox项目引用这个头文件，这样做的用意是：

**1. 核心代码的复用**

Hazel项目被设计成一个独立的库，它包含所有的核心逻辑。Sandbox 作为一个测试或示例项目，它可以直接链接 Hazel库，Hazel代码可以被多个不同的项目复用，而无需在每个项目中重复编写或复制核心功能。

**2. 分离逻辑和测试/运行环境**

* Hazel项目专注于核心功能的开发，不涉及具体的应用场景，例如 UI、输入等。
* Sandbox 主要用于测试 Hazel的功能， 这样可以保证 Hazel代码的纯净，减少对外部环境的依赖，使其更易于维护和移植。

**3. 程序入口点作为头文件**

由于 Sandbox 只是 Hazel的一个测试环境，它可能并不真正控制程序的入口逻辑，而是希望 Hazel定义应用的启动方式。这样，Sandbox 只需要关心 Hazel提供的接口。

**4. 便于不同的前端环境（如引擎编辑器）调用**

* 如果 Hazel作为一个游戏引擎核心，可能会有多个不同的前端：
  + Sandbox：用于调试和运行单个游戏实例
  + Editor：用于编辑器模式
* 通过这种 EntryPoint.h 方式，不同的前端可以共享 Hazel的启动逻辑，而又能够以不同的方式初始化自己的环境。

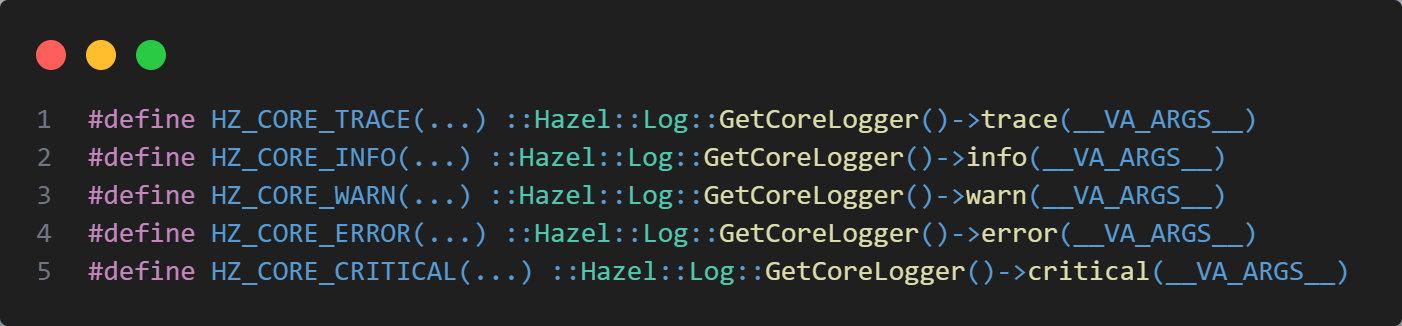
# **日志系统**

* 简单封装了spdlog库，提供CoreLogger和ClientLogger，Hazel项目使用CoreLogger，打印的日志会有HAZEL前缀，测试项目使用ClientLogger，打印的日志会有APP前缀。
* 提供宏简化使用，共有5个等级，从上到下重要性递增。

细节：

1、宏变长参数。…表示宏有任意数量的参数，\_\_VA\_ARGS\_\_会被替换为宏调用时传入的参数。

2、Hazel前的双冒号。表示在全局命名空间下寻找，防止错误。例如，在APP命名空间内有一个Hazel类，如果Hazel前没有双冒号，那么使用该宏时，就会找到APP命名空间下的Hazel类，导致错误。



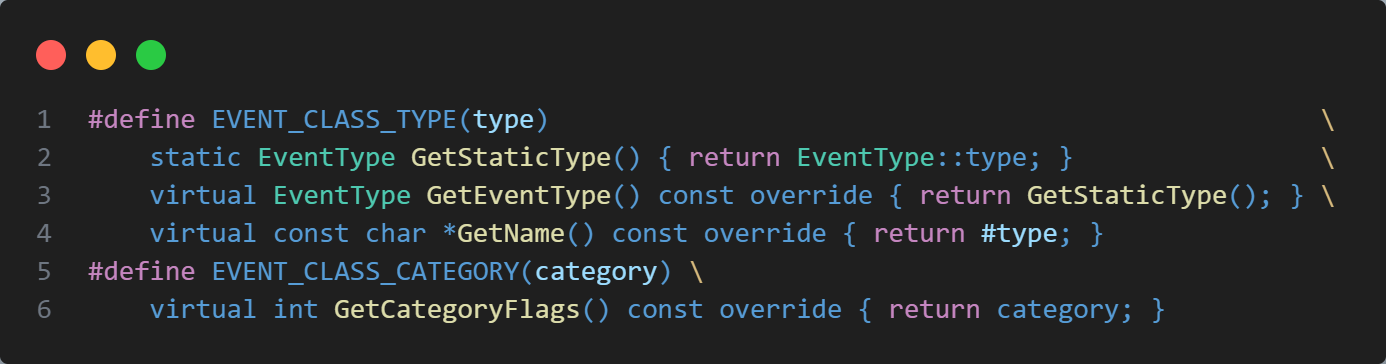
# **事件和窗口**

## **事件系统**

### **事件抽象类Event**

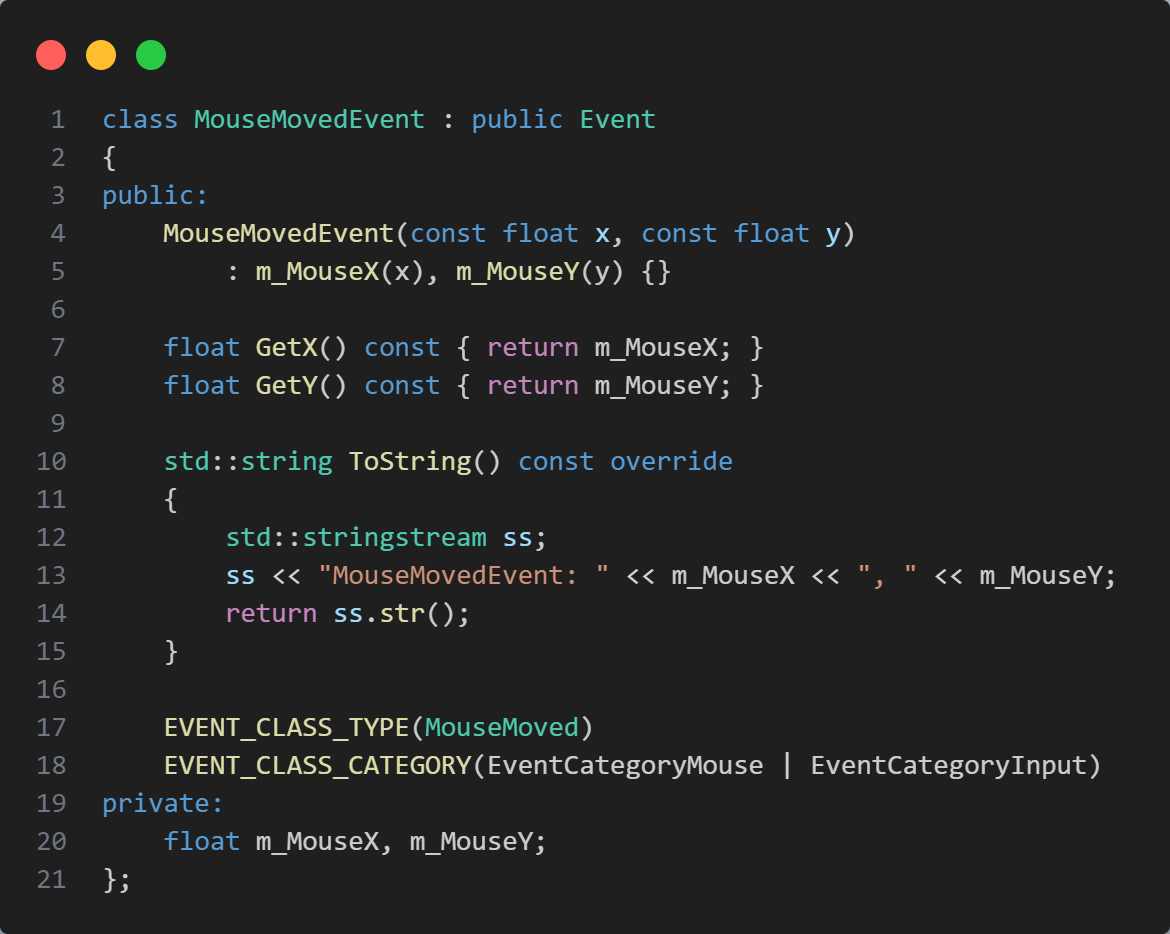
提供访问事件Type和事件Category的接口，其中事件Type代表具体的类型，例如鼠标按下，键盘抬起等；事件Category是事件的类别，例如键盘事件、输入事件，采用位运算管理。例如按下键盘，既是键盘事件，又是输入事件，可表示为EventCategoryKeyboard | EventCategoryInput。

事件抽象类中的每个接口都要重写，使用宏避免大量重复的代码。#type表示将传入的宏参数转化为const char\*。事件的派生类只需使用这两个宏就可以定义自身的具体类型和事件类别。



### **事件具体类**

分为鼠标事件、键盘事件、应用程序事件，它们都继承事件抽象类Event，并扩展自己的逻辑，下面是鼠标移动事件的例子。

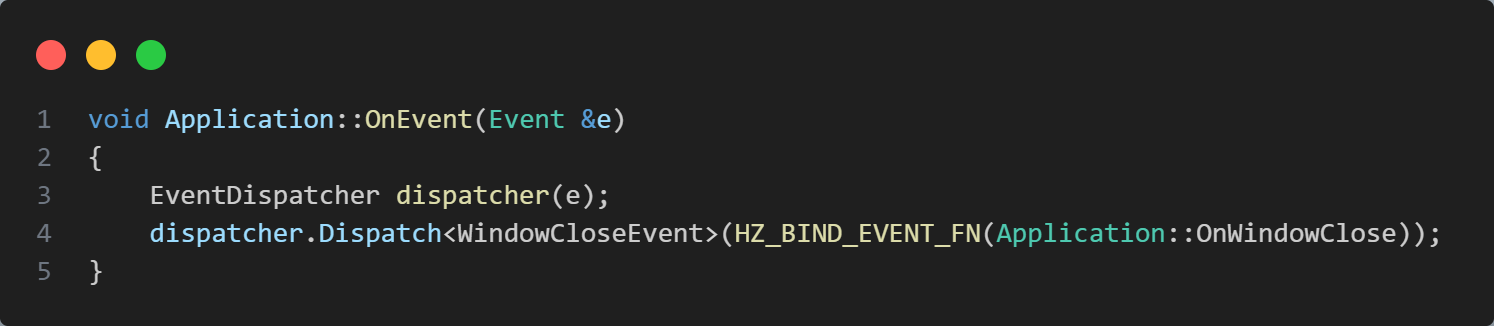


### **事件转发器EventDispatcher**

在用到事件系统的地方使用事件对象初始化EventDispatcher对象，根据事件类型是否与转发器的模版参数匹配来决定是否处理该事件。Dispatch函数接受一个可调用对象作为事件处理回调函数，返回true代表该事件被捕获。这个传入的可调用对象必须返回一个bool值，如果可调用对象返回true，Handled会置为true，意思是希望将事件的传播阻断在本层；如果可调用对象返回false，意思是事件可以继续传播。例如我们不希望UI层的事件传递到场景层，那么UI层的事件处理回调函数就要返回true，接下来可通过Handled决定是否让事件继续传递。



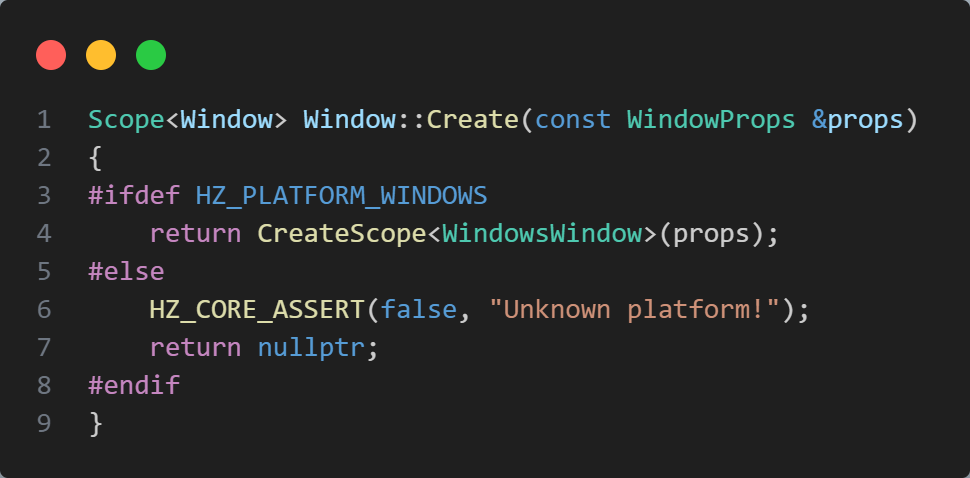
使用示例：EventDispatcher对象希望捕获一个WindowCloseEvent事件，事件发生时回调Application::OnWindowClose函数。事件发生时，Dispatch函数内部会判断该事件是不是WindowCloseEvent事件，如果是，则捕获成功，回调Application::OnWindowClose函数。



## **窗口系统**

### **抽象窗口**

定义了一系列平台无关的API，例如获取窗口宽高，设置我们自定义的回调函数。回调函数形式是参数为Event对象，返回值为void。在平台层可以继承该类，并定义具体的窗口，例如Windows系统使用Windows系统原生窗口，苹果系统使用苹果系统原生窗口。本项目暂时只支持Windows系统，并使用GLFW作为窗口。在创建窗口时，使用宏进行选择性编译，创建对应平台的窗口，如下所示：



### **具体窗口（平台特定窗口）**

暂时只支持Windows系统，具体窗口为GLFW。内部有一个WindowData结构体，含有窗口的标题、宽高、回调函数。在Init函数中初始化glfw窗口，并使用glfwSetWindowUserPointer(m\_Window, &m\_Data)定义窗口用户指针，方便接下来glfw窗口的原生回调函数的注册。注册回调函数时，只需使用glfwGetWindowUserPointer取出WindowData结构体指针，并调用我们自定义的回调函数。注册glfw回调的示例如下，当发生窗口Resize的事件时，glfw会响应Lambda表达式内的内容，取出WindowData结构体指针，重新设置窗口的宽高，并产生一个WindowResizeEvent事件，以该事件作为参数，调用我们自定义的回调函数。

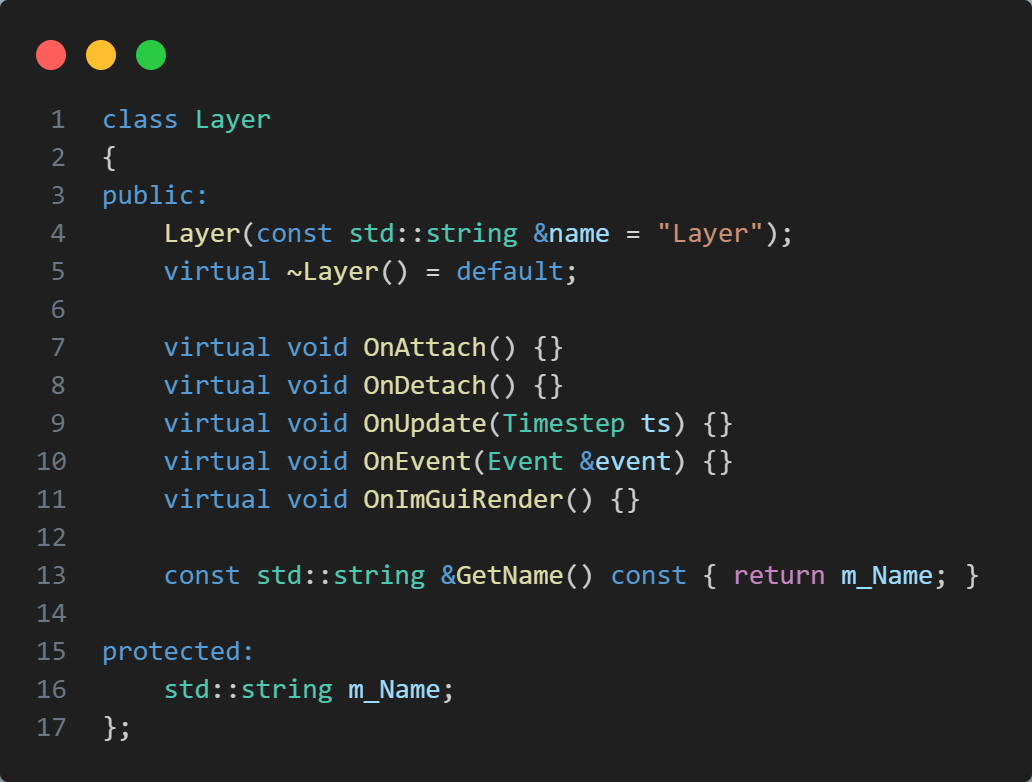


# **层（Layer）和层栈（LayerStack）**

## **层（Layer）**

可以类比为PS中的图层，上层可以覆盖下层，每一层处理对应的事件，也可以决定事件是否继续向下传播。例如点击UI发生的事件，一般不想让这个点击事件影响到游戏场景，所以就在UI层处理事件后阻断事件的传播。

层的定义如下。OnAttach和OnDetach分别定义层进入层栈和弹出层栈时应该执行的逻辑，OnUpdate会每帧调用，OnEvent处理本层的事件，OnImGuiRender负责本层的UI渲染。



## **层栈（LayerStack）**

层栈是一个vector，是层的容器。遍历时从尾部开始，即越靠近尾部的层会先开始处理。PushLayer是在m\_LayerInsertIndex的位置插入层，插入过程中会调用层的Attach函数，插入后m\_LayerInsertIndex会自增，指向下次插入时的位置。PushOverlay是插入到尾部，即覆盖图层，处理优先级最高。一般来说，容器的前半部分为普通层，可能是游戏场景；容器的后半部分是覆盖层，一般是编辑器和UI，事件需要先经过覆盖图层，再决定是否要传入到游戏场景。

PopLayer是根据传入的层，在容器前半部分（m\_LayerInsertIndex之前）进行查找，删除传入的普通层；PopOverlay是根据传入的层，在容器后半部分（m\_LayerInsertIndex之后）进行查找，删除传入的覆盖层。



## **ImGUI层**

OnAttach初始化了ImGUI的环境，定义了主题，字体，窗口上下文等；OnDetach销毁ImGUI的环境。Begin在每帧渲染之前调用，清除之前的UI数据，准备新一帧的渲染；End在每帧渲染之后调用，结束当前帧的UI渲染，并将数据提交给ImGUI内部的渲染器。这些代码不用深究，以抄官方示例为主。

BlockEvents表示ImGUI层是否阻断事件，这个函数在编辑器层调用。因为最后ImGUI会覆盖整个窗口，中间会留一个ImGUI的视口区，场景会渲染到后台的帧缓冲，再从帧缓冲取出，渲染到ImGUI的视口区。因此，所有的事件都会被ImGUI层捕获，所以需要决定某些事件是否需要继续传递。具体为，如果鼠标位于视口区，就认为用户想操作场景，此时调用BlockEvents函数将m\_BlockEvents设为false，不阻断事件，否则将阻断事件，由ImGUI层处理事件。

ImGUI层在应用程序初始化时会插入层栈的尾部，每帧最先处理。

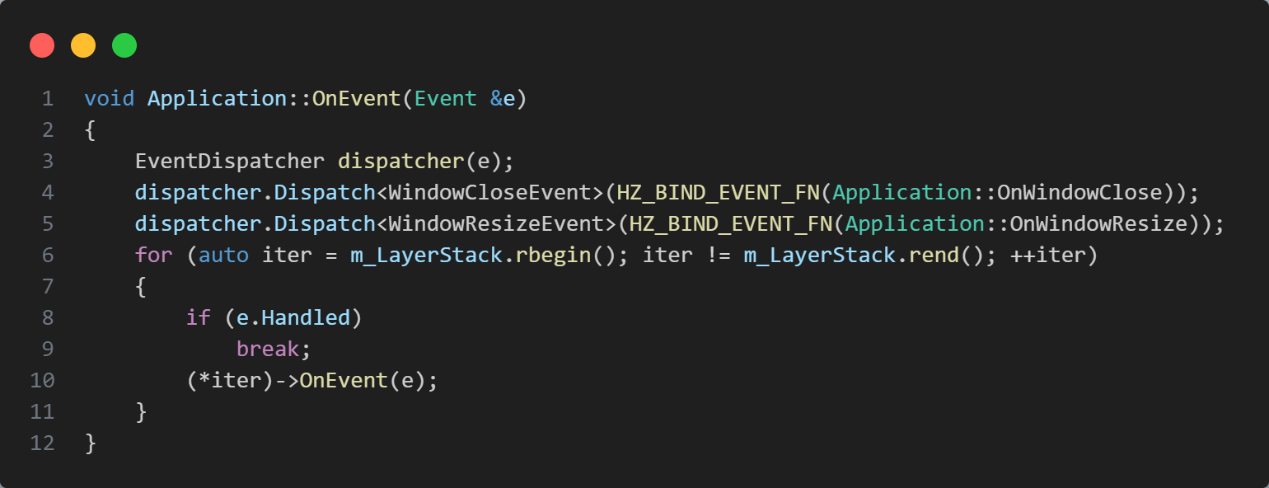


# **Application与引擎运行的流程**

测试项目Hazel-Editor（最开始是Sandbox）中定义了一个类HazelEditor，继承Application，并定义了CreateApplication函数，new出HazelEditor对象。在EntryPoint.h中的main函数中调用CreateApplication，就实例化了全局唯一的Application对象。HazelEditor类先调用父类Application的构造函数，Application与窗口之间采用观察者模式，Application在构造函数中实例化出窗口，并设置Application::OnEvent为回调函数，窗口并不知道Application的存在，它只需要发生事件时调用Application注册的回调函数即可。Application的构造函数中还会进行渲染器、脚本系统等一系列初始化，最后会使用PushOverlay函数，将ImGUI层放在层栈尾部，最优先处理。父类Application构造完毕后，在HazelEditor自己的构造函数中，会将编辑器层（EditorLayer）使用PushLayer的方式插入层栈，在ImGUI层之后处理。

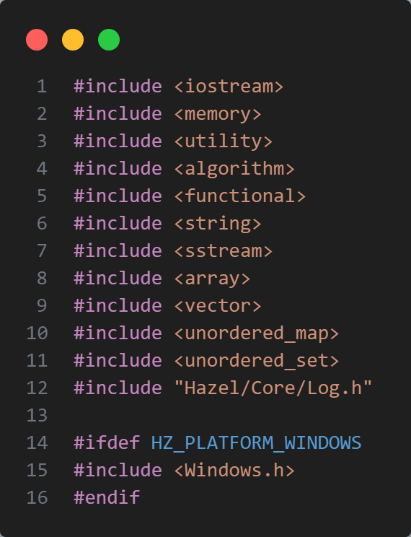
Application初始化完成后，会调用Run方法，开始每帧的循环。每帧会反向遍历层栈，调用每一层的OnUpdate方法，在编辑器层的OnUpdate方法中，会调用渲染命令将场景渲染到帧缓冲中。每一层更新完毕后，又会反向遍历层栈，调用OnImGuiRender方法更新UI，在编辑层的OnImGuiRender方法中，会取出帧缓冲中的数据，渲染到视口区域。因此，整体上说，OnUpdate是逻辑处理（虽然内部使用了渲染命令，但没有立即呈现到屏幕上），OnImGuiRender是渲染处理。UI渲染全部完毕后，每帧的最后会执行窗口的OnUpdate方法，执行glfwPollEvents检查是否有事件发生，执行glfwSwapBuffers交换颜色缓冲。

如果有事件发生，glfwPollEvents就会检测到。在初始化窗口时，窗口的Init方法内部就已经注册好了glfw的回调函数，回调函数中又会调用Application::OnEvent。所以有事件发生时，Application中的OnEvent方法就会被调用。如下所示，在Application中直接检测的是窗口关闭和Resize事件。如果是窗口关闭事件，下一帧将跳出Run函数的循环，程序会结束，如果是Resize事件，会调用渲染器的OnWindowResize方法，改变视口大小。接下来会反向遍历层栈，调用每一层的OnEvent去处理事件，直到该事件被处理，则跳出循环。在ImGUI层的OnEvent函数中，如果鼠标不在视口区域，则ImGUI层会阻断事件的传播，所以在调用Editor层的OnEvent函数之前循环就会跳出，就不会让操作UI影响到场景。



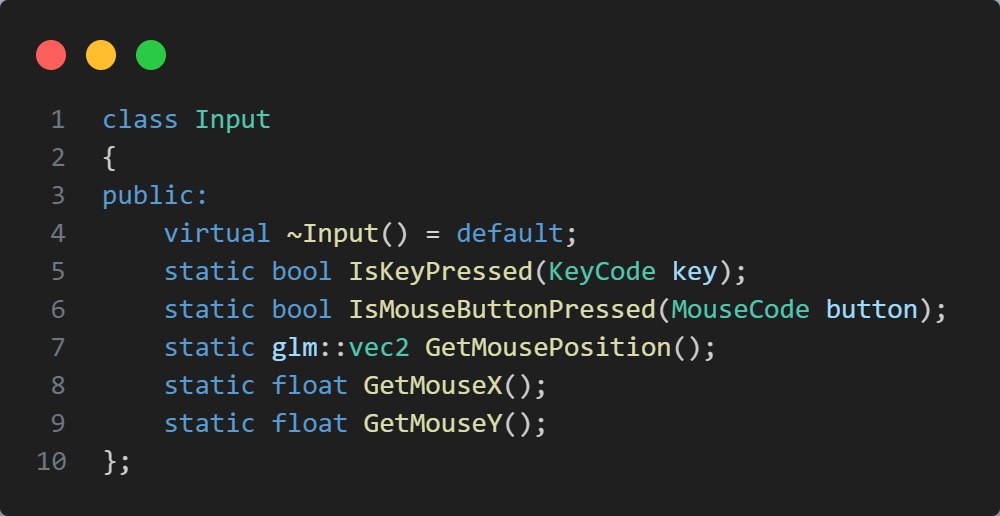
# **预编译头**

大型项目中，某些标准库头文件可能被反复引入，每次编译时都会重新解析、语义分析、编译这些头文件，重复且耗时。可以创建一个头文件，包含了不会频繁更改的常用头文件（如pch.h），如下所示。在CMake中，采用target\_precompile\_headers指定要处理为预编译头文件的路径，构建时编译器将路径下的文件进行预编译。项目中的其他 .cpp 文件只需包含 pch.h，编译器在看到它时，直接加载预编译好的结果，不再重复编译，从而提升整体编译速度。在使用预编译头的 .cpp 文件中，必须将 pch.h 放在最上面。



# **输入轮询系统**

与事件系统的区别是，事件是触发后调用回调函数，执行某些逻辑，整体是被动的；输入轮询系统是主动检测是否产生了某个输入。Input是一个工具类，如下所示。在不同平台上定义对应的cpp文件实现这些函数。例如Windows平台上就定义WindowsInput，并根据使用的窗口返回是否产生输入。未来可以继续扩展到其他平台和其他窗口。



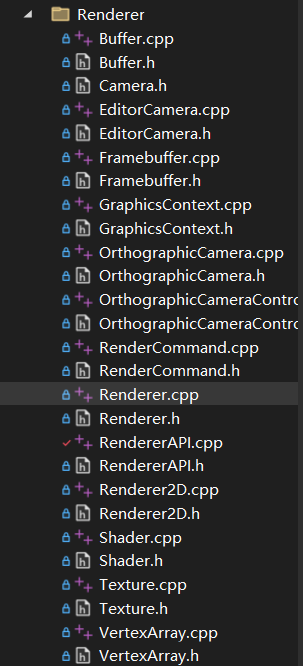
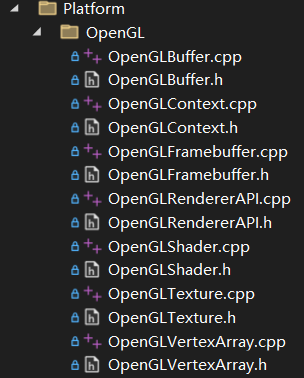


# **渲染系统**

## **总览**

Hazel/Renderer目录下的代码都是平台无关的，负责提供统一的接口。在Platform下的代码会继承Hazel/Renderer目录下的类，做特定平台的实现。目前只支持OpenGL，所以Platform/OpenGL下的类就是Hazel/Renderer下的类在OpenGL上的具体实现。例如VertexArray类提供了Bind接口，那么OpenGLVertexArray类就会实现Bind接口，内部的具体实现就使用了glBindVertexArray这个OpenGL的函数。

Renderer类是渲染系统中最上层的模块，负责初始化渲染命令，初始化渲染器，提交DrawCall等。但由于教程中渲染系统做到一半时决定主要先实现2D渲染功能，在之后加入编辑器后Renderer类逐渐弃用，上层模块直接调用Renderer2D类提供的画四边形的接口进行测试。

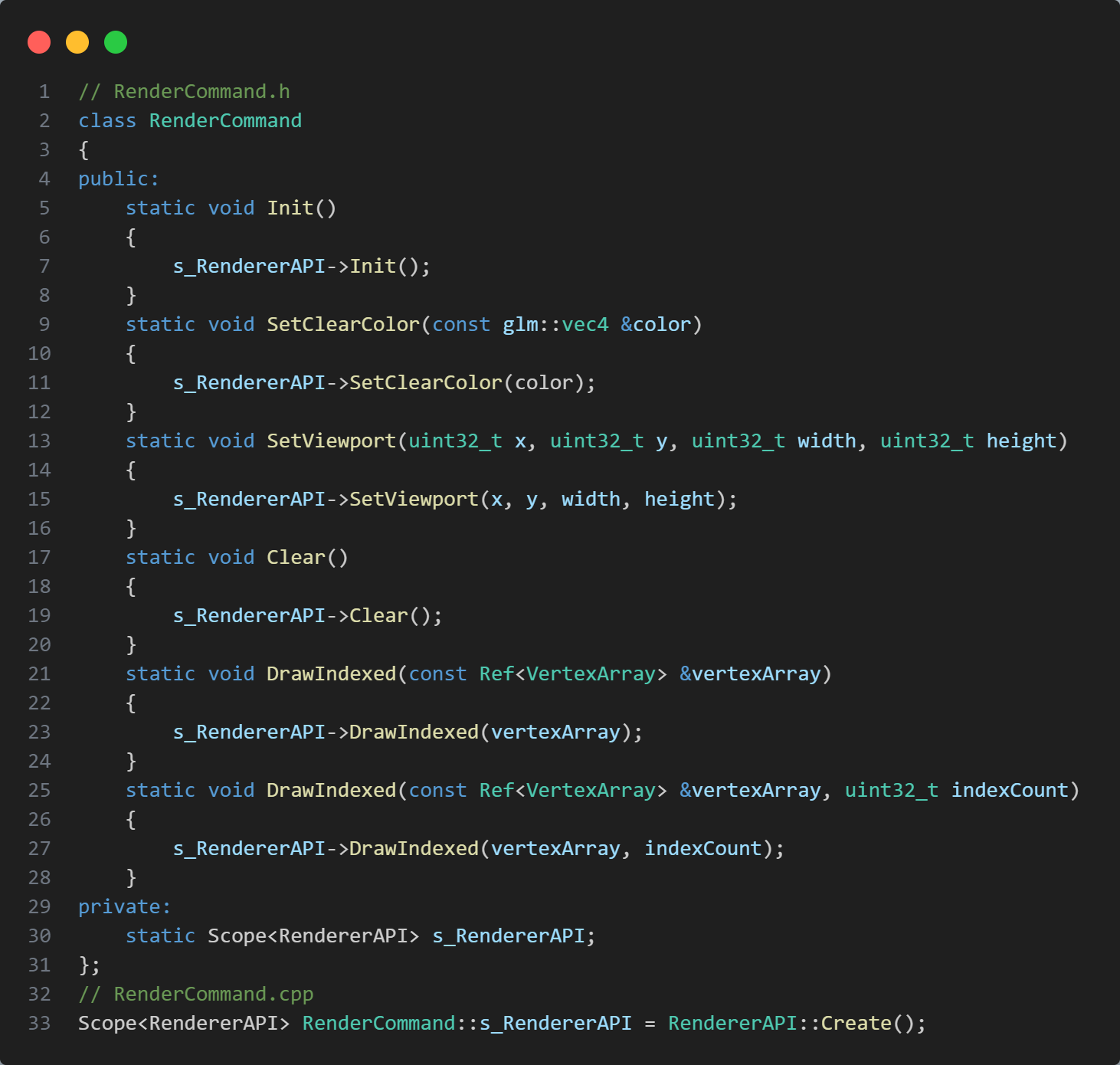
## **渲染接口**

### **抽象渲染接口**

RendererAPI中的s\_API是一个静态成员，定义在全局作用域下，在main函数执行前就会初始化为OpenGL接口，在RenderCommand中通过调用Create函数就可以将渲染API初始化为OpenGL（**注意，这里可能会产生C++中的静态初始化顺序问题，当你在多个不同的 cpp 文件中定义静态变量，并且它们之间存在依赖关系，比如一个静态变量在初始化时使用了另一个 cpp 文件中的静态变量，那么它们的初始化顺序是未定义的，你有可能在另一个静态变量尚未初始化时就使用了它，教程中没有处理，但也没有出现问题，暂时先这样**）。

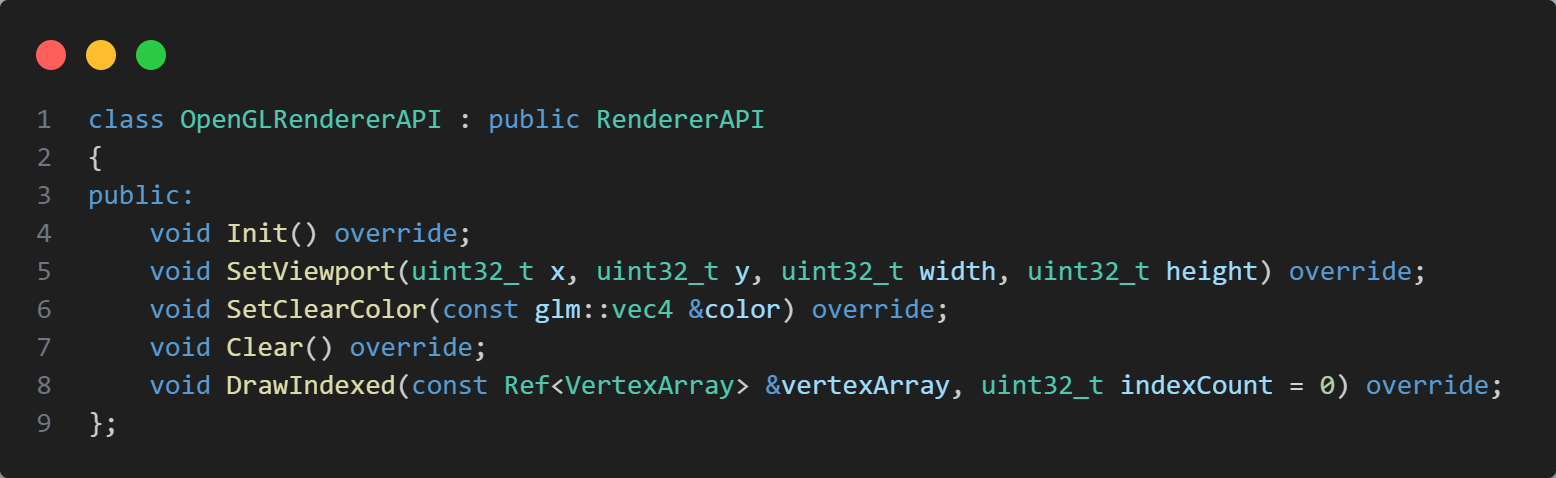
RenderCommand是向渲染器和其他上层模块提供渲染命令的，经过了渲染接口的抽象，渲染器只需调用RenderCommand提供的绘制命令，无需关心具体的图形API。例如，渲染器只需调用DrawIndexed函数，由于图形API已经初始化为OpenGL，内部会调用OpenGLRendererAPI.cpp中实现的DrawIndexed函数，底层为glDrawElements。





### **具体渲染接口**

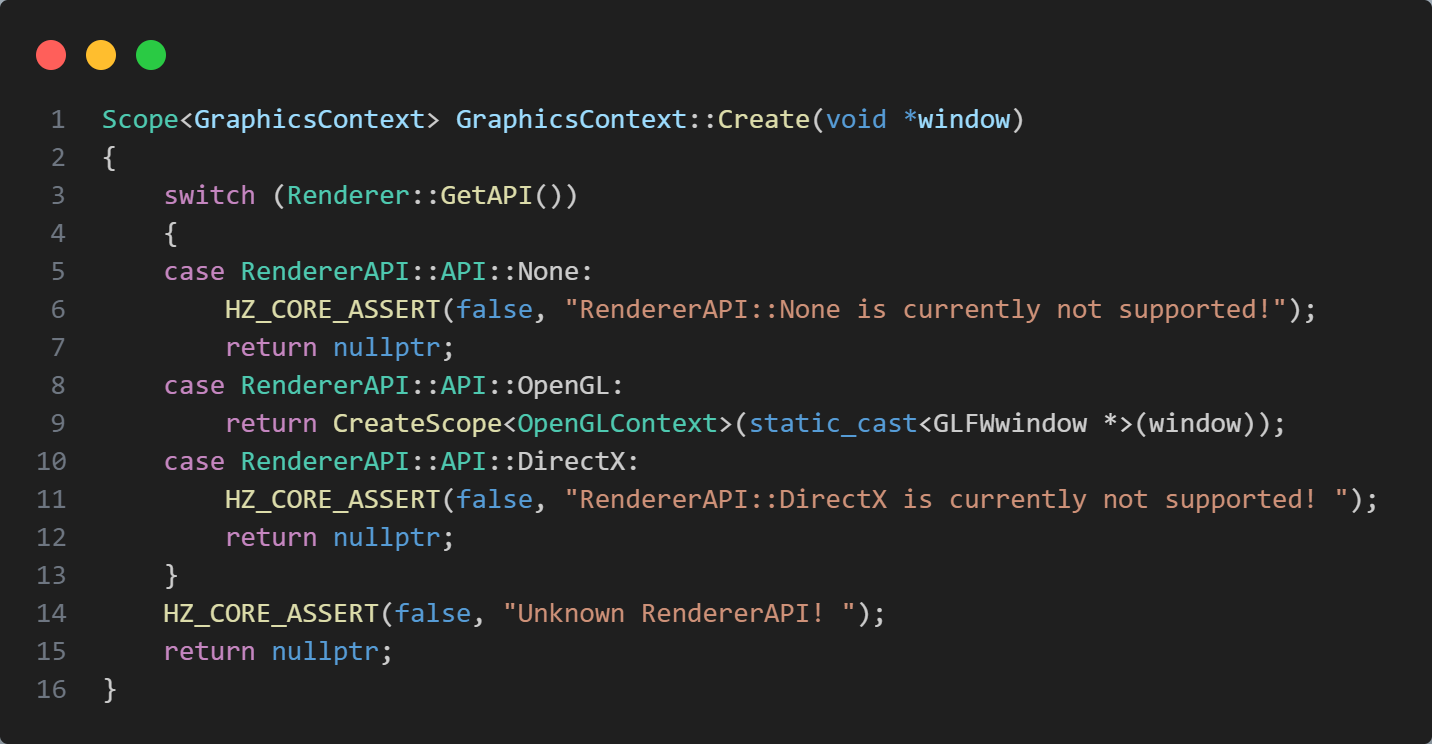
OpenGLRendererAPI实现了RendererAPI的接口，底层调用了OpenGL的函数。例如SetClearColor调用的是glClearColor，用某种颜色清空缓冲区（即背景色）；SetViewport调用的是glViewport，用于调整视口大小。



## **渲染上下文**

渲染接口的创建在main函数执行之前，是最早的，因为必须要先指定接口，之后的工作才知道去找哪个具体实现。接下来就会进入main函数，Application的构造。

在Application的构造函数中，最先初始化的是渲染上下文，这是在窗口初始化的过程中进行的，因为渲染上下文需要提供窗口句柄，即glfwMakeContextCurrent(window)，window就是当前窗口的句柄。上下文的初始化是类似的，根据渲染接口选择具体的渲染上下文进行创建，如下所示。由于渲染接口已经初始化为OpenGL，那么创建的就是OpenGL渲染上下文，上下文中含有GLFW窗口句柄。



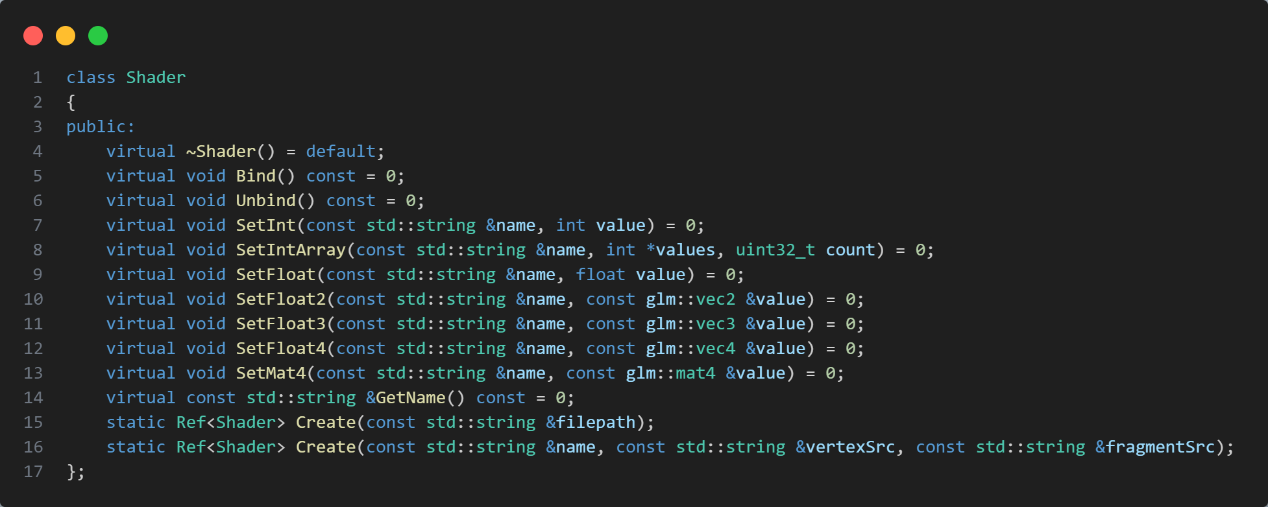
在OpenGLContext的初始化函数中，会使用glad加载OpenGL函数的地址，在这之后就可以使用OpenGL的函数了。此外，OpenGLContext的主要工作是使用SwapBuffers进行窗口双缓冲交换，每帧在窗口的OnUpdate方法中调用。



## **着色器**

### **抽象着色器**

封装方式类似于LearnOpenGL，提供图形API无关的接口。Create时根据图形API创建具体着色器（项目中是OpenGL着色器），可以通过shader文件或字符串创建。



### **具体着色器**

OpenGL着色器中的m\_RendererID就是GPU着色器对象句柄。教程中规定着色器文件的格式为，以#type开头，空格之后接vertex，下面就是顶点着色器；再以#type开头，空格之后接fragment，下面就是片元着色器。所以PreProcess函数就是做上述着色器文件解析工作的。如果文件格式正确，返回的哈希表长度为2，key分别是GL\_VERTEX\_SHADER和GL\_FRAGMENT\_SHADER，value就是顶点着色器和片元着色器对应的字符串。Compile函数接受这个哈希表作为参数，就可以取出顶点着色器和片元着色器进行编译了。

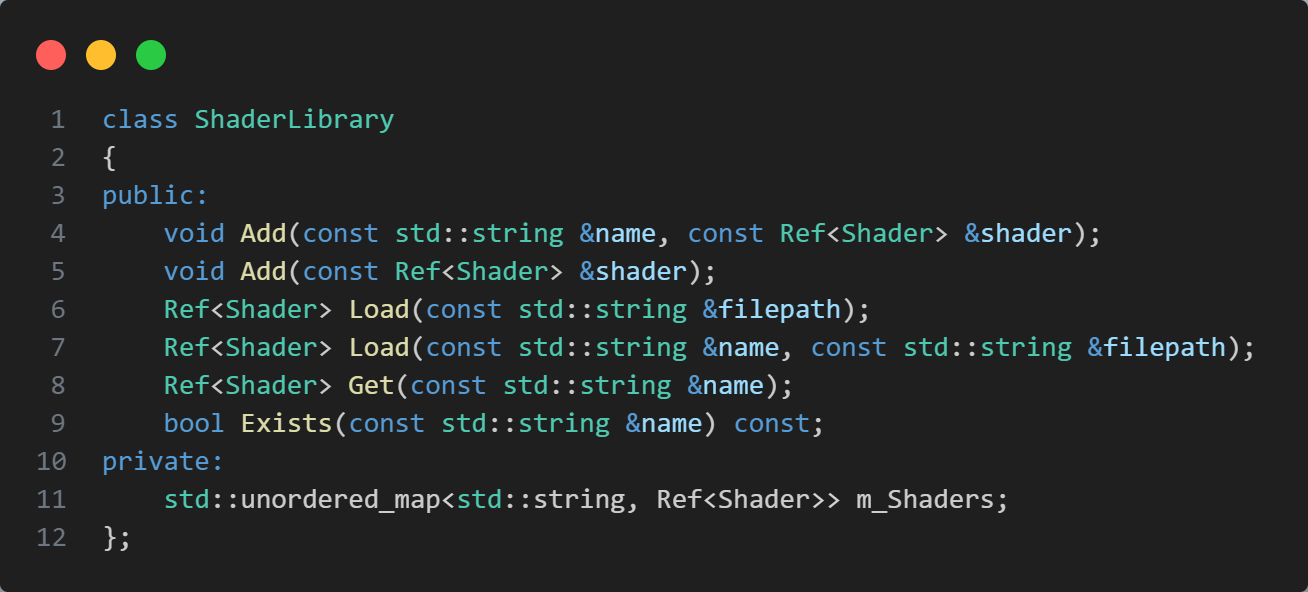
着色器编译以及抽象Shader接口的实现都与LearnOpenGL是类似的。



### **抽象着色器资产（ShaderLibrary）**

ShaderLibrary是更高一层的抽象，上层模块只需使用Load加载着色器文件，无需关系着色器的细节，Load内部调用的是Shader的Create函数。创建好的Shader会放入哈希表进行管理，避免多次创建相同的Shader对象。需要会着色器内部的Uniform变量赋值时，就使用Get函数，传入着色器的名称，获得Shader对象，再使用Set系列接口即可。

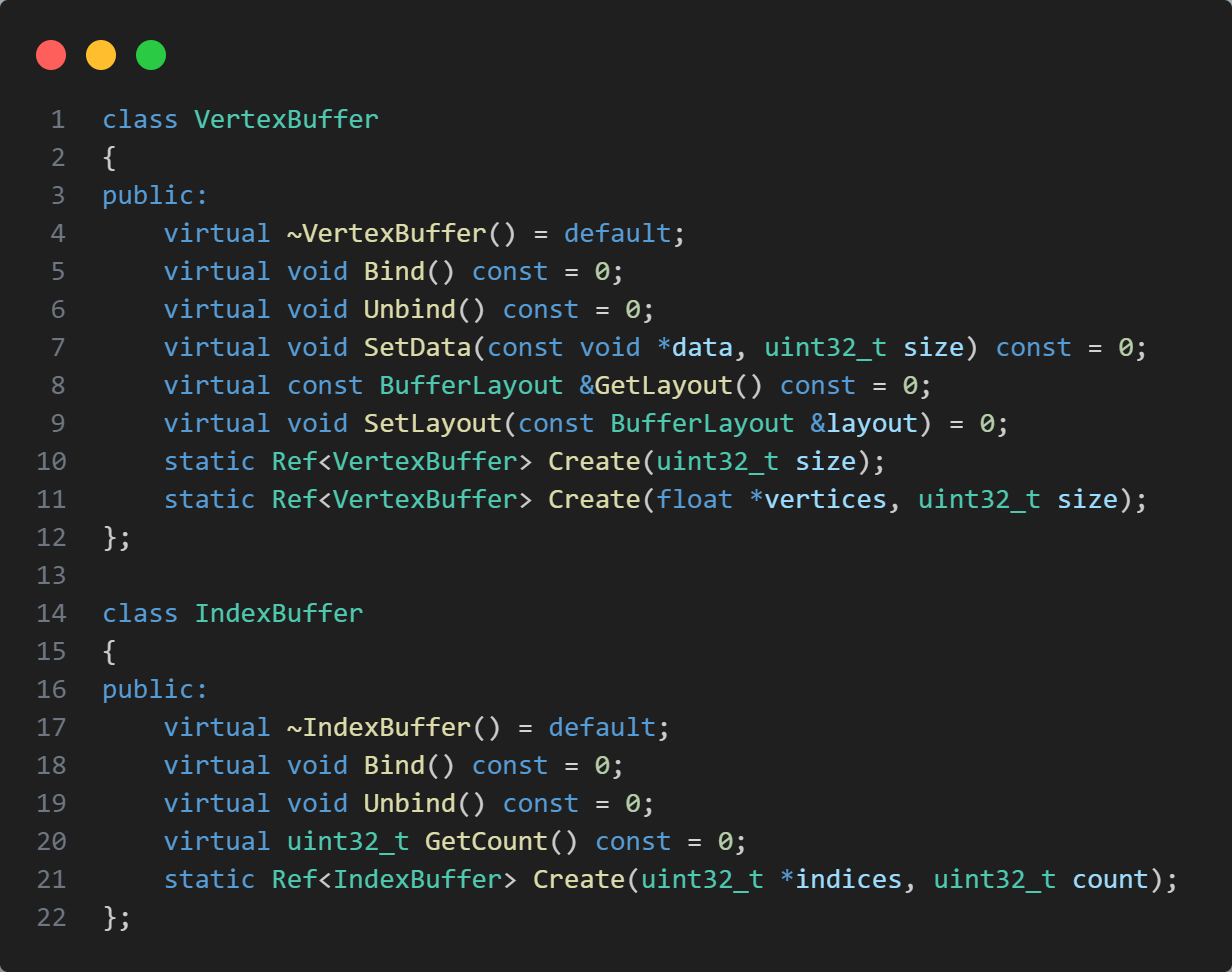
目前的编辑器测试类（EditorLayer）没有使用该类，因为场景简单，所有物体都使用了一种着色器，所以着色器暂时在Renderer2D中的渲染数据中写死，直接调用了Shader的Create函数。



## **缓冲区和缓冲区布局**

### **顶点缓冲和索引缓冲（VBO和EBO/IBO）**

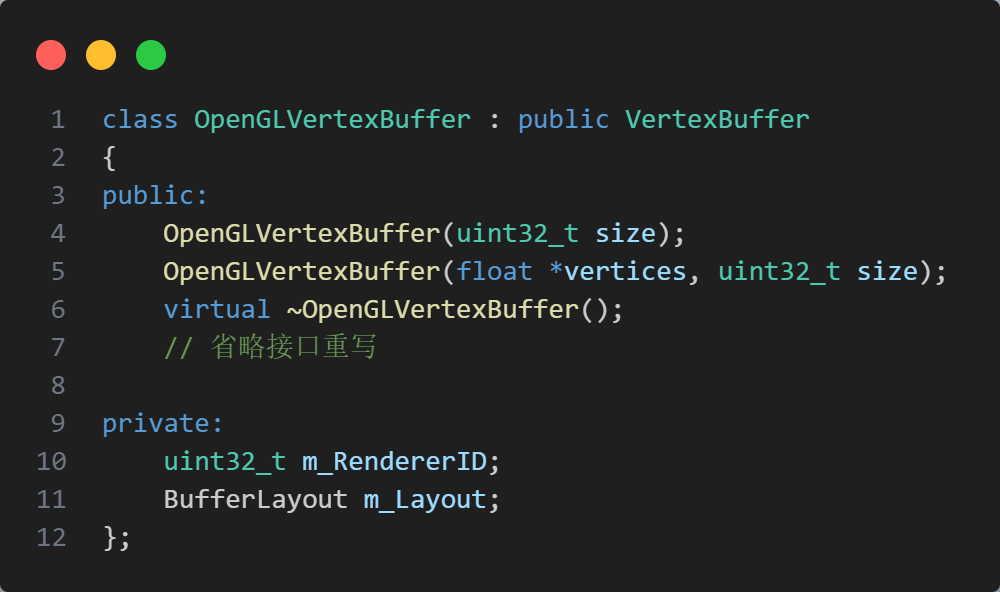
下面是抽象的VBO和EBO对象，Create函数是类似的，根据渲染API创建具体图形API的VBO和EBO。VBO的SetData函数允许将传入的数据写入缓冲区，SetLayout可以设置顶点的布局。这两个函数会在下面具体分析。



下面是OpenGL的VBO对象，EBO是类似的。m\_RendererID是GPU端缓冲区对象的句柄，与Shader是类似的，教程中所有的GPU端对象句柄全部命名为m\_RendererID。整体上说，OpenGLVertexBuffer类就是封装了glGenBuffer，glBindBuffer，glBufferData系列函数。在VBO和EBO的构造函数中，教程中使用了glCreateBuffers而不是glGenBuffers，glCreateBuffers是OpenGL4.5引入的函数，创建缓冲区后无需绑定，而是使用glNamedBufferData(buffer, size, data, GL\_STATIC\_DRAW)直接操作缓冲区。但是教程中没有这么做，使用了glCreateBuffers后还是使用了glBindBuffer绑定缓冲区，总体来说是传统的方式。

第一个构造函数只传入了缓冲区大小，没有传入缓冲数据，所以在glBufferData中倒数第二个参数指定为nullptr，意思是只开辟缓冲区，暂时不传入数据，在之后使用SetData，传入顶点数据，内部使用了glBufferSubData改变缓冲区的内容。由于数据可能会经常改变，所以构造函数中glBufferData的最后一个参数应指定为GL\_DYNAMIC\_DRAW，适用于粒子，布料等顶点经常改变的场景。

第二个构造函数就是常见的方式，用于静态的模型，与LearnOpenGL是一致的，在glBufferData时就传入顶点数据，并且将最后一个参数指定为GL\_STATIC\_DRAW。



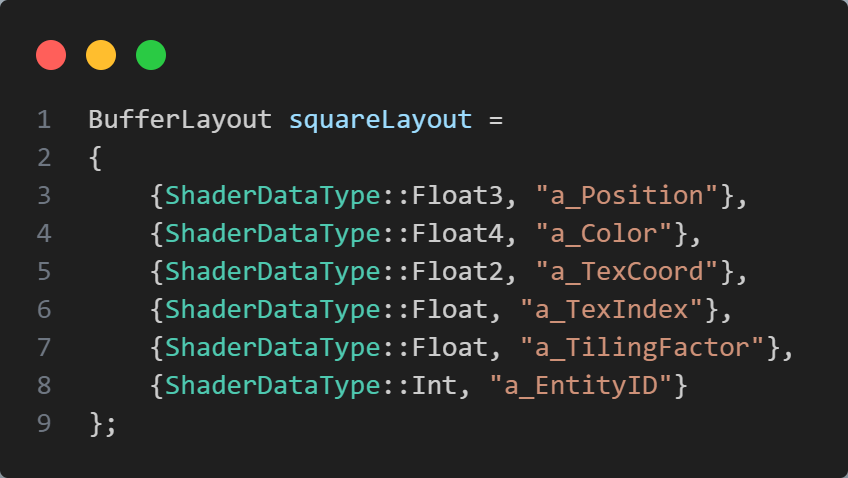
### **缓冲区布局**



举个例子，顶点数据可能由位置3个float（位置），4个float（颜色），2个float（纹理）组成，所以BufferElement描述的就是顶点数据中每个属性的信息，例如位置属性的Name为position，数据类型是Float3，size是4\*3，偏移量是0，Normalized为false（不归一化），这些信息和glVertexAttribPointer是一一对应的。

BufferLayout对象内部的vector就存储了多个BufferElement。以上述顶点为例，vector就会存储3个BufferElement，每个BufferElement对象内部就含有对应属性的信息。

使用BufferLayout的示例如下，传入数据类型和属性名称后，CalculateOffsetsAndStride函数会自动将每个BufferElement的属性size，偏移量解析出来。一个顶点的大小（步长,m\_Stride）最后也会计算出来。这些数据准备以后，会用于VAO对象中。



## **顶点数组对象（VAO）**

VAO和之前的GPU相关资产是类似的，抽象的VertexArray负责定义接口，OpenGLVertexArray实现这些接口。在OpenGLVertexArray中，m\_VertexBufferIndex代表顶点属性编号，即glEnableVertexAttribArray时传入m\_VertexBufferIndex激活该属性，并在glVertexAttribPointer的第一个参数指定要操作第几号顶点属性（例如位置是0号属性，颜色是1号属性），这些属性编号会在顶点着色器中的layout（location=X）一一对应。

VAO中使用了vector存储多个VBO对象。如果VBO对象采用interleaved buffer形式，即单个VBO中储存了位置、颜色、纹理等，那么一般来说一个VAO只对应一个VBO；如果VBO对象采用single buffer形式，即第一个VBO存储位置，第二个VBO储存颜色等等，那么在AddVertexBuffer就会调用多次，vector中就会有多个VBO对象。项目中目前全部使用的是interleaved buffer。



SetIndexBuffer比较简单，就是绑定EBO对象，同时保存当前VAO对应的EBO。AddVertexBuffer对应了激活顶点属性，写入顶点描述信息的过程。取出VBO中的顶点布局信息后，根据数据类型使用不同的策略写入顶点描述信息。浮点数到浮点4维向量共用一个策略，这与LearnOpenGL中的绝大部分情况是一样的（因为大部分情况下顶点数据信息都是浮点数）。整数到整数4维向量和布尔值共用一个策略，使用glVertexAttribIPointer写入顶点描述信息，与glVertexAttribPointer的区别是没有normalized参数，目前的项目中，物体的实体ID属性是整数。3\*3矩阵和4\*4矩阵共用一个策略，因为矩阵不能一次性传入，需要拆成3维向量或4维向量，每个向量占用一个顶点属性位置。

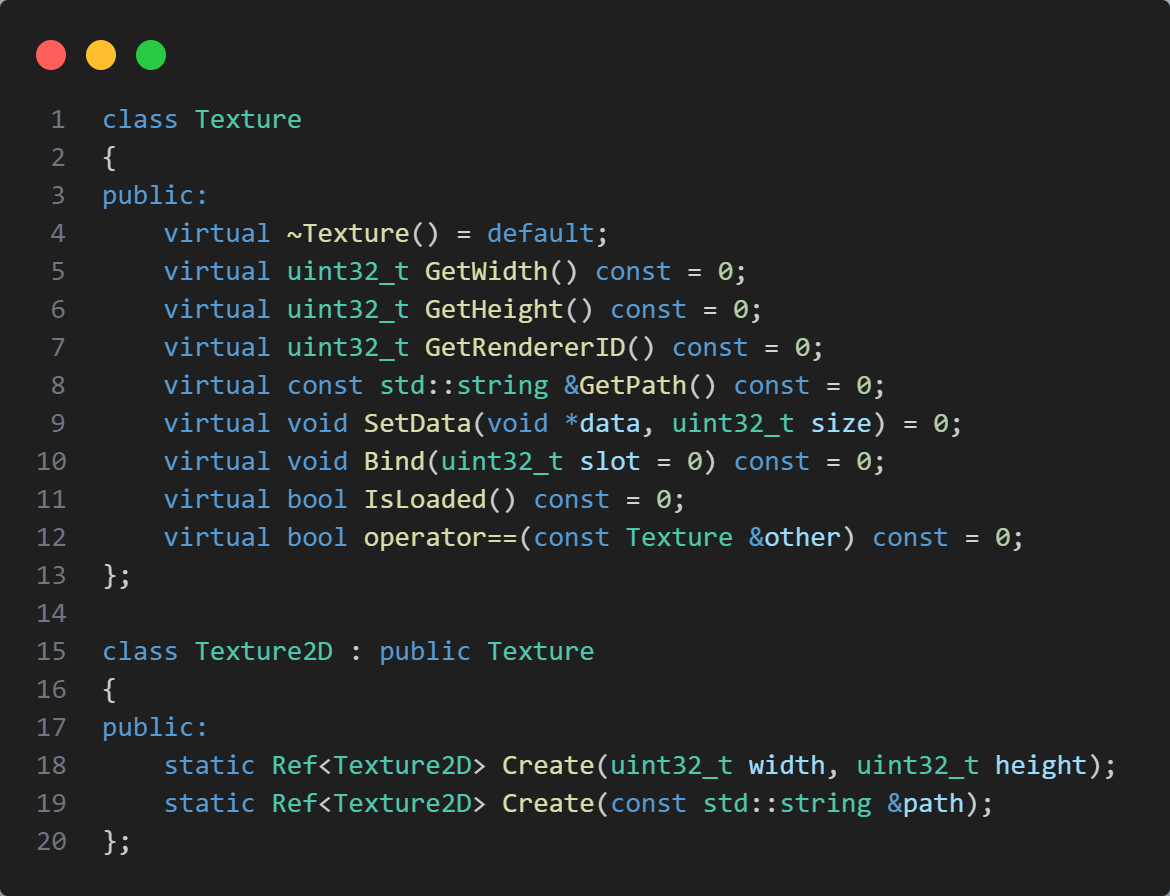
注意到在写入矩阵描述信息时，每次循环之后都使用了glVertexAttribDivisor，第一个参数是顶点属性编号，第二个参数是属性经过N个实例后更新一次，用于实例化渲染。因为顶点属性是矩阵时，往往用于实例化渲染。例如实例化渲染时，顶点位置是相同的，但我们希望每个实例的模型变换矩阵不同，假设有100个实例，就可以将100个不同的模型变换矩阵写入VBO，再写入矩阵描述信息，最后将glVertexAttribDivisor第二个参数指定为1，表示每1个实例更新一次模型变换矩阵。



## **纹理**

### **抽象纹理**

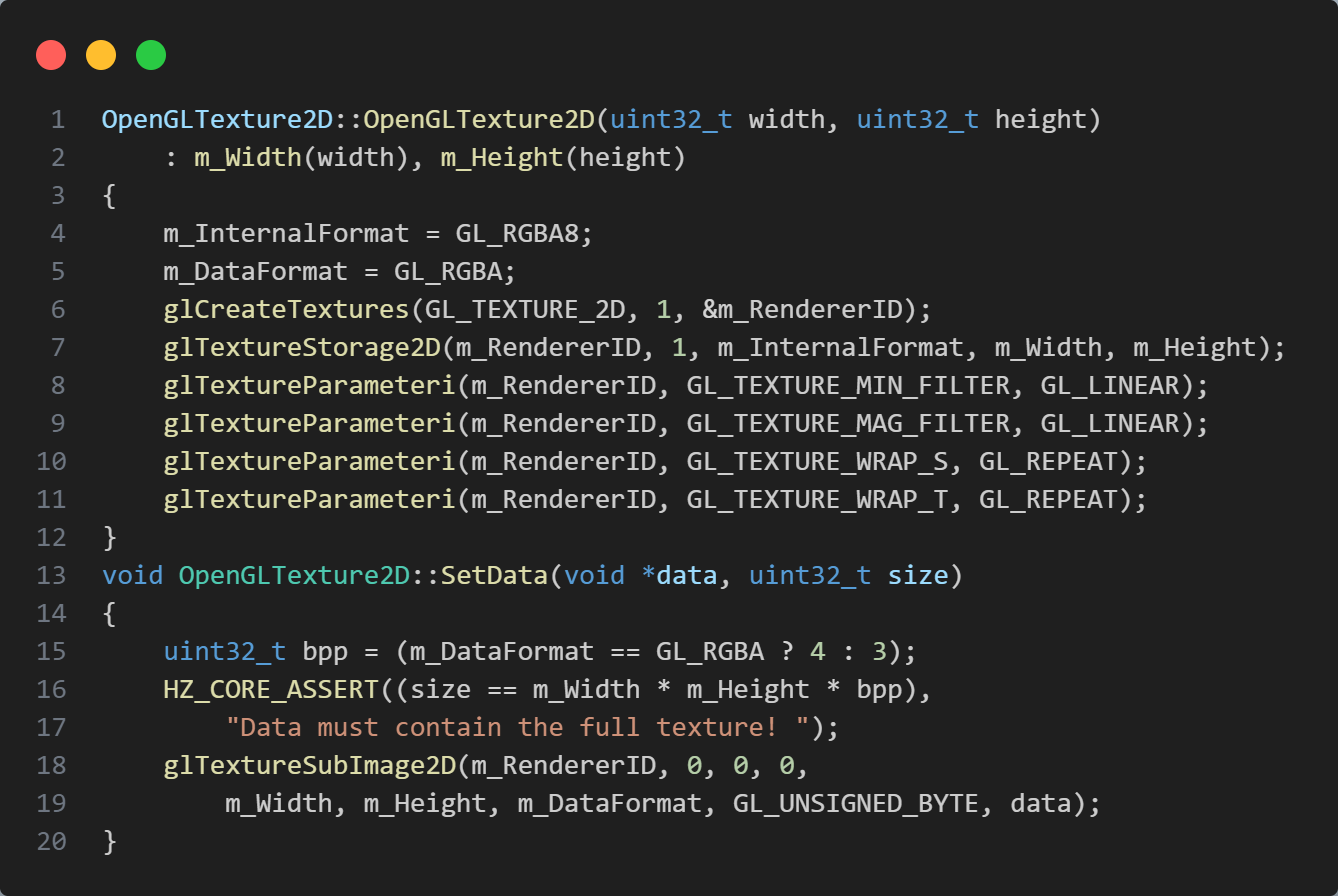
目前使用的纹理是Texture2D类，未来有可能支持更丰富的纹理，例如天空盒cubemap。第一个Create函数是开辟一片空的显存，等待之后的填充；第二个Create函数是将图片用stb\_image库读入内存后，再发送到显存。



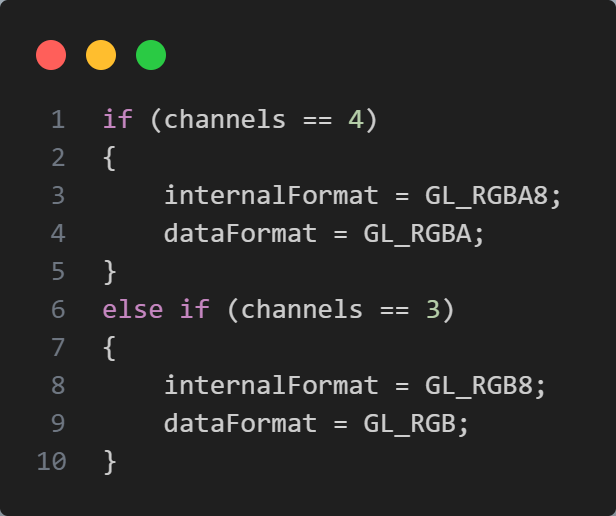
### **具体纹理**

在第一个构造函数中，使用了glCreateTextures避免了传统方式生成纹理句柄后还要绑定。glTextureStorage2D需要OpenGL4.2以上版本，它的第一个参数是纹理句柄，所以它是无需绑定的；第二个参数是要为纹理分配多少个mipmap层级，1表示不使用mipmap，仅使用基本层级；第三个参数GPU端的纹理数据格式。该函数只分配空间，不写入具体的纹理数据。后面的glTextureParameteri系列函数与传统的glTexParameteri不同的是，glTextureParameteri第一个参数是纹理句柄，所以它也是无需提前绑定的，glTexParameteri的第一个参数是目标纹理插槽，2D纹理是GL\_TEXTURE\_2D，但是我们使用glCreateTextures时已经绑定了目标纹理插槽，所以glTextureParameteri无需指定target。

后续使用glTextureSubImage2D写入纹理的数据。其中第2个参数是level，即向哪个mipmap层级的纹理写入数据，项目中暂时没有使用mipmap，所以指定为0，向初始层级纹理写入数据。第3、4个参数是写入的起点相对于x方向和y方向的偏移。



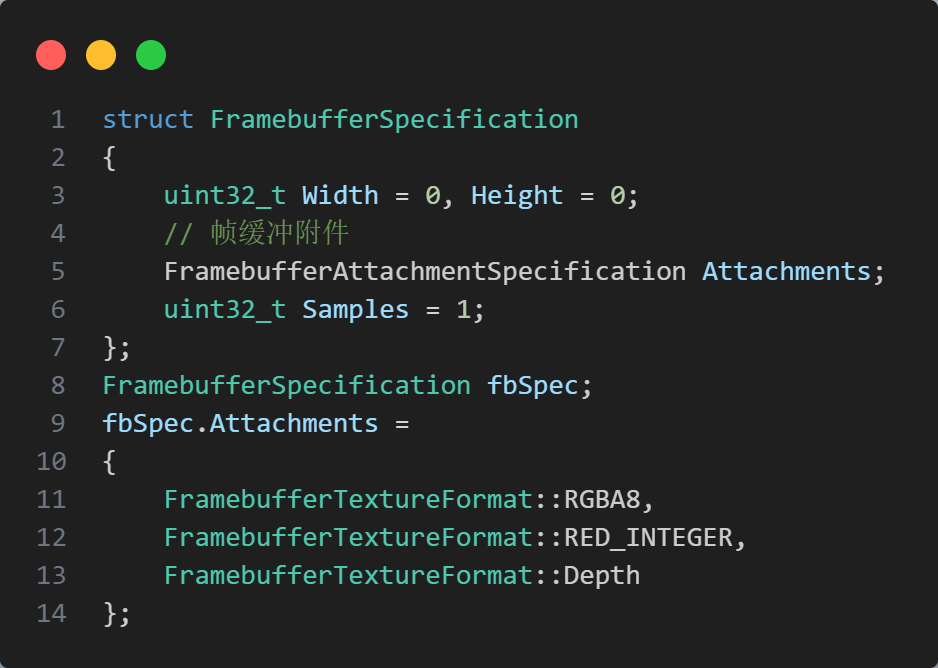
第二个构造函数提供的图片的路径，所以可以通过stb\_image读入图片后将图片传入GPU显存中，与上述的步骤基本是相同的。在读入图片的过程中，需要根据图片的通道数指定CPU端的格式（dataFormat）与GPU端的格式（internalFormat）。假设图片是3通道，在LearnOpenGL中，dataFormat和internalFormat都指定为GL\_RGB，这样其实不推荐，因为GL\_RGB对于GPU来说只是一个默认格式提示符，不是明确的格式，OpenGL 会推测成对应的精度版本（可能是 GL\_RGB8，也可能不是），依赖驱动；而指定为GL\_RGB8就明确指定了每个通道的精度，避免潜在的错误。



## **帧缓冲**

帧缓冲主要与编辑器配合使用，因为需要将场景渲染到纹理上之后再渲染到编辑器的视口上，同时鼠标点击物体选择的功能也与帧缓冲相关。这里只介绍帧缓冲对象如何封装，具体封装在编辑器部分说明。

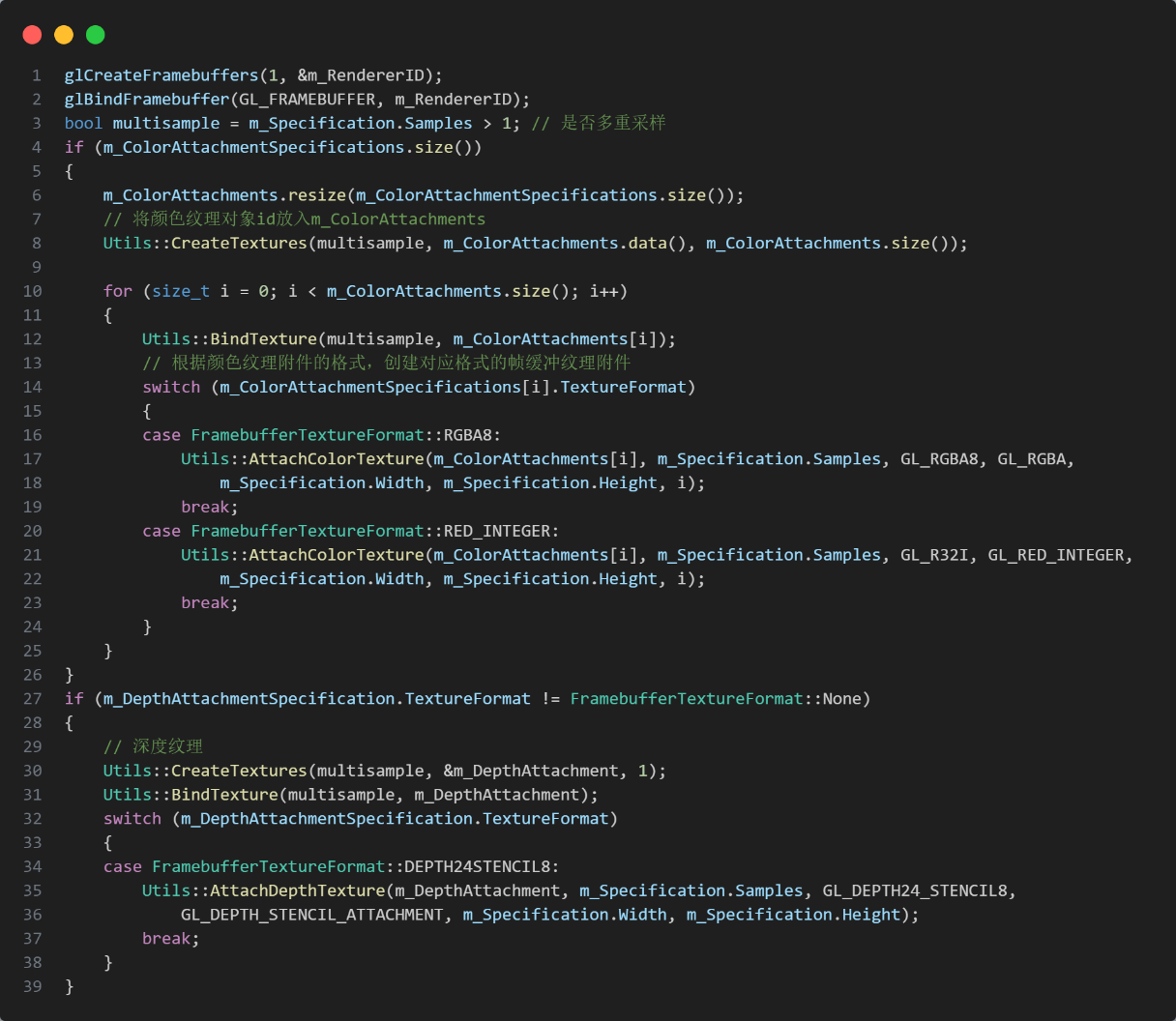
我们封装的OpenGLFramebuffer需要传入一个FramebufferSpecification对象进行初始化，FramebufferSpecification定义如下，其中Samples表示是否多重采样（用于MSAA），FramebufferAttachmentSpecification内部用vector存储了所有的纹理缓冲附件，并支持基于初始化列表的初始化。所以FramebufferSpecification的初始化类似于下面这样，表明FramebufferSpecification内部会管理三个纹理缓冲附件，前两个是颜色相关，最后一个是深度相关。这个对象设置好以后，就可以用它初始化OpenGLFramebuffer对象。



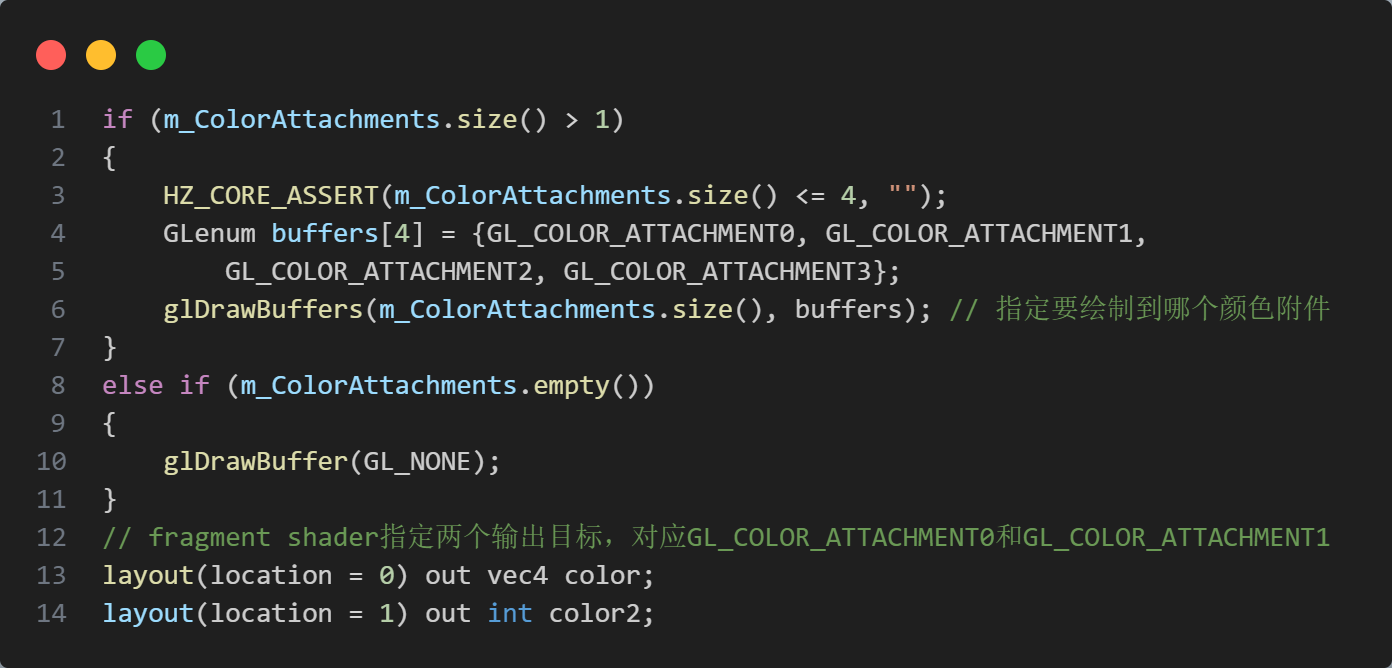
在OpenGLFramebuffer的构造函数中，会将传入的FramebufferSpecification对象进一步解析，将颜色纹理附件放入m\_ColorAttachmentSpecifications，OpenGLFramebuffer中的深度纹理附件只有一个，所以如果FramebufferSpecification对象中含有深度纹理附件，就会直接赋值。在上面的例子中，m\_ColorAttachmentSpecifications中就会含有两个颜色纹理附件，m\_DepthAttachmentSpecification的格式将从None变为Depth，意思是我们新创建的这个帧缓冲是有深度纹理附件的，支持深度测试相关的操作。



根据解析的结果，Invalidate函数中将会正式地创建帧缓冲对象，关键代码如下。注意m\_ColorAttachments与m\_DepthAttachment保存的是缓冲附件的句柄，要与上面的m\_ColorAttachmentSpecifications和m\_DepthAttachmentSpecification区别开来。首先遍历多个颜色纹理附件，创建多个纹理对象，并把句柄保存在m\_ColorAttachments数组中，然后根据每个颜色纹理附件的格式调整纹理的参数，并附加到当前的帧缓冲对象上（AttachColorTexture的功能）。对于深度纹理附件也是类似的，调整纹理参数后附加到当前的帧缓冲对象上。颜色纹理附件的类型为GL\_COLOR\_ATTACHMENT0+index深度纹理附件的的类型为GL\_DEPTH\_STENCIL\_ATTACHMENT。



如果帧缓冲对象附加了多个颜色纹理附件，就需要使用glDrawBuffers，否则只会渲染到GL\_COLOR\_ATTACHMENT0。使用了glDrawBuffers后需要配合片元着色器代码实现多目标的输出。如果没有颜色纹理附件，那就可以将glDrawBuffers参数设为None，只渲染深度纹理附件。



## **渲染器**

Renderer和Renderer2D都是静态类，为测试项目提供封装好了绘制API，是渲染系统中最上层的模块。由于教程的教学转向了2D，以及编辑器系统的引入，Renderer类在最后几乎不作为绘制API的提供者了，它只负责初始化、关闭、响应窗口大小变化、获取当前使用的图形API，绘制API由Renderer2D提供。

### **渲染合批**

QuadVertex代表每个顶点的数据。TexIndex表示应该到哪一个纹理插槽去采样纹理，因为渲染合批后，每一批的物体要使用什么纹理是不确定的，例如第1批物体可能总共使用了某两个纹理，第2批物体可能总共使用了另外3个纹理。因此，片元着色器中需要定义一个uniform变量，是一个长度为32的纹理采样器数组，在渲染开始之前配置好该uniform变量。在这样的条件下，TexIndex不适合作为uniform变量，因为这会导致一个批次的物体都去某个纹理插槽上采样，所以必须每个顶点都有TexIndex数据，片元着色器中就能知道应该去哪里采样。TilingFactor是纹理的缩放因子，在采样时缩放纹理坐标，就可以让纹理重复地铺满整个方形。最后一个EntityID与编辑器功能，之后再详细说明。

Renderer2DData保存了每批次的渲染数据，每批次最多绘制20000个方形，由于方形需要4个顶点和6个索引，所以每批次最大顶点和最大索引数也就确定了下来，每批次最大纹理数为32是因为OpenGL最多支持32个纹理插槽。接下来的VAO，VBO，EBO都会在处理批次数据的过程中配置好。QuadVertexBufferBase和QuadVertexBufferPtr配合使用，每次读取数据的时候指针向后移动，每次处理完一个批次后两个指针之差就是一个批次数据的大小。TextureSlotIndex初始化为1，因为假设物体没有纹理，在最后会生成一个纯白的纹理放在0的位置，这样便于统一着色器代码。



## **渲染系统运行流程**

* 渲染接口初始化。在main函数执行之前，RendererAPI中的静态成员s\_API会初始化为OpenGL，RenderCommand负责向上层模块提供设置ClearColor，清理缓冲区，设置视口，绘制等命令，它内部储存了RendererAPI静态对象，在main函数执行之前就会创建该静态对象，对象内部的s\_API是OpenGL。
* 创建渲染上下文。进入main函数后，开始构造Application单例，在创建窗口的过程中，以当前窗口句柄为参数创建渲染上下文，因为上下文的创建需要提供窗口句柄。即我们封装的窗口对象持有渲染上下文对象，渲染上下文对象持有窗口句柄。在渲染上下文创建的过程中，会使用glad加载OpenGL的函数，接下来渲染系统的其它模块才能正常使用OpenGL函数实现功能。
* 初始化渲染器。

1、初始化RenderCommand，因为在main函数执行之前就已经确定好图形API了，所以RenderCommand的初始化就是使用OpenGL对渲染做一些全局配置，例如开启深度测试，开启颜色混合等。

2、初始化2D渲染器。

（1）缓冲区。申请绘制一个批次的四边形所需的空间，默认一个批次最大有20000个四边形，所以空间的大小为80000个顶点，这会调用OpenGLVertexBuffer的构造函数，在GPU开辟一个80000个顶点大小的VBO空间，但暂不放入数据。开辟缓冲区后，需要指定缓冲区布局，格式为{属性数据类型，属性名}，例如”a\_pos”（位置属性）类型为Float3，”a\_color”（颜色属性）类型为Float4，缓冲区布局对象会根据传入的信息解算出每种属性的大小、偏移量等，用于之后的顶点描述信息的指定。VBO初始化完成后，就初始化EBO，申请一片6\*20000个整数的空间（每个四边形6个索引），将顶点索引数据送入GPU的EBO缓冲区即可。

（2）顶点数组。创建OpenGLVertexArray对象，将刚才处理好的OpenGLVertexBuffer和OpenGLIndexBuffer对象设置到OpenGLVertexArray对象中。重点是OpenGLVertexBuffer的设置，由于刚才已经为OpenGLVertexBuffer对象处理好了VBO内部的布局，所以在这里会调用glVertexAttribPointer相关函数设置顶点描述信息。

（3）顶点数据空间。在CPU申请一片80000个顶点的数据空间，准备每批次处理后送入GPU的VBO缓冲区。当前批次数据的起始位置指针QuadVertexBufferBase就是该空间的起始位置。

（4）纹理。创建一个长度为32的纹理采样器数组（纹理插槽最多32个），因为不知道每批次的物体使用了多少个纹理，不可能对于每批次的物体去指定采样器，所以直接准备好所有的纹理采样器，根据每个顶点的纹理索引数据决定使用哪个纹理采样器进行采样。创建默认的白色纹理，只需要创建1\*1大小的即可，之后通过纹理Wrap直接平铺整个四边形，将白色纹理设置到0号插槽，如果物体没有纹理资源，就在0号插槽采样。

（5）着色器。加载默认的纹理着色器，着色器中有uniform采样器数组变量，所以需要把刚才的纹理采样器数组上传到着色器中。

* 绘制场景到帧缓冲。

（1）创建编辑器层时，就创建了新的帧缓冲，场景将会绘制到这个帧缓冲中的颜色纹理附件上。

（2）每帧都会使用RenderCommand提供的命令清除背景色，清除缓冲，即glClear(GL\_COLOR\_BUFFER\_BIT | GL\_DEPTH\_BUFFER\_BIT)。每帧都会更新相机的视图和投影矩阵，相机相关的设计会在场景和编辑器中说明。

（3）将相机的视图和投影矩阵上传到着色器的uniform变量中，开始一个批次。每个批次开始时，都会重置需要绘制的索引数，需要绘制的纹理索引，批次数据的指针起点。

（4）遍历场景中带有渲染相关组件（SpriteRendererComponent）的物体，对每个物体都调用Renderer2D提供的绘制API，每个物体调用绘制API都会使得批次内的数据增加，批次的相关描述信息也会改变，例如每次绘制都会使该批次需要绘制的索引数加6。

（5）为了更全面地说明，假设要绘制的四边形带有纹理。进入绘制API时，首先判断需要绘制的索引数是否大于批次允许的最大索引数，如果是，就需要使用Flush函数，提交一个批次，具体为，将当前批次的数据送入GPU端的VBO缓冲区中，并激活该批次物体用到的所有纹理单元，调用RenderCommand提供的绘制命令（底层是glDrawElements）渲染一个批次的物体。如果批次索引数未满，则往下执行，考察当前纹理是否已经存在于纹理插槽中，如果存在，则说明之前的物体已经加载过，就无需再次加载，否则需要加载纹理，放入纹理插槽最新的索引中。如果此时纹理插槽已满，即每个索引都记录了纹理，那么就需要提交该批次的数据，再开一个新批次，将刚才加载进来的纹理记录到纹理插槽中。正常情况下是该批次未满的，所以会遍历四边形的每个顶点，将顶点的位置、颜色、uv坐标等属性数据记录到Renderer2DData中，每记录一个顶点，QuadVertexBufferPtr就往后移动，时刻指向缓冲区含有有效数据的部分的结尾。一个四边形的数据处理完毕后，需要绘制的索引数加6。

（6）结束一帧的绘制。遍历完了场景的所有可渲染物体后，在最后一个批次中，场景物体的数据可能不足以占满一个批次的大小，所以会强制调用Flush函数将这个批次的数据提交。

* 渲染到视口。上述步骤只是将场景渲染到了后台帧缓冲的颜色纹理附件中，在UI的更新中，会取出帧缓冲的数据，渲染到视口上。

# **实体组件系统（ECS）**

## **概述**

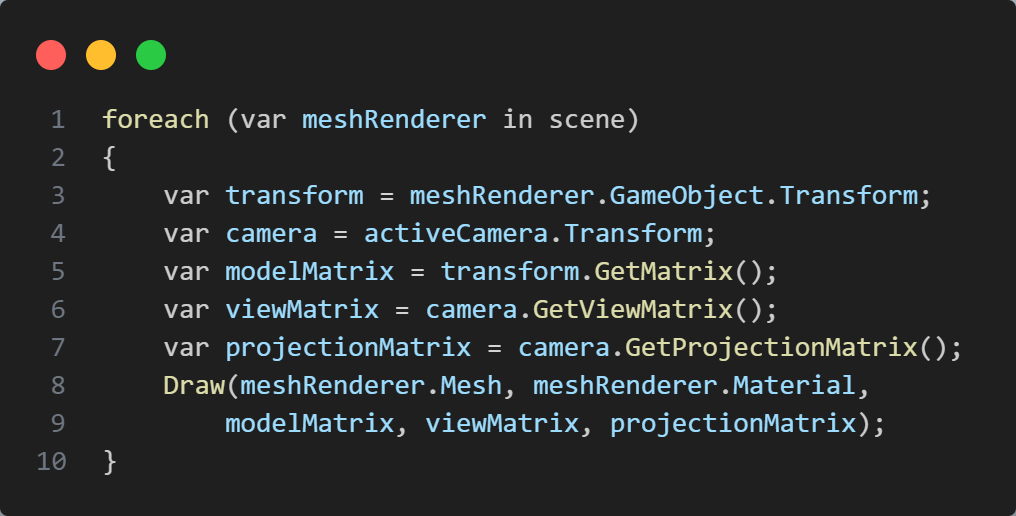
实体，组件，系统之间的关系：

1、实体 (Entity) ：实体是系统中唯一的标识符，通常是一个简单的ID。它本身不包含任何数据或逻辑。 作用：实体作为其他数据（即组件）和逻辑（即系统）的载体，用来标识和操作这些数据或逻辑。

2.、组件 (Component) ：组件是包含数据的结构体或类，但不包含逻辑。每种组件代表一种特定的数据类型，例如位置、速度、健康值等。 作用：组件用于存储实体的状态信息。每个实体可以拥有一个或多个组件，从而描述它的各种属性。

3. 系统 (System) ：系统包含处理特定类型组件数据的逻辑。系统会遍历所有具有所需组件的实体，执行相应的操作。 作用：系统负责更新和处理组件数据，实现游戏逻辑或其他功能。每个系统通常专注于一个特定的任务，如物理模拟、渲染或AI决策等。

Unity的GameObject-Component的设计是非常接近ECS的思想的，例如场景中的一个游戏对象要被渲染出来，就必须拥有Transform组件和MeshRenderer组件，如果在运行模式下，场景中还必须要有一个Camera组件的游戏对象。因此，MeshRenderer、Transform、Camera组件会被Unity的渲染系统处理，渲染系统会通过Transform组件获取模型矩阵，通过Camera组件获取视图和投影矩阵，通过MeshRenderer组件获取渲染所需的各种参数和资源，就将物体渲染了，大致逻辑如下。类似地，刚体和碰撞相关组价也会被Unity的物理系统统一处理。



但是，Unity中还有自定义的脚本组件（MonoBehavior对象），这些组件内除了数据还有逻辑，严格来说这是不符合ECS定义的，Unity 并没有一个明确的“System”层来统一处理MonoBehaviour逻辑，而是通过每个 MonoBehaviour 的生命周期函数来驱动行为。因此，说Unity的GameObject-Component的设计是非常接近ECS的思想的，并不是严格遵守了ECS思想。

本项目的ECS结构就是模仿了Unity，除了脚本组件，其余的组件只含数据，不含逻辑。ECS往往要与场景系统结合，但本章注重于实体和脚本相关代码的实现，场景系统中会详细说明场景系统与ECS是如何配合的。项目的ECS系统使用了entt库。

## **实体（Entity）**

相当于Unity中的GameObject。我们的实体类中含有一个entt::entity句柄，创建实体类对象时，会初始化这个句柄，这个句柄可以视为一个ID，但在项目的场景系统中没有直接使用这个ID，而是使用了UUID和哈希表管理对象。实体类还含有场景指针，表明该实体在哪个场景，对实体进行组件操作时，也要用到场景对象中的entt::registry（管理所有实体（Entity） 和组件（Component） 的容器）。

GetComponent和AddComponent与Unity是类似的，添加和删除组件时会更新场景对象中的entt::registry，在AddComponent中，args是构造组件所需的参数（可以没有）。实体被创建时，默认带有ID组件（唯一标识该物体），Tag组件（实体名），Transform组件（实体的位置旋转缩放），其中ID组件是用户无法操作的。AddOrReplaceComponent在复制实体时使用，因为复制实体时会先创建一个同名（Tag组件）新实体（ID随机初始化），同时也会有一个默认初始化的Transform组件，在复制时需要遍历原实体的所有组件（除了Tag组件和ID组件），所以会用原实体的Transform组件替换新实体的Transform组件，这样复制出的新实体就和原实体有一样的位置旋转缩放。



## **组件（Component）**

定义了一系列能够附加在实体上的组件，添加时用到了entt::registry，传入实体的句柄（entt::entity类型的变量和初始化组件所需的参数即可。下面是每个实体必定拥有的三个组件的定义，其中Transform组件可以返回物体的模型矩阵。



## **原生脚本组件（NativeScriptComponent）及其对应实体（ScriptableEntity）**

注意，这个并不是C#脚本系统，而是使用C++编写的原生脚本。原生脚本在目前的作用是可以快速测试功能，只需写一个类继承ScriptableEntity并实现周期函数的逻辑即可。实际情况中，原生脚本可以由引擎开发人员编写，然后编译成平台特定的动态库（.dll，.so），通过引擎提供的插件相关的接口与引擎集成，实现更加底层的优化或一些平台特定功能。

注意ScriptableEntity并不是实体，而是一个给NativeScriptComponent提供更新能力的类，类似于Unity中的MonoBehavior，所有的自定义逻辑都需要继承MonoBehavior，类似地，所有的原生脚本的自定义逻辑都需要继承ScriptableEntity。

使用方法介绍：写一个类TestScript继承ScriptableEntity，并重写OnCreate，OnDestroy，OnUpdate周期函数。然后让场景中的某个实体添加NativeScriptComponent组件，并使用Bind函数将TestScript与NativeScriptComponent联系起来，Bind之后，NativeScriptComponent中的InstantiateScript和DestroyScript函数指针会初始化完毕。游戏运行时，会借助entt::registry遍历场景中所有含有NativeScriptComponent组件的实体，第一次执行时，NativeScriptComponent内部管理的ScriptableEntity指针为空，此时会调用InstantiateScript指向的可调用对象，实例化一个TestScript对象，并且TestScript中持有的实体m\_Entity也会被赋值为当前正在遍历的实体，然后调用TestScript重写的OnCreate执行组件自定义的初始化逻辑。随后每帧更新时，都会调用TestScript重写的OnUpdate函数执行每帧的自定义逻辑。



## **系统（System）**

总体来说，本项目的系统还没有特别体系化。理想的情况应该是，每帧按照一定的顺序处理游戏对象，例如按照逻辑，物理，渲染的顺序，那么就是先遍历所有拥有脚本组件的实体，调用脚本系统的API对实体状态进行更新；再遍历所有拥有刚体组件的实体，调用物理系统的API，获取碰撞器、Transform等组件对实体状态进行更新；最后遍历所有拥有SpriteRenderer组件的实体，结合Transform组件，调用渲染系统的API渲染物体。目前只有渲染系统比较符合上述过程，脚本系统正在完善中，但是对于学习来说应该足够了。

# **场景系统**

## **场景类**

场景类中最重要的是m\_Registry，这在先前也提到过，是一个entt::registry的注册表，负责管理所有的实体与组件，代表了场景中的所有游戏对象。每次操作实体时，都要通过实体类的场景对象指针，访问m\_Registry，从而操作与实体相关的组件。

m\_EntityMap的作用是建立UUID和实体handle之间的联系，因为UUID是我们自定义的实体唯一标识符，需要与entt中的实体handle建立联系，从而正确操作我们自定义的Entity类的对象（因为Entity类对象中最重要的就是entt的实体handle）。

创建实体的两个函数都会创建时自动附加ID组件、Tag组件、Transform组件。其中CreateEntity主要用于用户在菜单中创建实体，内部会调用CreateEntityWithUUID，随机给一个UUID值；CreateEntityWithUUID主要用于场景反序列化时，即读入场景文件时通过文件记录的实体UUID创建实体。

Runtime相关的三个周期函数都是在游戏运行模式下调用的，OnUpdateEditor是在编辑模式下调用的，例如OnUpdateRuntime会在点击运行按钮后每帧调用，涉及到脚本、物理、场景相机、渲染等所有功能；而OnUpdateEditor是非运行模式下调用，只涉及渲染，不运行脚本，也没有物理模拟。

GetEntityByUUID在之后的脚本系统会提及，通过UUID到entt句柄的映射，获取我们自己封装的Entity对象，然后通过GetComponent获取组件，操作组件，将这些暴露给C#，就可以通过C#代码操作实体了。

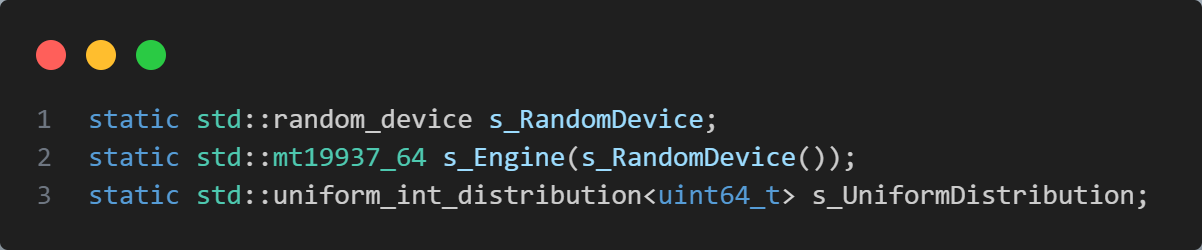


静态Copy函数用于运行模式下拷贝一份场景，如果不拷贝的话，运行之后场景中的物体的状态会发生变化，无法回到运行前的状态，所以运行前拷贝一份场景，运行结束后销毁，编辑器的场景依然保持运行前的状态。静态的Copy函数最终会调用下面的函数，其中(lambda表达式, …)是C++17的折叠表达式解包，…Component是变长模版参数，相当于对每一个Component，都调用一次lambda表达式内的逻辑。例如，假设调用CopyComponent<TransformComponent,CameraComponent>(dstRegistry, srcRegistry, entityMap)，那么就会找到所有含有TransformComponent的源实体，将源实体的Transform组件的数据拷贝到目标实体上，之后对CameraComponent也做同样的操作。

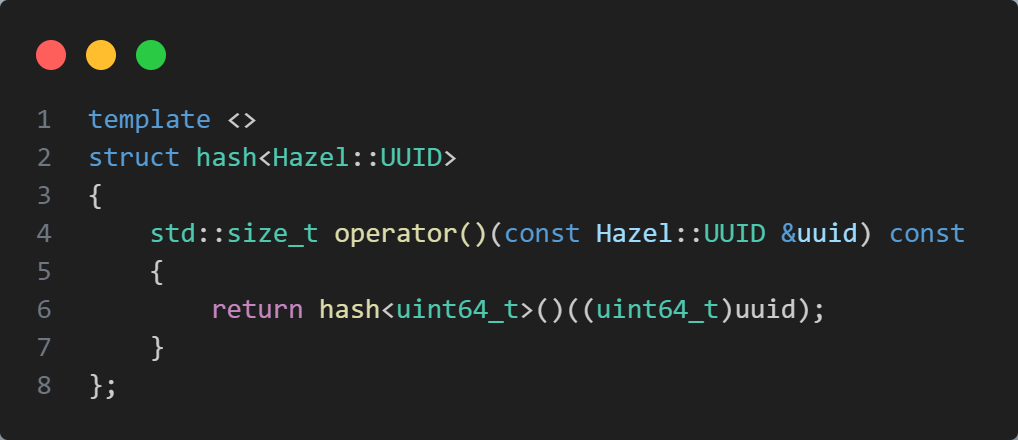


## **全局唯一标识符（UUID）**

s\_RandomDevice用于生成随机种子。通常由系统提供熵源（如硬件、操作系统等），生成一个高质量的种子。s\_Engine使用 std::mt19937\_64（Mersenne Twister 64位），是一个高质量、伪随机数生成器,以 s\_RandomDevice 作为种子初始化。s\_UniformDistribution在UUID的构造函数中会以s\_Engine作为随机数生成器初始化，返回一个均匀分布的 64 位整数（即随机值在 [0, 2^64-1] 之间均匀分布）。虽然并不是绝对唯一，但只要不会短时间内生成数亿个游戏对象，已经够用了。



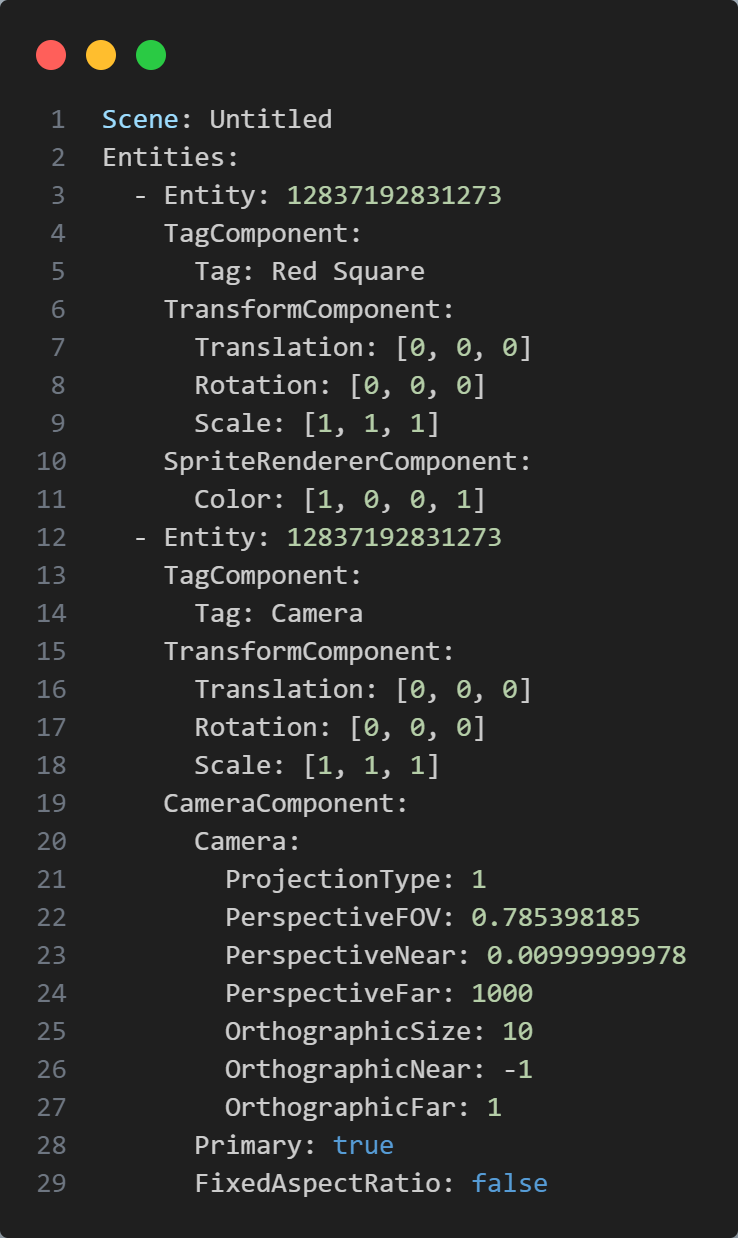
前面说到场景类中有哈希表将我们自定义的UUID映射到entt的句柄，当自定义的类作为哈希表的key时，需要对std::hash做特化处理，实现如下。因为我们自定义的UUID类中本质上就是保存了一个64位整数，所以通过转换函数重载operator uint64\_t() const { return m\_UUID; }，就可以让自定义的UUID类对象转化为64位整数，然后使用标准库中的hash<uint64\_t>来生成哈希值。



## **场景序列化与反序列化**

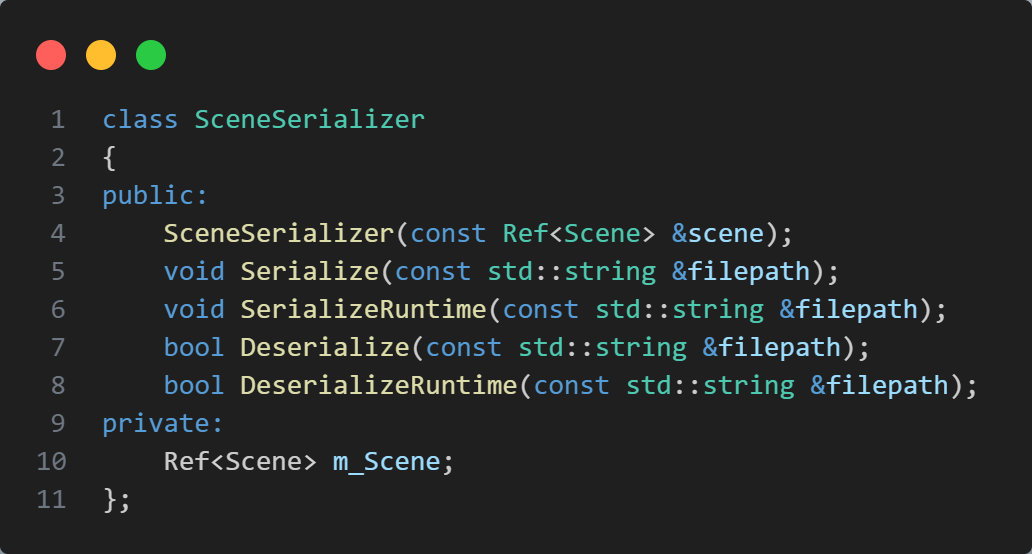
### **概述**

项目将场景保存到yaml文件中，采用yaml-cpp库进行yaml文件的读写。yaml文件由多个节点组成，这个节点可能是映射、序列等，序列化的场景yaml文件内容如下。其中Scene: Untitled是一个节点，表示一组映射，即场景名是Untitled。Entities是一个节点，内容是一组列表，代表场景中的所有实体，而每个列表项中又包含多组映射，例如实体UUID，实体含有的组件等，组件的数据又是多组映射，可见所有的节点都是可以嵌套的。



### **序列化器类**

现在支持编辑模式下的序列化与反序列化，SerializeRuntime和DeserializeRuntime还未实现。序列化器的构造需要传入场景，这样才能访问场景中的entt::registry注册表，在序列化时访问所有实体；在反序列化时，也需要场景类的CreateEntityWithUUID函数，读取了yaml文件以后，将实体创建出来。



序列化的函数如下所示。YAML::Emitter对象可以生成yaml文件，BeginMap代表开始生成映射，第一个映射的key是Scene，value是Untitled，第二个映射的key是Enitities，value是一个序列，遍历场景的所有实体后，使用EndSeq和EndMap结束序列和映射的输入。最后使用std::ofstream将文件写到硬盘上。

SerializeEntity函数用于遍历场景的所有实体，并把实体上所有组件的信息传入yaml对象。但目前采用了非常笨的办法，就是对于所有的组件，在函数中依次进行判断，如果实体含有这个组件，就进行序列化，之后可能会想办法，例如使用模板元编程或者反射简化此部分的代码。



Deserialize函数中，先使用一个YAML::Node对象接受读入的yaml文件，接下来就可以像使用字典一样使用该对象。首先读取场景名，然后获取实体列表，遍历这个实体列表，最重要的是实体的UUID和Tag，获取这两个信息以后，会创建出该实体。其它的组件与序列化类似，都是通过逐个判断的笨方法读取的，之后会考虑改进。

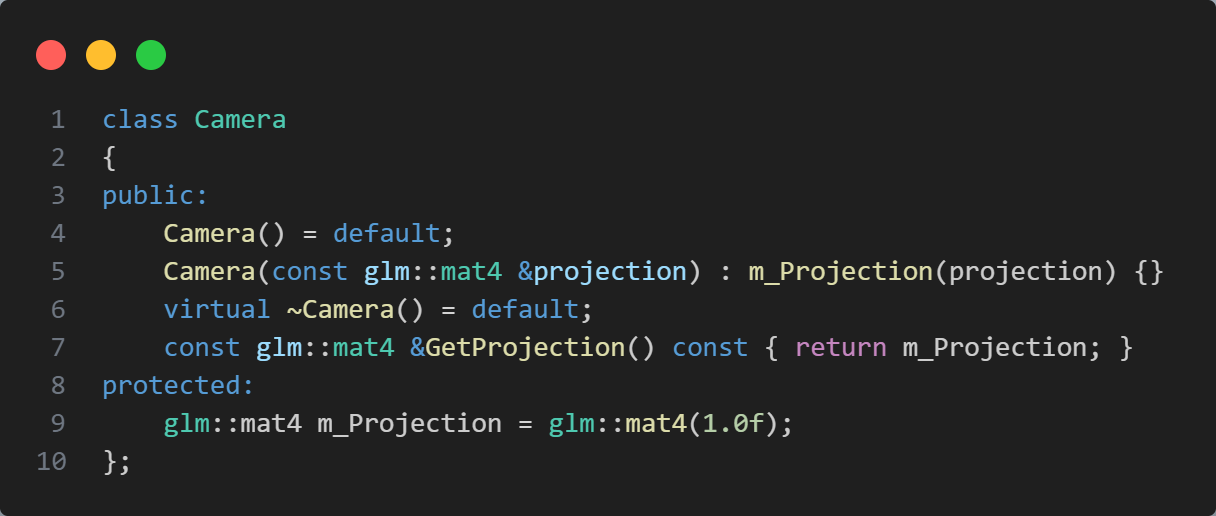


# **相机系统**

相机系统有编辑器相机，场景相机。编辑器相机用于编辑器模式，场景相机用于游戏运行模式。

## **抽象相机**

抽象相机内含有投影矩阵，提供了获取投影矩阵的API。由于场景相机是作为一个实体的组件内的数据存在的，场景相机的视图矩阵实际上是它所在实体的模型矩阵的逆矩阵，所以抽象相机基类中不含视图矩阵。



## **编辑器相机**

编辑器相机是固定焦点，旋转时会按照固定的半径围绕这个焦点，平移时会移动这个焦点。按鼠标中键平移，Alt+鼠标左键旋转，与Unity是类似的。

目前编辑器相机只支持透视投影，横纵比默认为16:9，焦点默认为世界空间原点。当场景的模式为Editor时，它的Update函数在EditorLayer（编辑器项目的测试层）中的OnUpdate函数中调用。同理，当场景的模式为Editor时，OnEvent函数也是在EditorLayer中的OnEvent函数调用。

在编辑器相机的构造函数、OnUpdate、OnMouseScroll函数中，会调用UpdateView函数，更新视图矩阵。在UpdateView函数中，会计算相机目前的位置，获取相机朝向的四元数，用上述两个信息可以获取编辑器相机的模型矩阵，模型矩阵求逆就是视图矩阵。

计算相机朝向四元数的代码如下glm::quat(glm::vec3(-m\_Pitch, -m\_Yaw, 0.0f))。pitch和yaw为负数的原因：视口坐标是从上往下增加，以鼠标向下拖动为例，pitch为正数，假设应用该pitch，相机就会抬头看，但是我们的操作直觉更希望向下拖动时看到物体上方的信息，所以相机应低头看，传入的pitch取负号；以鼠标向右拖动为例，yaw为正数，假设应用该yaw，相机绕y轴旋转，由右手定则可知相机向左转头，但是我们的操作直觉更希望向右拖动时看到物体左方的信息，所以相机应向右转头，转入的yaw取负号。转头方向正确，再配合旋转时相机按照固定半径围绕焦点的策略，就可以全方位地观察一个物体。

拥有相机朝向四元数之后，可以通过glm::rotate(GetOrientation(), glm::vec3(0.0f, 0.0f, -1.0f))获取相机的front方向，其中GetOrientation就是上述的相机朝向的四元数。这句代码可以理解为，相机默认是朝向-z轴平视的，四元数本质上是一个保持长度的旋转变换，所以相当于是把未旋转的相机应用旋转，使它看向目标角度，返回的就是当前相机归一化的front方向。

获取front向量后，用相机焦点减去front方向向量，再乘以相机离焦点的距离，就可以获得相机的位置。拥有相机的位置就拥有了平移变换矩阵，拥有相机朝向四元数后也可以通过glm::toMat4获取旋转变换矩阵，两个矩阵相乘就获得了相机的模型矩阵，模型矩阵求逆就得到了相机视图矩阵，UpdateView函数的功能就完成了。

UpdateProjection函数会在视口大小发生变化时调用，用新的横纵比更新投影矩阵。

在OnUpdate函数中，每帧获取鼠标的位置，并与前一帧的位置作差，获取鼠标的移动。如果鼠标中键按下，则应用移动，改变焦点的位置；如果按下左Alt键的同时又按下鼠标左键，则应用旋转。操作与Unity是类似的。注意旋转函数只根据鼠标的移动改变pitch和yaw，但是相机是沿距离焦点固定距离的球面旋转的，因为每帧调用UpdateView函数都会更新相机位置，相机位置永远在距离焦点固定距离的球面上。

在OnEvent函数中，只捕获鼠标滚轮事件，相机会根据鼠标滚轮的移动，靠近或远离焦点，每次鼠标滚轮事件后都会调用UpdateView函数更新相机的位置。如果相机离焦点的距离小于1，那么相机焦点会前移，并一直保持与相机距离为1。



## **场景相机**

场景相机相对比较简单，因为它是Camera组件的数据，附在某个实体上，所以SceneCamera类中只含数据，几乎不含逻辑。重点是RecalculateProjection函数，它会重新计算投影矩阵，在构造函数、设置视口的函数、设置相机数据的函数中都会调用一次。



## **相机系统和场景系统的工作过程**

当EditorLayer（测试用编辑器层）被推入层栈时，场景和编辑器相机会被初始化。引擎开始每帧更新，此时引擎处于Editor模式，所以会先后调用编辑器相机的OnUpdate与场景的OnUpdateEditor。因为相机会影响场景渲染，所以要先更新相机再更新场景。在编辑器相机的OnUpdate中，会轮询鼠标和按键，如果按下相应按钮，就调用编辑器相机的旋转、平移等方法，OnUpdate的最后会更新编辑器相机的视图矩阵。在场景的OnUpdateEditor中，首先获取编辑器相机的投影矩阵与视图矩阵的乘积，将这个设置为着色器uniform变量的值，开启一个渲染批次。然后遍历场景中所有拥有Transform组件和SpriteRenderer组件的实体，根据Transform组件的数据获取模型矩阵，根据SpriteRenderer组件的数据获取渲染时的一些参数，更新批次的数据。所有实体渲染完毕后提交目前批次的数据，将场景渲染出来，一帧的渲染结束。

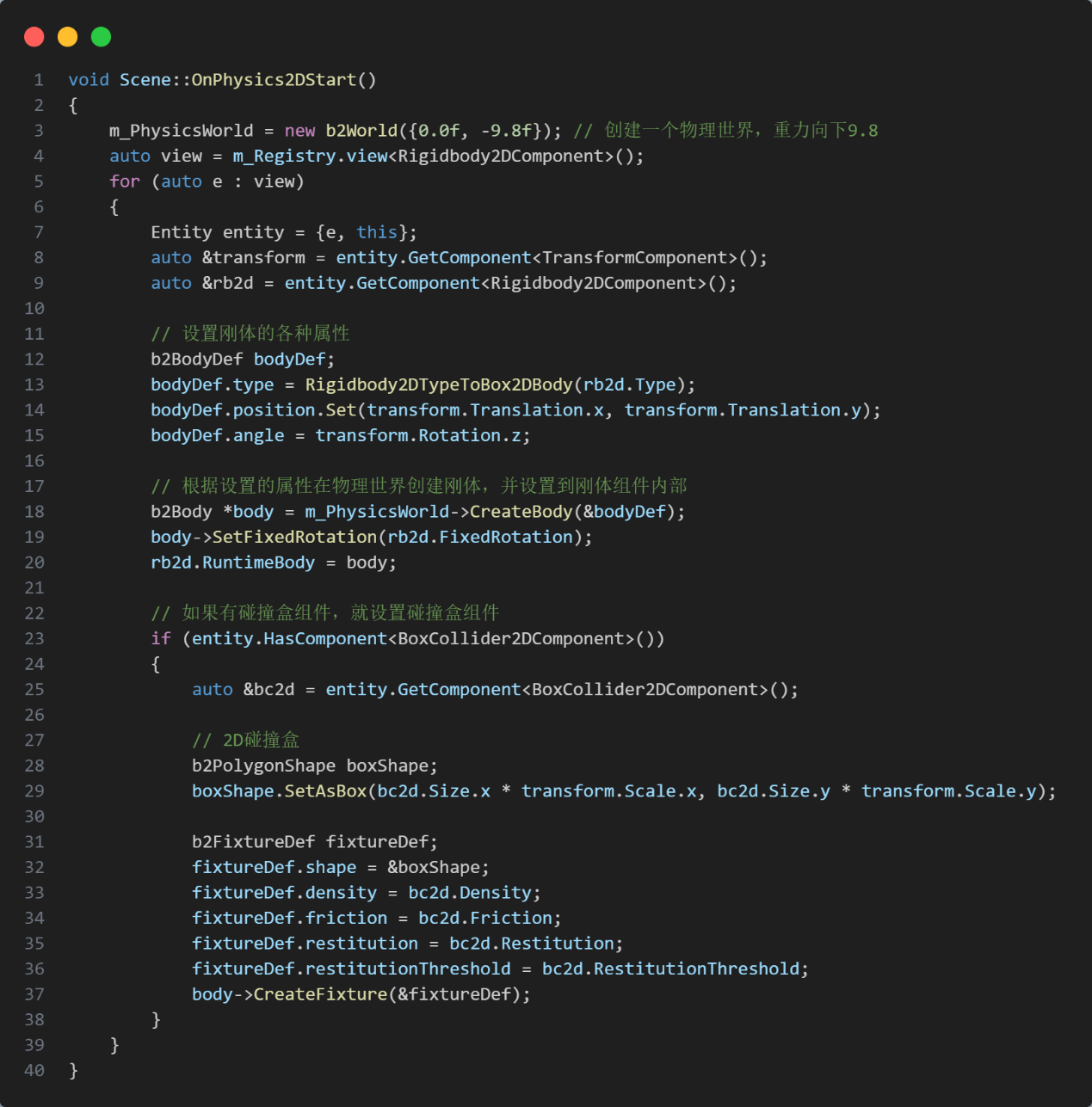
如果用户启动游戏，引擎处于Play模式，此时只调用场景的OnUpdateRuntime函数。在OnUpdateRuntime函数中，会搜集所有含有Transform和Camera组件的实体，遍历这些实体，找到主相机（是否主相机在Camera组件内设置）及其Transform数据。如果主相机存在，那么就会开始渲染场景，视图矩阵为相机组件所在实体的模型矩阵的逆矩阵（通过Transform组件数据计算），投影矩阵保存在场景相机类中，读出即可。将投影矩阵和视图矩阵的乘积设置为着色器uniform变量的值，后面的步骤与编辑器模式下就是类似的了。

# **物理系统**

目前物理模拟相关的功能还没有成系统，初始化和更新都写在了场景类中。目前支持2D物理模拟，使用box2d库，实现了简单的刚体、碰撞体的模拟。

OnPhysics2DStart会在进行Play模式下调用，初始化物理模拟的环境。最开始会初始化一个重力向下9.8的物理世界（box2d的b2World对象），然后搜索场景中所有拥有刚体组件的实体，设置box2d刚体的类型（静态、动态、运动学）、位置、旋转，根据刚才设置的属性实例化一个box2d刚体，并将该box2d刚体对象设置到我们的Rigidbody2D组件内部，即我们的Rigidbody2D组件持有一个box2d刚体对象，在运行时可以直接取出，进行物理更新。

如果拥有刚体组件的实体还拥有碰撞盒组件，则继续设置碰撞盒组件。先根据BoxCollider2D组件中的数据初始化一个2D碰撞盒，然后用该碰撞盒和BoxCollider2D组件中的其他数据初始化一个box2d的碰撞盒对象，最后将这个对象设置到刚体中。各种物理参数的含义暂不做深究。



在Play模式每帧更新时（注意编辑器模式不需要物理模拟），调用b2World对象的Step方法进行一次物理模拟。第2个参数影响碰撞中物体速度计算的精度，迭代次数越多，模拟越稳定但开销越大；第3个参数影响碰撞后物体“穿透”或“卡住”的程度，值越高，越不容易穿模，但开销越大。第1个参数是每次模拟推进的时间，在Unity中默认是0.02s，box2d文档中也建议这个值设定一个固定值，但项目中暂时用帧率代替（即Unity中的Time.deltaTime）。要实现固定时长物理模拟可以设置一个时间累计器，当累计器大于0.02s的时候就调用一次Step，这个之后再改进。

一次物理模拟完成后，需要计算物理相关物体的空间变换状态，所以先获取场景中所有拥有刚体组件的实体。遍历这些实体，拿出Rigidbody2D组件中的box2d刚体对象，将box2d刚体对象中新计算出来的位置，旋转记录到实体的Transform组件中。注意在一帧的更新中，是先进行物理模拟，再进行渲染，所以物理模拟完成后更新的Transform数据就会被正确地渲染出来，用户看到的是实体本次物理模拟完成后的状态。

