GameEngine7

2024年10月7日 16:21

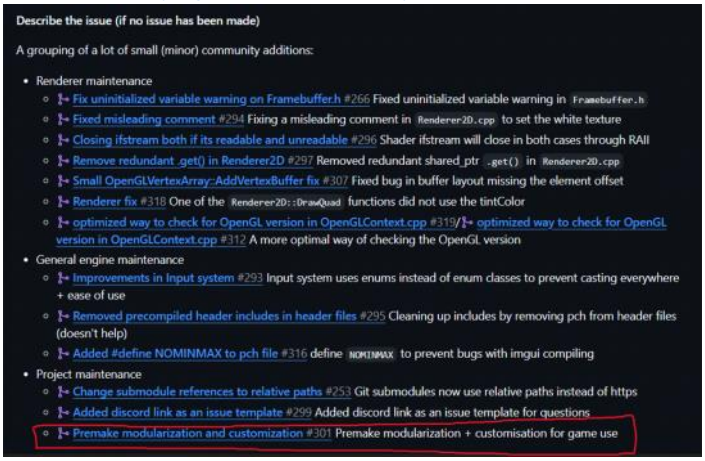
----------Saving & Loading scene----------

》》》》更改 Premake 文件构架

这一集中 Cherno 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}"
}
```

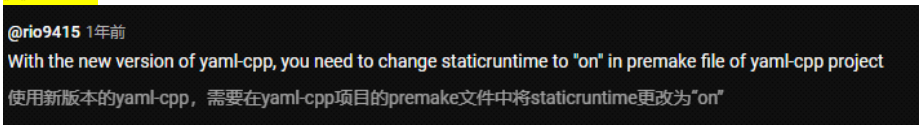
但是当我将 '-' 更改为 '_' 时，这样的问题便消失了。

```
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

》》 AND..



》》》》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。

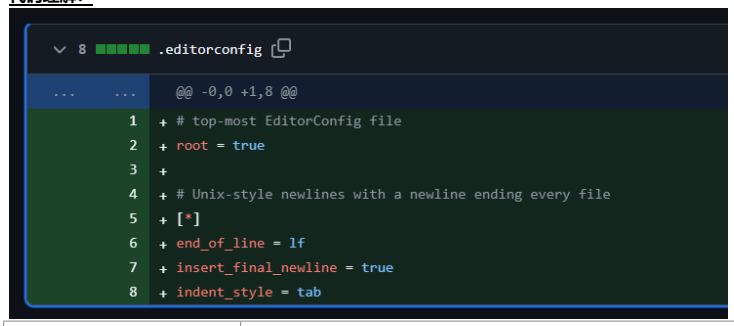
使用规范：

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/ ）

代码理解：



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：

YAML is a human-readable data serialization language that is	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

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. 来自 <https://www.redhat.com/en/topics/automation/what-is-yaml>	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。

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。	- 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/jbeder/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	#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;
	}

写入 YAML：
使用 YAML::Emitter 可以生成 YAML 文件
	#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;
	}

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. #include <yaml-cpp/yaml.h> YAML::Node node = YAML::Load("key: value"); std::string value = node["key"].as<std::string>();

Sequences 和 Maps

Sequences（序列）是一个有序列表，表示一组无命名的值。	fruits: Apple

它们在 YAML 中用短横线表示：	- Apple - Banana - Cherry

在 YAML-CPP 中，你可以这样处理序列：	Eg.
	`YAML::Node sequence = YAML::Load("[Apple, Banana, Cherry]");`
	`for (const auto& item : sequence) {`
	`  std::cout << item.as<std::string>() << std::endl;`
	`}`

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.
	`YAML::Node map = YAML::Load("name: John Doe\nage: 30");`
	`std::string name = map["name"].as<std::string>();`
	`int age = map["age"].as<int>();`

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 提供了方便的方法来将 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。

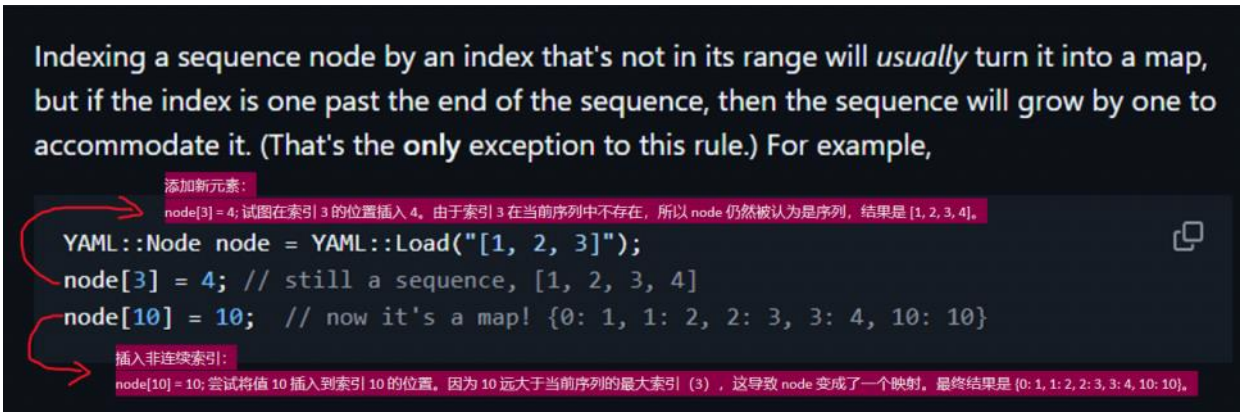
示例：从 YAML::Node 转换到原生数据类型	`YAML::Node node = YAML::Load("name: John Doe\nage: 30");`
	`std::string name = node["name"].as<std::string>();`
	`int age = node["age"].as<int>();`

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。

如果你想在该位置设置一个映射，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] = 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 node["1"] = 5，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。

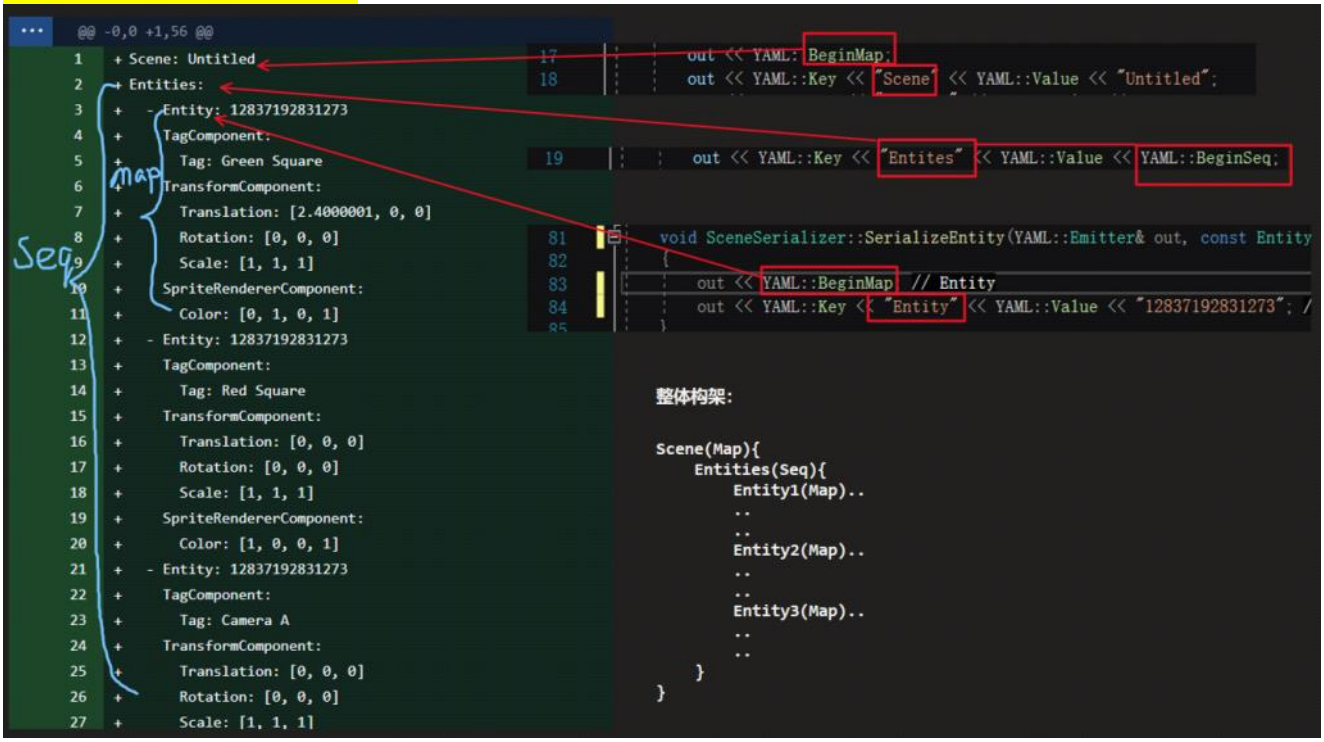
rdbuf()

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



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。

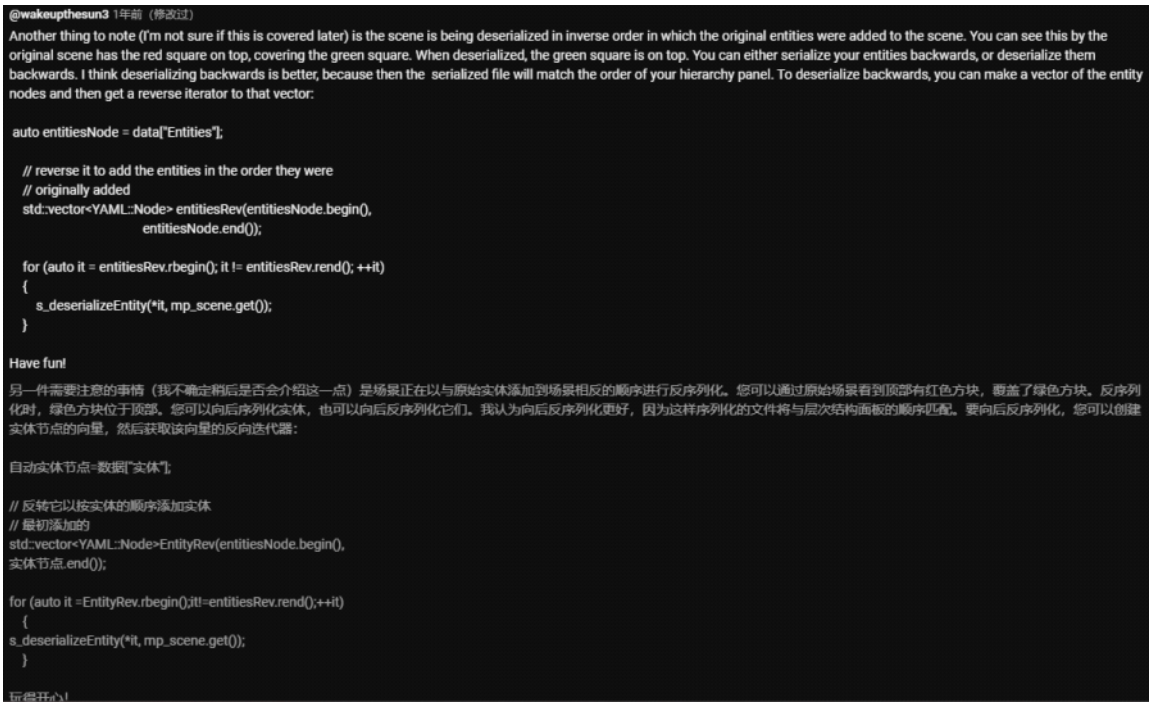
```
// According to data, we reproduce the scene
auto entities = data["Entity"];
if(entities)
{
    for (auto entity : entities)
    {
        // reproduce codes
    }
}
return true;
```

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

```
+                               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");
+                               }
```

》》》》注意事项/可改进事项

AND



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

》》》》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。