GameEngine4

2023年12月3日 21:48

----------Shader Library（着色器库）----------

》》》》终于开始着色器库的设计，来看看着色器库是用来干嘛的。

着色器库的设置是静态的，用于自动加载着色器内容、诊断着色器。

高度抽象的设置可以为我们自动化识别接口，使在程序中的调用更加简洁，同时也隐藏了一些细节（自动化处理，不用手动显示设置）

》》》》以下是一些设计中的理解：

》》在 ShaderLibary 中，使用之前定义的指针 Ref 和 Scope，不需要包含头文件即可使用，为什么。

原理：

命名空间的作用域在整个程序中都有效，只要命名空间被正确声明和定义，其成员在程序的任何地方都可见。

不使用命名空间的话，直接在代码中使用 Nut::Ref 也是可行的，

》》关于Cherno对于从文件名截取着色器名称的手法，在 C++ 版本的逐步发展中，还有其他更便捷的方法。

原版：

```
// Get Shader's name though filepath name
// maybe :        1.assets/textures/shader.glsl        2./shader       3.shader.glsl    4.shader

size_t lastSlash = filepath.find_last_of('/\\');
lastSlash = (lastSlash == std::string::npos ? 0 : lastSlash + 1);
size_t lastDot = filepath.rfind('.');
lastDot = (lastDot == std::string::npos ? filepath.size() : lastDot);

size_t count = lastDot - lastSlash;
m_Name = filepath.substr(0, count);
```

新版：（C++17）

```
#include <filesystem>

std::filesystem::path path = filepath;
m_Name = path.stem().string(); // Returns the file's name stripped of the extension.
```

截取自评论：

@This new feature is very useful for use in Asset Managers as well which I'm currently implementing in my project, it supports native file path directory seeking.

@And it is bulletproof. His code isn't. If the path is "../testures/texture", the count will be negative, because the last dot is before the last slash. And this is a valid path.

----------How to build 2D engine----------

没什么要记的，后面也都会涉及。这一集就是一个大概思路。

----------Camera Controllers----------

》》》》大致思路

将实际控制通过 Camera Controller 来调用，而不是像以前一样通过 Camera 类来直接的修改和更新 Camera 的值。

现在的 Camera 可以认为是一个深层次的、存放了一些计算方法的库，controller 会调用这些方法，而用户只需要使用 Controller 来进行操纵。

》》》》std::max()?

std::max 是 C++ 标准库中的一个函数模板，用于比较两个值，并返回其中较大的值。它定义在 <algorithm> 头文件中。

》》》》claudialDE 2019

一个可以更改 VS2019 背景的插件，这就是我昨天没有更选代码的原因。因为我去折腾壁纸了。

----------Resizing----------

》》》》minimized 这个变量的意义？

举个例子：

比如你在打一个游戏（英雄联盟），现在你打开了购买装备的界面，然后你又做了切换应用的操作。

此时游戏会被最小化到后台，购买界面也会随游戏被最小化。也不可以在脱离游戏界面时被操作。

只有你重新进入游戏界面，继续运行游戏时，这个购买界面才可以被响应（进行购买、关闭等操作）

》》》》关于函数设计时的思考。

application 中的 OnWindowResize 是为了让视口填充整个窗口。

OrthoGraphicCameraController 中的 OnWindowResized 是为了让视口中的物体在窗口大小调整时候比例自适应，而且呈现正确的效果。

但是函数

```
bool OrthoGraphicCameraController::OnWindowResized(WindowResizeEvent e)
{
    m_AspectRatio = (float)e.GetWidth() / (float)e.GetHeight();            //设置回调的宽高比
    m_Camera.SetProjectionMatrix(-m_AspectRatio * m_ZoomLevel, m_AspectRatio * m_ZoomLevel, -m_ZoomLevel, m_ZoomLevel);
```

```
    return false;
}
```
是这样设计的，
所以 m_AspectRatio 会随 width（分子）的大小调整正确变化，但是与 Height（分母）的大小变化刚好相反。
而且由于 m_AspectRatio 是 width 除以 height，所以当使用鼠标在窗口对角进行操作时（即对 Width 和 Height 同时进行改变），渲染的物体比例不变。

这是一个待处理的瑕疵。

---------- maintenance ----------

》》》》pushd 和 popd 的使用

pushd：	将当前目录入栈，并切换到指定的目录。
popd：	从栈中弹出最近保存的目录，并切换到该目录。

eg.
::使用 pushd 命令切换目录并将当前目录推入栈中：
pushd <目标目录>
::使用 popd 命令从栈中弹出最近保存的目录并切换到该目录：
popd

---------- preparing for 2D rendering ----------

》》》》没什么要记的，但是我顺手同步跟新一下错误修复的 submit，并做笔记。

》》》》gitignore 规范
.gitignore 文件的规范是通过简单的文本格式来定义忽略规则。每个规则占用一行，用于指定要忽略的文件、文件夹或者特定模式。

文件匹配：

使用斜杠（/）	表示路径分隔符。
使用星号（*）	表示匹配任意数量的字符（除了路径分隔符）。
使用双星号（**）	表示匹配任意数量的字符（包括路径分隔符）。
使用问号（?）	表示匹配一个任意字符。
使用感叹号（!）	表示不忽略的文件或文件夹。

注释：

使用井号（#）	开头的行表示注释，这些行可能会被Git忽略，一般另起一行来写。

注释可以写在规则的上方，用于对规则进行解释或提供其他相关信息。

eg.
```
# 忽略所有的编译输出文件
*.o
*.exe
*.dll

# 忽略bin文件夹及其下的内容
/bin/

# 忽略.vscode文件夹
.vscode/

# 不忽略lib文件夹下的example.txt文件
!lib/example.txt
```

》》*bin\和**bin\之间的区别：
***bin**：表示匹配当前目录下的任意一级子目录中的bin文件夹。例如，src/bin/、lib/bin/等。
****bin**：表示匹配当前目录及其所有子目录中的bin文件夹。例如，src/bin/、src/utils/bin/、lib/bin/等。

》》\bin\和bin\在.gitignore文件中的区别：
****\bin**：这个规则表示匹配当前目录及其所有子目录中的bin文件夹。例如，src/bin/、src/utils/bin/、lib/bin/等都会被匹配到。
****bin**：这个规则表示匹配当前目录及其直接子目录中的名为bin的文件夹。例如，src/bin/、lib/bin/会被匹配到，但是src/utils/bin/不会被匹配到。

》》vs\和.vs\的区别：
从 '\' 看出这都表示忽略文件夹（只不过名称不一样，'.'开头的一般是隐藏文件夹）

》》》》什么是 constexpr？
constexpr是C++11引入的一个关键字，用于声明可以在编译时求值的常量表达式。它可以用于变量、函数、构造函数等的声明中。

》》constexpr中所有参数必须是字面类型。那
```
template<typename T>
using Scope = std::unique_ptr<T>;
template<typename T, typename ... Args>
constexpr Scope<T> CreateScope(Args&& ... args)
{
    return std::make_unique<T>(std::forward<Args>(args)...);
```

}
这个函数声明为什么是正确的呢?

因为：C++17对constexpr函数进行了一些扩展，放宽了函数参数的限制。
如果函数的参数满足以下条件之一，那么即使函数本身不是字面量函数，也可以声明为constexpr函数:

1.参数类型是字面类型；
2.参数是指向字面类型的指针或引用；
3.参数是数组类型，并且其元素类型是字面类型。

具体分析：
在CreateScope中，虽然模板参数Args可以包含任意类型的参数，但在调用std::make_unique<T>时，参数是通过完美转发传递的
因此参数类型和值在编译期是可以确定的，可以满足constexpr函数的要求。

》》》》constexpr （常量表达式）的好处。
constexpr的好处：提高程序的性能、安全性和可维护性，尤其适用于需要在编译期确定结果的函数。

》》》》template<typename T, typename ... Args> 模版中的 typename ... Args是什么?
》》》》CreateScope(Args&& ... args)中的参数是什么?

typename ... Args 是一个模板参数包，意味着 Args 是一个包含零个或多个模板参数的集合。（可以是任意数量的其他类型。）
Args&& ... args 中的 Args&& 是一种右值引用折叠语法。表示将模板参数包 Args 中的每个参数都作为右值引用传递给函数。
其中：

Args	是一个模板参数包。
&&	表示右值引用，表示参数 args 是右值引用类型。
...	则表示参数包展开，即将参数包中的每个参数都单独地传递给函数。
args	是函数的参数名。在函数中，它代表了接受模板参数包 Args 中传递进来的参数。

》》》》在函数的返回值中，我发现了 std::forward 这是什么东西? 怎样使用?

概念：	std::forward 是一个 C++ 标准库中的函数模板，用于实现完美转发（Perfect Forwarding）。
作用：	std::forward 允许在函数模板中保持 被传入的参数的类别（左值或右值）和常量性。 通过将参数以原始的值类别（左值或右值类型）传递，来避免不必要的拷贝和转换，提高程序的性能和效率。 // 后面会涉及到一些概念，先将笔记看下去 ~
参数与返回值:	它接受一个参数，并返回相同类型的参数，并且根据参数的值类别（左值或右值）不同，它会将参数转发为左值引用或右值引用。（以此确保参数值属性的正确传递）

使用：
在例子 std::forward<Args>(args)... 中： <Args>指定了传递给 std::forward 函数的参数类型，(args)...指定了传入的每个参数。
结果：
整个表达式的作用是将模板参数包 args 中的每个参数都通过 std::forward 转发给其他函数，并保持其原始的值类别。
这样就实现了完美转发的效果，使得参数在传递过程中保持了原始的左值或右值特性。

》》什么是完美转发? 什么是左值引用? 什么是右值引用?

--- 0. 值的类别（左值、右值）

左值（Lvalue）

概念：	有名称 / 在内存中有确定位置的表达式或对象。
意义：	可以被引用、修改和取地址。
eg.	int x = 5; 中的x就是一个左值。

右值（Rvalue）

概念：	不具有名称 / 在内存中没有确定位置的临时表达式或对象。
意义：	不能被引用，只能被移动或复制。
eg.	int y = 3 + 2; 中的3 + 2就是一个右值。

--- 1.引用的两种类型（左值引用、右值引用）。这两种引用基本是对左右值用法的拓展，其本旨与左右值相像。

左值引用:

概念：	使用 & 符号声明的引用类型。
意义：	表示对一个命名对象的引用，该对象可以被修改。
eg.	int x = 10; int& y = x;

右值引用:

概念：	使用 && 符号声明的引用类型。

意义：	表示对一个临时对象或将要销毁的对象的引用，该对象不能被修改。
eg.	int&& z = 20;

--- 2.
完美转发：完美转发是一种技术，通常涉及使用 std::forward 函数模板来将参数转发为左值引用或右值引用。
在将各种类型的参数传入 std::forward 的过程中，保留原始的值的左右值特性，避免不必要的拷贝和转换，提高程序的性能和效率。

》》》》如果在 return 中不使用 std::forward 呢，有什么结果?

return std::make_unique<T>(std::forward<Args>(args)...);
return std::make_unique<T>(args...);
虽然后者没有错误，但是会丧失参数原本的语义。这会带来性能上的损失，尤其是在处理大型对象时。

》》为什么说会导致性能的损失呢?
一般来说，在传入具有左右值属性的参数时，编译器会根据左右引用的不同属性选择动态转移或复制(拷贝)来传递。也就是移动语义和复制语义。
（通常，右值引用可以触发移动语义，左值引用可以触发复制语义）

如果不使用 std::forward 来保证左右值属性的正确传递，将会导致参数传递时丧失了对应的语义。这意味着即使传递的是右值，也会进行复制构造，而不是移动构造。
然而复制构造需要分配额外的时间和内存来进行深度复制操作，这会消耗相对较大的内存储备，降低性能表现。

》》》》什么是移动语义和复制语义？为什么右值触发移动语义，左值触发复制语义?

在C++中，值的语义（复制或移动）和值的类别（左值和右值）是密切相关的。

复制语义：

概念：	在将一个对象传递给另一个对象时，会创建该对象的一个全新的、独立的拷贝，两者之间没有关联。
实现方式：	通过拷贝构造函数来实现。

移动语义：

概念：	它允许在不复制内存的情况下将对象从一个位置（例如临时对象）转移到另一个位置。它适用于临时对象或者不再需要的对象。
不同之处：	移动语义将资源的所有权从一个对象转移到另一个对象，而不是创建资源的拷贝。
实现方式：	通过移动构造函数和移动赋值运算符来实现。

开销：

复制语义（大）	分配额外的内存和时间来进行深度复制，这会消耗大量的内存，尤其是对于包含大量数据成员的对象（包含大量数据）。
移动语义（小）	显著提高程序的性能和效率（特别是动态分配资源时）。

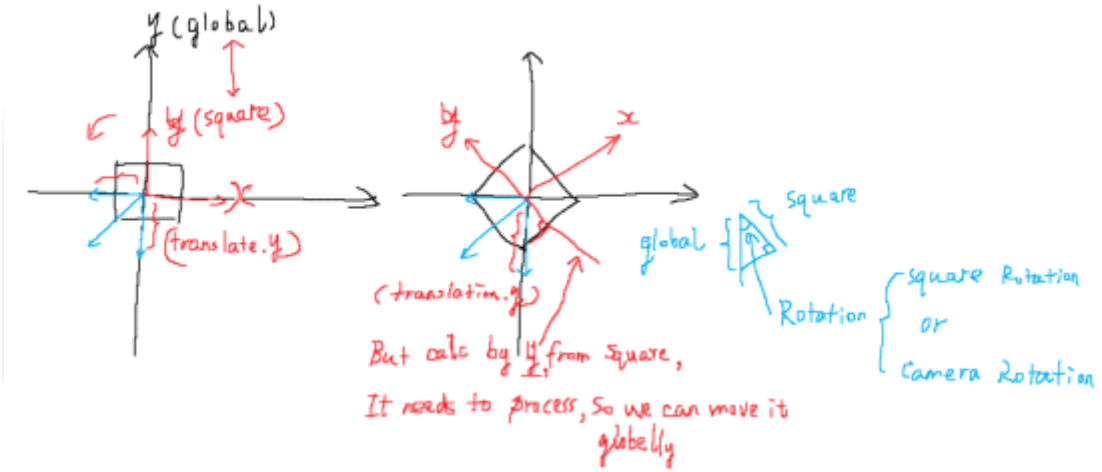
右值引用触发移动语义：

原因：	右值引用通常绑定到临时对象或将要销毁的对象上，这些对象不再需要原来的值，因此可以使用移动语义安全地将其资源移动到新的对象中。
意义：	这可以在处理大型对象或需要频繁传递所有权的情况下提高程序性能。

左值引用触发复制语义：

原因：	左值引用通常绑定到具名对象上，这些对象仍然需要保持原来的值。因此，对左值引用进行操作时会触发复制语义，即将原来对象的值复制到新的对象中。
意义：	这样对象的值也不会被意外地修改。

关于通过物体在物体（局部）的坐标轴上的偏移量 计算全局坐标偏移量的理解。



》》》》**在宏定义中，#ifdef _WIN32这个条件能够在32位操作平台上被自动触发，为什么?**

_WIN32	是一个预定义的宏，通常由Windows的编译器在编译过程中自动定义。

这个宏用来标识当前代码是否在Windows平台上编译运行的一个条件宏，由编译器根据编译环境自动设置。

》》》》**在.gitmodules文件中，branch = XXX这个指令的操作是什么意思？有什么作用?**

作用：	指定每一个子模块在项目中使用的分支名称。
意义：	对于每个子模块，都可以指定使用的特定分支，这样可以确保每个子模块都使用了正确的代码版本。 可以保证主项目和子模块之间的关联始终指向相应的稳定版本或者需要的特性分支。

#define EVENT_CLASS_TYPE(type) static EventType GetStaticType() { return EventType::##type; }\
和
#define EVENT_CLASS_TYPE(type) static EventType GetStaticType() { return EventType::type; }\
这两段代码在使用上有没有什么不同?

虽说这两个宏定义都是在填入 type 的时候定义一个函数 GetStaticType() 并返回 EventType,但是不同之处在于 ##
1.有 ## 时，填入的 type 会被拼接在 EventType 之后，并在预处理阶段动态的合成，结果会作为一个临时生成的类型，并返回。
2.无 ## 时，填入的 type 会根据填入的 type 返回一个事件类型，但此时填入的 type 如果生成的结果是没有被设置过的一种 type，会导致返回报错，因为 EventType::type 并不一定存在。
而不是像上面的那样返回一个临时类型，而无论正误。

》》》》模板中的参数包是什么，怎样使用？什么是参数包展开？ 参数包和完美转发的关系是什么?

- **参数包（Parameter Pack）**是C++11及以后版本中模板元编程中的一个特性，它允许你定义一个可以接受任意数量模板参数的模板类或模板函数。
  参数包用...来表示，它可以与类型参数（typename... Args）或值参数（auto... args 或 Args... args）一起使用。
- **参数包展开（Parameter Pack Expansion）**是对参数包中每个元素进行操作的语法，它使用...来指示编译器对参数包中的每个元素执行的操作。

  通常与模板函数和模板类中的参数包一起使用。
- **操作规则**：

- ***值参数包***：
  值参数包用于模板函数，允许函数接受任意数量的值作为参数。例如：

  ```
  template<typename... Args>
  void foo(Args... args) {
      // 在这里可以对 args 进行操作
  }
  ```

  args 是一个值参数包，它可以接受任意数量和类型的值作为函数参数。

- ***参数包展开***：
  参数包展开通常与递归模板或函数一起使用，以便对参数包中的每个元素执行操作。在函数模板中，可以使用递归终止函数和递归函数来展开参数包。例如，下面的函数使用递归模板来打印参数包中的所有值：

  ```
  template<typename First, typename... Rest>
  void print(First value, Rest... rest) {
      std::cout << value << std::endl;
      Function(rest...); // 参数包展开
  }
  ```

在这个例子中，print 函数首先处理参数包中的第一个值，然后调用Function来处理剩余的值（通过参数包展开）。

- ***std::forward和完美转发***:
  当处理值参数包时，经常需要保持原始参数的左值或右值性（lvalue or rvalue）。std::forward 函数模板用于在模板函数中完美转发参数，它接受一个参数类型和参数本身，并返回该参数的正确引用类型（左值引用或右值引用）。

```
template<typename... Args>
void wrapper(Args&&... args) {
    foo(std::forward<Args>(args)...); //完美转发参数包
}
```

在这个例子中，wrapper 函数接受任意数量和类型的参数，并使用 std::forward 将它们完美转发给 foo 函数。

<TargetConditionals.h>文件是什么？有什么作用?
苹果提供的 SDK 中的 usr/include/TargetConditionals.h 文件，会自动配置编译器编译的代码所需要使用的微处理器指令集、运行系统以及运行时环境。

<TargetConditionals.h>适用于所有的编译器，但是它只能被运行于 Mac OS X 系统上的编译器所识别。

----------Starting our 2D Renderer----------

》》》基本上是做一些绘制图像方法的抽象化，使之后的调用简单明了，易于使用。

因为时间的安排，有三周没学习游戏引擎了，很想酣畅淋漓的学习一下，Let's jump in and see what we get.

》》》》#pragma region 和 #pragma endregion 的使用
效果：
- 当你折叠 #pragma region 所在的代码块时，整个区域的代码将被隐藏起来，只显示 #pragma region 行。
- 当你展开 #pragma region 时，其中的代码块将再次显示出来。

Eg.
```
#pragma region Some Notes for following codes
// Your code here
#pragma endregion
```

》》》》宏定义是否可以不用包含头文件而跨文件使用?
当你在一个文件中定义了一个宏，比如 #define Bit(x) (1 << x)，这个宏定义实际上是在预处理阶段展开的。
这意味着：
第一：一旦宏在一个文件中定义了，它可以在该文件中的任何地方使用，包括函数内部和全局作用域。
第二：现在假设你有两个不同的源文件：file1.cpp 和 file2.cpp。
　　如果 Bit(x) 这个宏定义是在 file1.cpp 中定义的，而 file2.cpp 中需要使用这个宏，通常情况下是可以直接使用的，即使你在 file2.cpp 中没有显式地包含 file1.cpp 的内容或头文件。

跨文件使用的原理：
这种跨文件使用宏定义的原理在于编译器在处理源文件时是将所有文件合并到一个单独的翻译单元中。因此，编译器会将 file1.cpp 和 file2.cpp 分别预处理，并将宏定义展开为相应的代码。只要宏定义在编译器再次遇到它时已经在之前展开过了，就可以在后续的代码中使用它，不受文件边界的限制。

》》》》函数传参的理解

现有一个函数声明，其参数要求为 const glm::vec3&，如果你尝试直接传递一个由三个浮点数组成的元组 (position.x, position.y, 0.0f)，会遇到问题，因为这不是合法的 C++ 语法。
C++ 不支持直接传递裸值的元组作为参数。

- **使用花括号初始化列表**
  参数填入{ position.x, position.y, 0.0f } 时，花括号 {} 引用了 C++11 引入的统一初始化语法，这种语法可以创建一个临时的 glm::vec3 对象。
  具体来说，这个语法会被编译器解释为:
  Eg.
```
glm::vec3 temp(position.x, position.y, 0.0f);
↓
DrawQuad(temp, size, color);
```
  这里临时的 glm::vec3 对象会在 DrawQuad 调用时被隐式创建，并传递给函数。

- **直接使用括号**
  这在 C++ 中是不合法的语法。圆括号在这种上下文中意味着逗号运算符，而不是构造一个对象。所以参数填入(position.x, position.y, 0.0f)实际上会被解释为：
  Eg.
```
DrawQuad(0.0f, size, color);
```
  这是因为逗号运算符会依次计算每个操作数并返回最后一个操作数的结果，在这种情况下就是 0.0f，这显然不符合 DrawQuad 函数期望的参数类型。

》》》》在 C++ 中，通常无法通过父类的实例直接访问子类的成员函数。

除非子类的成员不是子类特有的，而是通过父类的虚函数重写的。

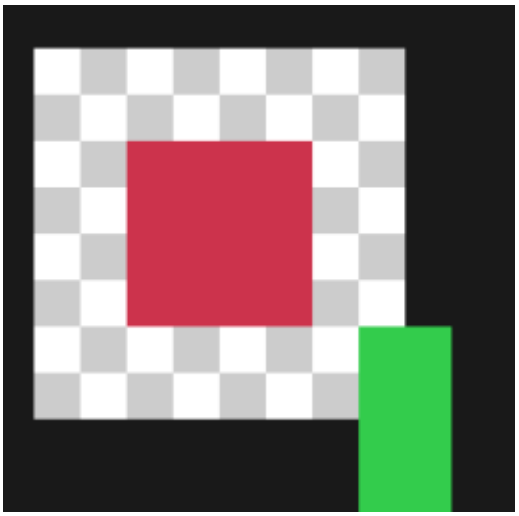
---------- 2D Renderer Transform & 2D Renderer Texture ----------

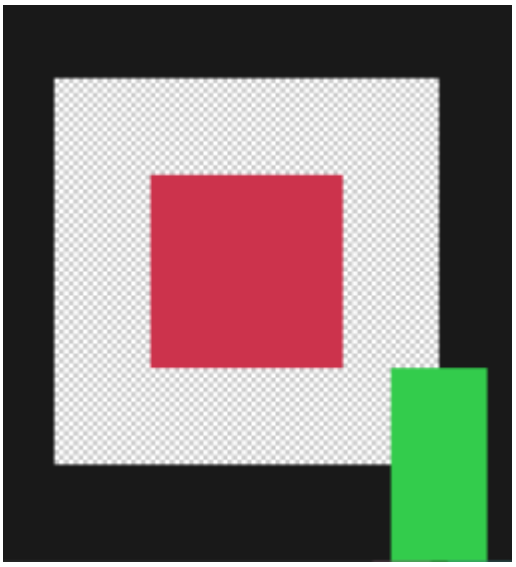
》》》》Transform 没啥要记的

》》》》Texture这节中，纹理着色器修改 color = texture(u_Texture, v_TexCoord * 1.0); 为 color = texture(u_Texture, v_TexCoord * 10.0); 怎样理解?

- color = texture(u_Texture, v_TexCoord * 1.0);
  v_TexCoord 是传入的纹理坐标，相当于在对应的纹理（0.0, 0.0)到 (1.0, 1.0) 之间的采样，乘 1 时，是相互对应的状态。
- 如果是color = texture(u_Texture, v_TexCoord * 10.0) 若v_TexCoord 范围依旧在（0.0, 0.0)到 (1.0, 1.0) 之间，乘 10 之后：
  以前采样时移动 0.1 就相当于在纹理上移动 0.1 ，是一一对应的
  现在采样时移动 0.1 就相当于在纹理上移动了 1.0，是倍数关系。故在（0.0, 0.1)到 (0.1, 0.1) 之间的采样就已经囊括了整个纹理图片。

  此时语句会导致在原有的图片之上，将纹理在相同的空间中重复的采样。
  视觉上的效果便是纹理大小缩小了 10 倍（因为之前是在 (0.0, 0.0)到 (1.0, 1.0) 之间的采样，对应过来是在（0.0, 0.1)到 (0.1, 0.1) 之间的采样 ），但填充了相同的空间。





----------Single Shader----------

》》》》为什么在未设置 white color shader 且未解绑 texture shader 的时候，绘制出来的两个本应未含纹理的方形却包含了纹理?

前提：使用了一个着色器进行绘制之后，如果只绘制颜色方形，由于结果是纹理乘以颜色的结果，所以不能不绑定一个纹理，此时只能选择使用白色纹理。
纹理方形也如此，需要使用一个白色颜色乘以纹理。
原因：因为在 Renderer2D 中，绘制颜色方形和绘制纹理方形都使用了同一个着色器对象，而且这个着色器对象本来是各调用各的（flat color shader 和 texture shader），但是在去除了 flat color shader 之后均使用 texture shader，这两个着色器都对纹理进行了采样操作。
结果：所以在颜色方形未对纹理绑定白色 white color 的时候，会受到后来纹理方形在绘制时绑定 texture shader 并激活对应纹理单元绘制纹理的影响，此时颜色方形未绑定新的着色器对象，而纹理方形（背景图案）也未对 texture shader 进行解绑，进而导致颜色方形也绘制了本应只会出现在背景上的纹理。

值得一提的是，在纹理方形 DrawQuad 代码中，texture->Bind(); 使用的是 glBindTextureUnit(slot, m_RendererID);
这一句代码的意义是 glBindTexture() 和 glActiveTexture() 两句代码之和，包含绑定纹理和激活纹理单元。
如果需要解绑，可以使用 glBindTexture(GL_TEXTURE2D,0) 来解绑的效果。如果选择使用 glBindTextureUnit(0, RendererID); 则需要指定解绑的纹理位于那个纹理单元。
区别：

glBindTexture(GL_TEXTURE_2D, 0); (GLenum target, GLuint texture)	解绑“当前”激活的纹理单元上，"指定"类型的纹理。（GL_TEXTURE_2D）
glBindTextureUnit(unit, 0); (GLuint unit, GLuint texture)	解绑“指定”纹理单元 unit 上，"所有"类型的纹理。

关于为什么要将 glBindTexture(GL_TEXTURE_2D, 0) 放在 OpenGLRendererAPI.cpp 中，是因为若放在 Renderer2D 中要额外包含 #include <glad/glad.h>，而且这违背了我们在客户端cpp中尽量不直接使用 gl 函数的想法。

》》》》SetData 这个函数的作用?
分配一个指向内存块的指针，该指针包含绘制的纹理颜色信息，并将其上传到GPU。
基本上，SetData 函数就是将 glTextureSubImage2D 这个函数从 OpenGLTexture 构造函数中分离出来的，用于加载新的或重新加载一个纹理。

》》》》SetData 是怎样绘制出来白色的?
在通过文件绘制纹理的函数中，有这么一句代码：stbi_uc* data = stbi_load(path.c_str(), &width, &height, &channels, 0);
通过 glTextureSubImage2D(m_RendererID, 0, 0, 0, m_Width, m_Height, dataFormat, GL_UNSIGNED_BYTE, data); 得以通过文件绘制。

在通过数据（例如 0xffffffff ）绘制时，我们手动填入了 Data。
然后通过 glTextureSubImage2D(m_RendererID, 0, 0, 0, m_Width, m_Height, m_DataFormat, GL_UNSIGNED_BYTE, data); 进行绘制。

》》为什么填入 0xffffffff 就确认绘制的是白色呢?
首先，关于 Data，在函数 glTextureSubImage2D 中，我们确定过绘制的规范 dataFormat 是 RGBA，也就是说传入的 Data 需要包含 R G B A 四个通道的数据，而每一个通道有 255 种选择。
所以需要填入一个数据，囊括所有可能。如二进制数（0000~1111 表示十六种可能），表示 0 ~ 255 则需要使用 8 个位数来表示（0000 0000 ~ 1111 1111）恰好足够表示一个通道的 255 种颜色比例。 故 R G B A 需要 4 个 8 位的数字在二进制下分别表示，转化到十六进制则 R G B A 需要 4 个 2 位的数字表示。（255 == 1111 1111 == 0xff)(511 == 0001 1111 1111 == 0x1ff),这就解释了 0xffffffff 的设置。
0x ff ff ff ff
R G B A

因此，我们分配了一个包含了白色这个颜色的内存，只不过通过十六进制来表示，然后传给 glTextureSubImage2D() 的最后一个参数（白色在规范化的RGBA中是{1,1,1,1}，即{255,255,255,255}）

---------- Intro to profiling ----------
》》》》std::chrono::steady_clock 是什么类型?
std::chrono::time_point<std::chrono::steady_clock>是什么意思?
- std::chrono::steady_clock 是 C++ 标准库中的一个时钟类型，用于提供稳定的、不会受系统时钟调整影响的时间测量。它通常被用来测量时间间隔，特别是需要高精度和稳定性的场合。
- std::chrono::time_point<std::chrono::steady_clock> 是一个时间点类型，表示某个特定时刻的时间点，此处具体到 steady_clock 所定义的时钟单位。

```
std::chrono::time_point<std::chrono::steady_clock> m_StartTimePoint;
```
意为：
m_StartTimePoint 类型为 std::chrono::time_point<std::chrono::steady_clock>。
用来存储一个特定时刻的时间点，记录某个操作的开始时间或者其他需要时间戳的场合。

》》》》long long start = std::chrono::time_point_cast<std::chrono::microseconds>(m_StartTimePoint).time_since_epoch().count();
的作用。
- **std::chrono::time_point_cast**: 一个模板函数，用于转换一个时间点的时间单位或者时钟类型。
- **std::chrono::microseconds:** 表示时间单位的精度为微秒
- **time_since_epoch:** 是 std::chrono::time_point 类的成员函数，用于返回该时间点与 Unix 时间（1970 年 1 月 1 日 00:00:00 UTC）的时间间隔。
  该函数返回一个 std::chrono::duration 对象，表示时间间隔。（std::chrono::duration 在此处的类型为 std::chrono::microseconds ）
- **count** 是 std::chrono::duration 类的成员函数，用于获取该时间间隔表示的时钟周期数。
  （返回类型根据 std::chrono::duration 的模板参数确定。此处根据 time_since_epoch 被确定为 std::chrono::microseconds::rep 型， 这通常是 long 或 long long 类型）

----------Visual Profiling----------
》》》》Profile 翻译

Profile	配置文件
Profiling	分析（性能分析）

》》》》__declspec 是什么意思
__declspec是MSVC中专用的关键字，用于**修饰函数、变量或类的属性。**
它可以配合dllexport或dllimport来导出或导入DLL中的函数或类。它也可以配合no return来告诉编译器函数不会返回。
同时：__declspec关键字应该出现在声明的前面。

详细说明：
__declspec(dllexport)用于Windows中的动态库中，声明导出函数、类、对象等供外面调用，省略给出.def文件,即将函数、类等声明为导出函数，供其它程序调用，作为动态库的对外接口函数、类等。

.def文件(模块定义文件)是包含一个或多个描述各种DLL属性的Module语句的文本文件。
.def文件或__declspec(dllexport)都是将公共符号导入到应用程序或从DLL导出函数。如果不提供__declspec(dllexport)导出DLL函数，则DLL需要提供.def文件。

__declspec(dllimport)用于Windows中，从别的动态库中声明导入函数、类、对象等供本动态库或exe文件使用。当你需要使用DLL中的函数时，往往不需要显示地导入函数，编译器可自动完成。不使用__declspec(dllimport)也能正确编译代码，但使用__declspec(dllimport)使编译器可以生成更好的代码。编译器之所以能够生成更好的代码，是因为它可以确定函数是否存在于DLL中，这使得编译器可以生成跳过间接寻址级别的代码，而这些代码通常会出现在跨DLL边界的函数调用中。
声明一个导入函数，是说这个函数是从别的DLL导入。一般用于使用某个DLL的exe中。

可以查看 GameEngine2（单击查看）
》》》》关于 Cherno 所说的 OnUpdate 和 OnImGuiRender 的区别怎么理解

》》》》关于 InstrumentationTimer 代码设计的思路可以观看（https://youtu.be/xlAH4dbMVnU?si=dMRPaxna5yrvS8IK）

OneNote竟然可以直接在笔记这里看视频，cool！

》》》》Instrumentor 代码
https://gist.github.com/TheCherno/31f135eea6ee729ab5f26a6908eb3a5e

》》》》关于 Instrumenter 的设计理解。

- **关于单例：可以查看笔记 》》》》什么是单例模式?**
  或者查看以下示例，理解。

单例模式是一种设计模式	其主要目的是确保类只有一个实例，并提供全局访问点。

```
#include <iostream>

class Singleton {
public:
    // 获取单例实例的静态方法
    static Singleton& getInstance() {
        // 使用静态局部变量确保只初始化一次
        static Singleton instance;
        return instance;
    }

    // 示例方法
    void showMessage() {
        std::cout << "Hello, I am a singleton instance!" << std::endl;
    }

private:
    // 私有构造函数，禁止外部直接实例化
    Singleton() {
        // 构造函数私有化，只能在类内部访问
    }

    // 禁止拷贝构造和赋值操作符
    Singleton(const Singleton&) = delete;
    Singleton& operator=(const Singleton&) = delete;
};

int main() {
    // 获取单例实例并调用方法
    Singleton& instance = Singleton::getInstance();
    instance.showMessage();

    // 尝试创建新实例会失败
    // Singleton anotherInstance; // Error: Singleton constructor is private

    return 0;
}
```

- **Instrumentor** 的代码如果被理解为是单例类的话，一般情况下单例类的构造函数应该是private，但是在 Instrumentor 中是 public？
  可能是 Cherno 失误了？我并不理解 Instrumentor()是 public 而不是 private 。
  但使用 `static Instrumentor& Get() {static Instrumentor instance;  return instance;}` 确实保证了对象只能实例化一次。

- **WriteProfile 的作用是什么?**
  该函数在 void Stop( ) 中会被调用，声明为： Instrumentor::Get().WriteProfile({ m_Name, start, end, threadID });
  通过初始化列表，我们传入某一个函数的名称、开始时间、终止时间、所处线程ID，初始化 void WriteProfile(const ProfileResult& result) 中的 result 参数，
  进而使得 WriteProfile 函数能够使用对应的信息通过文件流对 .json file 写入对应代码，因而我们使用 .json 文件查看函数运行情况。

- **if (m_ProfileCount++ > 0)**
  m_OutputStream << ",";
  的作用是什么?

  首先我们要明确 WriteProfile 函数使用的位置。
  - 先使用 BeginSession ，开启一段会话。
    BeginSession 开启会话，其中对某一个文件路径打开文件流。 m_OutputStream.open(filepath);

然后使用WriteHeader函数对打开的文件先写入头标。 WriteHeader();
最后将当前会话的名称存放下到m_CurrentSession中。 m_CurrentSession = new InstrumentationSession{ name };

- 确定正确位置开启会话之后，会话会囊括一些代码（函数 ... etc），(其中可能囊括入口点中层级较高的函数，其调用了很多低层次的函数，我们会在较靠近底层逻辑的地方使用 timer 计时)
  在入口点引用或调用的函数中，我们通过 HZ_PROFILE_SCOPE 这类宏构造一个 InstrumentorTimer 类型的对象，依据作用域对函数进行记录。
  ::Hazel::InstrumentationTimer timer##__LINE__(name);
  在作用域结束时，会调用到 Stop 函数，这里使用了 WriteProfile 记录函数信息，并将其写入 .json 文件中。
  Instrumentor::Get().WriteProfile({ m_Name, start, end, threadID });
- 使用 EndSession 。在处理充分之后，我们可以终止一个阶段的会话。并开启下一段会话，以此类推。
  对一个单独会话中的每一个 Timer 都进行了处理之后， .json 文件可能会被多次 Stop 的调用影响，故完全的记录了对应阶段代码运行时的信息。此时我们使用 EndSession 收尾。
  通过使用 WriteFooter() 对 .json 文件写下结束尾标，这样一来 .json 文件算是完整了。 WriteFooter();
  关闭文件流。 m_OutputStream.close();
  释放指针所指向的对象 delete m_CurrentSession;
  指针本身不再指向有效对象（恢复初始状态） m_CurrentSession = nullptr;
  一个Session会话中所有待处理的函数已经被操作过了，m_ProfileCount 被清零。 m_ProfileCount = 0;

- **了解了 WriteProfile 使用的位置，我们看看 if (m_ProfileCount++ > 0) m_OutputStream << ",";**
  当对第一个 Timer 进行处理时，运行到此处，在代码逻辑上会先对 m_ProfileCount 判断，然后进行自增。此时 m_ProfileCount 当然是0，故不会进行操作，并自增一次。
  但是在第二次或者说遇到第二个 PROFILE_SCOPE / PROFILE_FUNCTION 时（m_ProfileCount 属于 Instrumentor 而不是 Instrumentor Timer ，相当于在全局记录一个会话中的全部 Timer 性能分析），if (m_ProfileCount++ > 0) 是满足的，所以进行了一次操作。

- **If 中进行了什么操作呢?**
  就是多写了一个逗号。（标红的逗号）

  Eg.(.json 文件的结果应该类似于 ...)

```
{"otherData": {},"traceEvents":[
   { "cat":"function",
     "dur":" (result.End - result.Start)" ,
     "name":" name ",
     "ph":"X",
     "pid":0,
     "tid":"result.ThreadID",
     "ts":"result.Start" }
]} ,
{"otherData": {},"traceEvents":[
   { "cat":"function",
     "dur":" (result.End - result.Start)" ,
     "name":" name ",
     "ph":"X",
     "pid":0,
     "tid":"result.ThreadID",
     "ts":"result.Start" }
]}
```

- std::replace(name.begin(), name.end(), '"', '\''); 的作用?

std::replace(name.begin(), name.end(), '"', '\'')	是一个标准库算法 std::replace 的调用用于在字符串 name 中将所有出现的双引号 " 替换为单引号 '。

具体来说:

name 是一个字符串（需要是一种容器，比如 std::string）。	name.begin() 和 name.end() 表示了操作的范围，即从 name 的开始到结束。
'"'	是要被替换的值，这里是双引号。
'\''	是替换为的值，这里是单引号。

效果是防止 Name 中出现的 " 影响了 .json 文件中的 " ，导致字符串的提前终止或语法错误。

- uint32_t threadID = std::hash<std::thread::id>{}(std::this_thread::get_id()) 的作用?

std::this_thread::get_id():	这个函数调用返回当前线程的 std::thread::id 对象。
std::hash<std::thread::id>{}:	这部分利用了 std::hash 模板，特化为 std::hash<std::thread::id>。用于将接收的 std::thread::id 对象转换为哈希值。

- 哈希模板：std::hash<std::thread::id>::operator()
  std::hash 是一个通用的哈希函数模板，定义在 <functional> 头文件中。用来将任何可哈希化的数据类型映射到一个哈希值中。
- {} 的作用? 为什么?

std::hash<std::thread::id>{}	使用了大括号列表初始化的语法。

  - 在C++中，大括号初始化用于创建临时对象，并确保对象正确地初始化。对于 std::hash<std::thread::id> 来说，这会创建一个默认构造的哈希函数对象。
    一旦有了 std::hash<std::thread::id> 类型的临时对象，接下来就会调用其括号操作符 operator() ,既 std::hash<std::thread::id>::operator()(std::this_thread::get_id())，这一步会将 std::this_thread::get_id() 返回的线程 ID 作为参数传递给std::hash<std::thread::id>对象的括号操作符。
  - 虽然理论上也可以将 std::this_thread::get_id() 作为一个参数传递给 std::hash<std::thread::id> 的构造函数 / 另一个成员函数，但是标准库为了统一接口和使用习惯
    最终选择了通过重载 operator() 来实现哈希值的计算。符合C++的通用原则，使标准库的使用更加一致。

- m_OutputStream.flush(); 这句代码起到什么作用?
  根据Cherno所说，在对代码进行分析的时候，高精度的时间是有一定影响的，而且硬件与延迟也会对结果有一定影响，换言之，**数据的实时性非常重要。**
  如果不能通过 Flush 及时的将结果刷新到文件流中，性能数据（例如时间戳、执行时间等）便不会被及时写入到文件中，也有可能因为缓冲区而延迟或丢失。

》》》》 .json 文件一般用于什么用途? .json 文件的格式是什么?

```
{
"otherData": {},
"traceEvents":[
  { "cat":"function",
```

```
    "dur":" (result.End - result.Start)",
    "name":" name ",
    "ph":"X",
    "pid":0,
    "tid":"result.ThreadID",
    "ts":"result.Start" }  ]
}
```

这种格式代表什么意思？起到什么作用？

JSON（JavaScript Object Notation）文件通常用于存储和交换数据。它是一种轻量级的数据交换格式，易于人们阅读和编写，也易于机器解析和生成。

- 主要用途：

  配置文件：存储配置信息，例如应用程序的参数设置、默认值、用户首选项等。

  数据存储：存储和传输结构化数据，例如日志记录、用户数据、产品信息等。（通用且灵活）

  Web API 的数据交换格式：使用 JSON 作为数据传输的格式。客户端和服务器之间的通信可以通过 JSON 数据来进行，这在现代 Web 开发中非常常见。

  数据交换和导入导出：作为不同应用程序之间数据交换的中介格式。可以将数据从一种应用程序导出，并导入到另一种应用程序中。

JSON 文件的格式简单且直观：

- 对象（Object）：是一个无序的键值对集合。每个键值对由一个键（字符串）和一个值组成，键值对之间用逗号分隔，整个对象使用花括号 {} 包围。

```
{
  "name": "John Doe",
  "age": 30,
  "city": "New York"
}
```

- 数组（Array）：是一个有序的值集合。每个值可以是任何合法的 JSON 数据类型，值之间使用逗号分隔，整个数组使用方括号 [] 包围。

```
[
  "apple",
  "banana",
  "cherry"
]
```

- 值（Value）：可以是字符串、数值、布尔值、对象、数组或者 null。字符串需要使用双引号括起来，数值和布尔值直接写出，对象和数组则是嵌套的结构。

- （格式）

```
{
"otherData": {},
"traceEvents":[                                  事件追踪（数组对象）
   { "cat":"function",                           Category
     "dur":" (result.End - result.Start)" ,      Duration
     "name":" name ",                            Function Name
     "ph":"X",                                   Phase（相位/阶段）
     "pid":0,                                    ProgramID
     "tid":"result.ThreadID",                    ThreadID
     "ts":"result.Start" }    ]                  Start
}
```

》》》》打开 Chrome://tracing 之后

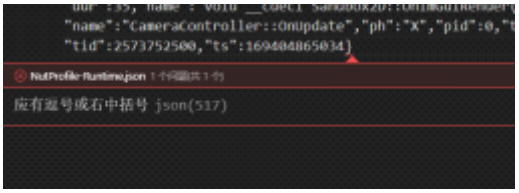
Try the new Perfetto UI! Learn more.

》》》》使用时切记！！

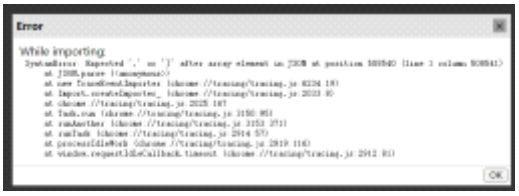
值得一提的是，运行结束时应该通过关闭窗口来实现，而不是直接单击 VS2019 中的终止按钮。否则函数

```
void WriteFooter()
{
    m_OutputStream << "]}";
    m_OutputStream.flush();
}
```

不会正常运行，导致 .json 文件结尾处没能正常的写入]} ，从而破坏了 .json 文件的格式。（如图显示报错）而且文件夹中也没有 Profile-Shutdown.json 文件。

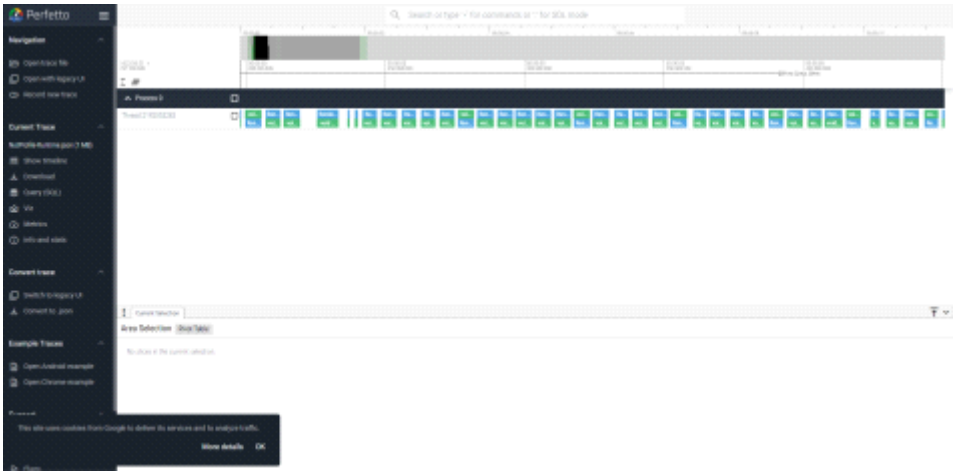


分析工具也会报错：



所以切记要正确退出程序。

》》》》Perfetto使用操作

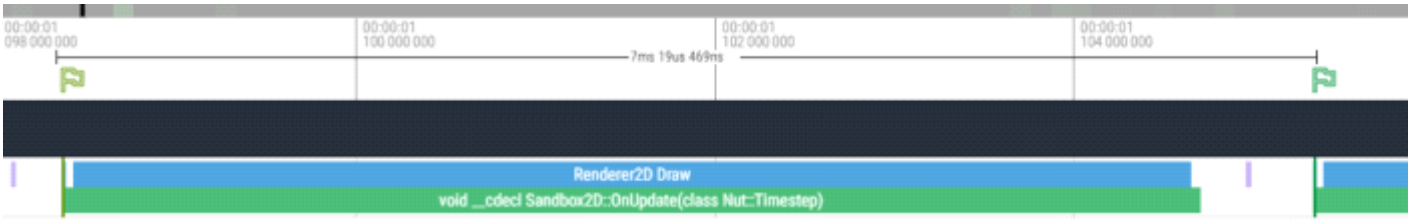


》》》》VSync的效果

每台显示器都有自己的刷新率，例如60Hz、120Hz、144Hz或165Hz。这表示显示器每秒钟可以更新多少次图像。
但GPU通常会尽可能快地生成图像帧，速率可能高于或低于显示器的刷新率。当显示器正在显示一帧图像的同时，GPU可能已经开始生成下一帧图像。
如果这两个动作不同步，显示器会在屏幕上的两个不同位置显示这两个图像，导致画面撕裂的现象，即一幅图像的上半部分和下半部分不一致，看起来像是屏幕被撕裂了一样

而Vsync的主要目的是使GPU的帧率与显示器的刷新率保持一致，当Vsync开启时，GPU会等待显示器刷新完当前帧之后，再开始渲染下一帧。
此时GPU会以显示器刷新率的倍数（如60Hz、120Hz等）来生成图像帧。
开启前（尽可能快的刷新：690us）

开启后（根据显示器的刷新165hz：7ms）



----------Instrumentation----------

》》》》没什么要记的，就是将分析函数放置在各个位置。

NUT_PROFILE_FUNCTION()	对函数使用
NUT_PROFILE_SCOPE()	对函数中的某一个/某一段代码进行分析（在分析时，其图形会出现在 function 之下）

----------Improve our 2D renderering ----------

》》》》Basicly 3 things to do : TilingFactor,Rotation,TintColor, Simply to do.

》》》》在使用时，复习到C++基础知识。
》》》》关于常量引用和非常量引用

Eg.
声明了一个函数 void func(int& arg)；但在使用的时候： func(12)；却报错说参数不匹配。
当声明一个函数 void func(int& arg)；时，它的参数 arg 是一个整型的 **非常量引用**（即可以被修改且可以被引用的值）。
这种引用需要一个可以被修改的左值作为参数，但是 func(12)；中的 12 是一个字面常量，它是一个右值，所以不能被直接绑定到非常量引用上。

解决：

使用变量而不是常量：	int x = 12; func(x); *// 可以将变量 x 传递给 func*
使用临时变量：	如果不想改变函数签名，可以创建一个临时变量并将其传递给函数： func(int(12)); *// 创建一个临时变量并传递给 func*
使用常量引用或传值：	如果函数 func 并不需要修改参数的值，可以将参数声明为常量引用或直接传递参数的值。

-

常量引用：	void func(const int& arg); *//使用常量引用* func(12); *//这样是合法的*
直接传值：	void func(int arg); *//直接传值* func(12); *// 也是合法的*

》》》》接下来我将提交一些关于Bug修复、维护、小功能实现的代码（来自Pull&Requests）

----------Made a Game in an Hour using Hazel----------

》》》》我大概看了一下，硬编码的地方还是挺多的，与后期游戏引擎的操作还是有一定区别，看看视频就行了。

----------Hazel Engine Prospect in 2020----------

》》》》Cherno 对于游戏引擎的一些展望，一些想实现的功能和一些想法。看看就行。