

----- 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)
(前庭仅供个人参考，并无侵犯版权的想法，若违反版权条款，并非本人意图)

个人在学习过程中觉得值得查阅的几个文档：

| | |
|--|---|
| 游戏开发者大会文档 (关于 SPIR-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/VulkanGLSL1pp.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/oMys8a78T) |
| 大原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：

| | |
|---|---|
| <p>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）</p> <p>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。</p> <p>(参考文献1-来自 LearnOpenGL 教程：https://learnopengl-cn.github.io/04%20Advanced%20OpenGL/08%20Advanced%20GLSL/)</p> <p>(参考文献2-来自 Vulkan 教程：https://vulkan-tutorial.com/Uniform_buffers/Descriptor_layout_and_buffer#page-Uniform-buffer)</p> <p>建议阅读全文，这样理解更加深刻。</p> | <p>• Uniform buffer</p> <h3>使用Uniform缓冲</h3> <p>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，但我们还没有讨论如何使用它们。</p> <p>首先，我们需要调用 <code>glGenBuffers</code>，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 <code>GL_UNIFORM_BUFFER</code> 目标，并调用 <code>glBufferData</code>，分配足够的内存。</p> <pre>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);</pre> <p>现在，每当我们需要对缓冲更新或插入数据，我们都会绑定到 <code>uboExampleBlock</code>，并使用 <code>glBufferSubData</code> 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？</p> <p>在OpenGL上下文中，定义了一些绑定点(Binding Point)，我们可以将一个Uniform缓冲中链接至它。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。下面的这个图示例了这个：</p> <p>• Uniform buffer.</p> <h3>Uniform buffer 均匀缓冲</h3> <p>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.</p> <p>在下一章中，我们将指定包含着色器 UBO 数据的缓冲。但我们需要首先创建该缓冲。我们将每帧复制数据到该缓冲，因此，在这种情况下，它只会增加额外的开销，并可能会降低性能而不是提高性能。</p> <p>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</p> <p>我们应该有多个缓冲。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，并写入 GPU 当前未读取的统一缓冲。</p> <p>To that end, add new class members for <code>uniformBuffers</code>, and <code>uniformBuffersMemory</code>:</p> <p>为此，为 <code>uniformBuffers</code> 和 <code>uniformBuffersMemory</code> 添加新的类成员：</p> <pre>VkBuffer IndexBuffer; VkDeviceMemory IndexBufferMemory; std::vector<VkBuffer> uniformBuffers; std::vector<VkDeviceMemory> uniformBuffersMemory; std::vector<uint> uniformBuffersMapped;</pre> <p>Similarly, create a new function <code>createUniformBuffers</code> that is called after <code>createIndexBuffer</code> and allocates the buffers:</p> <p>类似地，创建一个新函数 <code>createUniformBuffers</code>，该函数在 <code>createIndexBuffer</code> 之后调用并分配缓冲：</p> <pre>void InitVulkan() { ... createVertexBuffer(); createIndexBuffer(); createUniformBuffers(); }</pre> |
| <p>3.OpenGL 和 Vulkan 在着色器语言上的使用规范，还有不同之处。</p> <p>参考文献：OpenGL教程 (https://learnopengl-cn.github.io/02%20Lighting/03%20Materials/)</p> <p>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/VulkanGLSL1pp.pdf)</p> <p>附录： 参考文献：Github 中文 Readme (https://github.com/zenny-chen/GLSL-for-Vulkan)</p> | <p>• GLSG 中的结构体示例：</p> <pre>#version 330 core struct Material { vec3 ambient; vec3 diffuse; vec3 specular; float shininess; }; uniform Material material;</pre> <p>在片段着色器中，我们创建一个结构体(struct)来存储物体的材质属性。我们也可以把它们存储为独立的uniform值，但是作为一个结构体来存储会更简单一些。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。</p> <p>• 如果查看 Vulkan API 在编写着色器时使用 GLSL 的语法规则，可以查看 Github 仓库 (中文：https://github.com/zenny-chen/GLSL-for-Vulkan) 或查看 Vulkan 的官方入门指南 (http://vulkan-tutorial.com/Introduction)</p> |

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

或前往 Vulkan! 教程门户网站 (<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:

```
gl_VertexIndex
gl_InstanceIndex
```

- 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



oeb - December 17, 2020

How Vulkan GLSL Differs from OpenGL GLSL

5

Shader combinations of separate texture data and samplers:

```
uniform sampler s;
uniform texture2D t;
vec4 rgba = texture( sampler2D( t, s ), vST );
```

Descriptor Sets:

```
layout( set=0, binding=0 ) ... ;
```

Push Constants:

```
layout( push_constant ) ... ;
```

Specialization Constants:


```
layout( constant_id = 3 ) const int N = 5;
```

- Only for scalars, but a vector's components can be constructed from specialization constants

Specialization Constants for Compute Shaders:

```
layout( local_size_x_id = 8, local_size_y_id = 16 );
```

- This sets `gl_WorkGroupSize.x` and `gl_WorkGroupSize.y`
`gl_WorkGroupSize.z` is set as a constant



oeb - December 17, 2020

4.SPIR-V的使用思路，使用逻辑。

参考文献: SPIR-V 官网 (https://www.khronos.org/spi/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/VulkanGLSL1pp.pdf>)

参考文献: 2016 年 3 月 - 游戏开发者大会 (<https://www.neilheming.dev/web-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>)

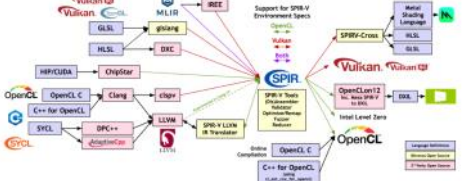
• SPIR-V 的生态系统:

SPIR-V Language Ecosystem

SPIR-V 语言生态系统

The SPIR-V ecosystem includes a rich variety of language front-ends (compilers), development tools and run-times (consumers).

SPIR-V 生态系统包括丰富的语言前端(编译器)、开发工具和运行时(消费者)。



• 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**, 现在与 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 的更详细的信息

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 是面向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 有一些关于 SPIR-V 及其优点的非常好的 [白皮书](#), 以及对表示的清晰描述。这里和这里还有 2016 年 Vulkan DevDay 的两种精彩的 Khronos 演示 (两者的视频)。

• SPIR-V 管线:



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

SPIR-V 簡介

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。

| | |
|---|---|
| <p>OpenGL 与 SPIR-V 的工作模式:</p> <p>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 着色器的中间表示, SPIR-V 也被引入到 OpenGL 中。</p> <p>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。</p> <p>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。</p> <p>Vulkan 与 SPIR-V 的工作模式:</p> <p>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。</p> <p>参考文献: 游戏开发者大会 2016 (https://www.neilhenning.dev/wp-content/uploads/2015/03/AnintroductionToSPIR-V.pdf)</p> | <p>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 着色器的中间表示, SPIR-V 也被引入到 OpenGL 中。</p> <p>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。</p> <p>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。</p> <p>Vulkan 与 SPIR-V 的工作模式:</p> <p>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。</p> <p>参考文献: 游戏开发者大会 2016 (https://www.neilhenning.dev/wp-content/uploads/2015/03/AnintroductionToSPIR-V.pdf)</p> <p>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。</p> <p>示例:</p> <pre> ; 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" %30 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 </pre> |
|---|---|

实际使用实例：

| | |
|--------------------------------|--|
| 1. GLSL 源代码编写 | <p>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。</p> <p>示例：GLSL 着色器</p> <pre>#version 450 out vec4 FragColor; void main() { FragColor = vec4(1.0, 0.0, 0.0, 1.0); // 输出红色 }</pre> |
| 2. GLSL 编译为 SPIR-V | <p>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。</p> <p>工具1：</p> <p>glslang (Khronos 提供的编译器，广泛用于将 GLSL 转换为 SPIR-V)。</p> <p>编译过程：</p> <p>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。</p> <p>工具2：</p> <p>你也可以使用命令行工具 glslangValidator 来编译 GLSL 代码。</p> <p>编译过程：</p> <p>使用命令：<code>glslangValidator -V shader.glsl -o shader.spv</code></p> <p>这将会把 <code>shader.glsl</code> 编译成 <code>shader.spv</code>，即 SPIR-V 二进制文件。</p> |
| 3. 加载 SPIR-V 到 Vulkan 或 OpenGL | <p>3.1 在 OpenGL 中使用 SPIR-V</p> <p>前提提要：</p> |

| | |
|------------|---|
| | <p>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。</p> <p>加载过程：</p> <p>示例：</p> <pre>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);</pre> <p>-----</p> <p>3.2 在 Vulkan 中使用 SPIR-V</p> <p>加载过程：</p> <p>创建一个 VkShaderModule 对象，该对象包含 SPIR-V 二进制代码。</p> <p>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）。</p> <p>示例：Vulkan 使用 SPIR-V</p> <pre>// 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</pre> |
| 4. 执行着色器程序 | <p>在 OpenGL 中，SPIR-V 着色器程序被链接到程序对象中，并通过调用 <code>glUseProgram</code> 来激活该程序，之后通过绘制调用来执行。</p> <p>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 <code>vkCmdDraw</code>）或计算命令（例如 <code>vkCmdDispatch</code>）来执行。</p> |

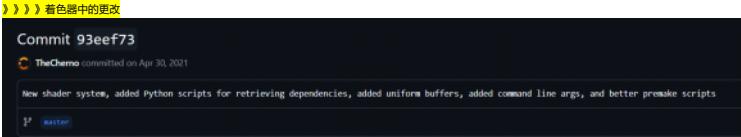
》》》上述涉及语言的纵向对比图

| | |
|---|--|
| GLSL | <pre>#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 }</pre> |
| <p>SPIR-V</p> <p>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。</p> <p>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。</p> <p>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。</p> | <p>SPIR-V</p> <pre>#302 2307 0000 0100 0100 0000 0e00 0000 0000 0000 1100 0200 0100 0000 0000 0000 0100 0000 474c 534c 2e73 7464 2e34 3530 0000 0000 0e00 0300 0000 0000 0100 0000 0f00 0000 0400 0000 0400 0000 6d61 695e 0000 0000 0900 0000 1000 0300 0400 0000 0700 0000 0300 0300 0200 0000 8c00 0000 0500 0400 0400 0000 6d61 695e 0000 0000 0500 0000 0900 0000 675c 5f46 7261 6743 6ffc 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 0800 0000 0a00 0000 cdc c3e 2b00 0400 0600 0000 0b00 0000 cdc c43f 2b00 0400 0600 0000 0c00 0000 0000 803f 2c00 0700 0700 0000 0d00 0000 6a00 0000 0a00 0000 0000 0000 0e00 0000 3600 0500 0200 0000 0400 0000 0000 0000 0300 0000 f800 0200 0500 0000 1e00 0100 0900 0000 0000 0000 f000 0100 3800 0100</pre> <p>SPIR-V Assembly</p> <pre>; 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</pre> |
| OpenGL | <pre>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);</pre> |

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;
```



以下是详细解释:

1 premake脚本更改
(and better premake scripts)

2 + -- Hazel Dependencies
3 +
4 + VULKAN_SDK = os.getenv("VULKAN_SDK")

```
15 + IncludeDir["shaders"] = "${wks.location}/Hazel/vendor/shaders/include"
16 + IncludeDir["SPIRV_Cross"] = "${wks.location}/Hazel/vendor/SPIRV-Cross"
17 + IncludeDir["VulkanSDK"] = "${VULKAN_SDK}/include"
18 +
19 + LibraryDir = {}
20 +
21 + LibraryDir["VulkanSDK"] = "${VULKAN_SDK}/lib"
22 + LibraryDir["VulkanSDK_Debug"] = "${wks.location}/Hazel/vendor/VulkanSDK/lib"
23 +
24 + Library = {}
25 + Library["Vulkan"] = "${LibraryDir.VulkanSDK}/vulkan-1.lib"
26 + Library["VulkanGLES"] = "${LibraryDir.VulkanSDK}/VkLayer_utility.lib"
27 +
28 + Library["ShaderC_Debug"] = "${LibraryDir.VulkanSDK_Debug}/shaderc_shared.lib"
29 + Library["SPIRV_Cross_Debug"] = "${LibraryDir.VulkanSDK_Debug}/spirv-cross-core.lib"
30 + Library["SPIRV_Cross_GLES_Debug"] = "${LibraryDir.VulkanSDK_Debug}/spirv-cross-gles.lib"
31 + Library["SPIRV_Tools_Debug"] = "${LibraryDir.VulkanSDK_Debug}/SPIRV_Tools.lib"
32 +
33 + Library["ShaderC_Release"] = "${LibraryDir.VulkanSDK}/shaderc_shared.lib"
34 + Library["SPIRV_Cross_Release"] = "${LibraryDir.VulkanSDK}/spirv-cross-core.lib"
35 + Library["SPIRV_Cross_GLES_Release"] = "${LibraryDir.VulkanSDK}/spirv-cross-gles.lib"
```

premake5.lua

```
1 --
2 -- @ - 3.4 - 4.5 88
3 include "${wks.location}/premake/customization/solution_items.lua"
4 + include "Dependencies.lua"
5
6 workspace "Hazel"
7 architecture "x86_64"
8
9 @ - 23,17 +24,6 @ workspace "Hazel"
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000
```

```
73 +         "H[library.ShaderC_Debug]",
74 +         "H[library.SPIRV_Cross_Debug]",
75 +         "H[library.SPIRV_Cross_GLSL_Debug]"
76 +     }
77 +
78 +     filter "configurations:Release"
79 +     defines "NDEBUG"
80 +     runtime "Release"
81 +     optimize "on"
82 +
83 +     links
84 +     [
85 +         "H[library.ShaderC_Release]",
86 +         "H[library.SPIRV_Cross_Release]",
87 +         "H[library.SPIRV_Cross_GLSL_Release]"
88 +     ]
89 +
90 +     filter "configurations:Dist"
91 +     defines "NDEBUG"
92 +     runtime "Release"
93 +     optimize "on"
94 +
95 +     links
96 +     [
97 +         "H[library.ShaderC_Release]",
98 +         "H[library.SPIRV_Cross_Release]",
99 +         "H[library.SPIRV_Cross_GLSL_Release]"
100 +     ]
```

2.py脚本
(Python scripts for retrieving dependencies)

```
1 + import subprocess
2 + import pkg_resources
3 +
4 + def install(package):
5 +     print(f"Installing {package} module...")
6 +     subprocess.check_call([python, "-m", "pip", "install", package])
7 +
8 + def validate_package(package):
9 +     required = { package }
10 +     installed = {pkg.key for pkg in pkg_resources.working_set}
11 +     missing = required - installed
12 +
13 +     if missing:
14 +         install(package)
15 +
16 + def validate_packages():
17 +     validate_package('requests')
18 +     validate_package('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。

```
1 + import os
2 + import subprocess
3 + import CheckPython
4 +
5 + # Make sure everything we need is installed
6 + CheckPython.ValidatePackages()
7 +
8 + import Vulkan
9 +
10 + # Change from Scripts directory to root
11 + os.chdir('../')
12 +
13 + if (not Vulkan.CheckVulkanSDK()):
14 +     print("Vulkan SDK not installed.")
15 +
16 + if (not Vulkan.CheckVulkanSDKDebugLibs()):
17 +     print("Vulkan SDK debug libs not found.")
18 +
19 + print("Running premake...")
20 + subprocess.call(["cmd", "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。

```
1 + import requests
2 + import sys
3 + import time
4 +
5 + from fake_useragent import UserAgent
6 +
7 + def DownloadFile(url, filepath):
8 +     with open(filepath, 'wb') as f:
9 +         ua = UserAgent()
10 +         headers = {'User-Agent': ua.stream}
11 +         response = requests.get(url, headers=headers, stream=True)
12 +         total = response.headers.get('content-length')
13 +
14 +         if total is None:
15 +             f.write(response.content)
16 +         else:
17 +             downloaded = 0
18 +             total = int(total)
19 +             startTime = time.time()
20 +             for data in response.iter_content(chunk_size=max(int(total/1000), 1024*1024)):
21 +                 downloaded += len(data)
22 +                 f.write(data)
23 +                 done = int(10*downloaded/total)
24 +                 percentage = (downloaded / total) * 100
25 +                 elapsedTime = time.time() - startTime
26 +                 avgPerSecond = (downloaded / 1024) / elapsedTime
27 +                 avgSpeedString = '{:2f} MB/s'.format(avgPerSecond)
28 +                 if (avgPerSecond > 1024):
29 +                     avgPerSecond = avgPerSecond / 1024
30 +                 avgSpeedString = '{:2f} MB/s'.format(avgPerSecond)
```

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”。

用于检查和安装 Vulkan SDK

`InstallVulkanPrompt()`: 提示用户是否安装 Vulkan SDK。

CheckVulkanSDKDebugLibs(): 检查 Vulkan SDK 的调试库是否存在

Downloaded from <http://ajph.org/> on November 10, 2015

10. *Journal of the American Medical Association*, 2000; 284: 1039-1044.

100

Page 10 of 10

36 45 17 6

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

```
# Hazel/src/Hazel/Renderer/Renderer2D.cpp
```

```
1 //
2 // @ -3,9 +3,10 @@
3
4 #include "Hazel/Renderer/VertexArray.h"
5 #include "Hazel/Renderer/Index.h"
6 #include "Hazel/Renderer/TextureManager.h"
7 #include "Hazel/Renderer/RenderCommand.h"
8
9 #include <glm/gtc/matrix_transform.hpp>
10 #include <glm/gtc/type_ptr.hpp>
11
12 namespace Hazel {
13
14     @ -43,6 +45,13 @@ namespace Hazel {
15
16         glm::vec4 QuadVertexPosition[4];
17
18         Renderer2D::Statistics Stats;
19
20         struct CameraData
21         {
22             glm::mat4 ViewProjection;
23             CameraData* ViewFrustum;
24             Ref<RenderTarget> ColorRenderTarget;
25
26     };
27 }
```


5 着色器系统更新:
(New shader system)

```
1 // src/main/scala/Timer.scala
2
3 import akka.actor._
4
5 class Timer {
6
7   // class Timer
8   {
9     public void reset() {
10       // reset()
11     }
12
13     // void Timer::reset()
14     {
15       // m_start = std::chrono::high_resolution_clock::now();
16     }
17
18     // float Timer::Elapsed()
19     {
20       // return std::chrono::duration_cast<std::chrono::duration<
21       // float>::type>(m_start - std::chrono::high_resolution_clock::now()).count();
22     }
23
24     // float Timer::ElapsedMillis()
25     {
26       // return Elapsed() * 1000.0f;
27     }
28
29     // private:
30 }
```

[illegible]

6 平台工具的更新（打开或保存文件）

```

> H:\src\Platform\Windows\WindowsPlatformUtil.cpp
18 19
19 20 namespace Hwapi {
21 22
23 23 - std::optional<string> FindBinaries(ScopeIn(const char* filter)
24 24 + std::string FindBinaries(ScopeIn(const char* filter)
25 25
26 26 OPEN_LLWMMIA api;
27 27
28 28 CHW sFile[256] = { 0 };
29 29
30 30 Hw -18,19 -20 namespace Hwapi {
31 31
32 32 if (GetCmdLineParam(hFile) == TRUE)
33 33 return api.IsProfile();
34 34
35 35 return std::empty();
36 36
37 37
38 38 - std::optional<string> FindBinaries(ScopeIn(const char* filter)
39 39 + std::string FindBinaries(ScopeIn(const char* filter)
40 40
41 41 {
42 42 OPEN_LLWMMIA api;
43 43
44 44 CHW sFile[256] = { 0 };

```

[illegible]

>>> 以下是这些 py 文件的结构:

运行脚本时，请关闭代理。

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。

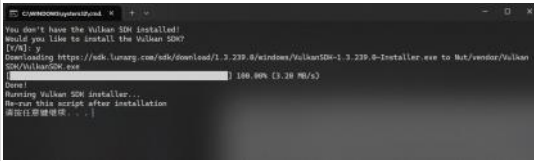
附录：如果你想进入官网查看适合你系统的 SDK，以下是网址 -> (<https://vulkan.lunarg.com/sdk/home>)



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。

```
48 VulkanSDKDebugLibURL = "https://files.lunarg.com/sdk/1.2.178.0/vulkanSDK-1.2.178.0-DebugLibs.zip"
49 OutputDirectory = "nut/vendor/VulkanSDK"
50 TempZipFile = ("OutputDirectory/VulkanSDK.zip")
51
52
53 def CheckVulkanSDKDebugLibs():
54     shadercdlib = Path(("OutputDirectory/lib/shaderc_shared.lib"))
55     if (not shadercdlib.exists()):
56         print("No Vulkan SDK debug libs found. (checked {shadercdlib})")
57         print("Downloading", VulkanSDKDebugLibURL)
58         with urlopen(VulkanSDKDebugLibURL) as zipresp:
59             with ZipFile(BytesIO(zipresp.read())) as zfile:
60                 zfile.extractall(OutputDirectory)
61         print("Vulkan SDK debug libs located at {OutputDirectory}")
62     return True
```

我们重新运行一遍，使用更新之后的 SDK install。

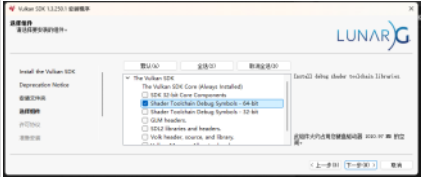


于是运行后出现这样的窗口：

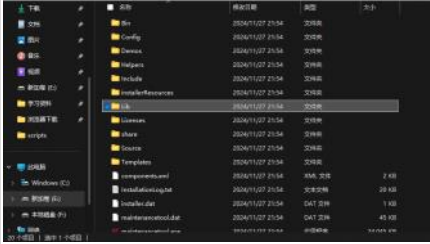


安装 vulkan SDK

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树架：



问题三

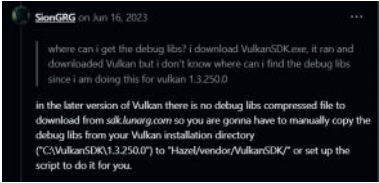
我们发现 Cherno 另外下载了一个 Debuglib.zip，并对其进行了一些处理。

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

```
48 VulkanSDKDebugLibURL = "https://files.lunarg.com/sdk/1.2.178.0/vulkanSDK-1.2.178.0-DebugLibs.zip"
49 OutputDirectory = "nut/vendor/VulkanSDK"
50 TempZipFile = ("OutputDirectory/VulkanSDK.zip")
51
52
53 def CheckVulkanSDKDebugLibs():
54     shadercdlib = Path(("OutputDirectory/lib/shaderc_shared.lib"))
55     if (not shadercdlib.exists()):
56         print("No Vulkan SDK debug libs found. (checked {shadercdlib})")
57         print("Downloading", VulkanSDKDebugLibURL)
58         with urlopen(VulkanSDKDebugLibURL) as zipresp:
59             with ZipFile(BytesIO(zipresp.read())) as zfile:
60                 zfile.extractall(OutputDirectory)
61         print("Vulkan SDK debug libs located at {OutputDirectory}")
62     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

```
VulkanSDKDebugLibURL = "https://files.lunarg.com/sdk/1.2.178.0/vulkanSDK-1.2.178.0-DebugLibs.zip"
OutputDirectory = "nut/vendor/VulkanSDK"
TempZipFile = ("OutputDirectory/VulkanSDK.zip")
```

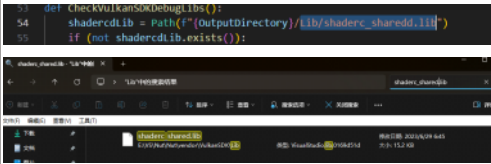
首先，我对这个 shaderc_shredd.lib 的路径有点疑惑：因为我的确查找到了 shaderc_shared.lib 这个库，而不是 shaderc_shredd.lib，

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。

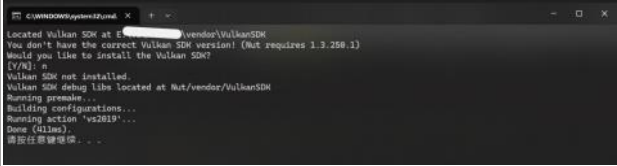
我便做了以下更改：（仅仅是口头提醒一下~）

随后我重新运行 Setup.bat，并拒绝再次安装 installer，使得有这样的结果：

```
def CheckVulkanSDKDebugLibs():
    shadercLib = Path(f"{OutputDirectory}/lib/shaderc_shredd.lib")
    if (not shadercLib.exists()):
        print("No Vulkan SDK debug libs found. (checked {shadercLib})")
        print("Downloading", VulkanSDKDebugLibURL)
        with urlopen(VulkanSDKDebugLibURL) as zipresp:
            with zipfile.ZipFile(zipresp.read()) as zipfile:
                zipfile.extractall(OutputDirectory)
        print(f"Vulkan SDK debug libs located at {OutputDirectory}")
        return True
```

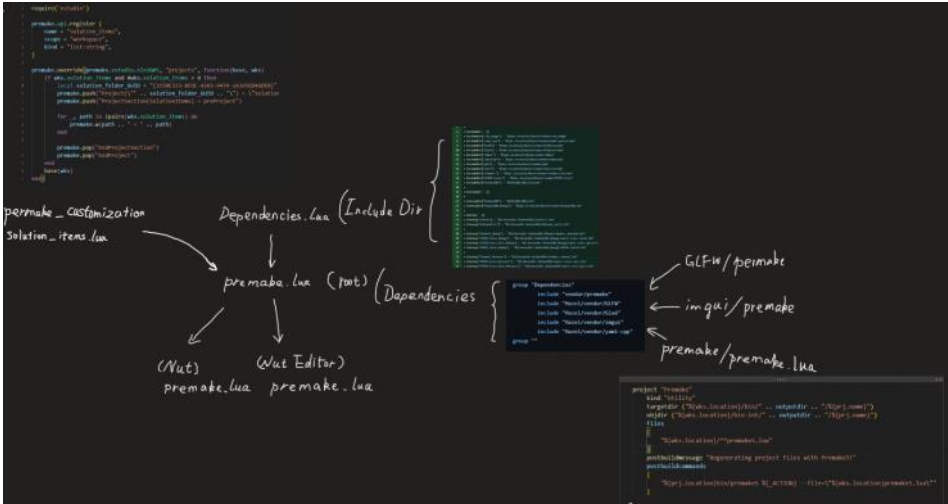


```
def CheckVulkanSDKDebugLibs():
    shadercLib = Path(f"{OutputDirectory}/lib/shaderc_shredd.lib")
    if (not shadercLib.exists()):
        print("No Vulkan SDK debug libs found. (checked {shadercLib})")
        # print("Downloading", VulkanSDKDebugLibURL)
        # with urlopen(VulkanSDKDebugLibURL) as zipresp:
        #     with zipfile.ZipFile(zipresp.read()) as zipfile:
        #         zipfile.extractall(OutputDirectory)
        print("Please check you Vulkan SDK files. (checked https://vulkan.lunarg.com/sdk/home and find you version, or reinstall you Vulkan SDK by installer)")
        # return False
        raise RuntimeError("Vulkan SDK debug libs not found, process aborted.")
    print(f"Vulkan SDK debug libs located at {OutputDirectory}")
    return True
```



我想应该是对的。

》》》》现在我们已成功安装了 Vulkan，现在则需要更新 premake 文件内容。
这是将实现的 premake 文件架构图（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）。

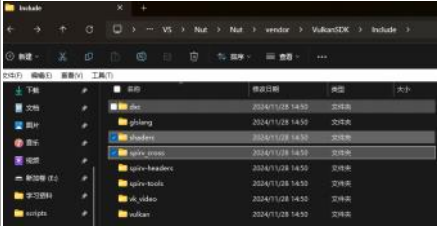
第一步，我们在项目的根目录下重新编写一个 premake 文件，这个文件主要用来索引 vendor 中的外部库（API）

```

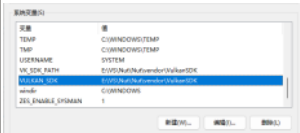
1  #if defined __linux__
2  #include "vk_layer_dispatch_table.h"
3  #include "vk_layer_utils.h"
4
5  #define VKMAN_SDK "os_getenv("VKMAN_SDK")"
6
7  #includeDir = {}
8  #includeDir["stb_image"] = "%(wks.location)/nut/vendor/stb_image"
9  #includeDir["yaml_cpp"] = "%(wks.location)/nut/vendor/yaml-cpp/include"
10 #includeDir["glfw"] = "%(wks.location)/nut/vendor/glfw/include"
11 #includeDir["glad"] = "%(wks.location)/nut/vendor/glad/include"
12 #includeDir["imgui"] = "%(wks.location)/nut/vendor/imgui"
13 #includeDir["tinyxml2"] = "%(wks.location)/nut/vendor/tinyxml2"
14 #includeDir["glm"] = "%(wks.location)/nut/vendor/glm"
15 #includeDir["fmt"] = "%(wks.location)/nut/vendor/fmt/include"
16 #includeDir["shaderc"] = "%(wks.location)/nut/vendor/shaderc/include"
17 #includeDir["SPIRV_cross"] = "%(wks.location)/nut/vendor/SPIRV_cross"
18 #includeDir["vulkan_sdk"] = "%(VKMAN_SDK)/include"
19
20 #libraryDir = {}
21
22 #libraryDir["vulkan_sdk"] = "%(VKMAN_SDK)/lib"
23 #libraryDir["vulkan_sdk_debug"] = "%(wks.location)/nut/vendor/VulkanSDK/lib"
24
25 #library = {}
26
27 #library["shaderc_debug"] = "%(libraryDir.vulkan_sdk_debug)/shaderc_shared.lib"
28 #library["SPIRV_cross_debug"] = "%(libraryDir.vulkan_sdk_debug)/spirv-cross-core.lib"
29 #library["SPIRV_cross_GLSL_debug"] = "%(libraryDir.vulkan_sdk_debug)/spirv-cross-gslc.lib"
30 #library["SPIRV_tools_debug"] = "%(libraryDir.vulkan_sdk_debug)/SPIRV-tools.lib"
31
32 #library["shaderc_release"] = "%(libraryDir.vulkan_sdk)/shaderc_shared.lib"
33 #library["SPIRV_cross_release"] = "%(libraryDir.vulkan_sdk)/spirv-cross-core.lib"
34 #library["SPIRV_cross_GLSL_release"] = "%(libraryDir.vulkan_sdk)/spirv-cross-gslc.lib"

```

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



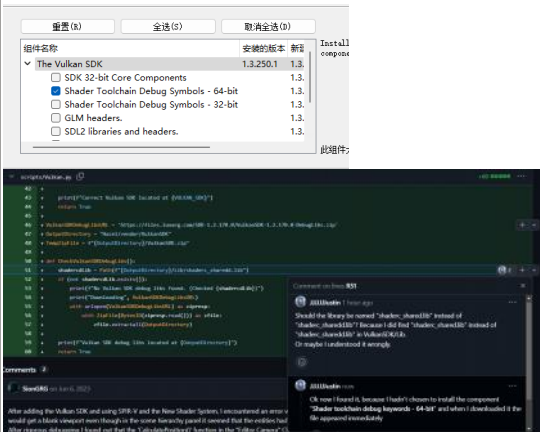
系统变量示例:



而且由于我没有下载某些组件，这使很多文件并不存在。(将其标注出来)

[illegible]

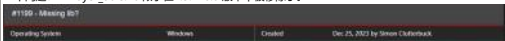
于是我决定下载拓展(shader toolchain debug symbols), 这一通过运行 maintenancetool.exe 文件实现:



这个组件将会解决这部分问题：

[illegible]

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



VKLayer_utils.lib 这个文件也不存在。



所以这是 premake 文件最新的样子:

```

def load_data():
    """加载数据"""
    # 加载训练数据
    train_data_loader = DataLoader(
        dataset=MyDataset(train_data_path, train_data_loader),
        batch_size=128,
        shuffle=True,
        num_workers=4,
        pin_memory=True,
    )

    # 加载验证数据
    val_data_loader = DataLoader(
        dataset=MyDataset(val_data_path, val_data_loader),
        batch_size=128,
        shuffle=False,
        num_workers=4,
        pin_memory=True,
    )

    return train_data_loader, val_data_loader

# 训练模型
def train_model(model, train_data_loader, val_data_loader):
    """训练模型"""
    # 训练模型
    for epoch in range(1, 101):
        # 训练模型
        train_loss, train_acc = train_one_epoch(model, train_data_loader)
        # 验证模型
        val_loss, val_acc = validate_one_epoch(model, val_data_loader)

        # 打印训练和验证结果
        print(f'Epoch {epoch}: train_loss={train_loss}, train_acc={train_acc}, val_loss={val_loss}, val_acc={val_acc}')

    return model

# 保存模型
def save_model(model, save_path):
    """保存模型"""
    torch.save(model.state_dict(), save_path)

# 加载模型
def load_model(save_path):
    """加载模型"""
    model_state_dict = torch.load(save_path)
    model = MyModel()
    model.load_state_dict(model_state_dict)
    return model

# 主函数
def main():
    # 加载数据
    train_data_loader, val_data_loader = load_data()

    # 训练模型
    model = train_model(MyModel(), train_data_loader, val_data_loader)

    # 保存模型
    save_model(model, 'model.pth')

    # 加载模型
    model = load_model('model.pth')

    # 测试模型
    test_loss, test_acc = test_one_epoch(model, test_data_loader)

    # 打印测试结果
    print(f'Test loss={test_loss}, test_acc={test_acc}')

if __name__ == '__main__':
    main()

```

》》》》 操作步骤:

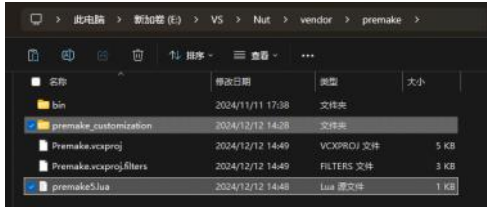
》》111 现在我们将 Nut/premake.lua 中的表单独存放在另一个文件中(Dependencies.lua)

其中包括：

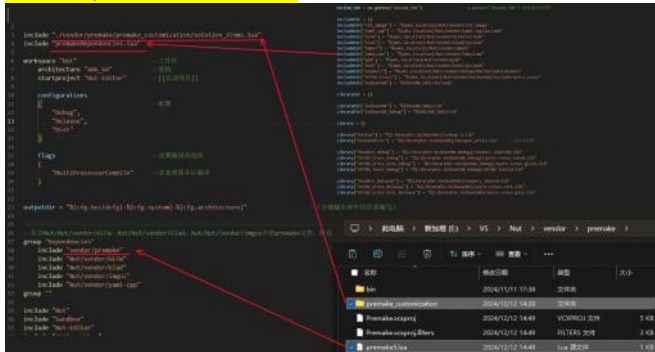
[illegible]

》》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



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

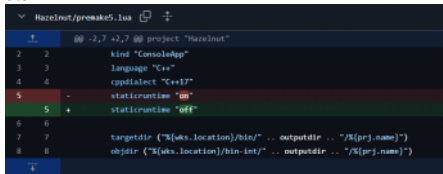
具体内容是：Nut-premake 文件需要包含 Vulkan 的库目录，并在对应配置下添加相关链接。

》》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。

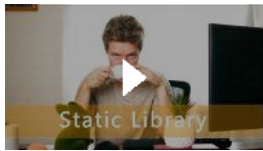
这表示禁用静态链接运行时库，使用动态链接的运行时库。意味着程序在运行时将依赖外部的动态链接库（DLL），而不是将运行时库直接嵌入到可执行文件中。

示例:



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 I 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 PullRequests 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。 |

》》》

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

》》》上一集提交过多, 我先将内容浏览器做完, 并提交。

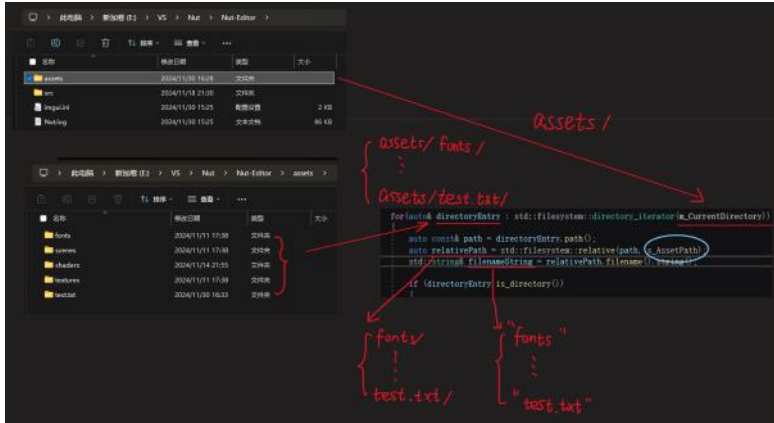
》》》std::filesystem::relative()

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

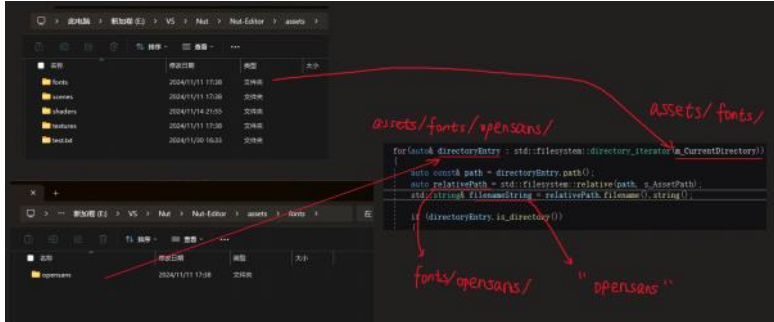
如果 s_AssetPath 是 C:\Projects\MyGame\Assets, path 是 C:\Projects\MyGame\Assets\Models\Character.obj, 那么 std::filesystem::relative(path, s_AssetPath) 会返回 Models\Character.obj, 这是 path 相对于 s_AssetPath 的相对路径。

》》》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((ImGuiTextureID)icon->GetRenderID(), { thumbnailSize, thumbnailSize }, { 0, 1 }, { 1, 0 })){
    if (ImGui::IsItemHovered() && ImGui::IsMouseClicked(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((ImGuiTextureID)icon->GetRenderID(), { thumbnailSize, thumbnailSize }, { 0, 1 }, { 1, 0 });
if (ImGui::IsItemHovered() && ImGui::IsMouseClicked(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) -----

》》》BeginDragDropTarget() 使用细则

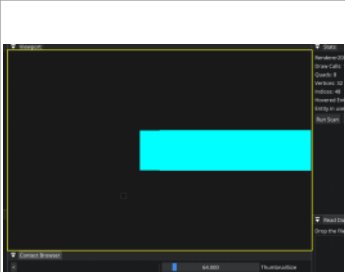
如果手动走进了 Chernov 的代码, 我们会发现, 使用 DragDrop 功能只需要两步操作: 设置拖动物源, 设置拖动目标。

| | |
|---------|---|
| 拖动物源的设置 | <pre>// Allow drag & drop function if (ImGui::BeginDragDropSource()) { const wchar_t* itemPath = relativePath.c_str(); ImGui::SetDragDropPayload("CONTENT_BROWSER_ITEM", itemPath, wcslen(itemPath) * sizeof(wchar_t)); ImGui::EndDragDropSource(); }</pre> |
| 拖动目标的设置 | <pre>int32_t textureID = 0; ImGui::Image(textureID, ImVec2(m_VisportSize.x, m_VisportSize.y), ImVec2(0, 0), ImVec2(1, 1)); // Define boundary values int32_t midbound = ImGui::GetIO().KeyMap[ImGuiKey_Space]; midbound.x = m_VisportSize.x; midbound.y = m_VisportSize.y; m_VisportSize.x = (midbound.x, midbound.y); m_VisportSize.y = (midbound.x, midbound.y); if (ImGui::BeginDragDropTarget()) { if (ImGui::AcceptDragDropPayload("CONTENT_BROWSER_ITEM")) { const wchar_t* path = (const wchar_t*)ImGui::GetDragDropPayload->Data; OpenDataFile(filesystem::path(AssetPath / path)); } ImGui::EndDragDropTarget(); }</pre> |

》》》可是还需要注意一点:

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TextChild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。

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

ImGui::Text("targetSize = ImGui::GetIO().KeyMap[ImGuiKey_Space];");
// 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ImGui::TextChild创建一个可交互的区域
    // 通过ImGui::TextChild创建一个可交互的区域
    // Allow drag & drop
    if (ImGui::BeginDragDropTarget())
    {
        if (ImGui::AcceptDragDropPayload("CONTENT_BROWSER_ITEM"))
        {
            const wchar_t* path = (const wchar_t*)ImGui::GetDragDropPayload->Data;
            ReadFile( filesystem::path( 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ImGui::TextChild创建一个可交互的区域
    ImGui::TextChild(targetSize.x, targetSize.y); // 只创建一个空区域

    // Allow drag & drop
    if (ImGui::BeginDragDropTarget())
    {
        if (ImGui::AcceptDragDropPayload("CONTENT_BROWSER_ITEM"))
        {
            const wchar_t* path = (const wchar_t*)ImGui::GetDragDropPayload->Data;
            ReadFile( filesystem::path( 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，但希望你它占据一个特定的空间： <pre>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();</pre> |