GameEngine8

2024年11月18日 11:39

---------- SPIR-V & New shader system ----------

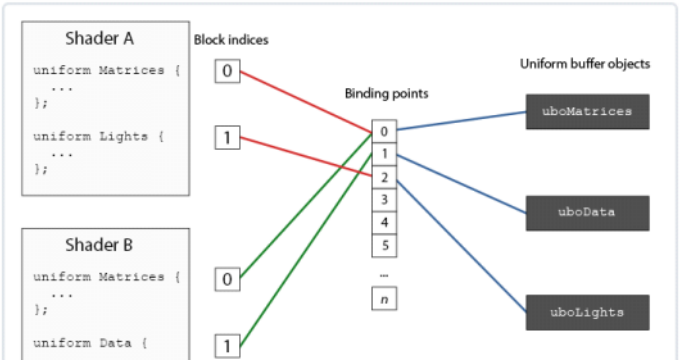
》》》》这次 Cherno 做了很多提交，所以我的笔记可能篇幅较长，但我会仔细记录。
请认真浏览。

》》》》 basic architecture layout of this episode（本集基本构架）
（截图仅供个人参考，并无侵犯版权的想法。若违反版权条款，并非本人意愿）

个人在学习过程中觉得最值得查阅的几个文档：

游戏开发者大会文档 （关于 SPRI-V 与 渲染接口 OpenGL/Vulkan 、GLSL/HLSL 之间的关系，SPIR-V 的工具及其执行流程	https://www.neilhenning.dev/wp-content/uploads/2015/03/AnIntroductionToSPIR-V.pdf
俄勒冈州立大学演示文档 （ SPIR-V 与 GLSL 之间的关系， SPIR-V 的实际使用方法：Win10 ）	https://web.engr.oregonstate.edu/~mjb/cs557/Handouts/VulkanGLSL.1pp.pdf
Vulkan 官方 Github Readme 文档 （ GLSL 与 SPIR-V 之间的映射关系，以及可以在线使用的编辑器，非常好用）	https://github.com/KhronosGroup/Vulkan-Guide/blob/main/chapters/mapping_data_to_shaders.adoc 在线文档示例（ https://godbolt.org/z/oMz58a78T ）
大阪Khronos开发者大会（SPIR-V 语言的规范，及其意义）	https://www.lunarg.com/wp-content/uploads/2023/05/SPIRV-Osaka-MAY2023.pdf

前 33 分钟，基本上讲述以下几点：

1.着色器将会支持 OpenGL 和 Vulkan ，故着色器中做了更改（涉及到 OpenGL 和 Vulkan 在着色器语法上的不同：比如 Uniform 的使用）	
2.为了避免性能浪费，并高效的使用数据/统一变量，将采用 UniformBuffer 这种高级 GLSL。 （参考文献1-来自 LearnOpenGL 教程： https://learnopengl-cn.github.io/04%20Advanced%20OpenGL/08%20Advanced%20GLSL/ ） （参考文献2-来自 Vulkan 教程： https://vulkan-tutorial.com/Uniform_buffers/Descriptor_layout_and_buffer#page_Uniform-buffer ） 建议阅读全文，这样理解更加深刻。	• Uniform buffer 使用Uniform缓冲 我们已经讨论了如何在着色器中定义Uniform块，并设定它们的内存布局了，但我们还没有讨论该如何使用它们。 首先，我们需要调用glGenBuffers，创建一个Uniform缓冲对象。一旦我们有了一个缓冲对象，我们需要将它绑定到GL_UNIFORM_BUFFER目标，并调用glBufferData，分配足够的内存。 ``` unsigned int uboExampleBlock; glGenBuffers(1, &uboExampleBlock); glBindBuffer(GL_UNIFORM_BUFFER, uboExampleBlock); glBufferData(GL_UNIFORM_BUFFER, 152, NULL, GL_STATIC_DRAW); // 分配152字节的内存 glBindBuffer(GL_UNIFORM_BUFFER, 0); ``` 现在，每当我们需要对缓冲更新或者插入数据，我们都会绑定到uboExampleBlock，并使用glBufferSubData来更新它的内存。我们只需要更新这个Uniform缓冲一次，所有使用这个缓冲的着色器就都使用的是更新后的数据了。但是，如何才能让OpenGL知道哪个Uniform缓冲对应的是哪个Uniform块呢? 在OpenGL上下文中，定义了一些绑定点(Binding Point)，我们可以将一个Uniform缓冲链接至它。在创建Uniform缓冲之后，我们将它绑定到其中一个绑定点上，并将着色器中的Uniform块绑定到相同的绑定点，把它们连接到一起。下面的这个图示展示了这个：  • Uniform buffer:

Uniform buffer 均匀缓冲

In the next chapter we'll specify the buffer that contains the UBO data for the shader, but we need to create this buffer first. We're going to copy new data to the uniform buffer every frame, so it doesn't really make any sense to have a staging buffer. It would just add extra overhead in this case and likely degrade performance instead of improving it.

在下一章中，我们将指定包含着色器 UBO 数据的缓冲区，但我们需要首先创建此缓冲区。我们将每帧将新数据复制到统一缓冲区，因此拥有暂存缓冲区实际上没有任何意义。在这种情况下，它只会增加额外的开销，并且可能会降低性能而不是提高性能。

We should have multiple buffers, because multiple frames may be in flight at the same time and we don't want to update the buffer in preparation of the next frame while a previous one is still reading from it! Thus, we need to have as many uniform buffers as we have frames in flight, and write to a uniform buffer that is not currently being read by the GPU

我们应该有多个缓冲区，因为多个帧可能同时在飞行，我们不想在前一帧仍在读取时更新缓冲区以准备下一帧！因此，我们需要拥有与飞行中的帧一样多的统一缓冲区，并写入 GPU 当前未读取的统一缓冲区

To that end, add new class members for `uniformBuffers`, and `uniformBuffersMemory`:

为此，为 `uniformBuffers` 和 `uniformBuffersMemory` 添加新的类成员：

```
VkBuffer indexBuffer;
VkDeviceMemory indexBufferMemory;

std::vector<VkBuffer> uniformBuffers;
std::vector<VkDeviceMemory> uniformBuffersMemory;
std::vector<void*> uniformBuffersMapped;
```

Similarly, create a new function `createUniformBuffers` that is called after `createIndexBuffer` and allocates the buffers:

类似地，创建一个新函数 `createUniformBuffers` ，该函数在 `createIndexBuffer` 之后调用并分配缓冲区：

```
void initVulkan() {
    ...
    createVertexBuffer();
    createIndexBuffer();
    createUniformBuffers();
```

3.OpenGL 和 Vulkan 在着色器语言上的使用规范，还有不同之处。

参考文献：OpenGL教程（ https://learnopengl-cn.github.io/02%20Lighting/03%20Materials/ ）

参考文献：俄勒冈州立大学演示文件《 GLSL For Vulkan 》（ https://eecs.oregonstate.edu/~mjb/cs557/Handouts/VulkanGLSL.1pp.pdf ）

附录：

参考文献：Github 中文 Readme（ https://github.com/zenny-chen/GLSL-for-Vulkan ）

参考文献： Vulkan 教程官网（ https://vulkan-tutorial.com/Introduction ）

- GLSG 中的结构体示例：

```
#version 330 core
struct Material {
    vec3 ambient;
    vec3 diffuse;
    vec3 specular;
    float shininess;
};

uniform Material material;
```

在片段着色器中，我们创建一个结构体(Struct)来储存物体的材质属性。我们也可以把它们储存为独立的uniform值，但是作为一个结构体来储存会更有条理一些。我们首先定义结构体的布局(Layout)，然后简单地以刚创建的结构体作为类型声明一个uniform变量。

- 如果想查看 Vulkan API 在编写着色器时使用 GLSL 的语法规范，可以查看 Github 仓库（中文： https://github.com/zenny-chen/GLSL-for-Vulkan ）
或者在 Vulkan 教程官网中搜寻（ https://vulkan-tutorial.com/Introduction ）

- 不同之处：

How Vulkan GLSL Differs from OpenGL GLSL 4

Detecting that a GLSL Shader is being used with Vulkan/SPIR-V:

- In the compiler, there is an automatic
  #define VULKAN 100

Vulkan Vertex and Instance indices:	OpenGL uses:
gl_VertexIndex	gl_VertexID
gl_InstanceIndex	gl_InstanceID

- Both are 0-based

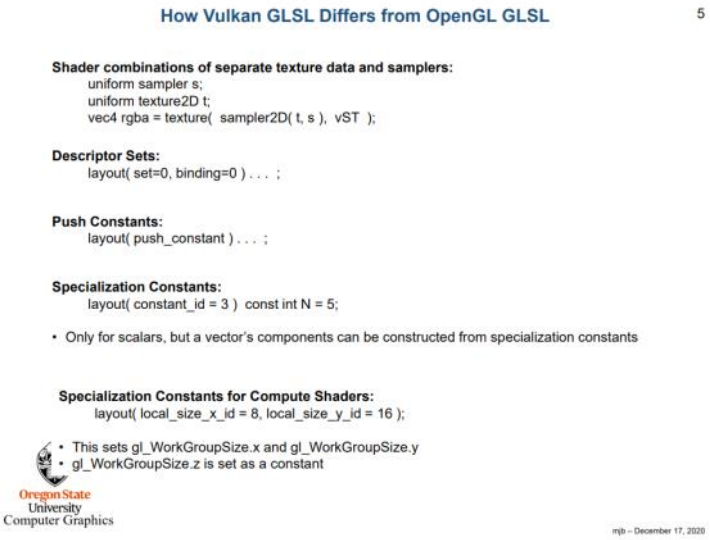
gl_FragColor:

- In OpenGL, gl_FragColor broadcasts to all color attachments
- In Vulkan, it just broadcasts to color attachment location #0
- Best idea: don't use it at all – explicitly declare out variables to have specific location numbers

Oregon State University Computer Graphics

mjb – December 17, 2020

How Vulkan GLSL Differs from OpenGL GLSL 5



4.SPIR-V 的使用思路，使用逻辑。

参考文献：SPIR-V 官网（ https://www.khronos.org/api/index_2017/spir ）

参考文献：Vulkan 教程（ https://vulkan-tutorial.com/Drawing_a_triangle/Graphics_pipeline_basics/Shader_modules ）

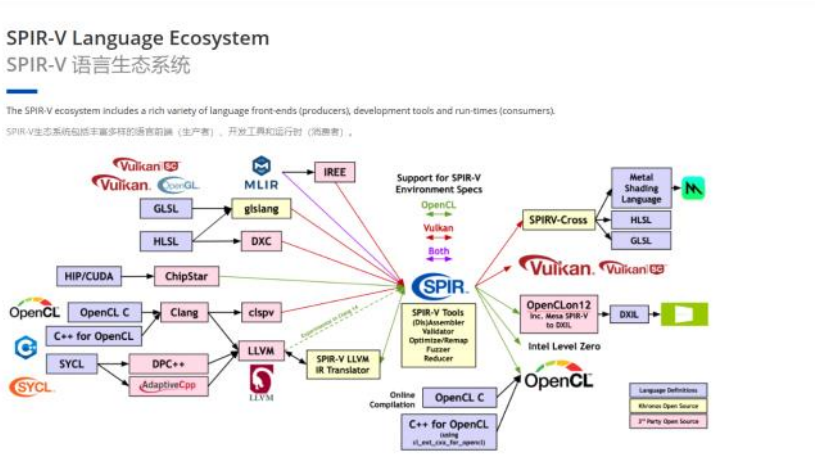
参考文献： Vulkan 指南
（ https://docs.vulkan.org/guide/latest/what_is_spirv.html ）

参考文献：俄勒冈州立大学演示文件（ https://web.engr.oregonstate.edu/~mjb/cs557/Handouts/VulkanGLSL.1pp.pdf ）

参考文献：2016 年 3 月 - 游戏开发者大会
（https://www.neilhenning.dev/wp-content/uploads/2015/03/AnIntroductionToSPIR-V.pdf ）

附件：关于 SPIR-V 也可以参考 SPIR-V 的 github 仓库:
（ https://github.com/KhronosGroup/SPIRV-Guide ）

• SPIR-V 的生态系统:



• SPIR-V 的概念:

Unlike earlier APIs, shader code in Vulkan has to be specified in a bytecode format as opposed to human-readable syntax like GLSL and HLSL. This bytecode format is called SPIR-V and is designed to be used with both Vulkan and OpenCL (both Khronos APIs). It is a format that can be used to write graphics and compute shaders, but we will focus on shaders used in Vulkan's graphics pipelines in this tutorial.

与早期的 API 不同，Vulkan 中的着色器代码必须以字节码格式指定，而不是像GLSL和HLSL这样的人类可读语法。这种字节码格式称为SPIR-V，旨在与 Vulkan 和 OpenCL（均为 Khronos API）一起使用。它是一种可用于编写图形和计算着色器的格式，但在本教程中我们将重点关注 Vulkan 图形管道中使用的着色器。

Vulkan Guide / Logistics Overview / What is SPIR-V

What is SPIR-V 什么是 SPIR-V

NOTE

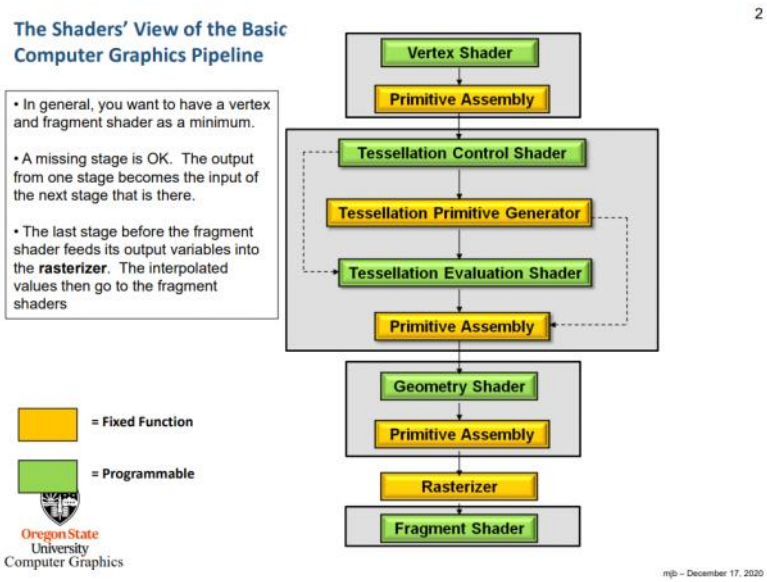
Please read the SPIRV-Guide for more in detail information about SPIR-V

请阅读SPIRV 指南，了解有关 SPIR-V 的更多详细信息

SPIR-V is a binary intermediate representation for graphical-shader stages and compute kernels. With Vulkan, an application can still write their shaders in a high-level shading language such as GLSL or HLSL, but a SPIR-V binary is needed when using vkCreateShaderModule. Khronos has a very nice white paper about SPIR-V and its advantages, and a high-level description of the representation. There are also two great Khronos presentations from Vulkan DevDay 2016 here and here (video of both).

SPIR-V是图形着色器阶段和计算内核的二进制中间表示。使用 Vulkan，应用程序仍然可以使用高级着色语言（例如 GLSL 或HLSL）编写着色器，但使用vkCreateShaderModule时需要 SPIR-V 二进制文件。 Khronos 有一份关于 SPIR-V 及其优点的非常好的白皮书，以及对表示的高级描述。这里和这里还有 2016 年 Vulkan DevDay 的两场精彩的 Khronos 演示（两者的视频）。

• SPIR-V 管线:



• SPIR-V 的使用流程：
参考接下来的笔记：（ 实际使用流程： ）
或者参考（ https://www.neilhenning.dev/wp-content/uploads/2015/03/AnIntroductionToSPIR-V.pdf ）

》》》》 SPIR-V SPIR-V ？什么是 SPIR-V？ SPIR-V SPIR-V

SPIR-V 简介

SPIR-V (Standard Portable Intermediate Representation for Vulkan) 是一种低级中间表示语言（Intermediate Representation, IR），通常是由高层语言（如 GLSL 或 HLSL）编译而成，主要用于图形和计算程序的编译。（开发者写的 GLSL 或 HLSL 代码会被编译成 SPIR-V，然后交给 Vulkan 或 OpenCL 、OpenGL等图形计算 API 来执行。）

SPIR-V 允许开发者编写更加底层的图形或计算代码，并通过它来与图形硬件交互。

实际使用流程：

OpenGL	通常使用 GLSL（OpenGL Shading Language）来编写着色器代码
Vulkan	使用 SPIR-V（Standard Portable Intermediate Representation for Vulkan）作为着色器的中间语言。

为什么说 SPIR-V 是中间语言?
在 Vulkan 中，着色器代码（如顶点着色器、片段着色器等）首先用高级语言（如 GLSL 或 HLSL）编写，然后通过工具（如 glslang）编译成 SPIR-V 字节码，最后通过 Vulkan API 加载并使用这些字节码。

OpenGL 与 SPIR-V的工作模式：	在 Vulkan 出现之前，OpenGL 是主要的图形 API，GLSL 是 OpenGL 使用的着色器语言。随着 Vulkan 的推出，SPIR-V 成为了 Vulkan 着色器的中间表示，SPIR-V也被引入到 OpenGL 中。 尽管 OpenGL 一直使用 GLSL 作为着色器语言，但 OpenGL 4.5 及更高版本已经支持通过 SPIR-V 加载编译好的着色器二进制文件。 这意味着OpenGL 虽然仍旧使用 GLSL 来编写着色器，但编译过程可以将 GLSL 代码转化为 SPIR-V，之后在 OpenGL 中加载 SPIR-V 二进制代码进行执行。这一过程通过 glslang（Khronos 提供的 GLSL 编译器）实现。
Vulkan 与 SPIR-V 的工作模式： 参考文献：游戏开发者大会2016（ https://www.neilhenning.dev/wp-content/uploads/2015/03/AnIntroductionToSPIR-V.pdf）	Vulkan 作为低级 API，要求所有着色器都以 SPIR-V 格式存在。由于着色器源代码通常使用高级着色器语言（如 GLSL 或 HLSL）编写，所以需要先编译成 SPIR-V 二进制格式，然后将该 SPIR-V 二进制代码上传到 GPU 进行执行。 作用：SPIR-V 使 Vulkan 可以实现跨平台的着色器支持，依靠 SPIR-V 这种中间语言，着色器能够在不同平台和硬件上正常运行。SPIR-V 规范的语言比纯文本的着色器语言（如 GLSL）更接近底层硬件，便于优化和硬件加速。 示例： `; SPIR-V` `; Version: 1.0` `; Generator: Khronos Glslang Reference Front End; 1` `; Bound: 14` `; Schema: 0` `OpCapability Shader` `%1 = OpExtInstImport "GLSL.std.450"` `OpMemoryModel Logical GLSL450` `OpEntryPoint Fragment %4 "main" %9` `OpExecutionMode %4 OriginUpperLeft` `OpSource GLSL 450` `OpName %4 "main"` `OpName %9 "out_colour"`

uction to SPIR-V.pdf)	; SPIR-V ; Version: 1.0 ; Generator: Khronos Glslang Reference Front End; 1 ; Bound: 14 ; Schema: 0 OpCapability Shader %1 = OpExtInstImport "GLSL.std.450" OpMemoryModel Logical GLSL450 OpEntryPoint Fragment %4 "main" %9 OpExecutionMode %4 OriginUpperLeft OpSource GLSL 450 OpName %4 "main" OpName %9 "out_colour" OpDecorate %9 Location 0 %2 = OpTypeVoid %3 = OpTypeFunction %2 %6 = OpTypeFloat 32 %7 = OpTypeVector %6 4 %8 = OpTypePointer Output %7 %9 = OpVariable %8 Output %10 = OpConstant %6 0.4 %11 = OpConstant %6 0.8 %12 = OpConstant %6 1 %13 = OpConstantComposite %7 %10 %10 %11 %12 %4 = OpFunction %2 None %3 %5 = OpLabel OpStore %9 %13 OpReturn OpFunctionEnd

实际使用实例：

1. GLSL 源代码编写	首先，编写 GLSL 源代码。这些 GLSL 代码通常包括顶点着色器、片段着色器、计算着色器等。 示例：GLSL 着色器 `#version 450` `out vec4 FragColor;` `void main() {` `FragColor = vec4(1.0, 0.0, 0.0, 1.0); // 输出红色` `}`
2. GLSL 编译为 SPIR-V	将 GLSL 源代码转换为 SPIR-V 二进制格式，得到一个平台无关的二进制文件，这意味着 SPIR-V 代码可以在不同的硬件和操作系统上运行。 工具1： glslang（Khronos 提供的编译器，广泛用于将 GLSL 转换为 SPIR-V）。 编译过程： GLSL 代码通过 glslang 编译器进行语法检查和优化，并得到一个二进制文件。 工具2： 你也可以使用命令行工具 glslangValidator 来编译 GLSL 代码。 编译过程： 使用命令：glslangValidator -V shader.glsl -o shader.spv 这将会把 shader.glsl 编译成 shader.spv，即 SPIR-V 二进制文件。
3. 加载 SPIR-V 到 Vulkan 或 OpenGL 中	3.1 在 OpenGL 中使用 SPIR-V 前情提要： 从 OpenGL 4.5 开始，OpenGL 也支持通过 SPIR-V 加载编译好的着色器二进制文件。流程与 Vulkan 类似，只不过 OpenGL 在内部做了更多的高层封装。 加载过程： 示例： `GLuint program = glCreateProgram();` `// 加载 SPIR-V 二进制文件` `GLuint shader = glCreateShader(GL_VERTEX_SHADER);` `glShaderBinary(1, &shader, GL_SHADER_BINARY_FORMAT_SPIR_V, spirvData, spirvDataSize);` `glSpecializeShader(shader, "main", 0, nullptr, nullptr);` `// 绑定和链接程序` `glAttachShader(program, shader);` `glLinkProgram(program);` --- 3.2 在 Vulkan 中使用 SPIR-V 加载过程： 创建一个 VkShaderModule 对象，该对象包含 SPIR-V 二进制代码。 使用 SPIR-V 二进制代码来创建 Vulkan 着色器管线（例如，创建顶点着色器和片段着色器的管线）。 示例：Vulkan 使用 SPIR-V `// 加载 SPIR-V 文件（假设你已经将 shader.spv 文件加载为二进制数据）` `VkShaderModuleCreateInfo createInfo = {};` `createInfo.sType = VK_STRUCTURE_TYPE_SHADER_MODULE_CREATE_INFO;`

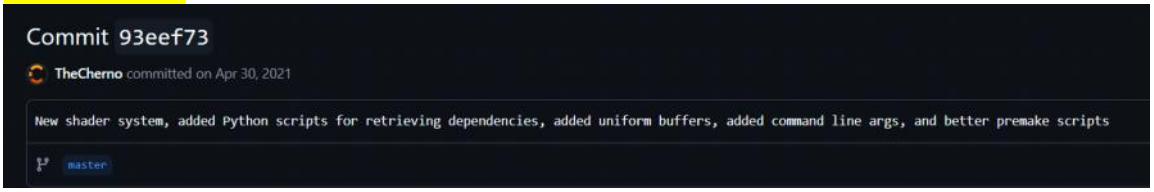
	createInfo.codeSize = shaderData.size(); createInfo.pCode = reinterpret_cast<const uint32_t*>(shaderData.data()); // 创建着色器模块 VkShaderModule shaderModule; VkResult result = vkCreateShaderModule(device, &createInfo, nullptr, &shaderModule); // 使用这个 shaderModule 来创建图形管线
4. 执行着色器程序	在 OpenGL 中，SPIR-V 着色器程序被链接到程序对象中，并通过调用 glUseProgram 来激活该程序，之后通过绘制调用来执行。 在 Vulkan 中， 着色器被绑定到渲染管线或计算管线中，随后可以通过绘制命令（例如 vkCmdDraw）或计算命令（例如 vkCmdDispatch）来执行。

》》》》上述涉及语言的纵向对比图

GLSL	#version 330 core in vec3 fragColor; // 从顶点着色器传递过来的颜色 out vec4 FragColor; // 输出颜色到屏幕 void main() { FragColor = vec4(fragColor, 1.0); // 输出最终颜色 }
SPIR-V SPIR-V 本身的核心是一个二进制格式，然而为了便于开发和调试，SPIR-V 也可以以类似汇编语言的文本形式表达，这种形式通常称为 SPIR-V Assembly。 它是 SPIR-V 的一种可读性较好的文本表示方式，开发者可以通过这种形式来编写、调试和优化 SPIR-V 代码，然后再将其转换为二进制格式以供图形 API 使用。 实际上，SPIR-V Assembly 代码最终还是会通过工具（如 spirv-as）转化为二进制格式，供 Vulkan 或 OpenGL 使用。	SPIR-V 0302 2307 0000 0100 0100 0800 0e00 0000 0000 0000 1100 0200 0100 0000 0b00 0600 0100 0000 474c 534c 2e73 7464 2e34 3530 0000 0000 0e00 0300 0000 0000 0100 0000 0f00 0600 0400 0000 0400 0000 6d61 696e 0000 0000 0900 0000 1000 0300 0400 0000 0700 0000 0300 0300 0200 0000 8c00 0000 0500 0400 0400 0000 6d61 696e 0000 0000 0500 0600 0900 0000 676c 5f46 7261 6743 6f6c 6f72 0000 0000 1300 0200 0200 0000 2100 0300 0300 0000 0200 0000 1600 0300 0600 0000 2000 0000 1700 0400 0700 0000 0600 0000 0400 0000 2000 0400 0800 0000 0300 0000 0700 0000 3b00 0400 0800 0000 0900 0000 0300 0000 2b00 0400 0600 0000 0a00 0000 cdcc cc3e 2b00 0400 0600 0000 0b00 0000 cdcc 4c3f 2b00 0400 0600 0000 0c00 0000 0000 803f 2c00 0700 0700 0000 0d00 0000 0a00 0000 0a00 0000 0b00 0000 0c00 0000 3600 0500 0200 0000 0400 0000 0000 0000 0300 0000 f800 0200 0500 0000 3e00 0300 0900 0000 0d00 0000 fd00 0100 3800 0100 SPIR-V Assembly ; SPIR-V ; Version: 1.0 ; Generator: Khronos Glslang Reference Front End; 1 ; Bound: 14 ; Schema: 0 OpCapability Shader %1 = OpExtInstImport "GLSL.std.450" OpMemoryModel Logical GLSL450 OpEntryPoint Fragment %4 "main" %9 OpExecutionMode %4 OriginUpperLeft OpSource GLSL 450 OpName %4 "main" OpName %9 "out_colour" OpDecorate %9 Location 0 %2 = OpTypeVoid %3 = OpTypeFunction %2 %6 = OpTypeFloat 32 %7 = OpTypeVector %6 4 %8 = OpTypePointer Output %7 %9 = OpVariable %8 Output %10 = OpConstant %6 0.4 %11 = OpConstant %6 0.8 %12 = OpConstant %6 1 %13 = OpConstantComposite %7 %10 %10 %11 %12 %4 = OpFunction %2 None %3 %5 = OpLabel OpStore %9 %13 OpReturn OpFunctionEnd

OpenGL	``` GLuint shaderProgram = glCreateProgram(); glAttachShader(shaderProgram, vertexShader); glAttachShader(shaderProgram, fragmentShader); glLinkProgram(shaderProgram); glUseProgram(shaderProgram); // 主要渲染循环 while (!glfwWindowShouldClose(window)) { glClear(GL_COLOR_BUFFER_BIT); glUseProgram(shaderProgram); ```
Vulkan	``` VkInstance instance; VkApplicationInfo appInfo = {}; appInfo.sType = VK_STRUCTURE_TYPE_APPLICATION_INFO; appInfo.pApplicationName = "Vulkan 示例"; appInfo.applicationVersion = VK_MAKE_VERSION(1, 0, 0); appInfo.pEngineName = "No Engine"; appInfo.engineVersion = VK_MAKE_VERSION(1, 0, 0); appInfo.apiVersion = VK_API_VERSION_1_0; VkInstanceCreateInfo createInfo = {}; createInfo.sType = VK_STRUCTURE_TYPE_INSTANCE_CREATE_INFO; createInfo.pApplicationInfo = &appInfo; ```

》》》》着色器中的更改



以下是详情解释：

1 premake脚本更改 （and better premake scripts）	``` + -- Hazel Dependencies + + VULKAN_SDK = os.getenv("VULKAN_SDK") ``` ``` + IncludeDir["shaderc"] = "%{wks.location}/Hazel/vendor/shaderc/include" + IncludeDir["SPIRV_Cross"] = "%{wks.location}/Hazel/vendor/SPIRV-Cross" + IncludeDir["VulkanSDK"] = "%{VULKAN_SDK}/Include" + + LibraryDir = {} + + LibraryDir["VulkanSDK"] = "%{VULKAN_SDK}/Lib" + LibraryDir["VulkanSDK_Debug"] = "%{wks.location}/Hazel/vendor/VulkanSDK/Lib" + + Library = {} + Library["Vulkan"] = "%{LibraryDir.VulkanSDK}/vulkan-1.lib" + Library["VulkanUtils"] = "%{LibraryDir.VulkanSDK}/VkLayer_utils.lib" + + Library["ShaderC_Debug"] = "%{LibraryDir.VulkanSDK_Debug}/shaderc_sharedd.lib" + Library["SPIRV_Cross_Debug"] = "%{LibraryDir.VulkanSDK_Debug}/spirv-cross-cored.lib" + Library["SPIRV_Cross_GLSL_Debug"] = "%{LibraryDir.VulkanSDK_Debug}/spirv-cross-glsld.lib" + Library["SPIRV_Tools_Debug"] = "%{LibraryDir.VulkanSDK_Debug}/SPIRV-Toolsd.lib" + + Library["ShaderC_Release"] = "%{LibraryDir.VulkanSDK}/shaderc_shared.lib" + Library["SPIRV_Cross_Release"] = "%{LibraryDir.VulkanSDK}/spirv-cross-core.lib" + Library["SPIRV_Cross_GLSL_Release"] = "%{LibraryDir.VulkanSDK}/spirv-cross-glsl.lib" ```

premake5.lua

```
@@ -1,4 +1,5 @@
 include "./vendor/premake/premake_customization/solution_items.lua"
+include "Dependencies.lua"

 workspace "Hazel"
         architecture "x86_64"
@@ -23,17 +24,6 @@ workspace "Hazel"

 outputdir = "%{cfg.buildcfg}-%{cfg.system}-%{cfg.architecture}"

--- Include directories relative to root folder (solution directory)
-IncludeDir = {}
-IncludeDir["GLFW"] = "%{wks.location}/Hazel/vendor/GLFW/include"
-IncludeDir["Glad"] = "%{wks.location}/Hazel/vendor/Glad/include"
-IncludeDir["ImGui"] = "%{wks.location}/Hazel/vendor/imgui"
-IncludeDir["glm"] = "%{wks.location}/Hazel/vendor/glm"
-IncludeDir["stb_image"] = "%{wks.location}/Hazel/vendor/stb_image"
-IncludeDir["entt"] = "%{wks.location}/Hazel/vendor/entt/include"
-IncludeDir["yaml_cpp"] = "%{wks.location}/Hazel/vendor/yaml-cpp/include"
-IncludeDir["ImGuizmo"] = "%{wks.location}/Hazel/vendor/ImGuizmo"
-
 group "Dependencies"
         include "vendor/premake"
         include "Hazel/vendor/GLFW"
```

------ ------ ------ ------ ------ ------ -------

Hazelnut/premake5.lua

```
@@ -2,7 +2,7 @@ project "Hazelnut"
         kind "ConsoleApp"
         language "C++"
         cppdialect "C++17"
-        staticruntime "on"
+        staticruntime "off"

         targetdir ("%{wks.location}/bin/" .. outputdir .. "/%{prj.name}")
         objdir ("%{wks.location}/bin-int/" .. outputdir .. "/%{prj.name}")
```

------ ------ ------ ------ ------ ------ -------

Hazel/premake5.lua

```
@@ -2,7 +2,7 @@ project "Hazel"
         kind "StaticLib"
         language "C++"
         cppdialect "C++17"
-        staticruntime "on"
+        staticruntime "off"

         targetdir ("%{wks.location}/bin/" .. outputdir .. "/%{prj.name}")
         objdir ("%{wks.location}/bin-int/" .. outputdir .. "/%{prj.name}")
@@ -40,7 +40,8 @@ project "Hazel"
                 "%{IncludeDir.stb_image}",
                 "%{IncludeDir.entt}",
                 "%{IncludeDir.yaml_cpp}",
-                "%{IncludeDir.ImGuizmo}"
+                "%{IncludeDir.ImGuizmo}",
+                "%{IncludeDir.VulkanSDK}"
         }

         links
@@ -67,12 +68,33 @@ project "Hazel"
                 runtime "Debug"
                 symbols "on"

+                links
+                {
+                        "%{Library.ShaderC_Debug}",
+                        "%{Library.SPIRV_Cross_Debug}",
+                        "%{Library.SPIRV_Cross_GLSL_Debug}"
+                }
```

Hazel/premake5.lua

```
+                       {
+                               "%{Library.ShaderC_Debug}",
+                               "%{Library.SPIRV_Cross_Debug}",
+                               "%{Library.SPIRV_Cross_GLSL_Debug}"
+               }
+
        filter "configurations:Release"
                defines "HZ_RELEASE"
                runtime "Release"
                optimize "on"

+               links
+               {
+                       "%{Library.ShaderC_Release}",
+                       "%{Library.SPIRV_Cross_Release}",
+                       "%{Library.SPIRV_Cross_GLSL_Release}"
+               }
+
        filter "configurations:Dist"
                defines "HZ_DIST"
                runtime "Release"
                optimize "on"
+
+               links
+               {
+                       "%{Library.ShaderC_Release}",
+                       "%{Library.SPIRV_Cross_Release}",
+                       "%{Library.SPIRV_Cross_GLSL_Release}"
+               }
```

2 py脚本
（Python scripts for retrieving dependencies）

scripts/CheckPython.py

```
@@ -0,0 +1,18 @@
+ import subprocess
+ import pkg_resources
+
+ def install(package):
+     print(f"Installing {package} module...")
+     subprocess.check_call(['python', '-m', 'pip', 'install', package])
+
+ def ValidatePackage(package):
+     required = { package }
+     installed = {pkg.key for pkg in pkg_resources.working_set}
+     missing = required - installed
+
+     if missing:
+         install(package)
+
+ def ValidatePackages():
+     ValidatePackage('requests')
+     ValidatePackage('fake-useragent')
```

1. 确保在执行过程中 requests 和 fake-useragent 这两个模块已经安装。如果没有安装，它会自动使用 pip 安装它们。

scripts/Setup.py

```
@@ -0,0 +1,20 @@
+ import os
+ import subprocess
+ import CheckPython
+
+ # Make sure everything we need is installed
+ CheckPython.ValidatePackages()
+
+ import Vulkan
+
+ # Change from Scripts directory to root
+ os.chdir('../')
+
+ if (not Vulkan.CheckVulkanSDK()):
+     print("Vulkan SDK not installed.")
+
+ if (not Vulkan.CheckVulkanSDKDebugLibs()):
+     print("Vulkan SDK debug libs not found.")
+
+ print("Running premake...")
+ subprocess.call(["vendor/premake/bin/premake5.exe", "vs2019"])
```

1.确保所需的 Python 包已经安装。
2.检查 Vulkan SDK 是否安装，并确保 Vulkan SDK 的调试库存在。
3.改变当前工作目录到项目根目录。
4.使用 premake 工具生成 Visual Studio 2019 项目的构建文件。

scripts/Utils.py

```
@@ -0,0 +1,41 @@
+ import requests
+ import sys
+ import time
+
+ from fake_useragent import UserAgent
+
+ def DownloadFile(url, filepath):
+     with open(filepath, 'wb') as f:
+         ua = UserAgent()
+         headers = {'User-Agent': ua.chrome}
+         response = requests.get(url, headers=headers, stream=True)
+         total = response.headers.get('content-length')
+
+         if total is None:
+             f.write(response.content)
+         else:
+             downloaded = 0
+             total = int(total)
+             startTime = time.time()
+             for data in response.iter_content(chunk_size=max(int(total/1000), 1024*1024)):
+                 downloaded += len(data)
+                 f.write(data)
+                 done = int(50*downloaded/total)
+                 percentage = (downloaded / total) * 100
+                 elapsedTime = time.time() - startTime
+                 avgKBPerSecond = (downloaded / 1024) / elapsedTime
+                 avgSpeedString = '{:.2f} KB/s'.format(avgKBPerSecond)
+                 if (avgKBPerSecond > 1024):
+                     avgMBPerSecond = avgKBPerSecond / 1024
+                     avgSpeedString = '{:.2f} MB/s'.format(avgMBPerSecond)
```

DownloadFile(url, filepath) 函数的作用是从指定 URL 下载文件，并显示实时的下载进度（包括下载进度条和速度）。
YesOrNo() 函数用于与用户进行交互，获取用户的确认输入，返回布尔值表示“是”或“否”。

scripts/Vulkan.py

```
@@ -0,0 +1,60 @@
+ import os
+ import subprocess
+ import sys
+ from pathlib import Path
+
+ import Utils
+
+ from io import BytesIO
+ from urllib.request import urlopen
+ from zipfile import ZipFile
+
+ VULKAN_SDK = os.environ.get('VULKAN_SDK')
+ VULKAN_SDK_INSTALLER_URL = 'https://sdk.lunarg.com/sdk/download/1.2.170.0/windows/vulkan_sdk.exe'
+ HAZEL_VULKAN_VERSION = '1.2.170.0'
+ VULKAN_SDK_EXE_PATH = 'Hazel/vendor/VulkanSDK/VulkanSDK.exe'
+
+ def InstallVulkanSDK():
+     print('Downloading {} to {}'.format(VULKAN_SDK_INSTALLER_URL, VULKAN_SDK_EXE_PATH))
+     Utils.DownloadFile(VULKAN_SDK_INSTALLER_URL, VULKAN_SDK_EXE_PATH)
+     print("Done!")
+     print("Running Vulkan SDK installer...")
+     os.startfile(os.path.abspath(VULKAN_SDK_EXE_PATH))
+     print("Re-run this script after installation")
+
+ def InstallVulkanPrompt():
```

用于检查和安装 Vulkan SDK

InstallVulkanSDK(): 下载并运行 Vulkan SDK 安装程序。
InstallVulkanPrompt(): 提示用户是否安装 Vulkan SDK。
CheckVulkanSDK(): 检查 Vulkan SDK 是否安装并且版本是否正确。
CheckVulkanSDKDebugLibs(): 检查 Vulkan SDK 的调试库是否存在，如果缺失则下载并解压。

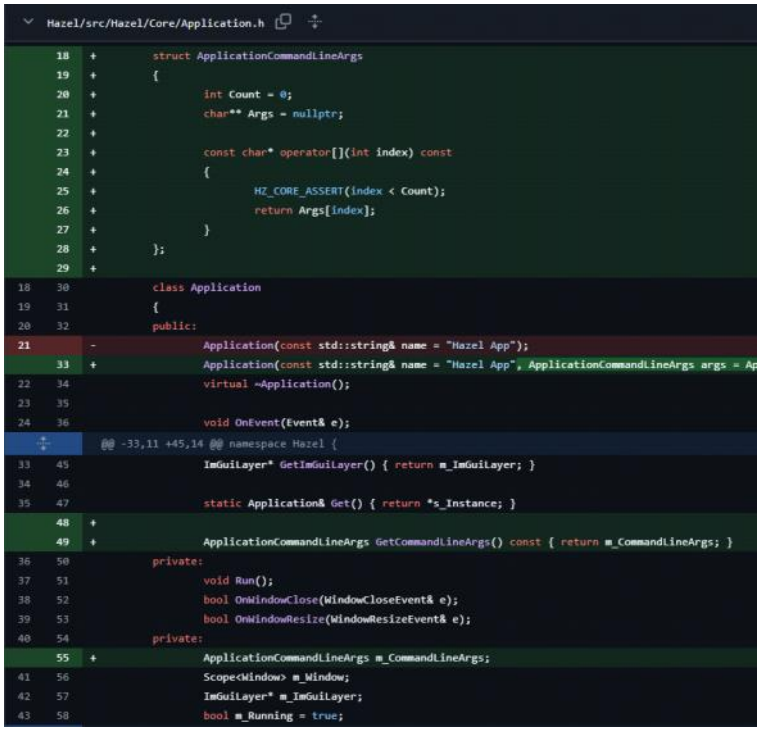
3 Application 中的 ApplicationCommandLineArgs（added command line args）

Hazel/src/Hazel/Core/Application.cpp

```
@@ -13,7 +13,8 @@ namespace Hazel {

        Application* Application::s_Instance = nullptr;

-       Application::Application(const std::string& name)
+       Application::Application(const std::string& name, ApplicationCommandLineArgs args)
+           : m_CommandLineArgs(args)
        {
            HZ_PROFILE_FUNCTION();
```

------ ------ ------ ------ ------ ------ -------

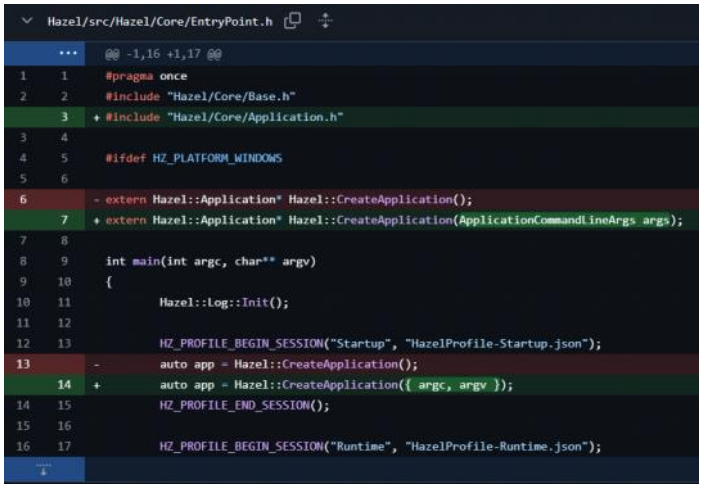
Hazel/src/Hazel/Core/Application.h



```
+       struct ApplicationCommandLineArgs
+       {
+               int Count = 0;
+               char** Args = nullptr;
+
+               const char* operator[](int index) const
+               {
+                       HZ_CORE_ASSERT(index < Count);
+                       return Args[index];
+               }
+       };
+
        class Application
        {
        public:
-               Application(const std::string& name = "Hazel App");
+               Application(const std::string& name = "Hazel App", ApplicationCommandLineArgs args = Ap
                virtual ~Application();

                void OnEvent(Event& e);
@@ -33,11 +45,14 @@ namespace Hazel {
                ImGuiLayer* GetImGuiLayer() { return m_ImGuiLayer; }

                static Application& Get() { return *s_Instance; }
+
+               ApplicationCommandLineArgs GetCommandLineArgs() const { return m_CommandLineArgs; }
        private:
                void Run();
                bool OnWindowClose(WindowCloseEvent& e);
                bool OnWindowResize(WindowResizeEvent& e);
        private:
+               ApplicationCommandLineArgs m_CommandLineArgs;
                Scope<Window> m_Window;
                ImGuiLayer* m_ImGuiLayer;
                bool m_Running = true;
```

------ ------ ------ ------ ------ ------ -------

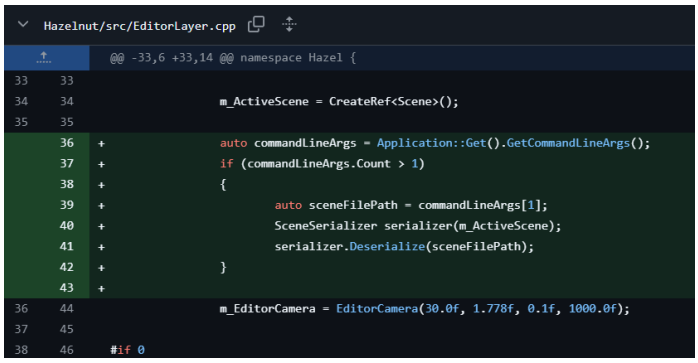
Hazel/src/Hazel/Core/EntryPoint.h



```
@@ -1,16 +1,17 @@
 #pragma once
 #include "Hazel/Core/Base.h"
+#include "Hazel/Core/Application.h"

 #ifdef HZ_PLATFORM_WINDOWS

-extern Hazel::Application* Hazel::CreateApplication();
+extern Hazel::Application* Hazel::CreateApplication(ApplicationCommandLineArgs args);

 int main(int argc, char** argv)
 {
         Hazel::Log::Init();

         HZ_PROFILE_BEGIN_SESSION("Startup", "HazelProfile-Startup.json");
-        auto app = Hazel::CreateApplication();
+        auto app = Hazel::CreateApplication({ argc, argv });
         HZ_PROFILE_END_SESSION();

         HZ_PROFILE_BEGIN_SESSION("Runtime", "HazelProfile-Runtime.json");
```

------ ------ ------ ------ ------ ------ -------

Hazelnut/src/EditorLayer.cpp



```
@@ -33,6 +33,14 @@ namespace Hazel {

                m_ActiveScene = CreateRef<Scene>();

+               auto commandLineArgs = Application::Get().GetCommandLineArgs();
+               if (commandLineArgs.Count > 1)
+               {
+                       auto sceneFilePath = commandLineArgs[1];
+                       SceneSerializer serializer(m_ActiveScene);
+                       serializer.Deserialize(sceneFilePath);
+               }
+
                m_EditorCamera = EditorCamera(30.0f, 1.778f, 0.1f, 1000.0f);

 #if 0
```

------ ------ ------ ------ ------ ------ -------

Hazelnut/src/HazelnutApp.cpp

```
@@ -8,8 +8,8 @@ namespace Hazel {
8  8          class Hazelnut : public Application
9  9          {
10 10         public:
11    -              Hazelnut()
12    -                  : Application("Hazelnut")
   11 +              Hazelnut(ApplicationCommandLineArgs args)
   12 +                  : Application("Hazelnut", args)
13 13             {
14 14                 PushLayer(new EditorLayer());
15 15             }
@@ -19,9 +19,9 @@ namespace Hazel {
19 19             }
20 20         };
21 21
22    -        Application* CreateApplication()
   22 +        Application* CreateApplication(ApplicationCommandLineArgs args)
23 23         {
24    -            return new Hazelnut();
   24 +            return new Hazelnut(args);
25 25         }
26 26
27    - }
   27 + }
```

------ ------ ------ ------ ------ ------ -------

4 Uniform Buffer 的定义以及使用，包括着色器更新（added uniform buffers）

Hazel/src/Hazel/Renderer/UniformBuffer.h

```
@@ -0,0 +1,16 @@
+ #pragma once
+
+ #include "Hazel/Core/Base.h"
+
+ namespace Hazel {
+
+         class UniformBuffer
+         {
+         public:
+                 virtual ~UniformBuffer() {}
+                 virtual void SetData(const void* data, uint32_t size, uint32_t offset = 0) = 0;
+
+                 static Ref<UniformBuffer> Create(uint32_t size, uint32_t binding);
+         };
+
+ }
```

Hazel/src/Hazel/Renderer/UniformBuffer.cpp

```
@@ -0,0 +1,21 @@
+ #include "hzpch.h"
+ #include "UniformBuffer.h"
+
+ #include "Hazel/Renderer/Renderer.h"
+ #include "Platform/OpenGL/OpenGLUniformBuffer.h"
+
+ namespace Hazel {
+
+         Ref<UniformBuffer> UniformBuffer::Create(uint32_t size, uint32_t binding)
+         {
+                 switch (Renderer::GetAPI())
+                 {
+                         case RendererAPI::API::None:    HZ_CORE_ASSERT(false, "RendererAPI::None is cu
+                         case RendererAPI::API::OpenGL:  return CreateRef<OpenGLUniformBuffer>(size, bi
+                 }
+
+                 HZ_CORE_ASSERT(false, "Unknown RendererAPI!");
+                 return nullptr;
+         }
+
+ }
```

------ ------ ------ ------ ------ ------ -------

Hazel/src/Platform/OpenGL/OpenGLUniformBuffer.h

```
@@ -0,0 +1,17 @@
+ #pragma once
+
+ #include "Hazel/Renderer/UniformBuffer.h"
+
+ namespace Hazel {
+
+         class OpenGLUniformBuffer : public UniformBuffer
+         {
+         public:
+                 OpenGLUniformBuffer(uint32_t size, uint32_t binding);
+                 virtual ~OpenGLUniformBuffer();
+
+                 virtual void SetData(const void* data, uint32_t size, uint32_t offset = 0) override;
+         private:
+                 uint32_t m_RendererID = 0;
+         };
+ }
```

Hazel/src/Platform/OpenGL/OpenGLUniformBuffer.cpp

```
@@ -0,0 +1,26 @@
+ #include "hzpch.h"
+ #include "OpenGLUniformBuffer.h"
+
+ #include <glad/glad.h>
+
+ namespace Hazel {
+
+         OpenGLUniformBuffer::OpenGLUniformBuffer(uint32_t size, uint32_t binding)
+         {
+                 glCreateBuffers(1, &m_RendererID);
+                 glNamedBufferData(m_RendererID, size, nullptr, GL_DYNAMIC_DRAW); // TODO: inv
+                 glBindBufferBase(GL_UNIFORM_BUFFER, binding, m_RendererID);
+         }
+
+         OpenGLUniformBuffer::~OpenGLUniformBuffer()
+         {
+                 glDeleteBuffers(1, &m_RendererID);
+         }
+
+
+         void OpenGLUniformBuffer::SetData(const void* data, uint32_t size, uint32_t offset)
+         {
+                 glNamedBufferSubData(m_RendererID, offset, size, data);
+         }
+
+ }
```

------ ------ ------ ------ ------ ------ -------

Hazel/src/Hazel/Renderer/Renderer2D.cpp

```
@@ -3,9 +3,11 @@

  #include "Hazel/Renderer/VertexArray.h"
  #include "Hazel/Renderer/Shader.h"
+ #include "Hazel/Renderer/UniformBuffer.h"
  #include "Hazel/Renderer/RenderCommand.h"

  #include <glm/gtc/matrix_transform.hpp>
+ #include <glm/gtc/type_ptr.hpp>

  namespace Hazel {

@@ -43,6 +45,13 @@ namespace Hazel {
                  glm::vec4 QuadVertexPositions[4];

                  Renderer2D::Statistics Stats;
+
+                 struct CameraData
+                 {
+                         glm::mat4 ViewProjection;
+                 };
+                 CameraData CameraBuffer;
+                 Ref<UniformBuffer> CameraUniformBuffer;
          };
```

------ ------ ------ ------ ------ ------ -------

Hazelnut/assets/shaders/Texture.glsl

```
@@ -1,7 +1,7 @@
  // Basic Texture Shader

  #type vertex
- #version 450
+ #version 450 core

  layout(location = 0) in vec3 a_Position;
  layout(location = 1) in vec4 a_Color;
@@ -10,6 +10,6 @@ layout(location = 3) in float a_TexIndex;
  layout(location = 4) in float a_TilingFactor;
  layout(location = 5) in int a_EntityID;

- uniform mat4 u_ViewProjection;
+ layout(std140, binding = 0) uniform Camera
+ {
+         mat4 u_ViewProjection;
+ };
+
+ struct VertexOutput
+ {
+         vec4 Color;
+         vec2 TexCoord;
+         float TexIndex;
+         float TilingFactor;
+ };
```

5 着色器系统更新：	Timer 的定义

(New shader system)

Hazel/src/Hazel/Core/Timer.h

```
@@ -0,0 +1,34 @@
+ #pragma once
+
+ #include <chrono>
+
+ namespace Hazel {
+
+ 	class Timer
+ 	{
+ 	public:
+ 		Timer()
+ 		{
+ 			Reset();
+ 		}
+
+ 		void Timer::Reset()
+ 		{
+ 			m_Start = std::chrono::high_resolution_clock::now();
+ 		}
+
+ 		float Timer::Elapsed()
+ 		{
+ 			return std::chrono::duration_cast<std::chrono::nanosec
  0.001f * 0.001f;
+ 		}
+
+ 		float Timer::ElapsedMillis()
+ 		{
+ 			return Elapsed() * 1000.0f;
+ 		}
+
+ 	private:
```

------ ------ ------ ------ ------ ------ -------

着色器更新

Hazel/src/Platform/OpenGL/OpenGLShader.cpp

```
@@ -6,26 +6,104 @@
 #include <glm/gtc/type_ptr.hpp>

+ #include <shaderc/shaderc.hpp>
+ #include <spirv_cross/spirv_cross.hpp>
+ #include <spirv_cross/spirv_glsl.hpp>
+
+ #include "Hazel/Core/Timer.h"
+
 namespace Hazel {

- 	static GLenum ShaderTypeFromString(const std::string& type)
- 	{
- 		if (type == "vertex")
- 			return GL_VERTEX_SHADER;
- 		if (type == "fragment" || type == "pixel")
- 			return GL_FRAGMENT_SHADER;
+ 	namespace Utils {
+
+ 		static GLenum ShaderTypeFromString(const std::string& type)
+ 		{
+ 			if (type == "vertex")
+ 				return GL_VERTEX_SHADER;
+ 			if (type == "fragment" || type == "pixel")
+ 				return GL_FRAGMENT_SHADER;
+
+ 			HZ_CORE_ASSERT(false, "Unknown shader type!");
+ 			return 0;
+ 		}
```

Hazel/src/Platform/OpenGL/OpenGLShader.h

```
@@ -41,10 +41,20 @@ namespace Hazel {
 	private:
 		std::string ReadFile(const std::string& filepath);
 		std::unordered_map<GLenum, std::string> PreProcess(const std::string& source);
- 		void Compile(const std::unordered_map<GLenum, std::string>& shaderSources);
+
+ 		void CompileOrGetVulkanBinaries(const std::unordered_map<GLenum, std::string>&
+ 		void CompileOrGetOpenGLBinaries();
+ 		void CreateProgram();
+ 		void Reflect(GLenum stage, const std::vector<uint32_t>& shaderData);
 	private:
 		uint32_t m_RendererID;
+ 		std::string m_FilePath;
 		std::string m_Name;
+
+ 		std::unordered_map<GLenum, std::vector<uint32_t>> m_VulkanSPIRV;
+ 		std::unordered_map<GLenum, std::vector<uint32_t>> m_OpenGLSPIRV;
+
+ 		std::unordered_map<GLenum, std::string> m_OpenGLSourceCode;
 	};

 }
```

6 平台工具的更新（打开或保存文件）

Hazel/src/Hazel/Utils/PlatformUtils.h

```
@@ -9,8 +9,8 @@ namespace Hazel {
 	{
 	public:
 		// These return empty strings if cancelled
- 		static std::optional<std::string> OpenFile(const char* filter);
- 		static std::optional<std::string> SaveFile(const char* filter);
+ 		static std::string OpenFile(const char* filter);
+ 		static std::string SaveFile(const char* filter);
 	};

 }
```

Hazel/src/Platform/Windows/WindowsPlatformUtils.cpp

```
@@ -10,7 +10,7 @@
10  10
11  11  namespace Hazel {
12  12
13      -       std::optional<std::string> FileDialogs::OpenFile(const char* filter)
    13  +       std::string FileDialogs::OpenFile(const char* filter)
14  14          {
15  15                  OPENFILENAMEA ofn;
16  16                  CHAR szFile[260] = { 0 };
@@ -28,10 +28,12 @@ namespace Hazel {
28  28
29  29                  if (GetOpenFileNameA(&ofn) == TRUE)
30  30                          return ofn.lpstrFile;
31      -                   return std::nullopt;
    31  +
    32  +                   return std::string();
    33  +
32  34          }
33  35
34      -       std::optional<std::string> FileDialogs::SaveFile(const char* filter)
    36  +       std::string FileDialogs::SaveFile(const char* filter)
35  37          {
36  38                  OPENFILENAMEA ofn;
37  39                  CHAR szFile[260] = { 0 };
```

Hazelnut/src/EditorLayer.cpp

```
405 413         void EditorLayer::OpenScene()
406 414         {
407     -           std::optional<std::string> filepath = FileDialogs::OpenFile("Hazel Scene (*.hazel)\0*.hazel\0");
408     -           if (filepath)
    415 +           std::string filepath = FileDialogs::OpenFile("Hazel Scene (*.hazel)\0*.hazel\0");
    416 +           if (!filepath.empty())
409 417             {
410 418                     m_ActiveScene = CreateRef<Scene>();
411 419                     m_ActiveScene->OnViewportResize((uint32_t)m_ViewportSize.x, (uint32_t)m_ViewportSize.y);
412 420                     m_SceneHierarchyPanel.SetContext(m_ActiveScene);
413 421
414 422                     SceneSerializer serializer(m_ActiveScene);
415     -                   serializer.Deserialize(*filepath);
    423 +                   serializer.Deserialize(filepath);
416 424             }
417 425         }
418 426
419 427         void EditorLayer::SaveSceneAs()
420 428         {
421     -           std::optional<std::string> filepath = FileDialogs::SaveFile("Hazel Scene (*.hazel)\0*.hazel\0");
422     -           if (filepath)
    429 +           std::string filepath = FileDialogs::SaveFile("Hazel Scene (*.hazel)\0*.hazel\0");
    430 +           if (!filepath.empty())
423 431             {
424 432                     SceneSerializer serializer(m_ActiveScene);
425     -                   serializer.Serialize(*filepath);
    433 +                   serializer.Serialize(filepath);
```

7 视口与摄像机更新：

Hazel/src/Hazel/Scene/Scene.cpp

```
@@ -142,7 +142,8 @@ namespace Hazel {
142 142         template<>
143 143         void Scene::OnComponentAdded<CameraComponent>(Entity entity, CameraComponent& component)
144 144         {
145     -           component.Camera.SetViewportSize(m_ViewportWidth, m_ViewportHeight);
    145 +           if (m_ViewportWidth > 0 && m_ViewportHeight > 0)
    146 +                   component.Camera.SetViewportSize(m_ViewportWidth, m_ViewportHeight);
146 147         }
147 148
148 149         template<>
```

Hazel/src/Hazel/Scene/SceneCamera.cpp

```
@@ -30,6 +30,7 @@ namespace Hazel {
30  30
31  31          void SceneCamera::SetViewportSize(uint32_t width, uint32_t height)
32  32          {
    33  +           HZ_CORE_ASSERT(width > 0 && height > 0);
33  34              m_AspectRatio = (float)width / (float)height;
34  35              RecalculateProjection();
35  36          }
```

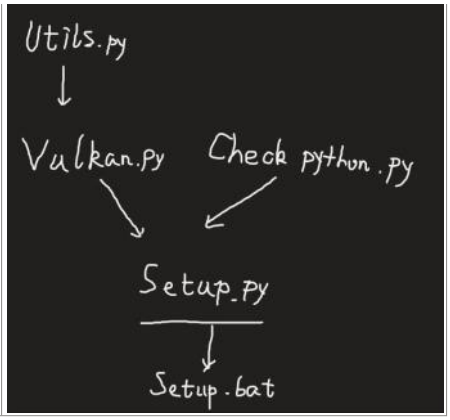
》》》》我将逐次的提交这些代码，并记录自己的疑虑

》》》我首先使用更新并使用 py 文件下载 Vulkan SDK

首先第一步：运行 bat 脚本，通过该文件下载 Vulkan SDK。

（Vulkan.py 文件使用了 Utils.py 中的函数，当你在 Hazel\scripts 的路径下通过 Setup.py 使用 Vulkan.py 时，Vulkan.py 会将 Vulkan 默认下载到 Nut/vendor/VulkanSDK。）

》》以下是这些 py 文件的结构：



》》问题零

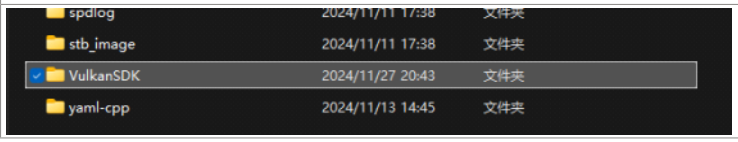
运行脚本时，请关闭代理。

》》问题一

如果将文件放在 Scripts 文件夹下，并直接通过 Setup.bat 运行 Setup.py 的话，会出现报错，表示文件路径已经不存在。--->

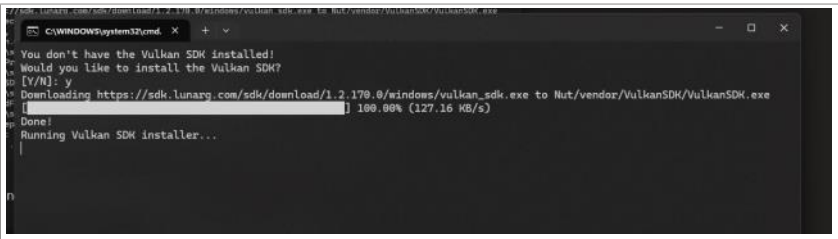
```
[Y/N]: y
Downloading https://sdk.lunarg.com/sdk/download/1.2.170.0/windows/vulkan_sdk.exe to Nut/vendor/VulkanSDK/VulkanSDK.exe
Traceback (most recent call last):
  File "Setup.py", line 13, in <module>
    if (not Vulkan.CheckVulkanSDK()):
  File "E:\VS\Nut\scripts\Vulkan.py", line 36, in CheckVulkanSDK
    InstallVulkanPrompt()
  File "E:\VS\Nut\scripts\Vulkan.py", line 29, in InstallVulkanPrompt
    InstallVulkanSDK()
  File "E:\VS\Nut\scripts\Vulkan.py", line 18, in InstallVulkanSDK
    Utils.DownloadFile(VULKAN_SDK_INSTALLER_URL, VULKAN_SDK_EXE_PATH)
  File "E:\VS\Nut\scripts\Utils.py", line 8, in DownloadFile
    with open(filepath, 'wb') as f:
FileNotFoundError: [Errno 2] No such file or directory: 'Nut/vendor/VulkanSDK/VulkanSDK.exe'
请按任意键继续. . .
```

这需要提前在 vendor 创建 VulkanSDK 文件夹。（记得修改 .py 中的下载路径，这取决于你的项目名称，还有你想下载到本机的路径）

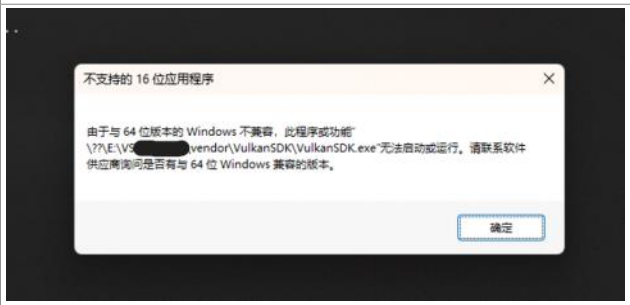


》》问题二

创建好 VulkanSDK 文件夹之后，重新运行 Setup.py，脚本运行之后开始尝试运行 Vulkan installer:



但是随后的弹窗中提示：



这可能是 Vulkan.py 中存放的 VulkanSDK 下载地址不适合 64 位系统，我将其更新为 2023 年的某一版本。

```
VULKAN_SDK = os.environ.get('VULKAN_SDK')
VULKAN_SDK_INSTALLER_URL = 'https://sdk.lunarg.com/sdk/download/1.2.170.0/windows/vulkan_sdk.exe'
NUT_VULKAN_VERSION = '1.2.170.0'
VULKAN_SDK_EXE_PATH = 'Nut/vendor/VulkanSDK/VulkanSDK.exe'


def InstallVulkanSDK():
    print('Downloading {} to {}'.format(VULKAN_SDK_INSTALLER_URL, VULKAN_SDK_EXE_PATH))
    Utils.DownloadFile(VULKAN_SDK_INSTALLER_URL, VULKAN_SDK_EXE_PATH)
    print("Done!")
    print("Running Vulkan SDK installer...")
    os.startfile(os.path.abspath(VULKAN_SDK_EXE_PATH))
    print("Re-run this script after installation")
```

--- ---

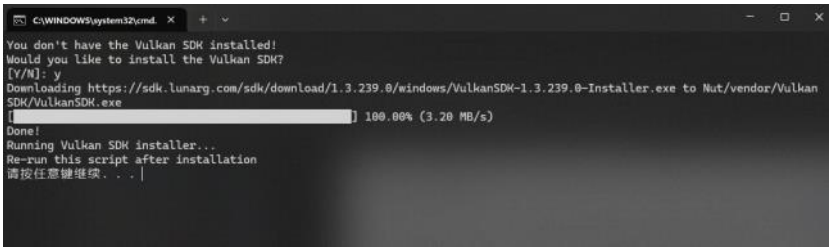
--- ---

附录：如果你想进入官网查看适合你系统的SDK，以下是网址 ->
（ https://vulkan.lunarg.com/sdk/home ）

当前我只更新了 SDK Installer 的安装地址，但是我还没有更新随后的 debug lib.zip，这是下一个问题会出现的地方，现在先不讨论。

```python
VulkanSDKDebugLibsURL = 'https://files.lunarg.com/SDK-1.2.170.0/VulkanSDK-1.2.170.0-DebugLibs.zip'
OutputDirectory = "Nut/vendor/VulkanSDK"
TempZipFile = f"{OutputDirectory}/VulkanSDK.zip"


def CheckVulkanSDKDebugLibs():
    shadercdLib = Path(f"{OutputDirectory}/Lib/shaderc_sharedd.lib")
    if (not shadercdLib.exists()):
        print(f"No Vulkan SDK debug libs found. (Checked {shadercdLib})")
        print("Downloading", VulkanSDKDebugLibsURL)
        with urlopen(VulkanSDKDebugLibsURL) as zipresp:
            with ZipFile(BytesIO(zipresp.read())) as zfile:
                zfile.extractall(OutputDirectory)
    print(f"Vulkan SDK debug libs located at {OutputDirectory}")
    return True
```

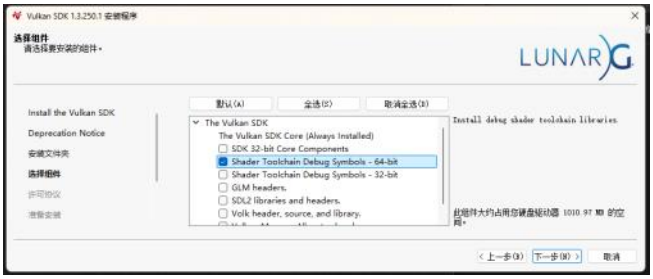
我们先重新运行一遍，使用更新之后的 SDK install。



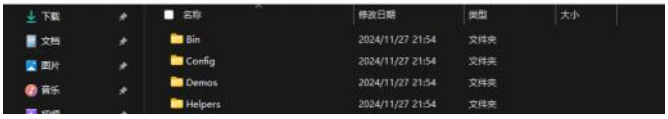
于是运行后出现这样的窗口：

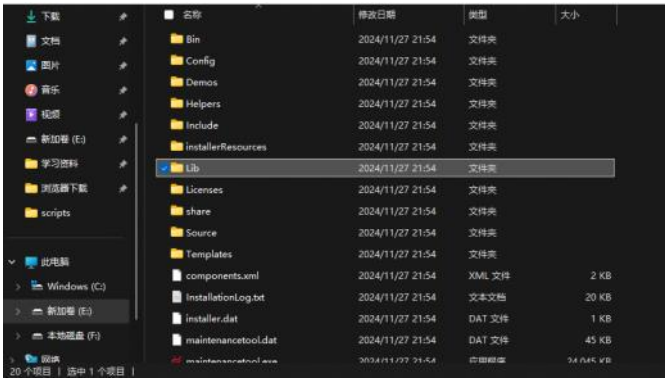
安装 vulkan SDK

我目前没有选择任何拓展，但在安装过程中，我不是很确定这个拓展和 DebugLibs 有没有什么直接关系。就先标注一下。
（毕竟这将会占用我1G空间 bushi）



随后便得到这样的文件构架：





》》问题三

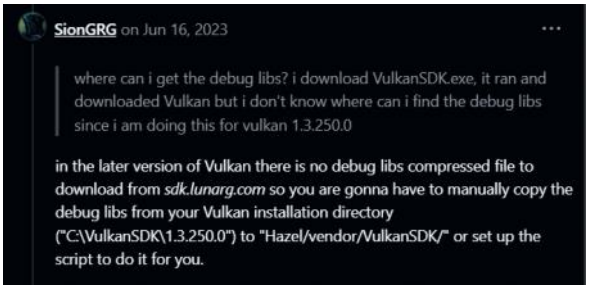
我们发现 Cherno 另外下载了一个 Debuglib.zip，并对其进行了一些处理。

但是在1.2.198.1版本之后，lunarg 公司不再支持 debuglibs 的单独下载。现在 SDK 中的调试库通常随着 Vulkan 库一起分发，不再单独打包成一个 zip 文件。

所以现在，这些文件通常直接包含在 Vulkan SDK 的核心目录下，特别是在 lib 目录中

```
VulkanSDKDebugLibsURL = 'https://files.lunarg.com/SDK-1.2.170.0/VulkanSDK-1.2.170.0-DebugLibs.zip'
OutputDirectory = "Nut/vendor/VulkanSDK"
TempZipFile = f"{OutputDirectory}/VulkanSDK.zip"


def CheckVulkanSDKDebugLibs():
    shaderdcLib = Path(f"{OutputDirectory}/Lib/shaderc_sharedd.lib")
    if (not shaderdcLib.exists()):
        print(f"No Vulkan SDK debug libs found. (Checked {shaderdcLib})")
        print("Downloading", VulkanSDKDebugLibsURL)
        with urlopen(VulkanSDKDebugLibsURL) as zipresp:
            with ZipFile(BytesIO(zipresp.read())) as zfile:
                zfile.extractall(OutputDirectory)
    print(f"Vulkan SDK debug libs located at {OutputDirectory}")
    return True
```

我们也可以从评论中窥见这一更改。（@SionGRG）



现在我们需要更改这个函数（ CheckVulkanSDKDebugLibs ）的逻辑

```
VulkanSDKDebugLibsURL = 'https://files.lunarg.com/SDK-1.2.170.0/VulkanSDK-1.2.170
OutputDirectory = "Nut/vendor/VulkanSDK"
TempZipFile = f"{OutputDirectory}/VulkanSDK.zip"


def CheckVulkanSDKDebugLibs():
    shaderdcLib = Path(f"{OutputDirectory}/Lib/shaderc_sharedd.lib")
    if (not shaderdcLib.exists()):
        print(f"No Vulkan SDK debug libs found. (Checked {shaderdcLib})")
        print("Downloading", VulkanSDKDebugLibsURL)
        with urlopen(VulkanSDKDebugLibsURL) as zipresp:
            with ZipFile(BytesIO(zipresp.read())) as zfile:
                zfile.extractall(OutputDirectory)
    print(f"Vulkan SDK debug libs located at {OutputDirectory}")
    return True
```

--- ---

首先，我对这个 shaderc_sharedd.lib 的路径有点疑惑：因为我的确查找到了 shaderc_shared.lib 这个库，而不是shaderc_sharedd.lib。

```
53  def CheckVulkanSDKDebugLibs():
54      shaderdcLib = Path(f"{OutputDirectory}/Lib/shaderc_sharedd.lib")
55      if (not shaderdcLib.exists()):
```

--- ---

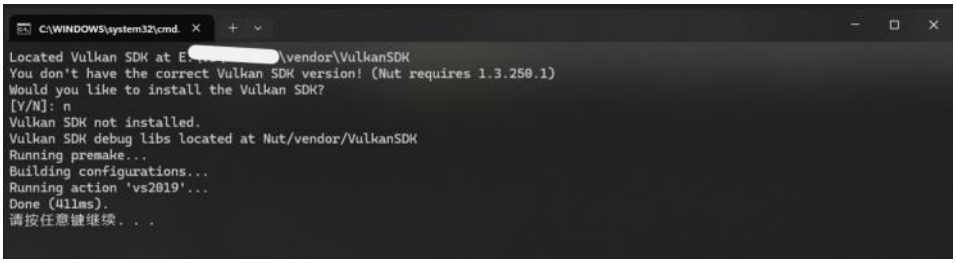
现在我开始更改，不过我发现原先的逻辑是：如果没有找到调试库，就在线去下载。
但现在这些文件将会在安装 Vulkan SDK 时，同步安装在文件夹中，所以如果没有找到的话，一定是安装是出了什么问题。

我便做了以下更改：（仅仅是口头提醒一下 :-））

```
# VulkanSDKDebugLibsURL = 'https://files.lunarg.com/SDK-1.2.170.0/VulkanSDK-1.2.170.0-DebugLibs.zip'
OutputDirectory = "Nut/vendor/VulkanSDK"
# TempZipFile = f"{OutputDirectory}/VulkanSDK.zip"


def CheckVulkanSDKDebugLibs():
    shaderdcLib = Path(f"{OutputDirectory}/Lib/shaderc_shared.lib")
    if (not shaderdcLib.exists()):
        print(f"No Vulkan SDK debug libs found. (Checked {shaderdcLib})")
        # print("Downloading", VulkanSDKDebugLibsURL)
```

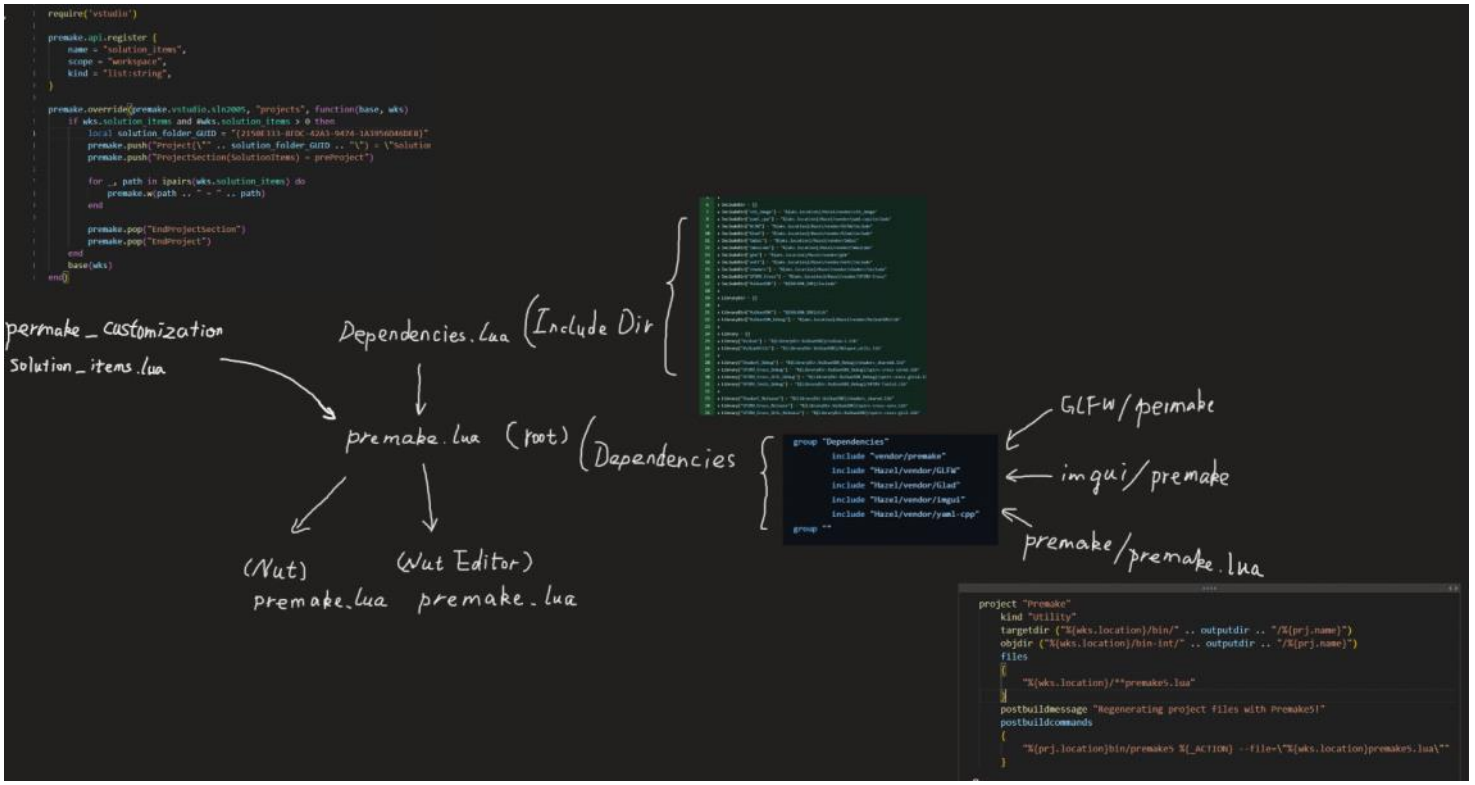
```
    # with urlopen(VulkanSDKDebugLibsURL) as zipresp:
    #     with ZipFile(BytesIO(zipresp.read())) as zfile:
    #         zfile.extractall(OutputDirectory)
    print(f"Please check you Vulkan SDK files. (Checked https://vulkan.lunarg.com/sdk/home and find you version, Or reinstall you Vulkan SDK by installer)")

    # 在找不到库文件时抛出异常并退出
    raise RuntimeError("Vulkan SDK debug libs not found, process aborted.")

print(f"Vulkan SDK debug libs located at {OutputDirectory}")
return True
```

随后我重新运行 Setup.bat，并拒绝再次安装 installer，便得到这样的结果：



我想应该是对了。

》》》》现在我们已经成功安装了 Vulkan，现在则需要更新 premake 文件内容。
这是将要实现的 premake 文件构架图（以及细则）

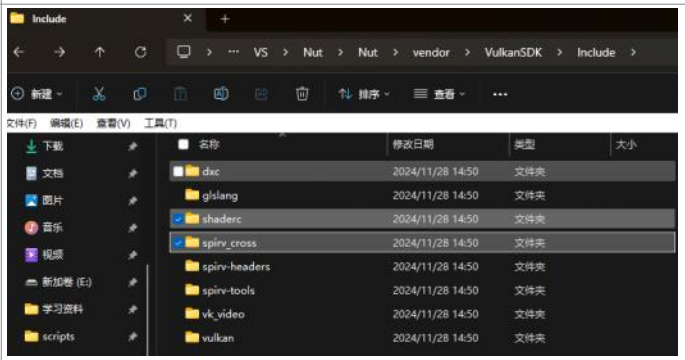


》》》》接下来我先更新 Premake Dependencies.lua 文件（这里为预处理，实际操作步骤在后面）。

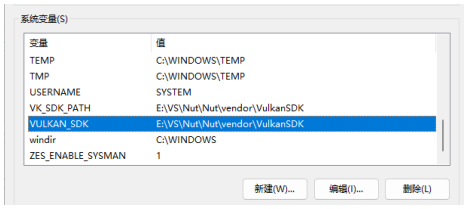
第一步，我们在项目的根目录下重新编写一个 premake 文件，这个文件主要用来索引 vendor 中的外部库（API）

```lua
-- Nut Dependencies

VULKAN_SDK = os.getenv("VULKAN_SDK")

IncludeDir = {}
IncludeDir["stb_image"] = "%{wks.location}/Nut/vendor/stb_image"
IncludeDir["yaml_cpp"] = "%{wks.location}/Nut/vendor/yaml-cpp/include"
IncludeDir["GLFW"] = "%{wks.location}/Nut/vendor/GLFW/include"
IncludeDir["Glad"] = "%{wks.location}/Nut/vendor/Glad/include"
IncludeDir["ImGui"] = "%{wks.location}/Nut/vendor/ImGui"
IncludeDir["ImGuizmo"] = "%{wks.location}/Nut/vendor/ImGuizmo"
IncludeDir["glm"] = "%{wks.location}/Nut/vendor/glm"
IncludeDir["entt"] = "%{wks.location}/Nut/vendor/entt/include"
IncludeDir["shaderc"] = "%{wks.location}/Nut/vendor/shaderc/include"
IncludeDir["SPIRV_Cross"] = "%{wks.location}/Nut/vendor/SPIRV-Cross"
IncludeDir["VulkanSDK"] = "%{VULKAN_SDK}/Include"

LibraryDir = {}

LibraryDir["VulkanSDK"] = "%{VULKAN_SDK}/Lib"
LibraryDir["VulkanSDK_Debug"] = "%{wks.location}/Nut/vendor/VulkanSDK/Lib"

Library = {}

Library["Vulkan"] = "%{LibraryDir.VulkanSDK}/vulkan-1.lib"
Library["VulkanUtils"] = "%{LibraryDir.VulkanSDK}/VkLayer_utils.lib"

Library["ShaderC_Debug"] = "%{LibraryDir.VulkanSDK_Debug}/shaderc_sharedd.lib"
Library["SPIRV_Cross_Debug"] = "%{LibraryDir.VulkanSDK_Debug}/spirv-cross-cored.lib"
Library["SPIRV_Cross_GLSL_Debug"] = "%{LibraryDir.VulkanSDK_Debug}/spirv-cross-glsld.lib"
Library["SPIRV_Tools_Debug"] = "%{LibraryDir.VulkanSDK_Debug}/SPIRV-Toolsd.lib"

Library["ShaderC_Release"] = "%{LibraryDir.VulkanSDK}/shaderc_shared.lib"
Library["SPIRV_Cross_Release"] = "%{LibraryDir.VulkanSDK}/spirv-cross-core.lib"
Library["SPIRV_Cross_GLSL_Release"] = "%{LibraryDir.VulkanSDK}/spirv-cross-glsl.lib"
```

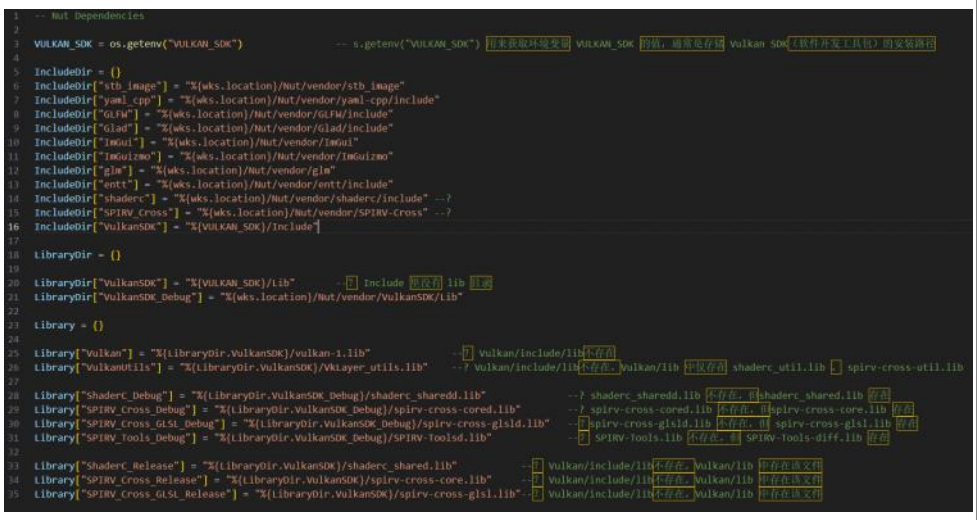
但我发现有些问题，比如 shaderc 和 spriv_cross 的路径已经发生改变，参考 1.3.250.1 版本：这两个文件夹位于 VulkanSDK/Include 下



系统变量示例：

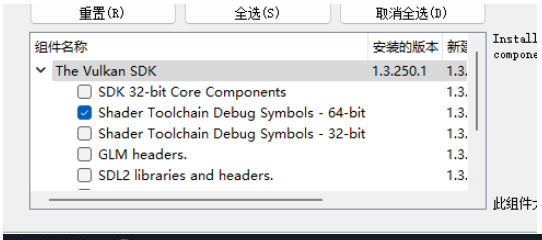


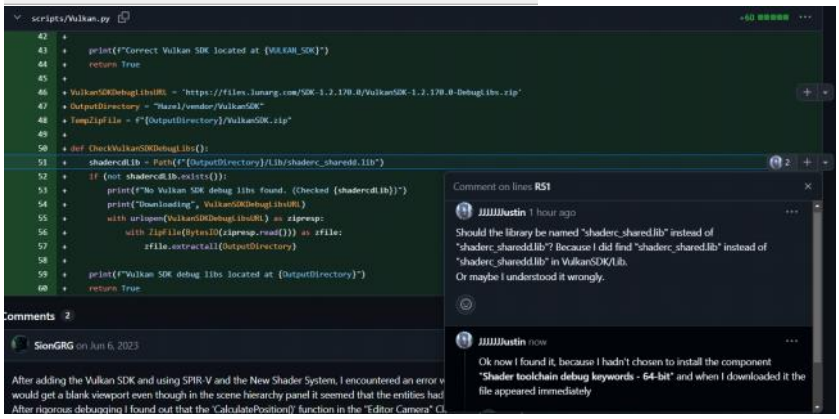
而且由于我没有下载某些组件，这使很多文件并不存在。（我将其标注出来）



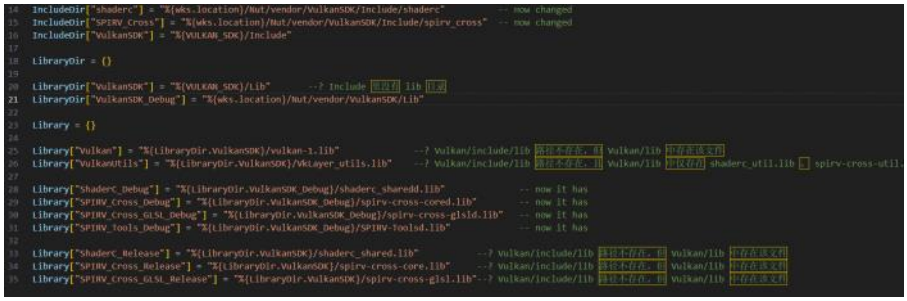
于是我决定下载拓展(shader toolchain debug symbols)，这一步

通过运行 maintenancetool.exe 文件实现：





这个组件将会解决这部分问题：



虽然下载了一个组件可以解决但部分问题，但是尽管在之后我下载了其余的所有组件， VulkanSDK/Include/Lib 这个路径都不存在（但是Vulkan/Lib 这个路径存在），且 VKLayer_utils.lib 这个文件也不存在。

一个问题：VKLayer_utils.lib 似乎在1.3.216.0 版本中被移除了。

所以这是 premake 文件最新的样子：

》》》》操作步骤：

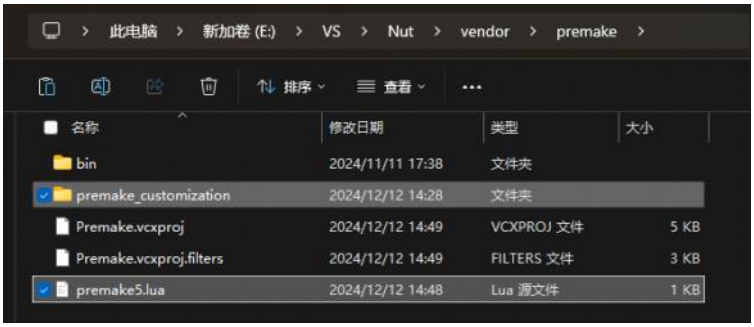
》》111 现在我们将 Nut/premake.lua 中的表单独存放在另一个文件中(Dependencies.lua)

其中包括：

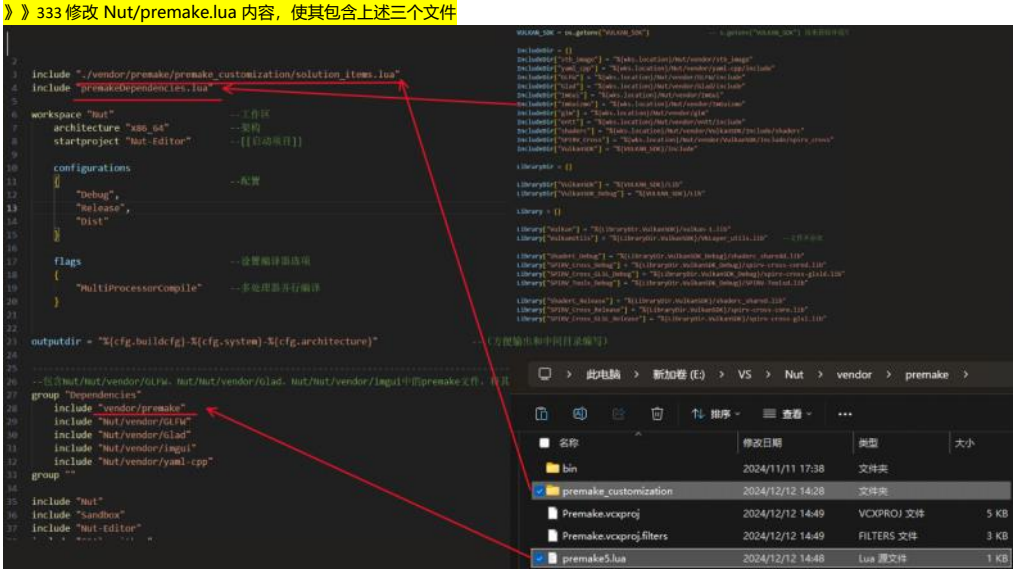
```
-------------------------- Nut Dependencies ---------------------------

VULKAN_SDK = os.getenv("VULKAN_SDK")                -- s.getenv("VULKAN_SDK") 用来获取环境

IncludeDir = {}
IncludeDir["stb_image"] = "%{wks.location}/Nut/vendor/stb_image"
IncludeDir["yaml_cpp"] = "%{wks.location}/Nut/vendor/yaml-cpp/include"
IncludeDir["GLFW"] = "%{wks.location}/Nut/vendor/GLFW/include"
IncludeDir["Glad"] = "%{wks.location}/Nut/vendor/Glad/include"
IncludeDir["ImGui"] = "%{wks.location}/Nut/vendor/ImGui"
IncludeDir["ImGuizmo"] = "%{wks.location}/Nut/vendor/ImGuizmo"
IncludeDir["glm"] = "%{wks.location}/Nut/vendor/glm"
IncludeDir["entt"] = "%{wks.location}/Nut/vendor/entt/include"
IncludeDir["shaderc"] = "%{wks.location}/Nut/vendor/VulkanSDK/Include/shaderc"
IncludeDir["SPIRV_Cross"] = "%{wks.location}/Nut/vendor/VulkanSDK/Include/spirv_cross"
IncludeDir["VulkanSDK"] = "%{VULKAN_SDK}/Include"

LibraryDir = {}

LibraryDir["VulkanSDK"] = "%{VULKAN_SDK}/Lib"
LibraryDir["VulkanSDK_Debug"] = "%{VULKAN_SDK}/Lib"

Library = {}

Library["Vulkan"] = "%{LibraryDir.VulkanSDK}/vulkan-1.lib"
Library["VulkanUtils"] = "%{LibraryDir.VulkanSDK}/VkLayer_utils.lib"     --文件不存在

Library["ShaderC_Debug"] = "%{LibraryDir.VulkanSDK_Debug}/shaderc_sharedd.lib"
Library["SPIRV_Cross_Debug"] = "%{LibraryDir.VulkanSDK_Debug}/spirv-cross-cored.lib"
Library["SPIRV_Cross_GLSL_Debug"] = "%{LibraryDir.VulkanSDK_Debug}/spirv-cross-glsld.lib"
Library["SPIRV_Tools_Debug"] = "%{LibraryDir.VulkanSDK_Debug}/SPIRV-Toolsd.lib"

Library["ShaderC_Release"] = "%{LibraryDir.VulkanSDK}/shaderc_shared.lib"
Library["SPIRV_Cross_Release"] = "%{LibraryDir.VulkanSDK}/spirv-cross-core.lib"
Library["SPIRV_Cross_GLSL_Release"] = "%{LibraryDir.VulkanSDK}/spirv-cross-glsl.lib"
```

》》222 在 Nut/vendor/premake 下（注意不是 Nut/Nut/vendor/ 这个路径）创建如下文件。（内容等会说明）

（链接： 》》》》接下来谈谈 vendor/premake 文件中我们新添的两个文件：premake5.lua 和 premake_customization/solution_items ）



》》333 修改 Nut/premake.lua 内容，使其包含上述三个文件



```
include "./vendor/premake/premake_customization/solution_items.lua"
include "premakeDependencies.lua"

workspace "Nut"                     --工作区
    architecture "x86_64"           --架构
    startproject "Nut-Editor"       --{{启动项目}}

    configurations
    {                               --配置
        "Debug",
        "Release",
        "Dist"
    }

    flags                           --设置编译器选项
    {
        "MultiProcessorCompile"     --多处理器并行编译
    }


outputdir = "%{cfg.buildcfg}-%{cfg.system}-%{cfg.architecture}"          --(方便输出和中间目录编写)

--包含Nut/Nut/vendor/GLFW、Nut/Nut/vendor/Glad、Nut/Nut/vendor/imgui中的premake文件，和其
group "Dependencies"
    include "vendor/premake"
    include "Nut/vendor/GLFW"
    include "Nut/vendor/Glad"
    include "Nut/vendor/imgui"
    include "Nut/vendor/yaml-cpp"
group ""

include "Nut"
include "Sandbox"
include "Nut-Editor"
```

》》444 修改 Nut/Nut/premake5.lua 和 Nut/Nut-Editor/premake5.lua 文件内容

具体内容是：	Nut-premake 文件需要包含 Vulkan 的库目录，并在对应配置下添加相关链接。

》》问题：

在此处我遇到一个问题，就是 Cherno 对这两个文件关闭了 staticruntime 设置。

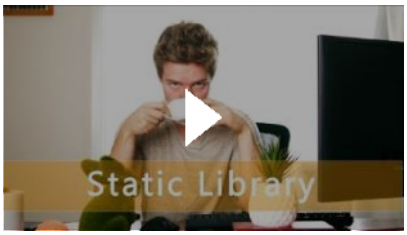
这表示禁用静态链接运行时库，使用动态链接的运行时库。意味着程序在运行时将依赖外部的动态链接库（DLL），而不是将运行时库直接嵌入到可执行文件中。

示例：

Hazelnut/premake5.lua

```
@@ -2,7 +2,7 @@ project "Hazelnut"
        kind "ConsoleApp"
        language "C++"
        cppdialect "C++17"
-       staticruntime "on"
+       staticruntime "off"

        targetdir ("%{wks.location}/bin/" .. outputdir .. "/%{prj.name}")
        objdir ("%{wks.location}/bin-int/" .. outputdir .. "/%{prj.name}")
```

而我印象里 Cherno 没有说明要转回使用动态库的方式，所以我没有将其打开。
（顺便一提，如果需要打开的话，还需要额外进行动态链接的配置操作，具体可以回看Cherno的视频： Static Libraries and ZERO Warnings | Game Engine series ）



》》》》接下来谈谈 vendor/premake 文件中我们新添的两个文件：premake5.lua 和 premake_customization/solution_items.lua
具体的 Pull&requests 记载于 #301（ https://github.com/TheCherno/Hazel/pull/301 ）

Premake5.lua	定义一个工具类型的项目 Premake，并且在构建后通过 premake5 工具来重新生成或更新项目文件。 这个脚本的目的是 **生成或重新生成构建项目文件**（如 `Visual Studio` 工程文件、`Makefile` 等），使用的是 `premake5` 工具。它是一个自动化构建的过程，通常用于生成构建系统（如 `Makefile` 或 `Visual Studio` 工程文件）等。
solution_items.lua	这段代码的作用是为 **Visual Studio 解决方案** 文件（.sln）添加一个新的部分，称为 **Solution Items**，并将工作区中指定的文件（通过 solution_items 命令）添加到这个部分中。 解决方案项是指那些不是属于任何特定项目的文件，例如文档、配置文件等，通常用于存储一些和整个解决方案相关但不属于某个单独项目的文件。 这添加了对 Visual Studio **解决方案项**（solution items）的支持。文档、配置文件、README 或其他相关文件将可以被作为解决方案项添加到解决方案中。

》》》》Application 中的 ApplicationCommandLineArgs （ added command line args）命令行参数

》》流程与定义的概述
首先，我们位于入口点的主函数中使用了（argc, argv) 来获取命令行信息。并且将参数传入到 CreateApplication() 中，以便后续使用这些信息：

```
#ifdef NUT_PLATFORM_WINDOWS

extern Nut::Application* Nut::CreateApplication(ApplicationCommandLineArgs

int main(int argc, char** argv)                    //将其设置为windows平台上的win
{
    Nut::Log::Init();

    NUT_CORE_WARN("Initialized Log!");
    NUT_INFO("Goodbye World!");

    NUT_CORE_WARN("Command line args:");
    for (int i = 0; i < argc; i++) {
        NUT_CORE_TRACE("Argument {0}: {1}", i, argv[i])
    }

    NUT_PROFILE_BEGIN_SESSION("Startup", "NutProfile-Startup.json");
    auto app = Nut::CreateApplication({ argc, argv });      // ? 这里的 ar
    NUT_PROFILE_END_SESSION();

    NUT_PROFILE_BEGIN_SESSION("Runtime", "NutProfile-Runtime.json");
    app->Run();
    NUT_PROFILE_END_SESSION();

    NUT_PROFILE_BEGIN_SESSION("Shutdown", "NutProfile-Shutdown.json");
    delete app;
    NUT_PROFILE_END_SESSION();
}


#endif
```

在入口点使用的 `Nut::CreateApplication({ argc, argv })`，实际上是在构造一个 `ApplicationCommandLineArgs` 类型的对象，并将 `argc` 和 `argv` 传递给它。

管线流程：

```
Application* CreateApplication(ApplicationCommandLineArgs args)
{
    return new NutEditor(args);
}
```

这里是 CreateApplication() 的定义。

CreateApplication() 中使用了NutEditor()

```
class NutEditor : public Application
{
public:
    NutEditor(ApplicationCommandLineArgs args)
        :Application("Nut Editor", args)
    {
        PushLayer(new EditorLayer());
    }

    ~NutEditor()
    {
    }
};
```

这里是 NutEditor() 的定义。

NutEditor() 是 Application() 的子类，故 NutEditor() 的构造函数会自动先使用父类 Application() 的构造函数，我们可以通过这个特性将 args 参数传给 Application() 的构造函数，并实现一些目的。

```
class Application
{
public:
    Application(const std::string& name = "Nut App", ApplicationCommandLineArgs args = ApplicationCommandLineArgs());
    virtual ~Application();

    void OnEvent(Event& e);                    //事件分发

    void PushLayer(Layer* layer);
    void PushOverlay(Layer* overlay);

    inline Window& GetWindow() { return *m_Window; }          //返回下面这个指向Window的指针
    inline ImGuiLayer* GetImGuiLayer() { return m_ImGuiLayer; }
    inline static Application& Get() { return *s_Instance; }    // ！！！返回的是 s_Instance 这个指向 Application 的指
                                                                // （为什么函数是引用传递？，因为application是一个单例
    inline ApplicationCommandLineArgs GetCommandLineArgs() const { return m_CommandLineArgs;
```

这是父类 Application() 构造函数的新定义。

同时我们新添了一个 GetCommandLineArgs() 的函数，用于获取私有变量 m_CommandLineArgs 中存放的数据。

```
auto sceneFilePath0 = commandLineArgs[0];

NUT_CORE_WARN(sceneFilePath0);
```

我们可以在运行时查看argv获取到的信息是什么。

```
[15:23:15] NUT: Version: 4.6.0 - Build 31.0.101.4502
[15:23:15] NUT: E:\VS\Nut\bin\Debug-windows-x86_64\Nut-Editor\Nut-Editor.exe
[15:23:15] NUT: Pixel data: -1
```

》》》》知识点

》》》》关于 Argc, Argv

1. argc 和 argv 的含义

定义：

在 C 和 C++ 程序中，argc 和 argv 是由编译器（如 GCC、Clang 或 Visual Studio）在程序启动时自动传递给程序的 main 函数的两个参数。用于传递命令行的输入参数。

- argc：是 argument count 的缩写，表示命令行参数的数量。它是一个整数，包含程序名和任何附加的命令行参数。
- argv：是 argument vector 的缩写，表示命令行参数的数组。它是一个字符指针数组，每个元素是一个指向命令行参数的字符串。

例如，当你运行一个程序 ./myapp input.txt --verbose 时，argc 和 argv 的内容如下：

argc = 3，因为有三个参数（程序名、input.txt 和 --verbose）	argv[0] = "./myapp"，表示程序的路径。 argv[1] = "input.txt"，表示第一个参数（输入文件）。 argv[2] = "--verbose"，表示第二个参数（开启调试模式）。

运行机制：

1. 内容传递。 （什么时候传递？传递什么内容？）	argc 和 argv 是由操作系统在启动程序时根据命令行输入自动传递的，不需要手动获取。 程序中 argc 和 argv 的值取决于你启动程序时后台输入到命令行中的命令或参数内容。在不同的操作系统上，命令行参数的格式和解释规则可能会有所不同。 比如： • 在 **Windows** 上，命令行参数是由 **命令提示符**（cmd.exe）或 **PowerShell** 等工具传递给程序的。 • 在 **Unix/Linux** 上，命令行参数是由 shell（如 Bash）传递给程序的。
2. 内容 （内容什么时候被确定？是否可以被随时改变？）	argc 和 argv 是**实时的**，但它们是**程序启动时**由操作系统从命令行提取的参数，并且在程序执行过程中保持不变。 所以一旦程序开始执行，argc 和 argv 的值就固定了，不能在程序运行过程中改变。

2.有没有类似 argc 和 argv 的参数?

C++ 标准库没有其他内建的类似 argc 和 argv 的机制。argc 和 argv 是 **main** 函数的参数，是 C++ 标准定义的，通常用于处理命令行参数。

不过，你可以使用其他自定义的数据结构来封装命令行参数，为它们提供更灵活的操作方式，

例如，在当前情况下，我们可以在 EditorLayer.cpp 中实时的获取到命令行参数信息并将其打印在控制台上：

```
void EditorLayer::OnAttach()
{
    NUT_PROFILE_FUNCTION();

    m_Framebuffer = FrameBuffer::Create({ 1280, 720, 1, {FrameBufferAt

    m_Texture = Texture2D::Create("assets/textures/Checkerboard.png");
    m_Emoji = Texture2D::Create("assets/textures/emoji.png");

    m_ActiveScene = CreateRef<Scene>();

    auto commandLineArgs = Application::Get().GetCommandLineArgs();
    if (commandLineArgs.Count > 1)
    {
        auto sceneFilePath = commandLineArgs[1];
        SceneSerializer serializer(m_ActiveScene);
        serializer.Deserialize(sceneFilePath);

    }
    auto sceneFilePath0 = commandLineArgs[0];

    NUT_CORE_WARN(sceneFilePath0);
```

或者在 EntryPoint.h 中尝试打印所有捕获的命令行参数：

```
void EditorLayer::OnAttach()
{
    NUT_PROFILE_FUNCTION();

    m_Framebuffer = FrameBuffer::Create({ 1280, 720, 1, {FrameBufferAt

    m_Texture = Texture2D::Create("assets/textures/Checkerboard.png");
    m_Emoji = Texture2D::Create("assets/textures/emoji.png");

    m_ActiveScene = CreateRef<Scene>();

    auto commandLineArgs = Application::Get().GetCommandLineArgs();
    if (commandLineArgs.Count > 1)
    {
        auto sceneFilePath = commandLineArgs[1];
        SceneSerializer serializer(m_ActiveScene);
        serializer.Deserialize(sceneFilePath);

    }
    auto sceneFilePath0 = commandLineArgs[0];

    NUT_CORE_WARN(sceneFilePath0);
```

得到这样这样的结果：

```
[15:18:51] APP: Goodbye World!
[15:18:51] NUT: Command line args:
[15:18:51] NUT: Argument 0: E:\VS\Nut\bin\Debug-windows-x86_64\Nut-Editor\Nut-Editor.exe
[15:18:51] NUT: Creating window: Nut Editor (1600 , 900)
[15:18:51] NUT: Initializing GLFW window..
[15:18:51] NUT: OpenGL info:
[15:18:51] NUT: Vendor: NVIDIA Corporation
[15:18:51] NUT: Renderer: NVIDIA GeForce RTX 3050 Laptop GPU/PCIe/SSE2
[15:18:51] NUT: Version: 4.6.0 NVIDIA 536.13
[15:18:51] NUT: E:\VS\Nut\bin\Debug-windows-x86_64\Nut-Editor\Nut-Editor.exe
```

3. 在怎样的影响下，获取的命令行参数或发生变化?

在通常情况下，一旦项目的构架被明确（比如依赖性、文件路径等等），仅对程序进行代码上的“软”处理无法修改从命令行中获取的指令内容，因为这个内容一般是在程序启动时cmd中的内容。此处我们可以看到命令为：“E:\VS\Nut\bin\Debug-windows-x86_64\Nut-Editor\Nut-Editor.exe”

如果想要对其进行修改，可能需要在VS的项目属性页面，进行相关修改：

4.Cherno 为什么进行这样的处理？这个新功能的意图是什么？

分析指令内容：

让我们分析获取的指令：“ E:\VS\Nut\bin\Debug-windows-x86_64\Nut-Editor\Nut-Editor.exe”，这个指令的 argc 为 1，表示只有一段连续的指令。所以 argv 是一个只有一个元素的数组 argv，argv[0] 的内容便是“ ”，而 argv[1] 自然为 null。

先决条件：

首先要明确一点，在 x64 、 Debug 的模式下，如果我们运行这个程序 (Nut-Editor) ,我们会从命令行中固定的获取到诸如：“ E:\VS\Nut\bin\Debug-windows-x86_64\Nut-Editor\Nut-Editor.exe”这样的命令如上文所说，在项目的构架被明确之后，获取到的内容一般就固定下来了。

实际使用时发生的情况：

现在 Cherno 设置了命令行参数的新功能，但其实并不是想通过在某处修改命令内容，或者实时根据命令的变化进行一些操作。而是为了在命令行中运行指令时，开启引擎并进入页面的时候，能够自动预先加载一个场景，让我们查看效果以了解详情：

这里是 Cherno 的使用场景：

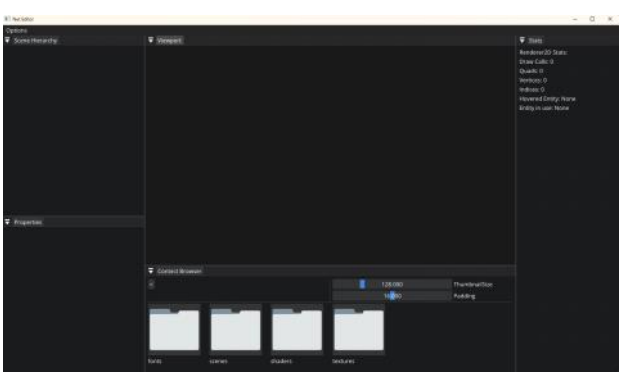
（旧）

一个黄褐色头发的男人，他打开了 cmd ，想要运行 Nut-Editor 应用，于是他输入了一句指令：

argc	1
argv[0]	"E:\VS\Nut\bin\Debug-windows-x86_64\Nut-Editor\Nut-Editor.exe"

通常情况下，这一段指令将直接打开引擎，但并不会在开启时加载一个场景。因为现在只有一段完整的指令，也就是说，这个条件判断不满足：

```
auto commandLineArgs = Application::Get().GetCommandLineArgs();
if (commandLineArgs.Count > 1)
{
    auto sceneFilePath = commandLineArgs[1];
    SceneSerializer serializer(m_ActiveScene);
    serializer.Deserialize(sceneFilePath);
}
auto sceneFilePath0 = commandLineArgs[0];
```

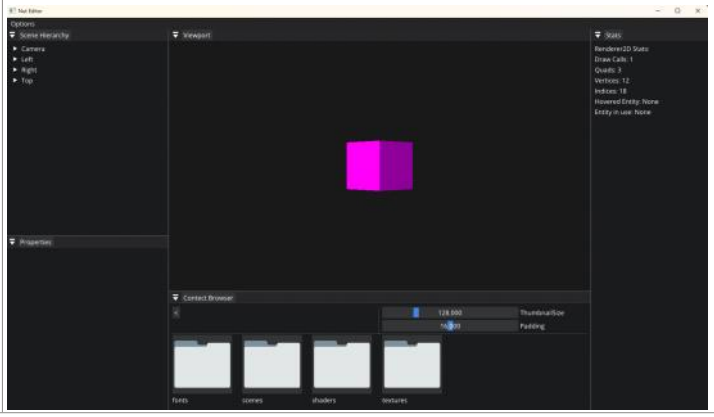


（新）

但是在新功能的加持下，如果我们在该指令之后添加了一个来自场景的目录：

argc	2
argv[0]	"E:\VS\Nut\bin\Debug-windows-x86_64\Nut-Editor\Nut-Editor.exe"
argv[1]	" E:\VS\Nut\Nut-Editor\assets\scenes\3DExample.yaml "

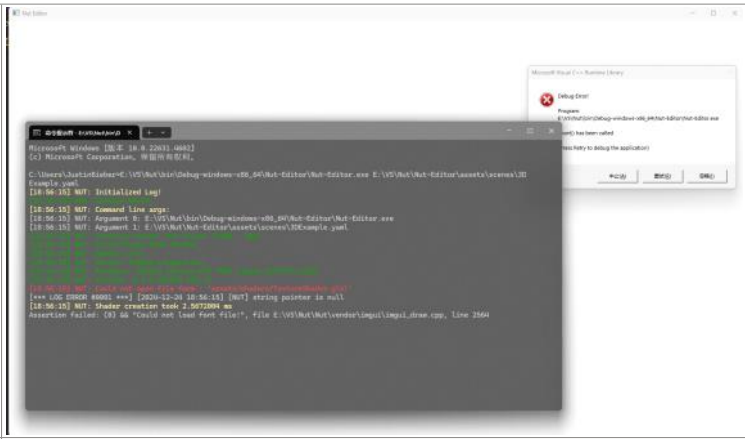
此时运行指令，你将会在启动时看到一个预先加载的场景。



》》》》遇到问题：

在理想状态下，运行指令后，程序应该能正常打开，但实际上我遇到了一些错误。

更新了着色器系统之后，我发现问题似乎出自文件路径。我猜测是绝对路径和相对路径导致的错误。



第一个错误："Could not open file from:...."

现在我将 Renderer2d.cpp 中的代码进行修改：（将此前的相对路径改为绝对路径）

```
// QuadVertex Ptr
s_Data.QuadVBBase = new QuadVertex[s_Data.MaxVertices];                    //保存指针初始位置

// Shader
s_Data.TextureShader = Shader::Create("E:/VS/Nut/Nut-Editorassets/shaders/TextureShader.glsl");

// UBO
s_Data.CameraUniformBuffer = UniformBuffer::Create(sizeof(Renderer2DData::CameraData), 0);

// Texture
```

第二个错误：我发现报错还来自这个函数：AddFontFromFileTTF ,于是我在使用这个函数的时候，将路径改为绝对路径（虽然这会导致该应用的可移植性降低），但着实是无奈之举。

```
ImFont* ImFontAtlas::AddFontFromFileTTF(const char* filename, float size_pixels, const ImFontConfig* font_cfg_template, const ImWchar* glyph_ranges)
{
    IM_ASSERT(!Locked && "Cannot modify a locked ImFontAtlas between NewFrame() and EndFrame/Render()!");
    size_t data_size = 0;
    void* data = ImFileLoadToMemory(filename, "rb", &data_size, 0);
    if (!data)
    {
        IM_ASSERT_USER_ERROR(0, "Could not load font file!");
        return NULL;
    }
    ImFontConfig font_cfg = font_cfg_template ? *font_cfg_template : ImFontConfig();
    if (font_cfg.Name[0] == '\0')
    {
        // Store a short copy of filename into into the font name for convenience
        const char* p;
        for (p = filename + strlen(filename); p > filename && p[-1] != '/' && p[-1] != '\\'; p--) {}
        ImFormatString(font_cfg.Name, IM_ARRAYSIZE(font_cfg.Name), "%s, %.0fpx", p, size_pixels);
    }
    return AddFontFromMemoryTTF(data, (int)data_size, size_pixels, &font_cfg, glyph_ranges);
}
```

ImGuiLayer.cpp 中：

更改前：

```
io.Fonts->AddFontFromFileTTF("assets/fonts/opensans/static/OpenSans-Bold.ttf", 20.0f);
io.FontDefault = io.Fonts->AddFontFromFileTTF("assets/fonts/opensans/static/OpenSans-Regular.ttf", 20.0f);
```

更改后：

```
: Layer("ImGuiLayer")
{
}

void ImGuiLayer::OnAttach()
{
    NUT_PROFILE_FUNCTION();

    // Setup Dear ImGui context
    IMGUI_CHECKVERSION();
    ImGui::CreateContext();
    ImGuiIO& io = ImGui::GetIO(); (void)io;
    io.ConfigFlags |= ImGuiConfigFlags_NavEnableKeyboard;      // Enable Keyboard Controls
    io.ConfigFlags |= ImGuiConfigFlags_NavEnableGamepad;       // Enable Gamepad Controls
    io.ConfigFlags |= ImGuiConfigFlags_DockingEnable;          // Enable Docking
    io.ConfigFlags |= ImGuiConfigFlags_ViewportsEnable;        // Enable Multi-Viewport / Platform Windows
    //io.ConfigViewportsNoAutoMerge = true;
    //io.ConfigViewportsNoTaskBarIcon = true;

    io.Fonts->AddFontFromFileTTF("E:/VS/Nut/Nut-Editor/assets/fonts/opensans/static/OpenSans-Bold.ttf", 20.0f);
    io.FontDefault = io.Fonts->AddFontFromFileTTF("E:/VS/Nut/Nut-Editor/assets/fonts/opensans/static/OpenSans-Bold.ttf", 20.0f);
```

问题三：纹理加载中的路径修复

关于纹理的加载：

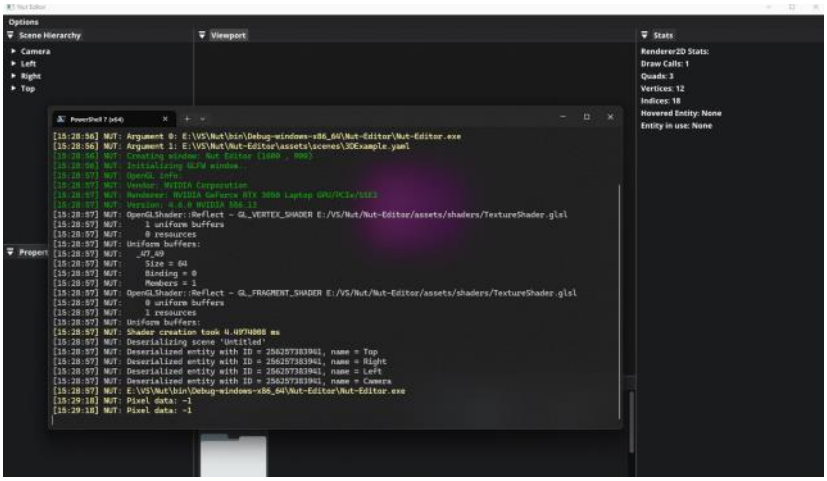
(ContentBrowserPanel.cpp)

```
ContentBrowserPanel::ContentBrowserPanel()
    :m_CurrentDirectory(s_AssetPath)
{
    m_FolderIcon = Texture2D::Create("Resources/Icons/ContentBrowser/DirectoryIcon3.png");
    m_FileIcon   = Texture2D::Create("Resources/Icons/ContentBrowser/FileIcon3.png");
}
```

(EditorLayer.cpp)

```
void EditorLayer::OnAttach()
{
    NUT_PROFILE_FUNCTION();

    m_Framebuffer = FrameBuffer::Create({ 1280, 720, 1, {FrameBufferAttachment

    m_Texture = Texture2D::Create("assets/textures/Checkerboard.png");
    m_Emoji = Texture2D::Create("assets/textures/emoji.png");

    m_ActiveScene = CreateRef<Scene>();
```

现在，便能够通过在终端输入：“E:\VS\Nut\bin\Debug-windows-x86_64\Nut-Editor\Nut-Editor.exe E:\VS\Nut\Nut-Editor\assets\scenes\3DExample.yaml”，来启动游戏引擎，并保证启动时预先加载了一个场景。



值得注意的是，由于缓存的存在，我们需要在保存文件之后，通过VS重新运行项目以刷新 bin/Nur-Editor/Nut-Editor.exe 文件。这样才能保证我们在终端使用指令运行游戏引擎的时候，得到最新的报错日志等信息。

TODO:

这里的调试手段就是将 相对路径 改为了 绝对路径 ，以此避免中断。但这非常影响项目的可移植性，我暂时没有想到好的解决办法，如果有人可以补充，或者此后我有了想法，我会将其合并于项目代码中。

》》》》添加Uniform Buffer

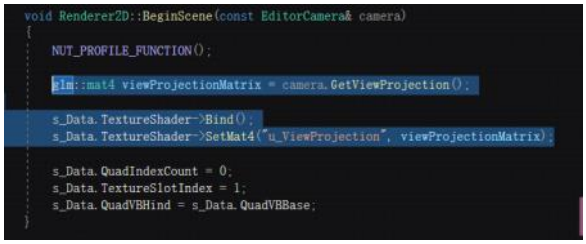
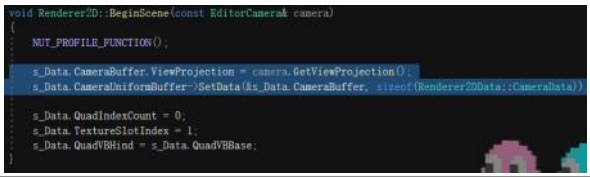
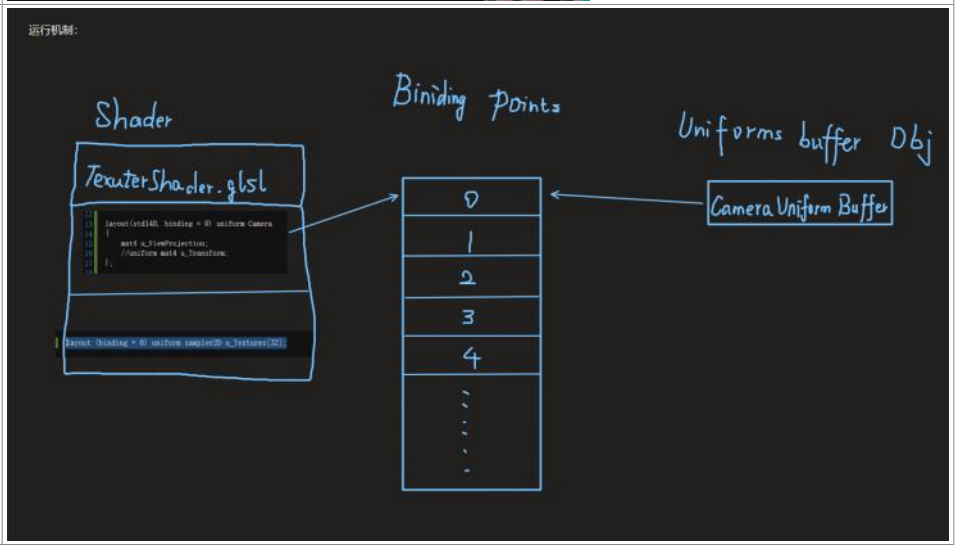
》》关于 Uniform Buffer 的定义：具体可以查看（ https://learnopengl-cn.github.io/04%20Advanced%20OpenGL/08%20Advanced%20GLSL/#uniform ）

etc....

》》操作步骤

现在我们了解了 Uniform Buffer 的原理及其使用方式，现在开始更新代码：

首先	是设置与定义 UniformBuffer （UniformBuffer.h, UniformBuffer.cpp, OpenGLUniformBuffer.h, OpenGLUniformBuffer.cpp）
接着	是修改着色器中的统一变量，将其改为统一变量块（ Uniform 块）
最后	需要更新实际绘制是，绑定统一变量的代码（之前是一个一个绑定，现在可以直接绑定 Uniform 块），使用时方便快捷。

示例：	 
运行机制： 具体可以参考 （https://learnopengl-cn.github.io/04%20Advanced%20OpenGL/08%20Advanced%20GLSL/#uniform_2）	

所以在设置了 Uniform Buffer 之后，可以取消绑定着色器并绑定统一变量的操作：

```
// Shader
s_Data.TextureShader = Shader::Create("assets/shaders/TextureShader.glsl");
//s_Data.TextureShader->Bind();
//s_Data.TextureShader->SetIntArray("u_Textures", samplers, s_Data.MaxTextureSlots);
s_Data.CameraUniformBuffer = UniformBuffer::Create(sizeof(Renderer2DData::CameraData), 0);
```

在更新代码以使用 Uniform Buffer 的时候，我发现一个问题：

前提：

我们在着色器中将两个统一变量更改为统一变量块，他们分别是："u_ViewProjection"和"u_Textures"。

这都是为了UBO的使用而做的更改，因为Uniform buffer的使用需要在着色器统一变量块与UBO之间建立一种联系：“ Binding Points “ -> 绑定点。

修改：

当然，我们也需要在着色器做完更改之后，再去更新相应的代码，比如：

修改前（未使用 Uniform Buffer） `s_Data.TextureShader->SetMat4("u_ViewProjection", camera.GetViewProjectionMatrix());`	在这里，我们直接将 u_ViewProjection 作为一个 mat4 变量传递给着色器，实现统一变量的直接绑定。
修改后（已使用 Uniform Buffer） `s_Data.CameraUniformBuffer = UniformBuffer::Create(sizeof(Renderer2DData::CameraData), 0);` `…..` `…..` `s_Data.CameraUniformBuffer->SetData(&s_Data.CameraBuffer, sizeof(Renderer2DData::CameraData));`	现在我们先创建了UBO，然后将着色器中的统一变量块(Uniform block)通过封装好的函数 "SetData()" ，绑定 UBO 到正确的绑定点 (Binding Point).

```
s_Data.TextureShader = Shader::Create("assets/shaders/Texture.glsl");
s_Data.TextureShader->Bind();
s_Data.TextureShader->SetIntArray("u_Textures", samplers, s_Data.MaxTextureSlots);
```

但是我发现在将 u_Textures 从统一变量设置为统一变量块之后，Cherno 不仅删除了之前显示绑定统一变量的代码，还没有对 u_Textures 进行类似的更新，这让我有点迷惑。

```
#type fragment
#version 450 core

layout(location = 0) out vec4 color;
layout(location = 1) out int color2;

in vec4 v_Color;
in vec2 v_TexCoord;
in float v_TexIndex;
in float v_TilingFactor;
in flat int v_EntityID;

layout (binding = 0) uniform sampler2D u_Textures[32]

void main()
{
    color = texture(u_Textures[int(v_TexIndex)], v_Te

    color2 = v_EntityID;
}
```

这是为何呢?
其实这和 Uniform buffer obj 没有很大的关系，这仅仅与 u_Textures 的一些特性有关。具体来讲，这和 OpenGL 纹理的特性相关。

答案:
纹理是 OpenGL 中的一种特殊资源，在着色器中使用 layout(binding = 0) 声明绑定点后，你只需对纹理进行绑定操作即可（将纹理绑定到对应的纹理单元），OpenGL 会自动处理纹理与着色器变量的映射。因此，在提前声明了 layout(binding = 0) 的情况下，纹理数组不需要像 UBO 那样通过 SetIntArray 或 SetData 来更新。

分析:
1. layout(binding = 0) 的原理
layout(binding = 0) 语法在 GLSL 中告诉 OpenGL，某个 uniform 变量（例如纹理或 UBO）会与一个 **绑定点**（binding point）关联。这种方式是 OpenGL 中的一种标准机制，允许你将资源（如纹理、UBO）直接绑定到特定的资源绑定点，从而避免了逐个设置 uniform 值的麻烦。
具体来说：

- **对于纹理（sampler2D、samplerCube 等）**：当你使用 layout(binding = N) 时，着色器的该纹理变量会与 OpenGL 中的绑定点 N 关联。
- **对于 Uniform Buffer Objects (UBO)**：UBO 的工作方式类似，也需要通过绑定点（binding = N）来绑定到 OpenGL 中某个绑定点

2. 但为什么纹理可以直接通过 layout(binding = 0) 来绑定，而不需要额外的操作?
对于纹理数组（sampler2D u_Textures[32]），其实你并不需要像 UBO 那样来传递数据。因为**纹理绑定**在 OpenGL 中已经是一个非常内建的机制，你只需要使用 layout(binding = N) 来声明绑定点，而不需要手动传递纹理单元索引。就能直接将这些纹理单元与着色器中的纹理数组自动对应。

》》》》OpenGL Shader 更新

》》》》接下来我将对着色器系统进行相关更新

》》》关于 Timer 的使用

示例:

```
class Timer
{
public:
	Timer()
	{
		Reset();
	}

	void Timer::Reset()
	{
		m_Start = std::chrono::high_resolution_clock::now();
	}

	float Timer::Elapsed()
	{
		return std::chrono::duration_cast<std::chrono::nanoseconds>(std::chro
	}

	float Timer::ElapsedMillis()
	{
		return Elapsed() * 1000.0f;
	}

private:
	std::chrono::time_point<std::chrono::high_resolution_clock> m_Start;
};
```

```
{
  // 创建一个 Timer 对象
  Hazel::Timer timer;

  // 第一个操作：模拟一个短时间的操作
  std::this_thread::sleep_for(std::chrono::milliseconds(500)); // 模拟 500 毫秒的延迟
  std::cout << "Time after first operation: " << timer.ElapsedMillis() << " ms\n";

  // 重置计时器
  timer.Reset();

  // 第二个操作：模拟一个稍长的操作
  std::this_thread::sleep_for(std::chrono::seconds(1)); // 模拟 1 秒的延迟
  std::cout << "Time after second operation: " << timer.ElapsedMillis() << " ms\n";

}
```

》》》》对着色器系统进行修改后，需要将 Premake 中的运行时静态链接关掉:

```
staticruntime "on"
改为
staticruntime "off"
```

包括 Nut/premake5.lua 、 Nut-Editor/premake5.lua、Nut/vendor/yaml-cpp/premake5.lua 这三个文件中的相关代码。

》》》》着色器中的 Location 要求

SPIR-V 作为 Vulkan 的中间表示语言，需要为每个输入/输出变量分配一个 **location** 值（为输入和输出变量明确指定 **location** 属性），以便于着色器编译器正确地将这些变量与 GPU 的管线绑定。在 OpenGL 中，某些输入/输出变量（如顶点属性、uniforms等）可以通过其他方式来绑定。而在 Vulkan 中，SPIR-V **显式**要求在着色器中为所有的输入和输出变量指定唯一的 location。

比如：

```
struct VertexOutput
{
    vec4  Color;
    vec2  TexCoord;
    float TexIndex;
    float TilingFactor;
};

layout(location = 0) out VertexOutput Output;
layout(location = 4) out flat int v_EntityID;

void main()
{
    Output.Color = a_Color;
    Output.TexCoord = a_TexCoord;
    Output.TexIndex = a_TexIndex;
    Output.TilingFactor = a_TilingFactor;
    v_EntityID = a_EntityID;

    gl_Position = u_ViewProjection * vec4(a_Position, 1.0);
    //gl_Position = u_ViewProjection * u_Transform * vec4(a_Position, 1.0);
}
```

》》》》我差不多是直接复制了 OpenGLShader 更新的代码，所以没有仔细查看，可能会补充关于更新的理解笔记，我也不知道。

TODO:

在我做出更改之前，如果有人补充着色器中代码更新的细则与用意，我可以将其合并进来。

》》》》平台工具的更新（打开或保存文件）

》》》》这里我不用做更改，因为我的代码似乎是正确的。

》》》》视口与摄像机更新

》》》》这里只是新增一些判断条件，非常简单。

Anyway，这一集的提交应该到此结束了。这期间过了很久，并不是因为这一集很难，而是因为期末事情比较多，时间比较赶紧。
现在终于提交完毕了，无论如何，请享受接下来的学习。

---------- Content browser panel ----------

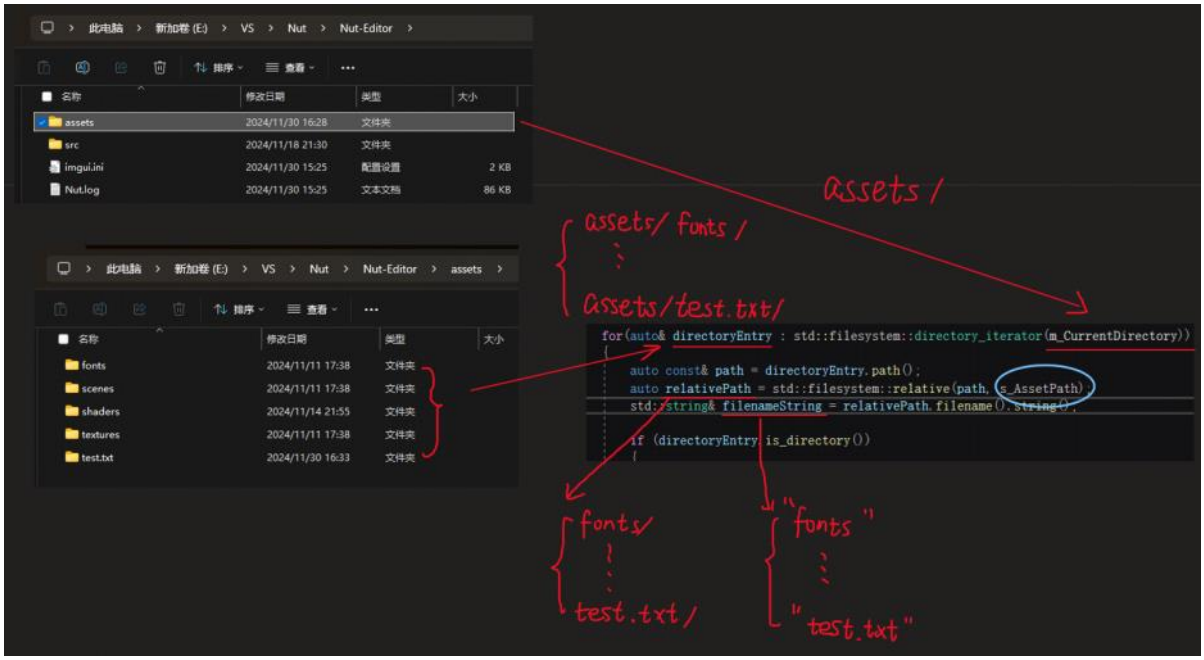
》》》》std::filesystem::relative( )

```
const auto& path = directoryEntry.path();
auto relativePath = std::filesystem::relative(path, s_AssetPath);
```

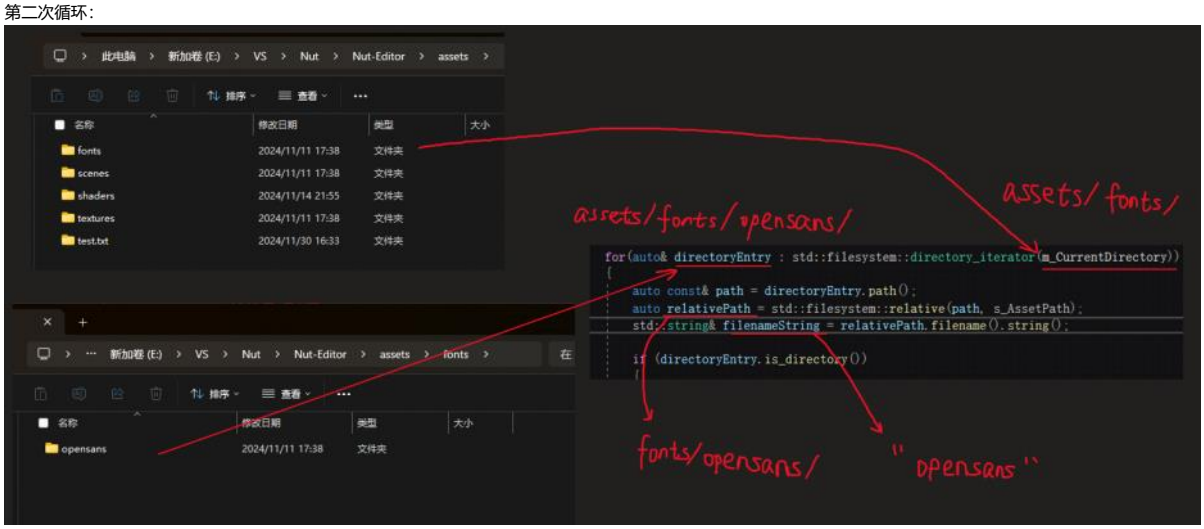
如果 s_AssetPath 是 C:\Projects\MyGame\Assets，path 是 C:\Projects\MyGame\Assets\Models\Character.obj。
那么 std::filesystem::relative(path, s_AssetPath) 会返回 Models\Character.obj，这是 path 相对于 s_AssetPath 的相对路径。

》》操作图示:

第一次循环:



第二次循环：



》》》》"/=" 运算符重载

Eg.

"m_CurrentDirectory /= path.filename();"

/= 运算符的重载

概念：

在 C++17 的 std::filesystem::path 中，/= 运算符是被重载的，用于拼接路径。其功能是将路径对象 path 中的部分与左侧的路径进行合并。

使用要求：

m_CurrentDirectory 是一个表示当前目录的路径，通常是一个 std::filesystem::path 类型的对象。path.filename() 返回的是 path 对象中的文件名部分，且其类型也是 std::filesystem::path。

示例说明：

假设

m_CurrentDirectory	C:\Projects\MyGame\Assets。
path	C:\Projects\MyGame\Assets\Models\Character.obj。
path.filename()	Character.obj。

那么，m_CurrentDirectory /= path.filename(); 的结果会是 **m_CurrentDirectory 等于 C:\Projects\MyGame\Assets\Character.obj**

》》》》ImGui::Columns(columnCount, 0, false);

ImGui::Columns()

原型：

void ImGui::Columns(int columns_count = 1, const char* id = NULL, bool border = true);

参数解释：

columns_count (类型：int，默认值：1)	功能：指定列的数量。默认值是 1，表示只有一列。如果你想创建多个列，可以设置为大于 1 的数字。
id (类型：const char*，默认值：NULL)	功能：这是一个可选的字符串，用来指定一个唯一的 ID。 如果多个列使用相同的 ID，ImGui 会为它们创建一个统一的状态。这个 ID 在 ImGui 的内部用于区分不同的列布局，但如果不需要区分，可以传入 NULL 或忽略它。
border (类型：bool，默认值：true)	功能：指定是否显示列之间的边框。如果为 true，列之间会有一个分隔线。如果为 false，则没有边框，列之间没有分隔线。

示例：	示例：ImGui::Columns(3) 表示创建 3 列布局。
	示例：ImGui::Columns(3, "MyColumns")，通过指定 ID，可以在后续的操作中区分不同的列布局。
	示例：ImGui::Columns(3, NULL, false) 表示创建 3 列，并且不显示列间的边框。

》》》》一段错误代码诱发的思考：

错误的：

```
Ref<Texture> icon = (directoryEntry.is_directory() ? m_FolderIcon : m_FileIcon);
if (ImGui::ImageButton((ImTextureID)icon->GetRendererID(), { thumbnailSize, thumbnailSize }, {0, 1}, {1, 0}))
{
    if (ImGui::IsItemHovered() && ImGui::IsMouseDoubleClicked(ImGuiMouseButton_Left))
    {
        if (directoryEntry.is_directory())
            m_CurrentDirectory /= path.filename();
    }
}
```

如果将 ImGui::ImageButton() 放在条件判断中，会导致优先判断按钮是否被单击，随后才会判断使用者是否在指定区域双击图标，这会导致鼠标双击的逻辑不能正常触发。

正确的

```
Ref<Texture> icon = (directoryEntry.is_directory() ? m_FolderIcon : m_FileIcon);
ImGui::ImageButton((ImTextureID)icon->GetRendererID(), { thumbnailSize, thumbnailSize }, { 0, 1 }, { 1, 0 });
if (ImGui::IsItemHovered() && ImGui::IsMouseDoubleClicked(ImGuiMouseButton_Left))
{
    if (directoryEntry.is_directory())
        m_CurrentDirectory /= path.filename();
}
```

》》》》ImGui::TextWrapped()

概念：

ImGui::TextWrapped() 是一个用于在 ImGui 中显示文本的函数，主要特点是当文本内容超出当前窗口或控件的宽度时，会自动换行显示。

这个特性适用于显示多行文本，因为文本宽度是动态的，可以适应父容器的大小。这避免了手动计算的麻烦。

函数原型：

void ImGui::TextWrapped(const char* fmt, ...);

void ImGui::TextWrapped(const std::string& str);

参数：

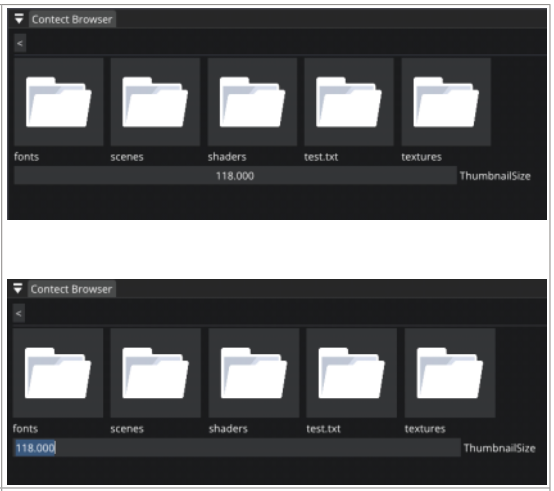
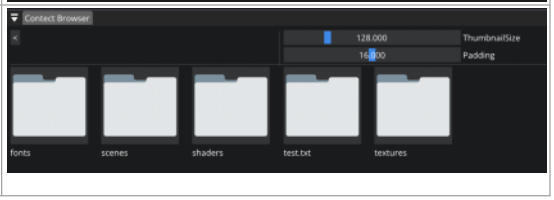
- fmt：一个格式化字符串，允许你使用 ImGui 的格式化语法来插入变量。例如，可以传入一个字符串，或者传入多个参数，通过 fmt 来格式化它们。
- str：传入一个 std::string 对象。它会自动转化为 C 字符串并显示在界面上。

用法：

1. 基本用法：	`ImGui::TextWrapped("This is a very long line of text that will automatically wrap when it reaches the edge of the window.");`
2. 与格式化字符串一起使用： 你可以通过格式化字符串来显示动态内容。例如显示文件名、错误信息等。	`const char* filename = "example.txt";` `ImGui::TextWrapped("The file %s has been loaded successfully.", filename);`
3. 使用 std::string： 如果你有一个 std::string 对象，也可以直接传给 TextWrapped。	`std::string filename = "example.txt";` `ImGui::TextWrapped(filename);  // 直接显示 std::string 的内容`

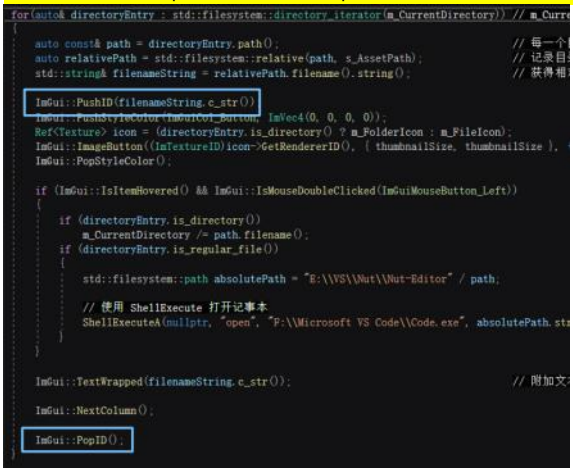
》》》》DragFloat 和 SliderFloat 的区别。

ImGui::DragFloat 和 ImGui::SliderFloat 的区别

DragFloat： 既可以通过鼠标在输入框中直接滑动，也可以输入值。	
SliderFloat： 只能操作滑块来改变大小。	

---------- Content browser panel (Drag & drop) ----------

》》》》PushID 和 PopID 的作用是什么？PopID 是否可以放在 if 条件判断之前？



```
for(auto& directoryEntry : std::filesystem::directory_iterator(m_CurrentDirectory)) // m_Curre
{
    auto const& path = directoryEntry.path();                                       // 每一个
    auto relativePath = std::filesystem::relative(path, s_AssetPath);               // 记录目
    std::string& filenameString = relativePath.filename().string();                 // 获得相

    ImGui::PushID(filenameString.c_str());
    ImGui::PushStyleColor(ImGuiCol_Button, ImVec4(0, 0, 0, 0));
    Ref<Texture> icon = (directoryEntry.is_directory() ? m_FolderIcon : m_FileIcon);
    ImGui::ImageButton((ImTextureID)icon->GetRendererID(), { thumbnailSize, thumbnailSize },
    ImGui::PopStyleColor();

    if (ImGui::IsItemHovered() && ImGui::IsMouseDoubleClicked(ImGuiMouseButton_Left))
    {
        if (directoryEntry.is_directory())
            m_CurrentDirectory /= path.filename();
        if (directoryEntry.is_regular_file())
        {
            std::filesystem::path absolutePath = "E:\\VS\\Nut\\Nut-Editor" / path;

            // 使用 ShellExecute 打开记事本
            ShellExecuteA(nullptr, "open", "P:\\Microsoft VS Code\\Code.exe", absolutePath.str
        }
    }

    ImGui::TextWrapped(filenameString.c_str());                                     // 附加文

    ImGui::NextColumn();

    ImGui::PopID();
```

一：PushID 和 PopID 的作用

在 ImGui 中，当你渲染多个相似的控件（例如多个互动式按钮）时，它们通常会基于 ID 来管理自己的状态（如是否被点击、是否被悬停）。如果没有使用 PushID，这些控件可能会因为共享相同的 ID 而相互干扰（例如，所有的按钮都会共享同一个按下状态，或者鼠标悬停状态）。

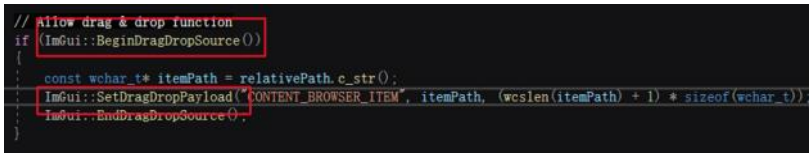
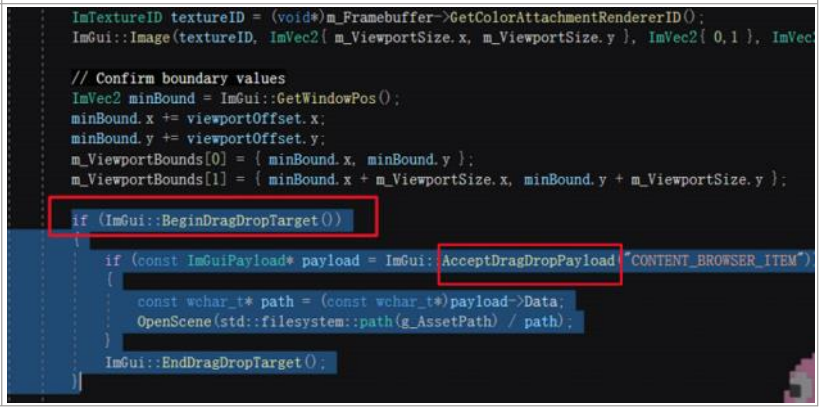
通过 PushID 和 PopID，你确保每次循环渲染时，都为每个控件生成一个独特的 ID，这样每个文件的按钮、拖放等行为都能独立工作。

二：PopID 是否可以放在 If 判断之前？：不可以

如果在创建完控件之后就结束 ID 的作用范畴，接下来的条件判断 if(ImGui::IsItemHovered() && ImGui::IsMouseDoubleClicked()) 将不再依赖于正确的 ID，而是随机的对某些按钮进行响应，这可能导致行为不一致或 UI 控件无法正常工作。

》》》》BeginDragDropTarget() 使用细则

如果手动跟进了 Cherno 的代码，我们会发现，使用 DragDrop 功能只需要两步操作：设置拖动源、设置拖动目标。

拖动源的设置 (ContentBrowserPanel.cpp)	
拖动目标的设置 （EditorLayer.cpp)	

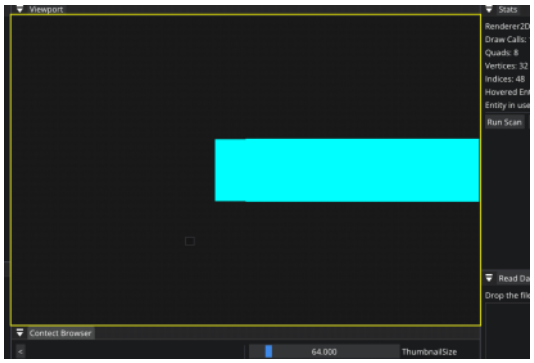
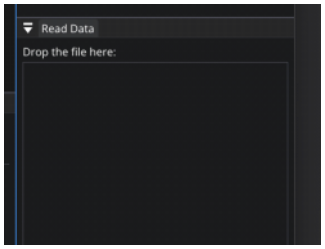
》》计算const char* 类型字符串

如果有这样一个变量： const char* path = "abc/def/g"。计算其长度时，如下两种方式，一个错一个对：

Sizeof(path)	错误：这行代码只计算了指针的大小，而不是整个字符串的大小。（指针->指的是 "abc/def/g"中首字符的内存位置，也就是 'a' 在内存中的存储位置。
(Strlen(path) + 1) * size(char)	正确： strlen() 计算字符串的长度，但不包含 '\0'，故加一。然后对其乘以 char 类型的大小，得到正确结果。

》》关于拖拽预览的绘制，还需要注意一点：

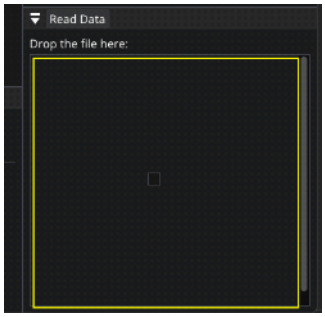
注意：在使用 BeginDragDropTarget( ) 之前，需要绘制一个有效的交互区域。

比如在视口的设置之后，我们使用了BeginDragDropTarget( )，你会发现在拖动文件到视口区域时，视口的可用区域会高亮，并且能够处理后续文件拖入操作。 可是如果注释掉 ImGui::Image() 这一行代码，你会发现拖动文件的功能会无响应。 这是因为 ImGui::Image 不仅显示了图像，还会自动处理它的交互区域，因此它是一个“有效”的拖放目标。	
如果你只绘制了一个窗口，或者在窗口中放置了Text,Child等“不可交互”的空间，可用区域高亮便不会出现。同样的，文件拖动也会不起作用。 Eg.	 

```
// ---------- Read Txt Data ----------
ImGui::Begin("Read Data");
ImGui::Text("Drop the file here:");

ImVec2 targetSize = ImGui::GetContentRegionAvail();

// 创建一个子窗口，用于显示拖放目标区域
if (ImGui::BeginChild("DropTarget", targetSize, true))
{
    // 通过Dummy元素创建一个可交互的区域
    //ImGui::Dummy(ImVec2(targetSize.x, targetSize.y));  // 只创建一个空的区域

    // Allow Drag & drop
    if (ImGui::BeginDragDropTarget())
    {
        if (const ImGuiPayload* payload = ImGui::AcceptDragDropPayload("CONTENT_BROWSER_ITEM"))
        {
            const wchar_t* path = (const wchar_t*)payload->Data;
            ReadTxtFile(std::filesystem::path(g_AssetPath) / path);
        }
        ImGui::EndDragDropTarget();
    }
}

ImGui::EndChild();
ImGui::End();
```

此时便需要我们创建一个可交互的区域：ImGui::Button、ImGui::Dummy 等等控件，以此来完善文件拖动的功能。

```
// 创建一个子窗口，用于显示拖放目标区域
if (ImGui::BeginChild("DropTarget", targetSize, true))
{
    // 通过Dummy元素创建一个可交互的区域
    ImGui::Dummy(ImVec2(targetSize.x, targetSize.y));  // 只创建一个空的区域

    // Allow Drag & drop
    if (ImGui::BeginDragDropTarget())
    {
        if (const ImGuiPayload* payload = ImGui::AcceptDragDropPayload("CONTENT
        {
            const wchar_t* path = (const wchar_t*)payload->Data;
            ReadTxtFile(std::filesystem::path(g_AssetPath) / path);
        }
        ImGui::EndDragDropTarget();
    }
}
```



》》什么是 ImGui::Dummy

概念：
ImGui::Dummy 是 ImGui 提供的一个函数，用于创建一个“占位符”或“虚拟”元素，它不会渲染任何实际的内容，但可以用来占据空间或提供一个交互区域。

主要用途：	占位符：ImGui::Dummy 可以作为一个占位符，帮助你设置一些占用空间但不渲染任何实际内容的区域。这对于需要控制布局、调整空间或创建拖放目标区域非常有用。 控制布局：通过 ImGui::Dummy，你可以创建精确的布局区域，而不会干扰其他控件的显示。例如，当你需要创建一个特定大小的区域来接收拖放操作时，可以使用 Dummy 来占据空间。
语法：	void ImGui::Dummy(const ImVec2& size);
参数：	size：指定占位符的大小，通常是一个 ImVec2（x 和 y 坐标）。这定义了 Dummy 占据的区域的大小。
示例：	假设你想在 ImGui 窗口中创建一个区域，它不会显示任何内容，但你希望它占据一个特定的空间： `ImGui::Begin("Example Window");` `// 创建一个大小为 200x200 的占位符区域` `ImGui::Dummy(ImVec2(200, 200));` `ImGui::End();`