

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

》》》这次 Chernov 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。
我做了一些笔记，请认真浏览。
实际操作步骤请转到：《》》》我将逐次的提交这些代码，并记录自己的疑惑

《》》》介绍与引入

《》》》 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 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/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：

<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者插入数据，我们就会绑定到<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。下面的这个图展示了这个：</p> <p>The diagram illustrates the mapping of uniform blocks from two shaders to a set of binding points and then to specific uniform buffer objects. Shader A contains 'uniform Matrices' and 'uniform Lights'. Shader B contains 'uniform Matrices' and 'uniform Data'. Binding points 0 and 1 are shown. Shader A's 'uniform Matrices' is mapped to binding point 0, and Shader B's 'uniform Matrices' is mapped to binding point 1. Binding point 0 is linked to 'uboMatrices', and binding point 1 is linked to 'uboData'. Binding point 2 is linked to 'uboLights'.</p> <p>• Uniform buffer:</p>
--	---

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 staging 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/VulkanGLSL1pp.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 的语法规则，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 GLSL4

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

OpenGL uses:

`gl_VertexID`
`gl_InstanceID`

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

mpg - December 17, 2020

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

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>)

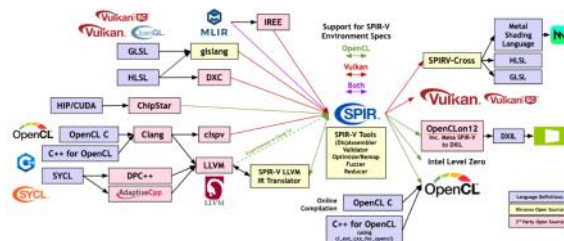
参考文献: 2016 年 3 月 - 游戏开发者大会
(<https://www.neilhenning.dev/wp-content/uploads/2015/03/AnIntroductionToSPIR-V.pdf>)

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 Language Ecosystem
SPIR-V 语言生态系统

The SPIR-V ecosystem includes a rich variety of language front-ends (producers), development tools and run-times (consumers).
(SPIR-V 生态系统包括丰富多样的语言前端 (生产者)、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**, 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 [SPIR-V Guide](#) for more in detail information about SPIR-V

请阅读 [SPIR-V 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 及其优势的非常好的 [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:

The Shaders' View of the Basic Computer Graphics Pipeline

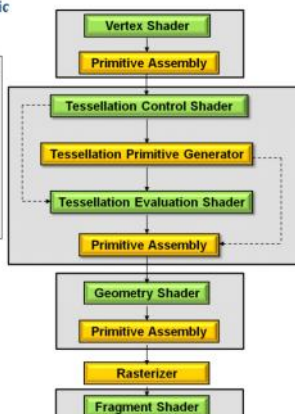
- In general, you want to have a vertex and fragment shader as a minimum.

- A missing stage is OK. The output from one stage becomes the input of the next stage that is there.

- The last stage before the fragment shader feeds its output variables into the rasterizer. The interpolated values then go to the fragment shaders

Fixed Function

Programmable



• SPIR-V 的使用流程:

》》》 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 H LSL）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：	<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 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。</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>
Vulkan 与 SPIR-V 的工作模式：	<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>
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>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" %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>

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： 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： 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： 你也可以使用命令行工具 glslangValidator 来编译 GLSL 代码。</p> <p>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。</p>
3. 加载 SPIR-V 到 Vulkan 或 OpenGL 中	<p>3.1 在 OpenGL 中使用 SPIR-V</p> <p>前提提要： 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。</p> <p>加载过程：</p> <p>示例：</p>

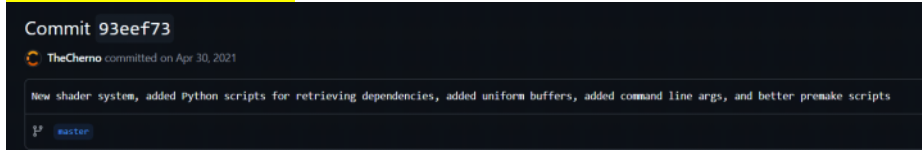
	<pre>GLuint program = glCreateProgram(); // 加载 SPIR-V 二进制文件 GLuint shader = glCreateShader(GL_VERTEX_SHADER); glShaderBinary(1, &shader, GL_SHADER_BINARY_FORMAT_SPIR_V, spirvData, spirvDataSize); glSpecializeShader(shader, "main", 0, nullptr, nullptr); // 绑定和链接程序 glAttachShader(program, shader); glLinkProgram(program);</pre> <div></div> <h3>3.2 在 Vulkan 中使用 SPIR-V</h3> <p>加载过程：</p> <p>创建一个 <code>VkShaderModule</code> 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 <code>spirv-as</code>）转化为二进制格式，供 Vulkan 或 OpenGL 使用。</p>	<pre>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 cdc0 cc3e 2b00 0400 0600 0000 0b00 0000 cdc0 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</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	<pre>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;</pre>

《》《》具体代码更改细则



以下是更新详情（图示）：

1 premake脚本更改
(and better premake scripts)

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

15 + IncludeDir["shaderc"] = "%{wks.location}/Hazel/vendor/shaderc/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["VulkanUtils"] = "%{LibraryDir.VulkanSDK}/VkLayer_utils.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_GLSL_Debug"] = "%{LibraryDir.VulkanSDK_Debug}/spirv-cross-glsl.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_GLSL_Release"] = "%{LibraryDir.VulkanSDK}/spirv-cross-glsl.lib"

-----

premake5.lua
... @@ -1,4 +1,5 @@
1 include "../vendor/premake/premake_customization/solution_items.lua"
2 + include "Dependencies.lua"
3
4 workspace "Hazel"
5 architecture "x86_64"
@@ -23,17 +24,6 @@ workspace "Hazel"
23
24
25 outputdir = "%{cfg.buildcfg}-%{cfg.system}-%{cfg.architecture}"
26
27
28 -- Include directories relative to root folder (solution directory)
29 - IncludeDir = {}
30 - IncludeDir["GLFW"] = "%{wks.location}/Hazel/vendor/GLFW/include"
31 - IncludeDir["Glad"] = "%{wks.location}/Hazel/vendor/Glad/include"
32 - IncludeDir["ImGui"] = "%{wks.location}/Hazel/vendor/ImGui"
33 - IncludeDir["glm"] = "%{wks.location}/Hazel/vendor/glm"
34 - IncludeDir["stb_image"] = "%{wks.location}/Hazel/vendor/stb_image"
35 - IncludeDir["entt"] = "%{wks.location}/Hazel/vendor/entt/include"
36 - IncludeDir["yaml_cpp"] = "%{wks.location}/Hazel/vendor/yaml-cpp/include"
37 - IncludeDir["imgui_ao"] = "%{wks.location}/Hazel/vendor/imgui_ao"
38 -
39 group "Dependencies"
40 include "vendor/premake"
41 include "Hazel/vendor/GLFW"
```

```
▼ Hazelnut/premake5.lua
+ @@ -2,7 +2,7 @@ project "Hazelnut"
2 2      kind "ConsoleApp"
3 3      language "C++"
4 4      cppdialect "C++17"
5 -      staticruntime "on"
5 +      staticruntime "off"
6 6
7 7      targetdir ("%{wks.location}/bin/" .. outputdir .. "%{prj.name}")
8 8      objdir ("%{wks.location}/bin-int/" .. outputdir .. "%{prj.name}")
+ +
```

```
▼ Hazel/premake5.lua
+ @@ -2,7 +2,7 @@ project "Hazel"
2 2      kind "StaticLib"
3 3      language "C++"
4 4      cppdialect "C++17"
5 -      staticruntime "on"
5 +      staticruntime "off"
6 6
7 7      targetdir ("%{wks.location}/bin/" .. outputdir .. "%{prj.name}")
8 8      objdir ("%{wks.location}/bin-int/" .. outputdir .. "%{prj.name}")
+ +
+ @@ -40,7 +40,8 @@ project "Hazel"
40 40      "%{IncludeDir.stb_image}",
41 41      "%{IncludeDir.entt}",
42 42      "%{IncludeDir.yaml_cpp}",
43 -      "%{IncludeDir.ImGuizmo}"
43 +      "%{IncludeDir.ImGuizmo}",
44 +      "%{IncludeDir.VulkanSDK}"
44 45      }
45 46
46 47      links
+ +
+ @@ -67,12 +68,13 @@ project "Hazel"
67 68      runtime "Debug"
68 69      symbols "on"
69 70
71 +      links
72 +      {
73 +          "%{Library.ShaderC_Debug}",
74 +          "%{Library.SPIRV_Cross_Debug}",
75 +          "%{Library.SPIRV_Cross_GLSL_Debug}"
76 +      }
```

```
▼ Hazel/premake5.lua
+ +
73 +      "%{Library.ShaderC_Debug}",
74 +      "%{Library.SPIRV_Cross_Debug}",
75 +      "%{Library.SPIRV_Cross_GLSL_Debug}"
76 +      }
77 +
70 78      filter "configurations:Release"
71 79      defines "HZ_RELEASE"
72 80      runtime "Release"
73 81      optimize "on"
74 82
83 +      links
84 +      {
85 +          "%{Library.ShaderC_Release}",
86 +          "%{Library.SPIRV_Cross_Release}",
87 +          "%{Library.SPIRV_Cross_GLSL_Release}"
88 +      }
89 +
75 90      filter "configurations:Dist"
76 91      defines "HZ_DIST"
77 92      runtime "Release"
78 93      optimize "on"
94 +
95 +      links
96 +      {
97 +          "%{Library.ShaderC_Release}",
98 +          "%{Library.SPIRV_Cross_Release}",
99 +          "%{Library.SPIRV_Cross_GLSL_Release}"
100 +      }
```

2 py脚本
(Python scripts for retrieving dependencies)

```
▼ scripts/CheckPython.py
+ +
+ + @@ -0,0 +1,18 @@
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 ValidatePackage(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 ValidatePackages():
17 +     ValidatePackage('requests')
18 +     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 @@
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(["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 @@
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.chrome}
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(50*downloaded/total)
24 +                 percentage = (downloaded / total) * 100
25 +                 elapsedTime = time.time() - startTime
26 +                 avgKBPerSecond = (downloaded / 1024) / elapsedTime
27 +                 avgSpeedString = '{:.2f} KB/s'.format(avgKBPerSecond)
28 +                 if (avgKBPerSecond > 1024):
29 +                     avgKBPerSecond = avgKBPerSecond / 1024
30 +                     avgSpeedString = '{:.2f} MB/s'.format(avgKBPerSecond)
```

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 @@
1 + import os
2 + import subprocess
3 + import sys
4 + from pathlib import Path
5 +
6 + import Utils
7 +
8 + from io import BytesIO
9 + from urllib.request import urlopen
10 + from zipfile import Zipfile
11 +
12 + VULKAN_SDK = os.environ.get("VULKAN_SDK")
13 + VULKAN_SDK_INSTALLER_URL = 'https://sdk.lunarg.com/sdk/download/1.2.170.0/windows/vulkan_sdk.exe'
14 + HAZEL_VULKAN_VERSION = '1.2.170.0'
15 + VULKAN_SDK_EXE_PATH = 'Hazel/vendor/VulkanSDK/VulkanSDK.exe'
16 +
17 + def InstallVulkanSDK():
18 +     print('Downloading {} to {}'.format(VULKAN_SDK_INSTALLER_URL, VULKAN_SDK_EXE_PATH))
19 +     Utils.DownloadFile(VULKAN_SDK_INSTALLER_URL, VULKAN_SDK_EXE_PATH)
20 +     print('Done!')
21 +     print("Running Vulkan SDK installer...")
22 +     os.startfile(os.path.abspath(VULKAN_SDK_EXE_PATH))
23 +     print("Re-run this script after installation")
24 +
25 + 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缩。

3 Application 中的 ApplicationCommandLineArgs
(added command line args)

```
Hazel/src/Hazel/Core/Application.cpp
+
+  @ -13,7 +13,8 @@ namespace Hazel {
13
14
15     Application* Application::s_Instance = nullptr;
16
17     Application::Application(const std::string& name)
18     : Application(const std::string& name, ApplicationCommandLineArgs args)
19     : m_CommandLineArgs(args)
20
21     {
22         HZ_PROFILE_FUNCTION();
23     }
24 }
```



```

Hazel/src/Hazel/Core/Application.h
18 + struct ApplicationCommandLineArgs
19 + {
20 +     int Count = 0;
21 +     char** Args = nullptr;
22 +
23 +     const char* operator[](int index) const
24 +     {
25 +         HZ_CORE_ASSERT(index < Count);
26 +         return Args[index];
27 +     }
28 + };
29 +
30 + class Application
31 + {
32 + public:
33 +     Application(const std::string& name = "Hazel App");
34 +     Application(const std::string& name = "Hazel App", ApplicationCommandLineArgs args = ApplicationCommandLineArgs());
35 +     ~Application();
36 +
37 +     void OnEvent(Event& e);
38 +
39 +     @ -37,11 +43,14 @ namespace Hazel {
40 +     InDllLayer* GetInDllLayer() { return m_InDllLayer; }
41 +
42 +     static Application& Get() { return *s_Instance; }
43 +
44 +     ApplicationCommandLineArgs GetCommandLineArgs() const { return m_CommandLineArgs; }
45 +
46 + private:
47 +     void Run();
48 +     bool OnWindowClose(WindowCloseEvent& e);
49 +     bool OnWindowResize(WindowResizeEvent& e);
50 +
51 + private:
52 +     ApplicationCommandLineArgs m_CommandLineArgs;
53 +     ScopedWindow m_Window;
54 +     InDllLayer* m_InDllLayer;
55 +     bool m_Running = true;

```

```

Hazel/src/Hazel/Core/EntryPoint.h
1 1 #pragma once
2 2 #include "Hazel/Core/Base.h"
3 3 #include "Hazel/Core/Application.h"
4 4
5 5 #ifdef HZ_PLATFORM_WINDOWS
6 6 - extern Hazel::Application* Hazel::CreateApplication();
7 7 + extern Hazel::Application* Hazel::CreateApplication(ApplicationCommandLineArgs args);
8 8
9 9 int main(int argc, char** argv)
10 10 {
11 11     Hazel::Log::Init();
12 12
13 13     HZ_PROFILE_BEGIN_SESSION("Startup", "HazelProfile-Startup.json");
14 14 - auto app = Hazel::CreateApplication();
15 15 + auto app = Hazel::CreateApplication({ argc, argv });
16 16     HZ_PROFILE_END_SESSION();
17 17
18 18     HZ_PROFILE_BEGIN_SESSION("Runtime", "HazelProfile-Runtime.json");

```

```

Hazelnut/src/EditorLayer.cpp
1 1 @ -33,6 +33,14 @ namespace Hazel {
2 2
3 3     m_ActiveScene = CreateRef<Scene>();
4 4
5 5
6 6 + auto commandLineArgs = Application::Get().GetCommandLineArgs();
7 7 + if (commandLineArgs.Count > 1)
8 8 + {
9 9 +     auto sceneFilePath = commandLineArgs[1];
10 10 +     SceneSerializer serializer(m_ActiveScene);
11 11 +     serializer.Deserialize(sceneFilePath);
12 12 + }
13 13
14 14     m_EditorCamera = EditorCamera(30.0f, 1.778f, 0.1f, 1000.0f);
15 15
16 16 #if 0

```

```

Hazelnut/src/HazelnutApp.cpp
1 1 @ -8,8 +8,8 @ namespace Hazel {
2 2
3 3     class Hazelnut : public Application
4 4     {
5 5     public:
6 6     Hazelnut()
7 7     : Application("Hazelnut")
8 8     {
9 9     Hazelnut(ApplicationCommandLineArgs args)
10 10     : Application("Hazelnut", args)
11 11     {
12 12     PushLayer(new EditorLayer());
13 13     }
14 14
15 15 @ -19,9 +19,9 @ namespace Hazel {
16 16
17 17
18 18
19 19 Application* CreateApplication()
20 20 Application* CreateApplication(ApplicationCommandLineArgs args)
21 21 {
22 22     return new Hazelnut();
23 23     return new Hazelnut(args);
24 24 }
25 25
26 26
27 27 - }
28 28 + }

```

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 @@
1 + #pragma once
2 +
3 + #include "Hazel/Core/Base.h"
4 +
5 + namespace Hazel {
6 +
7 +     class UniformBuffer
8 +     {
9 +     public:
10 +         virtual ~UniformBuffer() {}
11 +         virtual void SetData(const void* data, uint32_t size, uint32_t offset = 0) = 0;
12 +
13 +         static Ref<UniformBuffer> Create(uint32_t size, uint32_t binding);
14 +     };
15 + }
```

```
Hazel/src/Hazel/Renderer/UniformBuffer.cpp
+++ @@ -0,0 +1,21 @@
1 + #include "hzpch.h"
2 + #include "UniformBuffer.h"
3 +
4 + #include "Hazel/Renderer/Renderer.h"
5 + #include "Platform/OpenGL/OpenGLUniformBuffer.h"
6 +
7 + namespace Hazel {
8 +
9 +     Ref<UniformBuffer> UniformBuffer::Create(uint32_t size, uint32_t binding)
10 +     {
11 +         switch (Renderer::GetAPI())
12 +         {
13 +             case RendererAPI::API::None: HZ_CORE_ASSERT(false, "RendererAPI::None is cu
14 +             case RendererAPI::API::OpenGL: return CreateRef<OpenGLUniformBuffer>(size, bi
15 +         }
16 +
17 +         HZ_CORE_ASSERT(false, "Unknown RendererAPI!");
18 +         return nullptr;
19 +     }
20 +
21 + }
```

```
Hazel/src/Platform/OpenGL/OpenGLUniformBuffer.h
+++ @@ -0,0 +1,17 @@
1 + #pragma once
2 +
3 + #include "Hazel/Renderer/UniformBuffer.h"
4 +
5 + namespace Hazel {
6 +
7 +     class OpenGLUniformBuffer : public UniformBuffer
8 +     {
9 +     public:
10 +         OpenGLUniformBuffer(uint32_t size, uint32_t binding);
11 +         virtual ~OpenGLUniformBuffer();
12 +
13 +         virtual void SetData(const void* data, uint32_t size, uint32_t offset = 0) override;
14 +     private:
15 +         uint32_t m_RendererID = 0;
16 +     };
17 + }
```

```
Hazel/src/Platform/OpenGL/OpenGLUniformBuffer.cpp
+++ @@ -0,0 +1,26 @@
1 + #include "hzpch.h"
2 + #include "OpenGLUniformBuffer.h"
3 +
4 + #include <glad/glad.h>
5 +
6 + namespace Hazel {
7 +
8 +     OpenGLUniformBuffer::OpenGLUniformBuffer(uint32_t size, uint32_t binding)
9 +     {
10 +         glCreateBuffers(1, &m_RendererID);
11 +         glNamedBufferData(m_RendererID, size, nullptr, GL_DYNAMIC_DRAW); // TODO: Im
12 +         glBindBufferBase(GL_UNIFORM_BUFFER, binding, m_RendererID);
13 +     }
14 +
15 +     OpenGLUniformBuffer::~OpenGLUniformBuffer()
16 +     {
17 +         glDeleteBuffers(1, &m_RendererID);
18 +     }
19 +
20 +
21 +     void OpenGLUniformBuffer::SetData(const void* data, uint32_t size, uint32_t offset)
22 +     {
23 +         glNamedBufferSubData(m_RendererID, offset, size, data);
24 +     }
25 +
26 + }
```

```
Hazel/src/Hazel/Renderer/Renderer2D.cpp
+++ @@ -0,0 +1,11 @@
1 +
2 +
3 +
4 + #include "Hazel/Renderer/VertexArray.h"
5 + #include "Hazel/Renderer/Shader.h"
6 + #include "Hazel/Renderer/UniformBuffer.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 +
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 +
1001 +
1002 +
1003 +
1004 +
1005 +
1006 +
1007 +
1008 +
1009 +
1010 +
1011 +
1012 +
1013 +
1014 +
1015 +
1016 +
1017 +
1018 +
1019 +
1020 +
1021 +
1022 +
1023 +
1024 +
1025 +
1026 +
1027 +
1028 +
1029 +
1030 +
1031 +
1032 +
1033 +
1034 +
1035 +
1036 +
1037 +
1038 +
1039 +
1040 +
1041 +
1042 +
1043 +
1044 +
1045 +
1046 +
1047 +
1048 +
1049 +
1050 +
1051 +
1052 +
1053 +
1054 +
1055 +
1056 +
1057 +
1058 +
1059 +
1060 +
1061 +
1062 +
1063 +
1064 +
1065 +
1066 +
1067 +
1068 +
1069 +
1070 +
1071 +
1072 +
1073 +
1074 +
1075 +
1076 +
1077 +
1078 +
1079 +
1080 +
1081 +
1082 +
1083 +
1084 +
1085 +
1086 +
1087 +
1088 +
1089 +
1090 +
1091 +
1092 +
1093 +
1094 +
1095 +
1096 +
1097 +
1098 +
1099 +
1100 +
1101 +
1102 +
1103 +
1104 +
1105 +
1106 +
1107 +
1108 +
1109 +
1110 +
1111 +
1112 +
1113 +
1114 +
1115 +
1116 +
1117 +
1118 +
1119 +
1120 +
1121 +
1122 +
1123 +
1124 +
1125 +
1126 +
1127 +
1128 +
1129 +
1130 +
1131 +
1132 +
1133 +
1134 +
1135 +
1136 +
1137 +
1138 +
1139 +
1140 +
1141 +
1142 +
1143 +
1144 +
1145 +
1146 +
1147 +
1148 +
1149 +
1150 +
1151 +
1152 +
1153 +
1154 +
1155 +
1156 +
1157 +
1158 +
1159 +
1160 +
1161 +
1162 +
1163 +
1164 +
1165 +
1166 +
1167 +
1168 +
1169 +
1170 +
1171 +
1172 +
1173 +
1174 +
1175 +
1176 +
1177 +
1178 +
1179 +
1180 +
1181 +
1182 +
1183 +
1184 +
1185 +
1186 +
1187 +
1188 +
1189 +
1190 +
1191 +
1192 +
1193 +
1194 +
1195 +
1196 +
1197 +
1198 +
1199 +
1200 +
1201 +
1202 +
1203 +
1204 +
1205 +
1206 +
1207 +
1208 +
1209 +
1210 +
1211 +
1212 +
1213 +
1214 +
1215 +
1216 +
1217 +
1218 +
1219 +
1220 +
1221 +
1222 +
1223 +
1224 +
1225 +
1226 +
1227 +
1228 +
1229 +
1230 +
1231 +
1232 +
1233 +
1234 +
1235 +
1236 +
1237 +
1238 +
1239 +
1240 +
1241 +
1242 +
1243 +
1244 +
1245 +
1246 +
1247 +
1248 +
1249 +
1250 +
1251 +
1252 +
1253 +
1254 +
1255 +
1256 +
1257 +
1258 +
1259 +
1260 +
1261 +
1262 +
1263 +
1264 +
1265 +
1266 +
1267 +
1268 +
1269 +
1270 +
1271 +
1272 +
1273 +
1274 +
1275 +
1276 +
1277 +
1278 +
1279 +
1280 +
1281 +
1282 +
1283 +
1284 +
1285 +
1286 +
1287 +
1288 +
1289 +
1290 +
1291 +
1292 +
1293 +
1294 +
1295 +
1296 +
1297 +
1298 +
1299 +
1300 +
1301 +
1302 +
1303 +
1304 +
1305 +
1306 +
1307 +
1308 +
1309 +
1310 +
1311 +
1312 +
1313 +
1314 +
1315 +
1316 +
1317 +
1318 +
1319 +
1320 +
1321 +
1322 +
1323 +
1324 +
1325 +
1326 +
1327 +
1328 +
1329 +
1330 +
1331 +
1332 +
1333 +
1334 +
1335 +
1336 +
1337 +
1338 +
1339 +
1340 +
1341 +
1342 +
1343 +
1344 +
1345 +
1346 +
1347 +
1348 +
1349 +
1350 +
1351 +
1352 +
1353 +
1354 +
1355 +
1356 +
1357 +
1358 +
1359 +
1360 +
1361 +
1362 +
1363 +
1364 +
1365 +
1366 +
1367 +
1368 +
1369 +
1370 +
1371 +
1372 +
1373 +
1374 +
1375 +
1376 +
1377 +
1378 +
1379 +
1380 +
1381 +
1382 +
1383 +
1384 +
1385 +
1386 +
1387 +
1388 +
1389 +
1390 +
1391 +
1392 +
1393 +
1394 +
1395 +
1396 +
1397 +
1398 +
1399 +
1400 +
1401 +
1402 +
1403 +
1404 +
1405 +
1406 +
1407 +
1408 +
1409 +
1410 +
1411 +
1412 +
1413 +
1414 +
1415 +
1416 +
1417 +
1418 +
1419 +
1420 +
1421 +
1422 +
1423 +
1424 +
1425 +
1426 +
1427 +
1428 +
1429 +
1430 +
1431 +
1432 +
1433 +
1434 +
1435 +
1436 +
1437 +
1438 +
1439 +
1440 +
1441 +
1442 +
1443 +
1444 +
1445 +
1446 +
1447 +
1448 +
1449 +
1450 +
1451 +
1452 +
1453 +
1454 +
1455 +
1456 +
1457 +
1458 +
1459 +
1460 +
1461 +
1462 +
1463 +
1464 +
1465 +
1466 +
1467 +
1468 +
1469 +
1470 +
1471 +
1472 +
1473 +
1474 +
1475 +
1476 +
1477 +
1478 +
1479 +
1480 +
1481 +
1482 +
1483 +
1484 +
1485 +
1486 +
1487 +
1488 +
1489 +
1490 +
1491 +
1492 +
1493 +
1494 +
1495 +
1496 +
1497 +
1498 +
1499 +
1500 +
1501 +
1502 +
1503 +
1504 +
1505 +
1506 +
1507 +
1508 +
1509 +
1510 +
1511 +
1512 +
1513 +
1514 +
1515 +
1516 +
1517 +
1518 +
1519 +
1520 +
1521 +
1522 +
1523 +
1524 +
1525 +
1526 +
1527 +
1528 +
1529 +
1530 +
1531 +
1532 +
1533 +
1534 +
1535 +
1536 +
1537 +
1538 +
1539 +
1540 +
1541 +
1542 +
1543 +
1544 +
1545 +
1546 +
1547 +
1548 +
1549 +
1550 +
1551 +
1552 +
1553 +
1554 +
1555 +
1556 +
1557 +
1558 +
1559 +
1560 +
1561 +
1562 +
1563 +
1564 +
1565 +
1566 +
1567 +
1568 +
1569 +
1570 +
1571 +
1572 +
1573 +
1574 +
1575 +
1576 +
1577 +
1578 +
1579 +
1580 +
1581 +
1582 +
1583 +
1584 +
1585 +
1586 +
1587 +
1588 +
1589 +
1590 +
1591 +
1592 +
1593 +
1594 +
1595 +
1596 +
1597 +
1598 +
1599 +
1600 +
1601 +
1602 +
1603 +
1604 +
1605 +
1606 +
1607 +
1608 +
1609 +
1610 +
1611 +
1612 +
1613 +
1614 +
1615 +
1616 +
1617 +
1618 +
1619 +
1620 +
1621 +
1622 +
1623 +
1624 +
1625 +
1626 +
1627 +
1628 +
1629 +
1630 +
1631 +
1632 +
1633 +
1634 +
1635 +
1636 +
1637 +
1638 +
1639 +
1640 +
1641 +
1642 +
1643 +
1644 +
1645 +
1646 +
1647 +
1648 +
1649 +
1650 +
1651 +
1652 +
1653 +
1654 +
1655 +
1656 +
1657 +
1658 +
1659 +
1660 +
1661 +
1662 +
1663 +
1664 +
1665 +
1666 +
1667 +
1668 +
1669 +
1670 +
1671 +
1672 +
1673 +
1674 +
1675 +
1676 +
1677 +
1678 +
1679 +
1680 +
1681 +
1682 +
1683 +
1684 +
1685 +
1686 +
1687 +
1688 +
1689 +
1690 +
1691 +
1692 +
1693 +
1694 +
1695 +
1696 +
1697 +
1698 +
1699 +
1700 +
1701 +
1702 +
1703 +
1704 +
1705 +
1706 +
1707 +
1708 +
1709 +
1710 +
1711 +
1712 +
1713 +
1714 +
1715 +
1716 +
1717 +
1718 +
1719 +
1720 +
1721 +
1722 +
1723 +
1724 +
1725 +
1726 +
1727 +
1728 +
1729 +
1730 +
1731 +
1732 +
1733 +
1734 +
1735 +
1736 +
1737 +
1738 +
1739 +
1740 +
1741 +
1742 +
1743 +
1744 +
1745 +
1746 +
1747 +
1748 +
1749 +
1750 +
1751 +
1752 +
1753 +
1754 +
1755 +
1756 +
1757 +
1758 +
1759 +
1760 +
1761 +
1762 +
1763 +
1764 +
1765 +
1766 +
1767 +
1768 +
1769 +
1770 +
1771 +
1772 +
1773 +
1774 +
1775 +
1776 +
1777 +
1778 +
1779 +
1780 +
1781 +
1782 +
1783 +
1784 +
1785 +
1786 +
1787 +
1788 +
1789 +
1790 +
1791 +
1792 +
1793 +
1794 +
1795 +
1796 +
1797 +
1798 +
1799 +
1800 +
1801 +
1802 +
1803 +
1804 +
1805 +
1806 +
1807 +
1808 +
1809 +
1810 +
1811 +
1812 +
1813 +
1814 +
1815 +
1816 +
1817 +
1818 +
1819 +
1820 +
1821 +
1822 +
1823 +
1824 +
1825 +
1826 +
1827 +
1828 +
1829 +
1830 +
1831 +
1832 +
1833 +
1834 +
1835 +
1836 +
1837 +
1838 +
1839 +
1840 +
1841 +
1842 +
1843 +
1844 +
1845 +
1846 +
1847 +
1848 +
1849 +
1850 +
1851 +
1852 +
1853 +
1854 +
1855 +
1856 +
1857 +
1858 +
1859 +
1860 +
1861 +
1862 +
1863 +
1864 +
1865 +
1866 +
1867 +
1868 +
1869 +
1870 +
1871 +
1872 +
1873 +
1874 +
1875 +
1876 +
1877 +
1878 +
1879 +
1880 +
1881 +
1882 +
1883 +
1884 +
1885 +
1886 +
1887 +
1888 +
1889 +
1890 +
1891 +
1892 +
1893 +
1894 +
1895 +
1896 +
1897 +
1898 +
1899 +
1900 +
1901 +
1902 +
1903 +
1904 +
1905 +
1906 +
1907 +
1908 +
1909 +
1910 +
1911 +
1912 +
1913 +
1914 +
1915 +
1916 +
1917 +
1918 +
1919 +
1920 +
1921 +
1922 +
1923 +
1924 +
1925 +
1926 +
1927 +
1928 +
1929 +
1930 +
1931 +
1932 +
1933 +
1934 +
1935 +
1936 +
1937 +
1938 +
1939 +
1940 +
1941 +
1942 +
1943 +
1944 +
1945 +
1946 +
1947 +
1948 +
1949 +
1950 +
1951 +
1952 +
1953 +
1954 +
1955 +
1956 +
1957 +
1958 +
1959 +
1960 +
1961 +
1962 +
1963 +
1964 +
1965 +
1966 +
1967 +
1968 +
1969 +
1970 +
1971 +
1972 +
1973 +
1974 +
1975 +
1976 +
1977 +
1978 +
1979 +
1980 +
1981 +
1982 +
1983 +
1984 +
1985 +
1986 +
1987 +
1988 +
1989 +
1990 +
1991 +
1992 +
1993 +
1994 +
1995 +
1996 +
1997 +
1998 +
1999 +
2000 +
2001 +
2002 +
2003 +
2004 +
2005 +
2006 +
2007 +
2008 +
2009 +
2010 +
2011 +
2012 +
2013 +
2014 +
2015 +
2016 +
2017 +
2018 +
2019 +
2020 +
2021 +
2022 +
2023 +
2024 +
2025 +
2026 +
2027 +
2028 +
2029 +
2030 +
2031 +
2032 +
2033 +
2034 +
2035 +
2036 +
2037 +
2038 +
2039 +
2040 +
2041 +
2042 +
2043 +
2044 +
2045 +
2046 +
2047 +
2048 +
2049 +
2050 +
2051 +

```

```

HazelInput/assets/shaders/Texture.glsl
*** @ -1,7 +1,7 @@
1 1 // Basic Texture Shader
2 2
3 3 #type vertex
4 4 - #version 450
4 4 + #version 450 core
5 5
6 6 layout(location = 0) in vec3 a_Position;
7 7 layout(location = 1) in vec4 a_Color;
8 8
9 9 @ -10,76 +10,90 @@ layout(location = 3) in float a_TexIndex;
10 10 layout(location = 4) in float a_TilingFactor;
11 11 layout(location = 5) in int a_EntityID;
12 12
13 13 - uniform mat4 u_ViewProjection;
13 13 + layout(std140, binding = 0) uniform Camera
14 14 + {
15 15 +     mat4 u_ViewProjection;
16 16 + };
17 17
18 18 + struct VertexOutput
19 19 + {
20 20 +     vec4 Color;
21 21 +     vec2 TexCoord;
22 22 +     float TexIndex;
23 23 +     float TilingFactor;
24 24 + };

```

5 着色器系统更新:
(New shader system)

Timer 的定义

```

Hazel/src/Hazel/Core/Timer.h
*** @ -2,0 +1,34 @@
1 1 + #pragma once
2 2 +
3 3 + #include <chrono>
4 4 +
5 5 + namespace Hazel {
6 6 +
7 7 +     class Timer
8 8 +     {
9 9 +     public:
10 10 +         Timer()
11 11 +         {
12 12 +             Reset();
13 13 +         }
14 14 +
15 15 +         void Timer::Reset()
16 16 +         {
17 17 +             m_Start = std::chrono::high_resolution_clock::now();
18 18 +         }
19 19 +
20 20 +         float Timer::Elapsed()
21 21 +         {
22 22 +             return std::chrono::duration_cast<std::chrono::nanoseconds>
23 23 +                 (std::chrono::high_resolution_clock::now() - m_Start).count() * 0.001f;
24 24 +         }
25 25 +
26 26 +         float Timer::ElapsedMillis()
27 27 +         {
28 28 +             return Elapsed() * 1000.0f;
29 29 +         }
30 30 +     private:

```

着色器更新

```

Hazel/src/Platform/OpenGL/OpenGLShader.cpp
1 1 @ -6,26 +6,104 @@
2 2
3 3 #include <glm/gtc/type_ptr.hpp>
4 4
5 5 + #include <shaders/shaders.hpp>
6 6 + #include <spirv_cross/spirv_cross.hpp>
7 7 + #include <spirv_cross/spirv_gsl.hpp>
8 8 +
9 9 + #include "Hazel/Core/Timer.h"
10 10 +
11 11 + namespace Hazel {
12 12 +
13 13 +     static GLenum ShaderTypeFromString(const std::string& type)
14 14 +     {
15 15 +         if (type == "vertex")
16 16 +             return GL_VERTEX_SHADER;
17 17 +         if (type == "fragment" || type == "pixel")
18 18 +             return GL_FRAGMENT_SHADER;
19 19 +
20 20 +         namespace Utils {
21 21 +
22 22 +             static GLenum ShaderTypeFromString(const std::string& type)
23 23 +             {
24 24 +                 if (type == "vertex")
25 25 +                     return GL_VERTEX_SHADER;
26 26 +                 if (type == "fragment" || type == "pixel")
27 27 +                     return GL_FRAGMENT_SHADER;
28 28 +
29 29 +                 HZ_CORE_ASSERT(false, "Unknown shader type!");
30 30 +                 return 0;
31 31 +             }
32 32 +         }
33 33 +     }
34 34 + }

```

```

Hazel/src/Platform/OpenGL/OpenGLShader.h
1 1 @ -41,18 +41,28 @@ namespace Hazel {
2 2
3 3     private:
4 4     std::string ReadFile(const std::string& filePath);
5 5     std::unordered_map<GLenum, std::string> Preprocess(const std::string& source);
6 6
7 7     void Compile(const std::unordered_map<GLenum, std::string&> shaderSources);
8 8
9 9     void CompileGetVulkanShader(const std::unordered_map<GLenum, std::string&>
10 10     shaderSources);
11 11     void CreateProgram();
12 12     void Reflect(GLenum stage, const std::vector<int32_t, 16> shaderData);
13 13
14 14     private:
15 15     uint32_t m_RendererID;
16 16     std::string m_FilePath;
17 17     std::string m_Name;
18 18
19 19     std::unordered_map<GLenum, std::vector<int32_t>> m_VulkanSPIRV;
20 20     std::unordered_map<GLenum, std::vector<int32_t>> m_OpenGLSPIRV;
21 21
22 22     std::unordered_map<GLenum, std::string> m_OpenGLSourceCode;
23 23
24 24 };
25 25
26 26 }

```

6 平台工具的更新（打开或保存文件）

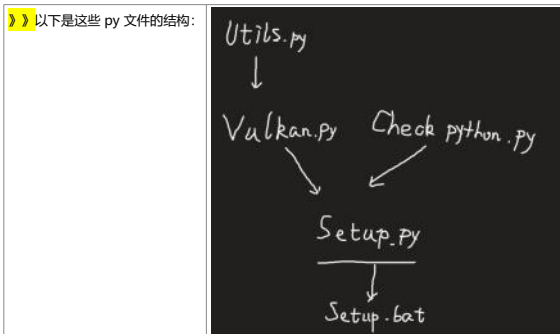
[illegible]

7 视口与摄像机更新:

```
// Hazel/src/Main/Scene/Scene.cpp
+ // -S42,-I S42,I @H namespace Hazel {
142 template<
143   void Scene::RenderComponent(ArchedCameraComponent *Entity entity, CameraComponent & component)
144 {
145     - component.Camera.SetViewportSize(m_VinportWidth, m_VinportHeight);
146     + if (m_VinportWidth > 0 && m_VinportHeight > 0)
147         + component.Camera.SetViewportSize(m_VinportWidth, m_VinportHeight);
148 }
149 template<
+
// Hazel/src/Main/Scene/CrossSection.cpp
+ // -M,S -M,T @H namespace Hazel {
10
11
12 void SceneCamera::SetViewportSize(int32_t width, int32_t height)
13 {
14     + HZ_CORE_ASSERT(width > 0 && height > 0);
15     + m_AspectRatio = (float)width / (float)height;
16     + RecalculateProjection();
17 }
```

》》》》我将逐次的提交这些代码，并记录自己的疑虑
 》》》一：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,)



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。-->

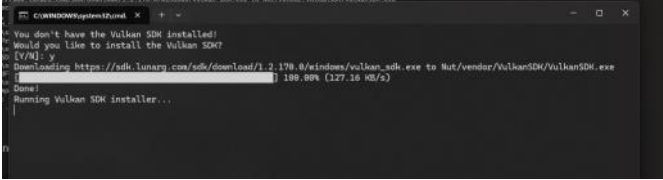
```
[V\N]: y
Downloading https://sdk.lunarg.com/sdk/download/1.2.170.0/windows/vulkan_sdk.exe to Nut/vendor/VulkanSDK/VulkanSDK.exe
Traceback (most recent call last):
  File "Setup.py", line 13, in <module>
    if (not Vulkan.CheckVulkanSDK()):
  File "E:\VS\Nut\scripts\Vulkan.py", line 36, in CheckVulkanSDK
    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, 'ab') as f:
FileNotFoundError: [Error 2] No such file or directory: 'Nut/vendor/VulkanSDK/VulkanSDK.exe'
请按任意键继续. . .
```

spdiog	2024/11/11 17:38	文件夹
stb_image	2024/11/11 17:38	文件夹
VulkanSDK	2024/11/27 20:43	文件夹
yaml-cpp	2024/11/13 14:45	文件夹

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，还有你想下载到本地的路径)

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。

```
10 VULKAN_SDK = os.environ.get('VULKAN_SDK')
11 VULKAN_SDK_INSTALLER_URL = 'https://sdk.lunarg.com/sdk/download/1.2.170.0/windows/vulkan_sdk.exe'
12 NUT_VULKAN_VERSION = '1.2.170.0'
13 VULKAN_SDK_EXE_PATH = 'Nut/vendor/VulkanSDK/VulkanSDK.exe'
14
15
16 ~ def InstallVulkanSDK():
17     print('downloading {} to {}'.format(VULKAN_SDK_INSTALLER_URL, VULKAN_SDK_EXE_PATH))
18     utils.DownloadFile(VULKAN_SDK_INSTALLER_URL, VULKAN_SDK_EXE_PATH)
19     print('done!')
20     print('Running Vulkan SDK installer...')
21     os.startfile(os.path.abspath(VULKAN_SDK_EXE_PATH))
22     print('do not run this script after installation')
```

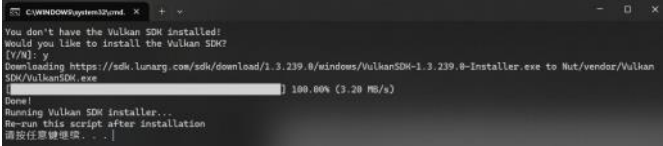
附录：如果你想进入官网查看适合你系统的 SDK，以下是网址 ->
(<https://vulkan.lunarg.com/sdk/home>)



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。

```
48 VulkanSDKDebugLibsURL = 'https://files.lunarg.com/SDK/1.2.170.0/VulkanSDK-1.2.170.0-DebugLibs.zip'
49 OutputDirectory = 'Nut/vendor/VulkanSDK'
50 TempZipFile = f'{OutputDirectory}/VulkanSDK.zip'
51
52
53 def CheckVulkanSDKDebugLibs():
54     shadercdLib = Path(f'{OutputDirectory}/Lib/shaderc_shared.lib')
55     if (not shadercdLib.exists()):
56         print(f'No Vulkan SDK debug libs found. (checked {shadercdLib})')
57         print('Downloading', VulkanSDKDebugLibsURL)
58         with urlopen(VulkanSDKDebugLibsURL) as zipresp:
59             with ZipFile(BytesIO(zipresp.read())) as zfile:
60                 zfile.extractall(OutputDirectory)
61         print(f'Vulkan SDK debug libs located at {OutputDirectory}')
62         return True
```

我们先重新运行一遍，使用更新之后的 SDK install，



The screenshot shows the Vulkan SDK 1.2.203.1 installation window. The 'Optional' tab is selected, displaying a list of components to be installed. The 'Shader Toolchain Debug Symbols - 64-bit' checkbox is checked. The 'Install' button is highlighted. The window also shows the 'Required' tab with 'The Vulkan SDK Core (Always installed)' checked. The 'Install' button is highlighted. The window also shows the 'Optional' tab with the following components: 'The Vulkan SDK Core (Always installed)', 'SDK 22-bit Core Components', 'Shader Toolchain Debug Symbols - 64-bit' (checked), 'Shader Toolchain Debug Symbols - 32-bit', 'GEM headers', 'SQL2 libraries and headers', 'Vulkan headers, source, and library', and 'Vulkan headers, source, and library'.

名称	修改日期	类型	大小
Bin	2024/11/27 21:54	文件夹	
Config	2024/11/27 21:54	文件夹	
Demos	2024/11/27 21:54	文件夹	
Helpers	2024/11/27 21:54	文件夹	
Include	2024/11/27 21:54	文件夹	
InstallerResources	2024/11/27 21:54	文件夹	
Lib	2024/11/27 21:54	文件夹	
Libraries	2024/11/27 21:54	文件夹	
Share	2024/11/27 21:54	文件夹	
Source	2024/11/27 21:54	文件夹	
Templates	2024/11/27 21:54	文件夹	
Components.xml	2024/11/27 21:54	XML 文件	2 KB
InstallationLog.txt		文本文件	20 KB
Installer.dat	2024/11/27 21:54	DAT 文件	1 KB
msiexecmconsole.dat	2024/11/27 21:54	DAT 文件	45 KB
msiexecmconsole.exe	2024/11/27 21:54	可执行文件	3.6 MB

```

48 VulkanSDKDebugLibsURL = "https://files.lunarg.com/SDK-1.2.170.0/vulkanSDK-1.2.170.0-DebugLibs.zip"
49 OutputDirectory = "Out/vendor/VulkanSDK"
50 TempZipFile = f"{OutputDirectory}\\VulkanSDK.zip"
51
52
53
54 def CheckVulkanSDKDebugLibs():
55     shadercLib = Path(f"{OutputDirectory}\\Lib\\shaderc_shared.lib")
56     if (not shadercLib.exists()):
57         print("No Vulkan SDK debug libs found. Check {shadercLib}")
58         print("Downloading", VulkanSDKDebugLibsURL)
59         with urlopen(VulkanSDKDebugLibsURL) as zipresp:
60             with ZipFile(BytesIO(zipresp.read())) as zfile:
61                 zfile.extractall(OutputDirectory)
62         print(f"Vulkan SDK debug libs located at {OutputDirectory}")
63     return True

```

where can i get the debug libs? i download VulkanSDK.exe, it ran and downloaded Vulkan but i don't know where can i find the debug libs since i am doing this for vulkan 1.3.250.0

in the later version of Vulkan there is no debug libs compressed file to download from sdk.lunarg.com so you are gonna have to manually copy the debug libs from your Vulkan installation directory ("C:\VulkanSDK\1.3.250.0\" to "Hazel/vendor/VulkanSDK/" or set up the script to do it for you.

```

VulkanSDKDebugLibURL = "https://files.lunarg.com/SDK-1.2.170.0/vulkansdk-1.2.170.0"
OutputDirectory = "nvt/vendor/VulkanSDK"
TempZipFile = f"{OutputDirectory}/VulkanSDK.zip"

def CheckVulkanSDKDebugLibs():
    shaderLib = path(f"{OutputDirectory}/Lib/shaderc_shared.lib")
    if (os.path.exists(shaderLib)):
        print("No Vulkan SDK debug libs found. (checked {shadercLib})")
        print("Downloading", VulkanSDKDebugLibURL)
        with urlopen(VulkanSDKDebugLibURL) as zipresp:
            with ZipFile(BytesIO(zipresp.read())) as zipfile:
                zipfile.extractall(OutputDirectory)
    print(f"Vulkan SDK debug libs located at {OutputDirectory}")
    return True

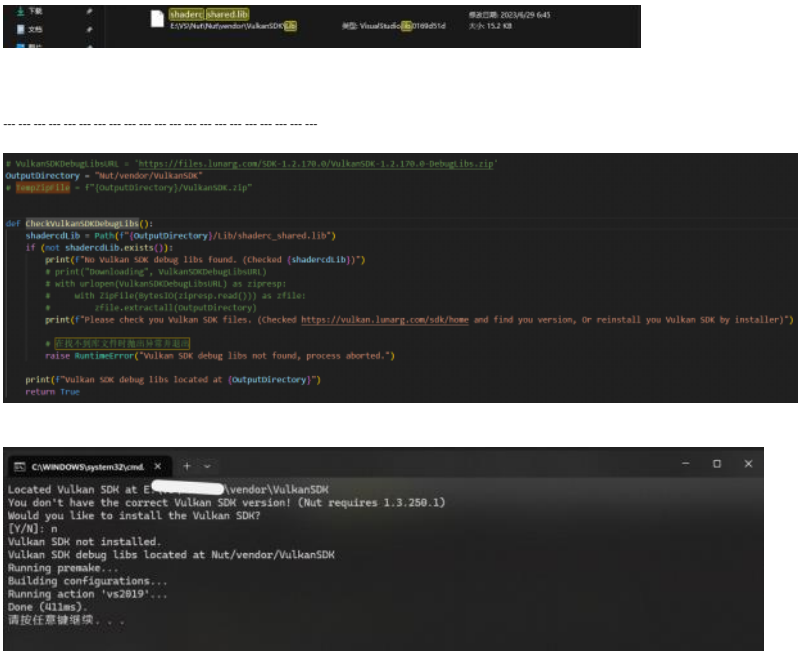
```

A screenshot of a Windows File Explorer window. The address bar shows the path 'lib' under the 'lib' folder. The file 'shadow_shared.lib' is selected and highlighted in blue. The toolbar includes options like '新建' (New), '删除' (Delete), '重命名' (Rename), '查看' (View), '搜索' (Search), '安全' (Security), '分享' (Share), '设置' (Settings), '更多' (More), and '详细信息' (Details). The status bar at the bottom shows '文件' (File), '编辑' (Edit), '查看' (View), and '工具' (Tools).

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。

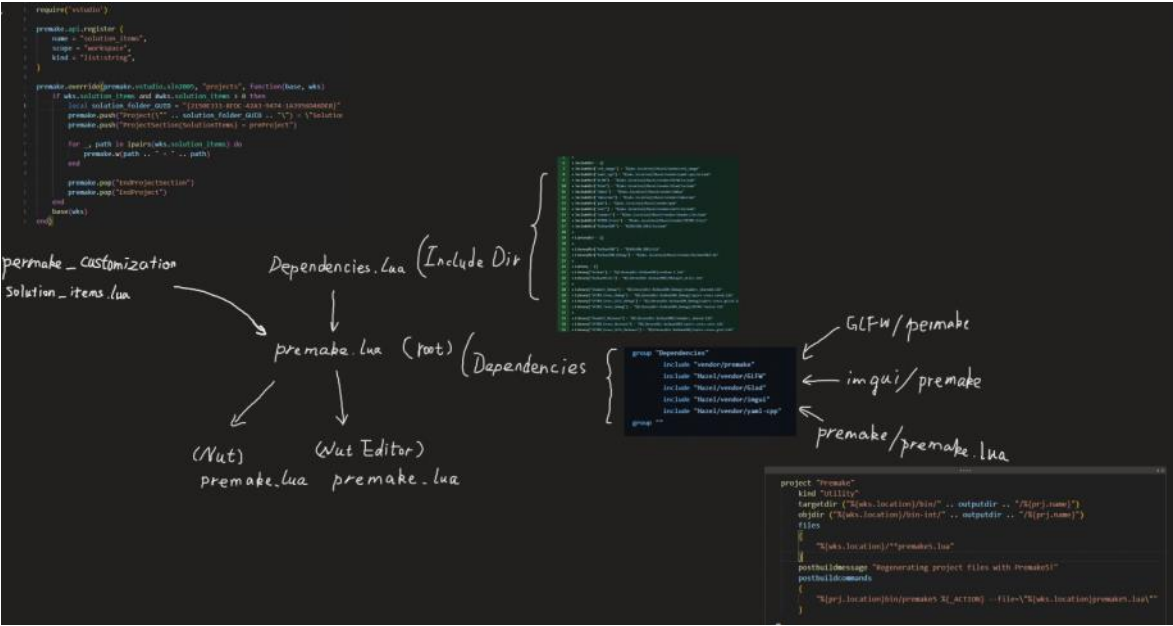
我便做了以下更改：（仅仅是口头提醒一下~）

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



我想应该是对的。

》》》二：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）

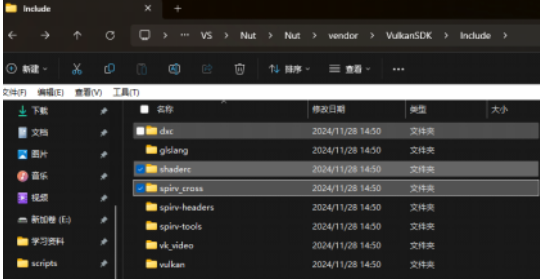


》》》接下来我先更新 Premake Dependencies.lua 文件（这里为预处理，实际操作步骤在后面）。

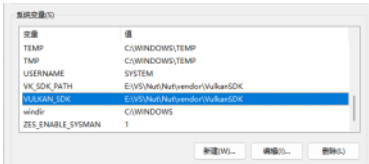
第一步，我们在项目的根目录下重新编写一个 premake 文件，这个文件主要用来索引 vendor 中的外部库（API）

```
1 --! Nut Dependencies
2
3 VULKAN_SDK = os.getenv("VULKAN_SDK")
4
5 IncludeDir = {}
6 IncludeDir["stb_image"] = "%{wks.location}/Nut/vendor/stb_image"
7 IncludeDir["yaml_cpp"] = "%{wks.location}/Nut/vendor/yaml-cpp/include"
8 IncludeDir["GLFW"] = "%{wks.location}/Nut/vendor/GLFW/include"
9 IncludeDir["Glad"] = "%{wks.location}/Nut/vendor/Glad/include"
10 IncludeDir["ImGui"] = "%{wks.location}/Nut/vendor/ImGui"
11 IncludeDir["ImGuiZmo"] = "%{wks.location}/Nut/vendor/ImGuiZmo"
12 IncludeDir["glm"] = "%{wks.location}/Nut/vendor/glm"
13 IncludeDir["entt"] = "%{wks.location}/Nut/vendor/entt/include"
14 IncludeDir["shaderc"] = "%{wks.location}/Nut/vendor/shaderc/include"
15 IncludeDir["SPIRV_cross"] = "%{wks.location}/Nut/vendor/SPIRV-cross"
16 IncludeDir["VulkanSDK"] = "%{VULKAN_SDK}/include"
17
18 LibraryDir = {}
19
20 LibraryDir["VulkanSDK"] = "%{VULKAN_SDK}/lib"
21 LibraryDir["VulkanSDK_Debug"] = "%{wks.location}/Nut/vendor/VulkanSDK/Lib"
22
23 Library = {}
24
25 Library["Vulkan"] = "%{LibraryDir.VulkanSDK}/vulkan-1.lib"
26 Library["VulkanUtils"] = "%{LibraryDir.VulkanSDK}/VkLayer_utils.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_GLSL_Debug"] = "%{LibraryDir.VulkanSDK_Debug}/spirv-cross-gslslib"
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_GLSL_Release"] = "%{LibraryDir.VulkanSDK}/spirv-cross-gslslib"
```

但我发现有些问题，比如 shaderc 和 spirv_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



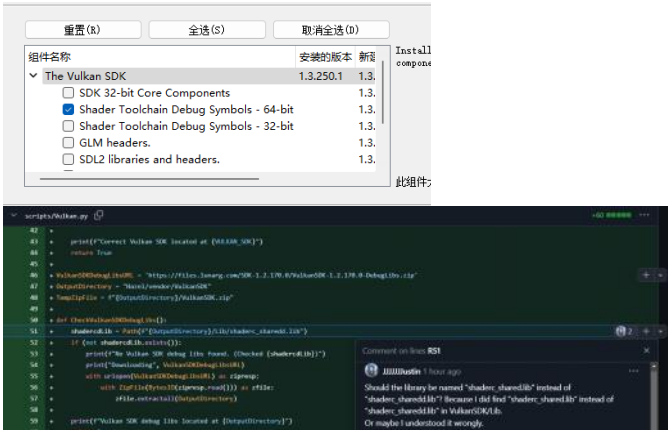
系统变量示例：

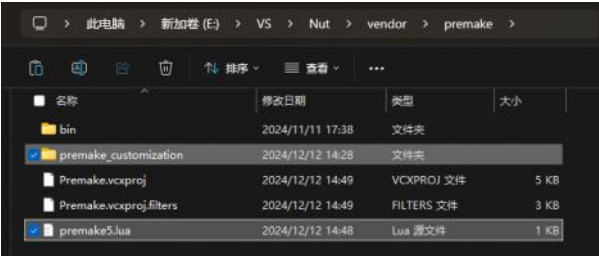


而且由于我没有下载某些组件，这使很多文件并不存在。（我将其标注出来）

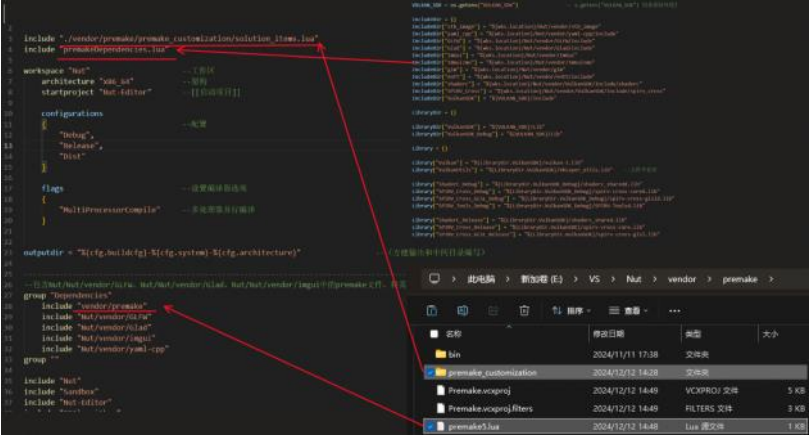
```
1 --! Nut Dependencies
2
3 VULKAN_SDK = os.getenv("VULKAN_SDK")
4
5 IncludeDir = {}
6 IncludeDir["stb_image"] = "%{wks.location}/Nut/vendor/stb_image"
7 IncludeDir["yaml_cpp"] = "%{wks.location}/Nut/vendor/yaml-cpp/include"
8 IncludeDir["GLFW"] = "%{wks.location}/Nut/vendor/GLFW/include"
9 IncludeDir["Glad"] = "%{wks.location}/Nut/vendor/Glad/include"
10 IncludeDir["ImGui"] = "%{wks.location}/Nut/vendor/ImGui"
11 IncludeDir["ImGuiZmo"] = "%{wks.location}/Nut/vendor/ImGuiZmo"
12 IncludeDir["glm"] = "%{wks.location}/Nut/vendor/glm"
13 IncludeDir["entt"] = "%{wks.location}/Nut/vendor/entt/include"
14 IncludeDir["shaderc"] = "%{wks.location}/Nut/vendor/shaderc/include"
15 IncludeDir["SPIRV_cross"] = "%{wks.location}/Nut/vendor/SPIRV-cross"
16 IncludeDir["VulkanSDK"] = "%{VULKAN_SDK}/include"
17
18 LibraryDir = {}
19
20 LibraryDir["VulkanSDK"] = "%{VULKAN_SDK}/lib"
21 LibraryDir["VulkanSDK_Debug"] = "%{wks.location}/Nut/vendor/VulkanSDK/Lib"
22
23 Library = {}
24
25 Library["Vulkan"] = "%{LibraryDir.VulkanSDK}/vulkan-1.lib"
26 Library["VulkanUtils"] = "%{LibraryDir.VulkanSDK}/VkLayer_utils.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_GLSL_Debug"] = "%{LibraryDir.VulkanSDK_Debug}/spirv-cross-gslslib"
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_GLSL_Release"] = "%{LibraryDir.VulkanSDK}/spirv-cross-gslslib"
```

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：





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



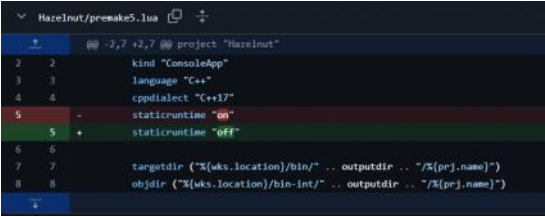
》》》444 修改 Nut/Nut/premake5.lua 和 Nut/Nut-Editor/premake5.lua 文件内容

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。

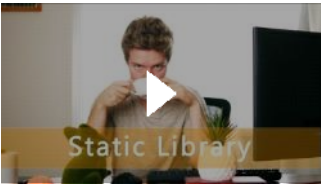
》》》问题：

在此处我遇到一个问题，就是 Chernobyl 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 Chernobyl 没有说明要转回使用动态库的方式，所以现在我没有将其打开。
(在之后的提交中，我修改掉了这里的代码，可以查看：》》》》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：)
(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 Chernobyl 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/Haze/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。

》》》三：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：

```
#ifndef 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_TRACE("Argument (0): {i}", 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。

<p>管线流程：</p> <pre>Application* CreateApplication(ApplicationCommandLineArgs args) { return new NutEditor(args); }</pre>	<p>这里是 <code>CreateApplication()</code> 的定义。</p> <p><code>CreateApplication()</code> 中使用了 <code>NutEditor()</code></p>
<pre>class NutEditor : public Application { public: NutEditor(ApplicationCommandLineArgs args) : Application("Nut Editor", args) { PushLayer(new EditorLayer()); NutEditor() } };</pre>	<p>这里是 <code>NutEditor()</code> 的定义。</p> <p><code>NutEditor()</code> 是 <code>Application()</code> 的子类，故 <code>NutEditor()</code> 的构造函数会自动先使用父类 <code>Application()</code> 的构造函数，我们可以通过这个特性将 <code>args</code> 参数传给 <code>Application()</code> 的构造函数，并实现一些目的。</p>
<pre>class Application { public: Application(const std::string& name = "Nut App", ApplicationCommandLineArgs = ApplicationCommandLineArgs()) : virtual _Application() { void OnEvent(Event& e); //事件分发 void PushLayer(Layer* layer); void PushOverlay(Layer* overlay); inline Window& GetWindow() { return *_window; } //返回下面这个指向Window的指针 inline InputLayer& GetInputLayer() { return *_inputLayer; } inline static Application& Get() { return *_instance; } // !!! 返回的是 *_instance 这个指向 Application 的指 // (为什么返回是引用体呢？因为Application是一个单例 inline ApplicationCommandLineArgs GetCommandLineArgs() const { return *_commandLineArgs; } auto sceneFilePath0 = commandLineArgs[0]; NUT_CORE_WARN(sceneFilePath0); } };</pre>	<p>这是父类 <code>Application()</code> 构造函数的新定义。</p> <p>同时我们新添了一个 <code>GetCommandLineArgs()</code> 的函数，用于获取私有变量 <code>m_CommandLineArgs</code> 中存放的数据。</p> <pre>[15:23:15] NUT: version: 0.0.0 - Build 31 0 101 0000 [15:23:15] NUT: E:\VS\Nut\bin\Debug-windows-x86_64\Nut-Editor\Nut-Editor.exe [15:23:15] NUT: Pixel data: -1</pre>

我们可以在运行时查看argv获取到的信息是什么。

》》》知识

》》》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，当你使用指令运行一个程序（`./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？)	<p>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。</p> <p>程序中 <code>argc</code> 和 <code>argv</code> 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。</p> <p>比如：</p> <ul style="list-style-type: none">• 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	<p>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</p>

(内容什么时候被确定? 是否可以被随时改变?)	变。 所以一旦程序开始执行, <code>argc</code> 和 <code>argv</code> 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, {FrameBufferFormat::RGBA8} });

    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, {FrameBufferFormat::RGBA8} });

    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);
}
```

得到这样的结果:

```
14:10:51.111 NUT: Application::Log
14:10:51.111 NUT: Command line args:
14:10:51.111 NUT: Argument 0: E:\VS\Nut\bin\Debug-windows-x86_64\Nut-Editor\Nut-Editor.exe
14:10:51.111 NUT: Creating window: Nut-Editor [1280, 720]
14:10:51.111 NUT: Initializing GLFW window.
14:10:51.111 NUT: Done.
14:10:51.111 NUT: Version: NUT2.0 Corporation
14:10:51.111 NUT: Hardware: NVIDIA GeForce RTX 3050 Laptop GPU/PCIe/SSE3
14:10:51.111 NUT: Version: 4.4.0 NUT2.0.0.1
14:10:51.111 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)	(新)
一个黄褐色头发的男人, 他打开了 <code>cmd</code> , 想要运行 <code>Nut-Editor</code> 应用, 于是他输入了	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:

一句指令:

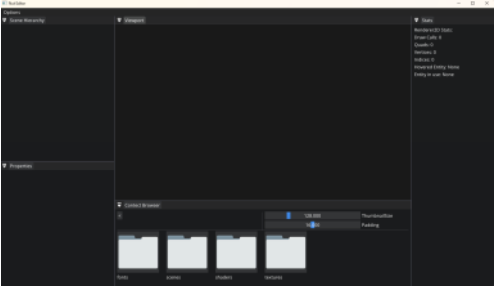


argc	1
argv[0]	"E:\VS\Nut\bin\Debug-windows-x86_64\Nut-Editor\Nut-Editor.exe"

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:

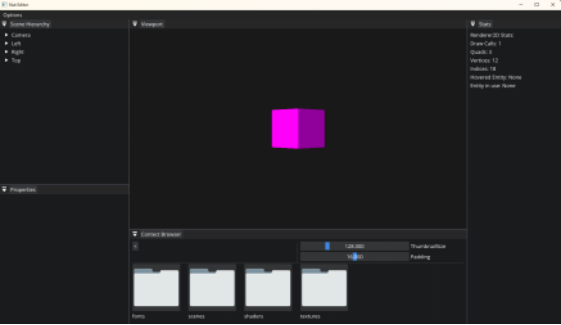
```
auto commandLineArgs = Application::Get().GetCommandLineArgs();
if (commandLineArgs.Count > 1)
{
    auto sceneFilePath = commandLineArgs[1];
    SceneSerializer serializer(m_ActiveScene);
    serializer.Deserialize(sceneFilePath);
}

auto sceneFilePath0 = commandLineArgs[0];
```



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"

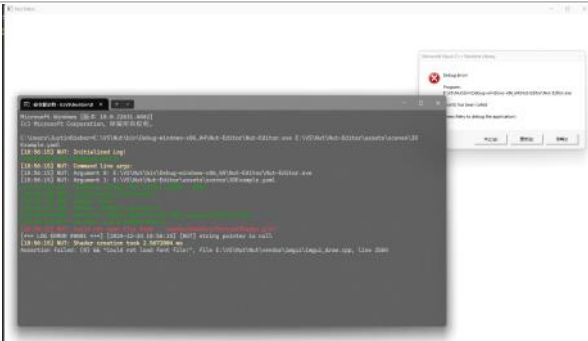
此时运行指令, 你将会在启动时看到一个预先加载的场景。



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)

```
100 // QuadVertex Ptr
101 s_Data.QuadVertexBuffer = new QuadVertex[s_Data.MaxVertices]; // 保存顶点初始值
102
103 // Shader
104 s_Data.TextureShader = Shader::Create("E:\VS\Nut\Nut-Editor\assets\shaders\TextureShader.glsl");
105
106 // UBO
107 s_Data.CameraUniformBuffer = UniformBuffer::Create(sizeof(Renderer2DData::CameraData), 0);
108
109 // 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。

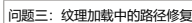
```
ImFontAtlas::AddFontFromFileTTF(const char* filename, float size_pixels, const ImFontConfig* font_cfg_template, const ImWchar* glyph_ranges)
{
    IM_ASSERT(!locked && "cannot modify a locked ImFontAtlas between NewFrame() and EndFrame/Render()");
    size_t data_size = 0;
    void* data = ImFileLoadToMemory(filename, "rb", &data_size, 0);
    if (!data)
    {
        IM_ASSERT(size_pixels > 0);
        return NULL;
    }
    ImFontConfig font_cfg = font_cfg_template ? *font_cfg_template : ImFontConfig();
    if (font_cfg.Name == NULL)
    {
        // Store a short copy of filename 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:



(ContentBrowserPanel.cpp)



现在，便能够通过终端输入：“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。



或者需要手动删除一些缓存文件（例如着色器缓存文件，OpenGLShader的更新中会涉及到），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。



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。

《《《《 四：添加Uniform Buffer

》关于 Uniform Buffer 的定义：具体可以查看（<https://learnopengl.com/Advanced-OpenGL/Uniform-Buffer-Objects>）

建议浏览该页面之后，再查看更新的代码。

Uniform缓冲对象

我们已经使用OpenGL很长时间了，学会了一些很酷的练习。您也遇到了一些麻烦的地方，比如，特使用多于一个的着色器时，对每个着色器的uniform变量都是相同的。我们还需要不断地设置它们。所以为什么要这么麻烦地重复设置它们呢？

OpenGL为我们提供了一个叫做(Uniform Buffer Object) (Uniform Buffer Object)的工具。它允许我们定义一系列在多个着色器程序中使用的全局uniform变量。使用此uniform缓冲区对象时，我们不需要重复设置每个uniform一次。因此，我们仍需要掌握一些关于着色器变量以及Uniform Buffer Object，特别是使用新的Uniform Buffer Object的一些知识。

因为UniformBlock中的成员是一个值，我们可以使用`UniformBlock`来构造它。我们将添加`ss_uniformBlock`成员函数，并将所有相关的Uniform数据存入该值。在UniformBlock中我们存储数据是有一些限制的。我们不会在这里讨论它。首先，我们将使用一个值来保存数据。将`projectionView`成员存储到新的`UniformBlock`（Uniform Block）中：

```

//compute HSB score
Input: function * w() {a vec3(0);

Input: (subDD) uniform matrices
{
    mat3 projection;
    mat3 view;
}

uniform mat3 model;

void main()
{
    gl_Position = projection * view * model * vec4(pos, 1.0);
}

```

在它们二者更多的例子中，我们经常会看到类似的情况，为两个类命名使用 `registerView` 和 `viewUniformBlock`。这看起来像 `UniformBlock` 的命名规范的一个变种的实现，因为如果我们不遵守这个命名规范的话，这将会很令人困惑。

这里，我们使用了一个叫做 `Attribute` 的类，它包含了一个 `4x4x4x4` 的 `float` 的数组。我们使用这个变量可以直接使用，不需要做任何操作。接下来，我们在 `OpenGL` 代码中将这些数据输入到串口中，每个实例用了这个 `UniformBlock` 的命名规范来命名这些变量。

你或许可能想知道 `layout (uniform_block)` 这个语句的含义。这是告诉编译器，应该从父类 `UniformBlock` 的变量中选择一个特殊的成员变量。这个成员变量是 `UniformBlock` 的 `UniformBlock` 成员。

Uniform快在层

使用Uniform缓冲

我们只讨论过了如何从数据集中取出10个样本，并从中取出5个内容来画了，下面我们讨论一下如何从这10个样本中取出5个。

首先，我们需要调用 `glGenBuffers`，创建一个 `Vertex` 缓冲对象。一旦我们有了一个缓冲对象，我们需要对它进行命名。我们使用 `GLuint` 变量来存储这个名称，并使用 `glBufferData` 分配内存的内容。

[illegible]

现在，每当我们想要对缓冲区的数据进行读取，我们就会调用`lockSampleBlock`，并使用`glTexBufferSubData`来更新内存，我们只需要更新这个Uniform值一次，所有使用这个缓冲区的着色器都使用的是更新的数据了。但是，如代码清单4.10所示，我们还需要对每个着色器做点事情，如代码清单4.11所示。

在OpenGL上下文中，定义了一些着色点(Binding Point)。我们可以将一个Uniform值绑定到着色点。在着色Uniform值时，我们将它绑定到着色中一个着色点上，并将着色器中的Uniform值绑定到相同的着色点。把它们连接到一起。下面的图表展示了这个：



Uniform缓冲对象

我们还在使用OpenGL成长时期了，学会了一些很酷的技巧，但也遇到了一些很麻烦的地方，比如，使用多于一个的着色器时，尽管大部分的uniform变量都是相同的，我们还是需要不断地设置它们。所以为什么要这么麻烦地重复设置它们呢？

OpenGL已经向我们提供了一个叫做Uniform Buffer Object的Uniform Buffer Object的工具，它允许我们定义一系列在多个着色器程序中相同的着色器uniform，也允许我们在Uniform Buffer Object中设置它们。我们不需要设置相同的uniform一次，当然，我们仍需手动设置每个着色器中不同的uniform，并让每个着色器中的Uniform Buffer Object对象包含一些数据。

因为Uniform缓冲块仍是一个缓冲块，我们可以使用`glBufferData`来创建它。我们添加如下的`GLuintIFORM_BUFFER`缓冲块，并给每个顶点表的`GLuintIFORM_BUFFER`缓冲块添加一些数据。我们稍等之后讨论它。首先，我们使用一个缓冲块的顶点数据。而`projectionOverload`缓冲块存储在`GLuintIFORM_BUFFER`缓冲块中：

```

@Override void core()
{
    InputGenerator = g; // in each offset;

    Input = (short[]) uniformMatrices;

    {
        mat1 = projMatrices;
        mat2 = view;
    }

    orTransform mat1 model;

    void main()
    {
        gl_Position = projMatrices * view * model * vec4(offset, 1.0);
    }
}

```

在我们大部分的例子中，我们都会在每个渲染迭代中，为每个着色器设置projection和view Uniform数据。这是利用Uniform表中定义的一个非常完美的例子，因为现在我们只需要存储这些矩阵一次就可以了。

这里，我们声明了一个叫做`matrices`的Uniform块，它包含了两个4x4矩阵。Uniform块中的变量可以直接访问，不需要加块名作为前缀。接下来，我们在OpenGL代码中将这些矩阵传入渲染中。每个声明了这个Uniform块的着色器都能访问这些矩阵。

你现在可能在想 `layout (out)` 这个语句是什么意思。它的意思是说，当GLSL的Uniform块对它的内容使用一个特定的内存布局。这个语句设置了**Uniform块布局**(Uniform Block Layout)。

Uniform块布局

Use和env块的内容是储存在一个虚拟对象中的，它实际上只是一块暂留内存，因为这块内存并不会保存它具体保存的是什么数据。

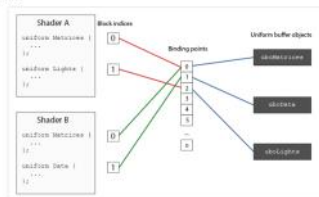
etc....

GL_UNIFORM_BUFFER的标, 并调用glBufferData, 分配目标内存。

```
unsigned int virtcomp(eBlock)
{
    g(buffers[1], virtcomp(eBlock));
    g(buffers[0], INTPTR_BUFFER, virtcomp(eBlock));
    g(buffers[0], INTPTR_BUFFER, 10, MALL, 0, STATDC, 0); // 内存管理统计
}
```

现在，每当我们需要为着色器或者输入数据，我们都会使用到`glUniformBlockPointer`，并使用`glUniformBlockIndex`来更新它的内存。我们只需要使用这个Uniform块一次，然后使用这个块+的着色器就使用它是更新后的数据了。但是，如何能让OpenGL知道哪个Uniform块对应的是哪个Uniform块呢？

在OpenGL上下文中，定义了一条[着色点（binding point）](#)，我们可以将一个Uniform绑定到着色点。在创建Uniform之前，我们可以把它绑定到着色点中的一个着色点上，并将着色点中的Uniform绑定到着色点的着色点，把它们连接在一起。下面的这个示例展示了这个：



》》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	是设置着色器UniformBuffer (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：	 <pre> void Render2D::BeginScene(const EditorCamera& camera) { NUT_PROFILE_FUNCTION(); glm::mat4 viewProjectionMatrix = camera.GetViewProjection(); s_Data.TextureShader->Bind(); s_Data.TextureShader->SetMat4("u_ViewProjection", viewProjectionMatrix); s_Data.QuadIndexCount = 0; s_Data.TextureSlotIndex = 1; s_Data.QuadVBIndex = s_Data.QuadVBBase; void Render2D::BeginScene(const EditorCamera& camera) { NUT_PROFILE_FUNCTION(); s_Data.CameraUniformBuffer.ViewProjection = camera.GetViewProjection(); s_Data.CameraUniformBuffer->SetData(glm::mat4(camera.GetData())); s_Data.QuadIndexCount = 0; s_Data.TextureSlotIndex = 1; s_Data.QuadVBIndex = s_Data.QuadVBBase; } </pre>
运行机制：	 <p>运行机制：</p> <p>具体可以参考 https://learnopengl-cn.github.io/04%20Advanced%20OpenGL/08%20Advanced%20GLSL/%uuniform_2</p>

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"); //创建纹理着色器
//s_Data.TextureShader->Bind(); //绑定纹理着色器
//s_Data.TextureShader->SetIntArray("u_Textures", samplers, s_Data.MaxTextureSlots); //设置纹理槽
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。

需要做的修改：

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：

<p>修改前 (未使用 Uniform Buffer)</p> <pre>s_Data.TextureShader->SetMat4("u_ViewProjection", camera.GetViewProjectionMatrix());</pre>	<p>在这里，我们直接将 <code>u_ViewProjection</code> 作为一个 <code>mat4</code> 变量传递给着色器，实现统一变量的直接绑定。</p>
<p>修改后 (已使用 Uniform Buffer)</p> <pre>s_Data.CameraUniformBuffer = UniformBuffer::Create(sizeof(Renderer2DData::CameraData), 0); s_Data.CameraUniformBuffer->SetData(&s_Data.CameraBuffer, sizeof(Renderer2DData::CameraData));</pre>	<p>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 <code>SetData()</code>，绑定 UBO 到正确的绑定点 (Binding Point)。</p>

Eg: u_ViewProjection 的更新结果:

```
void Renderer2D::BeginScene(const Camera& camera, const glm::mat4& viewMatrix)
{
    NUT_PROFILE_FUNCTION();

    s_Data.CameraBuffer.ViewProjection = camera.GetProjection() * glm::inverse(viewMatrix);
    s_Data.CameraUniformBuffer->SetData(&s_Data.CameraBuffer, sizeof(Renderer2DData));

    s_Data.QuadIndexCount = 0;
    s_Data.TextureSlotIndex = 1;
    s_Data.QuadVBIndex = s_Data.QuadVBBase;
}
```

疑问:

我发现 Chernov 虽然为 u_ViewProjection 进行了更新,但是在将 u_Textures 由统一变量设置为统一变量块之后,他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_TexCoord);
    color2 = v_EntityID;
}
```

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

思考:

这个问题的原因这是为何呢?

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) 会和一个 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。

》》》五: OpenGL Shader 更新

》》》接下来我将对着色器系统进行相关更新。(其中包括了: Vulkan 植入,日志错误提醒的更新,着色器缓存文件的生成)

》》》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::chrono::high_resolution_clock::now() - m_Start).count() / 1000.0f;
    }

    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：

```
17 //
18
19 struct VertexOutput
20 {
21     vec4 Color;
22     vec2 TexCoord;
23     float TexIndex;
24     float TilingFactor;
25 };
26
27 layout(location = 0) out VertexOutput Output;
28 layout(location = 4) out flat int v_EntityID;
29
30 void main()
31 {
32     Output.Color = a_Color;
33     Output.TexCoord = a_TexCoord;
34     Output.TexIndex = a_TexIndex;
35     Output.TilingFactor = a_TilingFactor;
36     v_EntityID = a_EntityID;
37
38     gl_Position = u_ViewProjection * vec4(a_Position, 1.0);
39     //gl_Position = u_ViewProjection * u_Transform * vec4(a_Position, 1.0);
40 }
41
```

》》》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:

（着色器更新中包括了： Vulkan 植入，日志错误提醒的更新，着色器缓存文件的生成 这几点）

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。

》》》六：平台工具的更新（打开或保存文件）

》》》这里我不需要做更改，因为我的代码似乎是正确的。

》》》七：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 -----

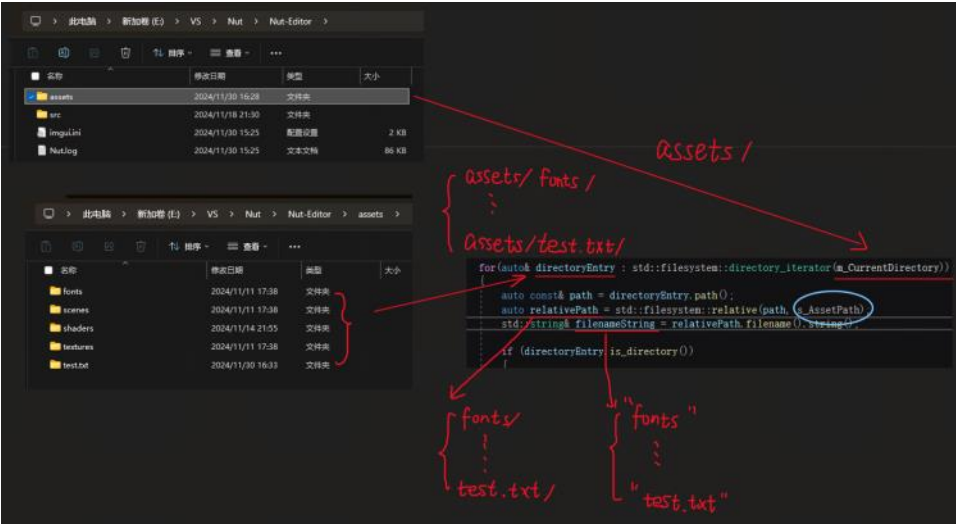
》》》std::filesystem::relative()

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

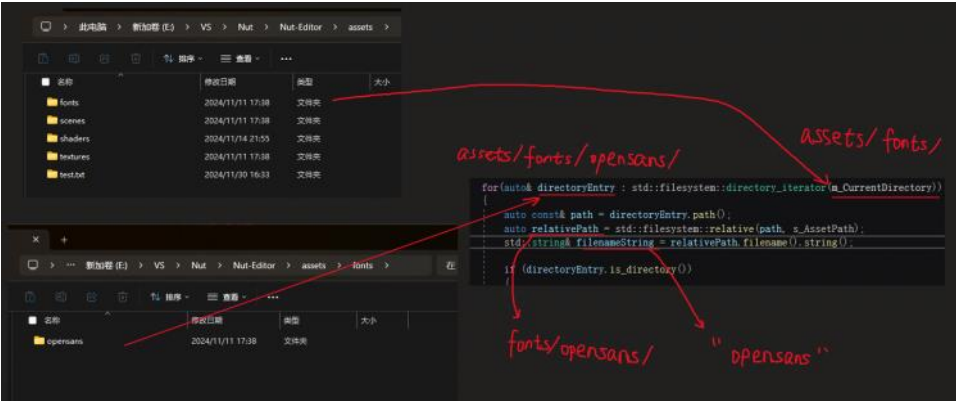
如果 s_AssetPath 是 C:\Projects\MyGame\Assets， path 是 C:\Projects\MyGame\Assets\Models\Character.obj。
那么 std::filesystem::relative(path, s_AssetPath) 会返回 Models\Character.obj，这是 path 相对于 s_AssetPath 的相对路径。

》》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->GetRenderedID(), { 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((ImTextureID)icon->GetRenderedID(), { 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) -----

》》》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)) // 当前目录
{
    auto const& path = directoryEntry.path();
    auto relativePath = std::filesystem::relative(path, m_AssetPath);
    std::string filenameString = relativePath.filename().string();
    // 获得缩略图

    ImGui::PushID(filenameString.c_str());
    ImGui::Image(textureID, ImVec2(40, 40, 0, 0));
    Ref(Texture) icon = (directoryEntry.is_directory() ? m_FolderIcon : m_FileIcon);
    ImGui::ImageButton((ImGuiTextureID)icon->GetRendererID(), { thumbnailSize, thumbnailSize },
    ImGui::StyleColor());

    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\\Vut\\Vut-Editor" / path;
            // 使用 ShellExecute 打开记事本
            ShellExecute(nullptr, "open", "F:\\Microsoft VS Code\\Code.exe", absolutePath.string(), nullptr, SW_SHOW);
        }
    }

    ImGui::TextWrapped(filenameString.c_str());
    // 增加文本

    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（例如多个交互式按钮）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 Chernov 的代码，我们会发现，使用 DragDrop 功能只需要两步操作：设置拖动源、设置拖动目标。

拖动源的设置 (ContentBrowserPanel.cpp)	<pre>// allow drag & drop function if (ImGui::BeginDragDropSource()) { const wchar_t* itemPath = relativePath.c_str(); ImGui::SetDragDropPayload("CONTENT_BROWSER_ITEM", itemPath, (wcslen(itemPath) + 1) * sizeof(wchar_t)); ImGui::EndDragDropSource(); }</pre>
拖动目标的设置 (EditorLayer.cpp)	<pre>ImGuiTextureID textureID = (void*)m_Framebuffer->GetColorAttachmentRendererID(); ImGui::Image(textureID, ImVec2(m_ViewPortSize.x, m_ViewPortSize.y), ImVec2(0, 1), ImVec2(1, 1)); // Confirm boundary values ImVec2 minBound = ImGui::GetWindowPos(); minBound.x += viewportOffset.x; minBound.y += viewportOffset.y; m_ViewPortBounds[0] = { minBound.x, minBound.y }; m_ViewPortBounds[1] = { minBound.x + m_ViewPortSize.x, minBound.y + m_ViewPortSize.y }; if (ImGui::BeginDragDropTarget()) { if (const ImGuiPayload* payload = ImGui::AcceptDragDropPayload("CONTENT_BROWSER_ITEM")) { const wchar_t* path = (const wchar_t*)payload->Data; OpenScene(std::filesystem::path(m_AssetPath) / path); } ImGui::EndDragDropTarget(); }</pre>

》》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。

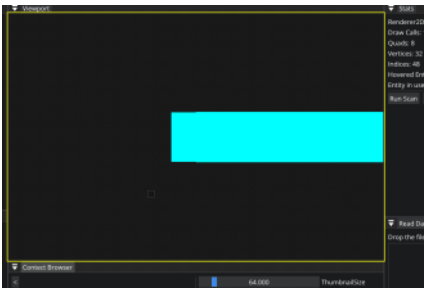
》》关于拖拽预览的绘制，还需要注意一点：

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。



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。

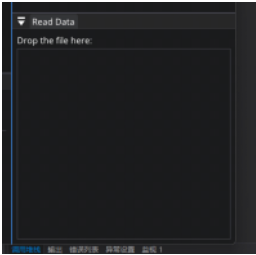
Eg.

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

// Create a dummy widget to act as a drag target
// ImGui::Dummy(ImGui::GetContentRegionAvail());
// Create a dummy widget to act as a drag target
// ImGui::Dummy(ImGuiVec2(targetSize.x, targetSize.y)); // Only create an empty area

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

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

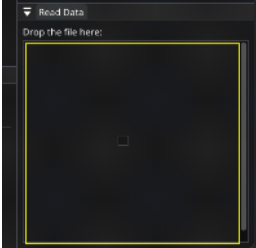


此时便需要我们创建一个可交互的区域： `ImGui::Button`、 `ImGui::Dummy` 等等控件，以此来完善文件拖动的功能。

```
// Create a dummy widget to act as a drag target
if (ImGui::BeginChild("DropTarget", targetSize, true))
{
    // Create a dummy widget to act as a drag target
    ImGui::Dummy(ImGuiVec2(targetSize.x, targetSize.y)); // Only create an empty area

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

ImGui::EndChild();
```



》》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： <code>ImGui::Dummy</code> 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 <code>ImGui::Dummy</code> ，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 <code>Dummy</code> 来占据空间。
语法：	<code>void ImGui::Dummy(const ImVec2& size);</code>
参数：	<code>size</code> : 指定占位符的大小，通常是一个 <code>ImVec2</code> (x 和 y 坐标)。这定义了 <code>Dummy</code> 占据的区域的大小。
示例：	假设你想在 <code>ImGui</code> 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： <code>ImGui::Begin("Example Window");</code> <code>// 创建一个大小为 200x200 的占位符区域</code> <code>ImGui::Dummy(ImVec2(200, 200));</code> <code>ImGui::End();</code>

----- Texture Drag&Drop -----

》》》很久没回来更新了，懒人一个。
之前把游戏引擎的视频看完了，但一直疏于更新，接下来我好好更新。（真的）

》》》没什么要记的

-----Something you need to know in GAME ENGINE -----

》》》看了一会看不动了，应该也没什么代码提交，So I skip that

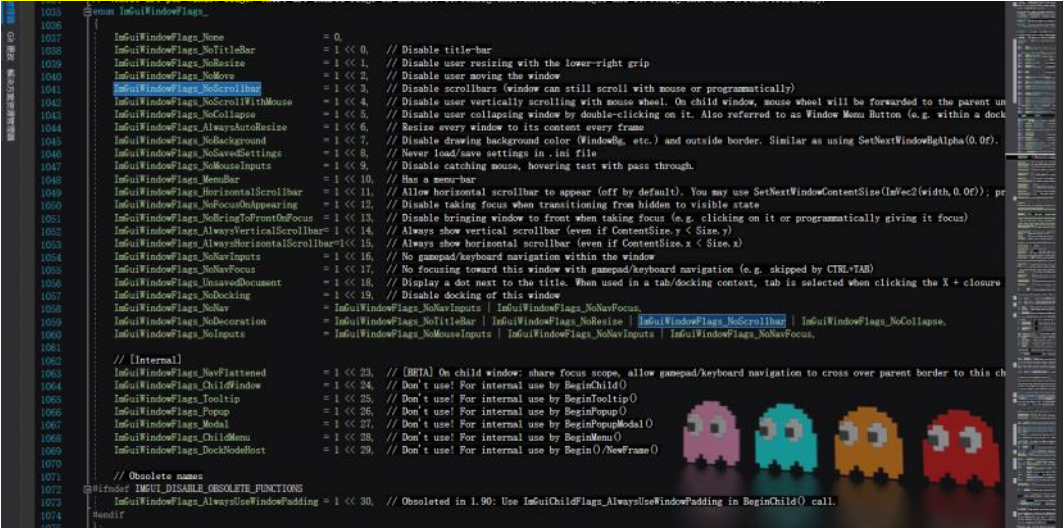
----- Play Button -----

》》》5:28 ~ 15:05 修复纹理撕裂的Bug

》》》后面好像也没什么好记的

ImGui::GetWindowContentRegionMax()	
函数签名:	ImVec2 ImGui::GetWindowContentRegionMax();
返回值:	ImVec2 类型，表示当前窗口内容区域的最大坐标（右下角的坐标）。
通常与ImGui::GetWindowContentRegionMin()一起使用。	

》》》一些 ImGuiWindowFlags_ 的定义:



》》》GL_LINEAR 和 GL_NEAREST 的概念及区别:

概念:

GL_NEAREST (也叫最近过滤, Nearest Neighbor Filtering) 是OpenGL默认的纹理过滤方式。当设置为GL_NEAREST的时候, OpenGL会选择中心点最接近纹理坐标的那个像素。下面图中你可以看到四个像素, 加号代表纹理坐标, 左上角那个纹理像素的中心距离纹理坐标最近, 所以它会被选择为样本颜色:

GL_LINEAR (也叫线性过滤, Bilinear Filtering) 它会根据纹理坐标附近的纹理像素, 计算出一个插值, 近似出该处纹理像素之间的颜色。一个纹理像素的中心距离纹理坐标越近, 那么这个纹理像素的颜色对最终的颜色贡献越大。下面你可以看到返回的颜色是邻近像素的混合色:

那么这两种纹理过滤方式有怎样的视觉效果呢? 让我们看看在一个很大的物体上应用一张低分辨率的纹理会发生什么吧 (纹理被放大了, 每个纹理像素都能看到):

GL_NEAREST产生了锯齿状的图案。我们能够清晰看到组成纹理的像素, 而GL_LINEAR能够产生更平滑的图案, 很模糊出单个的纹理像素。GL_LINEAR可以产生更真实的输出, 但也有开发者更喜欢0-bits风格, 所以他们会用GL_NEAREST选项。

对比

特性	GL_NEAREST	GL_LINEAR
工作原理	选择最接近纹理坐标的像素	对周围 4 个像素进行双线性插值
放大效果	像素化 (块状)	平滑
缩小效果	可能出现闪烁或锯齿	平滑
性能	计算简单, 性能开销低	计算复杂, 性能开销较高
通用场景	像素艺术、性能敏感场景	高质量纹理渲染、3D 游戏

----- 2D Physics -----

》》》概述

2: 51 ~ 9: 40 修复一个无法编译的错误

9:50 ~ 13:30 将 Box2D 设置为子模块

- 13: 50 ~ 16: 25 修改 Premake 文件
- 18: 30 ~ 25:18 Box2D 使用解释
- 25: 20 ~ 25: 58 引擎的一些小改变
- 26: 00 ~ 56: 32 设计组件和实际使用 Box2D
- 56: 32 ~ 1: 08: 00 UI 界面的设置以及成果运行展示
- 1: 08: 20 ~ 1: 19: 45 序列化与反序列化以及成果演示

》》》网址 Box2D (3.1.0)	
官网:	https://box2d.org/
文档:	https://box2d.org/documentation/
Github:	https://github.com/erincatto/box2d

非常建议在开始写代码前阅读 (3.1.0) :

简单演示参考->	https://box2d.org/documentation/hello.html
模拟时的代码参考->	https://box2d.org/documentation/md_simulation.html (包含了 ID, World, Body,Shapes,Contacts, Joints 的定义、初始化、概念等等)

》》》网址 (2.4.1)	
官网:	https://box2d.org/doc_version_2_4/
Github	https://github.com/erincatto/box2d/tree/9ebbbcd960ad424e03e5de6e66a40764c16f51bc

》》》开始之前

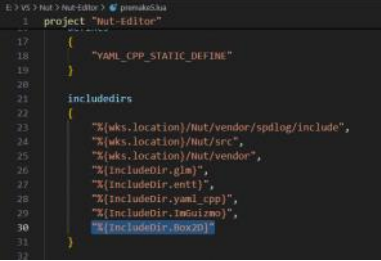


我发现有人曾经提示过版本更改的问题，Cherno 使用的是 2.4.1，当前已经更新到 3.1.0。但我选择先使用 3.1.0 试试看，毕竟新的库更前卫一些，我也想尝尝鲜。如果你想按照 Cherno 的想法来，就照他的方法做，使用 2.4.1。如果你想保持 2.4.1 版本中的 C++ 特性，且对性能要求不敏感，那就使用 2.4.1。

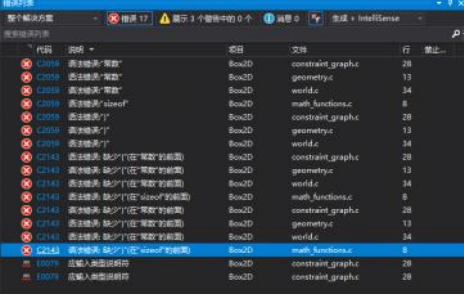
更改和操作:

3.0.1 相较于 2.4.1 有了较大变化，文件结构发生变化。另外，由 C++ 转换为了 C。（迁移指南: https://box2d.org/documentation/md_migration.html）

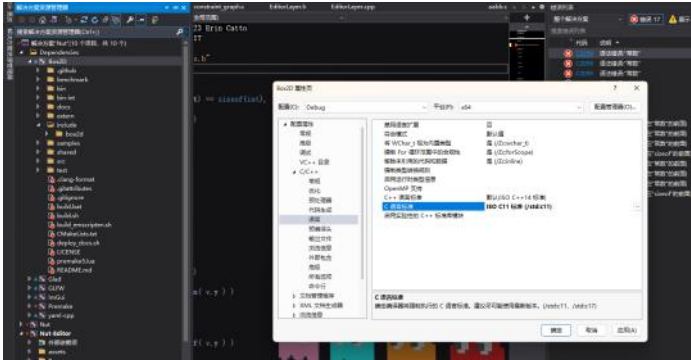
》》遇到的第一个错误：来自 Nut-Editor 的 LINK 错误，解决方案:



》》遇到的第二个错误：许多语法错误



并且所有的错误都指向一个函数: _Static_assert()。我想这是因为没有将 C 语言的编译器设置为 /std:c11，因为 _Static_assert() 是 c11 中的特性。在 C++11 中，这个函数被定义为 Static_assert()。我手动在 Box2D 的属性页中设置了 C 编译器，将其从默认 (旧MSVC) 修改为 ISO C11标准(/std:c11)



》》但是还有一个问题，就是我们无法将这个操作写在 premake 文件中，即无法将其脚本化。

我搜集了很多论坛和答案，但是 premake 好像无法为 msvc 提供合适的指令，也就是说没有可用的指令对 C 编译器的版本进行修改。
类似的指令有：buildoptions { "/std:c11" } 或者 buildoptions { "-std=c11" }，但是这两个指令似乎只能针对 GCC/Clang 的 C 编译器，对其进行自动化更改。

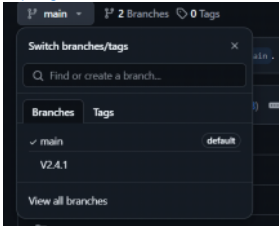
这会导致一个结果，如果我选用了 Box2D 的 3.1.0 版本，为了在项目中正常使用 Box2D，则必须修改 MSVC 中的 C 编译器（以修正报错。但是我无法在 premake 中脚本化这个操作，就只能手动设置。这时，如果在外部重新使用 bat 脚本（Win-GenProjects.bat）运行或更新项目，则会导致 Box2D API 受它本身的 Premake 脚本影响，从手动设置的 /std:c11 状态退回到默认（因为 premake 中的指令无法对 msvc 进行 C 编辑器的修改，即使写下类似的代码，也相当于空白，所以只要重新调用 Box2D 的 premake5.lua，就总是会撤回 VS 中手动设置 C 编辑器的版本）

这里我提供3个解决方案：

第一（我选择的）	退回至 Box2D 2.4.1 版本的使用，因为这个版本由C++开发而成，没有上述问题。虽然性能不如 3.1.0 好，但目前引擎还没有遇到性能瓶颈，而且在项目中植入 C++ 很容易。
第二	将 Box2D 在 MSVC 中 C 编辑器的修改，调整到独立于 premake 之外的脚本中，以实现自动化操作。
第三	对 Box2D.vcproj 直接进行修改，在该文件中直接标明 Box2d C 编辑器的版本为 /std:c11，这个操作可能会受到 premake 脚本重新运行的影响。（我是说可能，我也没有仔细思考）

》》》没想到经过一番查证和思索，到头来还是使用 Box2D 老版本。

<https://github.com/JJJJJJustin/box2d> 这是我 fork 之后创建的库，其中有两个分支：main 代表最新的 Box2d，V2.4.1 代表 2.4.1 版本，你们可以使用。

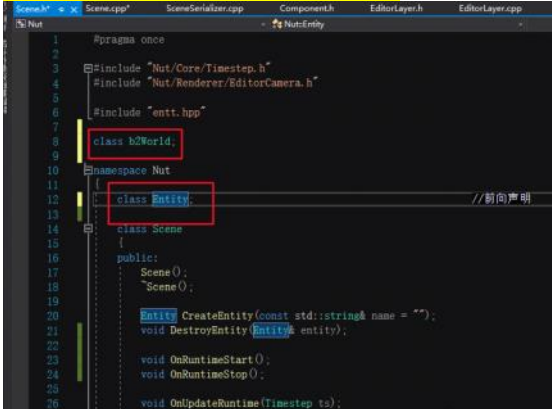


》》》我先提交序列化-反序列化部分的代码，然后再提交逻辑更新，以及UI设置。

》》》return {}; 和 return; 的区别

return;	用于 void 函数，表示结束函数。
return {};	用于有返回值的函数，它返回一个默认初始化的对象，通常会将返回值设为类型的默认值（例如，0、nullptr、空字符串等）。

》》》关于前向声明的位置问题（命名空间之内与命名空间之外）



命名空间的影响：

- 处于命名空间内部：Entity 的前向声明位于 Nut 命名空间内，这意味着编译器会认为这个 Entity 类是属于 Nut 命名空间的 Entity 类。所有在 Nut::Scene 类中使用 Entity 类型的地方，编译器都会以 Nut 命名空间下定义的 Entity 类进行条件判断。
- 移到命名空间外部：如果你将前向声明移到 Nut 命名空间外，那么 Entity 类将不再被视为 Nut 命名空间的一部分。此时，编译器将 Entity 视为全局作用域中的一个类。任何在 Nut 命名空间内使用 Entity 的地方，将无法正确识别它是属于 Nut 命名空间的类，编译器将会在全局作用域中寻找 Entity 类的定义。

后果：

- 如果 Scene 中的 Entity 使用的是命名空间内的 Entity（即 Nut-Entity），而前向声明被移到命名空间外部，就会导致编译错误，提示找不到 Nut.Entity。
- 编译器会试图查找一个全局作用域中的 Entity 类，而实际上你可能需要的是 Nut.Entity，因此会出现命名冲突或找不到类定义的问题。

))) 关于初始化 (OnRuntimeStart())

```

// Init the Box2D world & Attach all Physical properties
void Scene::OnRuntimeStart()
{
    m_PhysicsWorld = new b2World( { 0.0f, -9.8f } );
    auto& view = m_Registry.view<Rigidbody2DComponent>();
    for(auto& e : view)
    {
        Entity entity = { e, this };
        auto& tc = entity.GetComponent<TransformComponent>();
        auto& rb2c = entity.GetComponent<Rigidbody2DComponent>();

        b2BodyDef bodyDef;
        bodyDef.type = rb2c.Type;
        bodyDef.position.Set(tc.Translation.x, tc.Translation.y);
        bodyDef.angle = tc.Rotation.z;
        b2Body* body = m_PhysicsWorld->CreateBody(&bodyDef);

        body->SetFixedRotation(rb2c.FixedRotation);
        rb2c.RuntimeBody = body;

        if (entity.HasComponent<BoxCollider2DComponent>())
        {
            auto& bc2c = entity.GetComponent<BoxCollider2DComponent>();
            b2PolygonShape boxShape;
            boxShape.SetAsBox(bc2c.Size.x, bc2c.Size.y);

            b2FixtureDef fixtureDef;
            fixtureDef.shape = &boxShape;
            fixtureDef.density = bc2c.Density;
            fixtureDef.friction = bc2c.Friction;
            fixtureDef.restitution = bc2c.Restitution;
            fixtureDef.restitutionThreshold = bc2c.RestitutionThreshold;

            body->CreateFixture(&fixtureDef);
        }
    }
}

```

Creating a World

Every Box2D program begins with the creation of a **b2World** object. **b2World** is the physics hub that manages memory, objects, and simulation. You can allocate the physics world on the stack, heap, or data section. It is easy to create a Box2D world. First, we define the gravity vector.

```
b2Vec2 gravity(0.0f, -10.0f);
```

Now we create the world object.

```
b2World world(gravity);
```

Creating a Dynamic Body

So now we have a ground body. We can use the same technique to create a dynamic body. The main difference, besides dimensions, is that we must establish the dynamic body's mass properties.

First we create the body using **CreateBody**. By default bodies are static, so we should set the **b2BodyType** at construction time to make the body dynamic.

```
b2BodyDef bodyDef;
bodyDef.type = b2_dynamicBody;
bodyDef.position.Set(0.0f, 4.0f);
b2Body* body = world.CreateBody(&bodyDef);
```

Next we create and attach a polygon shape using a fixture definition. First we create a box shape:

```
b2PolygonShape dynamicBox;
dynamicBox.SetAsBox(1.0f, 1.0f);
```

Next we create a fixture definition using the box. Notice that we set density to 1. The default density is zero. Also, the friction on the shape is set to 0.3.

```
b2FixtureDef fixtureDef;
fixtureDef.shape = &dynamicBox;
fixtureDef.density = 1.0f;
fixtureDef.friction = 0.3f;
```

Caution: A dynamic body should have at least one fixture with a non-zero density. Otherwise you will get strange behavior.

Using the fixture definition we can now create the fixture. This automatically updates the mass of the body. You can add as many fixtures as you like to a body. Each one contributes to the total mass.

```
body->CreateFixture(&fixtureDef);
```

))) 关于内存泄漏问题

```

// Destroy the Box2D world
void Scene::OnRuntimeStop()
{
    delete m_PhysicsWorld;
    m_PhysicsWorld = nullptr;
}

```

如果 Delete 之后，不执行 `m_PhysicsWorld = nullptr`；这句代码，会出现什么情况？

悬挂指针 (Dangling Pointer)

- 执行 `delete m_PhysicsWorld`；会释放 `m_PhysicsWorld` 指向的内存，但 `m_PhysicsWorld` 本身仍然持有之前指向已释放内存的地址。
- 如果后续尝试使用 `m_PhysicsWorld`（例如访问它或再次删除它），会导致未定义的行为（通常是崩溃），因为 `m_PhysicsWorld` 现在是一个悬挂指针，指向已经无效的内存。

))) OnUpdateRuntime() 中的更新

```

void Scene::OnUpdateRuntime(Timestep ts)
{
    // Allow physics simulation & Update the transform of body pre-frame for rendering
    const int velocityIterations = 6;
    const int positionIterations = 2;
    m_PhysicsWorld->Step(ts, velocityIterations, positionIterations);

    auto& view = m_Registry.view<Rigidbody2DComponent>();
    for(auto& e : view)
    {
        Entity entity = { e, this };
        auto& tc = entity.GetComponent<TransformComponent>();
        auto& rb2c = entity.GetComponent<Rigidbody2DComponent>();

        b2Body* body = (b2Body*)rb2c.RuntimeBody;
        b2Vec2 position = body->GetPosition();
        float angle = body->GetAngle();

        tc.Translation.x = position.x;
        tc.Translation.y = position.y;
        tc.Rotation.z = angle;
    }
}

```

The suggested iteration count for Box2D is 8 for velocity and 3 for position. You can tune this number to your liking, just keep in mind that this has a trade-off between performance and accuracy. Using fewer iterations increases performance but accuracy suffers. Likewise, using more iterations decreases performance but improves the quality of your simulation. For this simple example, we don't need much iteration. Here are our chosen iteration counts.

```
int32 velocityIterations = 6;
int32 positionIterations = 2;
```

Note that the time step and the iteration count are completely unrelated. An iteration is not a sub-step. One solver iteration is a single pass over all the constraints within a time step. You can have multiple passes over the constraints within a single time step.

We are now ready to begin the simulation loop. In your game the simulation loop can be merged with your game loop. In each pass through your game loop you call **b2World::Step**. Just one call is usually enough, depending on your frame rate and your physics time step.

The Hello World program was designed to be simple, so it has no graphical output. The code prints out the position and rotation of the dynamic body. Here is the simulation loop that simulates 60 time steps for a total of 1 second of simulated time.

```
for (int32 i = 0; i < 60; ++i)
{
    world.Step(timeStep, velocityIterations, positionIterations);
    b2Vec2 position = body->GetPosition();
    float angle = body->GetAngle();
    printf("%.2f %.2f %.2f %.2f\n", position.x, position.y, angle);
}
```

The output shows the box falling and landing on the ground box. Your output should look like this:

))) OnRuntimeStart() 和 OnUpdateRuntime() 中代码的作用：

初始化 Box2d 世界, 并预先将所有物理属性附加到对象上

```
// Init the Box2d world & Attach all physical properties to the object beforehand
void Scene::OnRuntimeStart()
{
    m_PhysicsWorld = new b2World({ 0.0f, -9.8f });

    auto view = m_Registry.view<Rigidbody2DComponent>();
    for (auto e : view)
    {
        Entity entity = { e, this }; // Just for using member function of class Entity
        auto& tc = entity.GetComponent<TransformComponent>();
        auto& rb2c = entity.GetComponent<Rigidbody2DComponent>();

        b2BodyDef bodyDef;
        bodyDef.type = Rigidbody2DTypeToBox2DBody(rb2c.Type);
        bodyDef.position.Set(tc.Translation.x, tc.Translation.y);
        bodyDef.angle = tc.Rotation.z;
        b2Body* body = m_PhysicsWorld->CreateBody(&bodyDef);

        body->SetFixedRotation(rb2c.FixedRotation);
        rb2c.RuntimeBody = body;

        if (entity.HasComponent<BoxCollider2DComponent>())
        {
            auto& bc2c = entity.GetComponent<BoxCollider2DComponent>();

            b2PolygonShape boxShape;
            boxShape.SetAsBox(bc2c.Size.x, bc2c.Size.y);

            b2FixtureDef fixtureDef;
            fixtureDef.shape = &boxShape;
            fixtureDef.density = bc2c.Density;
            fixtureDef.friction = bc2c.Friction;
            fixtureDef.restitution = bc2c.Restitution;
            fixtureDef.restitutionThreshold = bc2c.RestitutionThreshold;

            body->CreateFixture(&fixtureDef);
        }
    }
}
```

允许物理模拟, 并每帧都更新物体的 Transform 以进行渲染

```
void Scene::OnUpdateRuntime(Timestep ts)
{
    {
        // Allow physics simulation & Update the body transform every frame for rendering
        const int velocityIterations = 6;
        const int positionIterations = 2;
        m_PhysicsWorld->Step(ts, velocityIterations, positionIterations);

        auto view = m_Registry.view<Rigidbody2DComponent>();
        for (auto e : view)
        {
            Entity entity = { e, this };
            auto& tc = entity.GetComponent<TransformComponent>();
            auto& rb2c = entity.GetComponent<Rigidbody2DComponent>();

            b2Body* body = (b2Body*)rb2c.RuntimeBody;
            b2Vec2 position = body->GetPosition();
            float angle = body->GetAngle();

            tc.Translation.x = position.x;
            tc.Translation.y = position.y;
            tc.Rotation.z = angle;
        }
    }
}
```

》》》我发现 yaml 的序列化系统似乎没有将 Texture 的结果进行保存, 每次进入引擎的场景之后, 纹理都会被刷新掉。

----- UUID -----

》》》xhash ?

》》》代码理解:

》解释代码:

1. `std::random_device s_RandomDevice;`
 - o `std::random_device` 是用于生成随机数的设备 (通常依赖硬件或操作系统提供的随机源), 用来初始化 `std::mt19937_64` 引擎。
2. `std::mt19937_64 s_Engine(s_RandomDevice());`
 - o `s_Engine` 是一个基于 `std::mt19937_64` 的随机数生成器。它的种子是从 `s_RandomDevice` 获取的一个值。
3. `std::uniform_int_distribution<uint64_t> s_UniformDistribution;`
 - o `s_UniformDistribution` 是一个均匀分布对象, 用于生成在某个范围内的随机整数。
 - o 当前这行代码并没有指定范围, 默认情况下它生成的随机数范围是 `{std::numeric_limits<uint64_t>::min(), std::numeric_limits<uint64_t>::max()}`, 即 `uint64_t` 类型的最小值到最大值。

```
static std::random_device s_RD; // 设置随机数种子, 用于初始化引擎
static std::mt19937_64 s_Engine(s_RD); // 根据随机数种子, 创建随机数生成器 (随机数引擎)
static std::uniform_int_distribution<uint64_t> m_UniformDistribution; // 创建分布对象 (用于约束随机数生成的范围、频率等特征)

UUID::UUID()
{
    m_UUID(m_UniformDistribution(s_Engine)) // 使用随机数引擎生成一个数字, 并使用分布对象将其转化
}
{
}
}
```

》范围设置 与 实际使用

默认情况下, `s_UniformDistribution` 可以生成的随机数范围是: `std::numeric_limits<uint64_t>::min() ~ std::numeric_limits<uint64_t>::max()`
即 $0 \sim 2^{64} - 1$ -> (0, 18446744073709551615)

如果需要设置范围, 可以这样操作:

```
int main() {
    // 随机设备，用于初始化引擎的种子
    std::random_device rd;

    // 创建 Mersenne Twister 引擎，并用随机设备初始化
    std::mt19937_64 engine(rd());

    // 创建均匀整数分布，范围是 [100, 1000]
    std::uniform_int_distribution<uint64_t> dist(100, 1000);

    // 从引擎中生成一个随机数，并使用分布转换它
    uint64_t randomValue = dist(engine);

    std::cout << "Generated random number: " << randomValue << std::endl;

    return 0;
}
```

))) 运算符重载的定义方式:

))))

```
operator uint64_t() const { return m_UUID; }
uint64_t operator() const { return m_UUID; }

这两个有什么区别? 哪一个运算符重载?
```

operator uint64_t() const { return m_UUID; }

- **目的:** 这是一个运算符重载，旨在将 **UUID** 类型的对象转换为 **uint64_t** 类型。
- **用途:** 当你想隐式或显式地将 **UUID** 类型转换为 **uint64_t** 类型时，会调用这个运算符。
- **示例:**

```
Nut::UUID uuid(123456);
uint64_t val = uuid; // 隐式调用 operator uint64_t()
```
- **结论:** 语法正确，并且是运算符重载。

uint64_t operator() const { return m_UUID; }

- **目的:** 这是一个函数调用运算符 (**operator()**)，它允许对象像函数一样被调用。这并不是类型转换运算符，而是定义了一个函数调用行为。
- **用途:** 你可以通过 **()** 操作符调用一个 **UUID** 对象。例如，如果你实例化了 **UUID** 对象，可以像调用函数一样调用它，**operator()** 会返回 **m_UUID** 的值。
- **示例:**

```
Nut::UUID uuid(123456);
uint64_t val = uuid(); // 调用 operator(), 返回 m_UUID
```
- **结论:** 语法正确，能够使得 **UUID** 对象像函数一样被调用。

结论:

- 如果你的目的是将 **UUID** 对象转换为 **uint64_t** 类型，那么 **第一个 (operator uint64_t())** 更为合适。因为它是运算符重载，能够让 **UUID** 对象与 **uint64_t** 类型之间进行无缝转换。
- 如果你的目的是让 **UUID** 对象像函数一样被调用并返回 **m_UUID**，那么 **第二个 (uint64_t operator() const)** 是正确的。它实现了函数调用操作符。

)) 同样的，这里也可以看到 函数调用操作符 的痕迹:

```
static std::random_device s_RD; // 设置随机数种子.
static std::mt19937_64 s_Engine(s_RD); // 根据随机数种子.
static std::uniform_int_distribution<uint64_t> s_UniformDistribution;
```

s_RD 只是对象本身，它本身并没有直接生成随机数。

s_RD() 是对 **std::random_device** 对象的调用，它会生成一个随机数（通常是 **unsigned int** 类型），并返回。

s_RD() 实际上是调用 **std::random_device** 的 **operator()**，它返回一个随机数，并将这个数作为种子传递给 **std::mt19937_64** 引擎。

)))) 一些思考:

```
struct IDComponent
{
    UUID ID;

    IDComponent() = default;
    IDComponent(const UUID& id)
    : ID(id) {}
    IDComponent(const IDComponent&) = default;
};
```

Cherno的提交中，并没有为 **IDComponent** 提供这个自定义的构造函数。我认为这是一个会出现争议的地方。

```
Entity Scene::CreateEntityWithUUID(UUID uuid, const std::string& name)
{
    Entity entity = { m_Registry.create(), this };
    entity.AddComponent<IDComponent>(UUID(uuid));
    entity.AddComponent<TransformComponent>(glm::vec3{ 0.0f });
    auto& tag = entity.AddComponent<TagComponent>();
    tag.Tag = name.empty() ? "Unnamed Entity" : name;

    return entity;
}
```

在这个函数中，我们为 **AddComponent<>()** 传入了参数: **uuid**。
就我的理解来看，这里传入的 **uuid**，将会被用来初始化 **IDComponent** 中的 **ID**，所以 **IDComponent** 中需要上述的构造函数。

在这里，我为 **AddComponent()** 填入的参数是 **UUID(uuid)**，意为我使用了 **UUID** 中的构造函数:

```
UUID::UUID()
: m_UUID(s_UniformDistribution(s_Engine))
{
}

UUID::UUID(uint64_t uuid)
: m_UUID(uuid)
{
}
```

而不是默认的构造函数，而默认的构造函数会生成随机的一段数字。

```
bool AddComponent(Args&&... args)
{
    NUT_CORE_ASSERT(!HasComponent<T>(), "This Entity already has component!");
    T& component = m_Scene->m_Registry.emplace<T>(m_EntityHandle, std::forward<Args>(args)...);
    m_Scene->OnComponentAdded<T>(*this, component);
    return component;
}
```

这里是 **AddComponent()** 的定义。

但从运行结果看来，Cherno 似乎并没有什么错误。不过我还没完成这一集的提交，稍后再来证明我是否正确。

)) 事实证明没什么影响。

》》》随机 ID 分发以及使用流程:

创建实体 (创建时使用 CreateEntity() 函数, 而不是 CreateEntityWithUUID(), 这会为新创建的实体分配一个随机 ID)

```
if (ImGui::BeginPopupContextWindow(0, 1 | ImGuiPopupFlags_NoOpenOverItems))
{
    if (ImGui::MenuItem("Create Empty Entity"))
        m_Context->CreateEntity("Empty Entity");
    ImGui::EndPopup();
}
```

序列化文件: (我们会将默认随机分配的 ID 保存进配置文件)

```
void SceneSerializer::SerializeEntity(YAML::Emitter& out, Entity& entity)
{
    NUT_CORE_ASSERT(entity.HasComponent<IDComponent>(), "Entity component do not have a universal uniform ID.");

    out << YAML::BeginMap; // Entity
    out << YAML::Key << "Entity" << YAML::Value << entity.GetComponent<IDComponent>().ID;

    if (entity.HasComponent<TagComponent>())
    {
        out << YAML::Key << "TagComponent";
        out << YAML::BeginMap;
        auto& tag = entity.GetComponent<TagComponent>().Tag;
```

反序列化文件: (读取文件中的文本数据, 并将其重新用于实体组件的初始化。同时, ID 将会一直保持在文本配置文件中, 实体只有在最初创建时通过 CreateEntity() 函数获得其 UUID, 之后便一直存储在配置文件中, 除非手动更改)

```
auto entities = data["Entities"];
if(entities)
{
    for (auto entity : entities)
    {
        uint64_t uuid;
        std::string name;

        uuid = entity["Entity"].as<uint64_t>();
        // Enter the TagComponent map,
        // and search for tag in submap(submap is stroed in TagComponent map)
        auto tc = entity["TagComponent"];
        if (tc)
            name = tc["Tag"].as<std::string>();

        NUT_CORE_TRACE("Deserialized entity with ID = {0}, name = {1}", uuid, name);
        Entity& deserializedEntity = m_Scene->CreateEntityWithUUID(uuid, name);
        DeserializeEntity(entity, deserializedEntity); // Update value
```

》》值得注意的是

如果你想像 Cherno 那样, 先通过 Ctrl + Shift + S, 在保存文件的时候, 更改所有实体的 ID。(将实体的 ID 从之前设置的固定 ID 改为 随机的 UUID, 则需要做以下更改)

先使用 UUID 的默认构造函数进行随机数分发:

```
void SceneSerializer::SerializeEntity(YAML::Emitter& out, Entity& entity)
{
    NUT_CORE_ASSERT(entity.HasComponent<IDComponent>(), "Entity component do not have a univ

    out << YAML::BeginMap; // Entity
    out << YAML::Key << "Entity" << YAML::Value << UUID();

    if (entity.HasComponent<TagComponent>())
    {
        out << YAML::Key << "TagComponent";
        out << YAML::BeginMap;

        auto& tag = entity.GetComponent<TagComponent>().Tag;
        out << YAML::Key << "Tag" << YAML::Value << tag;

        out << YAML::EndMap;
```

当分发的 ID 被存储在 yaml 配置文件中之后, 我们将其更改为图示。之后便可以不再更改这里的代码。

接下来只会有实体在创建之初, 才会拥有一个新的 UUID (具体参考上述: 》》》随机 ID 分发以及使用流程:)

否则场景在被加载时, 只会通过配置文件中的 ID 数据进行初始化, 只要文件中的数据不变, 实体的 ID 将一直保持。

```
void SceneSerializer::SerializeEntity(YAML::Emitter& out, Entity& entity)
{
    NUT_CORE_ASSERT(entity.HasComponent<IDComponent>(), "Entity component do not have a universal uniform ID.");

    out << YAML::BeginMap; // Entity
    out << YAML::Key << "Entity" << YAML::Value << entity.GetComponent<IDComponent>().ID;

    if (entity.HasComponent<TagComponent>())
    {
```

》》》一个疑惑: 这两个代码效果应该是一样的。

```
Entity Scene::CreateEntityWithUUID(UUID uuid, const std::string& name)
{
    Entity entity = { m_Registry.create(), this };

    entity.AddComponent<IDComponent>({UUID(uuid)});
    entity.AddComponent<TransformComponent>(glm::vec3{ 0.0f });
    auto& tag = entity.AddComponent<TagComponent>();
    tag.Tag = name.empty() ? "Unnamed Entity" : name;

    return entity;
}
```

```
Entity Scene::CreateEntityWithUUID(UUID uuid, const std::string& name)
{
    Entity entity = { m_Registry.create(), this };

    entity.AddComponent<IDComponent>({uuid});
    entity.AddComponent<TransformComponent>(glm::vec3{ 0.0f });
    auto& tag = entity.AddComponent<TagComponent>();
    tag.Tag = name.empty() ? "Unnamed Entity" : name;

    return entity;
}
```

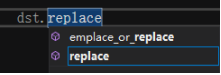

----- Playing and stopping Scene -----

》》》 这一集一共做了三件事：区别编辑器场景和运行时场景、制作复制实体的快捷键、重新设置保存 & 另存为这两个功能。

》》》 Unordered_map 中，通过下标访问对应值的方式 和 通过成员函数：at() 访问有什么不同？

<pre>map[uuid];</pre>	<ul style="list-style-type: none">• 如果 <code>uuid</code> 不存在，<code>std::map</code> 或 <code>std::unordered_map</code> 会自动插入一个新的元素，并将其值初始化为默认值（对于 <code>std::map</code>，键对应的值类型会调用默认构造函数）。• 这种方式不抛出异常，且可能会导致不期望的元素插入。
<pre>map.at(uuid);</pre>	<ul style="list-style-type: none">• <code>at()</code> 是一个成员函数，用于访问指定键 <code>uuid</code> 对应的值。如果找不到该键，它会抛出一个 <code>std::out_of_range</code> 异常。• 如果你不希望检索并插入哈希表中未存在的值，<code>at()</code> 更加安全，因为它能防止这种情况发生，并抛出异常。

》》》 entt::registry 的成员函数：replace 和 emplace_or_replace 有什么区别？



emplace_or_replace	<ul style="list-style-type: none">• 功能: 尝试在指定的实体上添加一个组件。如果该组件已存在，则会替换它。• 行为:<ul style="list-style-type: none">◦ 如果实体没有该组件，<code>emplace_or_replace</code> 会创建并添加该组件。◦ 如果实体已有该组件，<code>emplace_or_replace</code> 会替换现有的组件。• 用途: 当你不确定组件是否存在，并且希望确保该组件存在且始终是最新的时，使用 <code>emplace_or_replace</code>。
replace	<ul style="list-style-type: none">• 功能: <code>replace</code> 仅在实体已经拥有指定的组件时，才会进行替换操作。• 行为:<ul style="list-style-type: none">◦ 如果实体已经存在该组件，<code>replace</code> 会替换原有组件的值。◦ 如果实体没有该组件，<code>replace</code> 会什么都不做，并不会添加新组件。• 用途: 如果你确定组件已经存在，并且只想更新它，使用 <code>replace</code>。

----- Rendering Circles -----

》》》 概述

0:00 ~ 8:41 Hazel3D 中对于圆形渲染和多边形碰撞的演示

8:42 ~ 13:18 一些赘述

13: 18~19:36 CircleComponent 的设置、CircleComponent UI 绘制、CircleComponent 的添加途径

19:45~35:42 渲染代码的编写

35:45~42:42 着色器的编写

42:44 ~ 44:43 渲染函数的更新

44:44~54:22 调试与演示

54:43~1:00:00 边缘检测和选中物体

1:00:04~1:01:55 圆形渲染调试

1:02:00 ~ 1:04:27 序列化文件