Lab-2

图形绘制技术

Graphic Rendering Technology

Lab2

2026 春季学期

目录

Abstract

该文档是南京大学计算机系 2026 春季学期《图形绘制技术》课程实验 2 的实验手册。

Lab2 的主要内容是在 MoerEngine 的光栅化渲染器或其他渲染器中实现一个景深（Depth of Field, DoF）后处理效果。除了最基础的景深效果，作业还要求实现模糊半径的可视化、修正前背景融合问题、使用圆盘采样核等最基础的景深优化方法。Bonus 可选部分为对景深算法进行优化、并进行数据分析。

注意，本次作业文档以 MoerEngine 实时渲染引擎为例。如果你的机器无法编译 MoerEngine，或者你熟悉其他渲染器或者游戏引擎，你也可以自行在其他渲染器中实现景深效果。

各位同学如果在 Lab2 实验过程中有任何疑问或想法，包括但不限于

- 实验框架中出现的 Bug
- 手册中叙述不完善的部分
- 对实验安排的建议
- 更好的框架设计

欢迎联系助教，帮助我们完善课程的实验部分。TAs：

- 周辰熙 QQ：1305845549 Email：chenxizhou@smail.nju.edu.cn
- 俞贤皓 QQ：2943003

同时，课程的框架代码以及实验文档也会不断更新改进，请同学们关注助教发布的相关信息。

主讲老师：过洁 Email：guojie@nju.edu.cn

1. 作业提交

1.1 分数占比

基础景深效果：50%，可视化模糊半径：10%、前背景分离：20%，圆盘采样：20%。使用任意方法对景深进行性能优化，并进行数据分析 Bonus：20%。

1.2 你需要编写的部分

- `source/editor/raster_ui/RasterUI.cpp` 的 todo 部分。该文件负责实现 UI
- `source/runtime/render/renderer/raster/RasterConfig.h` 的 todo 部分。该文件负责定义景深相关的配置项
- `source/runtime/render/renderer/raster/DofPass.h` 的 todo 部分。该文件负责调用 RHI，实现景深 Pass，定义渲染管线中景深 Pass 的输入输出，以及与引擎其他部分的接口
- `source/runtime/render/shaderheaders/shared/raster/post_process/ShaderParameters.h` 的 todo 部分。该文件负责定义 Shader 参数结构体，为 C++ 端与 Shader 端共享头文件
- `shaders/pipelines/postprocess/color/Dof.hlsl` 的 todo 部分。该文件为 Shader 代码，负责实现景深效果

1.3 提交方式

请在 智汇南雍平台 上以 PDF 格式提交你的作业报告，要求包括：

- 对于基础部分，请给出实现细节，以及渲染结果的图片
- 对于 Bonus 部分，请给出优化思路、实现细节，以及优化前后的性能数据对比。此外也需要说明优化前后的渲染结果没有明显视觉差异。
- 对实验的建议和吐槽

2. 背景

景深是一个经典的光学现象。

请阅读 虚幻引擎 Cinematic Depth of Field 文档，这篇文章写的十分优秀，有许多用于解释光路的示意图，对于理解景深的成因有着帮助。

此外，实时渲染有比较多的前置知识需要了解。本章我们将介绍一些实时渲染的基础概念，来帮助你更好地理解后续的实验内容。如果你对某些概念有疑惑，欢迎向助教提问，或自行搜索资料。

2.1 实时渲染

首先，在 lab1 中，我们使用了 moer-lite，这是一个基于路径追踪的离线渲染器。它的设计目标，便是得到一张足够真实、尽量符合物理的图像。同时，这也意味着，离线渲染器的算法开销较大，耗费的时间比较长，无法满足实时交互的需求。相比之下，实时渲染器的设计目标则是：在尽可能短的时间内，得到一张画面尽可能好的图像。通常，我们会保证实时渲染器每秒钟至少可以输出 60 帧图像，以满足观感需要、并使持续交互成为可能。

而为了达到这个目标，实时渲染器通常会使用 **图形 API** 来调用显卡，通过显卡的并行计算能力来加速渲染过程。而这也是显卡早期的主要用途：实时渲染。而图形 API 的使用十分复杂，并且不同操作系统的图形 API 也存在差异，因此，许多实时渲染器都会在应用层（代码层面）提供一层抽象。这既是为了兼容不同操作系统，也是为了降低开发者的心智负担。这层抽象就是 **RHI（Rendering Hardware Interface，渲染硬件接口）**。在本次的作业中，我们就会使用到 MoerEngine 的 RHI 接口，来实现景深效果。

2.2 Shader

CPU 和 GPU 存在着许多差异，这些差异也包括代码的形式。在 GPU 上运行的代码，通常被称为 **着色器（Shader）**。

注：如果你熟悉 CUDA 并行计算，那你可以直接把 Shader 理解为 CUDA Kernel，它们都是运行在 GPU 上的程序。此外，CUDA 和图形 API 本质上都是用于控制显卡的接口。只不过，CUDA 面向通用计算，而图形 API 则面向图形渲染。

2.3 渲染管线

渲染从宏观来看，就是一个将三维场景数据（三角形、顶点、光源、材质）转换为二维图像的过程。

而一张显示在最终屏幕上的图像，通常不是经过一次计算就得到的，而是经历了很多个步骤。这一整套步骤就被称为 **渲染管线**。在实时渲染中，渲染管线通常分为多个阶段：

1. 先根据场景中的模型、光源、材质等数据，计算出一张初始画面
2. 再对这张画面进行一些额外的处理，例如添加阴影、反射、泛光、景深

3. 最后把处理后的画面显示到屏幕上

而本次作业中需要实现的 **景深**就属于第二类，也就是 **后处理效果**。所谓后处理，就是：输入是一张已经渲染好的全屏图像，输出一张全屏图像。

因此，本次作业中，我们要编写的 Shader 也较为基础。我们可以将编写的 Shader 代码，看作一个函数 $f(x, y, color, S)$，其中 x, y 是当前像素的位置，$color$ 是当前像素的颜色，S 是一些额外的输入（例如深度信息）。这个函数的输出，是一个新的颜色值。也就是说，我们编写的 Shader 代码，决定了：**对于每个像素，我们应该如何根据它的位置、颜色、深度等信息，计算出它最终应该显示成什么样子**。。

举个例子，我们直接假设 $f(x, y, color, S) = color * 0.5$，那么这个 Shader 的效果就是：把每个像素的颜色都变暗一半。我们将在下文实现一个类似的效果。

2.4 相机模型与景深

景深（Depth of Field, DoF）是一个经典的光学现象。简单来说，相机并不是把所有距离的物体都拍得一样清楚，而是只能让某一部分距离附近的物体最清晰。这个“最清楚的位置”可以理解为 **焦平面**。如果一个物体不在焦平面上，那么它在成像时就不会收缩成一个理想的点，而会扩散成一个小圆斑。

通常，在拍照时，我们都会通过 **对焦**来调整焦平面的位置，使得拍摄目标物位于焦平面上，足够清晰。而不在焦平面上的物体，则会受到虚化（模糊）的影响。这些物体距离焦平面越远，虚化效果越强。

在实时渲染中，我们很难像离线渲染那样，去完整地模拟整个光学系统。因此，我们会做大量的 **近似与化简**。对于景深来说，我们通常的做法是：**计算每个像素距离焦平面的距离，然后模糊这个像素**。

在实时渲染业界中，最成熟的景深算法并没有什么特别巧妙的思想，而是巨量工程 trick 的堆叠。如果你对此有兴趣，可以参考本章开头提到的虚幻引擎文档和虚幻引擎源码。而在本次作业中，我们只需要实现一个最基础的景深效果即可。

3. MoerEngine

本章我们将介绍 MoerEngine 的编译流程、引擎的基本使用方法。

值得注意的是，MoerEngine 目前仅支持 Windows 系统，并且不兼容早期显卡（因为引擎强制启用光追特性，所以此处推荐使用 NVIDIA RTX 系列显卡）。如果你的电脑无法编译和正常运行 MoerEngine，你也可以自行在其他渲染器（如 Unity、虚

幻引擎）中实现景深效果。如果你选择其他引擎，请提供编写的代码和截图，以说明你没有使用引擎自带的景深功能。

3.1 编译引擎

这部分可以参考代码仓库中的构建手册 `docs/BUILD.md`，本章也会详细地介绍一遍编译流程。

我们需要安装以下工具：

- Git (下载链接)：用于从 GitHub 上克隆 MoerEngine 的代码仓库
- CMake (下载链接)：用于生成构建系统文件
- Clang (LLVM) (下载链接)：用于编译 C++ 代码
- ninja (下载链接)：一个高效的构建系统，配合 CMake 使用
- Vulkan SDK (下载链接)：提供 Vulkan API 的开发工具和库

Git、CMake、Vulkan SDK 都提供了安装程序，按照安装程序执行即可，Clang 和 ninja 则需要手动配置环境变量。

Clang：

- 进入 Github Releases 页面：(下载链接)
- 选择 LLVM-22.x.x-win64.exe，下载并根据指引安装
- 安装完毕后，添加 /path/to/LLVM/bin（即你的安装目录下的 bin 文件夹）到系统环境变量 Path 中
- 在命令行输入 clang –version，如果能正确显示版本信息，则说明安装成功

ninja：

- 进入 Github Releases 页面：(下载链接)
- 选择 ninja-win.zip，下载，并解压到任意的一个目录
- 安装完毕后，添加 /path/to/ninja（即上文你选择的目录）到系统环境变量 Path 中
- 在命令行输入 ninja –version，如果能正确显示版本信息，则说明安装成功

环境全部安装完毕后，在命令行中执行以下代码，需要能够正常看到版本信息。

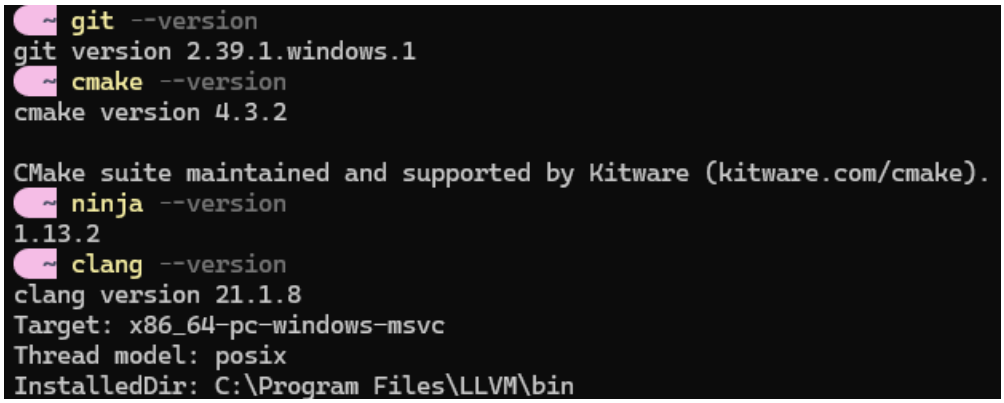


图 1: 环境示例截图

接着，我们既可以开始克隆 MoerEngine 的仓库，并进行编译了。你可以直接在一个目录下打开命令行，并按顺序执行以下命令：

```
# Clone仓库

# -b lab/lab2-dof：切换到lab/lab2-dof分支，该分支提供了代码框架
# --recurse-submodules：递归地克隆submodule
# --shallow-submodules：对子模块也进行浅克隆，节省空间和时间
# 如果在clone时忘记添加submodule相关参数，请执行git submodule update --init --depth 1
git clone -b lab/lab2-dof --recurse-submodules --shallow-submodules https://github.com/NJUCG/
    MoerEngine.git

cd MoerEngine

# 根据模板创建一份MoerEngine的配置文件
# 配置文件中，可以设置默认的渲染器（光栅化、光追）、默认分辨率等选项
cp template.MoerEngine.toml MoerEngine.toml

# 构建
cmake -B build -G Ninja -DCMAKE_C_COMPILER=clang -DCMAKE_CXX_COMPILER=clang++
cmake --build build -j16 # 修改16为你的cpu核心数或略低于你的cpu核心数

# 运行
./target/bin/Debug/MoerEditor.exe
```

理想的话，我们就可以成功编译并启动引擎了。

3.2 熟悉引擎

启动引擎后，你可以看到如下页面。其中，左侧是 场景颜色，右侧是 配置页。

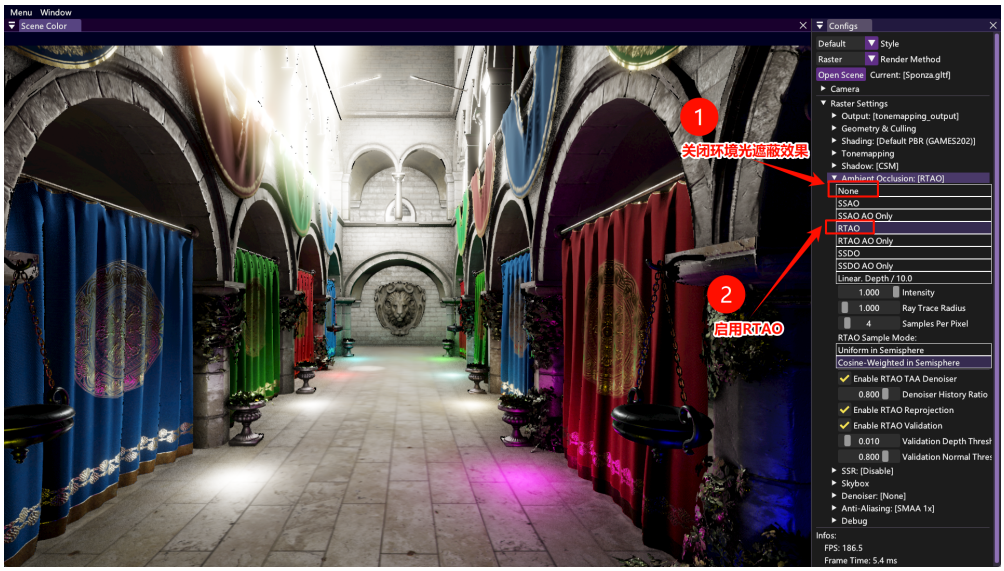


图 2: 光栅化渲染器渲染效果。右侧是配置面板

选择配置页的右上角 **Render Method**，可以切换引擎使用的渲染方法。目前共支持两种渲染方法：**Raytracing 光线追踪**和 **Rasterization 光栅化**。你可以选择 Render Method，并切换为 *Raytracing*。我们的实验在 Rasterization 光栅化模式下进行，所以在实验时请确保右上角的 Render Method 为 Raster。

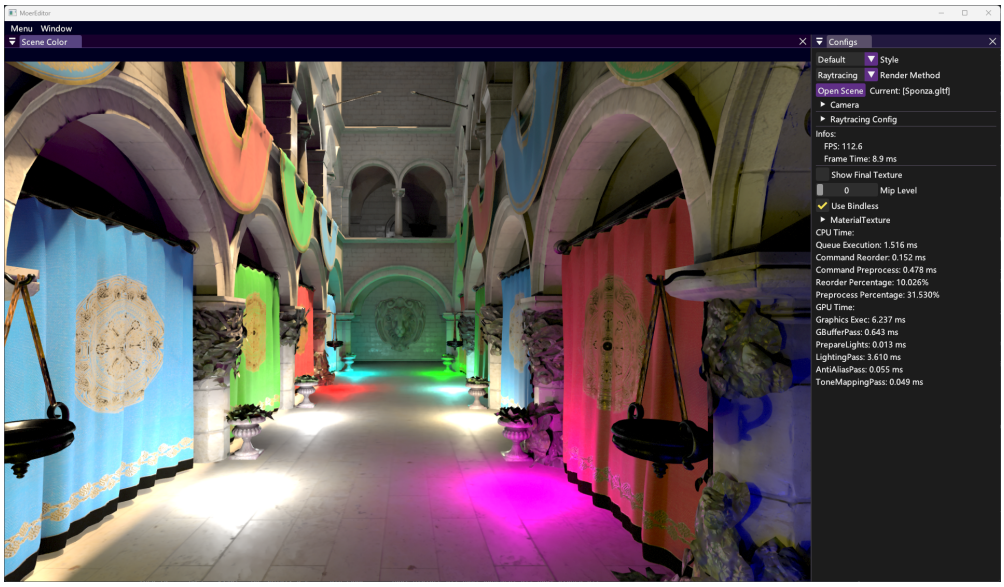


图 3: 启用 NRD 的光线追踪渲染器渲染效果

点击右侧的 `RasterSettings` - `AmbientOcclusion`，选择不同的环境光遮蔽算法，查看对应效果。如果 MoerEngine 光栅化在你的电脑上帧数比较低，可以将环境光遮蔽算法切换为 *SSAO*。

目前光栅化渲染器，默认启用 RTAO、自动曝光等后处理算法。你可以修改这些

算法的参数，查看对应的效果。

3.3 配置文件

在引擎的根目录下，有配置文件 `MoerEngine.toml`。

如果你是 1080p 屏幕，发现引擎每次启动都会占满整个桌面，那么可以在配置文件中修改引擎默认分辨率，即 `editor.width` 和 `editor.height` 字段。

如果你觉得光栅化渲染器帧数太低，可以修改 `engine.render.raster.low_quality_mode` 字段。

每次编译时，这个配置文件会被自动拷贝到可执行文件同目录下，从而在执行时生效。（你也可以直接修改可执行文件目录下的同名配置文件）

3.4 IDE 配置

- VSCode / Cursor：在插件商城安装 **clangd**

4. 基础景深效果

我们首先实现一个最基础的单 Pass 景深。

由于引擎规模较大，文件数量多，所以请善用 IDE 的搜索功能。例如，可以在 VSCode 中按下 Ctrl+P，输入文件名，来快速定位文件。

4.1 后处理 Pass

渲染管线是由非常多 Pass 组成的，在 `RasterRenderer.cpp` 中，可以找到如下代码：

```
// Post Process Passes
// - Ambient Occlusion
auto              ao_result        = ao_pass->Process(raster_context, raster_config, camera, time
    );
TextureWithHandle processing_image = ao_result.ao_with_color;
uint              ao_only_idx      = ao_result.ao_only_idx;

rtao_denoiser_pass->ProcessInPlace(raster_context, raster_config, ao_only_idx);

// - Denoiser Pass (Bilateral Filter)
processing_image = bfd_pass->Process(raster_context, raster_config, processing_image);

// - Screen Space Reflection
processing_image = ssr_pass->Process(raster_context, raster_config, camera, processing_image);

// - Anti-aliasing
processing_image = aa_pass->Process(raster_context, raster_config, camera, processing_image);
```

```
17
18 // - Bloom Pass
19 processing_image = bloom_pass->Process(raster_context, raster_config, processing_image);
20
21 // - Tonemapping Pass
22 processing_image = tonemapping_pass->Process(raster_context, raster_config, processing_image);
```

上述所有 Pass 都是 **后处理 Pass**。即，这个 Pass 的输入是全屏像素，输出也是全屏像素，这个 Pass 的 Shader 会对像素进行一些处理。例如，`ao_pass->Process(..)` 就是 环境光遮蔽效果。

4.2 添加一个后处理 Pass

熟悉一个项目最好的方式，就是直接上手改。

为了让你更好地理解引擎的结构，接下来的操作步骤默认在主分支上执行。这样可以确保，你可以在一个没有框架的主分支上，完成从无到有的景深 Pass 实现；或者添加一个自己的、新的 Pass。

我们也提供了一个景深算法的实现框架，你可以将代码切换到 `lab/lab2-dof` 分支（如果你遵循了本文的编译步骤，则引擎默认在 lab/lab2-dof 分支下）。这样，你就可以 忽略接下来的许多操作，不需要自己新建文件，只需要完成（复制并黏贴）TODO 部分的代码，即可完成最基础的景深效果。

本章需要大量复制代码，而 PDF 格式复制出的代码会丢失缩进等格式信息。所以，你可以访问引擎代码仓库的 `docs/DofExample.md` 来阅读本章节并复制代码。

以下是操作步骤：

1. 创建 Pass 源码

- 在 `source\runtime\render\renderer\raster\` 目录下，创建 `DofPass.h`
- 在 `DofPass.h` 内黏贴以下内容：

```
1 #pragma once
2
3 #include "scene/camera/Camera.h"
4 #include "shader/ShaderPipeline.h"
5 #include "shaderheaders/shared/raster/post_process/ShaderParameters.h"
6
7 #include "RasterConfig.h"
8 #include "RasterResource.h"
9 #include "RasterTool.h"
10
11 namespace Moer::Render::Raster {
12
13 class DofPipeline : public RasterPipeline {
14 public:
15     DEFINE_RASTER_PIPELINE_CLASS(DofPipeline);
```

```
    DEFINE_SHADER_CONSTANT_STRUCT(DofPipelineBindlessParam, param);
    DEFINE_SHADER_BINDLESS_ARRAY(bdls);
    DEFINE_SHADER_ARGS(bdls, param);
};

class DofPass {
public:
    DofPass(RasterContext& context) {
        GfxPsoCreateInfo pso_full_screen_info(
            RHIRasterizeInfo::Preset(),
            {},
            {RHIColorAttachmentInfo::Preset(context.textures.dof_output.tex->GetFormat())}
        );

        m_dof_pipeline = context.manager.Raster()
                             .Vertex("core/utils/FullScreenQuad.hlsl")
                             .Pixel("pipelines/postprocess/color/Dof.hlsl")
                             .Build<DofPipeline>(std::move(pso_full_screen_info));
    }

    TextureWithHandle Process(
        RasterContext&      context,
        const RasterConfig& ui_config,
        const Camera&       camera,
        TextureWithHandle   input_image
    ) {

        DofPipelineBindlessParam param;

        float2 resolution = float2(input_image.GetSize());

        param.resolution      = resolution;                                     // 分辨率
        param.resolution_inv  = float2(1.f / resolution.x, 1.f / resolution.y); // 分辨率倒数
        param.input_color_tex = input_image.hdl;                                // 输入颜色纹理
        param.debug_param     = 0.5f;                                           // 调试参数

        context.cmd_list.Gfx(m_dof_pipeline, context.bdls, param)
            .Draw(
                "Dof Pass",
                context.textures.dof_output.GetRect2D(),
                std::move(RasterTool::GetFullScreenDrawDatas()),
                ColorAttachment(context.textures.dof_output.tex)
            );

        return context.textures.dof_output;
    }

private:
    DofPipeline m_dof_pipeline;
};

} // namespace Moer::Render::Raster
```

2. 定义 Shader 参数

- 打开 source\runtime\render\shaderheaders\shared\raster\post_process\ShaderParameters.h

- 在 structSsrPipelineBindlessParam{...}; 后添加如下代码:

```
// 1. 该struct中的数据都会被传到Shader中
// 2. 该struct是C++端和Shader端(HLSL)共享的，所以我们只需要定义一次，就可以在两个语言中共享。
// 3. 该struct通过PushConstants传入Shader，适合存储每帧变化的数据
struct DofPipelineBindlessParam {
    float2 resolution;      // 分辨率
    float2 resolution_inv;  // 分辨率倒数
    uint   input_color_tex; // 输入颜色纹理handle
    uint   depth_tex;       // 深度纹理handle（预留）
    float  debug_param;     // 调试参数
    float  near_clip;       // 摄像机近裁剪面（预留）
    float  far_clip;        // 摄像机远裁剪面（预留）
    float  dof_intensity;   // DOF强度（预留）
    float  focus_plane_distance;   // 焦平面距离（预留）
    float  focus_plane_range;      // 焦平面范围（预留）
    uint   b_visualize_focus_plan; // 可视化焦平面（预留）
};
```

3. 创建纹理

- 打开 source\runtime\render\renderer\raster\RasterTextures.h
- 在 X(TexHandle,ssr_output,....)\ 后添加如下代码:

```
// 这是一个宏定义，会根据上下文的工具代码，自动创建纹理的申请、重建、释放代码
// dof_output:              纹理变量名
// Tex2DTag:                纹理类别
// PF_R16G16B16A16_SFLOAT:  纹理像素格式
// E_SAMPLED_COLOR:         纹理用途为 SAMPLED | COLOR_ATTACHMENT
X(TexHandle, dof_output, Tex2DTag, TexConfig::Default(PF_R16G16B16A16_SFLOAT).Usage(
    E_SAMPLED_COLOR))    \
```

4. 创建 Shader

- 在 shaders\pipelines\postprocess\color\ 目录下，创建 Dof.hlsl
- 在 Dof.hlsl 中黏贴:

```
#include "core/common/Bindless.hlsl"
#include "core/common/Common.hlsl"
BINDLESS_BINDINGS(3, 2, 4, 5)
#include "materials/Material.hlsli"
#include "shared/raster/ShaderParameters.h"

[[vk::push_constant]] ConstantBuffer<Moer::DofPipelineBindlessParam> param;

float4 main(float2 uv : TEXCOORD0) : SV_TARGET {
    // uv 即 屏幕坐标，值域为[0, 1]，表示了不同的像素

    float3 color       = TextureHandle(param.input_color_tex).Sample2D<float4>(uv).rgb;
    float3 debug_color = float3(1.0, 0.0, 1.0);

    color = lerp(color, debug_color, param.debug_param);

    return float4(color, 1.0);
}
```

5. 使用 Pass

- 打开 RasterRenderer.h，添加以下代码：

```
// 1. 前向声明
//    前向声明是为了加快编译速度，避免对应类型头文件被整体引入到当前文件中，从而导致项目编译时长增加

class SsrPass; // 已有代码
class DofPass; // 【新增代码】

// 2. Pass对象 声明
//    UniquePtr就是std::unique_ptr。这里使用指针，是为了兼容前向声明

UniquePtr<SsrPass> ssr_pass; // 已有代码
UniquePtr<DofPass> dof_pass; // 【新增代码】
```

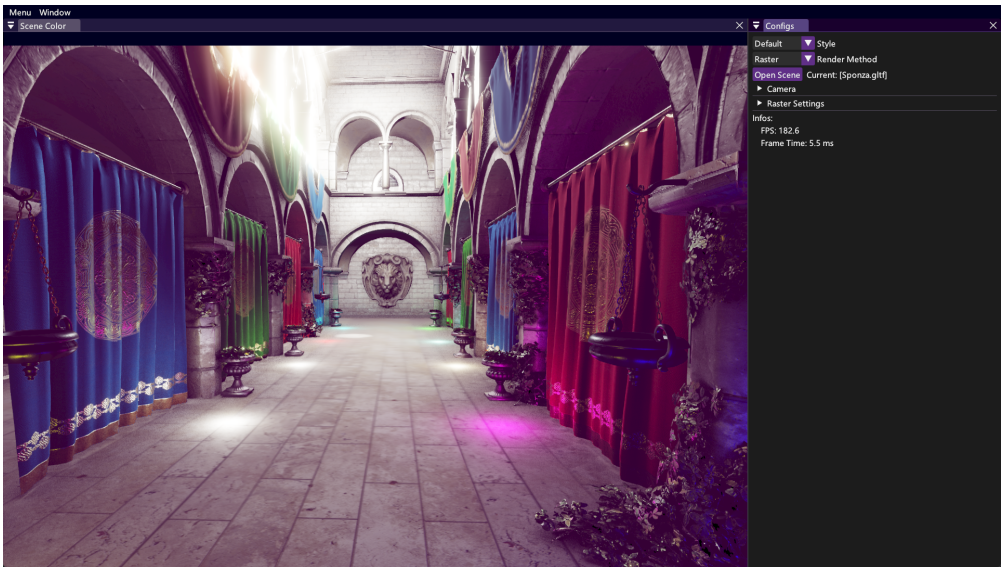
- 打开 RasterRenderer.cpp，添加以下代码：

```
// 1. 头文件

#include "SsrPass.h" // 已有代码
#include "DofPass.h" // 【新增代码】

// 2. Pass对象 初始化

ssr_pass = MakeUnique<SsrPass>(raster_context); // 已有代码
dof_pass = MakeUnique<DofPass>(raster_context); // 【新增代码】

// 3. Pass对象 执行

// - Screen Space Reflection
processing_image = ssr_pass->Process(raster_context, raster_config, camera, processing_image); // 已有代码

// - Depth of Field
processing_image = dof_pass->Process(raster_context, raster_config, camera, processing_image); // 【新增代码】
```

6. 重新编译与运行

```
cmake --build build -j16
./target/bin/Debug/MoerEditor.exe
```

- 你应该会看到如下画面（整个屏幕变紫）



将原图和紫色按照 50% 和 50% 的比例进行混合，得到的图像

回顾一下我们刚刚执行的核心代码：

```
// 从input_color_tex这个handle，获取上一个Pass的输出画面
float3 color = TextureHandle(param.input_color_tex).Sample2D<float4>(uv).rgb;

// 定义紫色
float3 debug_color = float3(1.0, 0.0, 1.0);

// 根据debug_param来混合上一帧的画面和紫色
color = lerp(color, debug_color, param.debug_param);
```

可以发现，这个效果是符合预期的。你可以修改 `DofPass.h` 中的 `param.debug_param=0.5f`，将 `0.5f` 修改为 `0.9f`，那么画面会接近全紫色。

这个时候，我们已经可以进行一些比较有意思的修改了。比如，我们想实现一个**检测边缘的十字算子**（就是机器学习课里学到的东西）。那么，我们就可以在 Shader 中对输入图像进行采样，并且实现这个功能。

下面这段代码就是 Cursor 编写的一个检测边缘 Shader。**你可以直接将 Dof.hlsl 的 main 函数替换为如下代码**：

```
// 可以直接替换原来的main函数
float4 main(float2 uv : TEXCOORD0) : SV_TARGET {

    TextureHandle input_texture = TextureHandle(param.input_color_tex);

    // uv 即 屏幕坐标，值域为[0, 1]，表示了不同的像素
    // 这里我们获取了屏幕分辨率的倒数，通过它来计算相邻像素的 uv坐标差值
    // 你可以注释掉下面这行代码，来可视化 uv
    // return float4(uv, 0.0, 1.0); // 很神奇吧!
    float2 delta_uv = param.resolution_inv;

    // 当前像素
    float3 center_color = input_texture.Sample2D<float4>(uv).rgb;
```

```
    // 左侧像素
    float3 left_color = input_texture.Sample2D<float4>(uv + float2(-delta_uv.x, 0.0)).rgb;
    // 右侧像素
    float3 right_color = input_texture.Sample2D<float4>(uv + float2(delta_uv.x, 0.0)).rgb;
    // 上方像素
    float3 top_color = input_texture.Sample2D<float4>(uv + float2(0.0, -delta_uv.y)).rgb;
    // 下方像素
    float3 bottom_color = input_texture.Sample2D<float4>(uv + float2(0.0, delta_uv.y)).rgb;

    // 十字方向梯度 (edge detection) : 简单横向 + 纵向差分
    float3 horizontal_gradient = right_color - left_color;
    float3 vertical_gradient   = bottom_color - top_color;

    // 用长度表示亮度
    float edge_strength = length(horizontal_gradient) + length(vertical_gradient);

    float3 edge_color = edge_strength.xxx;

    return float4(edge_color, 1.0);
}
```

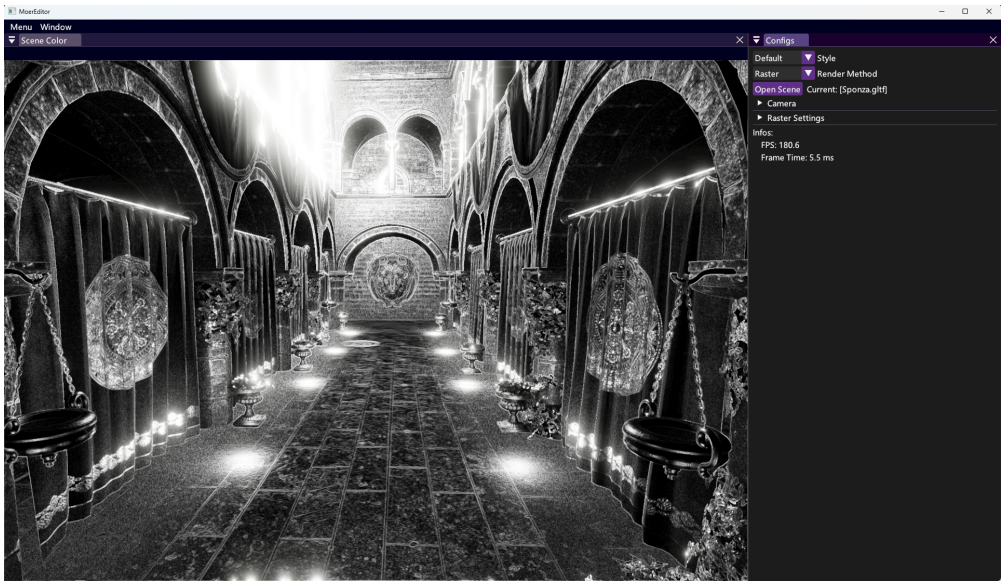
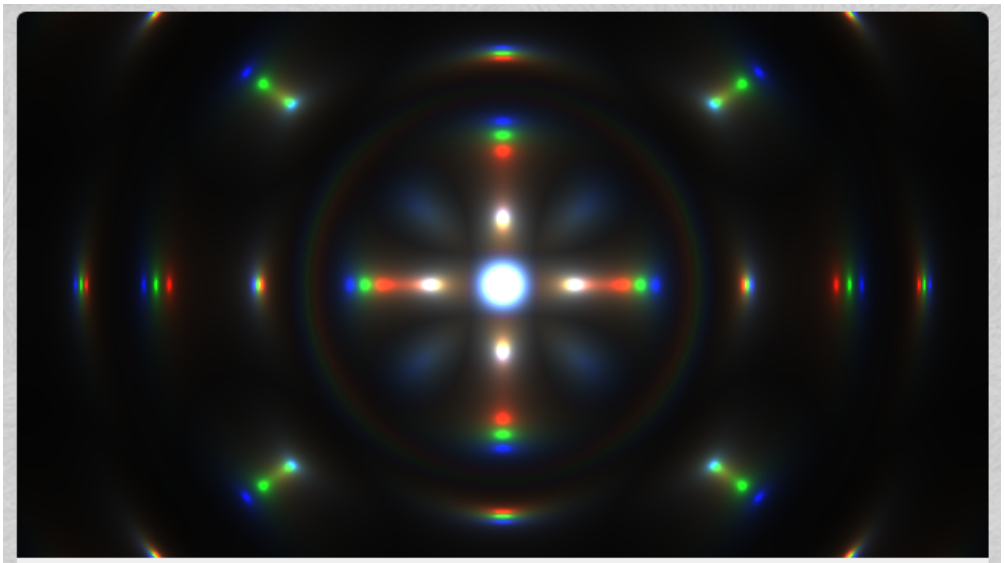


图 4: 边缘检测的十字算子

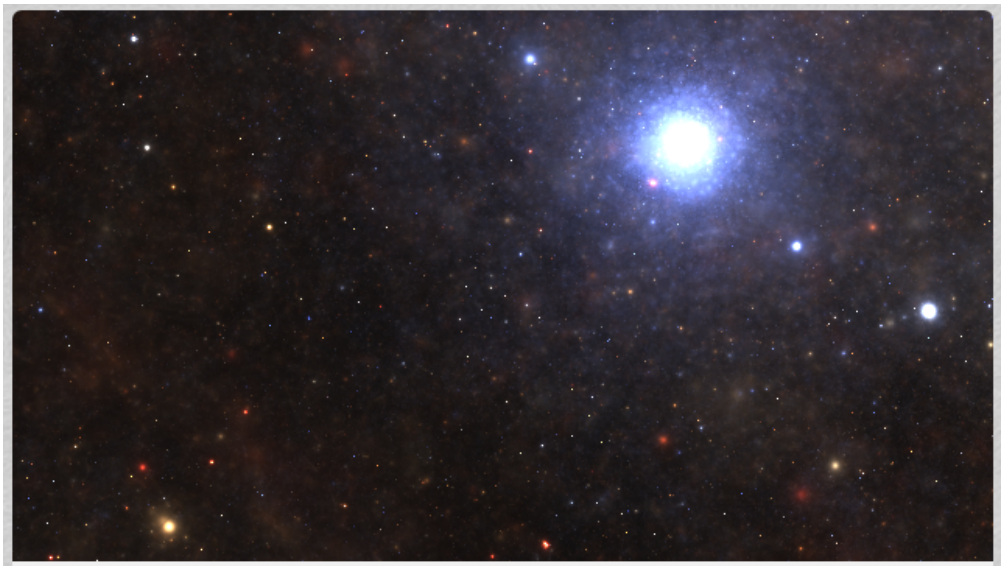
4.2.1 ShaderToy

Shader 可以实现无数种有趣、漂亮的动态效果。你可以在 ShaderToy 这个网站上找到许多 Shader。此处贴几个笔者非常喜欢的 Shader，你可以点进链接，直接查看这些动画效果和对应的代码。

13 行代码实现炫光，https://www.shadertoy.com/view/XsXXDn



渲染星空背景的 Shader，https://www.shadertoy.com/view/XlfGRj



在全屏 Pass 中渲染 3D 图形，https://www.shadertoy.com/view/NtlSDs

以上的效果，在任意一个渲染器的 Shader 中都可以轻松实现，MoerEngine 也不例外。

我们只需要再传入一些参数，例如系统时间、鼠标位置等数据，就可以将这些效果搬到引擎里。如果你想尝试的话，看完下一节之后，就可以自行进行迁移，本文不再赘述。（当然你也可以直接让 AI 迁移）

4.3 UI 控制渲染参数

如果每次修改一个参数，我们都要重启引擎，那就太麻烦了！所以，在运行时调整参数，是一个非常必要的功能。

接下来，我们将介绍如何在 MoerEngine 中用 UI 控制渲染参数：

1. 在 Config 中创建参数

- 打开 `source\runtime\render\renderer\raster\RasterConfig.h`，添加以下代码：

```
float ssr_metallic_threshold         = 0.5;   // 已有代码
float ssr_step_base                  = 0.025; // 已有代码

// MARK: Dof
float dof_debug_param = 0.5f; //  【新增代码】
float dof_intensity   = 1.0f; // DOF强度【新增代码】
float focus_plane_distance = 5.0f; // 焦平面距离【新增代码】
float focus_plane_range    = 1.0f; // 焦平面范围【新增代码】
uint  b_visualize_focus_plan = 0; // 可视化焦平面【新增代码】
```

2. 创建 UI 代码

- 打开 `RasterUI.cpp`，添加一下代码：

```
// MARK: SSR
if (ImGui::TreeNode("SSR", /* ...略 */)) { // 已有代码
    // ...略                                // 已有代码
}                                           // 已有代码

// 以下均为【新增代码】
// MARK: Dof
if (ImGui::TreeNode("Dof (Depth of Field)")) {
    ImGui::SliderFloat("Debug Param", &m_config.dof_debug_param, 0.0f, 1.0f);
    ImGui::SliderFloat("Intensity", &m_config.dof_intensity, 0.0f, 3.0f);
    ImGui::SliderFloat("Focus Plane Distance", &m_config.focus_plane_distance, 0.0f, 20.0f);
    ImGui::SliderFloat("Focus Plane Range", &m_config.focus_plane_range, 0.0f, 5.0f);

    bool visualize_focus_plane = m_config.b_visualize_focus_plan != 0;
    if (ImGui::Checkbox("Visualize Focus Plane", &visualize_focus_plane)) {
        m_config.b_visualize_focus_plan = visualize_focus_plane ? 1u : 0u;
    }
    ImGui::TreePop();
}
```

3. 修改 DofPass.h

- 打开 DofPass.h，将 param.debug_param=0.5f; 修改为 param.debug_param=ui_config.dof_debug_param;

4. 重置 Shader 代码（可选）

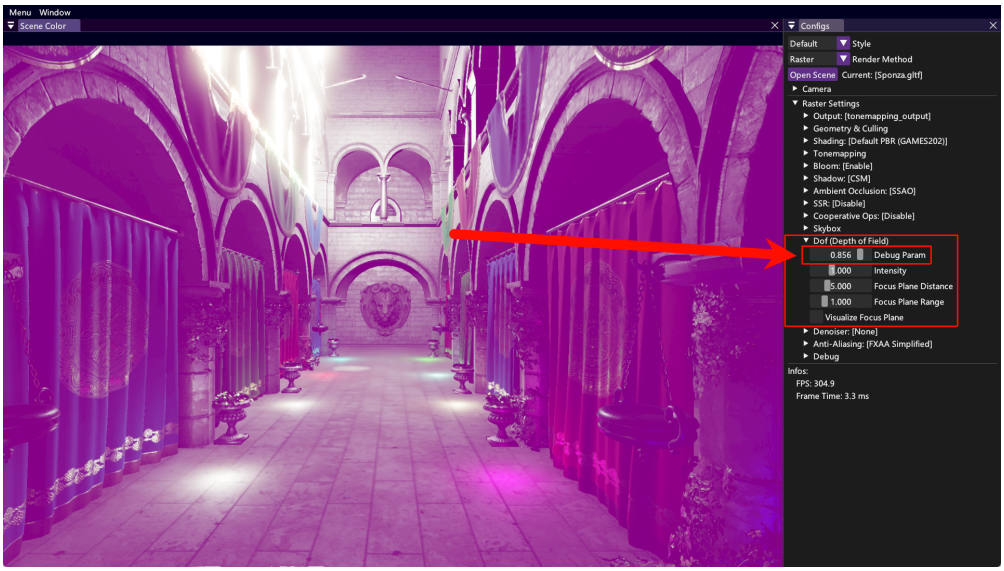
- 如果你没有黏贴边缘过滤的十字算子代码，则可以跳过这一步
- 打开 Dof.hlsl，将 main 函数修改为：

```
float4 main(float2 uv : TEXCOORD0) : SV_TARGET {
    // uv 即 屏幕坐标，值域为[0, 1]，表示了不同的像素

    float3 color       = TextureHandle(param.input_color_tex).Sample2D<float4>(uv).rgb;
    float3 debug_color = float3(1.0, 0.0, 1.0);

    color = lerp(color, debug_color, param.debug_param);

    return float4(color, 1.0);
}
```

5. 重新编译并运行

```
cmake --build build -j16
./target/bin/Debug/MoerEditor.exe
```

- 你应该会看到如下选项



4.4 获取深度信息

接下来，我们已经了解了景深所需要的、最基本的代码操作。接下来，让我们开始实现景深效果。

首先，每个像素的虚化程度是根据如下公式计算得到：

$$\text{虚化程度}(COC, Circle\ of\ Confusion) = \frac{\text{像素到摄像机的距离} - \text{焦平面距离}}{\text{某参数}}$$

所以，我们需要在 Shader 中获取深度信息。

我们之前已经在 `DofPipelineBindlessParam` 中预留了 `depth_tex`，此处我们直接传入渲染管线计算好的深度纹理信息：

- 打开 `DofPass.h`，添加：

```
param.input_color_tex = input_image.hdl; // 已有代码
param.debug_param     = ui_config.dof_debug_param; // 已有代码

param.near_clip = camera.GetNearClip(); // 摄像机近裁剪面【新增代码】
param.far_clip  = camera.GetFarClip(); // 摄像机远裁【新增代码】
param.depth_tex = context.textures.depth_linear_sampler.hdl; // 深度纹理【新增代码】
param.dof_intensity = ui_config.dof_intensity; // DOF强度【新增代码】
param.focus_plane_distance = ui_config.focus_plane_distance; // 焦平面距离【新增代码】
param.focus_plane_range = ui_config.focus_plane_range; // 焦平面范围【新增代码】
param.b_visualize_focus_plan = ui_config.b_visualize_focus_plan; // 可视化焦平面【新增代码】
```

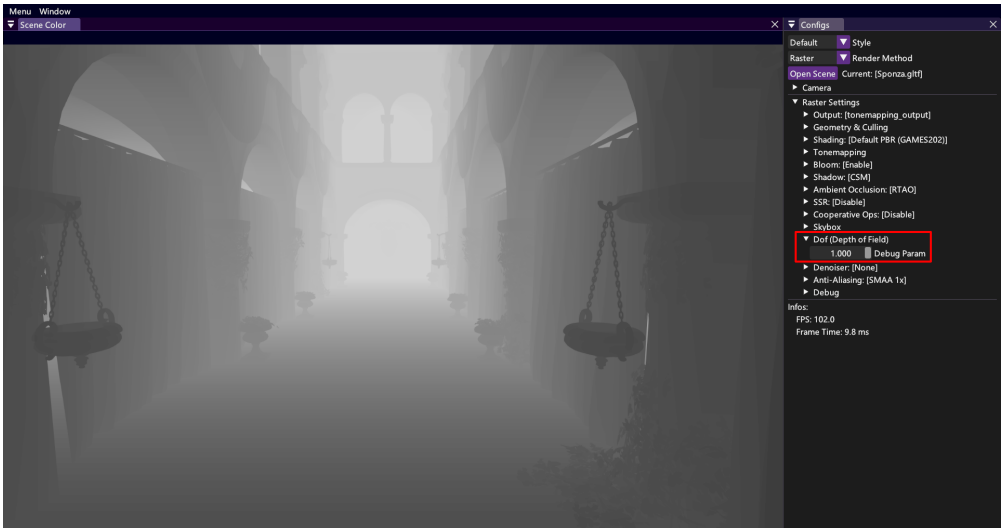
- 打开 `Dof.hlsl`，替换 main 函数为以下代码：

```
float4 main(float2 uv : TEXCOORD0) : SV_TARGET {
    // uv 即 屏幕坐标，值域为[0, 1]，表示了不同的像素
```

```
    float3 color = TextureHandle(param.input_color_tex).Sample2D<float4>(uv).rgb;

    // 深度：值域为 [0, 1]，近处为1，无限远处为0
    // - 另外，MoerEngine使用了Reverse-Z技术。如果你接触过其他渲染器，你会发现此处0和1的对应关系跟通常渲染器不同
    float depth = TextureHandle(param.depth_tex).Sample2D<float>(uv);

    // 线性化深度：值域为 [near_clip, far_clip]，单位为世界坐标距离
    float linearized_depth = param.near_clip * param.far_clip / (param.far_clip + (1.0 - depth) * (param.near_clip - param.far_clip));

    // 将深度直接映射为颜色
    float3 depth_color = linearized_depth;

    // 将深度混合到颜色中，debug_param控制混合程度
    color = lerp(color, depth_color, param.debug_param);

    return float4(color, 1.0);
}
```

- 重新编译并运行

  然后，将 DebugParam 混合参数调整为 1.0，你可以看到如下画面：



这就是我们场景的深度值，你也可以理解为 **每个点到摄像机的距离信息**。

4.5 计算景深效果

接下来，我们开始正式实现景深效果。

在前面的教程中，我们预留了许多 UI 参数，接下来我们将直接使用这些参数：

1. DOF Intensity：景深强度
2. Focus Plane Distance：焦平面距离

3. Focus Plane Range：焦平面范围

4. Visualize Focus Plane：一个 bool 值，勾选后则会在引擎中可视化焦平面区域

我们修改 `Dof.hlsl` 的 main 函数为如下代码：

```
float4 main(float2 uv : TEXCOORD0) : SV_TARGET {
    // uv 即 屏幕坐标，值域为[0, 1]，表示了不同的像素

    float3 color = TextureHandle(param.input_color_tex).Sample2D<float4>(uv).rgb;

    // 深度：值域为 [0, 1]，近处为1，无限远处为0
    // - 另外，MoerEngine使用了Reverse-Z技术。如果你接触过其他渲染器，你会发现此处0和1的对应关系跟通常渲染器不同
    float depth = TextureHandle(param.depth_tex).Sample2D<float>(uv);

    float coc = 0.0f;
    float focus_range = max(param.focus_plane_range, 1e-3);

    if (depth <= 1e-4) {
        // 如果当前像素是天空（无限远）
        coc = 1e10; // 一个非常大的值，表示完全虚化

    } else {
        // 正常情况

        // 线性化深度：值域为 [near_clip, far_clip]，单位为世界坐标距离
        float linearized_depth = param.near_clip * param.far_clip / (param.far_clip + (1.0 - depth) * (param.near_clip - param.far_clip));

        // 焦平面距离
        float focus_depth = clamp(param.focus_plane_distance, param.near_clip, param.far_clip);
        // 距离差值
        float signed_delta = linearized_depth - focus_depth;

        float coc_magnitude = max(abs(signed_delta) - focus_range, 0.0) / focus_range * param.dof_intensity;

        // 焦平面附近 coc = 0，背景 coc > 0，前景 coc < 0
        coc = sign(signed_delta) * coc_magnitude;
    }

    // 根据coc计算模糊半径，最大为6像素
    int blur_radius = min(max((int)round(abs(coc)), 0), 6);

    // 根据coc虚化像素
    float3 blur_color = color;
    if (blur_radius > 0) {
        blur_color = 0.0;
        float sample_count = 0.0;
        // 模糊，本质上就是获取周围像素的颜色，然后求平均值（卷积）
        // 下面的代码就是在以当前像素为中心，半径为blur_radius的范围内，采样颜色并求平均值
        [loop]
        for (int y = -blur_radius; y <= blur_radius; ++y) {
            [loop]
            for (int x = -blur_radius; x <= blur_radius; ++x) {
                float2 offset = float2(x, y) * param.resolution_inv;
                blur_color += TextureHandle(param.input_color_tex).Sample2D<float4>(saturate(uv +
```

```
        offset)).rgb;
50                  sample_count += 1.0;
51              }
52          }
53          blur_color /= sample_count;
54      }
55      color = blur_color;
56
57      // 可视化焦平面
58      if (param.b_visualize_focus_plan != 0) {
59          if (abs(coc) <= focus_range) {
60              color = color; // do nothing
61          } else if (coc > 0.0) {
62              color = lerp(color, float3(0.0, 0.0, 1.0), 0.7);
63          } else {
64              color = lerp(color, float3(0.0, 1.0, 0.0), 0.7);
65          }
66      }
67
68      return float4(color, 1.0);
69  }
```

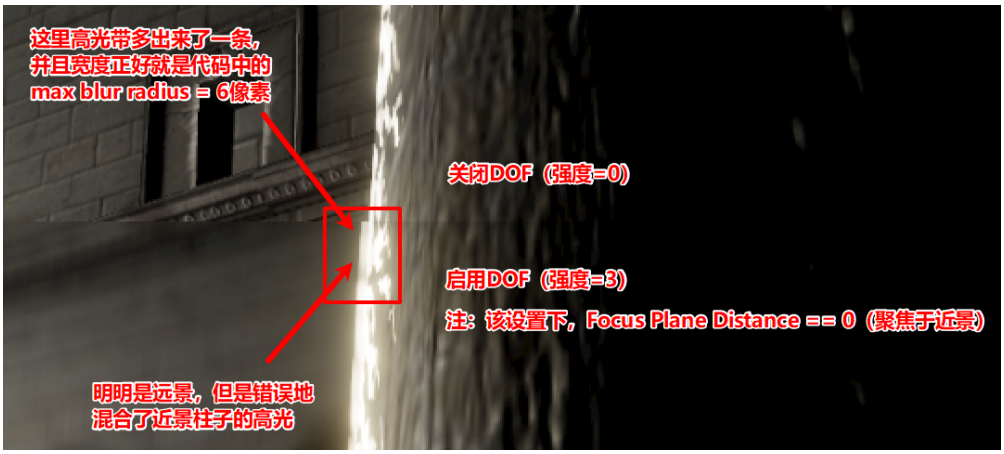
重新编译并运行引擎，你将得到如下画面：

至此，一个简单的景深效果便实现完毕。

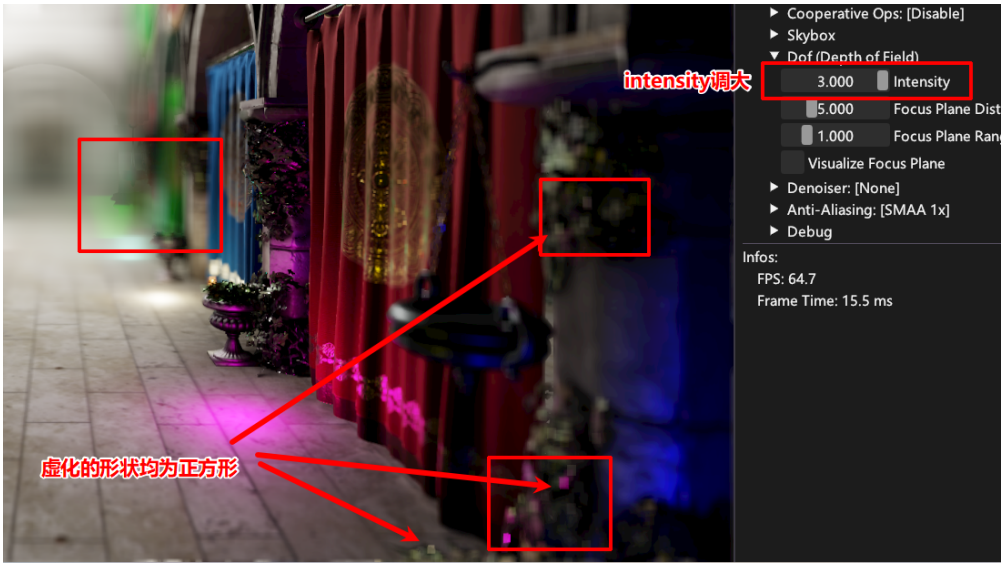
4.6 缺陷与优化

此处只实现了一个最基础的景深效果。目前的效果既不符合相机物理参数、视觉上也存在许多缺陷、性能还十分差劲：

- 我们目前在同一个纹理中计算前景和背景，这使得前背景的模糊会互相影响。下面这张图的例子，就是聚焦在前景、景深强度拉到最高，这个时候，**前景柱子的黑色就会扩散到背景中，导致背景错误变暗**。这个错误的原因，就是在对背景的像素进行虚化的时候，直接从前景中采样颜色。而这是不符合现实情况的。



- 我们的虚化均使用正方形的采样核，这会使得虚化效果不够拟真。如下图所示：



- 我们不考虑相机 fov 对应的焦距、相机光圈等物理参数，而是直接使用一系列无物理意义的参数来描述焦平面、虚化程度。

以上内容，都是我们可以做的修复和优化。在接下来的章节中，我们将不提供具体代码实现，而是留给你作为练习题，来完成这些优化和修复。

5. 可视化模糊半径

本章，我们要求实现一个可视化景深中模糊半径 Blur Radius 的效果。

在上文中，我们 Shader 代码存在如下这个片段：

```
// 可视化焦平面
if (param.b_visualize_focus_plan != 0) {
    if (abs(coc) <= focus_range) {
        color = color; // do nothing
```

```
    } else if (coc > 0.0) {
        color = lerp(color, float3(0.0, 0.0, 1.0), 0.7);
    } else {
        color = lerp(color, float3(0.0, 1.0, 0.0), 0.7);
    }
}
```

此处的逻辑易于理解：先读取一个 bool 值表示是否启用 Debug 功能，如果启用的话，则修改颜色值。

那么我们同样的，可以按照上文的流程，添加一个 UI 可选框，表示是否启用模糊半径的可视化功能。然后在 Shader 中读取这个值，并根据模糊半径 Blur Radius 的大小，来修改颜色值。

模糊半径越大，我们就让颜色越偏向白色；模糊半径越小，我们就让颜色越偏向黑色。这样，我们就可以通过颜色的变化，来直观地观察到模糊半径的大小了。

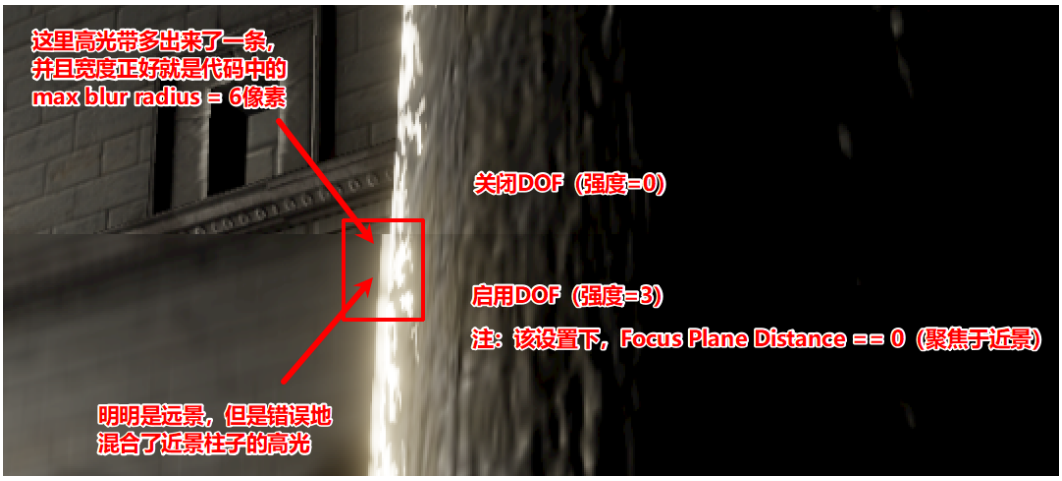
5.1 验证

启用可视化之后，能看到如下图所示的效果：

其中，纯黑色表示模糊半径为 0，纯白色表示模糊半径达到最大值。可以看到，在焦平面附近，模糊半径较小，因此颜色较暗；而在远离焦平面的地方，模糊半径较大，因此颜色较亮。

6. 前背景分离

在前文的第 4.6 小节“缺陷与优化”中，我们提出了一个缺陷：

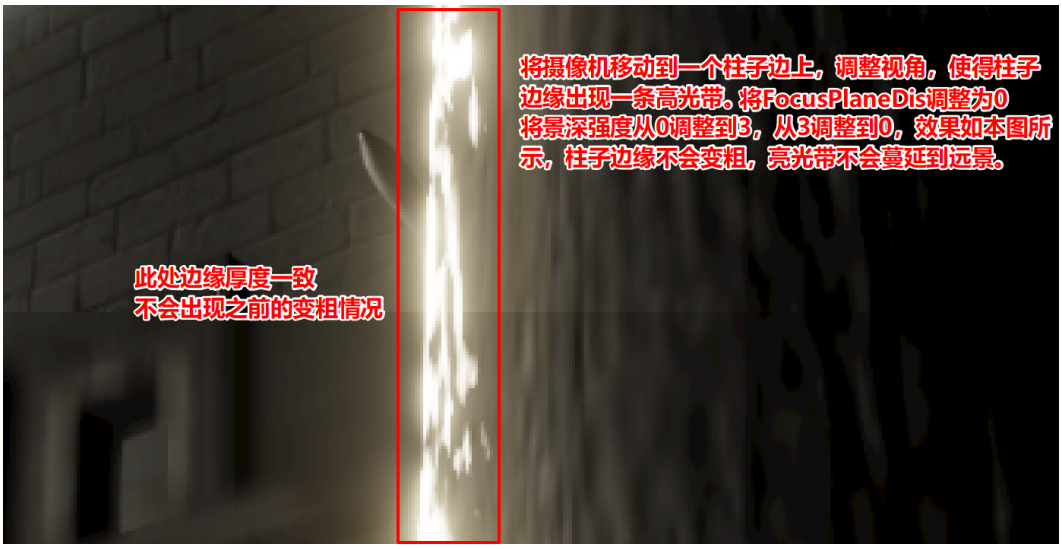


放大该图片后，我们可以看到下方启用景深后，左侧背景边缘出现了亮光。这是由于背景需要模糊，但我们的代码没有正确处理背景和前景的边界，导致前景的颜色泄漏到了背景上。

这一章，要求你修复这个缺陷，使得前背景能够正确分离，得到更自然的景深效果。你可以考虑在 Shader 的嵌套 for 循环中添加一些判断条件，来区分前景、焦平面、背景的像素：让背景只采样背景像素，前景只采样前景像素。

6.1 验证

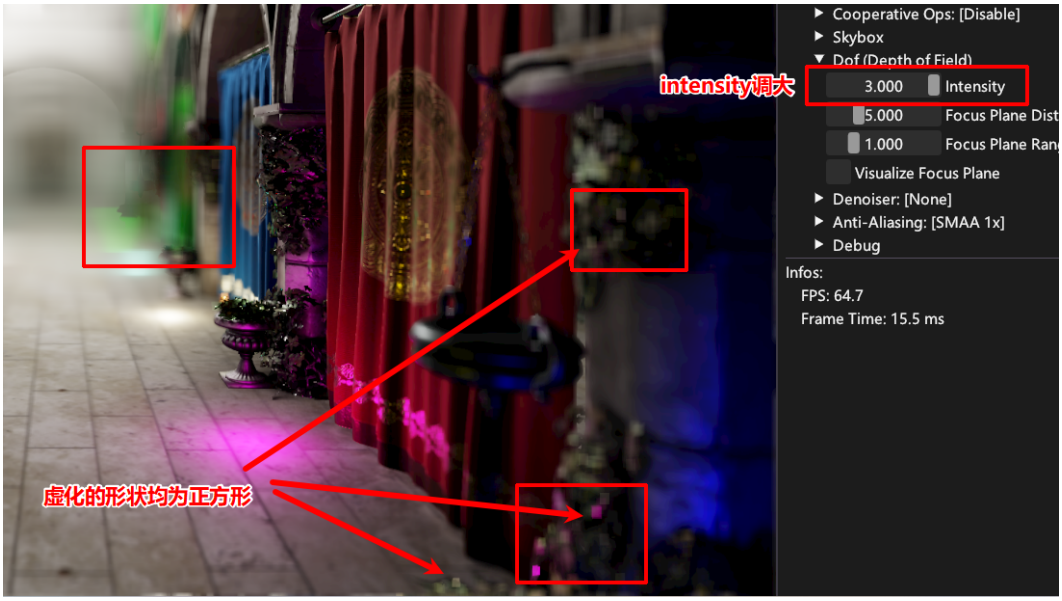
修复之后，之前的缺陷会消失。同样一个视角下，背景像素不会再被焦平面亮斑所污染：



请你说明修复方法、实现细节，并展示类似上图的画面来说明修复是否成功。

7. 圆盘采样

在前文的第 4.6 小节“缺陷与优化”中，我们提出了一个缺陷：

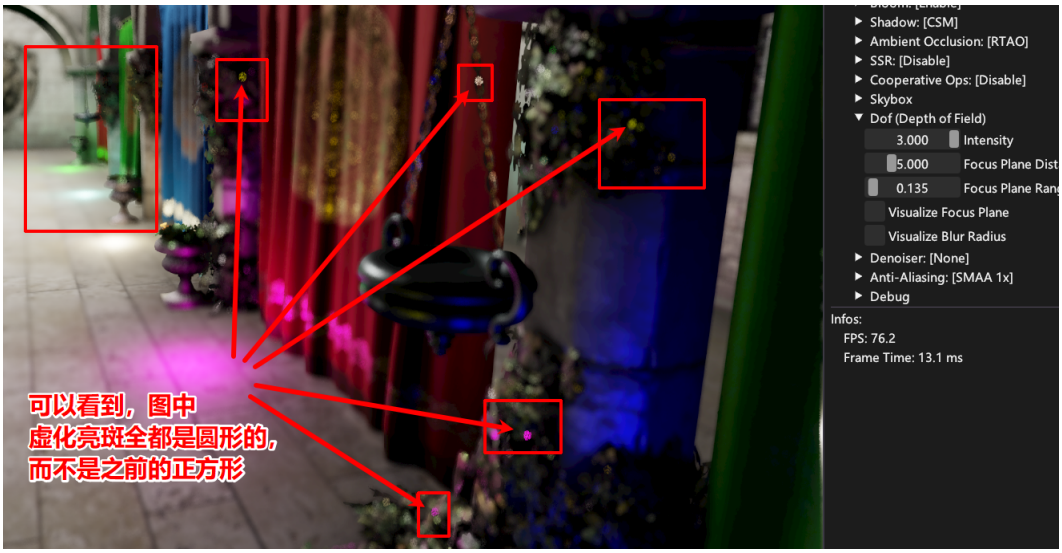


放大该图片后，我们可以看到前景和背景的虚化效果都十分不自然。前景出现了许多正方形光斑，通常情况下这是不美观的。

回顾代码，我们就可以发现，代码中我们直接使用了双重 for 循环，进行了一个正方形区域的采样。而在真实的光学系统中，透镜是一个圆盘，因此，正确的做法应该是：**在一个圆盘区域内，以均匀概率进行随机采样**。

7.1 验证

修复之后，之前的正方形光斑会消失。同样一个视角下，图像中不应该存在任何正方形的 Pattern：



请你说明修复方法、实现细节，并展示类似上图的画面来说明修复是否成功。

8. 性能优化（可选项）

在完成了上面的基础项之后，我们可以发现，景深效果在强度拉高的情况下，性能开销十分大。这主要是因为其中的两重 for 循环采样。假设屏幕是 1080p 分辨率，每个像素的模糊半径均为 6，那么我们总共就需要进行 1080 * 1920 * 6 * 6 7e7 次采样，每次采样都需要访问显存、读取颜色和深度、然后做一系列运算，此外，我们每帧还要进行几十次这套运算。

在实际的商业渲染器（例如虚幻引擎）中，我们会进行大量的优化，例如：

1. Poisson Disk 采样：用特殊的采样方式，而不是密铺地采样所有像素，从而降低采样次数
2. 半分辨率优化：先把画面缩小到半分辨率，进行景深处理后，再上采样回全分辨率。这样可以直接减少 75% 的像素数量，是一个在工业界中非常通用的后处理优化思路。景深虚化对像素具体颜色的依赖较弱，因此我们可以引入这种近似，来换取性能的提升
3. COC 预计算：我们在代码中是每个像素都计算了模糊半径（COC）。但实际上，我们可以先把所有像素都计算出 COC 和模糊半径 BlurRadius，然后把它们存到一个纹理中。这样，我们之后在采样的时候，就不需要进行那么多的运算，而是直接访问纹理来获取模糊半径
4. Tile-based 优化：我们可以把屏幕划分成一个个小块（Tile），然后对于每个 Tile，

我们只计算一次模糊半径的平均值，来代表这个 Tile 中所有像素的模糊半径。这样，我们就可以大幅度减少模糊半径的计算次数

本可选项要求你实现任意的一种或多种优化方法，并进行数据分析来说明优化的效果。其中，一些优化方法较为复杂，需要修改渲染管线，例如加入新的纹理、定义新的 Pass 等。此外，上述优化方案中最简单的一个优化方法为 Poisson Disk 采样 + 双线性插值采样。

你可以比较优化前后的帧率、CPU/GPU 占用率、以及画面质量。如果你实现了多种优化方法，你还可以比较它们之间的优劣，来说明不同优化方法的适用场景和效果差异。帧率方面，引擎自带的帧率统计工具浮动较大，你可以使用 NSight Graphics 等截帧工具，来最精确地测量优化前后的性能差异。

注：现代图形 API 为异步操作，所以请勿在 C++ 的 DofPass.h 内直接测量 CPU 端代码的耗时。此处测量的耗时结果是错误的，只代表了 CPU 端运行耗时，而非 GPU 侧的实际耗时。推荐使用 NSight Graphics 工具，或者粗略地测量帧率差异来评估优化效果。

9. 其他

MoerEngine 目前仍处于早期开发阶段，许多功能和细节都还没有完善。因此，如果你在完成作业的过程中遇到任何问题，都欢迎向助教提问。如果你觉得 MoerEngine 存在着 bug，你也可以直接在 Github 上提出 Issue。

此外，MoerEngine 目前为一个开源项目，如果你对引擎、渲染开发感兴趣，欢迎为 MoerEngine 贡献代码。具体可以参考引擎 README 文件中的 **如何贡献**部分。