GameEngine5

2024年7月12日 15:06

----------一些维护和更改----------

》》》》 Made Win-GenProjects.bat work from every directory

代码更改：

```
@echo off
->pushd ..\
->pushd %~dp0\..\
call vendor\bin\premake\premake5.exe vs2019
popd
PAUSE
```

为什么要做这样的更改?

当你通过命令行提示符打开并运行该文件时，这个批处理文件会在命令行提示符被打开的位置被运行，这会导致 pushd ..\这个语句原本的语义错误。（在命令行提示符的路径下临时切换工作目录至相对于命令提示符的上一级 '..\'，而不是相较于批处理文件的上一级）

为了避免这样的情况发生，我们需要将启用该批处理文件后的语句改为绝对的、固定的切换目录操作。

%~dp0 的含义：

- %0 是批处理文件本身的名称。
- %~dp0 是批处理文件的完整路径，包括驱动器号和路径。它扩展为批处理文件所在的目录。这个路径在批处理文件内部是固定的，不会改变。

%~dp0\ 有什么意义?

pushd ..\	是将当前目录更改为上一级目录。
pushd %~dp0\..\	是将当前目录更改为批处理文件所在目录的上一级目录（绝对路径）。

Eg.

```
@echo off
echo this is %%cd%% %cd%
echo this is %%~dp0 %~dp0
```

当你在其他目录（比如 C:\）运行这个批处理文件时，这两个路径会不同

%cd%	显示的是当前目录
%~dp0	显示的是批处理文件所在的目录

》》》》 Fix Build warnings to do with BufferElement Offset Type

WIN64	size_t =>	unsigned __int64	intptr_t =>	__int64
ELSE	size_t =>	unsigned int	intptr_t =>	int
In Any Case	Uint32_t =>	unsigned int		

(const void*)(intptr_t)element.Offset 中，Offset 仅仅是一个数值，没有任何与内存地址相关的含义。在这种情况下，intptr_t 的转换也是不必要的，因为它不会改变你正在做的事情的本质。

》》》》 Basic ref-counting system to terminate glfw

之前在 WindowsWindow.cpp 中，仅仅判断了GLFW窗口是否已经初始化？是否需要初始化上下文？然后根据判断结果进行 glfwDestroyWindow(m_Window); 这仅仅只是销毁了窗口。

而且我们并没有通过 `glfwTerminate()`; 函数对GLFW 库进行终止，并释放资源。

这一次，我们判断打开的 GLFW 窗口是否全部关闭，如果全部关闭，则对 GLFW 库进行终止。若对一个窗口关闭之后，仍有正在运行的窗口，则仅销毁需要关闭的窗口即可。

Uint8_t 的定义：	typedef unsigned char uint8_t;

》》》》 Added file reading check

```
std::string OpenGLShader::ReadFile(const std::string& filepath)
{
    std::string result;
    std::ifstream readIn(filepath, std::ios::in | std::ios::binary);
    if (readIn)                                 //是否成功打开
    {
        readIn.seekg(0, std::ios::end);
        size_t size = readIn.tellg();
        if (size != -1) {                       //是否成功获取
            ...
        }
        else { NUT_CORE_ERROR("Failed to read file from : '{0}'", filepath); }
    }
    else { NUT_CORE_ERROR("Could not open file form : '{0}'", filepath); }
}
```

》》》》 Code maintenance (#165)

构造函数 A() = {} 和 A() = default 有什么区别吗?

- 实现上：

A() = {}	是显式手动定义一个空的默认构造函数，不依赖于编译器生成。
A() = default	是告诉让编译器来生成默认构造函数。

- 性能上：
  这两种方式行为相同，运行时也生成相同的机器码。因此，在生成的代码执行效率上，不会有显著的区别。

- 结论：二者没有什么区别。

》》》》 x64 和 x86_64 有什么区别吗?

在大多数情况下，"x64" 和 "x86_64" 可以互换使用，都是用来描述64位操作系统和处理器架构的术语。表示支持64位操作系统和处理器架构的环境，可以看做时同义词。

"x86_64"	在一些技术文档和Linux系统中更常见
"x64"	则在Windows系统中更为普遍。两者本质上指向同一种64位架构，即AMD64或x86-64。

》》》》 Auto deducing an availabe __FUNCSIG__ definition (#174)

Created `HZ_FUNC_SIG` macro to deduce a valid pretty function name macro as `__FUNCSIG__` isn't available on all compilers

1. **#if defined(__GNUC__) || (defined(__MWERKS__) && (__MWERKS__ >= 0x3000)) || (defined(__ICC) && (__ICC >= 600)) || defined(__ghs__)**

- 这个条件判断首先检查是否定义了 __GNUC__，这是 GNU 编译器的宏。如果定义了，说明正在使用 GCC 编译器。
- 第二部分 (defined(__MWERKS__) && (__MWERKS__ >= 0x3000)) 检查 Metrowerks CodeWarrior 编译器版本是否大于等于 0x3000。
- 第三部分 (defined(__ICC) && (__ICC >= 600)) 检查 Intel C/C++ 编译器版本是否大于等于 600。
- 最后一个条件 defined(__ghs__) 检查是否使用 Green Hills 编译器。
- 如果任何一个条件为真，表示正在使用对应的编译器，因此选择 __PRETTY_FUNCTION__ 作为 NUT_FUNC_SIG 的值。__PRETTY_FUNCTION__ 是 GCC 和一些兼容的编译器提供的宏，用于获取带有类型信息的函数签名。

2. **#elif defined(__DMC__) && (__DMC__ >= 0x810)**

- 这个条件检查是否定义了 __DMC__ 并且版本号大于等于 0x810，表示使用 Digital Mars C/C++ 编译器。
- 如果条件成立，则选择 __PRETTY_FUNCTION__。

3. **#elif defined(__FUNCSIG__)**

- 这个条件检查是否定义了 __FUNCSIG__，这是 Microsoft Visual C++ 提供的宏，用于获取包含返回类型的函数签名。
- 如果条件成立，则选择 __FUNCSIG__ 作为 NUT_FUNC_SIG 的值。

4. **#elif (defined(__INTEL_COMPILER) && (__INTEL_COMPILER >= 600)) || (defined(__IBMCPP__) && (__IBMCPP__ >= 500))**

- 这个条件首先检查是否定义了 __INTEL_COMPILER 并且版本号大于等于 600，表示使用 Intel C++ Compiler。
- 第二部分 (defined(__IBMCPP__) && (__IBMCPP__ >= 500)) 检查 IBM XL C/C++ 编译器版本是否大于等于 500。
- 如果任何一个条件成立，则选择 __FUNCTION__ 作为 NUT_FUNC_SIG 的值。__FUNCTION__ 是一个标准 C99 定义的宏，用于获取简单函数名。

5. **#elif defined(__BORLANDC__) && (__BORLANDC__ >= 0x550)**

- 这个条件检查是否定义了 __BORLANDC__ 并且版本号大于等于 0x550，表示使用 Borland C++ 编译器。
- 如果条件成立，则选择 __FUNC__ 作为 NUT_FUNC_SIG 的值。

6. **#elif defined(__STDC_VERSION__) && (__STDC_VERSION__ >= 199901)**

- 这个条件检查是否定义了 __STDC_VERSION__ 并且版本号大于等于 199901，表示当前编译器支持 C99 标准。
- 如果条件成立，则选择 __func__ 作为 NUT_FUNC_SIG 的值。__func__ 是 C99 标准引入的标准宏，用于获取简单函数名。

7. **#elif defined(__cplusplus) && (__cplusplus >= 201103)**

- 这个条件检查是否定义了 __cplusplus 并且版本号大于等于 201103，表示当前编译器支持 C++11 标准。
- 如果条件成立，则同样选择 __func__ 作为 NUT_FUNC_SIG 的值。C++11 引入了对 __func__ 宏的支持。

8. **#else**

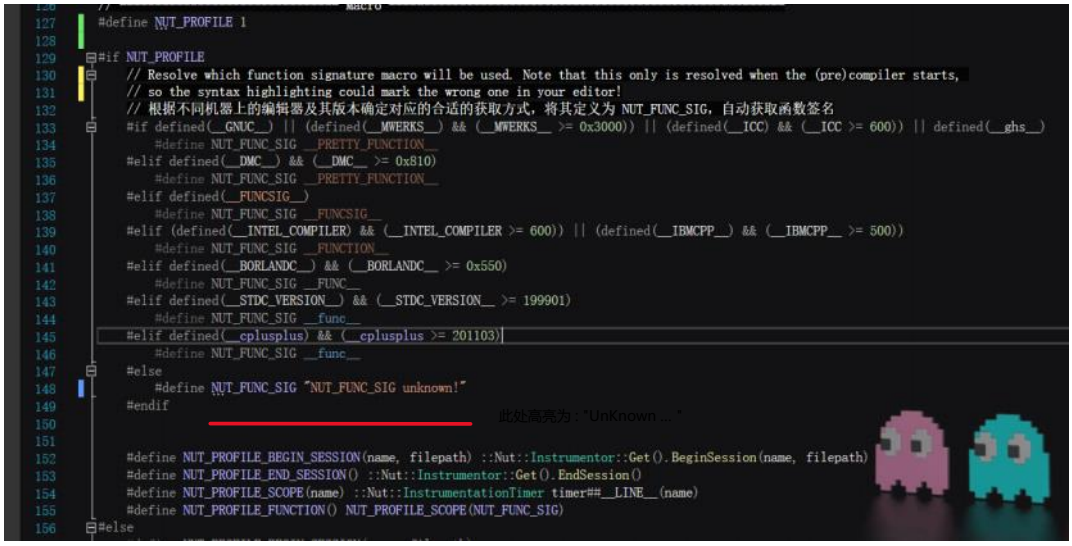
- 如果以上所有条件都不满足，则选择 "NUT_FUNC_SIG unknown!" 作为 NUT_FUNC_SIG 的默认值。这种情况下，编译器可能不支持预定义的函数签名获取方式，或者无法识别当前的编译环境。

```
// Resolve which function signature macro will be used. Note that this only
// is resolved when the (pre)compiler starts, so the syntax highlighting
// could mark the wrong one in your editor!
#if defined(__GNUC__) || (defined(__MWERKS__) && (__MWERKS__ >= 0x3000)) || (defined(__ICC) && (__ICC >= 600)) || defined(__ghs__)
    #define NUT_FUNC_SIG __PRETTY_FUNCTION__
#elif defined(__DMC__) && (__DMC__ >= 0x810)
    #define NUT_FUNC_SIG __PRETTY_FUNCTION__
#elif defined(__FUNCSIG__)
    #define NUT_FUNC_SIG __FUNCSIG__
#elif (defined(__INTEL_COMPILER) && (__INTEL_COMPILER >= 600)) || (defined(__IBMCPP__) && (__IBMCPP__ >= 500))
    #define NUT_FUNC_SIG __FUNCTION__
#elif defined(__BORLANDC__) && (__BORLANDC__ >= 0x550)
    #define NUT_FUNC_SIG __FUNC__
#elif defined(__STDC_VERSION__) && (__STDC_VERSION__ >= 199901)
    #define NUT_FUNC_SIG __func__
#elif defined(__cplusplus) && (__cplusplus >= 201103)
    #define NUT_FUNC_SIG __func__
#else
    #define NUT_FUNC_SIG "NUT_FUNC_SIG unknown!"
#endif

#define NUT_PROFILE_FUNCTION() NUT_PROFILE_SCOPE(NUT_FUNC_SIG)
```

关于注释：
Resolve which function signature macro will be used. Note that this only is resolved when the (pre)compiler starts, so the syntax highlighting could mark the wrong one in your editor!
意为Visual Studio 编辑器中的高亮显示可能是错误的结果，但这并不影响实际使用。

理解：
出现错误的原因是编辑器的语法高亮功能通常不能处理复杂的预处理宏选择逻辑，这种情况在使用条件编译和宏定义较多的情况下是比较常见的。
但这并不影响代码编译，因为这些定义将在运行时才被确定。



》》》》make runloop only accessible on the engine side (#188)

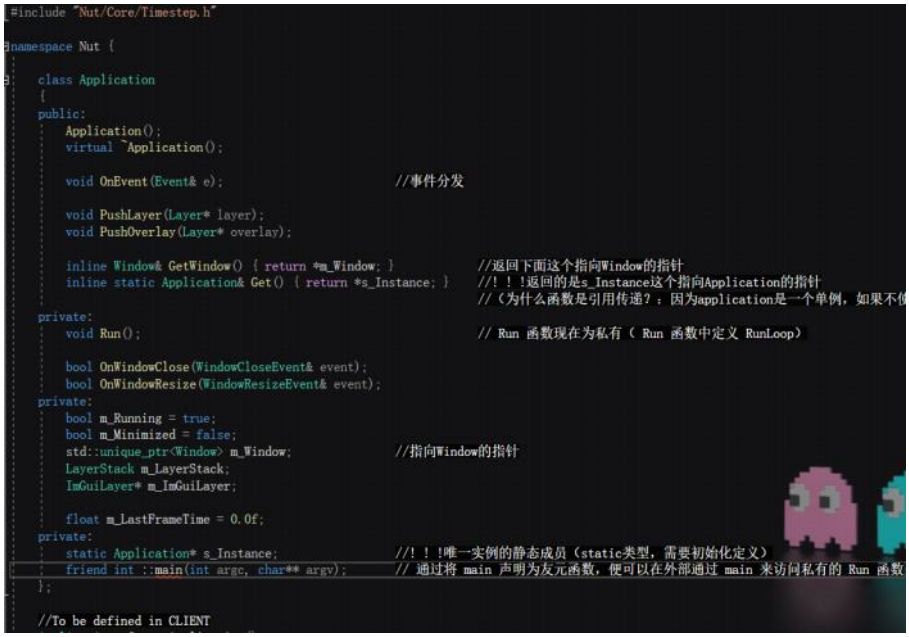
如何限制只在入口点（引擎端）访问 Run 函数（内含RunLoop的函数) 呢?
由于我们将主函数 main 设置在 EntryPoint 文件中，并在此处定义，所以想要只能在此处访问 Run 函数的话，我们需要将 Run 函数作为 Application 类中的私有成员函数，并将 main 函数作为 Application 类的友元函数。

所以我们在 Application 中将 Run 作为 Private，然后在 Private 中声明友元 friend int ::main(int argc, char** argv);
这引申出两个疑问:

- 为什么需要在友元声明中使用 ::main 而不是直接写 main 来表示友元函数?
- 为什么需要在全局空间中再次声明一次int main(int argc, char** argv);

1. 由于 Main 函数只被定义在外部文件中，如果在类内部声明为 friend int main(int argc, char** argv); 而没有使用 :: 指明它在全局作用域中，编译器可能会将其解释为当前编译单元内的 main 函数，而不是全局作用域的 main 函数。
这确保编译器正确理解友元函数的全局位置。

2. 如果不声明 int main 的话，在接下来使用友元函数的时候，会报错全局范围内没有 main。
在C++中，如果在一个类内声明了某个函数为友元函数，但是该函数的定义（或者至少声明）不在类声明之前全局范围内，编译器可能会报错。
如果在全局作用域中并没有先声明 int main(int argc, char** argv);，编译器可能会报错，因为在 friend 声明时需要指定 main 函数的存在，以便理解它是一个全局函数并将其声明为友元。
这个声明确保编译器理解 main 函数在全局范围内是存在的



》》关于 ++ 操作符的理解

- 在一般的 for 循环中，for(size_t i = 0; i <= x; i++), i++ 和 ++i 没什么不同。因为在循环条件中，只需检查 i 的当前值是否满足条件，无论是前置递增还是后置递增，条件的判断都是基于 i 的当前值。(在这个 for 循环中，i 起始值为 0，每次循环迭代结束后，i 会递增。循环条件 i <= x 每次循环迭代开始时都会被检查，如果条件为真，则执行循环体，然后执行递增操作 i++。这意味着 i 的值会在每次循环的末尾增加 1。)
甚至说，++i 还要比 i++性能/效率更高,因为 ++i 传递的不是副本，而 i++ 传递的是副本->一个临时对象。
- 在涉及到迭代器的增加操作时，**++iter 和 iter++ 有着微妙但重要的区别**

1. ++iter（前置递增操作符）

++iter	前置递增操作符，它的作用是先增加迭代器的值，然后返回增加后的迭代器。
行为：	++iter 会将迭代器 iter 指向的位置向前移动一个元素。
返回值：	返回的是增加后的迭代器 iter 的引用（即新的迭代器对象本身）。

在实际使用中，++iter 的返回值可以用于链式操作或者作为函数参数，因为它返回的是一个引用。

2. iter++（后置递增操作符）

iter++	后置递增操作符，它的行为略有不同：
行为：	iter++ 会返回迭代器 iter 的当前值，然后再将迭代器 iter 向前移动一个元素。
返回值：	返回的是增加前的迭代器 iter 的值（即旧的迭代器对象的副本），而不是增加后的迭代器。

这意味着 iter++ 在使用时，返回的值是旧位置的迭代器，不能直接作为函数参数或者链式操作的一部分，因为它的返回值是一个临时对象。

```
std::vector<int> vec = {1, 2, 3, 4, 5};

// 前置递增操作
auto it1 = vec.begin();
 auto incremented1 = ++it1; // 先增加，再返回
std::cout << *incremented1 << std::endl; // 输出 2

// 后置递增操作
auto it2 = vec.begin();
auto incremented2 = it2++; // 先返回，再增加
std::cout << *incremented2 << std::endl; // 输出 1
```

也就是说，在反向迭代中：

前置：

```
for (auto it = vec.rbegin(); it != vec.rend(); ++it)
 {
       std::cout << *it << " ";
}
```

每次循环体执行完成后，it 被递增，并且 *it 总是获取当前迭代器指向的元素值。在第二次循环开始前，it 先自增一次，然后用于使用。

后置：

```
for (auto it = vec.rbegin(); it != vec.rend(); it++)
{
      std::cout << *it << " ";
}
```

每次循环体执行完成后，it 被递增。这意味着在下一次迭代开始之前，it 仍然指向上一次迭代结束时的位置。因此，*it 取得的是上一次循环中的元素值，而不是当前迭代所需的元素值。
这在反向迭代器中尤其容易出现问题

》》》》什么是互斥锁？互斥锁和并行的关系是什么？

互斥锁（Mutex，互斥体）是一种用于多线程编程中的同步原语，用于确保在任何时刻，只有一个线程能够访问共享资源或临界区域，从而避免多个线程同时修改数据导致的不一致或竞态条件问题。
互斥锁提供两个主要操作：

- **锁定（Locking）**：当一个线程希望进入临界区域（访问共享资源）时，它会尝试获取互斥锁。

如果互斥锁当前未被其他线程占用（未锁定）	那么这个线程会成功获取锁，并进入临界区域。
如果互斥锁已经被其他线程占用（已锁定）	则当前线程可能会被阻塞，直到互斥锁被释放。

- **解锁（Unlocking）**：当一个线程使用完临界区域中的共享资源后，它会释放互斥锁，这样其他线程就有机会获取锁并继续执行。

结论：
互斥锁通常是基于硬件的原子操作（不可中断的操作）或操作系统提供的原语实现的，因此在锁定和解锁操作中能够保证线程安全，避免竞态条件。

互斥锁和并行的关系：
并行指的是多个线程或进程同时执行任务，通常在多核处理器或分布式系统中实现。
当多个线程或进程同时访问共享资源时，由于执行顺序不确定或未经同步导致的不正确的结果，称为竞态条件。
而互斥锁用于解决竞态条件，确保在任何时刻只有一个线程可以访问共享资源或临界区域。

》》》》互斥锁的使用

互斥锁的获取和释放：	使用 std::mutex 的 lock() 和 unlock() 方法来获取和释放互斥锁。

- 获取锁：调用 lock() 方法来获取互斥锁。
  如果当前没有其他线程持有锁，则当前线程获取锁并继续执行；
  如果其他线程已经持有锁，当前线程将被阻塞，直到获取到锁为止。
- 释放锁：调用 unlock() 方法来释放互斥锁，允许其他线程获取锁进入临界区。

```
m_Mutex.lock();
// 临界区代码，访问共享资源
m_Mutex.unlock();
```

使用 RAII 管理锁（推荐）：	RAII（资源获取即初始化）是一种管理资源生命周期的常用技术 在 C++ 中，可以使用 std::lock_guard 或 std::unique_lock 类来管理 std::mutex 的锁。

- `std::lock_guard`：

```
{
    std::lock_guard<std::mutex> lock(m_Mutex);
    // 临界区代码，访问共享资源
}   // 锁在此处自动释放
```

- `std::unique_lock`（更灵活，可以手动释放锁）：

```
{
    std::unique_lock<std::mutex> lock(m_Mutex);
    // 临界区代码，访问共享资源
```

```
void BeginSession
(const std::string& name, const std::string& filepath = "XXX")
```

- std::unique_lock（更灵活，可以手动释放锁）：

```
{
    std::unique_lock<std::mutex> lock(m_Mutex);
    // 临界区代码，访问共享资源
    lock.unlock(); // 手动释放锁
    // 其他非临界区代码
}   // 锁在此处自动释放
```

避免死锁：	在使用多个互斥量时，必须小心避免死锁（Deadlock）。死锁是指两个或多个线程互相等待对方持有的资源而无法继续执行的情况。

》》》》在代码中的RAII管理所为什么要放置于特定的地方，有什么考究?

BeginSession 函数：

- 在开始一个新会话时，需要确保对 m_CurrentSession 和 m_OutputStream 的操作是原子的，不被其他线程打断。
  std::lock_guard 在这里用来锁定 m_Mutex，确保在开始新会话时，只有一个线程能够执行这段代码，避免多线程同时进行会话操作导致的混乱。

EndSession 函数：

- 类似地，结束会话时也需要对 m_CurrentSession 和 m_OutputStream 进行操作，同样需要保证操作的原子性和线程安全性。

WriteProfile 函数：

- 在写入性能分析结果时，同样需要确保写操作的线程安全性，避免多个线程同时向输出流写入数据导致的混乱。
  std::lock_guard 在这里锁定 m_Mutex，确保每次写操作是串行执行的，不会被其他线程中断。

```
void BeginSession
(const std::string& name, const std::string& filepath = "XXX")
{
    std::lock_guard lock(m_Mutex);
    if (m_CurrentSession) {
        if (Log::GetCoreLogger()) {
            HZ_CORE_ERROR("....", name, m_CurrentSession->Name);
        }
        InternalEndSession();
    }
    m_OutputStream.open(filepath);
    if (m_OutputStream.is_open()) {
    ...
    } else {
    ...
    }
}

void EndSession()
{
    std::lock_guard lock(m_Mutex);
    InternalEndSession();
}

void WriteProfile(const ProfileResult& result)
{
    std::stringstream json;

    std::string name = result.Name;
    std::replace(name.begin(), name.end(), '"', '\'');

    Json<<...
    std::lock_guard lock(m_Mutex);
    if (m_CurrentSession) {
        m_OutputStream << json.str();
        m_OutputStream.flush();
    }
}
```

》》》为什么说写操作是串行的?

串行：	每个线程依次获取锁，执行完操作后释放锁，然后其他线程才能获取锁执行操作

在每次对 m_CurrentSession 和 m_OutputStream 进行写操作时，只有一个线程能够持有 m_Mutex，其他线程必须等待。这种方式保证了线程和数据的安全。

》》》》std::lock_guard 怎样使用

- 定义互斥锁对象（互斥量）
  `std::mutex m_Mutex; // 定义一个互斥量对象`
- 在对共享资源进行读写操作之前创建一个 `std::lock_guard` 对象，以确保在操作期间其他线程无法访问共享资源。

```
// 锁定互斥量，只有这个代码块中可以访问被保护的资源
{
std::lock_guard<std::mutex> lock(m_Mutex);

// 在这里可以安全地访问共享资源
// 例如：m_CurrentSession 或 m_OutputStream
 m_CurrentSession = ...;                 // 修改共享资源
 m_OutputStream << "Logging message\n";   // 写操作

 // lock_guard 在这个代码块结束时会自动释放锁
 }
```

效果：
std::lock_guard 在构造时会获取互斥量的锁，并在其作用域结束时（即超出大括号或离开作用域）自动释放锁。

需要注意的是：

- 不要手动调用 lock() 和 unlock() 函数来管理互斥量，因为这样容易出错并导致死锁。
- std::lock_guard 的生命周期应该尽量短，只在需要保护共享资源时才创建和使用，以减少锁的持有时间，提高并发性能。

》》》》什么是iomanip?
<iomanip> 是 C++ 标准库中的头文件，定义了一些与格式化输入输出相关的功能和工具。
它提供了一些用于控制输入输出格式的类和函数，能够帮助程序员在输出数据时进行格式化，比如设置输出的精度、字段宽度、对齐方式等。

常见功能包括：

- std::setprecision(int n)：设置浮点数的输出精度为 n 位小数。
- std::setw(int n)：设置输出的字段宽度为 n 个字符。
- std::left、std::right、std::internal：控制输出的对齐方式，左对齐、右对齐或者内部对齐。
- std::fixed、std::scientific、std::hexfloat：设置浮点数的输出格式，固定小数位、科学计数法、十六进制表示等。

》》》》什么是std::chorno::steady_clock?

std::chrono::time_point std::chrono::steady_clock：

- 使用 std::chrono::steady_clock 作为时钟类型。
- steady_clock 提供了稳定且不会被系统时间调整影响的时间。它适合于测量时间间隔和延迟等场景，不会受到系统时间修改的影响，即使系统时间发生变化，该时钟也保持稳定。
- 精度一般是微秒级别或者更高，取决于系统的实现。

std::chrono::time_point std::chrono::high_resolution_clock：

- 使用 std::chrono::high_resolution_clock 作为时钟类型。
- high_resolution_clock 是一个特定系统实现的时钟，提供了尽可能高的精度，通常比 steady_clock 更精确，但具体精度因平台而异。
- 精度可能是纳秒级别或者更高，但由于其实现依赖于具体系统，因此可能在不同平台上有所不同。

主要区别：

- 稳定性：steady_clock 是稳定的时钟，不受系统时间修改的影响；而 high_resolution_clock 的稳定性依赖于具体实现，通常也比较稳定，但在某些情况下可能受到系统时间修改的影响。
- 精度：high_resolution_clock 的精度通常比 steady_clock 更高，但具体精度取决于系统的硬件和实现。

》》》》什么是std::chrono::duration?

std::chrono::duration 是用于表示和处理不同时间单位的时间段的重要工具.

类型定义

std::chrono::duration 是一个模板类，其基本模板定义如下：

```
template<class Rep, class Period = std::ratio<1>>
class duration;
```

- Rep：表示持续时间的数值类型，通常是一个整数类型（如 int, long, double 等），用来存储持续时间的数量。
- Period：表示持续时间的时间单位，使用 std::ratio 类型来表示，例如 std::ratio<1, 1000> 表示毫秒，std::ratio<1, 1000000> 表示微秒，std::ratio<1, 1> 表示秒（默认单位）。

示例

使用 std::chrono::duration 表示不同时间单位的时间段和进行基本的运算：

```
#include <iostream>
#include <chrono>

int main() {
   //定义两个不同单位不同数值的duration变量

      std::chrono::duration<int, std::ratio<1, 1>> seconds(30); // 30 seconds
      std::chrono::duration<double, std::ratio<1, 1000>> milliseconds(550); // 550 milliseconds

 // 将其相加

   auto total_seconds = seconds + std::chrono::duration_cast<std::chrono::duration<int>>(milliseconds);

 // 查看结果

   std::cout << "Total duration: " << total_seconds.count() << " seconds\n";

return 0;
}
```

定义了一个 seconds 和一个 milliseconds 的 std::chrono::duration 对象，分别表示 30 秒和 550 毫秒。然后将这两个时间段相加，并将结果转换为秒，最后输出总的持续时间。

》》 std::chrono::time_point<std::chrono::steady_clock> m_StartTimepoint;
auto highResStart = std::chrono::duration<double, std::micro>{ m_StartTimepoint.time_since_epoch() };的效果是什么?

具体来说，假设 m_StartTimepoint 是一个 steady_clock 类型的时间点，那么 m_StartTimepoint.time_since_epoch() 返回的是一个 std::chrono::duration 类型，表示自 steady_clock 的自纪元（通常是系统启动后的时间）开始至 m_StartTimepoint 的持续时间。
随后，我们创建 highResStart，使用的模板参数为 <double, std::micro>，这表示我们希望将这个持续时间表示为一个 double 类型的数值，单位是微秒。
因此，auto highResStart = std::chrono::duration<double, std::micro>{ m_StartTimepoint.time_since_epoch() }; 的效果是将 m_StartTimepoint 的自纪元至今的时间传递给HigResStart，将其转换一个 double 值，表示了 m_StartTimepoint 的时间戳，以微秒为精度。

》》
auto elapsedTime =
    std::chrono::time_point_cast<std::chrono::microseconds>(endTimepoint).time_since_epoch() - std::chrono::time_point_cast<std::chrono::microseconds>(m_StartTimepoint).time_since_epoch();
和
auto elapsedTime = std::chrono::duration<double, std::micro>{
    std::chrono::time_point_cast<std::chrono::microseconds>(endTimepoint).time_since_epoch() - std::chrono::time_point_cast<std::chrono::microseconds>(m_StartTimepoint).time_since_epoch()
};
有什么区别?

两种方式在功能上是等价的，主要区别在于返回结果的类型和精度。

- 类型和精度不同：

第一种方式	返回的是一个 std::chrono::microseconds 类型的持续时间，表示两个时间点的微秒级时间差。
第二种方式	返回的是一个 std::chrono::duration<double, std::micro> 类型的持续时间，其中 double 是数值类型，表示两个时间点的微秒级时间差的浮点数表示。

因此，选择哪种方式取决于你的具体需求：

如果你只需要整数微秒表示，	第一种方式足够。
如果你需要微秒级别的小数精度 或者希望结果为 double 类型，	建议使用第二种方式。

》》》》std::setprecision和std::fixed是什么意思?

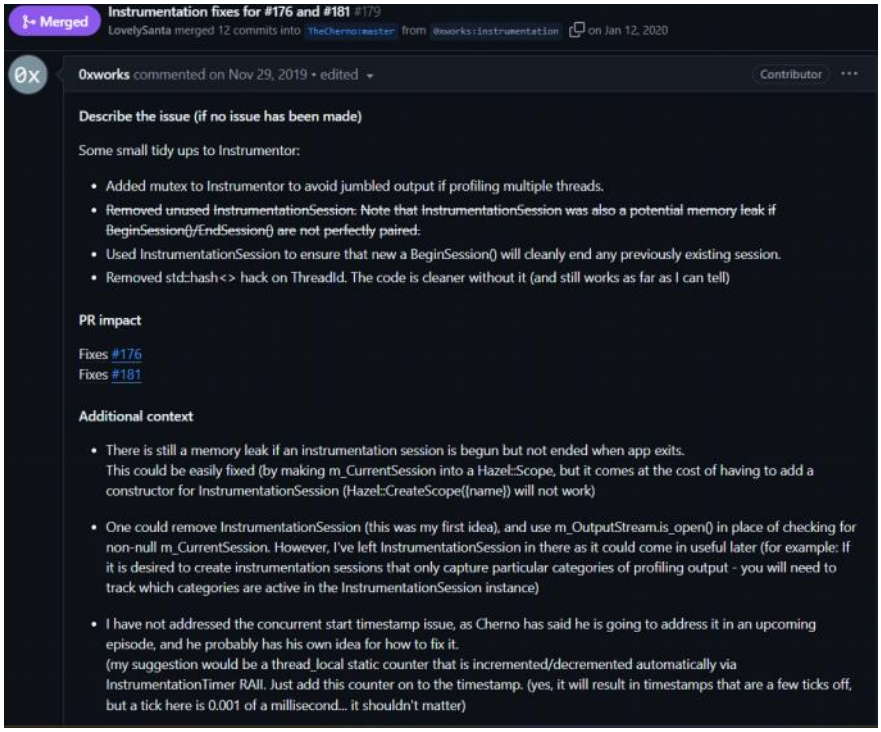
std::setprecision(3):

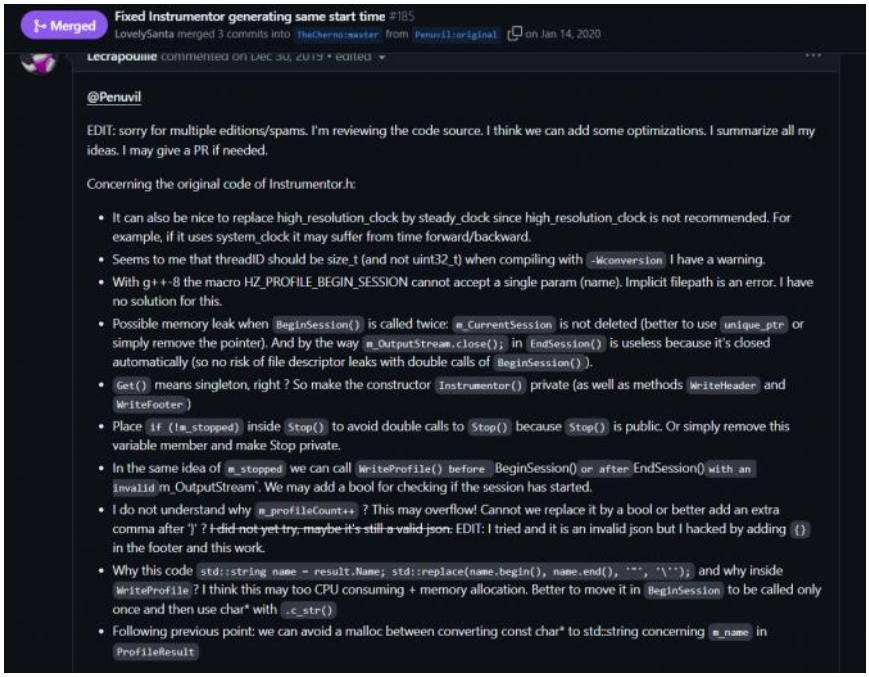
- 这个函数调用设置了浮点数的输出精度为3位小数。它是 <iomanip> 头文件中的函数，通过 std::setprecision 控制输出流的精度。
- 例如，如果将一个浮点数输出到流中并设置了 std::setprecision(3)，那么输出的浮点数将保留三位小数。

std::fixed:

- 这是另一个 <iomanip> 头文件中的修饰符，用于指定浮点数的输出格式为固定小数点表示法（即小数点后面始终保留指定的位数，不自动切换到科学计数法）。
- 当使用 std::fixed 后，浮点数将始终按照小数点后的位数进行输出，即使小数部分为0也会显示。
- 例如，如果一个浮点数是 3.14159265，使用 std::setprecision(3) 和 std::fixed 后输出为 3.142。

》》》》关于instrumentor和instrumentation的维护思路（未全部采用）



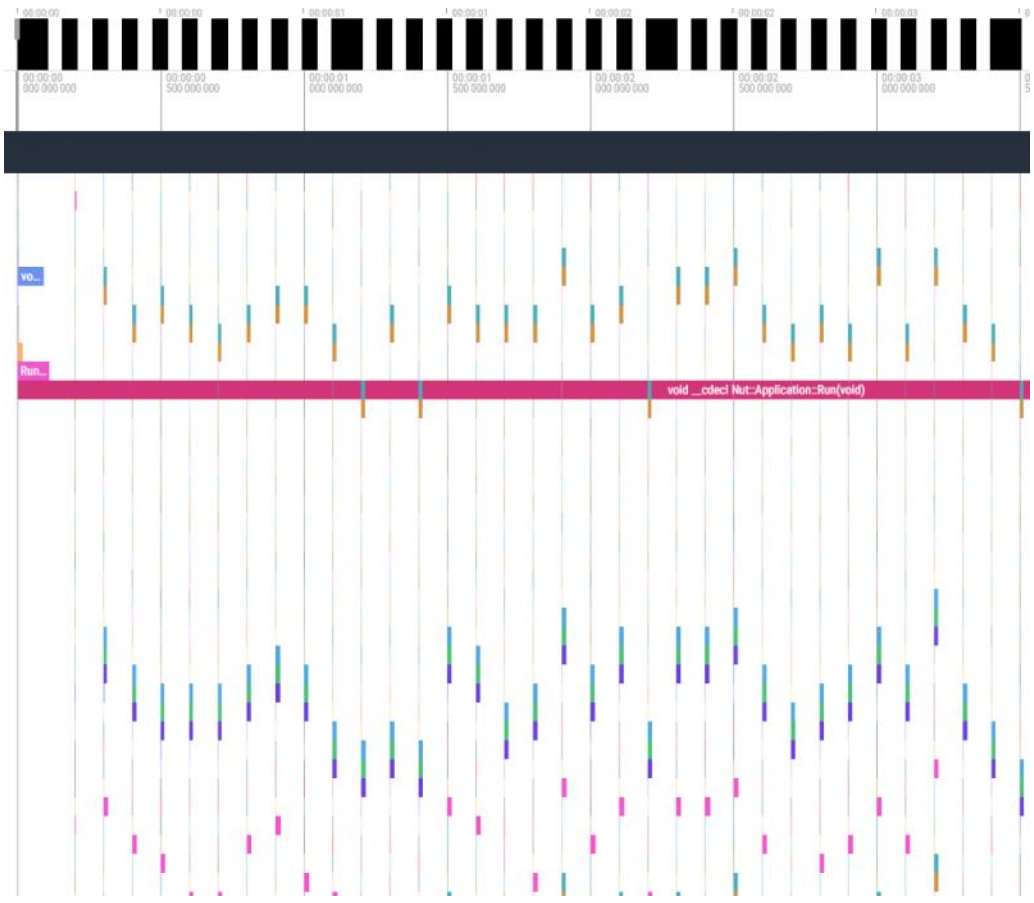


第二个维护的效果：Fixed Instrumentor generating same start time (#185)
（有的维护属于安全维护，包括防止内存泄漏或维护代码安全性，但是貌似 steady_clock 保证了时间稳定，然后 duration 对 StartTime 采用的更高精度使json文件更正确？ 我是这样想的）

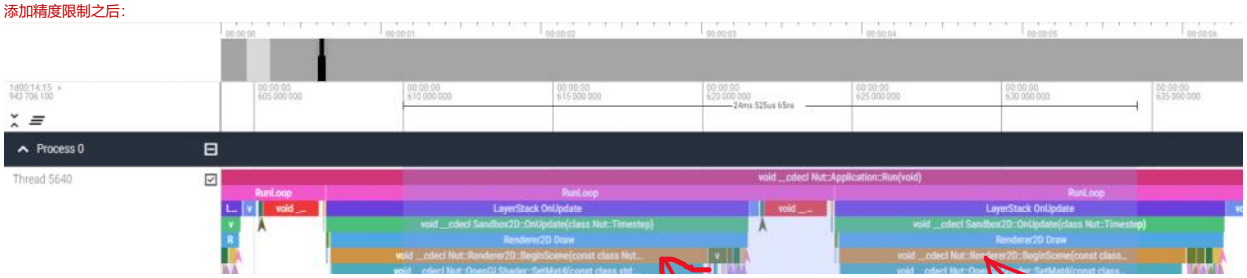
修改前：一些函数明明是相同的，却出现在不同的地方，这是因为 StartTime 出现了逻辑上的错误。

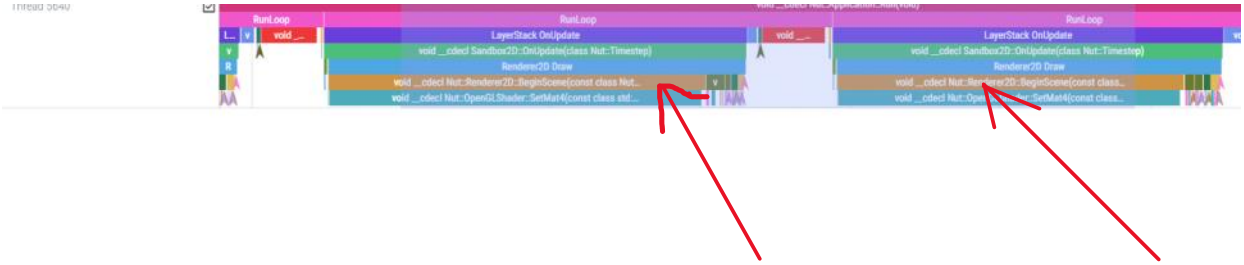
修改后：

由于修改后每个函数的 StartTime 精度很高，如果不只采用小数点后三位的话，json文件中会出现很大的间隔：



添加精度限制之后：





可以看到，来自同一个线程的 LayerStack::OnUpdate 在之前乱糟糟的，修改后处于同一个层次，很整齐。

》》》》关于 glDebugMessageControl 函数

glDebugMessageControl 函数用于控制 OpenGL 调试消息的生成和过滤。它的原型如下：

```
void glDebugMessageControl(GLenum source, GLenum type, GLenum severity, GLsizei count, const GLuint* ids, GLboolean enabled);
```

- source：指定调试消息的来源。可以是以下值之一：

GL_DEBUG_SOURCE_API:	由OpenGL API生成的调试消息。
GL_DEBUG_SOURCE_WINDOW_SYSTEM:	与窗口系统相关的调试消息。
GL_DEBUG_SOURCE_SHADER_COMPILER:	由着色器编译器生成的调试消息。
GL_DEBUG_SOURCE_THIRD_PARTY:	来自第三方库的调试消息。
GL_DEBUG_SOURCE_APPLICATION:	由应用程序自定义的调试消息。
GL_DEBUG_SOURCE_OTHER:	其它来源的调试消息。
GL_DONT_CARE:	表示不关心来源，接收所有来源的调试消息。

- type：指定调试消息的类型。可以是以下值之一：

GL_DEBUG_TYPE_ERROR:	错误消息。
GL_DEBUG_TYPE_DEPRECATED_BEHAVIOR:	不推荐使用的行为警告。
GL_DEBUG_TYPE_UNDEFINED_BEHAVIOR:	未定义的行为警告。
GL_DEBUG_TYPE_PORTABILITY:	不兼容性警告。
GL_DEBUG_TYPE_PERFORMANCE:	性能警告。
GL_DEBUG_TYPE_MARKER:	标记组。
GL_DEBUG_TYPE_PUSH_GROUP:	推组。
GL_DEBUG_TYPE_POP_GROUP:	弹组。
GL_DEBUG_TYPE_OTHER:	其它类型的调试消息。
GL_DONT_CARE:	表示不关心消息类型，接收所有类型的调试消息。

- severity：指定调试消息的严重性级别。可以是以下值之一：

GL_DEBUG_SEVERITY_HIGH:	高严重性的消息。
GL_DEBUG_SEVERITY_MEDIUM:	中等严重性的消息。
GL_DEBUG_SEVERITY_LOW:	低严重性的消息。
GL_DEBUG_SEVERITY_NOTIFICATION:	通知级别的消息。
GL_DONT_CARE:	表示不关心严重性级别，接收所有严重性级别的调试消息。

- count：指定 ids 数组中元素的数量，用于指定特定消息的ID。可以为0，表示没有指定特定的消息ID。
- ids：一个指向消息ID数组的指针，用于指定特定消息的ID。如果 count 为0或 ids 为 NULL，则表示不特定任何特定的消息ID。
- enabled：一个布尔值 (GL_TRUE 或 GL_FALSE)，指定是否启用指定条件下的调试消息。

```
glEnable(GL_DEBUG_OUTPUT);
```

- 这个函数调用启用了OpenGL的调试输出功能。启用后，OpenGL会生成调试消息，用于指示可能出现的错误、警告或其他状态信息。

```
glEnable(GL_DEBUG_OUTPUT_SYNCHRONOUS);
```

- 这个函数调用启用了OpenGL的同步调试输出功能。启用同步调试输出可以保证在调试消息的回调函数执行时，OpenGL的状态保持一致，有助于调试时的上下文理解和问题追踪。

```
glDebugMessageCallback(OpenGLMessageCallback, nullptr);
```

- 这个函数设置了一个回调函数 OpenGLMessageCallback，用于处理OpenGL生成的每条调试消息。每当OpenGL生成一个调试消息时，就会调用这个回调函数，并传递消息的详细信息作为参数。

```
glDebugMessageControl(GL_DONT_CARE, GL_DONT_CARE, GL_DEBUG_SEVERITY_NOTIFICATION, 0, NULL, GL_FALSE);
```

- 这个函数调用用于控制OpenGL生成的调试消息的生成和过滤。

》》》》<spdlog/sinks/basic_file_sink.h> 是用于在使用 SPDLOG 日志库时，添加支持将日志输出到基本文件的头文件。

Eg
logSinks.emplace_back(std::make_shared<spdlog::sinks::basic_file_sink_mt>("Hazel.log", true));

》》》》Log文件中的修改

- **创建日志输出目标：**
  std::vector<spdlog::sink_ptr> logSinks;
  logSinks.emplace_back(std::make_shared<spdlog::sinks::stdout_color_sink_mt>());
  logSinks.emplace_back(std::make_shared<spdlog::sinks::basic_file_sink_mt>("Hazel.log", true));

- logSinks 是一个存储 spdlog::sink_ptr（日志输出目标指针）的向量。
- emplace_back 函数用于将日志输出目标添加到 logSinks 中。
第一个日志输出目标是 stdout_color_sink_mt	表示标准输出并带有颜色。
第二个日志输出目标是 basic_file_sink_mt	表示写入到文件 "Hazel.log" 中，并设置为追加模式（true 参数）。

- **设置日志格式：**
  logSinks[0]->set_pattern("%^[%T] %n: %v%$");
  logSinks[1]->set_pattern("[%T] [%l] %n: %v");

- 对 logSinks 中的每个日志输出目标设置不同的日志格式。
第一个日志输出目标	使用彩色格式 %^[%T] %n: %v%$，其中 %^ 和 %$ 用于设置颜色。
第二个日志输出目标	使用不同的格式 [%T] [%l] %n: %v，其中 %l 表示日志级别。

- **创建日志记录器：**
  s_CoreLogger = std::make_shared<spdlog::logger>("HAZEL", begin(logSinks), end(logSinks));
  spdlog::register_logger(s_CoreLogger);
  s_CoreLogger->set_level(spdlog::level::trace);
  s_CoreLogger->flush_on(spdlog::level::trace);

  s_ClientLogger = std::make_shared<spdlog::logger>("APP", begin(logSinks), end(logSinks));
  spdlog::register_logger(s_ClientLogger);
  s_ClientLogger->set_level(spdlog::level::trace);
  s_ClientLogger->flush_on(spdlog::level::trace);

- 使用 std::make_shared 创建两个名为 "HAZEL" 和 "APP" 的 spdlog::logger 对象。
- 这些日志记录器将使用 logSinks 中的输出目标。

- 使用 spdlog::register_logger 将每个日志记录器注册到 spdlog 系统中，使其能够被全局访问。
- 设置日志记录器的默认日志级别为 trace，并且在每条日志写入时都进行刷新。

》》begin(logSinks) 和 end(logSinks)的作用？？
begin(logSinks) 和 end(logSinks) 是C++标准库中的函数，用于获取指向容器（例如 std::vector）中第一个元素和尾后位置的迭代器。
s_CoreLogger = std::make_shared<spdlog::logger>("HAZEL", begin(logSinks), end(logSinks));
这里的 begin(logSinks) 和 end(logSinks) 分别返回了 logSinks 向量的起始迭代器和尾后迭代器。这样做的目的是将 logSinks 向量中的所有日志输出目标都传递给 spdlog::logger 对象的构造函数，以便该日志记录器可以使用这些目标来进行日志输出。

》》》》为何 Nut.log 会出现在 sandbox 文件夹路径下呢？
可能是因为在入口点中的 auto app = Nut::CreateApplication(); 对象是由 CreateApplication 创建的，但是 CreateApplication 被定义在 sandboxApp 中。

----------Batch Rendering----------
》》》》虽然说还有个维护没做，但我想先学习批渲染，维护太乏味了=_=

》》》》为什么说引用指针可以有效防止程序崩溃？在自动管理生命周期的时候防止对象被提前删除？
引用指针的作用：
- 避免内存泄漏： 内存泄漏是指程序分配了一块内存后，由于没有正确释放，导致该内存无法再被程序使用。
  使用引用计数，系统可以在对象不再被引用时及时释放其内存，从而避免了内存泄漏。
- 避免空指针访问： 在某些情况下，程序可能会试图访问已经被释放的内存，这通常会导致空指针异常（Null Pointer Exception）。
  引用计数确保对在不再被引用时释放，从而避免了访问已释放内存的问题。

》》》》关于批渲染，可以先看看Cherno在OpenGL系列中的教程，看完之后再来GameEngine，会很轻松。

YouTube: https://youtu.be/Th4huqR77rI?si=qaGtoiw--Oec5Pf-

BiliBili:【【双语】【TheCherno】OpenGL】 https://www.bilibili.com/video/BV1Ni4y1o7Au/?p=28&share_source=copy_web&vd_source=ca2feff7d155a2579964dfa2c3173769
有中文字幕，貌似比油管上更全?

》》》》glBufferSubData 的作用：

释义：	glBufferSubData 是 OpenGL 中用于更新缓冲区对象部分数据的函数。
作用：	将数据复制到已绑定的缓冲区对象的指定位置。
声明：	void glBufferSubData(GLenum target, GLintptr offset, GLsizeiptr size, const void* data);
参数：	target 指定了目标缓冲区对象的类型， 例如 GL_ARRAY_BUFFER 表示顶点缓冲区，GL_ELEMENT_ARRAY_BUFFER 表示索引缓冲区等。 offset 指定了要更新的缓冲区数据的起始偏移量。 size 指定了要更新的数据的字节数。 data 指向要复制到缓冲区的数据的指针。
意义：	使用 glBufferSubData 可以在不重新分配整个缓冲区的情况下，只更新部分数据，这对于动态更新顶点数据或索引数据非常有用

》》》》指针的相减是有效的吗？如果在栈上分配两个变量base 和 hind，其相减之后得到的确实是之间的有效大小，若是在堆上分配两个指针 base 和 hind，其代表的内存可能是不相邻的，能否相减？

答：指针相减的有效性不取决于指针指向的内存地址是否连续，而是取决于它们指向的对象类型和大小。
C++ 编译器能够根据指针类型自动计算出正确的偏移量，这使得指针相减在计算对象之间的距离时有效。

- 如果 p 和 q 是指向数组元素的指针，例如 int* p 和 int* q，那么 p - q 将给出 p 和 q 之间的元素个数，而不管它们在内存中的确切位置。

》》》》绘制中的顶点顺序详见：https://learnopengl-cn.github.io/04%20Advanced%20OpenGL/04%20Face%20culling/
以供参考

----------Batch Rendering for Texture----------

》》》》std::array 的第二个参数：元素数量（大小）需要在编译时就确定下来
在 std::array 的模板参数中，第二个参数通常表示数组的大小，它必须是一个常量表达式，即在编译时期就能确定其值的表达式。这个值可以是一个整数常量、枚举常量、或者可以在编译时求值得到的表达式。

Eg.

```
std::array<int, 5> arr1; // 一个包含5个整数的std::array

std::array<double, 10> arr2; // 一个包含10个双精度浮点数的std::array

constexpr int size = 3;
std::array<char, size> arr3; // size 是一个编译时常量表达式

Static const int size = 3;
std::array<char, size> arr3; // size 是一个编译时常量表达式
```

》》》》这里没什么需要特别钻研的设计，我在代码上做了注释，多看两边视频或者代码或者注释完全能看懂。

- 一般步骤：
  选择合适的运算符：首先确定要重载的运算符，例如算术运算符 +, -, *, /，比较运算符 <, >, <=, >=, 等等。
  确定函数签名：根据选择的运算符确定重载函数的签名。每个运算符都有其特定的函数名称和参数列表。
  实现重载函数：按照确定的函数签名编写运算符重载函数的实现。
  测试和验证：确保重载函数在各种情况下表现正常，包括正常情况、边界情况和异常情况。

- 一般规范和注意事项：
  - 函数参数：大多数情况下，运算符重载函数至少有一个参数。例如，对于二元运算符（如 +），通常需要一个参数是当前对象本身，另一个是右操作数。对于一元运算符（如 ++），则只需要一个参数。
  - 成员函数 vs. 非成员函数：运算符重载函数可以作为类的成员函数或者非成员函数实现。成员函数版本会访问类的私有成员，非成员函数版本需要在参数列表中显式传递操作数。
  - 返回类型：通常应该返回与原生运算符相同类型的结果。例如，重载 + 运算符通常返回一个新的对象，该对象包含两个操作数相加的结果。
  - 避免副作用：确保运算符重载函数的行为与预期一致，不会对对象状态造成意外修改或者副作用。
  - 保持语义一致性：运算符重载函数的行为应该符合预期的数学和逻辑语义。例如，+ 运算符对于整数和浮点数的行为通常是相似的，应该保持类似的行为。

- 示例：

- 二元运算符重载示例（成员函数）：

```
class Vector {
private:
    int x, y;
public:
    Vector operator+(const Vector& other) const {
        Vector result;
        result.x = this->x + other.x;
        result.y = this->y + other.y;
        return result;
    }
};
```

- 一元运算符重载示例（成员函数）：

```
class Integer {
private:
    int value;
public:
    Integer operator++() { // 前缀++
        ++value;
        return *this;
    }
Integer operator++(int) { // 后缀++
        Integer temp = *this;
        ++value;
        return temp;
    }
};
```

- 非成员函数运算符重载示例：

```
class Complex {
private:
    double real, imag;
public:
    Complex(double r, double i) : real(r), imag(i) {}
friend Complex operator+(const Complex& a, const Complex& b) {
        return Complex(a.real + b.real, a.imag + b.imag);
    }
};
```

》》》》const 和 constexpr 的区别：

- const 变量是在运行时初始化的常量，其值在编译时无法确定。
- constexpr 变量是在编译时期初始化的常量表达式，其值可以用于需要常量表达式的上下文，例如数组大小、模板参数等。

》》》》*s_Data.Textures[i].get() == *texture.get() 的理解。

s_Data.Textures	std::array<Ref<Texture2D>, MaxTextureSlots> Textures;
Texture	const Ref<Texture2D>& texture

- s_Data.Textures[i].get() 获取 s_Data.Textures[i] 所管理的 Texture2D 对象的裸指针。
  texture.get() 获取 texture 智能指针所管理的 Texture2D 对象的裸指针。

- *s_Data.Textures[i].get() 和 *texture.get() 将这两个裸指针解引用，即获取它们所指向的 Texture2D 对象本身。
  而不是比较 .get() 获取的指针本身的值。

- 因此，*s_Data.Textures[i].get() == *texture.get() 表示比较 s_Data.Textures[i] 和 texture 所管理的 Texture2D 对象是否相等，而不是比较它们的指针地址或其他内容。