

----- scene viewport -----

》》》做了两件事：设置视口和设置相机比例

》》》为什么要设置 m\_ViewportSize 为 glm::vec2 而不是 ImVec2？

因为后面需要进行 != 运算，而 ImVec2 没有这个运算符的定义，只有 glm::vec2 有这个运算符的定义。

```
template<typename T, qualifier Q>
GLM_FUNC_QUALIFIER GLM_CONSTEXPR bool operator!=(vec<2, T, Q> const& v1, vec<2, T, Q> const& v2)
{
    return !(v1 == v2);
}
```

所以需要 ImVec2 接收 GetContentRegionAvail 返回的 ImVec2 类型的 panelSize，然后将两者进行比较。


```
ImVec2 panelSize = ImGui::GetContentRegionAvail();
if (m_ViewportSize != *(glm::vec2*)&panelSize)
{
}
```

》》》发现一个问题

```
ImVec2 panelSize = ImGui::GetContentRegionAvail(); // 获取面板大小
if (m_ViewportSize != *(glm::vec2*)&panelSize)
{
    m_ViewportSize = { panelSize.x, panelSize.y }; // 及时更新视口大小
    m_Framebuffer->Resize(m_ViewportSize.x, m_ViewportSize.y);
    m_CameraController.Resize(m_ViewportSize.x, m_ViewportSize.y);
}
ImGuiTextureID textureID = (void*)m_Framebuffer->GetColorAttachmentRendererID();
ImGui::Image(textureID, ImVec2{ m_ViewportSize.x, m_ViewportSize.y }, ImVec2{ 0, 1 }, ImVec2{ 1, 0 });
```



其中，无论对 m\_Framebuffer 是否调用 Resize，其渲染结果和响应好像都是一样的，并没有什么影响（实际上这应该对图像的分辨率有一定影响，但为何我没有发现什么明确特征？）。而且不调用 Framebuffer->Resize 的话，调整窗口大小的时候图像并不会出现闪烁的现象。（所以说闪烁正是因为帧缓冲对纹理附件的刷新而导致的）


》》》另一个问题


 @KennyTutorials 4年前

Im found a little bug when we double click on title bar / border, then engine is crashed and say that framebuffer isn't complete. Maybe this bug only on my engine? I think we just destroy the viewport window, but at same time trying drawing framebuffer on this window.



当我们双击标题栏/边框时，我发现了一个小错误，然后引擎崩溃并说帧缓冲区不完整。也许这个错误只出现在我的引擎上？我认为我们只是销毁视口窗口，但同时尝试在此窗口上绘制帧缓冲区。


  回复

 3 条回复



 @FlukierJupiter 4年前


use the ImGuiTabBarFlags\_NoTooltip flag when creating the ImGui window to prevent the window collapsing  
创建 ImGui 窗口时使用 ImGuiTabBarFlags\_NoTooltip 标志以防止窗口折叠

  回复

 @FlukierJupiter 4年前

you could just return from the invalidate function if the width or height is zero, before creating the frame buffer  
如果宽度或高度为零，则在创建帧缓冲区之前，您可以从无效函数返回

 2  回复

 @KennyTutorials 4年前

@FlukierJupiter Thanks for help! It works)

@FlukierJupiter 感谢您的帮助！有用)

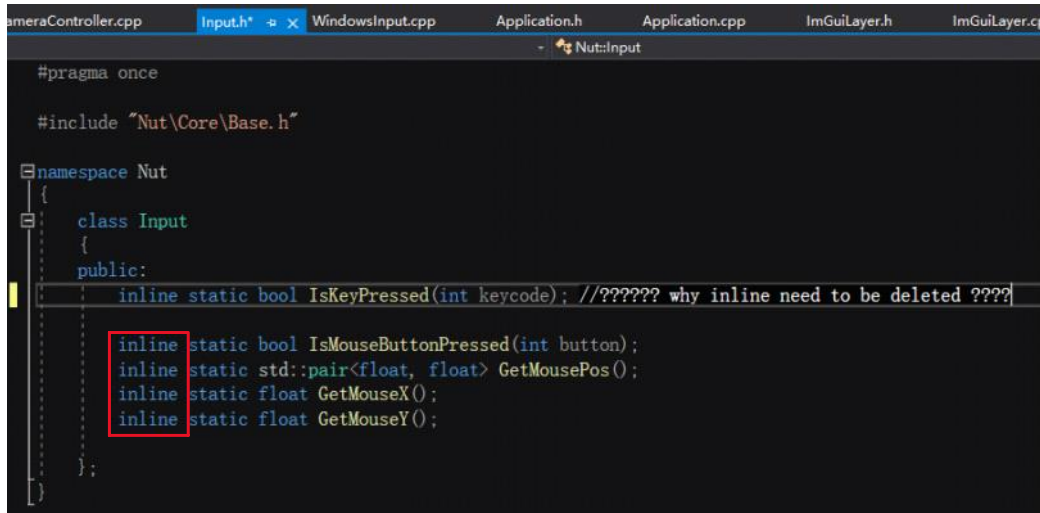
》》》值得一提的是，相机的纵横比更新函数参数需要为 float 类型的，而不是 uint 类型，否则会导致窗口尺寸过小时无渲染结果。

```
void OrthographicCameraController::Resize(float width, float height)
{
    m_AspectRatio = width / height;
    UpdateViewport();
}
```

## -----ImGui Layer Events-----

### 》》》发现一个问题:

Hazel中有一次维护是删除 inline 关键字的,我大致看了眼,觉得没有必要,就没有提交到 Nut,只是添加到待办里面了,这导致一个问题。



```
#pragma once

#include "Nut/Core/Base.h"

namespace Nut
{
    class Input
    {
    public:
        inline static bool IsKeyPressed(int keycode); //?????? why inline need to be deleted ???
        inline static bool IsMouseButtonPressed(int button);
        inline static std::pair<float, float> GetMousePos();
        inline static float GetMouseX();
        inline static float GetMouseY();
    };
}
```

操作:

在简化了Input.h之后,只剩下了5个函数的声明,而且这些函数在简化前都是内联函数,在.h文件中就已经定义过了。

建议:

所以在删除掉了定义之后,还应该删除inline关键字,我们要确保使用 inline 关键字的时候就对函数在头文件中定义,否则不添加inline关键字,避免出现错误。

如果仅仅删除了定义,但是没有删除inline关键字,就会出现 LNK2019 的报错,比如:

"public: static bool \_\_cdecl Nut::Input::IsKeyPressed(int)" (?IsKeyPressed@Input@Nut@@@SA\_NH@Z),  
函数 "public: void \_\_cdecl Nut::OrthoGraphicCameraController::OnUpdate(class Nut::Timestep)" (?OnUpdate@OrthoGraphicCameraController@Nut@@QEAXVTimestep@2@@Z) 中引用了该符号。

问题:

OrthoCameraController本应使用函数,可是为什么会查找不到,或者说对这个函数链接失败呢?

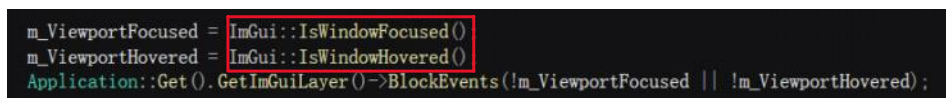
原因:

这正是因为我在头文件中只声明了函数为 inline,然后没在头文件中定义这个函数,而是在 CPP 文件中定义它。

此时编译器会在编译时找不到这个函数的定义,因为头文件已经告诉编译器这是一个 inline 函数,并期望在头文件中找到它的实现。

这样会导致链接错误或重复定义错误,这全都由于 CPP 文件中的定义与头文件中的 inline 声明不匹配。

### 》》》提醒:



```
m_ViewportsFocused = ImGui::IsWindowFocused();
m_ViewportsHovered = ImGui::IsWindowHovered();
Application::Get().GetImGuiLayer()->BlockEvents(!m_ViewportsFocused || !m_ViewportsHovered);
```

记得不要写成 ImGui::IsWindowFocused; ;)

## -----Where to go + Code review (前瞻与代码审核)-----

### 》》》Cherno 所做的

Cherno 在这一集前8分钟修复了一个小Bug,然后就算是开始审核代码了,基本上讲了自己对游戏引擎的理解与期望,还有接下来的进程。

### 》》》我将提交一些维护代码,因为这一集也没什么要做的。

### 》》》调整ImGui窗口大小时闪烁的原因是:我们在绘制ImGui窗口时同步更新了FrameBuffer和Camera



```
ImVec2 panelSize = ImGui::GetContentRegionAvail();
if (m_ViewportsSize != *(glm::vec2*)&panelSize)
{
    m_ViewportsSize = { panelSize.x, panelSize.y };
    m_Framebuffer->Resize(panelSize.x, panelSize.y);
    m_CameraController.Resize(panelSize.x, panelSize.y);
}
```

我们应该先更新，后绘制。

**问题出现原理以及解决方法：**

在 OnImGuiRender 函数中处理窗口大小变化时，你会在每一帧的渲染过程中检查窗口尺寸并同时处理窗口尺寸。因为在窗口调整时你重新创建了帧缓冲（Framebuffer），那么在调整过程中的某些渲染操作可能会使用未完全准备好的新帧缓冲，这就会导致显示的内容不稳定，从而产生闪烁。  
将窗口大小的调整逻辑提前放在 Onupdate 函数中，可以确保在每一帧的渲染之前已经完成了所有的帧缓冲调整。这意味着当 ImGuiRender 执行时，帧缓冲已经是正确的状态，减少了因帧缓冲调整导致的闪烁现象。

**未准备好的帧缓冲概念：**

- 1. 帧缓冲（Framebuffer）重建过程：  
当窗口大小变化时，通常需要重新创建或调整帧缓冲的尺寸，以适应新的窗口尺寸。这个过程包括删除旧的帧缓冲对象并创建新的对象，同时可能需要重新分配或调整与之关联的纹理和深度缓冲区。
- 2. 未准备好的帧缓冲：  
在帧缓冲重新创建或调整的过程中，新的帧缓冲可能尚未完全配置和初始化。例如，新的纹理可能尚未正确分配或绑定，或者深度缓冲区的设置尚未完成。在这个过渡期间，帧缓冲可能处于一个不稳定的状态，无法正确显示内容。

**》》》另外，还要注意一个问题：**

```
void EditorLayer::OnUpdate(Timestep ts)
{
    NUT_PROFILE_FUNCTION();

    // Logic Update
    ImVec2 panelSize = ImGui::GetContentRegionAvail();
    if (m_ViewportsSize != *((glm::vec2*)&panelSize) && m_ViewportsSize.x > 0.0f && m_ViewportsSize.y > 0.0f) {
        m_ViewportsSize = { panelSize.x, panelSize.y };
        m_Framebuffer->Resize((uint32_t)panelSize.x, (uint32_t)panelSize.y);
        m_CameraController.Resize(panelSize.x, panelSize.y);
    }

    if ((m_ViewportsSize.x != m_Framebuffer->GetSpecification().Width || m_ViewportsSize.y != m_Framebuffer->GetSpecification().Height)
        && m_ViewportsSize.x > 0.0f && m_ViewportsSize.y > 0.0f)
    {
        m_Framebuffer->Resize((uint32_t)m_ViewportsSize.x, (uint32_t)m_ViewportsSize.y);
        m_CameraController.Resize(m_ViewportsSize.x, m_ViewportsSize.y);
    }

    // Screen Update
    if (m_ViewportsFocused)
        m_CameraController.OnUpdate(ts);
}
```

第一种逻辑更新方式是不可取的，因为 ImGui::GetContentRefionAvail() 获取的是当前ImGui窗口的面板大小，需要在 ImGui 窗口绘制范围内进行使用，否则获取的Window值为nullptr，即没有找到可获取的ImGui窗口。  
第二种方式可取，因为每一次m\_Viewports在ImGui窗口事件触发时更新后，每当下一次绘制开始执行OnUpdate函数，m\_ViewportsSize已经是新窗口尺寸，而specification中存储的是旧窗口尺寸。这时触发Resize函数，随后帧缓冲m\_Framebuffer更新，相应的帧缓冲m\_Framebuffer中存储的specification也会更新为新窗口尺寸。  
下一次窗口大小改变时，也是类似的操作。

**逻辑：**

需要注意的是，这里的m\_Viewports值是在Onupdate函数执行后更新的，也就是说，图像的更新逻辑为：当前绘制时先判断逻辑，然后执行绘制。检测窗口尺寸变化的代码确实在绘制函数 OnImGuiRender中，不过没有直接绘制被更新的帧，而是将新窗口尺寸保留在全局变量m\_ViewportsSize中，在下次绘制开启前先在Onupdate更新窗体逻辑，然后在绘制函数中更新实际窗口尺寸。（简而言之，就是：当前帧检测到变化，但不更新，在下一帧开始时，发送变化值并执行更新）

-----ECS(实体系统)-----

》》》》76,77主要讲了EnTT的设计理念，使用方法以及使用案例，不是很难，我尽可能的做一些笔记以理解，同时上传77集的示例。

**》》》》ECS的概念：**

实体组件系统（ECS）是一种设计模式，常用于游戏开发和其他需要高性能和灵活性的应用程序中。它将对象分解为“实体”、“组件”和“系统”三个主要部分。  
实体是唯一的标识符，组件是数据容器，而系统是处理组件数据的逻辑模块。

**》》实体，组件，系统之间的关系：**

**1.实体 (Entity)**

定义：实体是系统中唯一的标识符，通常是一个简单的ID（比如unsigned int）。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)**

定义：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)**

定义：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。

**》》关系和使用方式**

**关系：**

实体与组件：  
实体是组件的容器，通过附加不同类型的组件来描述实体的属性和行为。实体本身没有实际的数据，只是一个标识符。  
组件与系统：  
系统会查询所有拥有特定组件集合的实体，并对这些实体的组件数据进行处理。例如，物理系统可能会查找所有拥有位置和速度组件的实体，并更新它们的位置数据。

**使用方式：**

创建实体：实例化一个新的实体并为其分配唯一标识符。  
添加组件：为实体附加一个或多个组件。每个组件存储实体的特定数据。  
定义系统：实现系统逻辑，这些系统会在每一帧或特定的事件触发时运行。  
更新：在每一帧或游戏循环中，系统会遍历实体并更新组件数据，实现游戏逻辑。

**示例**

假设你在开发一个简单的游戏，其中有角色实体（Player）和敌人实体（Enemy）。  
实体：  
playerEntity 和 enemyEntity。  
组件：  
PositionComponent：存储实体的位置数据 (x, y) 。  
VelocityComponent：存储实体的速度数据 (vx, vy) 。  
HealthComponent：存储实体的健康值。  
系统：  
MovementSystem：遍历所有具有 PositionComponent 和 VelocityComponent 的实体，并根据速度更新位置。  
HealthSystem：遍历所有具有 HealthComponent 的实体，处理伤害、恢复等健康相关的逻辑。

**》》ECS可以看做是一种数据结构吗，有什么优点？**

与其看做是数据结构，不如称之为是一种设计模式。

**优点：**

性能：ECS通过将数据（组件）分开存储，提高了缓存效率、减少内存碎片，从而提高性能。  
灵活性：ECS使得添加、删除和修改组件变得容易，而且系统将数据和行为分离的方式也使得数据和逻辑的耦合度降低，这有助于维护和扩展系统。使系统更加灵活和可扩展。

**》》》下载头文件：（[https://github.com/skypjack/entt/tree/master/single\\_include/entt](https://github.com/skypjack/entt/tree/master/single_include/entt)）**

这个库只需要加载头文件，因为entt是一个模板库，不需要链接。

**》》》entt::registry的原理与机制**

**概念：**

entt 的 entt::registry 是一个高度优化的数据结构，用于高效地管理实体及其组件。

**结构：**

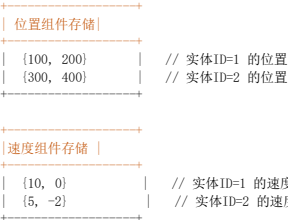
- 存放实体的结构  
实体ID：每个实体都有一个唯一的ID，这个ID用于标识实体，可以在内部数据结构中索引和检索。  
实体状态管理：entt::registry 维护一个实体池（entity pool）来跟踪实体的创建和销毁。通常是通过位图或其他类似的数据结构来管理实体。
- 管理组件的结构  
组件存储：entt::registry 为每种组件类型维护一个独立的存储结构，通常是一个类似于数组或向量的容器。每个组件的存储容器按所属的 实体ID进行索引，从而实现快速访问。  
组件映射：为了支持快速的组件查询和操作，entt::registry 使用组件映射（component map）来跟踪哪些实体拥有特定的组件。  
这种映射通常基于组件的签名（signature）来实现，签名是实体拥有的组件的集合。  
组件更新：entt::registry 允许高效的组件添加、删除和更新。组件的存储和管理通常采取增量更新的方式，以提高性能。

**存储结构图示：**

假设我们有两个实体（ID=1 和 ID=2），以及两个组件（位置和速度）。  
**实体管理：** entt::registry 维护一个实体池，跟踪所有有效的实体ID（比如，1 和 2）。



**位置组件：** 用一个数组或向量存储位置数据，比如 {{100, 200}, {300, 400}}，其中 {100, 200} 是玩家的位置，{300, 400} 是敌人的位置。  
**速度组件：** 用另一个数组或向量存储速度数据，比如 {{10, 0}, {5, -2}}，其中 {10, 0} 是玩家的速度，{5, -2} 是敌人的速度。



**位置映射:** entt::registry 使用一个哈希表将实体ID映射到位置数据。例如, ID=1 映射到 {100, 200}, ID=2 映射到 {300, 400}。  
**速度映射:** 类似地, ID=1 映射到 {10, 0}, ID=2 映射到 {5, -2}。

组件映射 (哈希表)	
位置:	
ID=1 -> {100, 200}	
ID=2 -> {300, 400}	
速度:	
ID=1 -> {10, 0}	
ID=2 -> {5, -2}	

#### 》》entt::registry的查询与更新:

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

// 定义组件
struct Position {
    float x, y;
};

struct Velocity {
    float vx, vy;
};

int main() {
    // 创建一个注册表
    entt::registry registry;

    // 使用 registry 创建实体并返回句柄
    auto entity = registry.create();

    // 为实体添加组件
    registry.emplace<Position>(entity, 10.0f, 20.0f);
    registry.emplace<Velocity>(entity, 1.0f, 2.0f);

    // 查询组件
    if (auto* pos = registry.try_get<Position>(entity)) {
        std::cout << "Position: (" << pos->x << ", " << pos->y << ")\n";
    } else {
        std::cout << "Entity has no Position component.\n";
    }

    if (auto* vel = registry.try_get<Velocity>(entity)) {
        std::cout << "Velocity: (" << vel->vx << ", " << vel->vy << ")\n";
    } else {
        std::cout << "Entity has no Velocity component.\n";
    }

    // 更新组件
    if (auto* pos = registry.try_get<Position>(entity)) {
        pos->x += 5.0f; // 增加 x 坐标
        pos->y -= 3.0f; // 减少 y 坐标
    }

    if (auto* vel = registry.try_get<Velocity>(entity)) {
        vel->vx *= 2.0f; // 增加 x 速度
        vel->vy *= 0.5f; // 减少 y 速度
    }

    return 0;
}
```

#### 》》》新版 Entt 的 entt::registry 好像没有 has 这个成员函数

#### 》》》结构化绑定:

```
auto group = m_Registry.group<TransformComponent, MeshComponent>();
for (auto entity : group)
{
    auto& [transform, mesh] = group.get<TransformComponent, MeshComponent>(entity);
}
```

这就是就是所谓的结构化绑定, 可以查看 Cherno C++ 系列的视频。

**Bilibili** [https://www.bilibili.com/video/BV11F411n7pw/?share\\_source=copy\\_web&vd\\_source=ca2feff7d155a2579964dfa2c3173769](https://www.bilibili.com/video/BV11F411n7pw/?share_source=copy_web&vd_source=ca2feff7d155a2579964dfa2c3173769)

**Youtube:** [STRUCTURED BINDINGS in C++](#)





Structured binding: 结构化绑定

语法: `auto [var1, var2, ...] = expression;`

auto 表示变量的类型将由编译器自动推导。  
[var1, var2, ...] 是结构化绑定的变量列表，用于接收解构后的值。  
expression 是一个可以解构的对象，通常是一个 std::tuple、std::pair 或自定义类型。

》》》On\_Construct 函数的意义:

```
m_Registry.on_construct<TransformComponent>().connect<&OnTransformConstruct>();
```

on\_construct 是用于注册回调函数，当某个组件类型的组件被添加到实体时，会触发这些回调。它允许你在组件创建时执行特定的操作。

connect<&OnTransformConstruct>(): 将 OnTransformConstruct 函数连接到这个信号。每当 TransformComponent 被添加到实体时，OnTransformConstruct 就会被调用。

-----Entities And Components-----



》》》感觉没啥要记的

`auto group = m_Registry.group<TransformComponent>(entt::get<SpriteComponent>);`

m\_Registry.group<TransformComponent>: 定义了一个组，其中的实体必须具有TransformComponent组件。  
entt::get<SpriteComponent>: 这是一个额外的条件，表示要包括SpriteComponent组件。  
这样，group中的实体不仅必须有TransformComponent，还必须有SpriteComponent。

-----Entity class-----

》》》强指针与弱指针的区别

强指针:	<p><b>定义:</b> 强指针是指在程序中直接引用对象的指针或引用。当一个对象有一个或多个强指针指向它时，该对象的<b>生命周期是由这些强指针管理的，直到所有强指针都被释放或指向其他对象。</b></p> <p><b>特性:</b> 只要存在一个强指针指向对象，该对象就不会被回收或释放。强指针会延长对象的生命周期，防止对象在被引用期间被销毁。</p>
弱指针:	<p><b>定义:</b> 弱指针是一种<b>不会阻止对象被回收的指针</b>。它通常用于引用那些可能会被销毁的对象，避免强引用循环所导致的内存泄漏。</p> <p><b>特性:</b> 弱指针不会改变对象的引用计数，这意味着即使存在弱指针指向对象，<b>只要没有强指针指向对象，该对象依然会被回收。</b> 弱指针通常与强指针一起使用，以避免内存泄漏问题。</p>

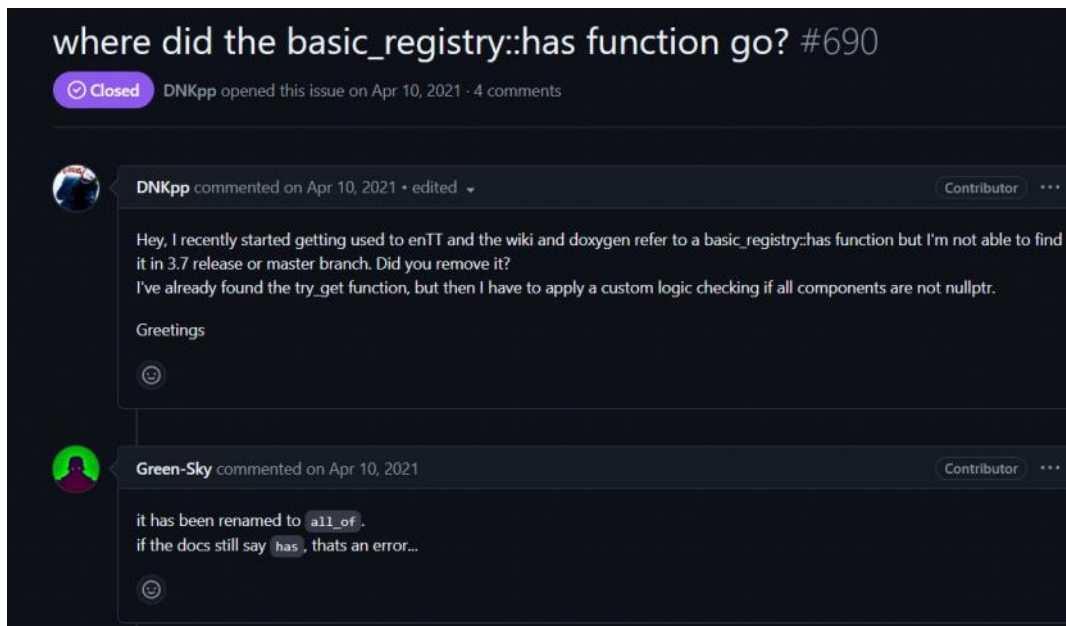
内存管理:

强指针	会增加对象的引用计数，使对象在所有强指针都被释放之前不会被回收。
弱指针	不会影响对象的引用计数，它只是一个辅助指针，用于访问对象，但不阻止对象的回收。

例如：  
在观察者模式的设计中，弱指针可以在观察对象时不干预对象生命周期。也就是说 std::weak\_ptr 可以通过避免持有对象的强引用来实现有效的内存管理。

》》》 registry.has() 函数被重命名了 现在叫 all\_of

You can find it with : <https://github.com/skypjack/entt/issues/690>



》》》关于 operator bool() const

这是一个**类型转换运算符 (type conversion operator)**，其作用是允许类 Entity 对象在需要布尔值的上下文中（如条件语句）被隐式地转换为 bool 类型。

例如:

```
operator bool() const { return m_EntityHandle != 0; }
```

在这里，m\_EntityHandle 是一个表示实体的句柄 (handle)。

如果 m\_EntityHandle 为 0 (null 或 空)，则 m\_EntityHandle != 0 返回 false，可以认为 Entity 对象无效（可能表示它已经被销毁或未初始化）。

否则，若 m\_EntityHandle 不为空，则 m\_EntityHandle != 0 结果为 true，可以认为它是有效的。

-----Camera system-----

》》》Cherno的意思好像是：在游戏编辑时，使用一种摄像机。但在游戏运行时，使用另外一种摄像机。当然这两种摄像机有别。

我的理解是，游戏在编辑的时候，需要编辑者全面/自由的观察游戏设计样貌，所以摄像机相对复杂一点，因为它要有能力在局部空间/世界空间中位移。而游戏运行时，大部分时间需要以一个固定的视角游玩，所以摄像机不用太复杂。

》》》primary 的作用

```
struct CameraComponent
{
    Hazel::Camera Camera;
    bool Primary = true;

    CameraComponent() = default;
    CameraComponent(const CameraComponent&) = default;
    CameraComponent(const glm::mat4& projection)
        : Camera(projection) {}
};
```

Primary 用于确定当前摄像机是否为主摄像机。也就是说当你使用此摄像机的时候，这个摄像机会被标记为主摄像机（当下正在使用的摄像机）。

根据这个标识，我们可以正确的对当下所观察的事物进行处理。

》》》Group 和 View 的区别:



## Views and Groups（视图和组）

视图是一种非侵入式工具，用于在不影响其他功能或增加内存消耗的情况下使用实体和组件。

组是一种侵入性工具，用于提高关键路径上的性能，但它也有代价。

视图主要有两种：编译时(也称为视图)和运行时(也称为runtime\_view)。

前者需要一个组件(或存储)类型的编译时列表，并因此可以进行一些优化。后者是在运行时使用数值类型标识符构建的，迭代速度稍慢。

在这两种情况下，创建和销毁视图的开销都不大，因为它们没有任何类型的初始化。

组有三种不同的类型：完全拥有组、部分拥有组和非拥有组。它们之间的主要区别在于性能。

组可以拥有一个或多个组件类型。它们可以重新排列池，以加快迭代速度。粗略地说：一个组拥有的组件越多，迭代它们的速度就越快。

一个给定的组件只能属于嵌套的多个组。用户必须仔细定义组，以获得最佳效果。

参考：（<http://t.csdnimg.cn/TZiz5>）

不同之处：

### 1. 基本概念

**view:**

概念：用于访问具有特定组件的实体。你可以创建一个视图来遍历所有包含某种或多种组件的实体。

特点：视图是相对轻量级的，并且对组件的访问是直接的。

视图通常是以某种组件的组合为基础进行过滤的。

**group:**

概念：是一个更复杂的数据结构，允许你对实体进行更多的分类和管理。

特点：group 不仅可以过滤组件，还可以根据实体的组件组合创建逻辑组。

group 可以指定需要的组件和额外的条件，以便管理和操作那些具有特定组件组合的实体。

### 2. 用法和功能

**view:**

用法：用于遍历符合组件条件的实体。

特点：访问速度较快，因为它不涉及复杂的分组逻辑，只是简单地提供对符合条件的实体的访问。

例子：

```
auto view = m_Registry.view<TransformComponent>();
for (auto entity : view) {
    auto& transform = view.get<TransformComponent>(entity);
    // 对实体进行操作
}
```

**group:**

用法：用于将实体分组到一个更复杂的集合中，考虑了多个组件及其组合。

特点：group提供了更高层次的管理功能，可以在创建时定义更复杂的过滤条件和数据访问逻辑。

例子：

```
auto group = m_Registry.group<TransformComponent>(entt::get<SpriteComponent>);
for (auto entity : group) {
    auto& transform = group.get<TransformComponent>(entity);
    auto& sprite = group.get<SpriteComponent>(entity);
    // 对实体进行操作
}
```

### 3. 性能和优化

**view:**

view主要依赖于组件的数据结构和缓存，通常较为高效，适合用于简单的组件过滤和遍历操作。

**group:**

由于其包含了更多的逻辑和条件，它可能在创建时需要额外的计算和管理开销。

但在需要处理更复杂的组件组合和分类时，group可以提供更好的性能优化。

### 总结:

使用view时，你是在基于组件的简单过滤进行操作，适合轻量级的遍历和访问。

使用group时，你可以进行更复杂的组件组合和条件过滤，更适合需要进行复杂数据管理和分类的场景。

这取决于你的具体需求和性能要求。

》》》》为什么这一集之后，键盘上的 wasd 对摄像机控制失效了。

因为之前绘制的时候，通过 BeginScene( OrthographicCamera camera ) 传入的是一个 OrthographicCamera，而且我们在更新的时候 CameraController 操控的也是 OrthographicCamera（每一次响应wasd，都在通过 UpdateMatrix() 更新 OrthographicCamera）。

但是在本次绘制的实体中，通过 BeginScene( Camera camera ) 我们传入的都是新类型 Camera，并且在 CameraController 的更新函数中没有对 Camera 类型摄像机进行更新的函数，这就导致在仅绘制 Camera 类型摄像机时，WASD 不起作用。



》》》为什么Cherno在代码中总是获取位移矩阵的第四列 (Transform[3])，以此对物体进行操纵呢？

因为OpenGL中的位移矩阵是列主序的，所以位移向量存储在矩阵的第4列（代码中表示为 0,1,2,3 这 4 个标号中的第 4 个：也就是 3）

## 位移

**位移**(Translation)是在原始向量的基础上加上另一个向量从而获得一个在不同位置的新向量的过程，从而在位移向量基础上**移动**了原始向量。我们已经讨论了向量加法，所以这应该不会太陌生。

和缩放矩阵一样，在4×4矩阵上有几个特别的位置用来执行特定的操作，对于位移来说它们是第四列最上面的3个值。

如果我们把位移向量表示为 $(T_x, T_y, T_z)$ ，我们就能把位移矩阵定义为：

$$\begin{bmatrix} 1 & 0 & 0 & T_x \\ 0 & 1 & 0 & T_y \\ 0 & 0 & 1 & T_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x + T_x \\ y + T_y \\ z + T_z \\ 1 \end{pmatrix}$$

这样是能工作的，因为所有的位移值都要乘以向量的w行，所以位移值会加到向量的原始值上（想想矩阵乘法法则）。而如果你用3x3矩阵我们的位移值就没地方放也没地方乘了，所以是不行的。

参考：（[变换 - LearnOpenGL.CN \(learnopengl-cn.github.io\)](#)）

》》》一个提示：

```
void Scene::OnUpdate(Timestep ts)
{
    Camera* mainCamera = nullptr;
    glm::mat4* mainTransform = nullptr;

    auto view = m_Registry.view<TransformComponent, CameraComponent>();
    for (auto entity : view)
    {
        auto& [transform, camera] = view.get<TransformComponent, CameraComponent>(entity);
        // 若检测到某摄像机被标记为主摄像机，则传出主摄像机的数据，然后跳出。
        if (camera.Primary)
        {
            mainCamera = &camera.Camera;
            mainTransform = &transform.Transform;
            break;
        }
    }

    if (mainCamera) {
        // Do some rendering (获取当前摄像机的投影矩阵 projection 和 位移矩阵 transform,
        Renderer2D::BeginScene(*mainCamera, inverse(*mainTransform));

        auto group = m_Registry.group<TransformComponent>(entt::get<SpriteComponent>); // 在所有含有
        for (auto entity : group) {
            auto& [transform, color] = group.get<TransformComponent, SpriteComponent>(entity);

            Renderer2D::DrawQuad(transform.Transform, color.Color);
        }

        Renderer2D::EndScene();
    }
}
```

Camera 中的投影矩阵被用为 BeginScene 的一个参数，所以要确保运行之前已经为 CameraComponent 的成员 Camera 添加了数据。

```
m_SecondCamera = m_ActiveScene->CreateEntity("Clip-Camera");
m_SecondCamera.AddComponent<SpriteComponent>(glm::vec4{ 0.5412f, 0.1686f, 0.8863f, 1.0f });
auto& secondController = m_SecondCamera.AddComponent<CameraComponent>(glm::ortho(-1.0f, 1.0f, -1.0f, 1.0f, -1.0f, 1.0f));
secondController.Primary = true;
```

》》》一个理解：

这一集需要我们重载 `Renderer2D::BeginScene()`

Cherno 是这样做的:

```
+ void Renderer2D::BeginScene(const Camera& camera, const glm::mat4&
+ transform)
+ {
+     HZ_PROFILE_FUNCTION();
+
+     glm::mat4 viewProj = camera.GetProjection() *
+     glm::inverse(transform);
+
+     s_Data.TextureShader->Bind();
+     s_Data.TextureShader->SetMat4("u_ViewProjection",
+     viewProj);
+
+     s_Data.QuadIndexCount = 0;
+     s_Data.QuadVertexBufferPtr = s_Data.QuadVertexBufferBase;
+
+     s_Data.TextureSlotIndex = 1;
+ }
+ 
```

我是这样做的

```
void Renderer2D::BeginScene(const Camera& camera, const glm::mat4& viewMatrix)
{
    NUT_PROFILE_FUNCTION();

    glm::mat4 viewProjectionMatrix = camera.GetProjection() * viewMatrix;

    s_Data.TextureShader->Bind();
    s_Data.TextureShader->SetMat4("u_ViewProjection", viewProjectionMatrix);

    s_Data.QuadIndexCount = 0;
    s_Data.TextureSlotIndex = 1;
    s_Data.QuadVBHind = s_Data.QuadVBBase;
}

if (mainCamera) {
    // Do some rendering (获取当前摄像机的投影矩阵 projection 和 位移矩阵 transform,
    Renderer2D::BeginScene(*mainCamera, inverse(*mainTransform));
}
```

其实差不多, 我在填入参数时就将 transform 位移矩阵转换成了 view 观察矩阵。

### 》》接下来看看重载 `BeginScene()` 这个函数目的:

- 在先前的 `Renderer2D::BeginScene()` 中, 我们传入的是 `OrthographicCamera`:

```
void Renderer2D::BeginScene(const OrthographicCamera& camera)
{
    NUT_PROFILE_FUNCTION();

    s_Data.TextureShader->Bind();
    s_Data.TextureShader->SetMat4("u_ViewProjection", camera.GetViewProjectionMatrix());

    s_Data.QuadIndexCount = 0;
    s_Data.TextureSlotIndex = 1;
    s_Data.QuadVBHind = s_Data.QuadVBBase;
}
```

这么做的用意是, 可以直接通过 `OrthographicCamera` 对象的成员函数获取视图投影矩阵 `ViewProjectionMatrix`, 然后将其上传为统一变量, 以便计算与呈现渲染结果。

- 但是在游戏运行时, 我们只想进行简单的计算, 并使用简单的 `Camera` 类型 (仅仅包含投影矩阵: `ProjectionMatrix`), 这时候想要绘制一个图形, 在使用 `Renderer2D::BeginScene()` 时, 就需要进行重载, 将 `Camera` 类型对象作为参数的一员。

```

void Renderer2D::BeginScene(const Camera& camera, const glm::mat4& viewMatrix)
{
    NUT_PROFILE_FUNCTION();

    glm::mat4 viewProjectionMatrix = camera.GetProjection() * viewMatrix;

    s_Data.TextureShader->Bind();
    s_Data.TextureShader->SetMat4("u_ViewProjection", viewProjectionMatrix);

    s_Data.QuadIndexCount = 0;
    s_Data.TextureSlotIndex = 1;
    s_Data.QuadVBHind = s_Data.QuadVBBase;
}

```

所以我们需要传入一个位移矩阵 transform 在 Renderer2D::BeginScene() 临时的计算一下视图投影矩阵 ViewProjectionMatrix，然后将其上传为统一变量。

➤ 值得一提的是，这个视图矩阵 viewMatrix 就是由位移矩阵 transform 进行转置运算得来的，这一点在之前的 OrthographicCamera.cpp 中也可见端倪：

```

void OrthographicCamera::UpdateViewMatrix()
{
    NUT_PROFILE_FUNCTION();

    glm::mat4 transform = glm::translate(glm::mat4(1.0f), m_Position)
        * glm::rotate(glm::mat4(1.0f), glm::radians(m_Rotation), glm::vec3(0.0f, 0.0f, 1.0f));

    m_ViewMatrix = glm::inverse(transform);

    m_ViewProjectionMatrix = m_ProjectionMatrix * m_ViewMatrix;
}

```

》》》纠正两个Cherno的错误：

```

void Scene::OnUpdate(Timestep ts)
{
    Camera* mainCamera = nullptr;
    glm::mat4* mainTransform = nullptr;

    auto view = m_Registry.view<TransformComponent, CameraComponent>();
    for (auto entity : view)
    {
        auto& [transform, camera] = view.get<TransformComponent, CameraComponent>(entity);
        // 若检测到某摄像机被标记为主摄像机，则传出主摄像机的数据，然后跳出。
        if (camera.Primary)
        {
            mainCamera = &camera.Camera;
            mainTransform = &transform.Transform;
            break;
        }
    }

    if (mainCamera) {
        // Do some rendering (获取当前摄像机的投影矩阵 projection 和 位移矩阵 transform,
        Renderer2D::BeginScene(*mainCamera, inverse(*mainTransform));

        auto group = m_Registry.group<TransformComponent>(entt::get<SpriteComponent>); // 在所
        for (auto entity : group) {
            auto& [transform, color] = group.get<TransformComponent, SpriteComponent>(entity);

            Renderer2D::DrawQuad(transform.Transform, color.Color);
        }

        Renderer2D::EndScene();
    }
}

```

```

void Scene::OnUpdate(Timestep ts)
{
    // Render 2D

    Camera* mainCamera = nullptr;
    glm::mat4* cameraTransform = nullptr;

    {
        auto group = m_Registry.view<TransformComponent,
            CameraComponent>();
        for (auto entity : group)
        {
            auto& [transform, camera] =
                group.get<TransformComponent, CameraComponent>(entity);

            if (camera.Primary)
            {
                mainCamera = &camera.Camera;
                cameraTransform =
                    &transform.Transform;
                break;
            }
        }
    }

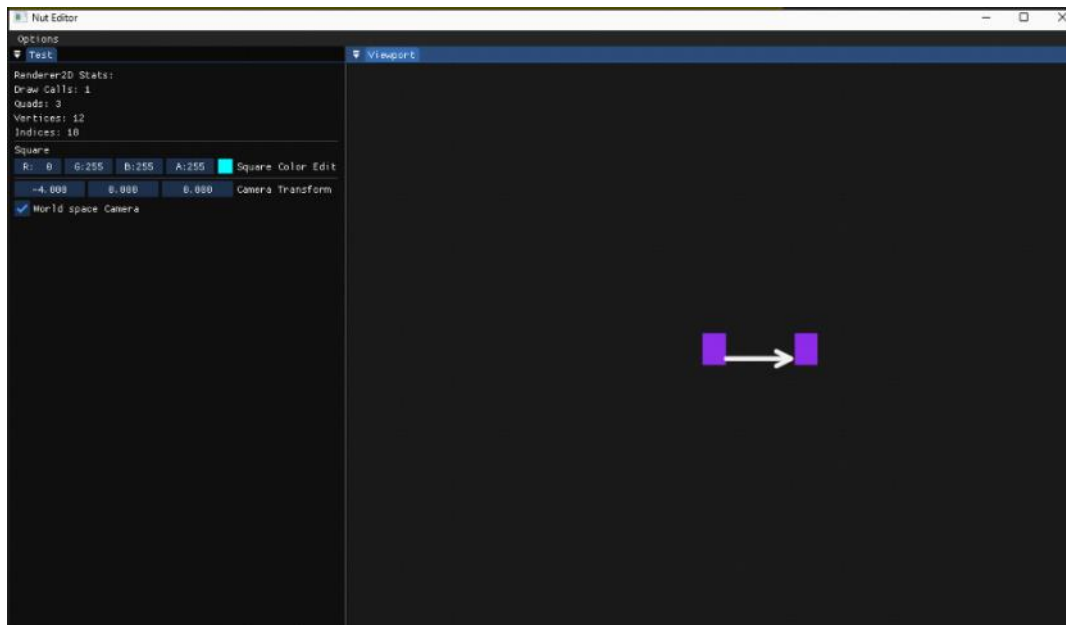
    if (mainCamera)
    {
        Renderer2D::BeginScene(mainCamera->GetProjection(),
            *cameraTransform);
    }
}

```

1.视图的命名是不是使用错了？

2.BeginScene() 的参数是不是填错了？

》》》又发现一个问题：绘制的图像在位移之后，其初始位置的图像却一直在绘制，没有清除。



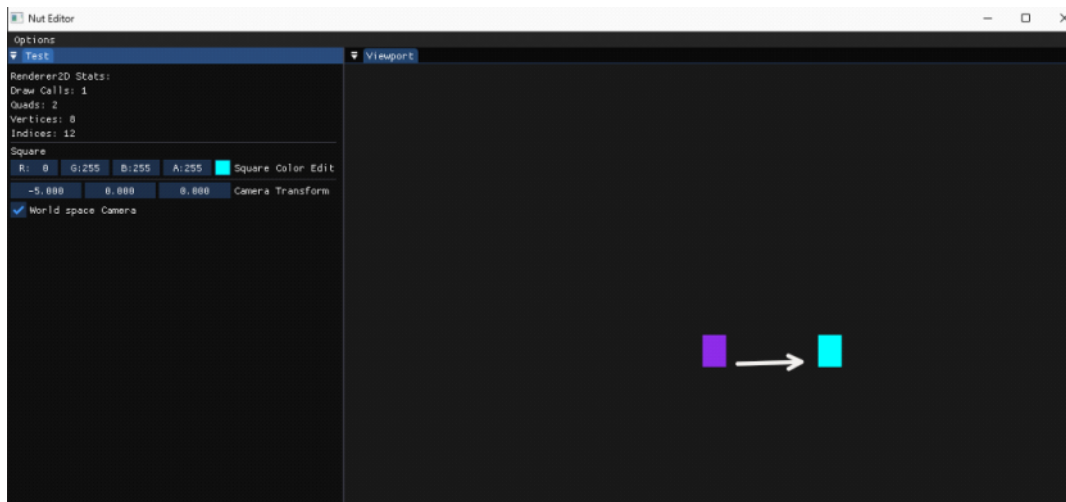
后来我发现原因出在这里：（注释掉之后运行情况正常）

```
m_CameraEntity = m_ActiveScene->CreateEntity("Main-Camera");
//m_CameraEntity.AddComponent<SpriteComponent>(glm::vec4( 0.5412f, 0.1686f, 0.8863f, 1.0f ));
auto& FirstController = m_CameraEntity.AddComponent<CameraComponent>(glm::ortho(-10.0f, 10.0f, -9.0f, 9.0f, -1.0f, 1.0f));
FirstController.Primary = true;

m_SecondCamera = m_ActiveScene->CreateEntity("Clip-Camera");
//m_SecondCamera.AddComponent<SpriteComponent>(glm::vec4( 0.5412f, 0.1686f, 0.8863f, 1.0f ));
auto& secondController = m_SecondCamera.AddComponent<CameraComponent>(glm::ortho(-1.0f, 1.0f, -1.0f, 1.0f, -1.0f, 1.0f));
secondController.Primary = false;
```

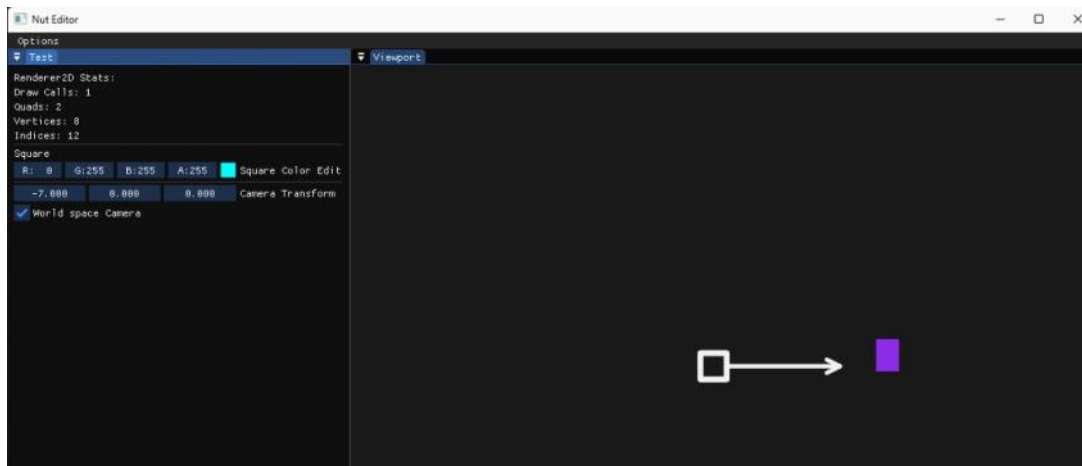
由于我为这两个摄像机实体都添加了 sprite 组件，所以在绘制图形的时候，额外绘制了当前摄像机比例下的两个图形。但不可否认的是，新添加的两个图像由于 CPU 运行顺序，导致两个新图在绘制时都是覆盖在旧图之上的。

如果解除1的注释，但不解除2的注释，效果是这样的。（初始位置多绘制一个图像）



如果解除2的注释，但不解除1的注释，效果是这样的。（总有一个紫色的新图像覆盖在旧的蓝图像之上）





为何会是这两句代码导致错误呢，其中的逻辑是什么？

想不出来，我现在想睡觉。

-----Scene Camera (fixed aspect ratio)-----

》》》在C++中，整数除法和浮点数除法有什么区别吗

整数除法：操作数都是整数（int, long, short等），结果也是整数。

浮点数除法：操作数至少有一个是浮点数（float, double, long double），结果是浮点数。

》》》什么类可以访问 其他类中 protected 类型的成员变量？

- 派生类可以访问基类（父类）中的 protected 成员。
  - < 派生类（子类）不能直接访问基类（父类）的 private 成员 >
- 友元类和友元函数可以访问类的 protected 成员。
- 类内部的成员函数可以访问该类的 protected 成员。

》》》关于子类与父类构造函数的关系。

前提：子类在没有显式地调用父类的构造函数时，编译器依旧会尝试隐式调用父类的默认构造函数。如果父类没有定义默认构造函数，那么编译器就会产生错误。

原因：

- 子类与父类构造函数的结构：
  - 子类的构造函数会首先调用父类的构造函数，以确保父类部分被正确初始化。子类构造函数可以通过初始化列表显式指定调用哪个父类构造函数。
- 初始化顺序：父类的构造函数先于子类的构造函数执行
  - 即使子类的构造函数中没有显式调用父类构造函数，父类的构造函数也会被自动调用。
  - 所以如果父类没有默认构造函数，子类构造函数就必须提供一个有效的父类构造函数调用。

》》》Button & Bottom: button是按钮，bottom是底部。

》》》为什么在正交矩阵的计算中，对于 bottom 和 top 这两个参数不用乘以纵横比 AspectRatio?

```
void SceneCamera::UpdateProjection()
{
    float orthoLeft = -m_OrthographicSize * m_AspectRatio * 0.5f;
    float orthoRight = m_OrthographicSize * m_AspectRatio * 0.5f;
    float orthoBottom = -m_OrthographicSize * 0.5f;           // Why not multiply by AspectRatio ?
    float orthoTop = m_OrthographicSize * 0.5f;

    m_ProjectionMatrix = glm::ortho(orthoLeft, orthoRight, orthoBottom, orthoTop, m_OrthographicNear, m_OrthographicFar);
}
```

因为 m\_OrthographicSize 已经算是定义了视口的高度。所以在确定的高度之下，只需要对宽度（从 Left 到 Right）进行比例换算即可得到正确的视觉效果。

如果只对高度（从 Bottom 到 Top）乘以纵横比 AspectRatio，但不对宽度进行计算，结果应该是相似的，只不过会变扁一点。

》》》一个提醒



```
>E:\VS\Nut\Nut\vendor\Entt\include\entt.hpp(17568,62): error C2440: "<function-style-cast>": 无法从"initializer list"转换为"Type"
2>    with
2>    [
2>        Type=Nut::CameraComponent
2>    ] (编译源文件 src\EditorLayer.opp)
2>E:\VS\Nut\Nut\vendor\Entt\include\entt.hpp(17568,62): message : 无效的联合初始化 (编译源文件 src\EditorLayer.opp)
```

#### 错误缘由:

这个错误是由于你尝试用不兼容的方式初始化 Nut::CameraComponent 对象。  
在 Entt 库的代码中，它期望某种特定的初始化方式初始化对象，但你的代码使用了不匹配的方式。

#### 排除错误:

这里就需要检查 Nut::CameraComponent 的构造函数或 Entt 库的文档。

#### 纠正错误:

```
m_CameraEntity = m_ActiveScene->CreateEntity("Main-Camera");
auto& firstController = m_CameraEntity.AddComponent<CameraComponent>(glm::ortho(-16.0f, 16.0f, -9.0f, 9.0f, -1.0f, 1.0f));
firstController.Primary = true;

m_SecondCamera = m_ActiveScene->CreateEntity("Clip-Camera");
auto& secondController = m_SecondCamera.AddComponent<CameraComponent>(glm::ortho(-1.0f, 1.0f, -1.0f, 1.0f, -1.0f, 1.0f));
secondController.Primary = false;
```

由于更换了 CameraComponent 中的成员 Camera ( Nut::Camera -> SceneCamera )，我们需要删除 AddComponent 的参数 glm::ortho(...)

(

- 因为子类 SceneCamera 的所有构造都不需要填入参数，而且父类的默认构造函数也不需要填入参数，所以在此处为 CameraComponent 的成员 SceneCamera Camera 初始化的时候不需要填入数据。
- 而且在默认情况下，我们还为父类的成员：投影矩阵 ProjectionMatrix 添加了一个默认值 glm::mat4(1.0f) ,以防没有填入参数对其带来未定义的错误。随后我们在 Camera 的子类 SceneCamera 中更新了投影矩阵，投影矩阵主要在 SceneCamera 中定义。（投影矩阵由 glm::ortho() 中填入的参数决定并定义。）

)。

```
void SceneCamera::UpdateProjection()
{
    float orthoLeft = -m_OrthographicSize * m_AspectRatio * 0.5f;
    float orthoRight = m_OrthographicSize * m_AspectRatio * 0.5f;
    float orthoBottom = -m_OrthographicSize * 0.5f; // Why not multiply by AspectRatio ?
    float orthoTop = m_OrthographicSize * 0.5f;

    m_ProjectionMatrix = glm::ortho(orthoLeft, orthoRight, orthoBottom, orthoTop, m_OrthographicNear, m_OrthographicFar);
}
```

#### 》》》》另一个提示:

由于我们去除了为摄像机组件 CameraComponent 填入的正交矩阵 glm::ortho(...), 所以现在两个摄像机实体 m\_CameraEntity 和 m\_SecondCamera 在空间中看起来是一样的（大小、比例...）。  
( [“GameEngine6” 页上的一个图片【来自: 》》》》一个提醒】](#) )

这是因为我们的投影矩阵在之前由摄像机组件 CameraComponent 填入的正交矩阵 glm::ortho(...)决定（本来填入的 glm::ortho(...) 直接为组件中的成员 Nut::Camera Camera 进行初始化，现在组件中的成员改为了 Nut::SceneCamera Camera，后者不需要填入矩阵参数），而现在由父类 SceneCamera 中的函数 UpdateProjection() 决定。  
( [“GameEngine6” 页上的图片【来自: 》》》》一个提醒】](#) )

一开始，组件成员为 Nut::Camera Camera 时，填入的正交矩阵 glm::ortho 直接被父类 Camera 的投影矩阵 m\_ProjectionMatrix 所用；现在组件成员改为 Nut::SceneCamera Camera 之后，不用填入正交矩阵 glm::ortho，但是这样也散失了灵活性，因为现在子类 SceneCamera 中从父类 Camera 继承的 m\_ProjectionMatrix 被固定的数据计算出来，而且上传为统一变量。

在 UpdateProjection() 函数中，所有的数据由私有成员计算得来，这些成员在类对象初始化时就已经定义了，而且每一个对象的默认值都一样：

```

// 继承 Camera 类中的私有变量 m_ProjectionMatrix, 用于该类中。
class SceneCamera : public Camera
{
public:
    SceneCamera();
    virtual SceneCamera() = default;

    void SetOrthographic(float orthographicSize, float orthographicNear, float orthographicFar);

    void SetOrthographicSize(float size) { m_OrthographicSize = size; }
    float GetOrthographicSize() const { return m_OrthographicSize; }

    void ViewportResize(uint32_t width, uint32_t height);
private:
    void UpdateProjection(); // UpdateProjection()
private:
    float m_AspectRatio = 0.0f;
    float m_OrthographicSize = 10.0f; // Camera's orthographic size
    float m_OrthographicNear = -1.0f, m_OrthographicFar = 1.0f;

    void SceneCamera::UpdateProjection()
    {
        float orthoLeft = -m_OrthographicSize * m_AspectRatio * 0.5f;
        float orthoRight = m_OrthographicSize * m_AspectRatio * 0.5f;
        float orthoBottom = -m_OrthographicSize * 0.5f; // Why not multiply by AspectRatio?
        float orthoTop = m_OrthographicSize * 0.5f;

        m_ProjectionMatrix = glm::ortho(orthoLeft, orthoRight, orthoBottom, orthoTop, m_OrthographicNear, m_OrthographicFar);
    }
};

```

父类 Camera 或子类 SceneCamera 所计算并更新的投影矩阵 m\_ProjectionMatrix 在这里被获取并上传至统一变量。



```

void Renderer2D::BeginScene(const Camera& camera, const glm::mat4& viewMatrix)
{
    NUT_PROFILE_FUNCTION();

    glm::mat4 viewProjectionMatrix = camera.GetProjection() * viewMatrix;

    s_Data.TextureShader->Bind();
    s_Data.TextureShader->SetMat4("u_ViewProjection", viewProjectionMatrix);

    s_Data.QuadIndexCount = 0;
    s_Data.TextureSlotIndex = 1;
    s_Data.QuadVBHind = s_Data.QuadVBBase;
}

```

-----Native Scripting (本机脚本) -----

》》》》十分抱歉因为前两天出去玩，后面又有事情耽搁，一直没更新。现在我试着恢复到工作状态中来。



》》》什么是脚本？

**概念：**  
在编程中，**脚本 (Script)** 通常指的是一种用于自动化任务的程序代码。  
脚本语言通常是解释性语言，意味着它们不需要编译成机器代码，可以直接由解释器逐行执行。

**常用于：**  
自动化任务，系统管理，网页开发，数据分析，测试等目的。  
**常见脚本语言：**  
常见的脚本语言包括Python、JavaScript、Bash、Perl和Ruby。

》》》本机脚本和普通的脚本有什么不同？

<b>普通脚本 (Managed Scripts) 以 Unity 为例</b>	1.语言： 通常使用 C# 编写。 2.运行环境： 运行在 Unity 的 Mono 或 .NET 运行时环境中。这些脚本是托管代码，由 Unity 的托管环境处理。 3.编译： 在 Unity 编辑器中，C# 脚本被编译成 .NET 程序集 (DLLs) 。 4.接口： 通过 Unity 的 MonoBehaviour 类和 Unity 的 API 访问和操作游戏对象和组件。 5.调试： 可以通过 Unity 编辑器的调试工具或 Visual Studio 等 IDE 进行调试。 6.特性： 因为采用的语言 C#， 使其易于编写和维护，因为它们利用了 .NET 的垃圾回收和其他高级语言功能。
<b>本机脚本 (Native Scripts)</b>	1.语言： 通常使用 C++ 或 C 编写。这些脚本通过 Unity 的本机插件接口 (Native Plugin Interface) 集成。 2.运行环境： 直接运行在操作系统的本机环境中，而不是 Unity 的托管环境中。 3.编译： 需要编译成平台特定的动态链接库 (DLLs、so 文件、dylib 文件等) 。 4.接口： 通过 Unity 提供的本机插件接口进行交互。Unity 允许通过 DllImport 等机制调用本机插件中的函数。 5.调试： 调试可能会更加复杂，因为它涉及到不同的工具和环境，通常需要使用本机开发工具（如 Visual Studio 的 C++ 部分）。 6.特性： C++ 提供了对更底层硬件和系统资源的访问，可以优化性能，但编写和维护更为复杂。还需要处理内存管理和其他低级问题。

使用场景

普通脚本：	适合大多数游戏逻辑、用户界面、输入处理等应用场景。
本机脚本：	适合需要高性能计算、平台特定功能、或与现有本机代码库集成的场景。 例如，进行复杂的数学计算或直接操作硬件等。

总结

普通脚本提供了 **更高层次的抽象和易用性**，而本机脚本则提供了 **更低层次的控制和优化能力**。

》》》函数指针：

Check it with:

Bilibili: [https://www.bilibili.com/video/BV1254y1h7Ha/?share\\_source=copy\\_web&vd\\_source=ca2feff7d155a2579964dfa2c3173769](https://www.bilibili.com/video/BV1254y1h7Ha/?share_source=copy_web&vd_source=ca2feff7d155a2579964dfa2c3173769)  
YouTube: [Function Pointers in C++](#)



#### 》》》std::function

##### 概念:

std::function 是 C++ 标准库中的一个可调用对象封装器，定义在 <functional> 头文件中。

它可以存储和调用任意类型的可调用对象，包括函数指针、Lambda 表达式、函数对象（即重载了 operator() 的类），甚至是绑定了参数的函数。

##### 使用:

定义: 声明一个 std::function 对象，并指定其接受的参数和返回类型。	<code>std::function&lt;void(int)&gt; func;</code>
赋值: 将函数指针、Lambda 表达式或函数对象等等赋值给 std::function 对象。	<code>func = [](int x) { std::cout &lt;&lt; x &lt;&lt; std::endl; }; func(10); // 输出 10</code>
使用: 直接调用 std::function 对象，就像调用普通函数一样。	<code>void print(int x) {     std::cout &lt;&lt; x &lt;&lt; std::endl; }  func = print; func(20); // 输出 20</code>

#### 》》》Lambda 表达式

**Lambda 表达式在 C++ 中是定义匿名函数的简洁方式。**

基本语法: `[capture](parameters) -> return_type { body }`

##### Lambda 表达式的组成部分

捕获列表 [capture]:	指定 Lambda 表达式可以访问的外部变量。 (决定了 Lambda 表达式如何访问外部变量)
参数列表 (parameters):	定义 Lambda 表达式的参数 (决定 Lambda 表达式接受哪些参数)
返回类型 -> return_type:	指定 Lambda 表达式的返回类型 (如果编译器无法自动推断的话)。这部分是可选的。
函数体 { body }:	Lambda 表达式的实现部分。

**捕获列表 [capture] 决定了 Lambda 表达式如何捕获外部变量。**你可以通过不同的方式来捕获这些变量:

[&]:	以引用的方式捕获所有外部变量。这意味着 Lambda 表达式使用并修改外部变量。
[=]:	以值的方式捕获所有外部变量。这意味着 Lambda 表达式会创建外部变量的副本，并且对这些副本的修改不会影响外部变量。
[&var]:	以引用的方式捕获指定的变量 var，其他外部变量不被捕获。
[=, &var]:	以值的方式捕获所有外部变量，但以引用的方式捕获指定的变量 var。

You Can Check it with:

**Bilibili:** [https://www.bilibili.com/video/BV195411A7KX/?share\\_source=copy\\_web&vd\\_source=ca2feff7d155a2579964dfa2c3173769](https://www.bilibili.com/video/BV195411A7KX/?share_source=copy_web&vd_source=ca2feff7d155a2579964dfa2c3173769)

**YouTube:** [Lambdas in C++](#)



### 》》》关于代码的一些理解

```
53 + struct NativeScriptComponent
54 + {
55 +     ScriptableEntity* Instance = nullptr;
56 +
57 +     std::function<void()> InstantiateFunction;
58 +     std::function<void()> DestroyInstanceFunction;
59 +
60 +     std::function<void(ScriptableEntity*)>
61 +     OnCreateFunction;
62 +     std::function<void(ScriptableEntity*)>
63 +     OnDestroyFunction;
64 +     std::function<void(ScriptableEntity*,
65 +     Timestep)> OnUpdateFunction;
66 +
67 +     template<typename T>
68 +     void Bind()
69 +     {
70 +         InstantiateFunction = [&]() { Instance
71 +         = new T(); };
72 +         DestroyInstanceFunction = [&]() {
73 +         delete (T*)Instance; Instance = nullptr; };
74 +         OnCreateFunction = []
75 +         (ScriptableEntity* instance) { ((T*)instance)->OnCreate(); };
76 +         OnDestroyFunction = []
77 +         (ScriptableEntity* instance) { ((T*)instance)->OnDestroy(); };
78 +         OnUpdateFunction = []
79 +         (ScriptableEntity* instance, Timestep ts) { ((T*)instance)-
80 +         >OnUpdate(ts); };
81 +     }
82 + }
```

### 》》Instance 的作用?

因为我们计划通过脚本组件中的 OnCreateFunction、OnDestroyFunction、OnUpdateFunction 来获取脚本类中自定义的 3 个函数。所以为了 OnCreateFunction、OnDestroyFunction、OnUpdateFunction 能够获取到指定脚本类中的函数，我们需要在调用函数时获取到合适的脚本类对象，然后在 OnCreateFunction、OnDestroyFunction、OnUpdateFunction 中调用该对象的成员函数。

### 》》为什么不能将脚本类中的函数 Create、Update、destroy 直接作为脚本组件的成员呢，而非要使用三个函数调用脚本类中的这三个函数?

个人理解:

1. 首先，如果是在脚本组件中定义了三个成员函数，分别用于传入脚本类成员函数，这样在初始化的时候调用起来就很麻烦，想象一下一个组件就要多调用好几次用于传入的函数。
2. 既然我们创建了一个脚本类，就可以直接通过类名来传递类中的成员函数（这个类中只有公用的成员函数，是一个工具类）。不过我们在 Bind 函数中一次性调用脚本类的成员函数，只调用一次 Bind 便可以自动处理脚本类中的所有函数。

### 》》为什么 Instance 需要是指针类型?

```
struct NativeScriptComponent
{
    ScriptableEntity* Instance = nullptr;

    // Set functions
    std::function<void()> InstantiateFunction;
    std::function<void()> DestroyInstanceFunction;

    std::function<void()> OncreateFunction;
    std::function<void()> OnDestroyFunction;
    std::function<void(Timestep)> OnUpdateFunction;

    // Send functions so we can use them later
    template<typename T>
    void Bind()
    {
        InstantiateFunction = [&]() { Instance = new T(); };
        DestroyInstanceFunction = [&]() { delete Instance; Instance = nullptr; };

        OncreateFunction = [] (ScriptableEntity* instance) {};
        OnDestroyFunction = [] (ScriptableEntity* instance) {};
        OnUpdateFunction = [] (ScriptableEntity* instance) {};
```

首先，我们需要接受一个脚本类句柄（也就是一个脚本类对象）来设置脚本组件中的函数，所以我们需要一个 ScriptableEntity 对象。可是这个对象 Instance 为什么需要是指针类型呢？

1. 因为我们初始化的时候将其指定为 nullptr（空指针），Instance 不是指针类型的话这样的定义就是错误的。
2. 我们使用 new、delete 进行生命周期的管理，这要求 Instance 是指针变量。

### 》》向下转换的概念及用例

向下转换：指的是将父类的指针或引用转换为子类的指针或引用。

这样做可以方便的操作子类的成员函数。比如此处，T 是 ScriptableEntity 的子类，当 ScriptableEntity\* 类型的变量 instance 被传入的时候，instance->只能调用 ScriptableEntity 的成员函数 GetComponent()，如果将 ScriptableEntity\* 类型的变量 instance 向下转换为子类 T\* 类型的变量，instance->就能调用 T 类的成员函数 OnCreate()。



```

OnCreateFunction = [](ScriptableEntity* instance) {((T*)instance)->OnCreate();};
OnDestroyFunction = [](ScriptableEntity* instance) {};
OnUpdateFunction = [](ScriptableEntity* instance) {};

```

》》注意:

((T\*)instance) 和 (T\*)instance 是有区别的:

1. ((T\*)instance)->OnCreate(): 将 instance 强制转换为 T\* 类型, 然后通过 -> 运算符调用 T 类型的 OnCreate() 成员函数。
2. (T\*)instance->OnCreate(): 受运算符优先级影响, 这行代码首先对 instance 使用 -> 运算符, 然后将结果转换为 T\* 类型, 这并不能调用到 T 类型的成员函数。

》》为什么有的函数需要 [&] 捕获外部变量, 有的函数则是 [] 不捕获外部变量?

```

struct NativeScriptComponent
{
    ScriptableEntity* Instance = nullptr;

    // Set functions
    std::function<void()> InstantiateFunction;
    std::function<void()> DestroyInstanceFunction;

    std::function<void()> OnCreateFunction;
    std::function<void()> OnDestroyFunction;
    std::function<void(Timestep)> OnUpdateFunction;

    // Send functions so we can use them later
    template<typename T>
    void Bind()
    {
        InstantiateFunction = [&]() { Instance = new T(); };
        DestroyInstanceFunction = [&]() { delete Instance; Instance = nullptr; }; // Why need to define Instance?
        OnCreateFunction = [](ScriptableEntity* instance) {((T*)instance)->OnCreate();};
        OnDestroyFunction = [](ScriptableEntity* instance) {((T*)instance)->OnDestroy();};
        OnUpdateFunction = [](ScriptableEntity* instance, Timestep ts) {((T*)instance)->OnUpdate(ts, Instance);};
    }
};

```

对于 Instantiate 和 DestroyInstance, 他们需要访问 Lambda 表达式作用域之外的 Instance 成员变量, 所以使用 [&] 捕获 Lambda 外部变量 (通常是 this 指针), 并将其作为引用。对于 OnCreate ..., 这三个函数只需要使用传递给他们的参数, 没有访问或修改外部变量, 因此捕获列表可以为空。

》》》enTT 中的 view 有一个成员函数 each, 这个函数是什么, 有什么用, 如何使用?

**概述:** 在 enTT 中, view 用于返回具有特定组件的所有实体, 而 each 函数可以为 view 获取的所有实体执行用户指定的操作。这个操作通常是通过回调函数 (比如 lambda 表达式) 来实现的。

**功能:**

each 函数会遍历视图中的所有实体和它们的组件, 并对每个实体及其组件执行指定的操作。这个操作是通过传入的回调函数实现的。

**参数:**

each 函数接受一个回调函数作为参数。回调函数通常是一个 lambda 表达式, 表达式中指定两个参数: 实体 ID 和对应的组件引用。

》》》》位移矩阵的[3][0], [3][1], [3][2] 是 x,y,z

```

void OnUpdate(Timestep ts)
{
    auto& transform = GetComponent<TransformComponent>().Transform;
    float speed = 5.0f;

    if (Input::IsKeyPressed(NUT_KEY_A))
        transform[3][0] += speed * ts;
    if (Input::IsKeyPressed(NUT_KEY_D))
        transform[3][0] -= speed * ts;
    if (Input::IsKeyPressed(NUT_KEY_W))
        transform[3][1] += speed * ts;
    if (Input::IsKeyPressed(NUT_KEY_S))
        transform[3][1] -= speed * ts;
}

```

$$\begin{bmatrix}
 1 & 0 & 0 & T_x \\
 0 & 1 & 0 & T_y \\
 0 & 0 & 1 & T_z \\
 0 & 0 & 0 & 1
 \end{bmatrix}$$

-----Native Scripting ( with virtual function) -----

》》》》What is the V-Table?

概念:

在 C++ 中，虚函数表（V-table）是支持多态性的一个机制。当类中包含虚函数时，编译器会为该类生成一个虚函数表。

结构:

虚函数表是一个指针数组，每个元素指向某个类中虚函数的具体实现。

工作原理:

虚函数表的生成:	每个包含虚函数的类都有一个虚函数表。虚函数表的元素按照虚函数在类中声明的顺序排列，每个元素指向虚函数的实际实现。
虚表指针（vptr）:	每个对象实例中包含一个指向其虚函数表的指针（vptr）。这个指针在对象创建时被初始化为指向相应类的虚函数表。
函数调用:	当通过基类指针或引用调用虚函数时，程序会通过 vptr 查找虚函数表，并调用正确的实现。这允许在运行时动态绑定到正确的函数实现，从而实现多态性。

示例:

如果两个不同的子类都继承了相同的父类，并对父类中的虚函数进行了不同的定义，这时虚函数表就为不同子类的变量调用相应的虚函数，以防程序运行错误。