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**

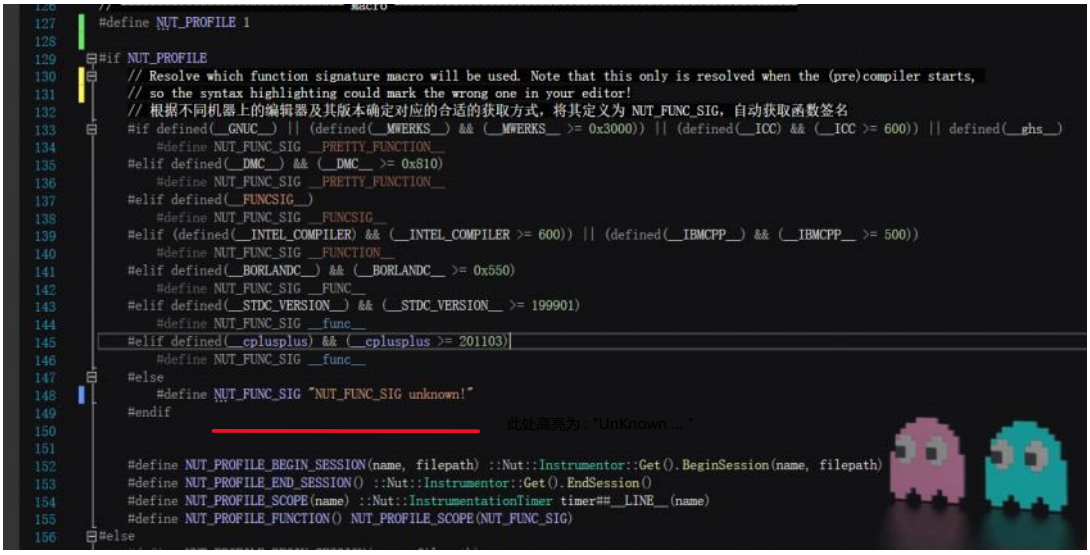
- 如果以上所有条件都不满足，则选择 "NUT_FUNC_SIG unknown!" 作为 NUT_FUNC_SIG 的默认值。这种情况下，编译器可能不支持预定义的函数签名获取方式，或者无法识别当前的编译环境。

```
// 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)

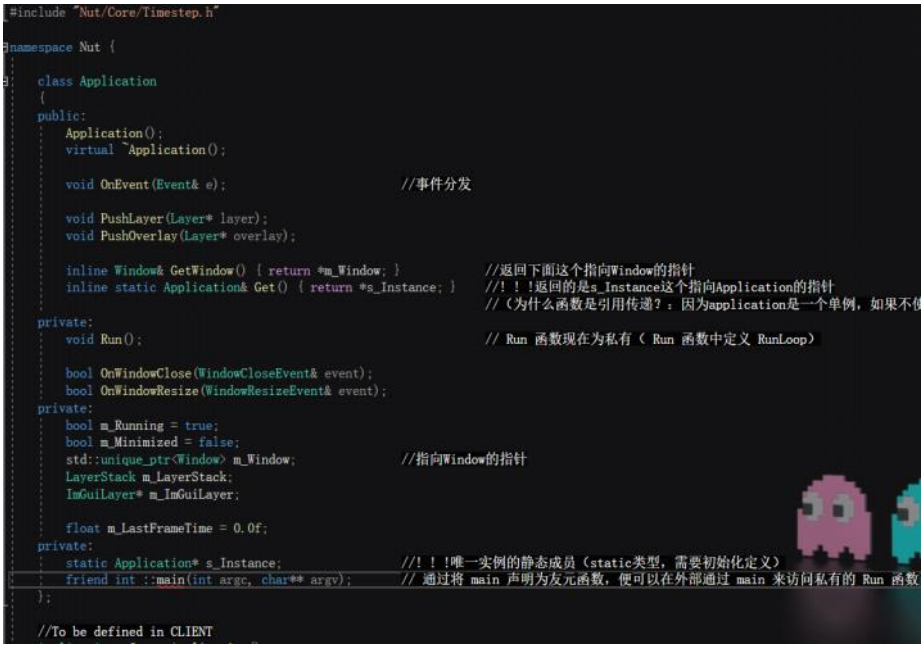
如何限制只在入口点（引擎端）访问 Run 函数（内含RunLoop的函数) 呢?
由于我们将主函数 main 设置在 EntryPoint 文件中，并在此处定义，所以想要只能在此处访问 Run 函数的话，我们需要将 Run 函数作为 Application 类中的私有成员函数，并将 main 函数作为 Application 类的友元函数。

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：

1. 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?
2. 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

》》》》