

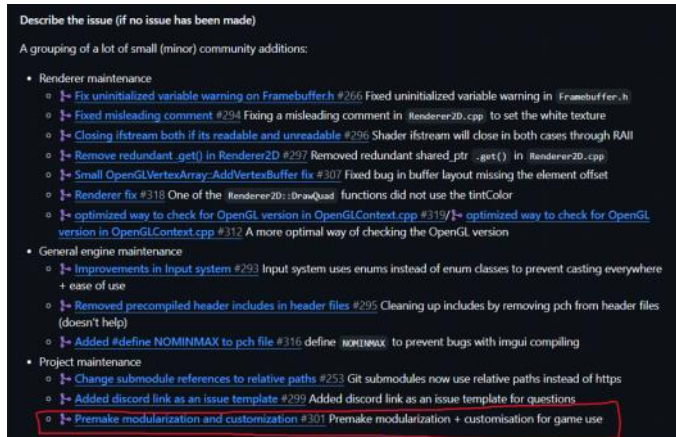
-----Saving & Loading scene-----

》》》更改 Premake 文件构架

这一集中 Chernov 对 premake 文件进行了操作，不过此时 Premake 文件的构架发生了改变（现在每个项目的 premake 被放置在项目的文件夹下，而不是集中放置在 Nut 根目录下的 Premake 文件中），这是因为之前的一次 pull request。

本来准备先完善引擎 UI，后面集中对引擎进行维护，现在看来就先提交一下这个更改吧。

具体可以参考：（<https://github.com/TheCherno/Hazel/pull/320>）



》》》一个问题：关于 premake 文件中的命名

当我将 `yaml-cpp` 作为键（Key），并以此来索引存储的值（Value），此时会出现一个错误：

Error: [string "return IncludeDir.yaml-cpp"]:1: attempt to perform arithmetic on a nil value (field 'yaml') in token: IncludeDir.yaml-cpp

（错误：[string "return IncludeDir.yaml-cpp"] : 1: 尝试对令牌中的零值（字段“yaml”）执行算术运算：IncludeDir.yaml-cpp）

编译器似乎将 `'-'` 识别为算术运算符，而不是文本符号，这导致他尝试进行算术运算操作。

```
IncludeDir = {}
IncludeDir["GLFW"] = "%{wks.location}/Nut/vendor/GLFW/include"
IncludeDir["Glad"] = "%{wks.location}/Nut/vendor/Glad/include"
IncludeDir["ImGui"] = "%{wks.location}/Nut/vendor/imgui"
IncludeDir["glm"] = "%{wks.location}/Nut/vendor/glm"
IncludeDir["stb_image"] = "%{wks.location}/Nut/vendor/stb_image"
IncludeDir["entt"] = "%{wks.location}/Nut/vendor/Entt/include"
IncludeDir["yaml-cpp"] = "%{wks.location}/Nut/vendor/yaml-cpp/include"
```

```
includedirs
{
    "src",
    "vendor/spdlog/include",
    "%{IncludeDir.GLFW}",
    "%{IncludeDir.Glad}",
    "%{IncludeDir.ImGui}",
    "%{IncludeDir.glm}",
    "%{IncludeDir.stb_image}",
    "%{IncludeDir.entt}",
    "%{IncludeDir.yaml-cpp}"
}
```

但是当我将 `'-'` 更改为 `'_'` 时，这样的问题便消失了。

```
IncludeDir = {}
IncludeDir["GLFW"] = "%{wks.location}/Nut/vendor/GLFW/include"
IncludeDir["Glad"] = "%{wks.location}/Nut/vendor/Glad/include"
IncludeDir["ImGui"] = "%{wks.location}/Nut/vendor/imgui"
IncludeDir["glm"] = "%{wks.location}/Nut/vendor/glm"
IncludeDir["stb_image"] = "%{wks.location}/Nut/vendor/stb_image"
IncludeDir["entt"] = "%{wks.location}/Nut/vendor/Entt/include"
IncludeDir["yaml_cpp"] = "%{wks.location}/Nut/vendor/yaml-cpp/include"
```

```

includedirs
{
    "src",
    "vendor/spdlog/include",
    "%{IncludeDir.GLFW}",
    "%{IncludeDir.glad}",
    "%{IncludeDir.ImGui}",
    "%{IncludeDir.glm}",
    "%{IncludeDir.stb_image}",
    "%{IncludeDir.entt}",
    "%{IncludeDir.yaml_cpp}"
}

```

》》》关于最新的YAML 导致链接错误的解决方案

编译器疑似在以动态库的方式尝试运行yaml-cpp 库，并发出了很多警告

```

C:\yaml-cpp\include\yaml-cpp\parser.h(95,28): warning C4251: "YAML::Parser::m_scanner" : class "std::unique_ptr<YAML::Scanner,std::default_delete<YAML::Scanner>>" 需要有 dll 接口由 class "YAML::Parser" 的客户端使用 (编译源文件 src\WutScene\SceneSerializer.cpp)
C:\yaml-cpp\include\yaml-cpp\parser.h(96,31): warning C4251: "YAML::Parser::m_directives" : class "std::unique_ptr<YAML::Directives,std::default_delete<YAML::Directives>>" 需要有 dll 接口由 class "YAML::Parser" 的客户端使用 (编译源文件 src\WutScene\SceneSerializer.cpp)
C:\yaml-cpp\include\yaml-cpp\parser.h(96): message : 参见 "std::unique_ptr<YAML::Directives,std::default_delete<YAML::Directives>>" 的声明 (编译源文件 src\WutScene\SceneSerializer.cpp)
C:\yaml-cpp\include\yaml-cpp\binary.h(65,30): warning C4251: "YAML::Binary::m_data" : class "std::vector<unsigned char,std::allocator<unsigned char>>" 需要有 dll 接口由 class "YAML::Binary" 的客户端使用 (编译源文件 src\WutScene\SceneSerializer.cpp)
C:\yaml-cpp\include\yaml-cpp\binary.h(18): message : 参见 "std::vector<unsigned char,std::allocator<unsigned char>>" 的声明 (编译源文件 src\WutScene\SceneSerializer.cpp)
C:\yaml-cpp\include\yaml-cpp\ostream_wrapper.h(49,29): warning C4251: "YAML::ostream_wrapper::m_buffer" : class "std::vector<char,std::allocator<char>>" 需要有 dll 接口由 class "YAML::ostream_wrapper" 的客户端使用 (编译源文件 src\WutScene\SceneSerializer.cpp)
C:\yaml-cpp\include\yaml-cpp\ostream_wrapper.h(49): message : 参见 "std::vector<char,std::allocator<char>>" 的声明 (编译源文件 src\WutScene\SceneSerializer.cpp)
C:\yaml-cpp\include\yaml-cpp\emitter.h(132,33): warning C4251: "YAML::Emitter::m_state" : class "std::unique_ptr<YAML::EmitterState,std::default_delete<YAML::EmitterState>>" 需要有 dll 接口由 class "YAML::Emitter" 的客户端使用 (编译源文件 src\WutScene\SceneSerializer.cpp)
C:\yaml-cpp\include\yaml-cpp\emitter.h(132): message : 参见 "std::unique_ptr<YAML::EmitterState,std::default_delete<YAML::EmitterState>>" 的声明 (编译源文件 src\WutScene\SceneSerializer.cpp)
C:\yaml-cpp\include\yaml-cpp\exceptions.h(155,58): warning C4275: 非 dll 接口 class "std::runtime_error" 用作 dll 接口 class "YAML::Exception" 的基 (编译源文件 src\WutScene\SceneSerializer.cpp)
C:\Microsoft Visual Studio\2019\Community\VC\Tools\MSVC\14.29.30133\include\stdexcept(101): message : 参见 "std::runtime_error" 的声明 (编译源文件 src\WutScene\SceneSerializer.cpp)
C:\yaml-cpp\include\yaml-cpp\exceptions.h(155): message : 参见 "YAML::Exception" 的声明 (编译源文件 src\WutScene\SceneSerializer.cpp)
C:\yaml-cpp\include\yaml-cpp\exceptions.h(164,15): warning C4251: "YAML::Exception::msg" : class "std::basic_string<char,std::char_traits<char>,std::allocator<char>>" 需要有 dll 接口由 class "YAML::Exception" 的客户端使用 (编译源文件 src\WutScene\SceneSerializer.cpp)
C:\Microsoft Visual Studio\2019\Community\VC\Tools\MSVC\14.29.30133\include\string(4871): message : 参见 "std::basic_string<char,std::char_traits<char>,std::allocator<char>>" 的声明 (编译源文件 src\WutScene\SceneSerializer.cpp)
C:\yaml-cpp\include\yaml-cpp\node.h(135,15): warning C4251: "YAML::Node::m_invalidKey" : class "std::basic_string<char,std::char_traits<char>,std::allocator<char>>" 需要有 dll 接口由 class "YAML::Node" 的客户端使用 (编译源文件 src\WutScene\SceneSerializer.cpp)
C:\Microsoft Visual Studio\2019\Community\VC\Tools\MSVC\14.29.30133\include\string(4871): message : 参见 "std::basic_string<char,std::char_traits<char>,std::allocator<char>>" 的声明 (编译源文件 src\WutScene\SceneSerializer.cpp)
C:\yaml-cpp\include\yaml-cpp\node.h(136,40): warning C4251: "YAML::Node::m_memory" : class "std::shared_ptr<YAML::detail::memory_holder>" 需要有 dll 接口由 class "YAML::Node" 的客户端使用 (编译源文件 src\WutScene\SceneSerializer.cpp)

```

初步解决方案:

@mjthebest7294 2年前
I had to define "YAML_CPP_STATIC_DEFINE" in the premake for the newer version of YAML, otherwise it will try to compile as a DLL
我必须在新版本 YAML 的预编译中定义 "YAML_CPP_STATIC_DEFINE", 否则它将尝试编译为 DLL

12 回复

6 条回复

@p3rk4n27 2年前
It now build yaml project but cant link it to engine... there are errors like unresolved external dllimport...
它现在构建 yaml 项目, 但无法将其链接到引擎... 存在诸如未解析的外部 dllimport 之类的错误...

2 回复

@mjthebest7294 2年前
@p3rk4n27 maybe the engine compiles as a .dll instead of a static .lib
@p3rk4n27 也许引擎编译为 .dll 而不是静态 .lib

》》 AND..

@rio9415 1年前
With the new version of yaml-cpp, you need to change staticruntime to "on" in premake file of yaml-cpp project
使用新版本的yaml-cpp, 需要在yaml-cpp项目的premake文件中将staticruntime更改为"on"

首先我已经在yaml-cpp的premake文件中声明了 "YAML_CPP_STATIC_DEFINE", 并且打开了staticruntime, 但我发现没有作用。

```

defines
{
    "YAML_CPP_STATIC_DEFINE"
}

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

```

接着解决:

问题是, 你还需要在你所使用项目的premake文件中再次声明 "YAML_CPP_STATIC_DEFINE"

```
project "Nut-Editor"
objdir = ("%wks.location)/bin-int/" .. outputdir .. "/" .. prj.name)

files
{
    "src/**/*.h",
    "src/**/*.cpp"
}

defines
{
    "YAML_CPP_STATIC_DEFINE"
}

includedirs
{
    "%{wks.location)/Nut/vendor/spdlog/include",
    "%{wks.location)/Nut/src",
    "%{wks.location)/Nut/vendor",
    "%{IncludeDir.glm}",
    "%{IncludeDir.entt}",
    "%{IncludeDir.yaml_cpp}"
}
```

@kingofspades9720 2周前

If you are having issues using YAML, one fix might be adding `#define YAML_CPP_STATIC_DEFINE` inside of the Hazel premake file not just inside of the yaml-cpp premake file.

如果您在使用 YAML 时遇到问题，一种解决方法可能是在 Hazel 预置文件内添加 `#define YAML_CPP_STATIC_DEFINE`，而不仅仅是在 yaml-cpp 预置文件内。

👍 2 🗨 回复

@yu_a_v14427 11个月前

for new version of yaml-cpp just add a `#define YAML_CPP_STATIC_DEFINE` before including `<yaml-cpp/yaml.h>` in any file.


and turn on `staticruntime` in premakefile of yaml-cpp

对于新版本的 yaml-cpp，只需在任何文件中包含 `<yaml-cpp/yaml.h>` 之前添加 `#define YAML_CPP_STATIC_DEFINE`，

并在 yaml-cpp 的 premakefile 中打开 `staticruntime`

👍 6 🗨 回复

总结：



1分钟前

As of October 2024, you can correctly run yaml-cpp with the following requirements:

1. Define `YAML_CPP_STATIC_DEFINE` in the premake file of yaml-cpp, and define `YAML_CPP_STATIC_DEFINE` in the project configuration file that uses yaml-cpp (such as premake)
2. Define `staticruntime` "on" in the yaml-cpp premake file

》》》什么是 .editorconfig 文件？有什么作用？

问题引入：在深入研究这次提交时，一个以 .editorconfig 署名的文件映入眼帘，这是什么文件？

文件介绍：

EditorConfig helps maintain consistent coding styles for multiple developers working on the same project across various editors and IDEs. The EditorConfig project consists of a file format for defining coding styles and a collection of text editor plugins that enable editors to read the file format and adhere to defined styles. EditorConfig files are easily readable and they work nicely with version control systems.

来自 <<https://editorconfig.org/>>

翻译：

EditorConfig 可帮助多个开发人员在不同的编辑器或 IDE 上维护同一个项目的编码风格，使其保持一致。EditorConfig 项目包含一个用于定义编码风格的文件格式和一组文本编辑器插件，这些插件可让编辑器读取文件格式并遵循定义的风格。EditorConfig 文件易于阅读，并且可与版本控制系统完美配合。

作用：

通过使用 EditorConfig 文件，团队中的每个成员可以确保他们的代码遵循相同的格式，降低因代码风格不一致而引起的问题。许多现代代码编辑器和 IDE（如 Visual Studio Code、Atom、JetBrains 系列等）都支持 EditorConfig，可以自动读取这些规则并应用到打开的文件中。

使用规范：

文件名：	文件名为 .editorconfig，通常放在项目根目录。
------	-------------------------------

键值对格式：	使用 key = value 的形式定义规则，每条规则占一行。 空行和以 # 开始的行会被视为注释。
范围选择器：	使用 [*] 表示应用于所有文件，也可以使用其他模式如 *.js 或 *.py 来指定特定文件类型。
支持的属性：（支持的键值对）	常用属性包括： root：指示是否为顶层文件。 end_of_line：指定行结束符（如 lf, crlf, cr）。 insert_final_newline：是否在文件末尾插入换行符。 indent_style：设置缩进样式（如 tab 或 space）。 indent_size：指定缩进的大小，可以是数字或 tab。 charset：文件字符集（如 utf-8, latin1 等）。 trim_trailing_whitespace：是否修剪行尾空白。

详情参考文档：（<https://spec.editorconfig.org/>）

Table of Contents
<ul style="list-style-type: none"> • EditorConfig Specification <ul style="list-style-type: none"> ◦ Introduction (informative) ◦ Terminology ◦ File Format <ul style="list-style-type: none"> ▪ No inline comments ▪ Parts of an EditorConfig file ◦ Glob Expressions ◦ File Processing ◦ Supported Pairs

代码理解:

```

 8  .editorconfig
...  @@ -0,0 +1,8 @@
...
1  + # top-most EditorConfig file
2  + root = true
3  +
4  + # Unix-style newlines with a newline ending every file
5  + [*]
6  + end_of_line = lf
7  + insert_final_newline = true
8  + indent_style = tab

```

root = true:	指示这是一个顶层的 EditorConfig 文件，编辑器在找到此文件后不会再向上查找其他 EditorConfig 文件。
[*]:	表示应用于所有文件类型的规则。
end_of_line = lf:	指定行结束符为 Unix 风格的换行符（LF，Line Feed）。这通常在类 Unix 系统（如 Linux 和 macOS）中使用。
insert_final_newline = true:	指定在每个文件的末尾插入一个换行符。这是一种良好的编码习惯，许多项目标准要求这样做。
indent_style = tab:	指定缩进样式为制表符（tab），而不是空格。这会影响代码的缩进方式。

《《《《拓展：什么是 Hard tabs? 什么是 Soft tabs?

Hard Tabs	是使用制表符进行缩进，具有灵活性但可能导致跨环境的不一致。
Soft Tabs	是使用空格进行缩进，保证了一致性但文件体积可能更大。

选择使用哪种方式通常取决于团队的编码标准或个人偏好。

》》》》Y A M L U know what I'm saying

》》》》YAML YAML YAML

》》》》关于这次 premake 构架的维护，我只上传了一部分，剩下的留到之后维护时再做。现在我去了解一下 YAML。

》》》》YAML, What is yaml ? What we can do by yaml ?

介绍:

<p>YAML is a human-readable data serialization language that is often used for writing configuration files. Depending on whom you ask, YAML stands for yet another markup language or YAML ain't markup language (a recursive acronym), which emphasizes that YAML is for data, not documents.</p> <p>来自 <https://www.redhat.com/en/topics/automation/what-is-yaml></p>	<p>YAML 是一种人类可读的数据序列化语言，通常用于编写配置文件。根据使用的对象，YAML 可以代表另一种标记语言或者说 YAML 根本不是标记语言（递归缩写），这强调了 YAML 用于数据，而不是文档。</p>
--	--

理解:

在程序中，我们可以使用 yaml 对文件进行两种操作：序列化和反序列化（Serialize & Deserialize）。

序列化意味着我们可以将复杂的数据转变为字节流，进而可以将其轻易保存到文件或数据库中。

反序列化则意味着我们可以对已经序列化的数据进行逆处理，进而将数据转换回原始的数据结构或对象状态。

基础:

基本结构

映射（Map）：键值对的集合。	key: value
序列（Sequence）：有序的元素列表。	<ul style="list-style-type: none"> - item1 - item2 - item3

2. 嵌套结构

YAML 支持嵌套映射和序列，可以组合使用：	person: name: John Doe age: 30 hobbies: - reading - cycling
------------------------	--

3. 数据类型

YAML 支持多种数据类型，包括：字符串，数字，布尔值，Null 值。	例如： string: "Hello, World!" number: 42 boolean: true null_value: null
-------------------------------------	---

》》》yaml-cpp 的使用（详情请阅览：<https://github.com/ibeder/yaml-cpp/blob/master/docs/Tutorial.md>）

在 C++ 中使用 yaml-cpp 库，可以方便地处理 YAML 数据的读取和写入。（以下是读取 Yaml 文件和写入 Yaml 文件的示例）

读取 YAML	<pre>#include <iostream> #include <yaml-cpp/yaml.h> int main() { YAML::Node config = YAML::LoadFile("config.yaml"); std::string name = config["person"]["name"].as<std::string>(); std::cout << "Name: " << name << std::endl; return 0; }</pre>
写入 YAML: 使用 YAML::Emitter 可以生成 YAML 文件	<pre>#include <iostream> #include <yaml-cpp/yaml.h> int main() { YAML::Emitter out; out << YAML::BeginMap; out << YAML::Key << "name" << YAML::Value << "John Doe"; out << YAML::Key << "age" << YAML::Value << 30; out << YAML::EndMap; std::cout << out.str() << std::endl; // 输出生成的 YAML return 0; }</pre>

YAML::Node

定义：YAML::Node 是 YAML-CPP 中的一个核心类，表示 YAML 文档中的一个节点。一个节点可以是标量（单个值）、序列（列表）或映射（键值对）。通过 YAML::Node，你可以以编程方式访问和操作 YAML 数据结构。	创建和使用 YAML::Node： Eg. <pre>#include <yaml-cpp/yaml.h> YAML::Node node = YAML::Load("key: value"); std::string value = node["key"].as<std::string>();</pre>
--	--

Sequences 和 Maps

Sequences （序列）是一个有序列表，表示一组无命名的值。它们在 YAML 中用短横线表示：	fruits: - Apple - Banana - Cherry
在 YAML-CPP 中，你可以这样处理序列：	Eg. <pre>YAML::Node sequence = YAML::Load("[Apple, Banana, Cherry]"); for (const auto& item : sequence) { std::cout << item.as<std::string>() << std::endl; }</pre>
Maps （映射）是一组键值对，表示命名的值。它们在 YAML 中用冒号分隔表示：	person: name: John Doe age: 30
在 YAML-CPP 中，你可以这样处理映射：	Eg. <pre>YAML::Node map = YAML::Load("name: John Doe\nage: 30"); std::string name = map["name"].as<std::string>(); int age = map["age"].as<int>();</pre>

Sequences 和 Maps 的不同之处

序列和映射都是 YAML::Node 的一种。你可以在一个映射中嵌套序列，反之亦然。

不同之处：

序列：	没有键，每个项都有顺序。
映射：	每个项都有唯一的键，顺序不重要。

Converting To/From Native Data Types

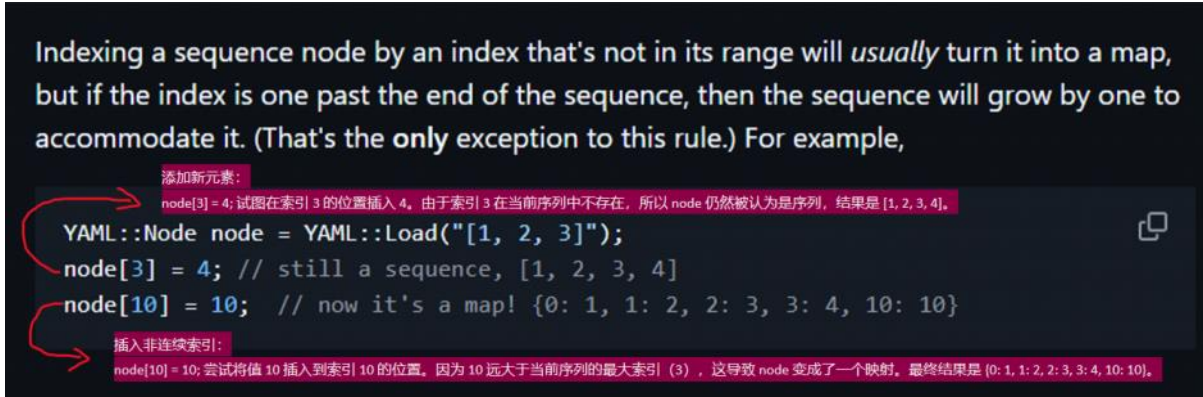
YAML-CPP 提供了方便的方法将 YAML::Node 转换为 C++ 的原生数据类型。你可以使用 as<T>() 方法进行转换。

示例：从 YAML::Node 转换到原生数据类型	YAML::Node node = YAML::Load("name: John Doe\nage: 30"); std::string name = node["name"].as<std::string>(); int age = node["age"].as<int>();
示例：从原生数据类型转换到 YAML::Node	YAML::Node newNode; newNode["name"] = "Jane Doe"; newNode["age"] = 28; // 序列 YAML::Node fruits; fruits.push_back("Apple"); fruits.push_back("Banana"); newNode["fruits"] = fruits; // 输出为 YAML 格式 std::cout << newNode << std::endl;

》》由此引出两个疑惑：

问题一：

查阅文档时，我发现当插入的索引超出当前序列的范围时，YAML-CPP 会将节点视为映射，而不是继续保持序列



结论：动态类型：YAML::Node 的类型是动态的，可以在运行时根据操作的不同而变化。当你使用整数索引时，它保持序列。当你使用非连续的索引或字符串键时，它会转变为映射。

问题二：如何为Node添加一个映射？

在 `YAML::Node node = YAML::Load("[1, 2, 3]");` 的情况下，使用 `node[1] = 5` 是不合适的。

如果你想让 `node[1]` 表示一个映射，`node[1] = 5` 会将序列中索引为 1 的元素（即第二个元素）设置为整数 5，而不是将其更改为一个映射。

如果你想在当前位置设置一个映射，你可以这样做：	YAML::Node node = YAML::Load("[1, 2, 3]"); node[1] = YAML::Node(YAML::NodeType::Map); // 创建一个新的映射 node[1]["key"] = "value"; // 向映射中添加键值对
结构：	- 1 - key: value - 3
或者：	YAML::Node node = YAML::Load("[1, 2, 3]"); // 将 node[1] 设置为一个新的映射 node[1] = YAML::Node(YAML::NodeType::Map); // 设置键为原来的值 2，并赋值为 5 node[1][2] = 5; // 这里的 2 是之前 index 1 的值
结构：	- 1 - 2: 5 - 3

注意：

如果你使用了 <code>node["1"] = 5</code> ，由于 "1" 是一个字符串键，而不是数字索引，这将使程序尝试在 node 中以 "1" 为键插入值 5。node 原本是一个序列，但它会因此转变为一个映射。	最终结果会是 { 0: 1, 1: 2, 2: 3, "1": 5}, 其中 "1" 是一个新的字符串键。
---	---

》》》ifstream 和 ofstream 之间的关系

二者定义在 <fstream> 头文件中，管理文件流。

std::ifstream:	用于从文件中读取数据（输入文件流）。
Std::ofstream:	用于向文件中写入数据（输出文件流）。

易混淆: <iostream> 和文件流没有关系, <iostream> 是提供输入或输出流的标准库, 主要包括 std::cin, std::cout, std::cerr 等。

》》 ofstream 的使用: std::ofstream 用于创建和写入文件

```
// Send file-stream
std::ofstream fout(filepath); // Create and open a file from the filepath
fout << out.c_str();          // Writing data
```

》》 ifstream 的使用: std::ifstream 用于读入文件, 进而对读入的文件进行一些处理。

(图例: 逐行读取文件内容到字符串中)

```
std::ifstream file(filepath);
std::string line;
// 逐行读取文件内容
while (std::getline(file, line)) {
    std::cout << line << std::endl;
}
```

或者 (比上述方法更加高效, 迅捷)

```
std::ifstream file(filepath);
std::stringstream fileContent;
fileContent << file.rdbuf();
```

《《《《 什么是 rdbuf();

在 C++ 中, rd 通常是 "read" 的简写, 意味着与读取操作相关的函数。

rdbuf()

释义:	rdbuf() 是 C++ 中的一个成员函数, 可以直接访问流的底层缓冲区。它通常用于与输入输出流 (如 std::ifstream, std::ofstream, std::iostream 等) 交互。
返回类型:	std::streambuf* 返回指向与流关联的 std::streambuf 对象的指针。该指针可以用于直接进行低级别的输入输出操作。
优点: 直接访问缓冲区	rdbuf() 返回一个指向当前流缓冲区的指针 (即 std::streambuf 对象), 允许你直接从流中读取或写入数据。这意味着, 你可以将整个文件的内容一次性读入, 而不需要逐行或逐字符地读取, 从而提高了效率。 当处理大型文件时, 逐行读取会涉及多次 I/O 操作, 这可能导致性能瓶颈。而使用 rdbuf() 可以减少这些 I/O 操作, 因为它一次性读取整个缓冲区的数据。

类似的 rd 开头的函数还有 rdstate()	含义: rdstate() 是一个成员函数, 用于获取流的状态标志。它返回一个整数, 表示流的当前状态, 包括是否已达到文件结束、是否发生了错误等。
--------------------------	--

》》》》 FIEL STRUCTURE U know what I'm saying

》》》》 YAML YAML YAML

》》》》 YAML 文件构架, YAML 文件设置思路

整体构架:

```
Scene(Map){
  Entities(Seq){
    Entity1(Map)..
    ..
    Entity2(Map)..
    ..
    Entity3(Map)..
    ..
  }
}
```

因此我们也可解释 Deserialize() 函数中做出的操作: 从 data(map) 中取出序列 entities(seq), 然后通过 For 循环对序列中的 entity(map) 进行读取, 随后根据读取的数据去复现场景。

需要提醒的是: Map 中的元素不能重复, Seq 中的元素可以重复。

```
// According to data, we reproduce the scene
auto entities = data["Entity"];
if(entities)
{
    for (auto entity : entities)
    {
        // reproduce codes
    }
}

return true;
```

》》》YAML::Emitter out << YAML::Flow;

概念: out << YAML::Flow 是 C++ 中使用 YAML 库 (如 yaml-cpp) 时的一种语法, 它用来设置 YAML 输出的格式为“流式” (flow style)。

详解: YAML::Flow 是一个常量, 指示输出的 YAML 数据应采用流式表示形式。

流式表示形式将集合 (如数组和映射) 以更紧凑的方式表示, 例如使用方括号 [] 表示数组, 使用花括号 {} 表示映射。

使用场景:

当你想要以更紧凑的格式输出数据时, 可以使用流式表示形式。相比于块状表示 (block style), 流式表示在视觉上更简洁, 适用于小型数据结构或在单行内表示数据。

假设你有一个简单的 YAML 数据结构, 如果不使用 YAML::Flow, 输出可能是这样的 (块状表示):	items: - item1 - item2
而如果使用 YAML::Flow, 此时, 输出将是:	items: [item1, item2]

示例代码:

```
#include <yaml-cpp/yaml.h>
#include <iostream>
int main() {
    YAML::Emitter out;

    out << YAML::Flow; // 设置为流式格式
    out << YAML::BeginMap
        << YAML::Key << "items" << YAML::Value
        << YAML::BeginSeq
            << "item1" << "item2"
        << YAML::EndSeq
        << YAML::EndMap;
    std::cout << out.str() << std::endl;
}
```

》》》设计上的理解

1.Serialize

```
if (entity.HasComponent<CameraComponent>())
{
    out << YAML::Key << "CameraComponent";
    out << YAML::BeginMap;

    auto& cc = entity.GetComponent<CameraComponent>();
    auto& camera = cc.Camera;

    out << YAML::Key << "Camera" << YAML::Value;
    out << YAML::BeginMap;{
        out << YAML::Key << "ProjectionType" << YAML::Value << (int)camera.GetProjectionType();
        out << YAML::Key << "PerspectiveFOV" << YAML::Value << camera.GetPerspectiveVerticalFOV();
        out << YAML::Key << "PerspectiveNear" << YAML::Value << camera.GetPerspectiveNearClip();
        out << YAML::Key << "PerspectiveFar" << YAML::Value << camera.GetPerspectiveFarClip();
        out << YAML::Key << "OrthographicSize" << YAML::Value << camera.GetOrthographicSize();
        out << YAML::Key << "OrthographicNear" << YAML::Value << camera.GetOrthographicNearClip();
        out << YAML::Key << "OrthographicFar" << YAML::Value << camera.GetOrthographicFarClip();
    }
    out << YAML::EndMap;

    out << YAML::Key << "Primary" << YAML::Value << cc.Primary;
    out << YAML::Key << "Fixed Aspect Ratio" << YAML::Value << cc.FixedAspectRatio;

    out << YAML::EndMap;
}
```

2.Yaml data


```
Scene: Untitled
Entities:
- Entity: 256257383941
  TagComponent:
    Tag: Clip-Camera
  TransformComponent:
    Translation: [0, 0, 0]
    Rotation: [0, 0, 0]
    Scale: [1, 1, 1]
  CameraComponent:
    Camera:
      ProjectionType: 1
      PerspectiveFOV: 0.785398185
      PerspectiveNear: 0.00999999978
      PerspectiveFar: 1000
      OrthographicSize: 5
      OrthographicNear: -1
      OrthographicFar: 1
    Primary: false
    Fixed Aspect Ratio: false
```

3. Deserialize

```
auto cameraComponent = data["CameraComponent"];
if(cameraComponent)
{
    auto& cameraProps = cameraComponent["Camera"];
    auto& cc = entity.AddComponent<CameraComponent>();

    int projectionType = cameraProps["ProjectionType"].as<int>();
    cc.Camera.SetProjectionType((SceneCamera::ProjectionType)projectionType);
    cc.Camera.SetPerspectiveVerticalFOV(cameraProps["PerspectiveFOV"].as<float>());
    cc.Camera.SetPerspectiveNearClip(cameraProps["PerspectiveNear"].as<float>());
    cc.Camera.SetPerspectiveFarClip(cameraProps["PerspectiveFar"].as<float>());
    cc.Camera.SetOrthographicSize(cameraProps["OrthographicSize"].as<float>());
    cc.Camera.SetOrthographicNearClip(cameraProps["OrthographicNear"].as<float>());
    cc.Camera.SetOrthographicFarClip(cameraProps["OrthographicFar"].as<float>());
    // Unlike Camera, Primary is a separate key-value mapping,
    // while Camera is a map that requires further access.
    cc.Primary = cameraComponent["Primary"].as<bool>();
    cc.FixedAspectRatio = cameraComponent["Fixed Aspect Ratio"].as<bool>();
}
```

》》 There are few issues you need to know:

- 1.字符匹配: 查找value所用的key需要正确无误, 比如你想查找yaml文件中的 Fixed Aspect Ratio, 你就必须用 Fixed Aspect Ratio 作为索引, 而不是 FixedAspectRatio.
- 2.Map访问: 如果你在map中查找其中存储的元素, 你只能访问顶层的元素, 而不能访问靠底层的元素. 比如 CameraComponent

```
CameraComponent: ← map
  Camera: ← element 1 (also a map)
    ProjectionType: 1
    PerspectiveFOV: 0.785398185
    PerspectiveNear: 0.00999999978
    PerspectiveFar: 1000
    OrthographicSize: 5
    OrthographicNear: -1
    OrthographicFar: 1
  Primary: false ← element 2 (key-value)
```

图例此时处于 cameraComponent, 你通过这两个 key 访问其中的 value: Camera 和 Primary (比如调用 cameraComponent["Camera"])。可是如果你想通过 CameraComponent 访问 projectionType, 就不能使用 cameraComponent["ProjectionType"] 这样的语句, 而必须使用 cameraComponent["Camera"]["ProjectionType"], 否则会报错。

》》》 Cherno 将文件保存在 .hazel 后缀的文件中, 这可以吗? 为什么? 只有 Hazel 才能处理这种文件吗? ?

```
+         if (ImGui::MenuItem("Serialize"))
+         {
+             SceneSerializer serializer(m_ActiveScene);
+
+             serializer.Serialize("assets/scenes/Example.hazel");
+         }
+
+         if (ImGui::MenuItem("Deserialize"))
+         {
+             SceneSerializer serializer(m_ActiveScene);
+
+             serializer.Deserialize("assets/scenes/Example.hazel");
+         }
+     }
```

这取决于你的打开方式, 现在 Hazel (或者 Nut) 有能力接受这种文件, 通过我们定义的函数, Hazel (或者 Nut) 通过文件流合理读入文件, 然后识别文件内容并且做出了相应操作。如果你使用 word 打开这种文件, 应用程序就会通过文本格式打开这个文件, 这样也是被允许的, 因为我们就是以文本的形式去设置了 yaml 文件。不过使用其他打开方式可能就会出错。

》》》注意事项/可改进事项

@Kariminordinate 3年前

Is there any reason you've implemented this without reflection? Or is it just simplicity? For my engine I've added simple metaprogramming to allow for reflection, which means I don't have to write serializers for every member variable, or write code to show it in the editor.

您是否有任何理由不加反思就实施了这一点？或者只是简单？对于我的引擎，我添加了简单的元编程以允许反射，这意味着我不必为每个成员变量编写反序列化器，或编写代码以在编辑器中显示它。

👍 25

🗨

🔒

^ 7 条回复

@qx-jd9mh 3年前

@mattmurphy7030 "game devs" are allergic to template metaprogramming

@mattmurphy7030 "game devs"对模板元编程过敏

👍 2

🗨

🔒

@ipotrick6686 3年前

how?

如何？

👍

🗨

🔒

@erniegutierrez410 3年前

He doesn't have a clue

他不知道

👍 1

🗨

🔒

@johnjackson9767 3年前

Yes. Reflection information makes this a breeze.

是的。反射信息使这变得轻而易举。

👍

🗨

🔒

@utkarsh9547 3年前

Is there a place where I can go to learn about this?!

有没有地方可以去学习这方面的知识？！

👍

🗨

🔒

@utkarsh9547 3年前

@johnjackson9767 Is there a place I can learn about this???? I'm fairly annoyed by having to type out the statements for each member variable...

@johnjackson9767 有地方可以了解这个吗？？？我对必须为每个成员变量键入语句感到相当恼火.....

👍

🗨

🔒

@NillKitty 2年前 (修改过)

I have the same question. This seems like needless error prone work -- I'm sure at least 10 times a year you're going to add a member to a class and forget to add it to the serializer, or you add it to the serializer but forget it in the deserializer. Why do it this way when you can just enumerate over the reflected fields in the class and just use their variable/member name as the key? Then you only write code once per data type, not once per member.

我有同样的问题。这似乎是不必要的容易出错的工作——我确信每年至少有 10 次你要向类中添加成员但忘记将其添加到序列化器中，或者你将其添加到序列化器中但忘记了解串器。当您可以在枚举类中的反射字段并使用它们的变量/成员名称作为键时，为什么要这样做呢？然后，您只需为每个数据类型编写一次代码，而不是为每个成员编写一次代码。

AND

@wakeupthesun3 1年前 (修改过)

Another thing to note (I'm not sure if this is covered later) is the scene is being deserialized in inverse order in which the original entities were added to the scene. You can see this by the original scene has the red square on top, covering the green square. When deserialized, the green square is on top. You can either serialize your entities backwards, or deserialize them backwards. I think deserializing backwards is better, because then the serialized file will match the order of your hierarchy panel. To deserialize backwards, you can make a vector of the entity nodes and then get a reverse iterator to that vector.

auto entitiesNode = data["Entities"];

// reverse it to add the entities in the order they were
// originally added
std::vector<YAML::Node> entitiesRev(entitiesNode.begin(), entitiesNode.end());

for (auto it = entitiesRev.rbegin(); it != entitiesRev.rend(); ++it)
{
 s_deserializeEntity(*it, mp_scene.get());
}

Have fun!

另一件需要注意的事情（我不确定稍后是否会介绍这一点）是场景正在以与原始实体添加到场景相反的顺序进行反序列化。您可以通过原始场景看到顶部有红色方块，覆盖了绿色方块。反序列化时，绿色方块位于顶部。您可以向后序列化实体，也可以向后反序列化它们。我认为向后反序列化更好，因为这样序列化的文件将与层次结构面板的顺序匹配。要向后反序列化，您可以创建实体节点的向量，然后获取该向量的反向迭代器：

自动实体节点=数据["实体"];

// 反转它以按实体的顺序添加实体
// 最初添加的
std::vector<YAML::Node>EntityRev(entitiesNode.begin(), 实体节点.end());

for (auto it = EntityRev.rbegin(); it!=entitiesRev.rend(); ++it)
{
 s_deserializeEntity(*it, mp_scene.get());
}

玩得开心！

分区 GameEngine 的第 10 页

》》》什么是 commdlg 库?

```

1  #include <winapifamily.h>
2
3  1.  commdlg.h  -- This module defines the 32-Bit Common Dialog APIs
4  *
5  *  Copyright (c) Microsoft Corporation. All rights reserved.
6  *
7  *
8  *
9  *
10

```

概念: `commdlg.h` 是 Windows API 中的一个头文件，它用于实现标准对话框功能，如文件打开和文件保存对话框。这个头文件定义了与这些对话框相关的结构、常量和函数，使开发者能够方便地在应用程序中集成文件选择功能。

在使用 `commdlg.h` 时，通常会涉及到以下几个函数：

GetOpenFileName:	用于显示“打开文件”对话框。
GetSaveFileName:	用于显示“保存文件”对话框。

》》》glfw3.h 和 glfw3native.h 这两个库之间的不同

不同: `glfw3.h` 和 `glfw3native.h` 之间的区别在于：`glfw3.h` 用于创建 `glfw` 类型的窗口，`glfw3native.h` 用于获取原生操作系统中的窗口的句柄，以此来进行更底层的操作。

<p>1. <GLFW/glfw3.h></p> <p>目的: 这是 GLFW 的主要头文件，提供了创建窗口、处理输入、管理 OpenGL 上下文、以及其他与窗口和输入相关的功能。</p> <p>内容: 包含了 GLFW 的所有核心功能，比如：创建和管理窗口和上下文、处理键盘、鼠标等输入事件、管理 OpenGL 扩展、定时器等功能</p>	<p>2. <GLFW/glfw3native.h></p> <p>目的: 这个头文件提供了平台特定的功能，通常用于访问底层原生窗口句柄或其他系统级别的功能。</p> <p>内容: 包含了一些函数，这些函数允许你获取与操作系统相关的窗口句柄。例如，在 Windows 系统上，你可以通过它获得 <code>HWND</code> 句柄；在 X11 上，你可以获得相应的 X11 窗口 ID。</p> <p>使用场景: 如果你需要与操作系统的原生 API 进行交互（例如，在窗口中嵌入第三方控件，或与其他库集成），则可能需要使用这个头文件。</p>
--	--

》》》关于 glfw3native.h 和 #define GLFW_EXPOSE_NATIVE_WIN32

首先：`glfwGetWin32Window()` 是 `glfw3native.h` 中的函数（以下是其定义）。不过该函数在使用之前，需要通过定义 `GLFW_EXPOSE_NATIVE_WIN32` 将其暴露出来。

```

161  1.  glfwGetWin32Window()  -- Returns the 'HWND' of the specified window.
162  *
163  *  [error](@ref error_handling) occurred.
164  *
165  *  Possible errors include @ref GLFW_NOT_INITIALIZED and @ref
226  *  GLFW_PLATFORM_UNAVAILABLE.
227  *
228  *  @remark The 'HDC' associated with the window can be queried with the
229  *  [GetDC](@ref GetDC) function.
230  *  @code
231  *  HDC dc = GetDC(glfwGetWin32Window(window));
232  *  @endcode
233  *  This DC is private and does not need to be released.
234  *
235  *  @thread_safety This function may be called from any thread. Access is not
236  *  synchronized.
237  *
238  *  @since Added in version 3.0.
239  *
240  *  @ingroup native
241  */
242  GLFWAPI HWND glfwGetWin32Window(GLFWwindow* window);
243  #endif

```

》》》对话框函数设计思路:

```

std::string FileDialogs::OpenFile(const char* filter)
{
    OPENFILENAMEA ofn;
    CHAR szFile[260] = { 0 };
    ZeroMemory(&ofn, sizeof(OPENFILENAME));
    ofn.lStructSize = sizeof(OPENFILENAME);
    ofn.hwndOwner = glfwGetWin32Window((GLFWwindow*)Application::Get().GetWindow().GetNativeWindow());
    ofn.lpstrFile = szFile;
    ofn.nMaxFile = sizeof(szFile);
    ofn.lpstrFilter = filter;
    ofn.nFilterIndex = 1;
    ofn.Flags = OFN_PATHMUSTEXIST | OFN_FILEMUSTEXIST | OFN_NOCHANGEDIR;
    if (GetOpenFileNameA(&ofn) == TRUE)
    {
        return ofn.lpstrFile;
    }
    return std::string();
}

```

这个函数 FileDialogs::OpenFile 的作用是打开一个文件对话框，让用户选择一个文件，并返回所选文件的路径。

下面逐句解释代码的功能：（标蓝的函数/类型名/变量有额外笔记）

std::string FileDialogs::OpenFile(const char* filter)	定义一个名为 OpenFile 的静态成员函数，接受一个字符串参数 filter，该参数用于指定文件类型过滤器（例如仅显示文本文件或图像文件 / 或者 Cherno 填入的： "Hazel Scene (*.hazel)\0*.hazel\0" ）
{	创建一个 OPENFILENAMEA 结构体实例 ofn，该结构体用于存储文件对话框的各种信息。
OPENFILENAMEA ofn;	
CHAR szFile[260] = { 0 };	声明一个字符串组 szFile，用于存储用户选择的文件路径，大小为 260 字节（这是 Windows 系统中路径的最大长度限制）。
ZeroMemory(&ofn, sizeof(OPENFILENAME));	将 ofn 结构体的内存清零，以确保它的所有字段都被初始化为零。
ofn.lStructSize = sizeof(OPENFILENAME);	设置 ofn 结构体的大小，以便 Windows 知道使用哪个版本的结构体
ofn.hwndOwner = glfwGetWin32Window((GLFWwindow*) Application::Get().GetWindow().GetNativeWindow());	获取当前窗口的句柄并赋值给 ofn.hwndOwner，这样文件对话框会相对于这个窗口显示。
ofn.lpstrFile = szFile;	将 szFile 的地址赋值给 ofn.lpstrFile，以便在用户选择文件后可以将文件路径写入这个数组中。
ofn.nMaxFile = sizeof(szFile);	设置 ofn.nMaxFile 为 szFile 的大小，以告诉对话框可以存储的最大路径长度。
ofn.lpstrFilter = filter;	设置 ofn.lpstrFilter 为传入的 filter 参数，以定义可见的文件类型（例如 ".txt.cpp"）。
ofn.nFilterIndex = 1;	设定过滤器的索引，通常设为 1 表示使用第一个过滤器。
ofn.Flags = OFN_PATHMUSTEXIST OFN_FILEMUSTEXIST OFN_NOCHANGEDIR;	设置对话框的标志： <ul style="list-style-type: none"> • OFN_PATHMUSTEXIST：用户选择的路径必须存在。 • OFN_FILEMUSTEXIST：用户选择的文件必须存在。 • OFN_NOCHANGEDIR：打开对话框时，不更改当前工作目录。
if (GetOpenFileNameA(&ofn) == TRUE)	调用 Windows API 函数 GetOpenFileNameA 显示文件对话框。如果用户选择了一个文件，返回值为 TRUE。
{ return ofn.lpstrFile; }	如果文件选择成功，返回 ofn.lpstrFile 中存储的文件路径。
return std::string(); }	如果用户取消了操作或发生错误，返回一个空的字符串。

《《 关于 OPENFILENAMEA 的定义

```

typedef struct tagOFNA {
    DWORD      lStructSize;
    HWND       hwndOwner;
    HINSTANCE   hInstance;
    LPCSTR      lpstrFilter;
    LPSTR       lpstrCustomFilter;
    DWORD       nMaxCustFilter;
    DWORD       nFilterIndex;
    LPSTR       lpstrFile;
    DWORD       nMaxFile;
    LPSTR       lpstrFileName;
    DWORD       nMaxFileName;
    LPCSTR      lpstrInitialDir;
    LPCSTR      lpstrTitle;
    DWORD       Flags;
    WORD        nFileOffset;
    WORD        nFileExtension;
    LPCSTR      lpstrDefExt;
    LPARAM      lCustData;
    LPOFNHOOKPROC lpfnHook;
    LPCSTR      lpTemplateName;
#ifdef _MAC
    LPEDITMENU  lpEditInfo;
    LPCSTR      lpstrPrompt;
#endif
#ifdef (_WIN32_WINNT >= 0x0500)
    void *      pvReserved;
    DWORD       dwReserved;
    DWORD       FlagsEx;
#endif // (_WIN32_WINNT >= 0x0500)
} OPENFILENAMEA, *LPOPENFILENAMEA;

```

《《关于 ZeroMemory 函数的定义

在 minwinbase.h 函数中:

```

39  | #define ZeroMemory RtlZeroMemory
20266 | #define RtlZeroMemory(Destination, Length) memset((Destination), 0, (Length))

```

《《 ofn.lpstrFilter = filter; 之中, filter 为什么是 const char* ? 填入的时候有什么规范?

lpstrFilter 格式

示例:

```
const char *filter = "Hazel Files (*.hazel)\0*.hazel\0All Files (*.*)\0*.*\0\0";
```

格式:

每一组文件类型描述由两部分组成 -> 描述字符串和扩展名字符串:

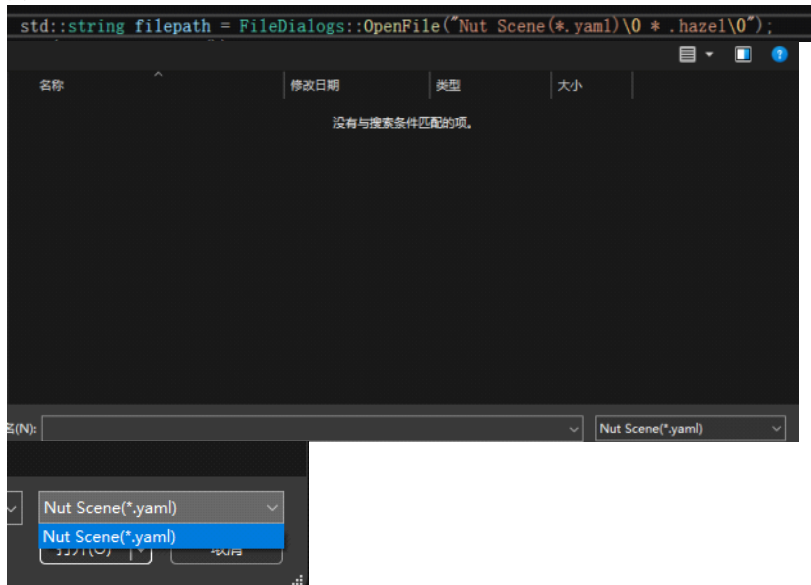
描述字符串是用户在对话框中看到的文件类型名称, 扩展名字符串指定了可以被选择的文件扩展名。各组之间用 \0 分隔, 最后以两个 \0 结束。

对话框如何识别和表示:

Hazel Files (*.hazel)\0*.hazel\0 -> 在文件对话框中, 用户会看到 "Hazel Files (*.hazel)" 作为文件类型的选项。当选择这个选项后, 对话框会过滤出所有以 .hazel 结尾的文件。

All Files (*.*)\0*.*\0 -> 如果用户选择 "所有文件 (.)", 则会显示所有文件, 包括 .hazel 文件。

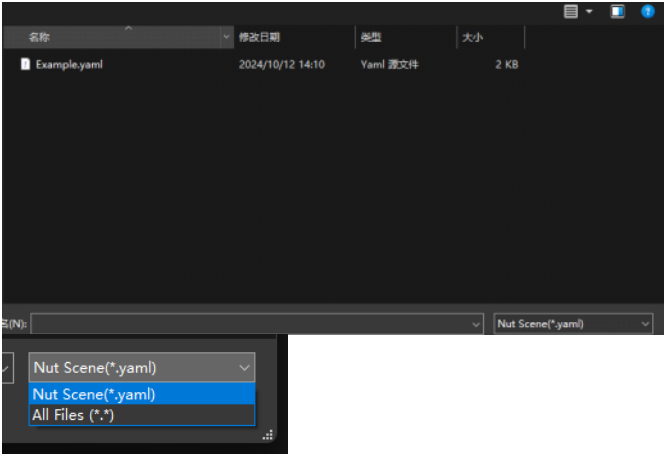
例如:



Nut Scene(*.yaml) 为显示的文本提示, 通常设置为你可以选择的过滤器, 通过文本提示, 代码会索引到合适的过滤器

.hazel 则会根据你选择的文本索引到你设置的过滤器, 然后对所有文件进行过滤, 如果符合.hazel 后缀, 便显示在对话框窗口中。

```
std::string filepath = FileDialogs::OpenFile("Nut Scene(*.yaml)\0 * .yaml\0All Files (*.*)\0*.*\0\0");
```

《《 GetOpenFileNameA 函数的作用：

GetOpenFileNameA 是 Windows API 中的一个函数，用于显示一个标准的“打开文件”对话框，让用户选择一个文件。

函数原型	BOOL GetOpenFileNameA(LPOPENFILENAMEA lpofn);
参数	lpofn: 指向 OPENFILENAMEA 结构体的指针，该结构体包含了对话框的配置信息和用户选择的文件路径。
返回值	如果用户成功选择了一个文件并点击“确定”，函数返回非零值（通常是 TRUE）。 如果用户取消对话框或发生错误，返回值为零（FALSE）。可以通过调用 GetLastError 来获取更多错误信息。

《《 Flags 详细解释：

在 OPENFILENAME 结构体中，可以设置多个标志（Flags）来控制对话框的行为。

OFN_PATHMUSTEXIST:	含义：用户输入的路径必须存在。如果用户在对话框中输入了一个路径（而不是从浏览器中选择），这个路径必须是有效的。 触发情况：当用户输入一个不存在的路径并尝试打开文件时，会出现错误提示，说明路径无效。
OFN_FILEMUSTEXIST:	含义：用户选择的文件必须存在。即使用户在对话框中选择了文件，如果该文件不存在，就会阻止选择。 触发情况：当用户选择的文件实际上在磁盘上不存在时，系统会显示错误提示，说明所选文件不存在。
OFN_NOCHANGEDIR:	含义：打开对话框时不改变当前工作目录。默认情况下，打开文件对话框可能会改变程序的当前工作目录以便于访问文件。 触发情况：这个标志的作用是确保选择文件后，程序的当前工作目录保持不变，即使用户选择了不同的文件路径。

《关于 OFN_NOCHANGEDIR 的详细说明

背景：在 Windows 应用程序中，当前工作目录是指程序在文件系统中默认访问的位置。当应用程序启动时，它会有一个初始的工作目录，通常是可执行文件所在的位置。

使用场景：

当用户打开文件对话框并选择一个文件时，默认情况下，Windows 会将当前工作目录更改为用户选择的文件所在的路径。这意味着如果用户选择了一个不同位置的文件，之后程序的所有文件访问操作都会基于这个新的工作目录。

然而，在某些情况下，这种行为可能会导致问题。例如：

- 依赖于相对路径：如果程序中的文件操作使用相对路径，工作目录的变化可能会导致文件访问失败。
- 多次调用：如果你的应用程序需要频繁打开文件，改变工作目录可能会使管理变得复杂，特别是在需要回到原始路径时。

假设你有一个文本编辑器应用程序，用户可以打开和编辑多个文件。在这个程序中，用户最开始可能在 C:\Documents 中打开一个文件。

```
OPENFILENAME ofn;           // common dialog box structure
char szFile[260];           // buffer for file name

// Initialize OPENFILENAME
ZeroMemory(&ofn, sizeof(ofn));
ofn.lStructSize = sizeof(ofn);
ofn.hwndOwner = hwnd;
ofn.lpstrFile = szFile;
ofn.lpstrFile[0] = '\0';
ofn.nMaxFile = sizeof(szFile);
ofn.lpstrFilter = "Text Files\0*.TXT\0All Files\0*.*\0";
ofn.nFilterIndex = 1;
ofn.lpstrFileTitle = NULL;
ofn.nMaxFileTitle = 0;
ofn.lpstrInitialDir = NULL;
ofn.lpstrTitle = "Open File";
ofn.Flags = OFN_PATHMUSTEXIST | OFN_FILEMUSTEXIST | OFN_NOCHANGEDIR;

// Display the Open dialog box
if (GetOpenFileName(&ofn)) {
    // 用户选择了文件
    // 这里可以进行文件读取等操作
}
```

在此示例中：

如果没有设置 OFN_NOCHANGEDIR，选择一个位于 D:\OtherFiles 的文件可能会把当前工作目录从 C:\Documents 改为 D:\OtherFiles。这意味着接下来如果程序尝试以相对路径访问文件（例如，读取 data.txt），它会在 D:\OtherFiles 查找，而不是在用户原本的路径 C:\Documents。

加入 OFN_NOCHANGEDIR 后的效果：

通过加入 OFN_NOCHANGEDIR，即使用户选择了位于不同文件夹的文件，程序的当前工作目录仍然保持在 C:\Documents。这样，任何依赖于该目录的文件操作都不会受到影响。

》》》问题：触发断点 -> 为什么将 CreateRef 改为 Ref 时会触发断点异常？

```
void EditorLayer::OpenScene()
{
    std::string filepath = FileDialogs::OpenFile("Nut Scene(*.yaml)\0 * .yaml\0All Files (*.*)\0*\0\0");
    if (!filepath.empty())
    {
        m_ActiveScene = CreateRef<Scene>();
        m_ActiveScene->OnViewportResize((uint32_t)m_ViewPortSize.x, (uint32_t)m_ViewPortSize.y); // We use
        m_SceneHierarchyPanel.SetContext(m_ActiveScene); // We use

        SceneSerializer serializer(m_ActiveScene);
        serializer.Deserialize(filepath);
    }
}
```

因为 Ref (std::shared_ptr<T>) 创建了一个智能指针，但未分配内存；
而 CreateRef (std::make_shared<T>()) 创建并初始化一个智能指针，同时分配内存并构造对象。

》》》关于快捷键触发这个事件函数的设计：

```
bool EditorLayer::OnKeyPressed(KeyPressedEvent& event)
{
    // You can triggering this event only once,
    // because if you try to trigger this event again,
    // the event.GetRepeatCount will bigger than 0.
    // And function will return false
    if (event.GetRepeatCount() > 0)
        return false;

    bool ctrl = Input::IsKeyPressed(NUT_KEY_LEFT_CONTROL) || Input::IsKeyPressed(NUT_KEY_RIGHT_CONTROL);
    bool shift = Input::IsKeyPressed(NUT_KEY_LEFT_SHIFT) || Input::IsKeyPressed(NUT_KEY_RIGHT_SHIFT);
    switch (event.GetKeyCode())
    {
        case NUT_KEY_N: {
            if (ctrl)
                NewScene();

            break;
        }
        case NUT_KEY_O: {
            if (ctrl)
                OpenScene();

            break;
        }
        case NUT_KEY_S: {
            if (ctrl && shift)
                SaveSceneAs();

            break;
        }
    }
}
```

1.GetRepeat() 在这里的作用：防止处理重复按键事件。

e.GetRepeatCount() 函数用于获取按键的重复次数。当一个按键被按下并保持时（比如长按），操作系统会不断生成按键按下事件，这样会导致该事件被多次触发。所以检测到重复按键时，函数直接返回 false，意味着后续的代码（如处理快捷键的逻辑）将不会被执行。这可以防止同一个快捷键被多次触发，从而避免造成意外的重复操作。

2.这个函数为什么只在事件重复时返回一个布尔量，其他条件下不返回值呢。为什么 GetRepeat() 需要设置在函数最前面？

在事件触发之后，我们运行实际逻辑代码（OpenScene/NewScene），但是我们不能返回 true，因为 true 不会阻止事件运行，这将会导致事件会被连续触发。其次，如果在逻辑代码之后返回 false，而不是在函数开头返回 false，都会导致事件被触发第二次，因为在判断是否重复触发时依旧会先运行逻辑代码，随后判断出来结果是 false（应该阻塞），不过此时已经没有用了。

```

bool ctrl = Input::IsKeyPressed(NUT_KEY_LEFT_CONTROL) || Input::IsKeyPressed(NUT_KEY_RIGHT_CONTROL);
bool shift = Input::IsKeyPressed(NUT_KEY_LEFT_SHIFT) || Input::IsKeyPressed(NUT_KEY_RIGHT_SHIFT);
switch (event.GetKeyCode())
{
    case NUT_KEY_N: {
        if (ctrl)
            NewScene();

        break;
    }
    case NUT_KEY_O: {
        if (ctrl)
            OpenScene();

        break;
    }
    case NUT_KEY_S: {
        if (ctrl && shift)
            SaveSceneAs();

        break;
    }
}

if (event.GetRepeatCount() > 0)
    return false;
return true;

```

第一次触发事件

不满足, 故返回 true, 不阻塞

```

bool ctrl = Input::IsKeyPressed(NUT_KEY_LEFT_CONTROL) || Input::IsKeyPressed(NUT_KEY_RIGHT_CONTROL);
bool shift = Input::IsKeyPressed(NUT_KEY_LEFT_SHIFT) || Input::IsKeyPressed(NUT_KEY_RIGHT_SHIFT);
switch (event.GetKeyCode())
{
    case NUT_KEY_N: {
        if (ctrl)
            NewScene();

        break;
    }
    case NUT_KEY_O: {
        if (ctrl)
            OpenScene();

        break;
    }
    case NUT_KEY_S: {
        if (ctrl && shift)
            SaveSceneAs();

        break;
    }
}

if (event.GetRepeatCount() > 0)
    return false;
return true;

```

第二次触发事件 (重复触发)

会再次处理

条件满足, 返回 false 表示阻塞 (但此时运行了两次业务)

3. `GetKeyCode` 和 `GetRepeatCount` 中返回的 `keycode` 和 `count` 是在哪里自动获取的, 所以我们才能使用其返回值来进行判断
在之前设置的回调函数中: `WindowsWindow.cpp` 中

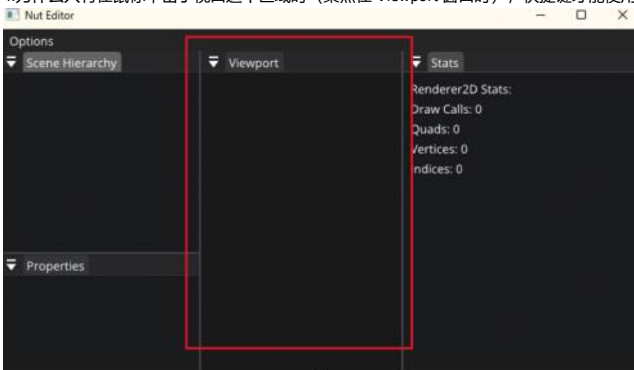
```

glfwSetKeyCallback(m_Window, [](GLFWwindow* window, int key, int scancode, int action, int mods)
{
    WindowData& data = *(WindowData*)glfwGetWindowUserPointer(window);

    switch (action)
    {
        case GLFW_PRESS: { KeyPressedEvent event(key, 0); data.EventCallback(event); break; }
        case GLFW_RELEASE: { KeyReleasedEvent event(key); data.EventCallback(event); break; }
        case GLFW_REPEAT: { KeyPressedEvent event(key, 1); data.EventCallback(event); break; }
    }
});

```

4. 为什么只有在鼠标单击了视口这个区域时 (聚焦在 Viewport 窗口时), 快捷键才能使用?



因为是在 `EditorLayer::OnEvent()` 函数中设置的事件分发。

```

void EditorLayer::OnEvent(Event& event)
{
    m_CameraController.OnEvent(event);

    EventDispatcher dispatcher(event);
    dispatcher.Dispatch<KeyPressedEvent>(NUT_BIND_EVENT_FN(EditorLayer::OnKeyPressed));
}

```

》》》这段 premake 代码什么意思？

```
57
58     filter "files:vendor/ImGuizmo/**/*.cpp"
59     flags {"NoPCH"}
60
```

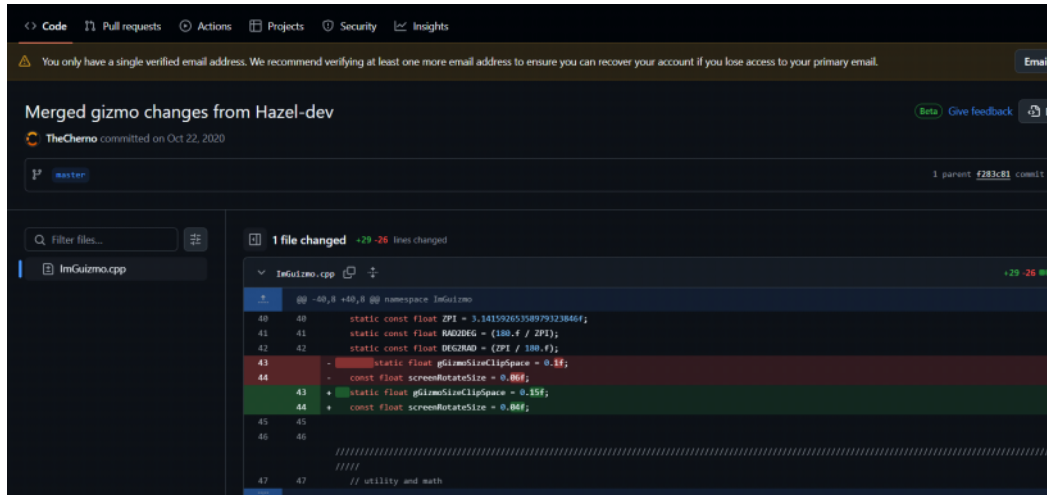
》》》为什么Nut-editor 中没有包含 ImGui 库目录却可以使用 ImGui, 但是没有包含 yaml-cpp 库目录的时候却不能使用 yaml-cpp?

```
includedirs
{
    "%{wks.location}/Nut/vendor/spdlog/include",
    "%{wks.location}/Nut/src",
    "%{wks.location}/Nut/vendor",
    "%{IncludeDir.glm}",
    "%{IncludeDir.entt}",
    "%{IncludeDir.yaml_cpp}",
    "%{IncludeDir.ImGuizmo}"
}
```

因为实际上 ImGui 在 "%{wks.location}/Nut/vendor" 中已经被包含了。

》》》代码设计: gizmo库中对于 ImGuizmo.cpp 的更改: (<https://github.com/TheCherno/ImGuizmo/commit/218d60bde7d22061ac525d0d71e05360b4dcf978>)

我猜想Cherno做的是一些对于 ImGuizmo 样式的更改, 并且由于时间原因, 最新的 ImGuizmo.cpp 结构发生了很多改变, 这让我不容易同步这个更改。所以我决定先保持默认值, 后续再查看是否需要同步此更改。



》》》什么是万向锁, 什么情况下会触发万向锁? 什么是四元数, 为什么四元数可以解决万向锁? glm/gtx/quaternion.hpp 中处理四元数的函数如何使用?

万向锁 (Gimbal Lock)

万向锁是一种在三维空间中旋转时遇到的现象, 通常发生在使用欧拉角表示旋转的情况下。它指的是在某些特定的旋转配置下, 系统的自由度减少, 使得无法进行预期的旋转。

万向锁通常在以下情况下发生:

旋转轴对齐:	当一个旋转轴与另一个旋转轴对齐时, 例如在俯仰角为 $\pm 90^\circ$ 时, 两个旋转轴重合, 这会导致系统失去一个自由度。
极限位置:	当物体达到某个极限位置 (如直立或水平), 就可能会导致无法实现某些方向的旋转。

万向锁的影响:

当发生万向锁时, 物体可能会无法按预期方向旋转, 或者某些旋转会变得不可用。这在动画、飞行控制和机器人运动等应用中会造成问题。

参考文档: (<https://medium.com/@lalesena/euler-angles-rotations-and-gimbal-lock-brief-explanation-de1d4764170>)

参考图片:

https://miro.medium.com/v2/resize:fit:498/format:webp/1*7JT7g5dLeZ-Q5uOYnX8hCw.gif
https://miro.medium.com/v2/resize:fit:400/format:webp/1*OCkgKWmTtmDttrukAX4hSA.gif

四元数 (Quaternion) 是一种扩展了复数的数学结构, 通常用于表示三维空间中的旋转。它由一个实数部分和三个虚数部分组成, 形式为:

$$q = w + xi + yj + zk$$

其中: w 是实数部分, x, y, z 是虚数部分 (向量部分), i, j, k 是虚单位。

四元数通过以下方式解决万向锁问题:


```
// 在前景绘制列表中添加 gizmo 绘制
ImGui::SetDrawList(ImGui::GetForegroundDrawList());
ImGui::SetRect(0, 0, ImGui::GetWindowWidth(), ImGui::GetWindowHeight());
ImGui::Manipulate(viewMatrix, projectionMatrix, ImGui::OPERATION::TRANSLATE, ImGui::MODE::LOCAL, &modelMatrix[0][0]);
```

》》》ImGuizmo::Manipulate()

释义：ImGuizmo::Manipulate 是 ImGuizmo 库中用于处理物体变换的函数。这个函数可以在用户界面中允许用户通过拖动来对物体进行平移、旋转和缩放操作。

函数原型：

```
bool Manipulate(const float* view, const float* projection,
               OPERATION operation, MODE mode,
               float* matrix, float* deltaMatrix,
               const float* snap = nullptr, const float* pivot = nullptr);
```

参数解析：

view:	相机的视图矩阵（cameraView），表示相机的位置和方向。
projection:	相机的投影矩阵（cameraProjection），用于确定场景中物体的透视效果。
operation:	变换操作类型（平移、旋转或缩放），通过 (ImGuizmo::OPERATION)m_GizmoType 指定。
mode:	变换模式，通常是 ImGuizmo::LOCAL（局部变换）或 ImGuizmo::GLOBAL（全局变换）。
matrix:	物体的变换矩阵（transform），表示物体在场景中的位置、旋转和缩放。
deltaMatrix:	可选参数，表示变化的矩阵。
snap:	可选参数，指定对变换进行捕捉的值（例如，步长）。如果为 nullptr，则不进行捕捉。
pivot:	可选参数，指定旋转的中心点。

》》》ImGuizmo::OPERATION 的定义

```
// call it when you want a gizmo
// Needs view and projection matrices.
// matrix parameter is the source matrix (where will be gizmo be drawn) and might be transformed by the function. Return deltaMatrix is optional
// translation is applied in world space
enum OPERATION
{
    TRANSLATE_X = (1u << 0),
    TRANSLATE_Y = (1u << 1),
    TRANSLATE_Z = (1u << 2),
    ROTATE_X    = (1u << 3),
    ROTATE_Y    = (1u << 4),
    ROTATE_Z    = (1u << 5),
    ROTATE_SCREEN = (1u << 6),
    SCALE_X     = (1u << 7),
    SCALE_Y     = (1u << 8),
    SCALE_Z     = (1u << 9),
    BOUNDS      = (1u << 10),
    SCALE_XU    = (1u << 11),
    SCALE_YU    = (1u << 12),
    SCALE_ZU    = (1u << 13),

    TRANSLATE = TRANSLATE_X | TRANSLATE_Y | TRANSLATE_Z,
    ROTATE    = ROTATE_X | ROTATE_Y | ROTATE_Z | ROTATE_SCREEN,
    SCALE     = SCALE_X | SCALE_Y | SCALE_Z,
    SCALEU    = SCALE_XU | SCALE_YU | SCALE_ZU, // universal
    UNIVERSAL = TRANSLATE | ROTATE | SCALEU
};
```

》》》代码设计：窗口响应

```
216 - Application::Get().GetImGuiLayer()->BlockEvents(!m_ViewportFocused || !m_ViewportHovered);
220 + Application::Get().GetImGuiLayer()->BlockEvents(!m_ViewportFocused && !m_ViewportHovered);
```

更改前：只要不满足 聚焦于窗口上/悬停在窗口上的任——一个条件，便会阻塞当前窗口中的事件（不再捕获鼠标或键盘的活动）

更改后：只有 窗口没有被聚焦且鼠标没有悬停在窗口上的时候，才会阻塞事件。

这让我们在结构面板中选中实体之后，只需要将鼠标悬停在 viewport 窗口上，便可以通过键盘调整/响应 Gizmo，这在实际使用中很舒服（本人亲测：>

》》》为什么这里需要使用 Const 标识？

```
Entity Scene::GetPrimaryCamera()
{
    auto view = m_Registry.view<CameraComponent>();
    for (auto entity : view)
    {
        const auto& cameraComponent = m_Registry.get<CameraComponent>(entity);
        if (cameraComponent.Primary == true)
            return Entity{ entity, this };
    }
    return {};
}
```

可能是因为没有需要对 cameraComponent 进行更改吧。

》》》这个函数的意义？

```
#pragma once

#include <glm/glm.hpp>

namespace Nut { namespace Math {
    bool DecomposeTransform(const glm::mat4& transform, glm::vec3& outTranslation, glm::vec3& outRotation, glm::vec3& outScale);
}}

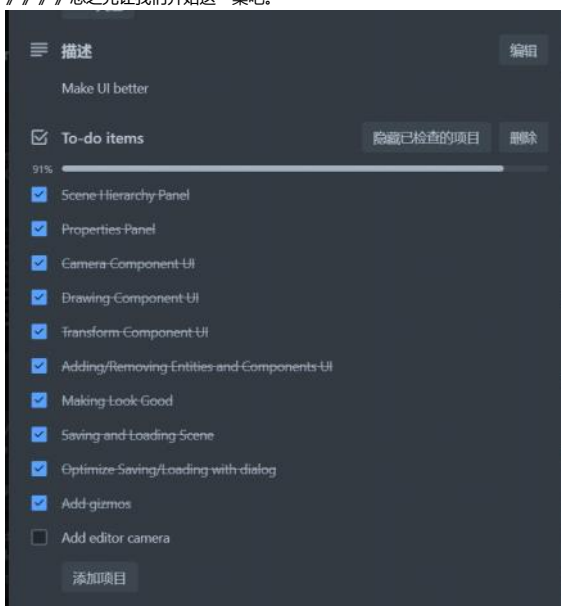
```

》》》这段代码的意义？

```
if (ImGui::IsUsing())
{
    glm::vec3 translation, rotation, scale;
    Math::DecomposeTransform(transform, translation, rotation, scale);
    glm::vec3 deltaRotation = rotation - tc.Rotation;
    tc.Translation = translation;
    tc.Rotation += deltaRotation;
    tc.Scale = scale;
}
```

----- Editor camera -----

》》》关于引擎的操作界面和简单操纵，我们已经完成了很多，Cherno 在接下来将开始实现鼠标选择实体，不过我可能会将之前遗漏的维护补充一下。
 》》》总之先让我们开始这一集吧。



》》》这一集中，主要围绕 编辑时摄像机 和 运行时摄像机 两个概念来设计。
 我目前是这样理解的：

```

void OnUpdate(Timestep ts);
void OnUpdateRuntime(Timestep ts);
void OnUpdateEditor(Timestep ts, EditorCamera& camera);
void OnViewportResize(uint32_t width, uint32_t height);

```

- 编辑时摄像机：用于在引擎中编辑物体，此摄像机为默认存在。只要你需要对物体进行编辑，那么直接建立一个 sprite 实体，在编辑器中便应该能够直接查看和编辑该实体，而不需要额外添加摄像机（运行时摄像机），这符合游戏引擎的使用逻辑。
- 运行时摄像机：用于在实际游戏程序运行时添加，此摄像机需要手动添加。游戏中不需要“上帝视角”的编辑时摄像机，而只需要用来观察物体的运行时摄像机，该摄像机不像编辑时摄像机一样默认存在，因为编辑器和游戏程序的逻辑是不同的。

这两个摄像机不是同时存在的：在编辑器中，底层默认存在一个编辑时摄像机，该相机只在编辑器运行时存在并更新；同样的，游戏程序中不使用编辑时摄像机，而是用运行时摄像机，该相机只在游戏程序运行时存在并更新。

由于我们现在在编辑器中设计程序，所以只需要对所有实体使用编辑时摄像机即可。对于游戏，我想我们应该需要在编辑器中添加一个游戏运行时摄像机，然后打开游戏程序，进行游玩。

----- Multiple Render Targets and Framebuffer refactor -----

》》》很久没 commit 了，之前准备先做维护，奈何 Chernov 后面实现的内容太过诱人，好奇心驱使我又观看了五六集，维护的事情后面再说吧 x-d
 》》》接下来我将努力实现 Mouse Picking，然后根据个人时间考虑内容浏览面板的制作，或者是提交一波维护，因为维护已经落下很久了。

》》》前言：

》》》gl_VertexID 在 GLSL 中关于顶点ID的一些细节：

gl_VertexID

gl_Position 和 gl_PointSize 都是**输出变量**，因为它们的值是作为顶点着色器的输出被读取的。我们可以对它们进行写入，来改变结果。顶点着色器还为我们提供了一个有趣的**输入变量**，我们只能对它进行读取，它叫做 gl_VertexID。

整型变量 gl_VertexID 储存了正在绘制顶点的当前ID。当（使用 glDrawElements）进行索引渲染的时候，这个变量会存储正在绘制顶点的当前索引。当（使用 glDrawArrays）不使用索引进行绘制的时候，这个变量会储存从渲染调用开始的已处理顶点数量。

虽然现在它没有什么具体的用途，但知道我们能够访问这个信息总是好的。

》》》这一节的概述：

Chernov 为了实现类似顶点ID的效果，于是开始更改帧缓冲的操作模式。这样一来就可以更方便的实现鼠标选中，在这一节中，Chernov 只是对帧缓冲的实现过程进行了完善，使代码可以通过初始化列表标识帧缓冲，并自动的识别这些标识用来创建帧缓冲。虽然是顶点ID的前瞻操作，但实际上只是一些铺垫，不必担心看不明白。

》》》代码的理解：

反复观看代码之后，我将所有代码分为两部分：设置结构体和 重构帧缓冲的创建，第一部分的作用是：通过声明一些结构体来简化帧缓冲的使用方式/简化帧缓冲的设计，第二部分的作用是：通过已有的条件，动态的构建帧缓冲。

这引出了我的一些问题：

In FrameBuffer.h

```

7 + enum class FramebufferTextureFormat
8 + {
9 +     None = 0,
10 +
11 +     // Color
12 +     RGBAB,
13 +
14 +     // Depth/stencil
15 +     DEPTH24STENCIL8,
16 +
17 +     // Defaults
18 +     Depth = DEPTH24STENCIL8
19 + };
20 +
21 + struct FramebufferTextureSpecification
22 + {
23 +     FramebufferTextureSpecification() = default;
24 +     FramebufferTextureSpecification(FramebufferTextureFormat format)
25 +         : TextureFormat(format) {}
26 +
27 +     FramebufferTextureFormat TextureFormat = FramebufferTextureFormat::None;
28 +     // TODO: filtering/wrap
29 + };
30 +
31 + struct FramebufferAttachmentSpecification
32 + {
33 +     FramebufferAttachmentSpecification() = default;
34 +     FramebufferAttachmentSpecification(std::initializer_list<FramebufferTextureSpecification> attachments)
35 +         : Attachments(attachments) {}
36 +
37 +     std::vector<FramebufferTextureSpecification> Attachments;
38 + };

```

》》这三个结构体有什么作用？之间有什么关系？

FramebufferTextureFormat:	这个枚举类定义了可用的帧缓冲纹理格式
FramebufferTextureSpecification:	一个使用枚举类型对象进行初始化的类，用于定义单个帧缓冲纹理的具体规格
FramebufferAttachmentSpecification:	<p>用于定义一组帧缓冲的附件</p> <p>(我认为这个结构体主要是用来服务于初始化的：比如 Chernov 的代码中的使用方式)</p> <pre>FramebufferSpecification compFramebuffersSpec; compFramebuffersSpec.Attachments = { FramebufferTextureFormat::RGBA8 }; compFramebuffersSpec.ClearColor = { 0.1f, 0.1f, 0.1f, 1.0f };</pre> <p>(再比如：)</p> <pre>26 26 m_CheckerboardTexture = Texture2D::Create("assets/textures/Checkerboard.png"); 27 27 28 28 FramebufferSpecification fbSpec; 29 + fbSpec.Attachments = { FramebufferTextureFormat::RGBA8, FramebufferTextureFormat::Depth }; 29 30 fbSpec.Width = 1280; 30 31 fbSpec.Height = 720; 31 32 m_Framebuffer = Framebuffer::Create(fbSpec);</pre> <p>由于这个 FramebufferAttachmentSpecification 的构造函数中使用的是初始化列表，所以我们可以通过上述方式对 framebufferSpecification 中的 Attachments 进行方便快捷的初始化操作。</p> <p>填入的初始化列表将被储存在 FramebufferSpecification 结构体中的 Attachments 中：</p> <pre>struct FramebufferSpecification { uint32_t Width = 0, Height = 0; // FramebufferFormat Format = FramebufferAttachmentSpecification Attachments; uint32_t Samples = 1; bool SwapChainTarget = false; };</pre> <p>并在 OpenGLFramebuffer 的构造函数中被使用：（第一个 Attachments 就是 FramebufferSpecification 结构体中的成员，第二个 Attachments 则是 FramebufferAttachmentSpecification 结构体中的成员->查看最开始的那张代码表）</p> <pre>OpenGLFramebuffer::OpenGLFramebuffer(const FramebufferSpecification& spec) : m_Specification(spec) { for (auto spec : m_Specification.Attachments.Attachments) { if (!Utils::IsDepthFormat(spec.TextureFormat)) m_ColorAttachmentSpecifications.emplace_back(spec); else m_DepthAttachmentSpecification = spec; } Invalidate(); }</pre>

In OpenGLFramebuffer.h

```
std::vector<FramebufferTextureSpecification> m_ColorAttachmentSpecifications;
FramebufferTextureSpecification m_DepthAttachmentSpecification = FramebufferTextureFormat::None;

std::vector<uint32_t> m_ColorAttachments;
uint32_t m_DepthAttachment = 0;
```

》》m_ColorAttachmentSpecification 和 m_ColorAttachments 的区别在哪里？分别起到什么作用？

前言：帧缓冲中的纹理与纹理附件的关系？帧缓冲中是否可以包括多个纹理？一个纹理中是否可以附加多个纹理附件？这些附件有什么作用？
一个帧缓冲绘制的场景中可以绘制一些物体，也可以绘制多个纹理，而纹理附件则是存储这些渲染结果的地方。此时如果你使用帧缓冲中的颜色附件作为纹理进行后续渲染，那个颜色附件就可以被称为纹理附件。
在帧缓冲中每一个纹理的创建都和普通纹理的创建过程差不多，不过我们需要对这些纹理额外的附加 Attach 一些附件。一个纹理可以附加多个附件（颜色附件可以同时附加多个，但深度附件一般只能附加一个），如果你在帧缓冲对象（FBO）中使用多个颜色附件，那并不意味着你能绘制更多的物体，而是指在一次渲染过程中，你可以同时输出多个数据流。

m_ColorAttachmentSpecification 用于存储所有的纹理附件，而 m_ColorAttachments 存储的是帧缓冲中所有纹理附件的 ID。

》》那么这个ID是怎么分配的呢？

```
// Attachments
if (m_ColorAttachmentSpecifications.size())
{
    m_ColorAttachments.resize(m_ColorAttachmentSpecifications.size());
    Utils::CreateTextures(multisample, m_ColorAttachments.data(), m_ColorAttachments.size());

    for (size_t i = 0; i < m_ColorAttachments.size(); i++)
    {
        Utils::BindTexture(multisample, m_ColorAttachments[i]);
        switch (m_ColorAttachmentSpecifications[i].TextureFormat)
        {
            case FramebufferTextureFormat::RGBA8:
                Utils::AttachColorTexture(m_ColorAttachments[i], m_Specification.Samples,
```

在进行附件的附加操作时，先有这样一句代码： m_ColorAttachments.resize(m_ColorAttachmentSpecifications.size());	其中 m_ColorAttachmentSpecifications.size() 返回的是 vector 中的元素数量（即纹理附件的数量），如此一来 m_ColorAttachments 的大小就被初始化为 m_ColorAttachmentSpecifications 中的元素数，数据全部为 0。
在随后的代码中： Utils::CreateTextures(multisample, m_ColorAttachments.data(), m_ColorAttachments.size());	CreateTextures 调用了glCreateTextures(TextureTarget(multisampled), count, outID); 这会在函数被调用的过程中自动为纹理分配 ID，这个 ID 正是我们填入的 m_ColorAttachments.data()（从一开始的全部为0到创建完纹理之后 gl 函数自动分配的 ID）

》》这个 ID 是怎样和片段着色器中的输出变量 Color 关联起来的？也就是说纹理附件的 ID 是怎样和片段着色器中的输出变量关联起来的？
比如 Cherno 是怎样明确纹理附件的 ID：0 就指的是片段着色器中的第1个颜色（输出变量），ID：1就指的是片段着色器中的第2个颜色（输出变量）。
Eq.

```
uint64_t textureID = m_Framebuffer->GetColorAttachmentRendererID(1);
ImGui::Image(reinterpret_cast<void*>(textureID), ImVec2{ m_V viewportSize.x, m_V viewportSize.y }, ImVec2{ 0, 1 }, ImVec2{ 1, 0 });
```

通过纹理 ID 索引到纹理附件

```
// Attachments
if (m_ColorAttachmentSpecifications.size())
{
    m_ColorAttachments.resize(m_ColorAttachmentSpecifications.size());
    Utils::CreateTextures(multisample, m_ColorAttachments.data(), m_ColorAttachments.size());

    for (size_t i = 0; i < m_ColorAttachments.size(); i++)
    {
        Utils::BindTexture(multisample, m_ColorAttachments[i]);
        switch (m_ColorAttachmentSpecifications[i].TextureFormat)
        {
            case FramebufferTextureFormat::RGBA8:
                Utils::AttachColorTexture(m_ColorAttachments[i], m_Specification.Samples, GL_RGBA8, m_Specification.Width,
                m_Specification.Height, i);
                break;
        }
    }
}
```

1 Utils::CreateTextures (multisample, m_ColorAttachments.data(), m_ColorAttachments.size());	首先将一组纹理创造出来，为其分配对应的ID
2 Utils::BindTexture(multisample, m_ColorAttachments[i]);	在 BindTexture 函数中，填入对应的纹理 ID 作为参数，在明确绑定了某一个纹理的情况下，为该纹理分配附加附件（在此基础上使用 AttachColorTexture 函数，将输出变量 Color 映射到颜色附件 0（GL_COLOR_ATTACHMENT0））
3 Utils::AttachColorTexture (m_ColorAttachments[i], m_Specification.Samples, GL_RGBA8, m_Specification.Width, m_Specification.Height, i);	在该函数的具体实现中，通过：glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0 + index, TextureTarget(multisampled), id, 0); 我们明确表示了在某一纹理的条件下，为其绑定的附件。 GL_COLOR_ATTACHMENT0 便是帧缓冲对象的第一个颜色附件。我们将该附件按顺序的附加到对应的纹理ID下。

通过纹理附件（颜色附件）索引到输出变量

现在我们已经可以通过纹理 ID 索引到想要访问的纹理附件了，可如何将 ID 与片段着色器中的颜色输出变量联系起来？

渲染目标的设置：在使用多重渲染目标（MRT）时，可以将片段着色器的输出变量映射到不同的颜色附件。

GLenum drawBuffers[2] = { GL_COLOR_ATTACHMENT0, GL_COLOR_ATTACHMENT1 }; glDrawBuffers(2, drawBuffers);	这将会把 Color0 的输出写入到 GL_COLOR_ATTACHMENT0，而 Color1 的输出写入到 GL_COLOR_ATTACHMENT1。
---	---

Eg.

```
if (m_ColorAttachments.size() > 1)
{
    HZ_CORE_ASSERT(m_ColorAttachments.size() <= 4);
    GLenum buffers[4] = { GL_COLOR_ATTACHMENT0, GL_COLOR_ATTACHMENT1, GL_COLOR_ATTACHMENT2, GL_COLOR_ATTACHMENT3 };
    glDrawBuffers(m_ColorAttachments.size(), buffers);
}
else if (m_ColorAttachments.empty())
{
    // Only depth-pass
    glDrawBuffer(GL_NONE);
}
```

》》》》代码上的疑问：为什么需要在附加附件之前创建纹理


```
// Attachments
if(m_ColorAttachmentSpecifications.size())
{
    m_ColorAttachmentIDs.resize(m_ColorAttachmentSpecifications.size());
    Utils::CreateTextures();

    for(size_t i = 0; i < m_ColorAttachmentIDs.size(); i++)
    {
        Utils::BindTextures();
        switch(m_ColorAttachmentSpecifications[i])
        {
            case FrameBufferAttachmentFormat::RGBA8:
                Utils::AttachColorTexture();
                break;
        }
    }
}

if (m_BufferAttachmentSpecification != FrameBufferAttachmentFormat::None)
{
    Utils::CreateTextures();
    Utils::BindTexture();
    switch(m_BufferAttachmentSpecification)
    {
        case FrameBufferAttachmentFormat::DEPTH24STENCIL8:
            Utils::AttachBufferTexture();
            break;
    }
}
```

为什么每附加一个附件就需要创建一次纹理？

前提：	Utils::CreateTextures 函数的作用是创建纹理对象，具体来说就是为帧缓冲区的颜色附件和深度附件分配和初始化必要的纹理资源。一个纹理可以包含多种不同的附件。
原因：	因为创建纹理是渲染管线中的关键步骤，为了确保在绘制时渲染结果可以储存在合适的纹理中，我们需要创建纹理。

虽然一个纹理对象可以附加多个颜色附件，且只能附加一个深度附件，那为什么需要在附加深度缓冲附件之前再次新建一个纹理？

这些纹理都代表什么，为什么可以创建这么多？

颜色附件和深度附件的规范

颜色附件：	数量：颜色附件用于存储渲染输出的颜色信息。可以有多个颜色附件，因为在许多渲染管线中，可能需要输出多个颜色通道（例如，颜色缓冲、法线缓冲、光照缓冲等）。 附件和纹理的关系：每个颜色附件可以使用不同格式的纹理，这样可以根据需求选择最合适的格式和精度。
深度附件：	数量：深度附件用于存储每个像素的深度信息，用于进行深度测试以判断哪些物体在前面、哪些在后面。在一个帧缓冲对象中只能有一个深度附件，这通常是因为深度测试只需要一个深度值来进行比较。

创建多个纹理的原因

性能和灵活性：	使用不同的纹理对象允许开发者根据需要选择不同的格式、分辨率和精度。例如，可能会希望使用高精度深度纹理，但对于颜色输出，可以选择较低精度的纹理。
资源管理：	创建多个纹理可以更好地管理内存和资源。根据不同的渲染需求，可以动态创建和销毁纹理，而不是固定使用一个纹理。
多重渲染目标（MRT）：	当使用多重渲染目标（MRT）时，可以同时将多个颜色输出渲染到不同的纹理。这种灵活性可以优化渲染流程，减少多次绘制的需求。

在深度附件之前创建纹理的原因

指定深度格式：	每个深度附件可能使用不同的格式（如深度16、深度24等），需要根据需要新建适当的纹理。
资源隔离：	将颜色和深度缓冲分开，可以更容易地管理和调试渲染过程。

》》》附件的附加代码：附加时使用不同函数的意义是什么？

```
static void AttachColorTexture(uint32_t id, int samples, GLenum format, uint32_t width, uint32_t height, int index)
{
    bool multisampled = samples > 1;
    if(multisampled)
    {
        glTexImage2DMultisample(GL_TEXTURE_2D_MULTISAMPLE, samples, format, width, height, GL_FALSE);
    }
    else
    {
        glTexImage2D(GL_TEXTURE_2D, 0, format, width, height, 0, GL_RGBA, GL_UNSIGNED_BYTE, nullptr);

        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_R, GL_CLAMP_TO_EDGE);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
    }

    glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0 + index, TextureTarget(multisampled), id, 0);
}
```

```
static void AttachBufferTexture(uint32_t id, int samples, GLenum format, GLenum attachmentType, uint32_t width, uint32_t height)
{
    bool multisampled = samples > 1;
    if (multisampled)
    {
        glTexImage2DMultisample(GL_TEXTURE_2D_MULTISAMPLE, samples, format, width, height, GL_FALSE);
    }
    else
    {
        glTexStorage2D(GL_TEXTURE_2D, 1, format, width, height);

        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_R, GL_CLAMP_TO_EDGE);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);

        // ----- Or you can use renderbuffer object to attach depth/stencil attachment -----
        //glCreateRenderbuffers(1, &id);
        //glBindRenderbuffer(GL_RENDERBUFFER, id);
        //glRenderbufferStorage(GL_RENDERBUFFER, format, width, height);
        //glBindRenderbuffer(GL_RENDERBUFFER, 0);
    }
    glFramebufferTexture2D(GL_FRAMEBUFFER, attachmentType, TextureTarget(multisampled), id, 0);
}
```

两种附件中使用的函数：glTexImage2D 和 glTexStroage2D 区别在哪里？

1. 创建纹理存储的方式

AttachColorTexture 中使用 glTexImage2D:	glTexImage2D(GL_TEXTURE_2D, 0, format, width, height, 0, GL_RGBA, GL_UNSIGNED_BYTE, nullptr); 这里调用 glTexImage2D 为纹理分配存储，并且在创建时可以上传像素数据（虽然这里传递的是 nullptr）。 <u>它适合需要动态更新纹理内容的情况，因为你可以在后续的调用中修改纹理数据，但是调用时可能会重新分配内存。</u>
AttachBufferTexture 中使用 glTexStorage2D:	glTexStorage2D(GL_TEXTURE_2D, 1, format, width, height); 这里使用 glTexStorage2D 创建了一个不可变的纹理存储。调用这个函数时，你只指定了格式和尺寸，之后不允许改变存储结构。 <u>它更适合静态纹理，因为它只需要在创建时分配一次内存，并且不需要再进行二次的内存分配和管理。</u>

适用场景：

glTexImage2D	适合于动态纹理，例如实时渲染时需要频繁更新的纹理颜色或样式....。
优劣:	能够灵活地上传和更新纹理数据，但可能导致性能下降。
glTexStorage2D	更适合静态纹理，如用于环境映射、天空盒或其他不需要在运行时修改的纹理。
优劣:	能够提供更好的性能和内存管理，但需要在创建时就确定纹理的格式和尺寸，之后不再改变。

》》关于纹理创建和附加操作的一点理解：

- 1 由于颜色附件（纹理附件）需要附加多个，所以需要创建多个纹理，并逐一对其进行 Attach 操作。
- 2 但是深度附件只需要添加一个，所以只创建一个纹理，并对其进行一次 Attach 操作。

```
// ----- Create Texture Attachment -----
bool multisample = m_Specification.Samples > 1;

if (m_ColorAttachmentSpecifications.size())
{
    m_ColorAttachmentIDs.resize(m_ColorAttachmentSpecifications.size());
    Utils::CreateTextures(multisample, m_ColorAttachmentIDs.size(), m_ColorAttachmentIDs.data());
    for (size_t i = 0; i < m_ColorAttachmentIDs.size(); i++)
    {
        Utils::BindTexture(multisample, m_ColorAttachmentIDs[i]);
        switch (m_ColorAttachmentSpecifications[i])
        {
            case FrameBufferAttachmentFormat::RGBA8:
                Utils::AttachColorTexture(m_ColorAttachmentIDs[i], m_Specification.Samples, GL_RGBA8, m_Specification.Width, m_Specification.Height);
                break;
        }
    }
}

// ----- Create Renderbuffer Object -----
if (m_BufferAttachmentSpecification != FrameBufferAttachmentFormat::None)
{
    Utils::CreateTextures(multisample, 1, &m_BufferAttachmentID);
    Utils::BindTexture(multisample, m_BufferAttachmentID);
    switch (m_BufferAttachmentSpecification)
    {
        case FrameBufferAttachmentFormat::DEPTH24STENCIL8:
            Utils::AttachBufferTexture(m_BufferAttachmentID, m_Specification.Samples, GL_DEPTH24_STENCIL8, GL_DEPTH_STENCIL_ATTACHMENT0);
            break;
    }
}
```

》》》以下两段代码的区别

```
// Mapping the fragment shader's output variables to different color attachments
if(m_ColorAttachmentIDs.size() > 1)
{
    // Only depth pass
}
```

和

```
// Mapping the fragment shader's output variables to different color attachments
if(!m_ColorAttachmentIDs.empty())
{
    // Only depth pass
}
```

的区别：

答：Size > 1 是数量从 2 开始的情况，!empty() 是数量只要不为0，也就是数量从 1 开始的情况。

》》》glDrawBuffers 函数的作用是什么？

```
// ----- Mapping the fragment shader's output variables to different color attachments -----
if (m_ColorAttachmentIDs.size() > 1)
{
    NUT_CORE_ASSERT(m_ColorAttachmentIDs.size() <= 4, "");

    GLenum buffers[4] = { GL_COLOR_ATTACHMENT0, GL_COLOR_ATTACHMENT1, GL_COLOR_ATTACHMENT2, GL_COLOR_ATTACHMENT3 };
    glDrawBuffers(m_ColorAttachmentIDs.size(), buffers);
}
else if (m_ColorAttachmentIDs.empty())
{
    // Only depth pass
    glDrawBuffer(GL_NONE);
}
```

----- Preparing mouse picking (Get mouse pos and Read pixel) -----

》》》关于这一集，可以分为三部分：1.调整帧缓冲的使用方法 2.读取当前帧缓冲区（或指定帧缓冲区）中的像素数据 3.获取正确的窗口大小边界，并获取鼠标相对于窗口中的位置

》》》关于读取当前帧缓冲区（或指定帧缓冲区）中的像素数据

glReadBuffer

原型：	void glReadBuffer(GLenum src);
参数：	<p>这些选项指定了你希望读取的帧缓冲对象（FBO）中的缓冲区。</p> <p>src（类型：GLenum）：指定你希望读取的源缓冲区。这个参数的有效值有：</p> <ul style="list-style-type: none">GL_FRONT_LEFT：从前面左侧的颜色缓冲区读取（传统的前台缓冲区，OpenGL 默认的缓冲区之一）。GL_FRONT_RIGHT：从前面右侧的颜色缓冲区读取（如果双缓冲渲染）。GL_BACK_LEFT：从后面左侧的颜色缓冲区读取（通常用于双缓冲模式）。GL_BACK_RIGHT：从后面右侧的颜色缓冲区读取（如果双缓冲渲染）。GL_COLOR_ATTACHMENT0, GL_COLOR_ATTACHMENT1, ... GL_COLOR_ATTACHMENTn：从多个渲染目标中的颜色附件中读取，这个是现代 OpenGL（FBO）中常用的选择。GL_DEPTH_ATTACHMENT：从深度缓冲区读取。GL_STENCIL_ATTACHMENT：从模板缓冲区读取。
释义：	<p>glReadBuffer 的主要作用是指定从哪个缓冲区读取数据。</p> <p>这在多渲染目标（MRT, Multiple Render Targets）或使用帧缓冲对象（FBO）时非常有用，因为在这些情况下，你可能渲染到多个不同的缓冲区，并且需要从特定的缓冲区获取像素数据。</p>

glReadPixels

原型：	void glReadPixels(GLint x, GLint y, GLsizei width, GLsizei height, GLenum format, GLenum type, void *pixels);
参数说明：	<p>x：读取区域左下角的 X 坐标，单位是像素。指定从帧缓冲中开始读取的位置。</p> <p>y：读取区域左下角的 Y 坐标，单位是像素。指定从帧缓冲中开始读取的位置。注意，在 OpenGL 中，Y 轴的坐标是从底部到顶部增加的，所以 y 的值越大，读取的区域就越靠上。</p> <p>width：指定要读取的区域的宽度（以像素为单位）。</p>

	<p>height: 指定要读取的区域的高度（以像素为单位）。</p> <p>举例（width, height）</p> <ul style="list-style-type: none">• 1, 1: 读取区域大小: 指定从 (x, y) 开始的区域为 1 像素宽, 1 像素高, 即仅读取一个像素的数据。 实际效果: 这意味着你会读取帧缓冲区中 x, y 位置处的单个像素的颜色信息。• 100, 100: 读取区域大小: 指定从 (x, y) 开始的区域为 100 像素宽, 100 像素高, 即读取一个 100x100 像素的矩形区域内所有像素的数据。 实际效果: 这意味着你会读取从 (x, y) 开始, 横向 100 个像素, 纵向 100 个像素的区域内所有像素的颜色信息。 <p>format: 指定返回像素数据的格式, 常见的有:</p> <ul style="list-style-type: none">• GL_RGB: 以 RGB 格式返回数据。• GL_RGBA: 以 RGBA 格式返回数据。• GL_DEPTH_COMPONENT: 返回深度值（通常是一个浮点数值）。• GL_STENCIL_INDEX: 返回模板缓冲区的值。 <p>type: 指定返回数据的类型。常见的类型包括:</p> <ul style="list-style-type: none">• GL_UNSIGNED_BYTE: 每个像素的每个分量用一个无符号字节（0~255）表示。• GL_FLOAT: 每个像素分量用浮点数表示（0~1 范围）。• GL_UNSIGNED_SHORT: 每个像素分量用无符号短整数表示（通常是 16 位）。• GL_UNSIGNED_INT: 每个像素分量用无符号整数表示（通常是 32 位）。 <p>pixels: 指向存储读取像素数据的内存区域的指针。读取到的像素数据将被存储在这个内存空间中。该内存的大小应该足够容纳所有读取的像素数据。</p>
释义:	<p>glReadPixels 是 OpenGL 中用于读取当前帧缓冲区（或指定帧缓冲区）中的像素数据的函数。</p> <p>它通常用于从渲染到屏幕或帧缓冲的内容中提取像素信息, 可以用于后处理、截图、或用于其他需要读取像素数据的操作。</p>

》》》关于窗口和鼠标交互部分代码的理解

```
// 确定窗口边界 (考虑标签栏)
auto viewportOffset = ImGui::GetCursorPos(); // Includes tab bar

auto windowSize = ImGui::GetWindowSize();
ImVec2 minBound = ImGui::GetWindowPos();
minBound.x += viewportOffset.x;
minBound.y += viewportOffset.y;

ImVec2 maxBound = { minBound.x + windowSize.x, minBound.y + windowSize.y };
m_ViewportsBounds[0] = { minBound.x, minBound.y };
m_ViewportsBounds[1] = { maxBound.x, maxBound.y };

// 确定鼠标在视口中的相对位置, 并做出对应的操作
auto[mx, my] = ImGui::GetMousePos();
mx -= m_ViewportsBounds[0].x;
my -= m_ViewportsBounds[0].y;
glm::vec2 viewportSize = m_ViewportsBounds[1] - m_ViewportsBounds[0];
my = viewportSize.y - my;
int mouseX = (int)mx;
int mouseY = (int)my;

if (mouseX >= 0 && mouseY >= 0 && mouseX < (int)viewportSize.x && mouseY < (int)viewportSize.y)
{
    int pixelData = m_Framebuffer->ReadPixel(1, mouseX, mouseY);
    HZ_CORE_WARN("Pixel data = {0}", pixelData);
}
```

----- ImGui::GetCursorPos()

`GetCursorPos()` cursor position in window coordinates (relative to window position)

`GetCursorPos()`窗口坐标中的光标位置（相对于窗口位置）

```
ImVec2 GetCursorPos()
```

ImGui::GetCursorPos()	<p>作用： 获取光标位置: 返回当前光标在窗口中的位置, 通常在布局过程中用于对齐和排列控件。</p> <p>返回值: 返回 ImVec2 类型变量, 表示光标的 x 和 y 坐标。返回值的 x 和 y 是相对于当前窗口的左上角的坐标, 单位为像素。</p>
-----------------------	--

注意

ImGui::GetCursorPos() 返回的是 **ImGui 光标的位置**, 而不是鼠标光标的位置。这里的“ImGui 光标”是一个特定的概念, 与 GUI 元素的布局和绘制有关。

ImGui 光标

概念: 在 ImGui 中, 光标是用于管理和控制 UI 元素（如按钮、文本框、滑动条等）排列和绘制的一个指示器。它并不是真正的鼠标光标, 而是指示下一个元素应该放置在何处的虚拟光标。

特性:

布局控制:	ImGui 使用即时模式的绘制方式, 光标帮助管理 UI 元素的排列。每当你在 ImGui 中绘制一个元素时, 光标的位置会自动更新, 指向下一个可放置元素的位置。例如, 当你调用 <code>ImGui::Button("Button")</code> 时, 按钮会被绘制在当前光标的位置。
-------	--

位置更新/使用方式:	每当添加一个新的 UI 元素时, 光标会自动向下移动, 准备好放置下一个元素。 例如, 你可以通过 <code>ImGui::GetCursorPos()</code> 获取当前光标的位置。
光标的偏移:	光标位置可以通过多种方式调整。 例如, 可以使用 <code>ImGui::SetCursorPos()</code> 函数手动设置光标位置, 或者通过调用 <code>ImGui::NewLine()</code> 来强制光标换行。
相对坐标:	<code>GetCursorPos()</code> 返回的是 相对于当前窗口的光标位置 , 这意味着它会考虑到窗口的布局 (如标签条、边距等), 并返回一个相对位置。

实际应用:
当你在 ImGui 中创建界面时, 通常不会直接处理鼠标光标的位置, 而是依赖于 ImGui 光标自动控制 UI 元素的布局。
Eg.
`ImGui::Text("Hello, World!");`
`ImGui::Button("Click Me!");`
在这个例子中, Text 和 Button 会根据 ImGui 光标的位置自动排列。当第一个元素被绘制后, 光标会向下移动, 按钮会出现在文本的下方。

ImGui光标的格式

坐标格式:	ImGui 光标的位置是一个具体的坐标, 表示相对于当前窗口的左上角 (0,0 点) 的位置。 这个坐标通常是以像素为单位的, 并以 <code>ImVec2</code> 类型返回, 其中包含两个浮点值: x 和 y, 分别表示水平方向和垂直方向的坐标。
-------	---

----- ImGui::GetWindowSize()

<code>ImGui::GetWindowSize()</code>	返回当前窗口的宽度和高度 (<code>ImVec2</code>), 便于在绘制控件时进行相应的布局。
-------------------------------------	--

----- ImGui::GetWindowPos()

<code>ImGui::GetWindowPos()</code>	作用: 获取当前 ImGui 窗口的位置信息, 通常用于确定窗口在屏幕上的位置。 坐标位置: 返回当前窗口左上角相对于屏幕的坐标位置, 便于进行布局和处理鼠标事件等。
------------------------------------	--

----- ImGui::GetMousePos()

<code>ImGui::GetMousePos()</code>	返回当前鼠标光标在屏幕上的坐标 (全局坐标)
-----------------------------------	------------------------

```
《《代码理解: 为何一下代码可以动态的计算出视口 (窗口) 的可用区域? (动态: 指的是可以根据标签栏是否存在, 计算出此时的窗口可用区域)
auto viewportOffset = ImGui::GetCursorPos(); // Includes tab bar
ImVec2 minBound = ImGui::GetWindowPos();

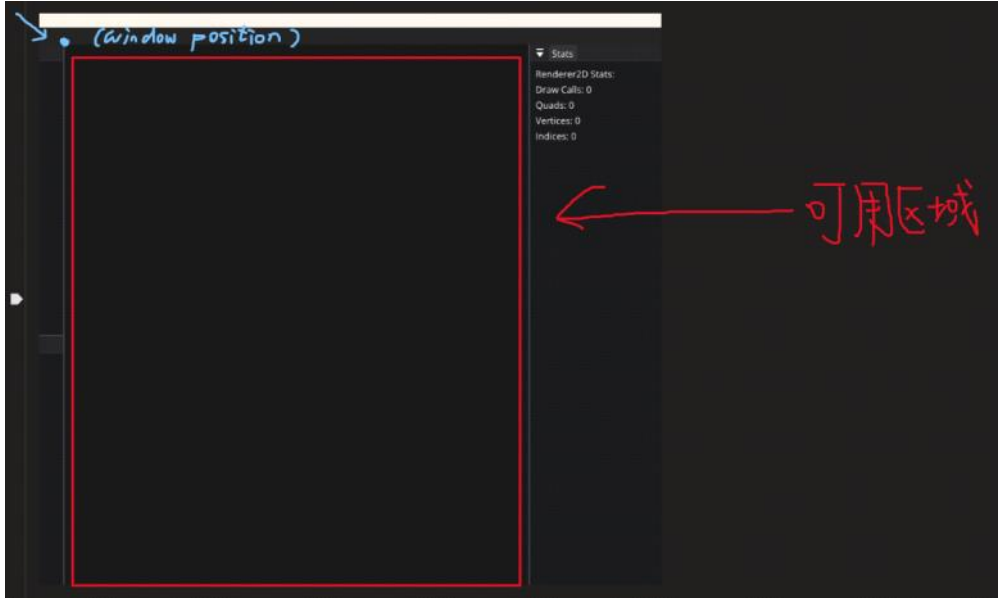
minBound.x += viewportOffset.x;
minBound.y += viewportOffset.y;
```

根据上述 `GetWindowSize()` 函数的理解, 我们得知这个函数返回的是 ImGui 光标的位置。这个光标记载了 ImGui 控件的位置, 当便签栏存在时, 这个函数返回的值会发生变化 (无标签栏时返回可用区域的左上角坐标; 有标题栏的时候, 会因为窗口中多绘制的标题栏控件, 而返回已经发生变化的坐标)

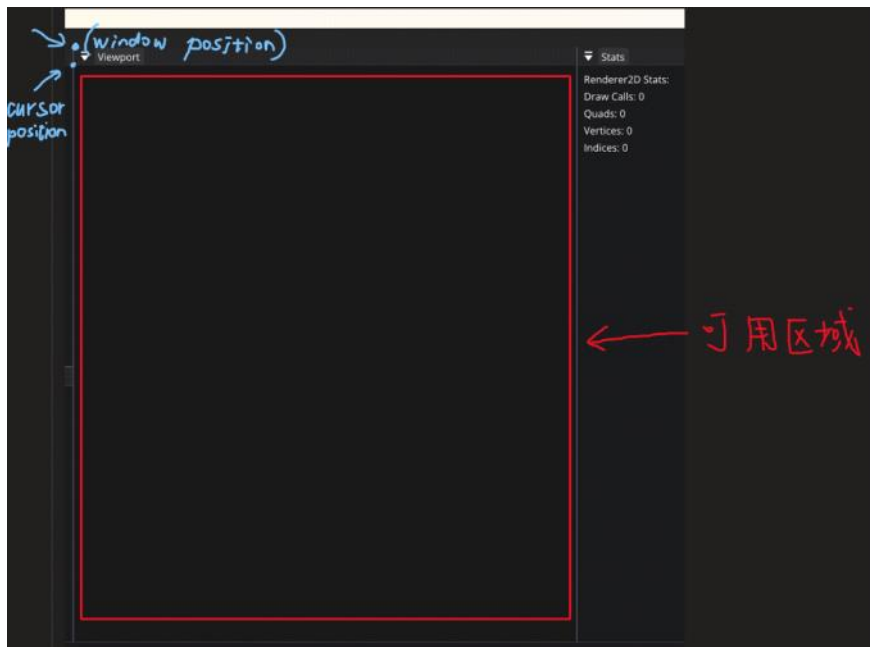
(全局坐标 + 相对坐标 -> 符合需求的全局坐标)

由于 `ImVec2 minBound = ImGui::GetWindowPos()` 这里得到的是窗口在屏幕中的全局坐标, 而 `ImGui::GetCursorPos()` 得到的是控件在窗口中的相对坐标, 所以对于这个全局坐标, 我们加上相对坐标, 便可以计算出控件存在时的可用范围。

无标签栏时:



有标签栏时:



《《代码理解：如何计算的相对坐标（相对坐标->指的是将鼠标的全局位置修改为相对于视口窗口的坐标，准确位置相对于视口窗口的左上角。因为对于鼠标坐标，我们都对其减去了minBound）》》

```
auto[mx, my] = ImGui::GetMousePos();
mx -= m_ViewportBounds[0].x;
my -= m_ViewportBounds[0].y;
```

(全局坐标 - 全局坐标 = 对于窗口的相对坐标)

计算全局鼠标坐标相对于窗口的位置，比如：假设全局鼠标坐标是 (500, 300)，窗口左上角的坐标是 (450, 250)。通过计算：

$mx = 500 - 450 = 50$

$my = 300 - 250 = 50$

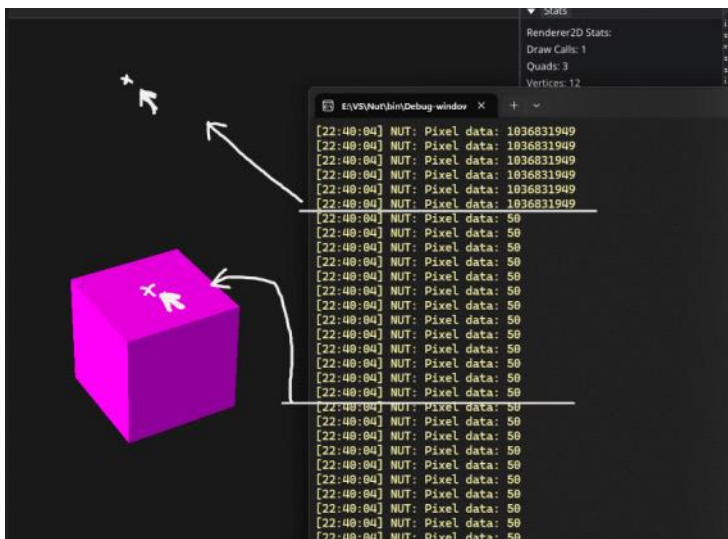
这表明，鼠标在窗口中的位置是 (50, 50)，即距离窗口左上角50像素的位置。

》》》一个小Bug：（白色小光标表示鼠标位置）

正确的操作：

```
// Confirm boundary values
ImVec2 windowSize = ImGui::GetWindowSize();
ImVec2 minBound = ImGui::GetWindowPos();
minBound.x += viewportOffset.x;
minBound.y += viewportOffset.y;
ImVec2 maxBound = { minBound.x + m_ViewportSize.x, minBound.y + m_ViewportSize.y };

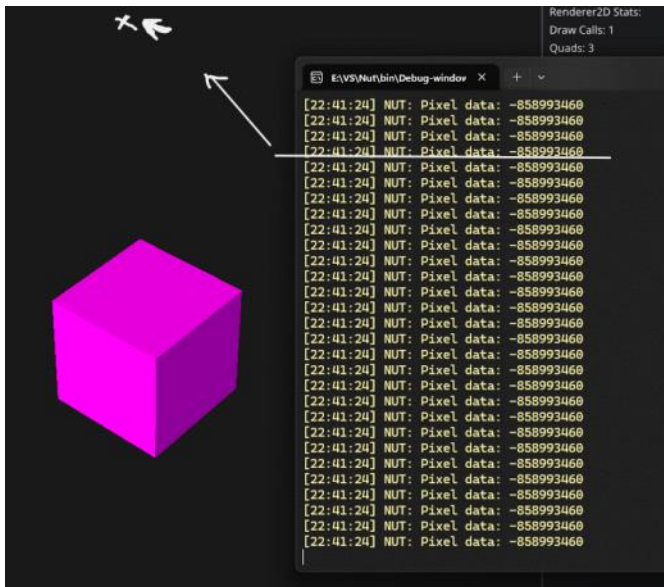
m_ViewportBounds[0] = { minBound.x, minBound.y };
m_ViewportBounds[1] = { maxBound.x, maxBound.y };
```



错误的操作：当鼠标在视口中，但是鼠标位于视口靠上方时，会出现无效值。

```
// Confirm boundary values
ImVec2 windowSize = ImGui::GetWindowSize();
ImVec2 minBound = ImGui::GetWindowPos();
minBound.x += viewportOffset.x;
minBound.y += viewportOffset.y;
ImVec2 maxBound = { minBound.x + windowSize.x, minBound.y + windowSize.y };

m_Viewports[0] = { minBound.x, minBound.y };
m_Viewports[1] = { maxBound.x, maxBound.y };
```



发生这个错误的原因是：ImGui::GetWindowSize() 获取的是整个窗口的尺寸，包括边框和标题栏等。而我们需要的是窗口的客户端区域尺寸不包括边框，否则在计算鼠标相对与窗口左上角的位置坐标时，我们获取的坐标 mouseX, mouseY 会发生偏移，想要解决这个问题，就应该使用 ImGui::GetAvailableRegion() 来获取窗口尺寸。

而 m_Viewports 恰好是这样获取的。

```
ImVec2 panelSize = ImGui::GetContentRegionAvail();
m_Viewports[0] = { panelSize.x, panelSize.y };
```

》》》疑问：

疑问1.0：为什么需要特别计算一个 viewportSize，以此作为判断依据呢？为什么不可以按照 maxBound 来判断呢？

```
// Read Pixels from attachment
glm::vec2 viewportSize = m_Viewports[1] - m_Viewports[0];

auto [mX, mY] = ImGui::GetMousePos();
mX -= m_Viewports[0].x;
mY -= m_Viewports[0].y;
mY = viewportSize.y - mY;

int mouseX = (int)mX;
int mouseY = (int)mY;

if(mouseX >= 0 && mouseY >= 0 && mouseX < (int)viewportSize.x && mouseY < (int)viewportSize.y)
{
    int pixelData = m_Framebuffer->ReadPixel(1, mouseX, mouseY);
    NUT_CORE_WARN("Pixel data: (0)", pixelData);
}
```

比如这样：

```
int mouseX = (int)mX;
int mouseY = (int)mY;

if(mouseX >= 0 && mouseY >= 0 && mouseX < m_Viewports[1].x && mouseY < m_Viewports[1].y)
{
    int pixelData = m_Framebuffer->ReadPixel(1, mouseX, mouseY);
    NUT_CORE_WARN("Pixel data: (0)", pixelData);
}
```

答：因为 m_Viewports[1] 中存储的 maxBound 是屏幕全局中的坐标，而这里我们获取的鼠标坐标是经过处理的视口相对坐标，所以如果需要进行条件判断，则必须按照 minBound 和 maxBound 计算出从 [0,0] 开始的整个视口的相对坐标，并将其作为判断依据。

疑问1.1：但是ReadPixel中也填写了mouseX, mouseY这两个相对坐标，这是否正确？

```
int pixelData = m_Framebuffer->ReadPixel(1, mouseX, mouseY);
```

正确，因为这个函数是使用在 Framebuffer 之下的，我们使用 mouseX 和 mouseY 来读取视口中的数据。虽然 mouseX, mouseY 并不是整个屏幕上的绝对坐标，但他们是窗口上的相对坐标。由于窗口完全被视口填充，且这个坐标的大小也被限制在视口的尺寸之内，所以这两个坐标可以表示视口（“Viewport”这个窗口）中的位置，因此这个函数参数的使用是正常的。

疑问2: 这里对于坐标进行整形转化的用意是什么? 不做会怎样?

```
// Read Pixels from attachment
glm::vec2 viewportSize = m_ViewportBounds[1] - m_ViewportBounds[0];

auto [mX, mY] = ImGui::GetMousePos();
mX -= m_ViewportBounds[0].x;
mY -= m_ViewportBounds[0].y;
mY = viewportSize.y - mY;

int mouseX = (int)mX;
int mouseY = (int)mY;

if(mouseX >= 0 && mouseY >= 0 && mouseX < (int)viewportSize.x && mouseY < (int)viewportSize.y)
{
    int pixelData = m_Framebuffer->ReadPixel(1, mouseX, mouseY);
    NUT_CORE_WARN("Pixel data: {0}", pixelData);
}
```

int mouseX = (int)mX; 和 int mouseY = (int)mY;	mX 和 mY 在前面是通过 ImGui::GetMousePos() 获取的鼠标位置，它们是浮点值（通常是 float 类型），表示鼠标在视口中的相对坐标。 因为图像或像素通常以整数坐标来进行处理，浮点值对它们没有直接的意义。所以通常需要将浮点型的鼠标坐标转换为整数，才能用于像素访问或图像读取等操作。
(int)viewportSize.x 和 (int)viewportSize.y	viewportSize.x 和 viewportSize.y 是 float 类型的值，但在进行鼠标坐标检查时，视口的宽度和高度应该是整数，这样才能与已经转换为整数的 mouseX 和 mouseY 进行比较，确保鼠标坐标在有效的范围内。

《 《 《 Tips: You can use (ImGui::GetForegroundDrawList()->AddRect(minBound, maxBound, IM_COL32(255, 255, 0, 255));) to draw the available region with colored rectangle

-----Clear texture attachments & Support multiple entity IDs-----

》 》 》 这一次我要提交两个部分：清除和多个实体ID（清除指的是将空白区域的鼠标输出改为：-1，多实体区域指的是每一个实体都将拥有一个ID，作为鼠标输出）

》 》 》 一些当下的理解：

当前，我们创建了一个帧缓冲对象，并且为其创建了两个颜色附件（纹理附件）、一个深度附件、一个模板附件。其中颜色附件分为：RGBA8、RED_INTEGER，RGBA8被用来绘制物体，RED_INTEGER被用来存储或处理实体ID。

```
m_Framebuffer = FrameBuffer::Create({ 1280, 720, 1, {FrameBufferAttachmentFormat::RGBA8, FrameBufferAttachmentFormat::RED_INTEGER, FrameBufferAttachmentFormat::Depth} });
```

而且由于我们使用了批渲染，所以理论上所有渲染的物体应该是一个整体，所以不用使用多个颜色附件。

在实际的代码操作中，我们会根据 FrameBuffer::Create 填入的参数顺序来分配附件索引，也就是说：

```
m_Framebuffer = FrameBuffer::Create({ 1280, 720, 1, {FrameBufferAttachmentFormat::RGBA8, FrameBufferAttachmentFormat::RED_INTEGER, FrameBufferAttachmentFormat::Depth} });
```

GL_ATTACHMENT0 GL_ATTACHMENT1 GL_ATTACHMENT2

如此一來，我们可以访问不同附件插槽上的数据，如果此时我们在着色器中设置了 color2，并将其作为 Attachment1，就可以读取到我们在着色器中对其设定的值。（此处示例为 color2 = 50）

```
32  
33     layout(location = 0) out vec4 color;  
34     layout(location = 1) out int color2;  
35  
void main()  
{  
    color = texture(u_Textures[i  
    //color = vec4(v_TexIndex, 0  
  
    color2 = 50;  
}
```

如果我们选择在特定情况下读取该附件的内容，并打印，才会有类似的效果。

```
m_ActiveScene->OnUpdateEditor(ts, m_EditorCamera); // Now we just update EditorCamera  
//m_ActiveScene->OnScript(ts); // 更新本机脚本  
  
// Read Pixels from attachment  
glm::vec2 viewportSize = m_ViewportBounds[1] - m_ViewportBounds[0];  
  
auto [mX, mY] = ImGui::GetMousePos();  
mX -= m_ViewportBounds[0].x;  
mY -= m_ViewportBounds[0].y;  
mY = viewportSize.y - mY;  
  
int mouseX = (int)mX;  
int mouseY = (int)mY;  
  
if(mouseX >= 0 && mouseY >= 0 && mouseX < (int)viewportSize.x && mouseY < (int)viewportSize.y)  
{  
    int pixelData = m_Framebuffer->ReadPixel(1, mouseX, mouseY);  
    NUT_CORE_WARN("Pixel data: {0}", pixelData);  
}
```

读取 ATTACHMENT1

但是只有鼠标停留在窗口中已绘制的物体上时，才可以访问到有效的 Attachment（如果鼠标停留在窗口内空白区域，此处没有绘制物体，而附件 Attachment 又恰恰是在绘制一个物体时才创建的，所以此时打印的值是无效数字：1036831949）

如果此时想清除空白区域的无效数字，则需要在访问所有区域之前，使用 ClearTexImage() 清空所有数据（使用 ClearTexImage() 将 Attachment0 + index 这里的数据临时更改为指定值：-1），这样一来当鼠标位于空白区域时，就会返回我们设置的值。（如果后续鼠标会位于物体上，我们只需要使用函数 ReadPixel() 重新读取 Attachment0 + index 存放的值即可。）

》》》关于 glClearTexImage()

作用：	是 OpenGL 4.5 引入的一个函数，它允许你直接清除纹理（texture）对象的内容（颜色、深度、模板或其他类型的数据），而不需要绑定纹理到帧缓冲区。这个函数提供了一种更高效的方式来清除纹理的内容，，避免了传统的方式（通过帧缓冲区绑定清除纹理）所带来的额外开销，这在图形渲染管线中尤其非常有用。
函数原型：	void glClearTexImage(GLuint texture, GLint level, GLenum format, GLenum type, const void *data);
参数说明：	1.texture (GLuint): 要清除的纹理对象的名称。纹理对象在创建时通过 glGenTextures 获取的 ID。该纹理对象必须是有效的。 2.level (GLint): 指定纹理的级别（level）。对于常规纹理，level 通常是 0，表示基础级别。对于多级渐远纹理（Mipmap textures），level 表示清除哪个 mipmap 级别的纹理。 3.format (GLenum): 指定清除时所使用的数据格式。常见的格式包括： GL_RED, GL_RG, GL_RGB, GL_RGBA 等。 GL_DEPTH_COMPONENT, GL_STENCIL_INDEX, GL_DEPTH_STENCIL 等用于深度和模板缓冲区的格式。 4.type (GLenum): 指定清除时所使用的数据类型。常见的数据类型包括： GL_UNSIGNED_BYTE, GL_FLOAT, GL_INT 等。 该类型应该与 format 兼容。 5.data (const void *): 指向一个数据缓冲区的指针，包含用于清除纹理的值。这个值的格式和类型必须与 format 和 type 相匹配。 常用用途：data 可能是一个填充了特定清除值的数组。例如对于 format = GL_RGBA 格式，可以将其设置为一个数组（例如：{0.0f, 0.0f, 0.0f, 1.0f}），以清除纹理为透明黑色。 在此处我们将 format = GL_RED_INTEGER, data = -1，表示我们将 GL_ATTACHMENT1（或其他位置）存储的 GL_RED_INTEGER 清除为-1

----- Unique Entity ID for mouse picking (Mouse picking) -----

》》》这一次提交的主要思路为：为顶点再添加一个属性 -> EntityID，这个 EntityID 会被实体默认生成的ID所填入，进而在顶点中标识某一个实体，我们也可以访问到。

具体思路是：

1. 确定新的顶点属性 layout(location = 5) in int a_EntityID
2. 根据这个新增的顶点属性，我们首先需要更改着色器中的对应语句。其次，还需要更改 VertexArray 的 SetLayout() 函数，以便我们能够正确的添加新增的顶点属性。最后，我们还需要更改实际的绘制函数，以便我们在绘制过程中保存对应的 entityID
- 3.调用更新后的绘制函数，为图像添加新的顶点属性
- 4.也可以通过读取出来的 EntityID 将实体信息放在 ImGui 窗口中实时查看

》》》我们在 OpenGL 上下文中访问的顶点属性数据（比如访问片段着色器中的输出变量 color2），此时这个变量存放在哪里？我们从哪里读取到这个数据？

前提：

着色器的使用阶段：	OpenGL 渲染管线分为多个阶段，其中顶点着色器和片段着色器是关键的两个阶段。
着色器的作用：	顶点着色器负责处理每个顶点的数据（如位置、颜色、纹理坐标等），而片段着色器则负责处理每个片段（即像素）的颜色和其它属性。

Eg.假设这里有一段着色器代码(gls)

```
Vertex shader
#version 330 core

layout(location = 0) in vec3 a_Position; // 顶点位置
layout(location = 1) in int a_EntityID; // 顶点的 EntityID（来自顶点缓冲区）

out flat int v_EntityID; // 传递给片段着色器的 EntityID

void main()
{
    gl_Position = vec4(a_Position, 1.0);
```

```
// 将 EntityID 从顶点着色器传递到片段着色器
v_EntityID = a_EntityID; }
```

```
Fragment shader
#version 330 core
```

```
// 接收来自顶点着色器的 EntityID
in flat int v_EntityID;
out vec4 FragColor;

void main()
{
    FragColor = v_EntityID;
}
```

首先让我们明确一下使用逻辑：

第一步，	我们在绘制的时候通过函数填入了 EntityID，随后通过代码中的 SetLayout() 明确 EntityID 的属性，并将其写入顶点缓冲区中。
第二步，	当顶点被写入后，在渲染管线阶段中，顶点着色器中的数据被传输给片段着色器，在片段着色器中我们也可以进行一些处理操作。
第三步，	我们在实际使用时，会使用函数读取片段着色器中的 FlagColor 变量，而 FlagColor = v_EntityID。

》此时我的问题是，在访问过程中，片段着色器中的 FlagColor 变量存储在哪里？因为 FlagColor = v_EntityID，我还想知道 v_EntityID 此时存放在哪里？

问题一：v_EntityID 存放在哪里？

传递过程：

当 OpenGL 上下文尝试访问 v_EntityID 时，是从顶点着色器传递来的。这是具体的传递过程：

顶点缓冲区存储：每个顶点有一个对应的 EntityID，通常通过顶点属性传入（比如通过 a_EntityID）。

顶点着色器处理：顶点着色器将 EntityID 从输入的顶点数据传递给片段着色器（通过 out 变量 v_EntityID）。

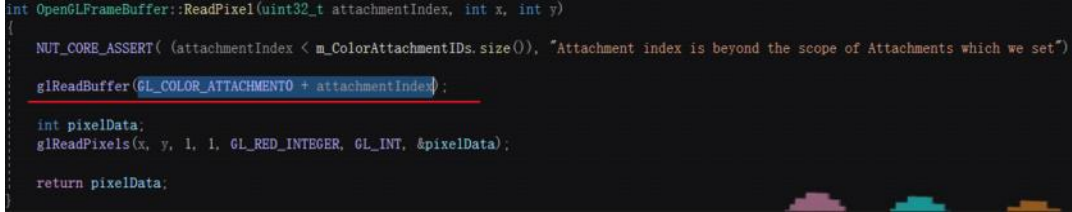
片段着色器访问：片段着色器通过 in 变量 v_EntityID 获取这个值，并根据它执行不同的渲染操作。

所以，v_EntityID 实际上是存储在 GPU 的顶点缓冲区中的，并在顶点着色器和片段着色器之间传递。

问题二：FlagColor (Color2)存放在哪里？

在大多数情况下，这些输出值被存储在与当前绑定的帧缓冲关联的颜色附件中，比如片段着色器的输出通常会存储在帧缓冲的一个 颜色附件（Color Attachment）中。

OpenGL 允许你绑定多个附件位置，我们可以通过 OpenGL 提供的宏常量：GL_ATTACHMENT0, GL_ATTACHMENT1，通过附件绑定的位置访问附件。



》》》GLSL 中的 flat 修饰词有什么作用？

EG.	in flat int v_EntityID;
概念：	flat 是一个修饰符，在 GLSL 中，它用于确保着色器中不同处理单元（比如不同的片段或像素）在使用此变量时不会进行插值。换句话说，使用 flat 关键字声明的变量在不同的计算单元中会保持相同的值，不进行插值，即每个计算单元直接使用传入的值。
兼容性：	插值操作与 int 类型不兼容 ，GLSL 的插值机制默认设计用于浮点类型数据。当使用 int 类型时，插值操作并不总是适合的，因为 int 是离散的，并且它不能像浮点数那样平滑过渡，所以如果你不加 flat 修饰符，着色器会尝试对 int 进行插值，这会导致不可预测的结果。 例如，在两个顶点之间，着色器可能会尝试对整数值进行插值，这显然是无意义的，因为整数类型值无法像浮动类型那样平滑过渡。

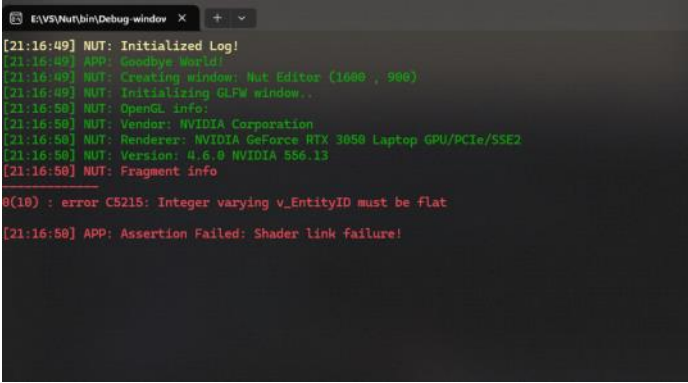
实际情况说明：

在没有 flat 修饰符的情况下，当你从顶点着色器向片段着色器传递数据时，数据会被**插值**。而插值是为了平滑过渡而设计的（通常应用于浮点数），所以，如果你希望 v_EntityID 在片段着色器中始终保持不变（不被插值），你需要使用 flat 修饰符来阻止插值。

例如，在光栅化过程中，可能有多个片段（像素）在不同的计算单元上处理，如果没有 flat 修饰符，v_EntityID 会进行插值操作，但由于 int 类型是离散的，插值可能会导致无效的值（例如，v_EntityID 在两个顶点之间的值不会是整数，可能导致错误）。此时使用 flat 可以避免这种插值，确保每个片段获得完全相同的值。

错误注意：

1.未明确使用 flat 关键字:
对于整型的输出变量, 如果没有明确使用 flat 修饰符, GLSL 会默认对 v_EntityID 进行插值。这可能会出现报错:



2. 使用 flat 时, 需要对着色器版本号进行更改:

```
2
3 #type vertex
4 #version 450 core
5

32
33 #type fragment
34 #version 450 core
35
```

更改后可能会出现警告: (再次运行一次程序便不会出现警告, 我猜想是因为着色器的版本号被改变了, 需要重新编译一次)

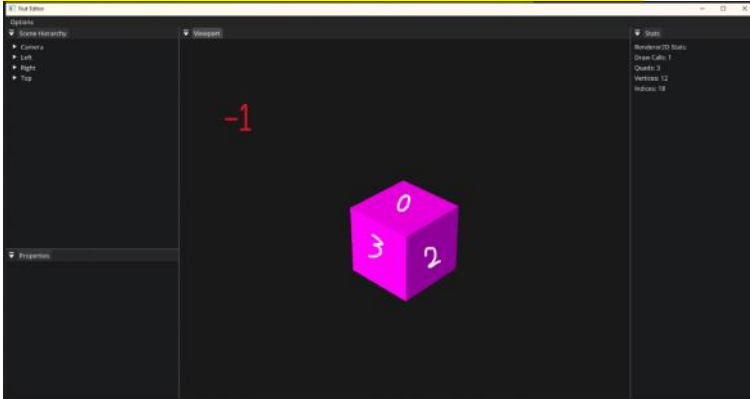
Program/shader state performance warning: Vertex shader in program 1 is being recompiled based on GL state.	这表明当前的顶点着色器 (在程序 1 中) 正在根据 OpenGL 的状态进行重新编译。_OpenGL 状态指的是一些全局的设置, 比如混合模式、视口尺寸、纹理绑定等。这些状态的改变可能导致需要重新编译着色器, 以确保它们能正确地与当前状态配合。
Program/shader state performance warning: Vertex shader in program 1 is being recompiled based on GL state, and was not found in the disk cache	这个部分表明, 重新编译的着色器没有找到预编译的缓存文件, 这可能意味着着色器的缓存没有被写入磁盘, 或者着色器程序本身没有缓存。每次重新编译会带来一定的性能开销, 尤其是在实时渲染中。

》》》glVertexAttribPointer() 有什么意义? 如何使用?

```
typedef void (APIENTRY PFNGLVERTEXATTRIBPOINTERPROC) (GLuint index, GLint size, GLenum type, GLsizei stride, const void *pointer);
GLAPI PFNGLVERTEXATTRIBPOINTERPROC glad_glVertexAttribPointer;
```

glVertexAttribPointer() 是 OpenGL 中的一个函数, 专用于设置整数类型的顶点属性指针。
这个函数的作用是告诉 OpenGL 如何在顶点缓冲区中读取顶点属性数据, 尤其是当这些数据是整数类型时。它类似于 glVertexAttribPointer(), 但专门处理整数类型数据, 而不涉及浮点数据。

》》》运行之后, 我注意到一个问题打印出来的 EntityID 为什么不是 0,1,2 而是 0,2,3 ?



这是因为我们在 yaml 文件中存储的实体顺序是这样的 (我们不仅存储了三个物体实体, 还存储了一个摄像机实体, 而这个摄像机实体由于一些原因被放置在第二个实体的位置上)

```

Entities:
- Entity: 256257383941
  TagComponent:
    Tag: Top ID:0
  TransformComponent:
    Translation: [0, 0.5, 0]
    Rotation: [1.57079637, -0.785398185, 0]
    Scale: [0.99999994, 0.999999642, 0.999999881]
  SpriteComponent:
    Color: [0.876447856, 0, 0.834712803, 1]
- Entity: 256257383941
  TagComponent:
    Tag: Camera ID:1
  TransformComponent:
    Translation: [0, 1.70000005, 4]
    Rotation: [-0.404916406, 0, 0]
    Scale: [1, 1, 1]
  CameraComponent:
    Camera:
      ProjectionType: 0
      PerspectiveFOV: 0.52359879
      PerspectiveNear: 0.00999999978
      PerspectiveFar: 1000
      OrthographicSize: 10
      OrthographicNear: -1
      OrthographicFar: 1
    Primary: true
    FixedAspectRatio: false
- Entity: 256257383941
  TagComponent:
    Tag: Right ID:2
  TransformComponent:
    Translation: [0.351999998, 0, 0.349999994]
    Rotation: [0, 0.785398185, 0]
    Scale: [1, 1, 1]
  SpriteComponent:
    Color: [0.54842025, 0, 0.586872578, 1]
- Entity: 256257383941
  TagComponent:
    Tag: Left ID:3
  TransformComponent:
    Translation: [-0.354999989, 0, 0.349999994]

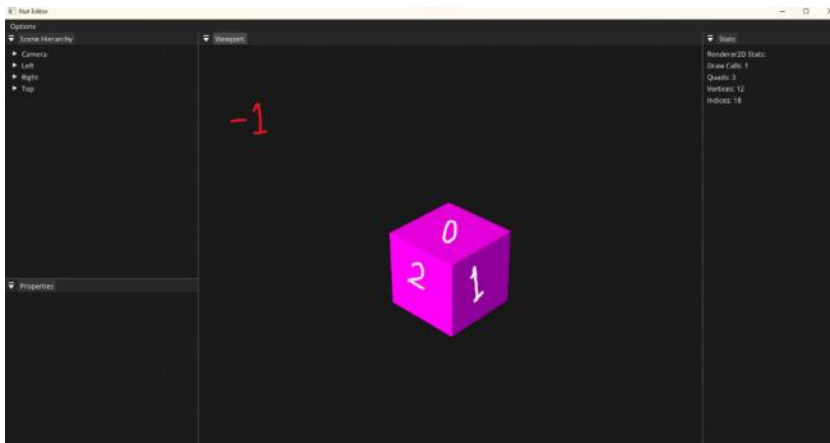
```

》而我们只需要在文件中调整几个实体的位置即可实现 0,1,2 的效果:

```

Scene: Untitled
Entities:
- Entity: 256257383941
  TagComponent:
    Tag: Top 0
  TransformComponent:
    Translation: [0, 0.5, 0]
    Rotation: [1.57079637, -0.785398185, 0]
    Scale: [0.99999994, 0.999999642, 0.999999881]
  SpriteComponent:
    Color: [0.876447856, 0, 0.834712803, 1]
- Entity: 256257383941
  TagComponent:
    Tag: Right 1
  TransformComponent:
    Translation: [0.351999998, 0, 0.349999994]
    Rotation: [0, 0.785398185, 0]
    Scale: [1, 1, 1]
  SpriteComponent:
    Color: [0.54842025, 0, 0.586872578, 1]
- Entity: 256257383941
  TagComponent:
    Tag: Left 2
  TransformComponent:
    Translation: [-0.354999989, 0, 0.349999994]
    Rotation: [0, -0.785398185, 0]
    Scale: [1, 1, 1]
  SpriteComponent:
    Color: [1, 0, 0.949807167, 1]
- Entity: 256257383941
  TagComponent:
    Tag: Camera 3
  TransformComponent:
    Translation: [0, 1.70000005, 4]
    Rotation: [-0.404916406, 0, 0]
    Scale: [1, 1, 1]
  CameraComponent:

```



》》》使用思路：这句代码是怎样实现对应功能的？

```
m_HoveredEntity = pixelData == -1 ? Entity() : Entity((entt::entity)pixelData, m_ActiveScene.get());
```

前提：在使用 Ctrl + O 之后，会调用一个函数 --> OpenScene(); 这个函数将会继续调用 --> serializer.Deserialize(filepath); 在 Deserializer() 中，有这两句代码：

```
Entity& deserializedEntity = m_Scene->CreateEntity(name); // Create a new entity in m_Scene with all default values
DeserializeEntity(entity, deserializedEntity); // Update values in this entity according to yaml file
```

也就是说，OpenScene() 函数会为我们创建实体，通过调整实体的属性，我们得以在引擎中查看文件的内容。

理解：所以在我们打开某个文件之后，画面中绘制的所有物体，其实就是我们创建的实体（这些实体在 OpenScene() 中被创建，并且调整了组件中的属性:Translate, Color）。如果此时我们在特定情况下，使用相同的数据再次创建一个新实体（Entity 类型对象），就可以确保新实体是对应旧实体的副本。

Eg.

Ctrl + O 打开文件，最终会运行此函数。其中高亮的代码部分会确保我们在读取文件之后，创建 YAML 文件中的实体

```
bool SceneSerializer::Deserialize(const std::string& filepath)
{
    // Read file, and send it to string stream
    std::ifstream file(filepath);
    std::stringstream fileContent;
    fileContent << file.rdbuf();

    // Load file contents and initialize data
    YAML::Node data = YAML::Load(fileContent);
    if (!data["Scene"])
        return false;

    // Trace this process
    std::string sceneName = data["Scene"].as<std::string>();
    NUT_CORE_TRACE("Deserializing scene ' {} '", sceneName);

    // According to data, we reproduce the scene
    auto entities = data["Entities"];
    if(entities)
    {
        for (auto entity : entities)
        {
            uint64_t uuid;
            std::string name;

            uuid = entity["Entity"].as<uint64_t>(); // TODO
            // Enter the TagComponent map,
            // and search for tag in submap(submap is stored in TagComponent map)
            auto tc = entity["TagComponent"];
            if (tc)
                name = tc["Tag"].as<std::string>();

            NUT_CORE_TRACE("Deserialized entity with ID = {0}, name = {1}", uuid, name);
            Entity& deserializedEntity = m_Scene->CreateEntity(name); // Create
            DeserializeEntity(entity, deserializedEntity); // Update
        }
    }
}
```

CreateEntity() 会创建实体，m_Registry.create() 会自动分配 entityID，这个ID是唯一的。

```
Entity Scene::CreateEntity(const std::string& name)
{
    Entity entity = { m_Registry.create(), this };
}
```

这时，比如在代码中，如果鼠标光标在视口范围内、且悬停在物体上方，我们就创建实体：HoveredEntity

```

if(mouseX >= 0 && mouseY >= 0 && mouseX < (int)viewportSize.x && mouseY < (int)viewportSize.y)
{
    int pixelData = m_Framebuffer->ReadPixel(1, mouseX, mouseY);
    NUT_CORE_WARN("Pixel data: {0}", pixelData);

    if (pixelData != -1 && m_HoveredEntity != Entity((entt::entity)pixelData, m_ActiveScene.get()))
        m_HoveredEntity = Entity((entt::entity)pixelData, m_ActiveScene.get());
    else if (pixelData == -1)
        m_HoveredEntity = Entity();

    //m_HoveredEntity = pixelData == -1 ? Entity() : Entity((entt::entity)pixelData, m_ActiveScene.get());
}

```

由于我们在 DrawQuad() 这个绘制函数中，将 EntityID(EntityHandle)传入了顶点着色器，所以通过 ReadPixel() 我们可以读取对应的 EntityID。并使用这个 EntityID(pixelData)创建一个实体。

那么这个实体 和 打开YAML文件时创建的实体 有什么联系呢？为什么我们可以通过 HoveredEntity 使用 GetTagcomponent(), 访问到先前实体的数据？

因为 Deserializer() 中创建的实体，是通过 EntityID(m_Registry.Create()) 和 Scene(this 即 m_ActiveScene)创建的，这两个参数都是唯一的标识，我们可以通过这样的唯一标识访问组件。

```

template<typename T>
T& GetComponent()
{
    NUT_CORE_ASSERT(HasComponent<T>(), "This Entity does not have component!");
    return m_Scene->m_Registry.get<T>(m_EntityHandle);
}

```

如果此时我们又新建了一个实体，比如HoveredEntity，并使用了相同的参数将其初始化，这样我们就拥有了先前实体的一个副本，这个副本就是 HoveredEntity。由于 HoveredEntity 中保存着这些唯一标识，我们当然可以使用 GetComponent() 访问到对应的组件，并且组件的内容 等同于 先前实体组件中的内容。

》》》代码中的 OpenGL 版本号管理代码，在哪里？

在代码中并没有明确表示OpenGL版本号(比如通过 GLFWWindowHint() 设置版本号)，所以GLFW 会检测你的图形卡及其驱动程序支持的 OpenGL 版本。如果没有明确指定版本，GLFW 会选择一个合适的默认版本，通常是当前系统支持的最新版本。

```

NUT_CORE_INFO("Vendor: {0}", (const char*)glGetString(GL_VENDOR));
NUT_CORE_INFO("Renderer: {0}", (const char*)glGetString(GL_RENDERER));
NUT_CORE_INFO("Version: {0}", (const char*)glGetString(GL_VERSION));

```

本机为：

```

[14:06:21] NUT: Version: 4.6.0 NVIDIA 556.13
[14:06:21] NUT: Driver: 460.39

```

》》》我对Cherno的代码做了一点更改

```

m_HoveredEntity = pixelData == -1 ? Entity() : Entity((entt::entity)pixelData, m_ActiveScene.get());

```

在这里，我觉得满足条件判断之后，会一直调用构造函数，可能比较耗费性能。

```

112     if(mouseX >= 0 && mouseY >= 0 && mouseX < (int)viewportSize.x && mouseY < (int)viewportSize.y)
113     {
464 (9.02%) 114         int pixelData = m_Framebuffer->ReadPixel(1, mouseX, mouseY);
115         NUT_CORE_WARN("Pixel data: {0}", pixelData);
116
117         if (pixelData != -1)
118             m_HoveredEntity = Entity((entt::entity)pixelData, m_ActiveScene.get());
119         else if (pixelData == -1)
120             m_HoveredEntity = Entity();
121     }
122
29 (0.56%) 123     m_Framebuffer->Unbind();
124

```

更改之后：

```

112     if(mouseX >= 0 && mouseY >= 0 && mouseX < (int)viewportSize.x && mouseY < (int)viewportSize.y)
113     {
478 (5.89%) 114         int pixelData = m_Framebuffer->ReadPixel(1, mouseX, mouseY);
83 (1.02%) 115         NUT_CORE_WARN("Pixel data: {0}", pixelData);
116
117         if (pixelData != -1 && m_HoveredEntity != Entity((entt::entity)pixelData, m_ActiveScene.get()))
118             m_HoveredEntity = Entity((entt::entity)pixelData, m_ActiveScene.get());
119         else if (pixelData == -1 && m_HoveredEntity != Entity())
120             m_HoveredEntity = Entity();
121     }
122
50 (0.62%) 123     m_Framebuffer->Unbind();
124
105 (1.29%) 125 }

```

不过从数据分析上看起来，虽然优化了一部分性能，但是本来好像就没有很大的性能负担。:-)

----- Left click to select entities -----

»»»»

----- Maintenace -----

»»»» 后续会整合 Jul/22 2020 之后的维护