GAMES104 现代游戏引擎：从入门到实践

作业 1 小试图形渲染

一、前言：

经过了 4 节游戏引擎图形渲染相关的课程之后，同学们应已对图形渲染的概念有了初步的认识，对其在整个引擎中的地位及大致流程和功能的理解也初具雏形。相信现在有的同学已经跃跃欲试，想在我们的开源小引擎 Pilot 中亲自实现一个渲染功能。然而图形渲染道阻且长，想要真正实现一个渲染功能，即使有了图形 API 的帮助，其所需的知识储备、代码细节仍然是非常繁杂的。

本次作业将为同学们做好所有前期准备，搭好框架，提供功能的入口，而让同学们实现具体的代码，希望带领同学们初探图形编程。

而对于已经和图形代码打过一些交道、有一定经验的同学，也可以自行阅读理解 Pilot 中的图形框架代码。

二、本次作业具体内容：

1. 在 Pilot 小引擎代码中找到 pilot/engine/shader/glsl/color_grading.frag，补充此 shader 代码中的 main 函数，以实现 ColorGrading 功能。若代码编译成功且实现方法正确，则可以看到进行 ColorGrading 渲染之后的结果。
2. 使用自定义的 LUT 图，并修改相应代码，实现具有个性的 ColorGrading 的效果。
3. （提高项，可选）添加一个新的 Pass 实现某个自己感兴趣的后处理效果。

三、代码框架说明：

1. 首先引擎中有一个 PRenderPassBase 基类，它定义了每一个渲染功能所需的基本步骤；而对于一个具体的渲染功能，则另定义一个子类继承于此基类。与实现 ColorGrading 相关的所有 C++代码写在一个 PColorGradingPass 子类中，并大部分位于 color_grading.cpp 文件中。同学们对这部分代码稍作阅读即可，暂不要求掌握。

```
class PRenderPassBase
{
public:
    struct FrameBufferAttachment
    {
        VkImage        image;
        VkDeviceMemory mem;
        VkImageView    view;
        VkFormat       format;
    };

    struct Framebuffer
    {
        int           width;
        int           height;
        VkFramebuffer framebuffer;
        VkRenderPass  render_pass;

        std::vector<FrameBufferAttachment> attachments;
    };

    struct Descriptor
    {
        VkDescriptorSetLayout layout;
```

2. 程序初始化时，会调用 PColorGradingPass::initialize 进行初始化，主要是定义实现此功能需要给 Vulkan 用到的各类设置、参数和资源等。

```
namespace Pilot
{
    void PColorGradingPass::initialize(VkRenderPass render_pass, VkImageView input_attachment)
    {
        _framebuffer.render_pass = render_pass;
        setupDescriptorSetLayout();
        setupPipelines();
        setupDescriptorSet();
        updateAfterFramebufferRecreate(input_attachment);
    }
```

3. 窗口大小改变时，会调用 PColorGradingPass::updateAfterFramebufferRecreate 进行同步，主要是将新创建的 framebuffer 同步到当前 ColorGradingPass 的 DescriptorSet。

```
void PColorGradingPass::updateAfterFramebufferRecreate(VkImageView input_attachment)
{
    VkDescriptorImageInfo post_process_per_frame_input_attachment_info = {};
    post_process_per_frame_input_attachment_info.sampler =
        PVulkanUtil::getOrCreateNearestSampler(m_p_vulkan_context->_physical_device, m_p_vulkan_context->_device);
    post_process_per_frame_input_attachment_info.imageView   = input_attachment;
    post_process_per_frame_input_attachment_info.imageLayout = VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL;

    VkDescriptorImageInfo color_grading_LUT_image_info = {};
    color_grading_LUT_image_info.sampler =
        PVulkanUtil::getOrCreateNearestSampler(m_p_vulkan_context->_physical_device, m_p_vulkan_context->_device);
    color_grading_LUT_image_info.imageView =
        m_p_global_render_resource->_color_grading_resource._color_grading_LUT_texture_image_view;
    color_grading_LUT_image_info.imageLayout = VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL;

    VkWriteDescriptorSet post_process_descriptor_writes_info[2];

    VkWriteDescriptorSet& post_process_descriptor_input_attachment_write_info =
        post_process_descriptor_writes_info[0];
    post_process_descriptor_input_attachment_write_info.sType           = VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;
    post_process_descriptor_input_attachment_write_info.pNext           = NULL;
    post_process_descriptor_input_attachment_write_info.dstSet          = _descriptor_infos[0].descriptor_set;
    post_process_descriptor_input_attachment_write_info.dstBinding      = 0;
```

4. 渲染循环中，当执行到 ColorGrading 的渲染时，会调用 PColorGradingPass::draw 进行绘制。目前它是渲染循环的最后一步，在它之前的绘制步骤已经将这一帧场景的物理渲染、光照阴影、天空盒绘制以及 tonemapping 执行完毕。

```
void PColorGradingPass::draw()
{
    if (m_render_config._enable_debug_untils_label)
    {
        VkDebugUtilsLabelEXT label_info = {
            VK_STRUCTURE_TYPE_DEBUG_UTILS_LABEL_EXT, NULL, "Color Grading", {1.0f, 1.0f, 1.0f, 1.0f}};
        m_p_vulkan_context->_vkCmdBeginDebugUtilsLabelEXT(m_command_info._current_command_buffer, &label_info);
    }

    m_p_vulkan_context->_vkCmdBindPipeline(
        m_command_info._current_command_buffer, VK_PIPELINE_BIND_POINT_GRAPHICS, _render_pipelines[0].pipeline);
    m_p_vulkan_context->_vkCmdSetViewport(m_command_info._current_command_buffer, 0, 1, &m_command_info._viewport);
    m_p_vulkan_context->_vkCmdSetScissor(m_command_info._current_command_buffer, 0, 1, &m_command_info._scissor);
    m_p_vulkan_context->_vkCmdBindDescriptorSets(m_command_info._current_command_buffer,
                                                 VK_PIPELINE_BIND_POINT_GRAPHICS,
                                                 _render_pipelines[0].layout,
                                                 0,
                                                 1,
                                                 &_descriptor_infos[0].descriptor_set,
                                                 0,
                                                 NULL);

    vkCmdDraw(m_command_info._current_command_buffer, 3, 1, 0, 0);
```

5. 调用 PColorGradingPass::draw 进行绘制时，会绑定好先前初始化好的 ColorGrading Pipeline 以指导 GPU 进行渲染运算。Pipeline 中就包括设定好

的 fragment shader。本次作业我们需要补充的 color_grading.frag 就是 fragment shader 的源代码。

```
void PColorGradingPass::setupPipelines()
{
    _render_pipelines.resize(1);

    VkDescriptorSetLayout      descriptorset_layouts[1] = {_descriptor_infos[0].layout};
    VkPipelineLayoutCreateInfo pipeline_layout_create_info {};
    pipeline_layout_create_info.sType          = VK_STRUCTURE_TYPE_PIPELINE_LAYOUT_CREATE_INFO;
    pipeline_layout_create_info.setLayoutCount = 1;
    pipeline_layout_create_info.pSetLayouts    = descriptorset_layouts;

    if (vkCreatePipelineLayout(
            m_p_vulkan_context->_device, &pipeline_layout_create_info, nullptr, &_render_pipelines[0].layout) !=
        VK_SUCCESS)
    {
        throw std::runtime_error("create post process pipeline layout");
    }

    VkShaderModule vert_shader_module =
        PVulkanUtil::createShaderModule(m_p_vulkan_context->_device, POST_PROCESS_VERT);
    VkShaderModule frag_shader_module =
        PVulkanUtil::createShaderModule(m_p_vulkan_context->_device, COLOR_GRADING_FRAG);
```

6. 在 color_grading.frag 中，我们将对单个像素进行处理。已提供的数据和资源有：像素的原颜色 in_color，以及 2D 贴图采样器 color_grading_lut_texture_sampler。其中，in_color 已经在上一个 Tone Mapping Pass 中转换到了 SRGB 空间。同学们需要用它们算出像素在经过 ColorGrading 后的新颜色 out_color。可能需要用到的函数有：

1） 获得 2D 贴图大小：highp ivec2 textureSize (uniform sampler2D sampler, 0)；

2) 获取 in_color 的值 ：highp vec4 color=subpassLoad(in_color).rgba;

3) 根据位置采样 2D 贴图：highp vec4 texture (uniform sampler2D sampler, highp vec2 pos)；

```
void main()
{
    highp ivec2 lut_tex_size          = textureSize(color_grading_lut_texture_sampler, 0);
    highp float _COLORS               = lut_tex_size.y;

    highp vec4 color = subpassLoad(in_color).rgba;


    highp vec2 uv = ...;
    highp vec4 color_sampled = texture(color_grading_lut_texture_sampler, uv);
}
```

7. 我们目前使用的是 Linear 采样器。

```
void PColorGradingPass::updateAfterFramebufferRecreate(VkImageView input_attachment)
{
    VkDescriptorImageInfo post_process_per_frame_input_attachment_info = {};
    post_process_per_frame_input_attachment_info.sampler =
        PVulkanUtil::getOrCreateNearestSampler(m_p_vulkan_context->_physical_device, m_p_vulkan_context->_device);
    post_process_per_frame_input_attachment_info.imageView   = input_attachment;
    post_process_per_frame_input_attachment_info.imageLayout = VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL;

    VkDescriptorImageInfo color_grading_LUT_image_info = {};
    color_grading_LUT_image_info.sampler =
        PVulkanUtil::getOrCreateLinearSampler(m_p_vulkan_context->_physical_device, m_p_vulkan_context->_device);
    color_grading_LUT_image_info.imageView =
        m_p_global_render_resource->_color_grading_resource._color_grading_LUT_texture_image_view;
    color_grading_LUT_image_info.imageLayout = VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL;

    VkWriteDescriptorSet post_process_descriptor_writes_info[2];
```

你可能需要在 color_grading.frag 中模拟采样器的行为。

8. 我们已为你预先提供了 7 张 LUT 图。你应当修改 engine/asset/global/rendering.global.json 中的以下资源的路径，来使用你自己 LUT 图。

```
"skybox_irradiance_map": {
  "negative_x_map": "asset/texture/sky/skybox_irradiance_X-.hdr",
  "positive_x_map": "asset/texture/sky/skybox_irradiance_X+.hdr",
  "negative_y_map": "asset/texture/sky/skybox_irradiance_Y-.hdr",
  "positive_y_map": "asset/texture/sky/skybox_irradiance_Y+.hdr",
  "negative_z_map": "asset/texture/sky/skybox_irradiance_Z-.hdr",
  "positive_z_map": "asset/texture/sky/skybox_irradiance_Z+.hdr"
},
"skybox_specular_map": {
  "negative_x_map": "asset/texture/sky/skybox_specular_X-.hdr",
  "positive_x_map": "asset/texture/sky/skybox_specular_X+.hdr",
  "negative_y_map": "asset/texture/sky/skybox_specular_Y-.hdr",
  "positive_y_map": "asset/texture/sky/skybox_specular_Y+.hdr",
  "negative_z_map": "asset/texture/sky/skybox_specular_Z-.hdr",
  "positive_z_map": "asset/texture/sky/skybox_specular_Z+.hdr"
},
"brdf_map": "asset/texture/global/brdf_schilk.hdr",
"color_grading_map": "asset/texture/lut/color_grading_lut_01.jpg",
"sky_color": {
  "r": 0.53,
  "g": 0.81,
  "b": 0.98
},
```

9. （提高项，选做）我们的渲染循环是 PVulkanManager::renderFrame。其中会调用 PMainCameraPass::draw 进行延迟渲染。

```cpp
void PMainCameraPass::draw(PColorGradingPass& color_grading_pass,
                           PToneMappingPass&  tone_mapping_pass,
                           PUIPass&           ui_pass,
                           uint32_t           current_swapchain_image_index,
                           void*              ui_state)
{
    {
        VkRenderPassBeginInfo renderpass_begin_info {};
        renderpass_begin_info.sType             = VK_STRUCTURE_TYPE_RENDER_PASS_BEGIN_INFO;
        renderpass_begin_info.renderPass        = _framebuffer.render_pass;
        renderpass_begin_info.framebuffer       = m_swapchain_framebuffers[current_swapchain_image_index];
        renderpass_begin_info.renderArea.offset = {0, 0};
        renderpass_begin_info.renderArea.extent = m_p_vulkan_context->_swapchain_extent;

        VkClearValue clear_values[_main_camera_pass_attachment_count];
        clear_values[_main_camera_pass_gbuffer_a].color          = {{0.0f, 0.0f, 0.0f, 0.0f}};
        clear_values[_main_camera_pass_gbuffer_b].color          = {{0.0f, 0.0f, 0.0f, 0.0f}};
        clear_values[_main_camera_pass_gbuffer_c].color          = {{0.0f, 0.0f, 0.0f, 0.0f}};
```

你可以模仿 color_grading.cpp 和 tone_mapping.cpp 添加一个自己感兴趣的 Pass。

你可能需要调整 PMainCameraPass::setupRenderPass 添加一个新的 Vulkan Subpass。

```
void PMainCameraPass::setupRenderPass()
{
    VkAttachmentDescription attachments[_main_camera_pass_attachment_count] = {};

    VkAttachmentDescription& gbuffer_normal_attachment_description = attachments[_main_camera_pass_gbuffer_a];
    gbuffer_normal_attachment_description.format  = _framebuffer.attachments[_main_camera_pass_gbuffer_a].format;
    gbuffer_normal_attachment_description.samples = VK_SAMPLE_COUNT_1_BIT;
    gbuffer_normal_attachment_description.loadOp  = VK_ATTACHMENT_LOAD_OP_CLEAR;
    gbuffer_normal_attachment_description.storeOp = VK_ATTACHMENT_STORE_OP_DONT_CARE;
    gbuffer_normal_attachment_description.stencilLoadOp  = VK_ATTACHMENT_LOAD_OP_DONT_CARE;
    gbuffer_normal_attachment_description.stencilStoreOp = VK_ATTACHMENT_STORE_OP_DONT_CARE;
    gbuffer_normal_attachment_description.initialLayout  = VK_IMAGE_LAYOUT_UNDEFINED;
    gbuffer_normal_attachment_description.finalLayout    = VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL;

    VkAttachmentDescription& gbuffer_metallic_roughness_shadingmodeid_attachment_description =
        attachments[_main_camera_pass_gbuffer_b];
```

同时，你可能需要调整 PVulkanManager::initializeDescriptorPool 增加相关 Descriptor 的数量。

```
bool Pilot::PVulkanManager::initializeDescriptorPool()
{
    // Since DescriptorSet should be treated as asset in Vulkan, DescriptorPool
    // should be big enough, and thus we can sub-allocate DescriptorSet from
    // DescriptorPool merely as we sub-allocate Buffer/Image from DeviceMemory.

    VkDescriptorPoolSize pool_sizes[5];
    pool_sizes[0].type            = VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC;
    pool_sizes[0].descriptorCount = 3 + 2 + 2 + 2 + 1 + 1 + 3 + 3;
    pool_sizes[1].type            = VK_DESCRIPTOR_TYPE_STORAGE_BUFFER;
    pool_sizes[1].descriptorCount = 1 + 1 + 1 * m_max_vertex_blending_mesh_count;
    pool_sizes[2].type            = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
    pool_sizes[2].descriptorCount = 1 * m_max_material_count;
    pool_sizes[3].type            = VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER;
    pool_sizes[3].descriptorCount = 3 + 5 * m_max_material_count + 1 + 1 ; // ImGui_ImplVulkan_CreateDeviceObjects
    pool_sizes[4].type            = VK_DESCRIPTOR_TYPE_INPUT_ATTACHMENT;
    pool_sizes[4].descriptorCount = 4 + 1 + 1;
```

四、作业提交说明

作业的评分分为基础与提高两部分。想要完成作业同学们需达成基础评分条件，这样便视为通过作业。有能力有兴趣的同学还可以尝试达成提高项以获取更多分数。

作业提交格式：

提交作业为一个压缩文件，命名为Games104_homework2.zip或Games104_homework2.rar，其中内容为：

1. 两个文件夹shader、source，里面分别是engine/shader目录和engine/source目录下的代码；

2. Games104_homework2_report.doc或Games104_homework2_report.docx或Games104_homework2_report.pdf格式的报告文档。报告内容包含以下几点：a.ColorGrading代码的截图及实现思路讲解；b.自定义的LUT图以及将其导入后运行的结果截图；c.如果做了提高项，则需加入新Pass的代码的截图以及实现思路讲解。

打分细则：

1. 基础：[60分] 修改color_grading.frag中的代码，正确实现ColorGrading功能并成功编译运行。（参照代码框架说明6和7）

2. 基础：[20分] 在达成基础的前提下成功导入个性化的LUT图替换原LUT图。（参照代码框架说明9）

3. 提高：[20分] 添加一个新的Pass实现某个自己感兴趣的后处理效果。(参照代码框架说明9）