GameEngine3

2024年5月25日 17:03

----------相机&原理----------

》》》》查看这两篇说明，一个是坐标系统，一个是摄像机
（https://learnopengl-cn.github.io/01%20Getting%20started/08%20Coordinate%20Systems/）
（https://learnopengl-cn.github.io/01%20Getting%20started/09%20Camera/）
来自 learn OpenGL，很不错的网站，通俗易懂。

完全阐明了这一集的知识点。

----------正交相机----------

》》》》原理：

最开始处于局部空间，在进行坐标变换和摄像机设置的时候，需要进行以下操作。

局部空间 * 模型矩阵 -> 世界空间	模型矩阵（类型：glm::translate）
世界空间 * 观察矩阵 -> 观察空间	模型矩阵（类型：glm::rotate）
观察空间 * 投影矩阵 -> 裁剪空间	模型矩阵（类型：glm::ortho/glm::prespective）
裁剪空间 + 坐标变换 -> 屏幕空间	自定义的操作（类型：glm::translate/glm::scale/glm::rotate）

一般会为坐标乘以 Projection * View * Model。
（比如着色器中会：gl_Position = projection * view * model * vec4(aPos, 1.0f); ）

》》》》inverse 函数在RecalcMatrix() 中的作用
首先，在 Cherno 的代码中，Cherno 将模型矩阵和观察矩阵设置后，放在 transform 中，然后赋值给 m_ViewMatrix.
其实，在代码中 m_ViewMatrix 实际上指代的应该是 m_ModelViewMatrix.

问题：
这时候讲讲为什么要使用 inverse 对 transform 进行转换，然后赋值给 m_ViewMatrix.

因为观察矩阵（View Matrix）通常需要描述观察者相对于世界空间的位置和方向，而不是物体相对于观察者的位置和方向。
也就是说该矩阵是用来作用于摄像机 camera 的，而不是作用于空间中的物体。

如果不进行逆操作，会将变换应用到物体上，那么结果将描述物体相对于 camera 的位置和方向，而不是 camera 相对于世界空间的位置和方向。

》》》》在着色器中，对坐标进行转换时进行乘法的顺序。
gl_Position = projection * view * model * a_Pos;矩阵有先后顺序，而且需要在坐标之前先进行运算。

》》》》glm::value_ptr 的概念与作用
概念：获取 GLM 类型（如矩阵、向量等）内部数据的指针
作用：将 GLM 类型转换为指向C++内部数据的指针，以便将数据作为参数传递给 OpenGL 函数或其他需要 >指针参数<的函数

eg.
```
glm::mat4 matrix = glm::mat4(1.0f);                                    // 创建一个4x4的单位矩阵
glUniformMatrix4fv(location, 1, GL_FALSE, glm::value_ptr(matrix));     // 将矩阵传递给OpenGL
```

----------Moving to sandbox （含摄像机移动）----------

》》》》关于垂直同步（V-Sync）的理解

》》V-Sync 用于解决图像撕裂的问题，什么是图像撕裂？为什么会出现图像撕裂？
一般情况下 GPU 的渲染速度会比屏幕的刷新率快，当 GPU 的渲染速度高于显示器的刷新率时，GPU 在显示器完成一次完整的刷新之前已经渲染了新的图像，而显示器可能还在显示上一帧的部分内容。
这导致在显示图像的某个位置上出现了不连续的线条，即图像撕裂。

》》
理解：
通过垂直同步对 GPU 和显示器的垂直刷新率进行同步，限制GPU输出的帧率，使其与显示器的刷新率保持一致。
（在 OpenGL 中，开启垂直同步后就是调用 glfwSwapInterval(0) 产生这样的效果）
这样，GPU只会在显示器完成一次完整的垂直刷新周期后才输出下一帧图像，从而避免了图像撕裂现象的发生。

缺点：
因为GPU必须等待显示器完成垂直刷新周期后才能输出新的帧，所以不能充分利用显卡性能，可能导致输入延迟和帧率下降。
在 CS::GO 中，我会选择将其关闭 XD

----------TimeStep----------

》》》》关于 Conversion Operators。

》》》》 operator float() { return Variable } 是啥?

概念:

转换运算符/转换操作符（Conversion Operators）是一种特殊的成员函数，

它们允许用户定义自定义类型之间的隐式或显式类型转换(允许你将 Timestep 对象隐式地转换为 float 之类的数据类型).

语法:

operator target_type();

参考视频:

来自 Cherno 的（https://www.youtube.com/watch?v=OK0G4cmeX-I&t=368s）

eg.

```
operator float() const { return m_Time; }      // This is for a varible which typed by "Timestep" can be use with arithmetic operator like + - * /
float GetSeconds() const { return m_Time; }
float GetMilliseconds() const { return m_Time * 1000.0f; }
private:
	float m_Time;
```

----------Transform（变换）----------

》》》》关于 SetPositoin 和 变换 的区别

SetPosition 和 SetRotation 是对摄像机的变换。这一集中我们需要对物体进行变换。

----------Texture（纹理）----------

》》》》Cherno 对于纹理系统设计的一些展望。

没有什么知识点感觉。

关于其中的一些细节Cherno 会慢慢实现的，到时候我再就 PBR 之类的知识点进行学习。

----------着色器抽象和统一变量----------

》》》》参考文献:

来自 Learn opengl：（https://learnopengl-cn.github.io/01%20Getting%20started/05%20Shaders/）

包含此集全部内容。（除了 ImGUI UI 的实现）

》》》》在类的成员函数之后调用 = default 是什么意思?

1.这个语句只可以出现在以下特殊成员函数之后:

默认构造函数
拷贝构造函数
移动构造函数
析构函数
拷贝赋值运算符
移动赋值运算符

2.意为指示编译器对其生成默认的实现，比如默认的构造函数。

》》》》关于 Cherno 将 Submit 的 Shader 类型的参数转换为 OpenGLShader 类型，这是否会报错?

问题概述:

在 C++ 中，你可以

1.将指向 派生类对象的指针/引用 转换为 指向基类对象的指针/引用,

2.将指向 基类对象的指针/引用 转换为 指向派生类对象的指针/引用。

但值得注意的是:

第一种情况称之为：向上转型（upcasting），无需进行显式转换

第二种情况称之为：向下转型（downcasting），需要进行显式类型转换。（正如 Cherno 所做的那样 std::d_p_cast<XX>(XX) ）

需要注意:

1.向上转型:

1.1在转换后如果只访问基类 Shader 中的成员，因为之前派生类对象 OpenGLshader 包含了基类 Shader 对象的所有成员，所以这没有什么差别，很安全。

1.2但是如果你希望转换后还能访问到派生类 OpenGLShader 中特有的成员，就需要用到虚函数这样动态加载的方法。

2.向下转型:

2.1向下转型是有风险的，因为只有当基类 Shader 指针或引用确实指向一个派生类 OpenGLShader 对象时，向下转型才是安全的。

（我们对于 Submit 函数 shader 传入的参数就是 OpenGLShader 类型的，所以安全）

2.2对于原来的参数， Shader shader，我们对其进行向下转换后，对于 OpenGLShader 类型的参数只能调用该派生类中的成员函数。

那么现在关于问题的答案是：不会。

//Cherno 传入了对应的派生类型的参数，而且处理的是向下转型, 使用的是派生类中的成员函数。

》》》》关于 std::dynamic_pointer_cast

1.初始指针：直接使用类型名称声明指针类型 eg: int*
优点是不会使用额外的性能用来维护
缺点是不安全。

2.智能指针：std::dynamic_pointer_cast
优点是提供更高的类型安全性，比如可以自动管理内存、处理异常
缺点是使用额外的性能来维护程序正确性。（其实在 >小项目< 中没多少）

概念：
std::dynamic_pointer_cast 是 C++ 标准库中的一个模板函数，用于在运行时进行动态类型转换。

----------指针(Ref & Scope)----------

》》》》关于智能指针，Cherno有一期视频介绍了。
check that out if you haven't already. （https://www.bilibili.com/video/BV1hv411W7kX/?spm_id_from=333.999.0.0&vd_source=64ca0934a8f5ef66a21e8d0bddd35f63）

》》》》什么是强引用？什么是弱引用？二者有什么作用？
声明：
在 C++ 中，通常没有与其他编程语言中的强引用和弱引用直接对应的概念。但是，可以使用普通指针和智能指针来模拟类似的引用行为。

概念（Java 中）：

- 强引用：
  强引用是一种对对象的正常引用，它会使对象的引用计数加一。
  只要存在强引用指向一个对象，该对象就不会被垃圾回收系统回收，即使内存紧张也不会被释放，

除非所有指向该对象的强引用都被解除（即没有任何对象持有对该对象的强引用），对象才会被销毁和释放内存。

- 弱引用
  弱引用是一种非强制性的引用，它不会增加对象的引用计数。
  弱引用允许对象被垃圾回收系统回收，即使还有弱引用指向该对象，只要没有强引用指向它，对象就可能被回收。

对应用例（C++）：

- 强引用：

在 C++ 中，使用普通指针或者智能指针来引用对象时，通常就会是一种强引用。
1.普通指针不具备自动管理内存的能力，需要手动释放资源
2.智能指针（如 std::shared_ptr 内含计数系统）同时也可以自动管理内存生命周期，当没有任何智能指针指向对象时，会自动释放资源。

- 弱引用：

在 C++ 中，可以使用 std::weak_ptr 来模拟弱引用的行为。
std::weak_ptr 是 std::shared_ptr 的伴随类，它允许观察 std::shared_ptr 指向的对象，但不会增加对象的引用计数。
因此，当所有 std::shared_ptr 指向对象的引用都释放时，即使存在 std::weak_ptr，对象也会被释放。

参考Java中的概念理解。

作用：
1.强引用可以确保对象在需要时不会被提前释放。
2.弱引用可以防止对象复用，同时避免内存泄漏。

》》》》关于unique_ptr 的理解。

unique_ptr 可以看做是强引用。

相较于普通指针：
std::unique_ptr 是一种独占所有权的智能指针，与普通的指针不同，它不允许多个指针同时指向同一块内存区域。
相较于共享指针 shared_ptr：
std::unique_ptr 不会增加引用计数，而是在其生命周期结束时自动释放所管理的对象。更加轻量。

》》》》什么是“原子”？
概念：
“原子”（atomic）是一个术语，用来描述不可分割、不被中断的操作或动作。
当我们说某个操作是原子的，意思是这个操作在执行过程中不会被其他操作打断，无论是因为多线程竞争还是其他原因。

常见的原子操作：
原子读写：确保读取或写入数据时不会受到其他线程的干扰。
原子增量/减量：确保对数值进行加减操作时是不可分割的。
原子比较和交换（CAS）：一种常用的原子操作，用于在多线程环境中实现锁定机制，确保在执行特定条件下的数据交换是原子的。

作用：
原子操作在并发和多线程环境中起关键作用，因为它确保了数据的一致性和完整性。在这些环境中，多个线程可能会同时尝试修改同一个数据。这时，如果没有原子操作，可能会发生竞态条件（race condition），导致数据错误或不可预测的行为。

使用：
C++：可以使用 <atomic> 头文件提供的 std::atomic 类型来确保原子性。

》》》》什么是“原子增量”？ 什么是“原子减量”？

原子增量（Atomic Increment）：
原子增量是指对共享变量进行加一操作，并且这个操作是原子性的，即不会被中断或者被其他线程干扰。
（在多线程环境中，当多个线程同时尝试对同一个共享变量进行增量操作时，如果不使用原子操作，可能会出现竞态条件（race condition）导致结果不确定或者出错。）
原子增量操作保证了在任何时刻只有一个线程能够对变量进行递增操作，从而确保了结果的正确性。

原子减量（Atomic Decrement）：
原子减量与原子增量相反，是对共享变量进行减一操作，并且保证操作的原子性。
类似于原子增量，原子减量操作也是为了避免竞态条件而设计的，确保在多线程环境下对共享变量进行减少操作时的正确性。

----------Texture（纹理）----------

》》》》部分知识参考（https://learnopengl-cn.github.io/01%20Getting%20started/06%20Textures/）
也可以去看Cherno OpenGL系列的视频。

》》》》glTextureStorage2D和 glTextureSubImage2D分别是什么用法，这两个函数有什么作用？这和glTexImage2D有什么不同？

- glTexImage2D：
  glTexImage2D 用于指定一个二维纹理图像的数据。
  这个函数会为纹理对象分配内存并设置其格式、宽度、高度以及初始数据。

当调用 glTexImage2D 时，如果纹理对象已经存在，则会重新分配内存并覆盖原有数据。

- glTextureStorage2D：
  glTextureStorage2D 用于分配存储纹理对象的内存空间，但不会填充具体的纹理数据。
  这个函数通常在创建纹理对象时使用，用于指定纹理的维度、级别数量和内部格式，但不会填充实际的像素数据。

与 glTexImage2D 不同，glTextureStorage2D 不会重新分配内存或更改纹理对象的大小，只会分配足够的内存空间。

- glTextureSubImage2D：
  glTextureSubImage2D 用于更新已分配内存的纹理对象的部分或全部像素数据。
  这个函数可以用于更新纹理对象的某个区域，而不影响其他区域。

它通常用于动态更新纹理数据，比如视频纹理、实时渲染等场景。

》》也就是说glTexImage2D就是glTextureStorage2D和 glTextureSubImage2D组合使用的效果吗？
差不多是的。因为：
1.glTexImage2D 提供了一个简单的接口，用于分配内存并设置初始纹理数据。它相当于一次性完成了内存分配和初始化数据的工作。

2.glTextureStorage2D 专门用于分配内存空间，但不填充具体数据。它通常用于创建纹理对象并指定其格式、大小和级别数量。
glTextureSubImage2D 则用于在已分配内存的纹理对象上更新数据。它允许你在不重新分配内存的情况下，动态更新纹理的像素数据。

所以可以通过组合 glTextureStorage2D 和 glTextureSubImage2D 来实现类似 glTexImage2D 的功能

》》》》glActiveTexture(GL_TEXTURE0); + glBindTexture(GL_TEXTURE_2D, texture1);
和 glBindTextureUint的区别。

- glActiveTexture(GL_TEXTURE0);
- glBindTexture(GL_TEXTURE_2D, texture1);

这是传统的绑定纹理的方式，适用于 OpenGL 2.0 及更高版本，
首先，使用 glActiveTexture() 设置活动的纹理单元，然后通过 glBindTexture() 将特定的纹理对象绑定到该纹理单元。
适合用于绑定和切换不同的纹理单元，以便在着色器中使用。

- glBindTextureUnit(unit, texture);

这个函数在 OpenGL 4.5 及更高版本中可用。
这个方法直接将纹理对象绑定到特定的纹理单元，而无需先激活纹理单元。

区别：
glBindTextureUnit() 更直接、更高效，因为它不需要显式调用 glActiveTexture()。有助于减少代码复杂性并提高性能。
因为可以直接操作纹理单元，而无需通过传统的激活-绑定流程。在 4.5 以上版本，可以用来替代 Active + Bind;

》》》》stb 库的支持
QUICK NOTES:
Primarily of interest to game developers and other people who can
avoid problematic images and only need the trivial interface

JPEG baseline & progressive (12 bpc/arithmetic not supported, same as stock IJG lib)
PNG 1/2/4/8/16-bit-per-channel

TGA (not sure what subset, if a subset)
BMP non-1bpp, non-RLE
PSD (composited view only, no extra channels, 8/16 bit-per-channel)

GIF (*comp always reports as 4-channel)
HDR (radiance rgbE format)
PIC (Softimage PIC)
PNM (PPM and PGM binary only)

Animated GIF still needs a proper API, but here's one way to do it:
http://gist.github.com/urraka/685d9a6340b26b830d49

- decode from memory or through FILE (define STBI_NO_STDIO to remove code)
- decode from arbitrary I/O callbacks
- SIMD acceleration on x86/x64 (SSE2) and ARM (NEON)

Full documentation under "DOCUMENTATION" below.

熟肉：快速注释：
主要是游戏开发者和其他有能力的人感兴趣
避免有问题的图像，只需要简单的界面

JPEG 基线和渐进（不支持 12 bpc/算术，与库存 IJG 库相同）
PNG 每通道 1/2/4/8/16 位

TGA（不确定是哪个子集，如果是子集）
BMP 非 1bpp、非 RLE
PSD（仅合成视图，无额外通道，每通道 8/16 位）

GIF（*comp 始终报告为 4 通道）
HDR（亮度rgbE格式）
PIC（Softimage PIC）
PNM（仅限 PPM 和 PGM 二进制）

动画 GIF 仍然需要适当的 API，但这是一种方法：
http://gist.github.com/urraka/685d9a6340b26b830d49

- 从内存或通过 FILE 解码（定义 STBI_NO_STDIO 以删除代码）
- 从任意 I/O 回调中解码
- x86/x64 (SSE2) 和 ARM (NEON) 上的 SIMD 加速

完整文档位于下面的“文档”下。

----------blending（混合）----------

》》》》函数使用自行查阅文档或资料，Cherno 系列视频或 Learn OpenGL文档。

》》》》关于为什么使用多层抽象来实现 Renderer:: 的方法?
为了高度抽象，以供多接口使用。
》》RenderCommand 的作用?
1.可以直接静态的调用一些命令，而不是实现某接口的方法。
2.方便多线程的使用。

----------Shader Asset Abstract（着色器资产抽象）----------

》》》》 bat语法参考（浏览器打开）
https://winddoing.github.io/post/29269ff3.html?highlight=bat

》》》》@echo off 和 IF %ERRORLEVEL% NEQ 0 (PAUSE) 在批处理文件中的意思?

@echo off
关闭批处理文件的命令回显功能。
命令回显是指在批处理文件执行时显示每个命令在命令提示符窗口中的执行结果。
作用：这使输出简洁，执行高效。

```
IF %ERRORLEVEL% NEQ 0 (
  PAUSE
)
```

1. %ERRORLEVEL% 代表上一个命令的退出代码（或错误代码）
Windows上，程序和命令执行完成后会返回一个退出代码。返回代码0通常表示成功，非零表示错误或异常。
2. IF %ERRORLEVEL% NEQ 0
如果 %ERRORLEVEL% 的值不等于0（即先前的命令发生了错误）
3. PAUSE
暂停批处理文件的执行，直到用户按下任意键。

作用：如果执行某个命令后发生错误，批处理文件就会暂停并等待用户响应。否则直接运行完毕。

》》》》在 gitignore 中标明 bin 和 bin/ 有什么区别?

bin：表示会忽略所有包含 bin 的文件或目录
bin/：表示会忽略名为 bin 的文件目录（文件夹）

》》》》https://github.com/TheCherno/Hazel/pull/107
》》》》在这个 pull request 中，两人讨论一个问题，即 OpenGLBuffer 文件中，OpenGLIndexBuffer

OpenGLIndexBuffer 函数定义里面（注意是：IndexBuffer 中）
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, m_RendererID);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, count * sizeof(uint32_t), indices, GL_STATIC_DRAW);
是否应该被写为：
glBindBuffer(GL_ARRAY_BUFFER, m_RendererID);
glBufferData(GL_ARRAY_BUFFER, count * sizeof(uint32_t), indices, GL_STATIC_DRAW);

@LovelySanta 说，在 4.5 中，vertex buffer 和 index buffer 只在 Bind( )
void OpenGLIndexBuffer::Bind() const {
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, m_RendererID); }
这里有区别，需要分别写做：GL_ARRAY_BUFFER 和 GL_ELEMENT_ARRAY_BUFFER,
在 OpenGLIndexBuffer 和 OpenGLVertexBuffer 中，则都在 glBufferBind 和 Data 中填入 GL_ARRAY_BUFFER.

文献附上：
4.5（翻阅第60页）https://registry.khronos.org/OpenGL/specs/gl/glspec45.core.pdf
4.6（翻阅第60页）https://registry.khronos.org/OpenGL/specs/gl/glspec46.core.pdf

结论：（绑定缓冲区时指明即可，创建缓冲区时不用区别开来）
在OpenGL 4.5核心配置中，只需将缓冲区绑定到GL_ARRAY_BUFFER，而不需要在创建索引缓冲区时绑定到GL_ELEMENT_ARRAY_BUFFER。
唯一需要将 indexbuffer 的 slot 设置为 GL_ELEMENT_ARRAY_BUFFER 的情况是在绑定时，以指定缓冲区包含索引。

》》》》premake 中的 flags：
flags:
用于设置编译器选项或标志的一个属性. "MultiProcessorCompile"
是一个 MSVC 编译器的选项，用于启用多处理器编译。

作用：
当这个标志被设置时，编译器会尝试使用计算机上的多个处理器来并行编译源文件，以加快编译速度。

》》》》读取文件的函数设计理解：

》》关于 ifstream 类对象的理解：

对象的概念：
std::ifstream 对象 (比如 in) 是一个文件流，它代表了一个与文件相关联的输入流。
对象持有：
该对象持有一系列操作文件的方法和属性，你可以通过这些方法和属性来读取文件内容、控制文件读取位置、判断文件是否打开等等。
一般来讲，in 本身不包含文件中的数据，他只是持有一些方法，对文件操作，在某些情况下（比如将文件读取出来）作用于你自己声明的 string 类型变量上。

》》函数设计：
1.std::ifstream in(filepath, std::ios::in, std::ios::binary); --> 打开文件以备后续读取
从 filepath 中访问文件

1.1 以输入（读取）模式打开文件（std::ios::in），
1.2 并以二进制模式打开文件（std::ios::binary），后续将文件作为原始二进制数据读取。

2.if (in)
如果成功读取，in 会被隐式转换为 true，if (in) 表示文件若打开 ...

3.in.seekg(0, std::ios::end);
读取文件之前设置文件读取位置，通过 seekg 让这行代码将文件读取位置设置为：读取位置为末尾，且不设置方向。

3.1 这里的参数 0 表示相对于某个参考点的 offset 偏移量，（用来确定相对于参考点的读取位置）
(正数表示向文件末尾方向移动多少个字节，负数表示向文件开头方向移动多少个字节，零表示不移动，origin设置的参考点位置已经满足我们的用例)
3.2 std::ios::end 则就是偏移量的参考点，表示相对于文件末尾的位置。（用来确定读取的位置，offset 设置在参考点之前或之后多少个字节的位置）
（可以是 std::ios::beg（文件开头）、std::ios::cur（当前读取位置）、std::ios::end（文件末尾）中的一个，
由于只有三个预定义的参数而没有其他参数可填，所以我们在必要时需要使用 offset 来确定一个比较准确的位置）。
eg.
// seekg(10, std::ios::beg) //将读取位置移动到文件起始位置后第10个字节处。
// seekg(-5, std::ios::end) //读取位置移动到文件末尾之前的5个字节处。

所以seekg 操作实际上是在移动文件流的读取位置指针，我们显式的设置这个指针的位置。

4.result.resize(in.tellg());
这行代码将字符串 result 的大小进行调整，以便容纳整个文件。
4.1 tellg 函数用于获取当前的读取位置，返回一个整数值。此时获取了文件末尾位置，0表示不用前后移动，就返回当前位置，
所以我们返回了这个末尾位置来为 result 确定大小。

5.in.seekg(0, std::ios::beg);
这行代码将文件读取位置重新设置到文件的开头，以便从文件开头开始读取数据。
5.1 目的：设置读取位置为开始，以便接下来的 read 函数作用时从正确的位置开始读取文件。

6.in.read(&result[0], result.size());
开始读取数据。
&result[0] 是 result 的起始指针，表示将文件读到哪个缓冲区中去。

result.size() 是要读多大的文件（读多少字节数的文件），result.size() 恰好是一整个文件的大小。

7.in.close();
关闭文件流。

》》我们通过 seekg 确定了文件目前的指针位置，也就是读取时开始的位置，那读取方向由谁确定?
问题分析：
seekg 函数用于确定读取位置，而具体的读取方向则由后续的读取操作来确定。
设置方式：
读取方向是由后续的读取操作（比如 getline、read 等），通过这些函数我们设置读取方向。

》》》》关于 unordered_map 的一些使用方式。（无序映射）
使用前需要包含头文件 <unordered_map>

1.初始化

```
std::unordered_map<KeyType, ValueType> Map
或者使用初始化列表
std::unordered_map<KeyType, ValueType> Map = {
    {key1, value1},
    {key2, value2}
};
```

2.插入元素

```
Map[key] = value;  或者使用 insert 函数   Map.insert({key, value});
```

3.访问元素

```
Map[key]  也可以使用一些成员函数，进行判断：
    Count（对应某键的值的数量）
    Map.count(key)

Find（查找元素）
auto it = Map.find(key)
```

find: 返回一个迭代器，指向查找到的元素，如果未找到，则返回指向容器末尾的迭代器，即 unordered_map::end()。

4.删除元素

```
Map.erase(key);
```

5.可以用于遍历

```
for (const auto& pair : Map) { }
```

或者使用迭代器遍历

```
for (auto it = Map.begin(); it != Map.end(); ++it) { }
```

》》》》在 Windows 系统中，为什么使用 " \r\n " 来代表新行（另起一行）
\r，\n：代表回车符和换行符。
在文本操控中，\r 一般用于控制光标在文本输出中返回到当前行的起始位置，而不换行。
故在 Windows 操作系统中，使用 "\r\n" 表示文本中的一行结束，并开始新的一行。

》》》》关于 find_first_of 和 find_first_not_of 函数
1. find_first_of
原型：

```
size_t find_first_of(const string& str, size_t pos = 0) const noexcept;
```

参数：
str 参数是要搜索的字符集合，pos 参数是搜索的起始位置，默认为 0。
返回值：
如果找到匹配的字符，则返回该字符在字符串中的位置索引；如果没有找到匹配的字符，则返回 string::npos。
释义：
在字符串中查找第一个与指定字符相匹配的字符，并返回其位置。

2.find_first_not_of
原型：

```
size_t find_first_not_of(const string& str, size_t pos = 0) const noexcept;
```

参数：
str 参数是要搜索的字符集合，pos 参数是搜索的起始位置，默认为 0。
返回值：
如果找到第一个不匹配的字符，则返回其在字符串中的位置索引；如果没有找到不匹配的字符，则返回 string::npos。
释义：
在字符串中查找第一个与指定字符不匹配的字符，并返回其位置。

eg.
"a bc \n abc aabcds \r\n"
find_first_of ("\r\n"); 返回 \r\n 在文本中的索引。
↓
"a bc \n abc aabcds \r\n"

"adb ccc\n bacccab\r\n"
find_first_not_of ("\r\n"); 返回此句中第一个不是 \r\n 的字符在文本中的索引（位置）

↓
"adb ccc\n bacccab\r\n"

》》》》关于 PreProcess 函数的理解：

1.提前处理：

```
const char* typeToken = "#type";                //确定类型助记符的格式
size_t typeTokenLength = strlen(typeToken);     //确定助记符的长度
size_t pos = source.find(typeToken, 0);         //从头开始查找助记符的位置并返回
while (pos != std::string::npos)    {}          //如果找到了助记符（开始时逐行都进行判断，如果找到则进入循环。但由于循环中会对 pos 进行刷新，可以在
                                    //退出当前循环之后，通过 pos = source.find(typeToken, nextLinePos); 跳过多余的语句快速进行判断）
```

2.循环中的设计：

```
size_t eol = source.find_first_of("\r\n", pos);             //使用 find_first_of 得到助记符这一行的末尾位置
HZ_CORE_ASSERT(..., "Syntax error");                        //断言
size_t begin = pos + typeTokenLength + 1;                   //计算助记符 #type 之后的字符应该出现的位置，并储存
std::string type = source.substr(begin, eol - begin);       //复制 #type 之后的类型标识 “vertex/fragment.pixel”
HZ_CORE_ASSERT(..., "Invalid shader type specified");       //断言

size_t nextLinePos = source.find_first_not_of("\r\n", eol); //使用 find_first_not_of 得到助记符下一行的字符串开头的位置并储存
pos = source.find(typeToken, nextLinePos);                  //将 pos 更新至从新行开始后遇到的下一个助记符 #type 的位置
shaderSources[...] = source.substr(nextLinePos, pos - (nextLinePos == std::string::npos ? source.size() - 1 : nextLinePos));

//如果找不到新行，即该类型着色器没有放置代码，且位于文件末尾。则直接从助记符的开始复制直到文件末尾。（下一个提问会分析这一设计）
//如果找到新行即该着色器是有实际代码的，则直接从实际代码的开始复制到下一个助记符之前。
```

3.为什么会在复制一个着色器代码时使用 source.size 这样一个完整的大小？万一后面还有着色器呢？

前提：

实际上，我们使用 size_t nextLinePos = source.find_first_not_of("\r\n", eol);来判断一个助记符 #type vertex 这一行之后有没有字符，然后将其复制。
无论是有字符还是空行，我们都返回位置。（有新行，且新行上有实际代码，返回位置。有新行，但新行没有实际代码，也返回位置。）
至于编译结果，我们交给编译器去判断，空与不空导致的结果会是某种情况下所需要的。

分析：

但是注意看 substr 的第二个参数 pos - (nextLinePos == std::string::npos ... ， 这里的 std::string::npos 指的是完全没有任何东西，包括 '\n' '\r'等。
也就是说，如果满足此条件，则表明 #type fragment 后面已经是文件末尾，后面没有任何东西了。

```
eg.
#type vertex
vertex shader code

#type fragment
```

注意，像是：

```
eg.
#type vertex
vertex shader code

#type fragment \n
```

提示：

这样不满足 nextLinePos == std::string::npos 条件，因为其之后虽然没有实际的代码，但在 find 函数的眼中，有 '\n', '\r' 的存在，会将 pos - (nextLinePos == std::string::npos ? source.size() - 1 : nextLinePos) 判别为 pos - nextLinePos 。
（尽管这样没有什么问题，有时候着色器在调试过程中确乎不会存放代码。

总结：

但注意这个 nextLinePos == std::string::npos 条件的满足条件，所以我们才会大胆的使用 source.size 这整个读出的字符串大小来计算位置。
因为此时 #type vertex 已经位于文件末尾了）

4.一些特殊情况呢，会怎么处理?

4.1（第一种）

如果处理的字符串是第一种情况

```
“#type vertex
adfdjkl
asgdas”
```

而不是第二种情况

```
“#type vertex
adfdjkl
asgdas

#type fragment”
```

结果：

则在前二种情况的第一次循环中，pos 会在 substr 处理之前通过 find() 进行更新，在第二种情况下 pos 是一个正常位置，会正确计算。
但在第一种情况的第一次循环中，pos 会因为找不到第二个 #type 而在更新后返回 std::string::npos, 这在计算时会出现 std::string::npos - nextLinePos。

4.2（第二种）

如果处理的字符串仅仅是

```
"#type vertex"
```

结果：

则在第一次循环中，处理 #type vertex 时，nextLinePos 会在 substr 运行之前被赋值为 std::string::npos ,pos 也会是 npos，
然后在 source.substr(nextLinePos, pos - (nextLinePos == std::string::npos ? source.size() - 1 : nextLinePos));
会返回 pos - (source.size() - 1) -> pos - source.size() + 1 -> std::string::npos - source.size() + 1.

思考：

根据查证，std::string::npos 是一个常量，代表了 std::string 类型中无效的位置。当 npos 与任何数值相减时，结果仍然是 npos，因为它代表的是一个无效的位置。
这表明上述情况会导致 substr () 返回无效字符串吗？这也许是一个 bug? 这果真是一个bug吗?

4.3 问题分析

首先我们另举一个例子来看：

```
“#type vertex
dasfgda
agdsa

#type fragment
sadfsga
asdgas”
```

问题分析：

我们得知在第一次while循环中没有什么异常，但在第二次 while 循环中：

size_t nextLinePos = source.find_first_not_of("\r\n", eol); 正常运行且返回有效数值，

但是会因为 #type fragment 之后没有出现类似的 #type XXX 而导致 pos = source.find(typeToken, nextLinePos); 返回 std::string::npos

问题解决：

4.3.1.此时循环是否会提前中断?

此时，尽管循环的条件是 (pos != std::string::npos) ,但我们没有使用条件判断手动退出，所以并不会提前退出。

而是会在更新 pos 之后，继续进行之后的语句，最后在下一次循环轮询时结束循环。

4.3.2.substr 的返回值是否无效?

并不是无效的，虽然

shaderSources[ShaderTypeFromString(type)] = source.substr(nextLinePos, pos - (nextLinePos == std::string::npos ? source.size() - 1 : nextLinePos));

使用 pos 来确定代码的结束位置，且 pos 在分析后应该是 std::string::npos,

但即使 pos 是 std::string::npos，std::string::substr 也会将结束位置自动设置为字符串的末尾。

因此即使没有下一个 #type 标记，代码仍然能够正确地截取到最后一个着色器的代码。

4.3.3.为什么substr ( ) 可以返回一个有效的值?

原型：

std::string substr(size_t pos = 0, size_t count = npos) const;

定义标明：

1.如果省略 count 参数（默认 count = npos），则函数默认截取从 pos 开始直到字符串的末尾。

2.如果填满参数，则当 pos 参数大于等于字符串的长度时，substr 会返回一个空字符串。

（人话：从整个字符串的末尾甚至更靠后的地方开始复制，这后面肯定没有字符了）

3.而如果 pos 参数小于字符串的长度但 count 参数过大导致超出字符串末尾时，substr 会自动将 count 调整为使得截取不超出字符串末尾的最大值。

（人话：开始位置正确，但是要复制的长度被设置的超出了字符串本身的大小，则截取字符串全部）

结论：

因此，当 count 参数被我们计算得到的值为 std::string::npos 时，substr ( ) 实际上会默认截取从 pos 开始直到字符串的末尾。

这就是为什么即使在没有下一个 #type 标记的情况下，代码仍然能够正确地截取到最后一个着色器的代码。

》》》》关于对一个指针类型变量使用 reset() 和 直接传递一个返回对应指针类型函数 这两种方法的不同。

eg.

m_TextureShader.reset(Nut::Shader::Create(textureVertexSrc, textureFragSrc));

m_TextureShader = Nut::Shader::Create(textureVertexSrc, textureFragSrc);二者的区别。

区别：

reset() 会在重新赋值之前释放原有的资源，从而避免内存泄漏。是一个安全的成员函数。

而直接使用 Create() 返回的指针（将指针赋值给一个改变量）并不会自动的进行资源释放，需要自己定义资源的正确释放。

》》》》发现 Cherno 的一个小错误

代码：

在PreProcess 函数中，Cherno 这样定义

```
while (pos != std::string::npos)
    {
        size_t begin = pos + tokenLength + 1;
        size_t end_of_line = source.find_first_of("\r\n", pos);
        NUT_CORE_ASSERT( end_of_line != std::string::npos, "Syntax error");
```

问题：

但是正常使用的时候，我发现 end_of_line 应该是一个有效的正确的值，却一直触发断言。

(end_of_line != std::string::npos) 返回的是一个 bool 量，其值确实为 true, 触发断言的原因是格式的错误。

改正：

正确形式应该是

NUT_CORE_ASSERT((end_of_line != std::string::npos), "Syntax error");

而不是

NUT_CORE_ASSERT(end_of_line != std::string::npos, "Syntax error");

原因：

如果不加这个括号，宏会被展开为

```
{
    if ( ! end_of_line != std::string::npos )
    { /* ... */ }
}
```

因为 ! 的优先级高于 !=，所以会变成if ((! end_of_line) != std::string::npos),这样一来即使 end_of_line 是 true（有效值），

!(end_of_line) 也会是 false，然后 false != std::string::npos 会是 true，这会导致断言的触发，而实际上代码并没有问题。

修改后则不会出现这种 end_of_line 被修改了的情况。

----------Shader Library（着色器库）----------

》》》》关于 Cherno 提到的问题

std::vector<GLenum> glShaders(shaderSources.size());　　//逻辑上是 resize

会导致初始化动态数组的时候提前新增两个空白的位置，而之后的 glShaders.push_back(shader); 才会推入正确的数据，而且位置位于前两个空白之后，

解决方式：
std::vector<GLuint> glShaders;
glShaders.reserve(shaderSources.size());　　//逻辑上是 reserve
这样可以为数组在内存中先分配确定的内存（先预留两个位置），当然也可以在两个位置之后继续动态拓展。
但并不用空白数据初始化这两个位置的内存，而是交给后面push_back进来的数据填充。

》》》》在临摹代码之前，我先将一些亟待解决的代码细节优化做了。以下是一些笔记。

》》(const void*)element.Offset 和 (const void*)(intptr_t)element.Offset的区别
(const void*)(intptr_t)element.Offset：
这个表达式先将 element.Offset 转换为 intptr_t 类型，然后将其强制转换为 const void* 类型。
其中intptr_t 是一种整数类型，足够大以容纳指针的位模式。（这样确保了数据传递和使用的正确性）

》》src/Nut/Input.h 中的更改是什么？为什么？
为什么：
首先，这个类是一个用于判断输入的抽象的基类，包含了一些序虚函数，作为接口被使用。
并且，这个类也被声明为单例类，意为这不能被实例化。（就算被实例化，这个对象也不包含任何数据，只包含能使用的接口）

是什么：
所以我们定义并禁用了两个函数：复制构造函数和赋值运算符
eg.
```
Input(const Input&) = delete;
Input& operator=(const Input&) = delete;
```
1.
= delete 关键字表明这两个函数是被删除的，也就是说被禁止使用。
2.
Input(const Input&) = delete;
复制构造函数在对象被复制时调用，用于将一个对象复制到另一个对象中。
通过将其定义为 = delete，编译器会禁止复制构造函数的使用。即禁止任何尝试复制该类对象的行为，从而确保该类对象不会被意外地复制。
3.
Input& operator=(const Input&) = delete;
赋值运算符用于将一个对象赋值给另一个对象。
通过将其定义为 = delete，编译器会禁止赋值运算符的使用。即禁止对该类对象进行赋值操作，从而确保对象不会被错误地赋值。