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 HZ_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 HZ_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 HZ_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 HZ_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 HZ_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 HZ_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 "HZ_FUNC_SIG unknown!" 作为 HZ_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 HZ_FUNC_SIG __PRETTY_FUNCTION__
#elif defined(__DMC__) && (__DMC__ >= 0x810)
    #define HZ_FUNC_SIG __PRETTY_FUNCTION__
#elif defined(__FUNCSIG__)
    #define HZ_FUNC_SIG __FUNCSIG__
#elif (defined(__INTEL_COMPILER) && (__INTEL_COMPILER >= 600)) || (defined(__IBMCPP__) && (__IBMCPP__ >= 500))
    #define HZ_FUNC_SIG __FUNCTION__
#elif defined(__BORLANDC__) && (__BORLANDC__ >= 0x550)
    #define HZ_FUNC_SIG __FUNC__
#elif defined(__STDC_VERSION__) && (__STDC_VERSION__ >= 199901)
    #define HZ_FUNC_SIG __func__
#elif defined(__cplusplus) && (__cplusplus >= 201103)
    #define HZ_FUNC_SIG __func__
#else
    #define HZ_FUNC_SIG "HZ_FUNC_SIG unknown!"
#endif

#define HZ_PROFILE_FUNCTION() HZ_PROFILE_SCOPE(HZ_FUNC_SIG)
```