GameEngine2

2024年5月25日 16:31

----------ImGui docking & viewport 停靠和视口----------

》》》》我所做的：

选择要使用的版本：
Cherno 使用的应该是1.67或者1.68版本中正在开发的、还未合并至主分支的 docking 中的代码。
做法：
所以我从 ImGui v1.68 版本拉取了 docking 分支，将其作为个人库的一个 docking 分支（之前将 v1.66b 拉取到了个人库中）
然后在 docking 分支中上传对应的 premake5.lua 文件，以备使用。

报错：
对了，在子模块根目录下一开始 git checkout docking 时候，会报错（error: pathspec 'docking' did not match any file(s) known to git）
此时 git branch 查看分支状况时候发现只有一个分支，这表明子模块还未更新
解决方法：
需要运行 git pull 拉取一下，然后可以查看到 docking 分支（我的电脑上是这样的，虽然git pull 之后提示 Already up to date.）

》》注意：
由于我之前并没有定义 GLFW_INCLUDE_NONE,也没有像 Cherno 一样将 glad.h 和 GLFW.h 按照一定顺序包含
所以会导致一些错误，现已修复。 可以参考：（http://t.csdnimg.cn/ogOD3）

》》》》关于库，分支的问题：

》》》一般情况下，一个库的不同分支中的文件一样吗?
答：
一个库的不同分支中的文件可能会有一些差异。
不同分支可能会有针对不同功能需求的变化，比如新添加的特性或优化，这导致文件的更改。
或者说一个分支会是另一个开发路线，进行下一个版本的更新，这都将导致文件的不同。

》》》这两个分支一般会是什么关系，是两个相互没有什么联系的文件区域吗?
答：
分支之间可以相互独立，也可以有一定程度的关联。
一般来讲，一个库的不同分支之间通常是有一定联系的，他们可能代表着同一库的不同版本、不同特性或不同目标的开发路径。

》》》两个分支可以被单独的下载或者使用吗?
答：
分支通常可以单独下载或使用，具体取决于代码管理工具（如Git）的支持和库的发布方式。

》》》不同分支的文件管理的状态?
不同的分支中的文件可以分开单独管理。
可以在不同的分支中对同一个仓库中的文件进行不同的修改，而不会相互影响。

》》》tags 和 branches 的区别
tags:
　　标签通常用于标识特定的版本或提交，一旦创建就不会随着新的提交而改变

branches:
　　分支用于代表不同的代码开发路径，可以持续地进行提交和修改

》》》》git clone 克隆的代码来自哪个分支?
如果没有指定特定的分支或标签，git clone 命令会默认克隆源库的主分支（通常是 master 或 main 分支）。

》》》》关于 ImGui，docking 分支从哪个版本开始正式投入使用?

从 ImGui 版本 1.80 开始，docking 功能被添加到主分支（master branch）中
因此在 1.80 版本及之后的版本，在主分支上便可以找到对 docking 功能的支持。

》》》从 1.80 开始，docking 功能已经被添加到主分支，为什么在主分支之外仍然存在一个 docking 分支呢?
因为开发团队为了保持代码的整洁和稳定性，在主分支之外继续维护一个用于开发和测试新功能的分支。

》》》如何理解被添加到主分支?
首先要知道，不同分支一般存放的代码有什么区别?
主分支：
　　在开源项目中，通常会有一个主要的代码库，其中包含了当前版本的稳定代码以及最新的功能开发。
　　这个主要的代码库就是主分支（master branch）或者叫主线。
其他分支：
　　其他分支一般用于不同的目的，比如开发新功能、修复 bug、实验性质的功能等。

添加到主分支的意思?
一个项目通常包含多个分支，docking 恰恰就是用来开发和测试 docking（停靠）功能的。
当 docking 功能开发完成并被认为稳定时，开发者就将其合并到主分支中，成为主要代码库的一部分。

这就是添加到主分支的意思。

》》》》origin 在 Git 中的意思

origin 是默认的远程仓库名称，通常指向你从中克隆或者拉取代码的远程仓库。
我们一般使用 origin 来表示默认的远程仓库，就而不必每次都指定完整的远程仓库名称。

例如：
git checkout -b docking origin/docking。
从 ImGui 的源仓库克隆到本地，并且将其命名为 origin，同时你在个人的远程仓库也叫 ImGui，那么 origin/docking 表示从名为 origin 的远程仓库中获取 docking 分支的引用。

》》》》子模块切换分支的方法

1. 进入子模块目录，找到想要切换分支的子模块目录，然后进入它的根目录切换到想要的分支：
git checkout branch_name　　//这会将子模块切换到名为 branch_name 的分支。

2. 返回到子模版目录：
cd ..

3.1 提交主项目的变更：
git add path/to/submodule

3.2 git commit -m "Switch submodule to branch_name"
（可选：如果你只是想在本地切换子模块的分支，而不需要将这个更改记录在主项目的提交历史中，那么提交主项目的变更就不是必须的。）

之后如果需要将父仓库推送到远程仓库，使用命令 git push 进行推送

》》》》Git 指令中 fetch 和 pull 的区别
fetch:
　　fetch 命令会从远程仓库下载新的提交和分支，但不会自动合并任何下载的更改到你当前的工作分支上，即不会修改你的工作目录中的文件
pull:
　　pull 命令实际上是执行了 fetch 命令，然后立即将远程分支的更改合并到当前分支中
　　（即 git fetch 和 git merge 命令的组合）

》》》》和 void ImGuiLayer::Begin( ) , void ImGuiLayer::End( ) ,void ImGuiLayer::OnImGuiRender( ) 的关系?
实际上参考 main.cpp 中的例子可以知道，m_ImGuiLayer 的 begin和end 分别对应

```
    // Start the Dear ImGui frame
    ImGui_ImplOpenGL3_NewFrame();
    ImGui_ImplGlfw_NewFrame();
    ImGui::NewFrame();
```

和

```
// Update and Render additional Platform Windows
// (Platform functions may change the current OpenGL context,
// so we save/restore it to make it easier to paste this code elsewhere.
// For this specific demo app we could also call glfwMakeContextCurrent(window) directly)
 if (io.ConfigFlags & ImGuiConfigFlags_ViewportsEnable)
 {
     GLFWwindow* backup_current_context = glfwGetCurrentContext();
     ImGui::UpdatePlatformWindows();
     ImGui::RenderPlatformWindowsDefault();
     glfwMakeContextCurrent(backup_current_context);
 }
```

Begin 是在真正的渲染前所做准备的工作（创建新帧）。
End 是对渲染内容进行的渲染呈现。

所以 OnImGuiRender ( )函数中对应main.cpp，应该实现的是

```
// 1. Show the big demo window
        if (show_demo_window)
           ImGui::ShowDemoWindow(&show_demo_window);

// 2. Show a simple window that we create ourselves. We use a Begin/End pair to create a named window.
    {
        static float f = 0.0f;
        static int counter = 0;

        ImGui::Begin("Hello, world!");                          // Create a window called "Hello, world!" and append into it.

        ImGui::Text("This is some useful text.");               // Display some text (you can use a format strings too)
    ........
    ........
        ImGui::End();
    }

    // 3. Show another simple window.
    if (show_another_window)
    {....省略......}
```

这便是真正要渲染的内容，其中正使用了 ImGui::Begin( ) ,ImGui::End( )
所以void ImGuiLayer::Begin( ) , void ImGuiLayer::End( ) ,void ImGuiLayer::OnImGuiRender( ) 的关系一目了然

》》》》关于 Cherno 所说的 OnUpdate 和 OnImGuiRender 的区别怎么理解

Application.cpp中：

```
        for (Layer* layer : m_LayerStack)
            layer->OnUpdate();                  //执行逻辑更新

        m_ImGuiLayer->Begin();
        for (Layer* layer : m_LayerStack)
            layer->OnImGuiRender();             // 进行渲染操作（执行渲染更新）
        m_ImGuiLayer->End();
```

OnUpdate:

OnUpdate() 函数被用来执行逻辑更新。通常情况下，OnUpdate() 函数用于更新应用程序的逻辑状态。
例如更新对象的位置、处理输入事件、执行物理模拟等等。这个过程在每一帧都会执行，以确保实时修正逻辑操作内容。

在每一帧的渲染过程中，首先进行逻辑更新，然后才进行渲染操作

OnImGuiRender:

得到正确的逻辑状态，进行内容的最新渲染结果。

》》》》Cherno 提出的问题
为什么在 Sandbox 中的运行语句会导致问题？

ImGui 被设置成静态库 .lib
Nut(Hazel) 是 .dll
Sandbox 是 .exe

现在 .dll（Hazel） 接受 .lib（ImGui） 中的函数，然后.exe（Sandbox）能够使用存放在 .dll 中的函数
虽然 .lib 中有所有的函数名称，但是 .dll 是动态加载的，如果一个函数在之后不被使用，.dll就不会执行那个函数。（.dll有能力删除.lib中未使用的内容）
正因为 .exe 使用的函数是 .dll 中没有加载过的（.dll 中没有包含的函数名称），所以在 .exe（Sandbox） 中使用这些函数就会导致程序崩溃。

----------渲染简介和渲染架构----------

讲了一些基础，有点晦涩，介于之前看过 OpenGL 教程，也就马马虎虎看完了。
涉及的要点可以自己搜索，我建议还是将后面几集做完了回头看。

----------渲染和维护----------

》》》》Cherno在程序的属性页进行了修改，虽然我照做了，但并没有成功
于是我在 ImGui 的 premake 文件中做了修改：

```
filter "system:windows"
    systemversion "latest"
    cppdialect "C++17"
    staticruntime "On"

    defines
    {
        "IMGUI_API=__declspec(dllexport)"
    }
```

然后成功实现了。（这个方法是我在youtube某一个视频下方找到的）

----------静态库和无警告----------

》》》》关于使用静态库的好坏，截取了一条评论作为参考。

Static libraries for these kinds of projects are totally fine! The reason is that the user is not gonna run more than 2 -4 applications at the same time, actually, 90% of the users will probably run only one application at a time. Dlls make sense when creating OS -level libraries that are being used by hundreds of processes at the same time, like a windowing API (win32) for example, or Xorg (Linux). Then yes, it would be a waste of memor y, but in the application layer usually, users run one application at a time.

熟肉：
此类项目的静态库完全没问题！ 原因是用户不会同时运行超过 2-4 个应用程序，实际上，90% 的用户可能一次只会运行一个应用程序。 当创建同时被数百个进程使用的操作系统级库时，Dll 很有意义，例如窗口 API (win32) 或 Xorg (Linux)。 是的，这会浪费内存，但在应用程序层中，用户通常一次运行一个应用程序。

----------渲染上下文----------

》》》》什么是句柄
概念： 句柄（Handle）是用于标识资源或对象的抽象概念。
形式： 通常是一个数值或者引用，用来表示系统所管理的资源，例如内存块、文件、图形界面元素等。
作用： 提供对某些资源的访问和操作方式，从而不需要直接访问资源本身。

举例：

文件句柄（C语言实现）：
FILE *fileHandle;
fileHandle = fopen("example.txt", "r"); // fopen() 打开文件将返回一个文件指针，可以视作文件句柄

在程序中打开一个文件时，系统会返回一个文件句柄，用于标识该文件。通过 fileHandle 获取句柄，可以用来对文件进行读取或写入操作。

》》》》为什么要抽象上下文?
为了拓展程序，增加其普适性。将上下文抽象，我们可以采用不同的API来进行上下文的设置。
比如使用OpenGL、Vulcan、DirectX 等。

----------首个三角形----------

》》》》涉及到很多OpenGL中的函数和知识，
可以看OpenGL的参考文档，
当然也可以看cherno的教程。
或者Learn OpenGL官网。

我看完了Cherno 的教程，LearnOpenGL学了一半，所以这里没啥要记的。
我可以把学习Cherno 时记得笔记放在笔记文件夹中，格式会有点乱，可以参考一下。
（为啥格式会这么乱啊，特地去修改了一次，如果格式任然乱也有可能因为缩放导致的，尝试缩放看看）

》》》》后面很多东西其实都是Cherno在他教程里面所教的，一定去看看他的视频。
基础知识和类的抽象什么的，稍有不同。Cherno的思路也很不错，赞。

----------OpenGL着色器----------

》》》》std::make_ptr<> 和 reset 的概念与区别
区别：
std::make_ptr 创建智能指针的全局函数
reset 则是智能指针对象的成员函数

- ***std::make_ptr<>***

概念：
是一个模板函数，用于创建智能指针，并将其初始化为指定类型的对象。
参数：
要创建的对象的构造函数中的参数。
返回值：
返回一个指向新分配的对象的智能指针。

- ***.reset( )***

概念：
智能指针类的成员函数，用于重新分配该智能指针所拥有的资源。
参数：
一个指针（指向新对象的指针）或者为空。
注意：
如果智能指针之前拥有资源，该资源会被释放。
如果不传递参数给 reset，则智能指针将被重置为空。

》》》》想起来一个东西，关于上一节的顶点属性的两个参数 stride（步幅） & offset（偏移量）不同情况下的理解

参数概念：
步幅（stride）指的是相邻顶点数据在数组中的字节间隔
偏移量（offset）指的是每个顶点属性在数组中的起始位置于数组本身开始位置之间的字节偏移量。

1.如果顶点是在结构体中放着的，像这样：

```
struct vertices {
    float position [3] ();
    float color[3] ();
}
```

实际的结构就会是：（位置和颜色分开存储，但每个属性的数据块在其各自区域内紧密排列）

```
{                       //紧密排列指的是空间内存放的都是顶点数据，没有其他东西
    x1, y1, z1,
    x2, y2, z2,
    x3, y3, z3,
    r1, g1, b1,
    r2, g2, b2,
    r3, g3, b3
}
```

stride：是相邻顶点数据在数组中的字节间隔
结果：
position 的 stride 就是 3 * sizeof(float)， color 的 stride 会是 4 * sizeof(float)
1.（能不能像下一个排列那样写成 7 * sizeof(float)： 不能，因为位置和颜色分开储存了）

2.（能不能都写成 0： 可以，因为每个属性的数据块是分开储存的，OpenGL可以不用知道 stride 有多大，直接顺着读取下一个相同属性的分量）
offset： 是指一个顶点属性开始时的位置的偏移量。

结果：

position 的 offset 就会是 0， color 的 offset 就会是9个浮点类型（9 * 4 = 36 byte），offset 会根据顶点属性的不同而变化，
每一个都不同，越向后某一个属性的 offset 就越大（累加）。

2.如果顶点所有属性全都放在一个数组中，像这样：

```
float vertices[] = {
      // 位置                // 颜色
      0.5f, -0.5f, 0.0f,   1.0f, 0.0f, 0.0f,   // 右下
     -0.5f, -0.5f, 0.0f,   0.0f, 1.0f, 0.0f,   // 左下
      0.0f,  0.5f, 0.0f,   0.0f, 0.0f, 1.0f    // 顶部
};
```

实际结构会是：

```
{
      x1, y1, z1, r1, g1, b1,
      x2, y2, z2, r2, g2, b2,
      x3, y3, z3, r3, g3, b3
};
```

stride： 指的是指的是相邻顶点数据在数组中的字节间隔

结果：

这里的一个完整的顶点应该是（x1, y1, z1, r1, g1, b1），position和color的stride 会是6个浮点类型（6 * 4 = 24 byte）
（能不能都写成 0： 不能，这里的顶点不是分开储存的，OpenGL将无法正确地从一个顶点的属性跳到下一个顶点的相同属性）
offset： 是指一个顶点属性开始时的位置的偏移量。

结果：

position 的 offset 就会是 0， color 的 offset 就会是3个浮点类型（3 * 4 = 12 byte），offset 会根据顶点属性的不同而变化，
每一个都不同，越向后某一个属性的 offset 就越大（累加）。

通过相同的 stride，得知一个完整的正确的顶点是什么样的。
通过不同的 offset 我们可以去合适的地方访问到正确的顶点数据。

这个问题曾让我犯难，希望这种解释足够明了。

----------渲染接口抽象----------

》》》》只包含了父类的头文件，是否可以使用子类中的函数

分情况：

1.使用的函数是父类中的虚函数，在子类中重写。
可以使用，因为父类的头文件包含子类虚函数的声明。

2.使用的函数是仅子类中有的。
不可以使用，编译器无法识别和访问子类的其他函数。

》》》》缓冲区抽象之后的初始化方式为什么是这样？
为什么在Application中不使用 Buffer 的子类 OpenGLBuffer 的构造函数来初始化顶点缓冲区对象
m_VertexBuffer.reset(OpenGLVertexBuffer(...));
甚至是不直接 OpenGLVertexBuffer VB(...) 这样来初始化对象呢？

反而是使用了 Create 这个静态函数
m_VertexBuffer.reset(VertexBuffer::Create(vertices, sizeof(vertices))); 来初始化对象。

这个问题在看了视频5/7时候令人不解，但是到后面发现。
Create 函数会根据实际情况（你选择的API）选择一个接口，在对应接口下，Create 会根据实际选择对应的文件，这其中包含该接口规范下编写的构造函数，能过做到对应情况下调用正确的接口函数。

所以这样使用的原因是，需要通过这个函数来选择接口，条件满足时，会自动选择恰当的构造函数。

》》》》glGenBuffers 和 glCreateBuffers 的区别

glGenBuffers：

概念：
glGenBuffers 函数是 OpenGL 旧版本函数，用于一次性生成一个或多个未命名的缓冲区对象的 -> 标识符。
不同：
生成的标识符并不会自动与任何缓冲区对象关联，仅仅是标识缓冲区对象的唯一名称。

glCreateBuffers：

概念：
glCreateBuffers 是 OpenGL 4.5 引入的函数，用于一次性创建一个或多个缓冲区 -> 对象，并返回对应的标识符。
不同：
glCreateBuffers 会自动将标识符和对象相关联，并将生成的缓冲区对象绑定到 GL_ARRAY_BUFFER 目标上（如果创建的是顶点缓冲区）。

总结：

在使用上，glCreateBuffers 更加方便，因为它不仅仅创建了缓冲区对象的标识符，还自动将其绑定到了适当的目标上。
（他简化了代码，减少了 GenBuffers 和 BindBuffer 的组合操作，提高可读和易用性。）
而 glGenBuffers 则需要在生成标识符后，再通过 glBindBuffer 将其绑定到目标上，多了一步操作。

理解：
但是从实际的使用情况上来看（Cherno 在代码中虽然使用了 Create ，但还是因为没有 Bind 而报错），二者应该是可以相互替换的。

》》》》关于类的静态成员变量的初始化，注意！！！
前提：
静态变量需要被初始化。

正确示范：
```
enum class RendererAPI
    {
        none = 0, OpenGL
    };

    class Renderer
    {
        static inline RendererAPI SetAPI(RendererAPI api) { s_API = api; }
        static inline RendererAPI GetAPI() { return s_API; }
    private:
        static RendererAPI s_API;
    };

    RendererAPI Renderer::s_API = RendererAPI::OpenGL;
    // 静态成员变量需要在类外被初始化
```

错误示范1（有警告，很明显错了）
```
enum class RendererAPI
    {
        none = 0, OpenGL
    };

    class Renderer
    {
        static inline RendererAPI SetAPI(RendererAPI api) { s_API = api; }
        static inline RendererAPI GetAPI() { return s_API; }
    private:
        static RendererAPI s_API = RendererAPI::OpenGL;
        // 静态变量不能在声明的时候进行定义
    };
```

错误示范2（无警告，但依旧错）
```
enum class RendererAPI
    {
        none = 0, OpenGL
    };

    class Renderer
    {
        static inline RendererAPI SetAPI(RendererAPI api) { s_API = api; }
        static inline RendererAPI GetAPI() { return s_API; }
    private:
        static RendererAPI s_API;
        Renderer::s_API = RendererAPI::OpenGL;
        // 实际上这是一个静态成员变量的声明和一个成员变量的定义，这实际上是两个不同的变量
    };
```

结论：
静态成员变量的初始化只能在类的外部进行，不能在类内进行。
静态成员变量的初始化只能在类的外部进行，不能在类内进行。
静态成员变量的初始化只能在类的外部进行，不能在类内进行。

》》》》#if 和 #ifdef 的区别
#if 和 #ifdef 的作用类似，但是语义不同。
#if 用来检查条件表达式的真假
#ifdef 用来检查标识符是否已经被定义

----------顶点缓冲区布局----------

》》》》enum class A : uint8_t 其中 " : uint8_t " 是什么意思？
这表示枚举类中每个枚举常量所分配的枚举值在内存中以 uint8_t 的形式存储，所以可以限制枚举值的范围。
枚举值指的就是代表每一个枚举变量/常量的数字。（默认情况下从0到后累加）

什么是 uint8_t ?
C++标准库中定义的无符号 8 位整数类型，其取值范围是 0 到 255。

》》》》什么是初始化列表 Initialized_list()
概念：
是一种用于初始化对象的机制，它允许在对象创建时提供一个包含初始值的列表。
用途：
通常用于初始化数组、容器、类的成员等。

eg:
```
// 使用初始化列表初始化数组
  int arr[] = {1, 2, 3, 4, 5};

// 使用初始化列表初始化 std::vector
  std::vector<int> vec = {6, 7, 8, 9, 10};
```
》》》》在设计完一切后，我发现一个问题，关于 stride 和 offset 的正确性。

记得上面我们说过的两种形式下的顶点结构所对应的不同的 stride 和 offset吗

1.

（位置和颜色分开存储，但每个属性的数据块在其各自区域内紧密排列）

```
{ x1, y1, z1,
 x2, y2, z2,
 x3, y3, z3,
 r1, g1, b1,
 r2, g2, b2,
 r3, g3, b3 }
```

2.

```
{ x1, y1, z1, r1, g1, b1,
  x2, y2, z2, r2, g2, b2,
  x3, y3, z3, r3, g3, b3 }
```

这两种情况下，position 和 color 分别对应 stride 和 offset 需要区别处理。

然而在我们在代码设计的过程中，这个函数计算出来的结果

```
void CalcOffsetAndStride(){
     m_Stride = 0;
     uint32_t offset = 0;
     for (auto& element : m_Elements){
          element.Offset = offset;
          offset += element.Size;
          m_Stride += element.Size;}
}
```

对应的应该是

```
{ x1, y1, z1, r1, g1, b1,
  x2, y2, z2, r2, g2, b2,
  x3, y3, z3, r3, g3, b3 };
```

时，所对应的两个 stride 和 两个 offset.

但是我发现在 Application 中进行声明的时候，BufferLayout layout = {{ShaderDataType::Float3, "a_Position"}, {ShaderDataType::Float4, "a_Color" }};

看起来像是

```
{ x1, y1, z1,
 x2, y2, z2,
 x3, y3, z3,
 r1, g1, b1,
 r2, g2, b2,
 r3, g3, b3 }
```

这种形式呢，于是我思考，为什么Cherno在设计的时候，照常使用并得到正常结果呢。

其实问题在于，我将

BufferLayout layout = {{ShaderDataType::Float3, "a_Position"}, {ShaderDataType::Float4, "a_Color"}}; 完全理解成了一段对顶点结构定义的语句。

实际上这只是一段为了方便计算 stride 和 offset ，也方便阅读代码设计的语句。

查看定义，BufferLayout(std::initializer_list<LayoutElement> elements)，我们填入的初始化表属于LayoutElement规格。

而查看

```
struct LayoutElement {
     std::string Name;
     ShaderDataType Type;
     uint32_t Size;
     uint32_t Offset;
     LayoutElement(ShaderDataType type, const std::string& name)
          :Type(type), Name(name), Size( ShaderDataTypeSize(type) ), Offset(0){ }
};
```

这里也没有任何对于顶点数据，或者顶点结构的定义。

因为我们（Cherno）在设计时并不是通过 BufferLayout layout = {{ShaderDataType::Float3, "a_Position"}, {ShaderDataType::Float4, "a_Color"}}; 来确定顶点的，我们会先行定义顶点（以这种方式）

```
float vertices[3 * 6] = {
  -0.5f, -0.5f, 0.0f, r1, g1, b1,
   0.5f, -0.5f, 0.0f, r2, g2, b2,
   0.0f,  0.5f, 0.0f r3, g3, b3
};
```

所以这样的设计是完全没有问题的。

//事实上，只要我沉住气看个十来分钟，就能看到Cherno使用

//float vertices[3 * 6] = {

// -0.5f, -0.5f, 0.0f, r1, g1, b1,

//   0.5f, -0.5f, 0.0f, r2, g2, b2,

//   0.0f, 0.5f, 0.0f r3, g3, b3

//};

//这样的语句了，我为什么不沉下心来看看呢?

//否则我绝对不会提出这么蠢的问题。 :-|

》》》》关于为什么在 BufferLayout 类型中使用 m_Stride 记录步幅，而不在LayoutElement(BufferElement)类型中和 offset 一样，将 stride 作为结构体的成员，用来记录步幅呢，反而在 Layou tElement(BufferElement) 结构体中，使用 offset 这个成员来记录偏移量。

因为 offset 对于每一个属性的顶点都不一样，需要区别起来，这样在 Calc 函数中进行 offset 的计算得到的不同 offset 会存放在其对应属性的结构体中

但是对于 stride，有一个规律，就是在

```
{ x1, y1, z1, r1, g1, b1,
  x2, y2, z2, r2, g2, b2,
  x3, y3, z3, r3, g3, b3 }
```

情况下，一旦结构是这样，且数据已经确定，每一个顶点属性函数 glVertexAttribPointer() 中 stride 参数这里填入的数据都会是一样的。

对于这个例子，位置 x,y,z 和 颜色 r,g,b 在顶点属性函数的 stride 参数这里填入的都是同样的 stride : sizeof(float) * 6。所以对于这种结构的顶点（仅对于第二种情况的这种排列方式来讲），offset 需要多个，但是 stride 只需要一个，故可以放在成员变量中这样定义。

》》》》m_Elements 是 std::vector<LayoutElement> 类型的，对于函数

BufferLayout(std::initializer_list<LayoutElement> elements)

:m_Elements(elements) 来说，为什么初始化表类型的参数在传入后，可以正确的被单独存放在 std::vector 的不同位置中?

理论上来看的话：初始化列表语法允许在创建对象时使用花括号 {} 来传递 “一组”值，并且编译器会将每个花括号内的值作为单独的初始化列表处理。
当你传递多个初始化列表用于初始化 BufferLayout 类型的对象时：
BufferLayout layout（{element1},{element2},{element3},{element4}）
每个初始化列表都会被视为一个单独的参数，在构造函数中会被分别处理。

这样，每个初始化列表中的元素都会被分别存储到 std::vector<LayoutElement> 中的不同位置，而不会被合并到同一个位置。

》》》》
int func();
int func() const;
const int& func();
const int& func() const; 的区别。

- **1.int func();**

非常量成员函数.
1.可以修改对象的成员变量
2.返回成员变量的拷贝

- **2.int func() const;**

成员函数
1.不会修改对象的成员变量
2.但是它返回的是成员变量的拷贝，而不是引用.(对返回值的修改不会影响到原始对象的成员变量。

- **3.const int& func();**

可以是一个全局函数，静态函数，或者是一个类的成员函数
1.在对应情况下，表示可以修改静态变量、全局变量或成员变量
2.返回一个对静态变量或者全局变量的常量引用

- **4.const int& func() const;**

成员函数
1.不会修改对象的成员变量
2.返回一个对成员变量的常量引用

》》为什么有的是成员函数有的不是，怎么辨别?
通常末尾有 const 关键字表明这个成员函数不会修改对象的成员变量，所以能够明确这种声明方式的函数通常会是一个成员函数。

》》const 在参数位置 void func(const char& str) 是什么意思?
1.函数不会修改这个参数的值
2.传递给函数的参数是一个对 char 的非常量引用
》这和 void func(char str) const 的区别是，前者说明不能修改类外变量，后者说明不能修改的是成员变量。

》》》》问题
1. 在 buffer.h 定义的时候，记得将 class BufferLayout 放在 class VertexBuffer 之前，
因为在 VertexBuffer 中 使用了 BufferLayout 类型声明变量，因为使用的东西需要是之前定义过的，所以要把 BuffeLayout 这个类的定义放在前面

2. 我在实现element.GLType 的时候，我没有想 Cherno 一样将 GetTypeToGLType() 这个函数放在 Application.cpp 中，而是在 Buffer.h中定义了。
这里就需要注意一个问题，由于没有选择在 Application 中将其定义为一个函数，而是在 Buffer.h 中这样定义

```
GLenum GetTypeToGLType() const {
    switch (Type) {
    case ShaderDataType::Float:   return GL_FLOAT;
    case ShaderDataType::Int:          return GL_INT;
    case ShaderDataType::Mat3:    return GL_FLOAT;
    }
    NUT_CORE_ASSERT(false, "Unknown ShaderDataType !");
    return 0;
}
```

为此我特地在 Buffer.h 中添加了 <glad/glad.h> 的声明，正是这一句代码，会导致 Sandbox 产生不能打开 glad/glad.h 文件的报错。
所以我在 premake 文件的 sandbox 项目中的 includedirs 添加了 "%{IncludeDir.Glad}"，这解决了问题。

》》》》注意
在将初始化列表作为参数填入 BufferLayout 的 layout 中时，要确保 vertices 中有对应属性的顶点。
如果只更新布局而不同步更新顶点的话，会导致无渲染结果。

----------VertexArray 顶点数组----------

》》》》发现个问题（多次包含头文件导致的重复定义错误。）

在使用单例类的时候，我直接在头文件中将类中的静态成员变量放在类之外定义了。
这会导致一个问题，如果在使用多个文件，且这些文件中都包含了这个类，这会导致静态成员变量的重定义。（会在链接时报错，fatal error LNK1169: 找到一个或多个多重定义的符号
error LNK2005: "private: static enum Nut::RendererAPI Nut::Renderer::s_API" 已经在 Nut.lib(Buffer.obj) 中定义；）

届时，将此定义置于 对应的 .cpp 文件中即可解决。。（或者使用 inline 关键字来定义静态成员变量。）

----------渲染流和提交----------

这一集就是抽象了代码，使结构明了，不用再显式的调用 gl 函数去渲染物体了

》》》》variable = new classname;
variable = new classname(); 的区别。

一般情况下，
1.variable = new classname;
指隐式的调用默认的构造函数（无参），但是如果未定义默认的构造函数（或者函数不可见），则不会通过编译
2.variable = new classname();
指显式的调用默认的构造函数（无参），如果没有默认的构造函数，编译器则会自动生成一个。

但是：在此处（Cherno 在文件 RendererCommand.cpp 中定义的）classname（OpenGLRendererAPI）是 RendererAPI 的子类。

所以在使用 new classname 的时候，会优先调用父类的构造函数。虽然父类文件中没有定义构造函数，但是该类是单例类，此时编译器会为父类生成一个默认的无参构造函数。
此时variable = new classname;
variable = new classname(); 没有差别。

》》》》Renderer、RendererAPI、OpenGLRendererAPI、RendererCommand 几个文件之间的关系

1.定义函数（RendererAPI、OpenGLRendererAPI）
RendererAPI 定义了几个虚函数，这几个虚函数将在对应的接口文件中被定义实现，这里我们选择定义在 OpenGL 接口中。
在 OpenGLRendererAPI 中，我们定义了几个虚函数的实现，这将完成我们在渲染循环中将要进行的渲染操作。

2.将函数打包在对应接口下（RendererCommand）
通过上面两个文件，我们只定义了操作，但是没有设计根据情况自动调用对应接口的操作类，所以我们定义了 RendererCommand
在 .cpp 中我们能够 RendererAPI* RenderCommand::s_RendererAPI = new OpenGLRendererAPI; 调用对应接口处，
然后在 .h 中设置内联函数，使用 RendererCommand 类中的 s_RendererAPI 来调用对应接口中的函数。
eg.
s_RendererAPI->SetClearColor(color);

3.按需调用函数（Renderer）
最后，仅仅在 Renderer 文件中对 RendererCommand 中打包好的函数进行调用即可，此时要调用的函数已经是对应接口下的（高度抽象的）函数了
之后，我们只需要将想调用的函数放在 Renderer 中，然后通过 Renderer 来访问就好了。

》》》》发现一个问题：为啥在application.cpp中
```
RendererCommand::Clear();
RendererCommand::SetClearColor({ 0.1f, 0.1f, 0.1f, 1 });
Renderer::BeginScene();

m_SquareShader->Bind();
Renderer::Submit(m_SquareVA);

m_Shader->Bind();
Renderer::Submit(m_VertexArray);

Renderer::EndScene();
```
调换
```
    m_SquareShader->Bind();
    Renderer::Submit(m_SquareVA);
```
和
```
    m_Shader->Bind();
    Renderer::Submit(m_VertexArray);
```
的顺序。

一开始是三角形覆盖在方形之上，但为啥转换过来之后却只绘制出来个方形，没看到三角形嘞？？？

对了这是因为渲染顺序的原因，后渲染的物体会覆盖在先渲染的平面图形之上，这是正常的。