

# Toy Renderer

---

孙嘉楷，3180105871

源代码位置：[SjJoK/LearnOpenGL \(github.com\)](https://SjJoK/LearnOpenGL.github.com)

## 目录

---

### Toy Renderer

目录

实验要求

实验原理

OpenGL

简介

渲染管线

GLFW

GLAD

Assimp

Dear ImGui

Gamma校正

Gamma值

人眼感知

校正

sRGB纹理

阴影

阴影映射

阴影失真

PCF

法线贴图

切线空间

HDR

色调映射

抗锯齿

MSAA

基于物理的着色

微表面模型

能量守恒

渲染方程

BRDF

法向量分布函数

菲涅尔方程

几何函数

PBR材质

源代码与分析

阴影

抗锯齿

顶点着色器

PBR着色

主函数

实验结果

个人总结

# 实验要求

使用OpenGL渲染一个Sketchfab上的模型，要求效果一致

## 实验原理

### OpenGL

#### 简介

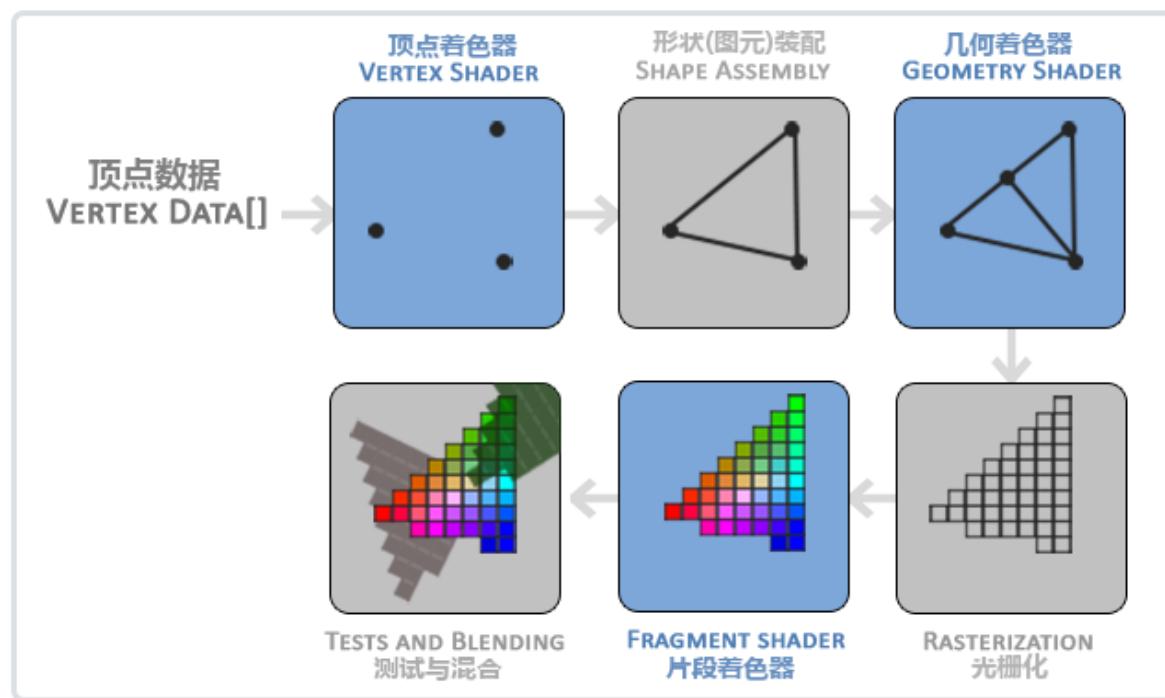
引用Khronos官网对OpenGL的介绍

OpenGL® is the most widely adopted 2D and 3D graphics API in the industry, bringing thousands of applications to a wide variety of computer platforms. It is window-system and operating-system independent as well as network-transparent. OpenGL enables developers of software for PC, workstation, and supercomputing hardware to create high-performance, visually compelling graphics software applications, in markets such as CAD, content creation, energy, entertainment, game development, manufacturing, medical, and virtual reality. OpenGL exposes all the features of the latest graphics hardware.

简单来说，OpenGL是一个被广泛应用的2D/3D图形API

为了使用采用核心模式 (Core-profile mode) 而非立即渲染模式 (Immediate mode) 的绘制模式，本次实验我使用的是OpenGL 3.3

#### 渲染管线



图形渲染管线的第一个部分是顶点着色器，它把一个单独的顶点作为输入。顶点着色器主要的目的是把局部坐标转化为裁剪坐标，同时顶点着色器允许我们对顶点属性进行一些基本处理。

图元装配阶段将顶点着色器输出的所有顶点作为输入，并所有的点装配成指定图元的形状。

图元装配阶段的输出会传递给几何着色器。几何着色器把图元形式的一系列顶点的集合作为输入，它可以通过产生新顶点构造出新的（或是其它的）图元来生成其他形状。

几何着色器的输出会被传入光栅化阶段，这里它会把图元映射为最终屏幕上相应的像素，生成供片段着色器使用的片段。在片段着色器运行之前会执行裁切。裁切会丢弃超出视图以外的所有像素，用来提升执行效率。

片段着色器的主要目的是计算一个像素的最终颜色。

在所有对应颜色值确定以后，最终的对象将会被传到最后一个阶段，进行模板测试、深度测试、Alpha测试，最后再混合。

## GLFW

GLFW是一个专门针对OpenGL的C语言库，它提供了一些渲染物体所需的最低限度的接口。

本次实验我使用GLFW作为窗口管理、输入处理等的框架

## GLAD

引用Github仓库中README的介绍

Multi-Language GL/GLES/EGL/GLX/WGL Loader-Generator based on the official specs.

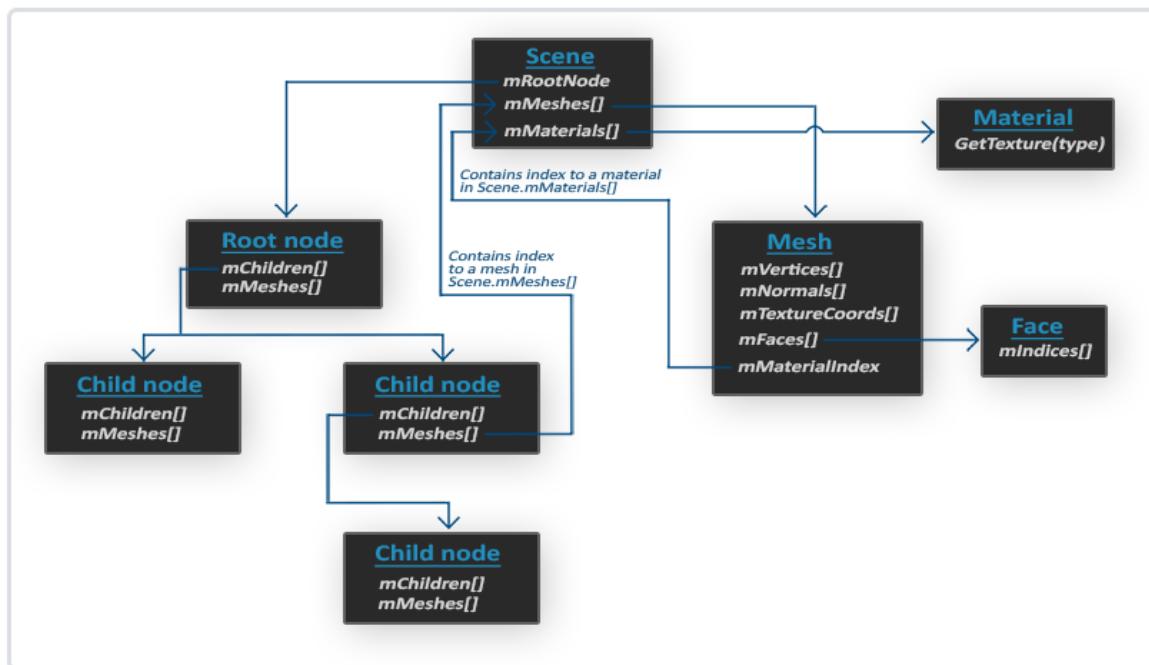
简单来说，GLAD帮我们处理了GL函数装载、生成等繁琐的问题

## Assimp

引用官网的介绍

The **Open Asset Import Library** (short name: Assimp) is a portable Open-Source library to import various well-known [3D model formats](#) in a uniform manner.

简单来说，是一个开源的3D模型导入，导入后数据模型如下



本次实验我使用Assimp进行模型导入

## Dear ImGui

引用Github仓库中README的介绍

Dear ImGui is a **bloat-free graphical user interface library for C++**. It outputs optimized vertex buffers that you can render anytime in your 3D-pipeline enabled application. It is fast, portable, renderer agnostic and self-contained (no external dependencies).

Dear ImGui is designed to **enable fast iterations** and to **empower programmers** to create **content creation tools and visualization / debug tools** (as opposed to UI for the average end-user). It favors simplicity and productivity toward this goal, and lacks certain features normally found in more high-level libraries.

Dear ImGui is particularly suited to integration in games engine (for tooling), real-time 3D applications, fullscreen applications, embedded applications, or any applications on consoles platforms where operating system features are non-standard.

简单来说，Dear ImGui是一个开箱即用的GUI库，可以很方便的用在我们的渲染器中来调整各种参数

## Gamma校正

### Gamma值

Gamma也叫灰度系数，每种显示设备都有自己的Gamma值，都不相同，有一个公式：设备输出亮度 = 电压的Gamma次幂，任何设备Gamma基本上都不会等于1，等于1是一种理想的线性状态，这种理想状态是：如果电压和亮度都是在0到1的区间，那么多少电压就等于多少亮度。

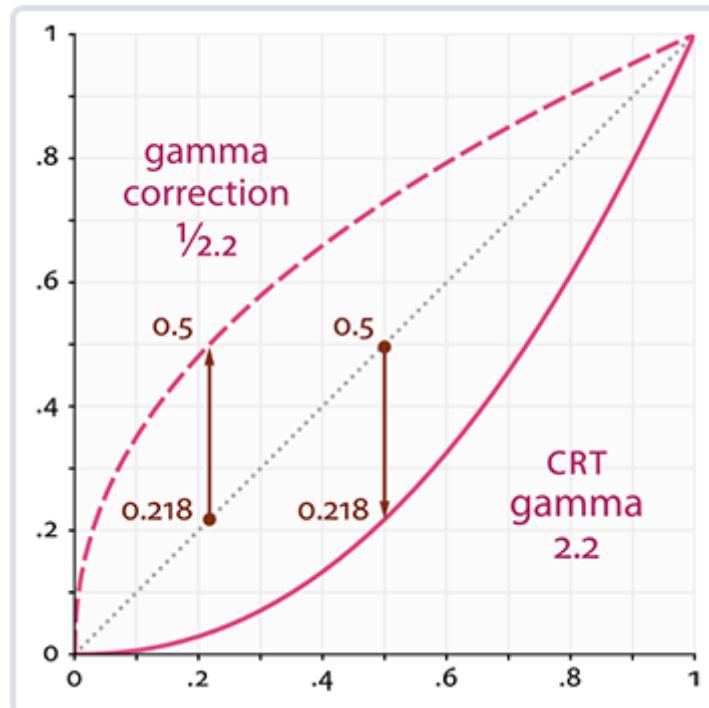
### 人眼感知

Perceived (linear) brightness =	0.0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0
Physical (linear) brightness =	0.0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0

由于人眼对颜色/灰阶的感知并不与物理亮度线性相关，而对比较暗的颜色变化更敏感。

### 校正

所以为了让我们在线性空间计算的颜色值被人眼更“正确”的感知到，所以我们需要在计算结束后进行Gamma校正



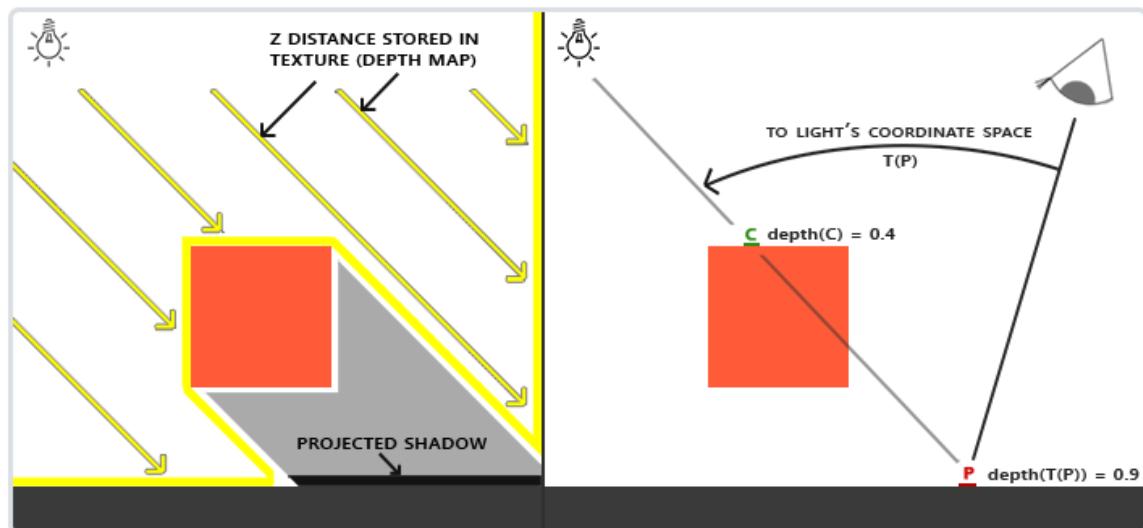
## sRGB纹理

艺术家们制作的纹理贴图是在sRGB空间的（因为艺术家们是根据屏幕效果来进行调整的），所以我们在读取时要进行逆向Gamma校正

## 阴影

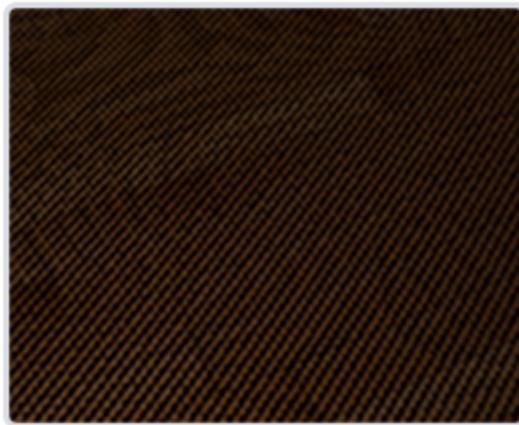
### 阴影映射

我们可以利用显卡的深度缓冲来进行阴影映射：在真正的渲染之前，先在光的位置为视角进行渲染，记录每个片段的深度值作为阴影贴图。然后在真正的渲染时，将世界坐标转化为以光为视角的坐标，在阴影纹理中采样，若其采样值小于其深度值，说明它被遮挡，光不在此片段着色

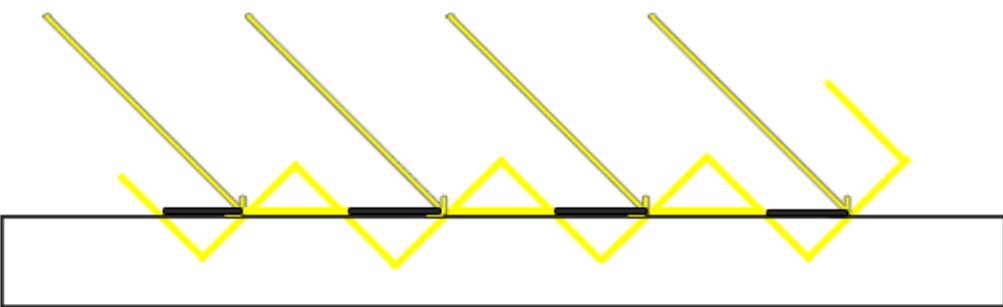


### 阴影失真

在实现上述阴影映射后，基本都会出现阴影失真的情况（我的项目中也出现了，但我在后续修复了，所以截图使用教程中的截图）

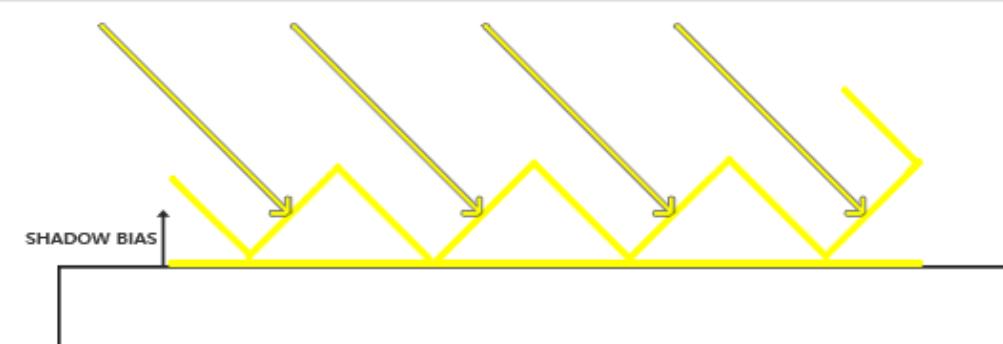


这是因为阴影贴图分辨率不够，出现了如下的情况



在距离光源比较远的情况下，多个片段可能从阴影贴图的同一个值中去采样，图片每个斜坡代表阴影贴图一个单独的纹理像素，可以看到，多个片段从同一个深度值进行采样，这就导致在同一深度值进行采样的临近片段，有些可能被视为处于阴影中，而有些没有。

我们可以采用一种非常Tricky的方法——阴影偏移，来解决上述问题



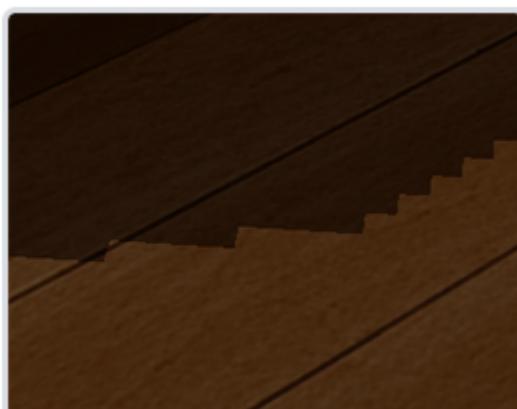
我们对阴影贴图应用一个很小的偏移量即可

但是有些表面坡度很大，仍然会产生阴影失真，需要更大的偏移量，所以我们采用下述方法来动态修改偏移量

```
float bias = max(0.05 * (1.0 - dot(normal, lightDir)), 0.005);
```

## PCF

同样受限于阴影贴图分辨率，当我们放大看阴影时，可能产生锯齿



我们可以采用PCF (percentage-closer filtering) 方法，在阴影贴图中多次采样，计算平均值，来作为阴影的值

```

float shadow = 0.0;
vec2 texelsize = 1.0 / textureSize(shadowMap, 0);
for(int x = -1; x <= 1; ++x)
{
    for(int y = -1; y <= 1; ++y)
    {
        float pcfDepth = texture(shadowMap, projCoords.xy + vec2(x, y) *
texelsize).r;
        shadow += currentDepth - bias > pcfDepth ? 1.0 : 0.0;
    }
}
shadow /= 9.0;

```

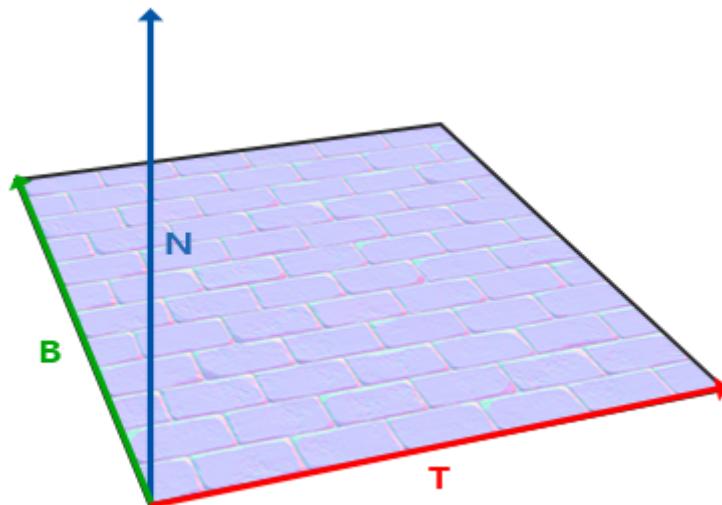
## 法线贴图

法线贴图为每个片段都可以通过纹理映射获得各自的法向，而不用通过顶点着色器计算三角形法线再插值得到，这可以很大程度上增加细节

当然，为了使用法线贴图，我们不能简单的使用纹理映射，因为法线的方向是会随着模型的方向变换的——所以我们需要将法线贴图中的，定义在切线空间中的法线变化到世界空间中

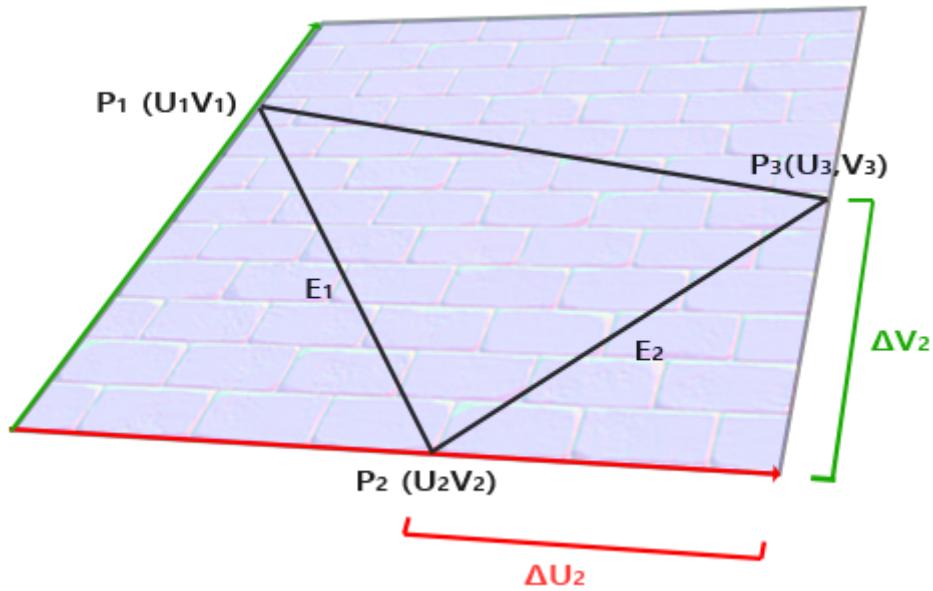
## 切线空间

已知上向量是表面的法线向量。右和前向量是切线(Tangent)和副切线(Bitangent)向量。下面的图片展示了一个表面的三个向量：



在使用Assimp导入模型时，可以自动计算切线与副切线，这很好，我们只需要把由经Model矩阵变化后的法向量、切线和副切线组成的TBN矩阵传入片段着色器，再用它变化采样得到的Normal值上即可

当然，对于任意一个三角形，我们也可以自己计算他的切线和副切线：



$$\begin{bmatrix} E_{1x} & E_{1y} & E_{1z} \\ E_{2x} & E_{2y} & E_{2z} \end{bmatrix} = \begin{bmatrix} \Delta U_1 & \Delta V_1 \\ \Delta U_2 & \Delta V_2 \end{bmatrix} \begin{bmatrix} T_x & T_y & T_z \\ B_x & B_y & B_z \end{bmatrix}$$

两边同乘 $\Delta U \Delta V$ 的逆矩阵

$$\begin{bmatrix} \Delta U_1 & \Delta V_1 \\ \Delta U_2 & \Delta V_2 \end{bmatrix}^{-1} \begin{bmatrix} E_{1x} & E_{1y} & E_{1z} \\ E_{2x} & E_{2y} & E_{2z} \end{bmatrix} = \begin{bmatrix} T_x & T_y & T_z \\ B_x & B_y & B_z \end{bmatrix}$$

这就可以计算出T和B了

$$\begin{bmatrix} T_x & T_y & T_z \\ B_x & B_y & B_z \end{bmatrix} = \frac{1}{\Delta U_1 \Delta V_2 - \Delta U_2 \Delta V_1} \begin{bmatrix} \Delta V_2 & -\Delta V_1 \\ -\Delta U_2 & \Delta U_1 \end{bmatrix} \begin{bmatrix} E_{1x} & E_{1y} & E_{1z} \\ E_{2x} & E_{2y} & E_{2z} \end{bmatrix}$$

## HDR

显示器只能显示0-1之间的值，而光照方程并没有数值上的限制，通过使片段的颜色超过1.0，我们有了一个更大的颜色范围，这也被称作HDR(High Dynamic Range, 高动态范围)

### 色调映射

我们在HDR中计算得到的结果终究要映射到0-1之间，这就叫做色调映射

有多种色调映射算法，在本次实验中，我采用了Reinhard色调映射和一个简单的曝光色调映射

Reinhard:

```
vec3 mapped = hdrColor / (hdrColor + vec3(1.0));
```

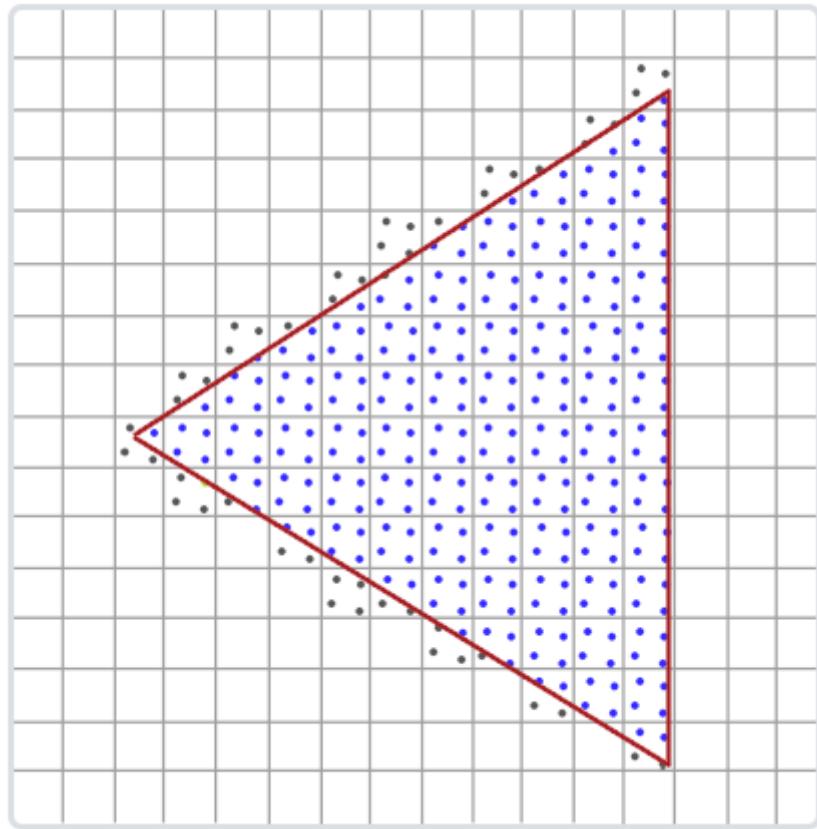
曝光色调映射

```
vec3 mapped = vec3(1.0) - exp(-hdrColor * exposure);
```

# 抗锯齿

抗锯齿有很多算法，这里主要介绍实验中使用的MSAA(MultiSample Anti-Aliasing)

## MSAA



对于每个片段/像素，会进行多次采样，计算某三角形遮盖了该像素的多少个子样本，然后只在像素中心运行一次片段着色器。最后根据被遮盖的子样本的数量来决定最后的像素颜色

## 基于物理的着色

注：在实验初期我采用的是Blinn-Phong模型，后期逐渐改写为PBR着色

一个基于物理的着色模型，需要满足以下三个条件：

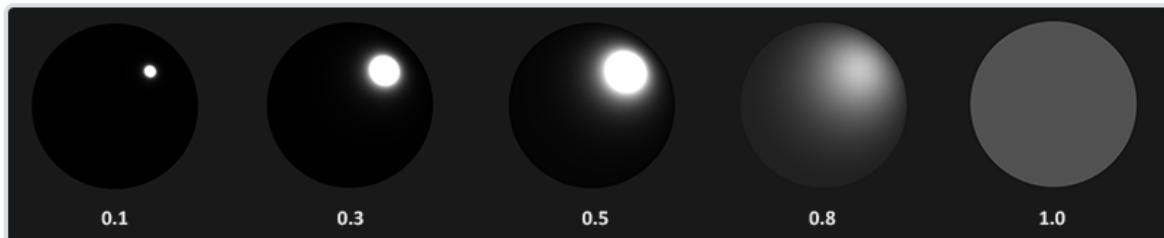
1. 基于微平面(Microfacet)的表面模型。
2. 能量守恒。
3. 应用基于物理的BRDF。

## 微表面模型

所有的PBR技术都基于微平面理论。这项理论认为，达到微观尺度之后任何平面都可以用被称为微平面的细小镜面来进行描绘。根据平面粗糙程度的不同，这些细小镜面的取向排列可以相当不一致：一个平面越是粗糙，这个平面上的微平面的排列就越混乱。这些微小镜面这样无序取向排列的影响就是，当我们特指镜面光/镜面反射时，入射光线更趋向于向完全不同的方向发散开来，进而产生出分布范围更广泛的反射。而与之相反的是，对于一个光滑的平面，光线大体上会更趋向于向同一个方向反射，造成更小更锐利的反射

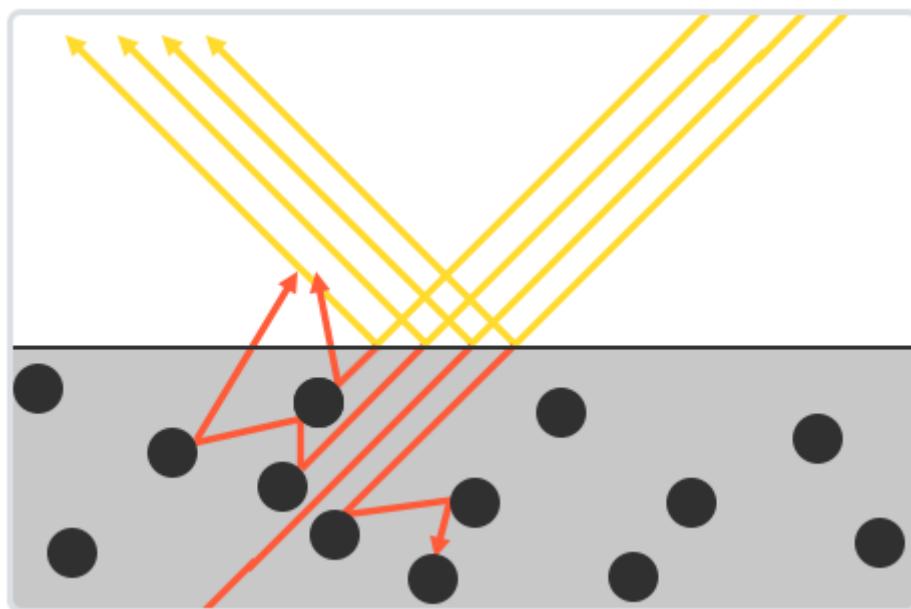


我们采用统计学的方法来概略的估算微平面的粗糙程度：我们可以基于一个平面的粗糙度来计算出半程向量与微平面平均取向方向一致的概率，下图是不同粗糙度对着色影响的例子：



## 能量守恒

为了遵守能量守恒定律，我们需要对漫反射光和镜面反射光之间做出明确的区分。当一束光线碰撞到一个表面的时候，它就会分离成一个折射部分和一个反射部分。反射部分就是会直接反射开来而不会进入平面的那部分光线，这就是我们所说的镜面光照。而折射部分就是余下的会进入表面并被吸收的那部分光线，这也就是我们所说的漫反射光照。



我们按照能量守恒的关系，首先计算镜面反射部分，它的值等于入射光线被反射的能量所占的百分比。然后折射光部分就可以直接由镜面反射部分计算得出：

```
float ks = 计算镜面部分...
float kd = 1.0 - ks
```

## 渲染方程

对一个没有自发光的物体，渲染方程为

$$L_o(p, \omega_o) = \int_{\Omega} f_r(p, \omega_i, \omega_o) L_i(p, \omega_i) n \cdot \omega_i d\omega_i$$

如果有自发光，在等式右侧的积分之前加入自发光项即可

上式中， $L_o$ 与 $L_i$ 代表某点 $p$ 某方向（立体角）上的辐射率（Radiance）， $\omega_i$ 与 $\omega_o$ 代表入射与出射方向（立体角）， $n$ 为点 $p$ 的法向量， $f_r$ 为BRDF(Bidirectional Reflective Distribution Function，双向反射分布函数)函数

## BRDF

本次实验中，我使用的是Cook-Torrance模型，因此只介绍Cook-Torrance模型

$$f_r = k_d f_{lambert} + k_s f_{cook-torrance}$$

这里的 $k_d$ 是早先提到过的入射光线中被折射部分的能量所占的比率，而 $k_s$ 是被反射部分的比率。

BRDF的左侧表示的是漫反射部分，这里用 $f_{lambert}$ 来表示。它被称为Lambertian漫反射，用如下的公式来表示：

$$f_{lambert} = \frac{c}{\pi}$$

BRDF的右侧表示的是镜面反射部分，这里用 $f_{cook-torrance}$ 来表示，其公式为：

$$f_{cook-torrance} = \frac{DFG}{4(\omega_o \cdot n)(\omega_i \cdot n)}$$

$D$ 代表法向量分布函数(Normal Distribution Function)， $F$ 代表菲涅尔方程(Fresnel Rquation)， $G$ 代表几何函数(Geometry Function)

- **法向量分布函数**：估算微平面平均取向方向与半程向量一致的概率，受表面粗糙度参数影响
- **菲涅尔方程**：菲涅尔方程描述的是在不同的表面角下表面所反射的光线所占的比率
- **几何函数**：当一个平面相对比较粗糙的时候，平面表面上的微平面有可能挡住其他的微平面从而减少表面所反射的光线，而几何函数就是来描述这个的

以上三种函数都有各自的形式，在本次实验中，我采用Trowbridge-Reitz GGX作为法向量分布函数，Fresnel-Schlick近似作为菲涅尔方程，Smith's Schlick-GGX作为几何函数

## 法向量分布函数

$$NDF_{GGXTR}(n, h, \alpha) = \frac{\alpha^2}{\pi((n \cdot h)^2(\alpha^2 - 1) + 1)^2}$$

此处 $n$ 为法向量（微平面平均取向方向）、 $h$ 为半程向量， $\alpha$ 为粗糙度参数

## 菲涅尔方程

菲涅尔（发音为Freh-nel）方程描述的是被反射的光线对比光线被折射的部分所占的比率，这个比率会随着我们观察的角度不同而不同。利用这个反射比率和能量守恒原则，我们可以直接得出光线被折射的部分以及光线剩余的能量。

菲涅尔方程非常复杂：

$$R_s = \left| \frac{n_1 \cos \theta_i - n_2 \cos \theta_t}{n_1 \cos \theta_i + n_2 \cos \theta_t} \right|^2 = \left| \frac{n_1 \cos \theta_i - n_2 \sqrt{1 - \left( \frac{n_1}{n_2} \sin \theta_i \right)^2}}{n_1 \cos \theta_i + n_2 \sqrt{1 - \left( \frac{n_1}{n_2} \sin \theta_i \right)^2}} \right|^2,$$

$$R_p = \left| \frac{n_1 \cos \theta_t - n_2 \cos \theta_i}{n_1 \cos \theta_t + n_2 \cos \theta_i} \right|^2 = \left| \frac{n_1 \sqrt{1 - \left( \frac{n_1}{n_2} \sin \theta_i \right)^2} - n_2 \cos \theta_i}{n_1 \sqrt{1 - \left( \frac{n_1}{n_2} \sin \theta_i \right)^2} + n_2 \cos \theta_i} \right|^2.$$

在上式中， $\theta_i$ 为入射光与法线的夹角， $\theta_t$ 为折射光与法线的夹角， $n_1$ 为入射光所在介质折射率， $n_2$ 为入射物体折射率，考虑到光的偏振， $R_s$ 与 $R_p$ 分别代表S偏振光和P偏振光在入射角为 $\theta_i$ 时反射光所占比率。

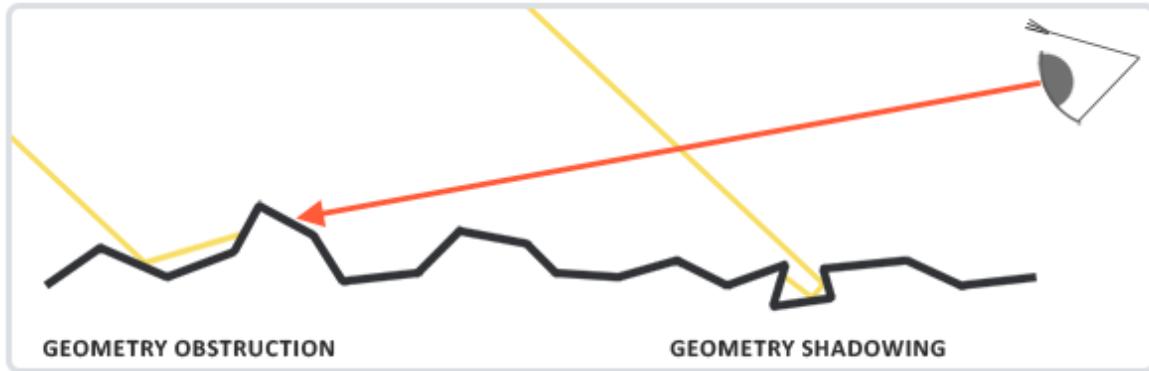
由于上述计算太过复杂，我们可以使用Fresnel-Schlick来近似计算菲涅尔方程

$$F_{Schlick}(h, v, F_0) = F_0 + (1 - F_0)(1 - (h \cdot v))^5$$

但是Fresnel-Schlick近似只对非导体表面有意义（计算结果接近），为了可以计算导体表面，我们传入表面对于法向入射 $F_0$ 的反应，然后利用Fresnel-Schlick近似进行插值计算即可

## 几何函数

为了更好的理解几何函数，先展示一张图



可以看到，微表面有可能相互遮蔽，而几何函数就是计算相互遮蔽的概率的。

$$G_{SchlickGGX}(n, v, k) = \frac{n \cdot v}{(n \cdot v)(1 - k) + k}$$

这里的 $k$ 是 $\alpha$ 基于几何函数是针对直接光照还是针对IBL光照的重映射：

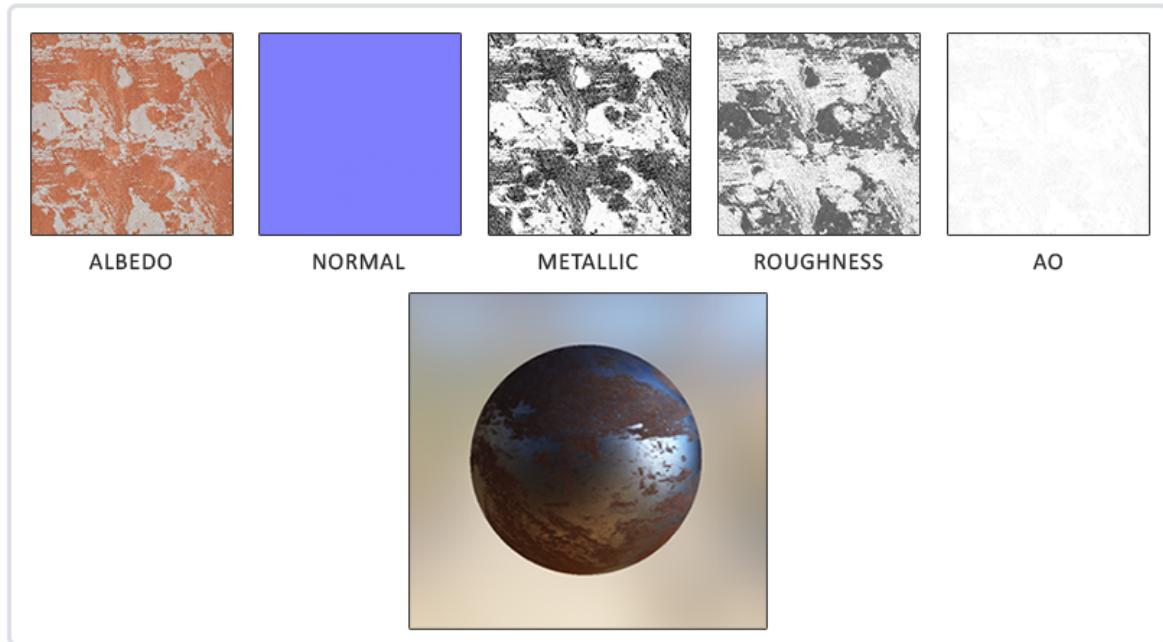
$$k_{direct} = \frac{(\alpha + 1)^2}{8}$$

$$k_{IBL} = \frac{\alpha^2}{2}$$

由于观察方向和光线方向都有可能产生遮蔽现象。所以我们要把两者都考虑进去

$$G(n, v, l, k) = G_{sub}(n, v, k)G_{sub}(n, l, k)$$

## PBR材质



为了有效利用PBR着色的各种参数，我们一般使用五种纹理贴图

- **反照率(Albedo):** 反照率(Albedo)纹理为每一个金属的纹素(Texel) (纹理像素) 指定表面颜色或者基础反射率。
- **法线(Normal):** 法线贴图使我们可以逐片段的指定独特的法线，来为表面制造出起伏不平的假象。
- **金属度(Metallic):** 金属贴图逐个纹素的指定该纹素是不是金属质地的。根据PBR引擎设置的不同，美术师们既可以将金属度编写为灰度值又可以编写为1或0这样的二元值。
- **粗糙度(Roughness):** 粗糙度贴图可以以纹素为单位指定某个表面有多粗糙。采样得来的粗糙度数值会影响一个表面的微平面统计学上的取向度。一个比较粗糙的表面会得到更宽阔更模糊的镜面反射 (高光)，而一个比较光滑的表面则会得到集中而清晰的镜面反射。
- **AO(Ambient Occlusion):** 环境光遮蔽贴图或者说AO贴图为表面和周围潜在的几何图形指定了一个额外的阴影因子。比如如果我们有一个砖块表面，反照率纹理上的砖块裂缝部分应该没有任何阴影信息。然而AO贴图则会把那些光线较难逃逸出来的暗色边缘指定出来。

## 源代码与分析

由于工程代码量过大，此处仅展示并分析重要的功能性代码

### 阴影

```
void get_depth_buffer(unsigned int& FBO, unsigned int& depthMap)
{
    glGenFramebuffers(1, &FBO);
    glBindFramebuffer(GL_FRAMEBUFFER, FBO);

    glGenTextures(1, &depthMap);
    glBindTexture(GL_TEXTURE_2D, depthMap);
    glTexImage2D(GL_TEXTURE_2D, 0, GL_DEPTH_COMPONENT,
                SHADOW_WIDTH, SHADOW_HEIGHT, 0, GL_DEPTH_COMPONENT, GL_FLOAT, NULL);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
```

```

glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);

glFramebufferTexture2D(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT, GL_TEXTURE_2D,
depthMap, 0);
glDrawBuffer(GL_NONE);
glReadBuffer(GL_NONE);
 glBindFramebuffer(GL_FRAMEBUFFER, 0);
 glBindTexture(GL_TEXTURE_2D, 0);
}

```

为阴影贴图建立buffer，附加阴影贴图为帧缓冲对象上附加的深度附件

```

glBindTexture(GL_TEXTURE_2D, depthMap);
glviewport(0, 0, SHADOW_WIDTH, SHADOW_HEIGHT);
glBindFramebuffer(GL_FRAMEBUFFER, depthMapFBO);
glClear(GL_DEPTH_BUFFER_BIT);
model = identity;
depthShader.use();
glm::mat4 lightProjection = glm::ortho(-ortho_length, ortho_length, -
ortho_length, ortho_length, near_plane, far_plane);
glm::mat4 lightView = glm::lookAt(camera.Position,
 -light_radius * glm::normalize(vec3(PBRLight_arr.direction[0],
PBRLight_arr.direction[1], PBRLight_arr.direction[2])),
 vec3(0.f),
 vec3(0.f, 1.f, 0.f));
lightSpaceMatrix = lightProjection * lightView;
depthShader.setMat4("lightSpaceMatrix", lightSpaceMatrix);
depthShader.setMat4("model", model);
ourModel->Draw(depthShader);
glBindFramebuffer(GL_FRAMEBUFFER, 0);
glBindTexture(GL_TEXTURE_2D, depthMap1);
glviewport(0, 0, SHADOW_WIDTH, SHADOW_HEIGHT);
glBindFramebuffer(GL_FRAMEBUFFER, depthMapFBO1);
glClear(GL_DEPTH_BUFFER_BIT);
model = identity;
depthShader.use();
glm::mat4 lightProjection1 = glm::ortho(-ortho_length, ortho_length, -
ortho_length, ortho_length, near_plane, far_plane);
glm::mat4 lightView1 = glm::lookAt(camera.Position,
 -light_radius * glm::normalize(vec3(PBRLight_arr1.direction[0],
PBRLight_arr1.direction[1], PBRLight_arr1.direction[2])),
 vec3(0.f),
 vec3(0.f, 1.f, 0.f));
lightSpaceMatrix1 = lightProjection1 * lightView1;
depthShader.setMat4("lightSpaceMatrix", lightSpaceMatrix1);
depthShader.setMat4("model", model);
ourModel->Draw(depthShader);
glBindFramebuffer(GL_FRAMEBUFFER, 0);
glviewport(0, 0, SCR_WIDTH, SCR_HEIGHT);
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

```

从光源渲染，得到阴影贴图

```

if (render_mode == SHADOW1)
{
    postshader.use();
    postShader.setInt("depthMap", 12);
    postShader.setFloat("near_plane", near_plane);
    postShader.setFloat("far_plane", far_plane);
    glBindVertexArray(quad_VAO);
    glDrawArrays(GL_TRIANGLES, 0, 6);
}

```

通过后处理Shader，绘制长方形并从阴影贴图中采样，以显示阴影贴图，下为后处理片段着色器

```

#version 330 core
out vec4 color;
in vec2 TexCoords;

uniform sampler2D depthMap;
uniform float near_plane;
uniform float far_plane;

float LinearizeDepth(float depth)
{
    float z = depth * 2.0 - 1.0; // Back to NDC
    return (2.0 * near_plane * far_plane) / (far_plane + near_plane - z * (far_plane - near_plane));
}

void main()
{
    float depthValue = texture(depthMap, TexCoords).r;
    //color = vec4(vec3(LinearizeDepth(depthValue) / far_plane), 1.0); // perspective
    color = vec4(vec3(depthValue), 1.0); // orthographic
}

```

对于定向光产生的阴影贴图，直接采用其r通道值即可（因为阴影贴图是帧缓冲对象上附加的深度附件）

## 抗锯齿

```

glfwWindowHint(GLFW_SAMPLES, 4);
 glEnable(GL_MULTISAMPLE);

```

直接使用glfw提供的多样本缓冲区

## 顶点着色器

```

#version 330 core
layout (location = 0) in vec3 aPos;
layout (location = 1) in vec3 aNormal;
layout (location = 2) in vec2 aTexCoord;
layout (location = 3) in vec3 tangent;
layout (location = 4) in vec3 bitangent;
out VS_OUT {
    vec3 FragPos;
    vec4 FragPosLightSpace;
}

```

```

    vec2 TexCoord;
    mat3 TBN;
} vs_out;
uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;
uniform mat4 normal_mat;
uniform mat4 lightSpaceMatrix;
void main()
{
    gl_Position = projection * view * model * vec4(aPos, 1.0);
    vec3 T = normalize(vec3(model * vec4(tangent, 0.0)));
    vec3 B = normalize(vec3(model * vec4(bitangent, 0.0)));
    vec3 N = normalize(vec3(model * vec4(aNormal, 0.0)));
    vs_out.FragPos = vec3(model * vec4(aPos, 1.0));
    vs_out.FragPosLightSpace = lightSpaceMatrix * vec4(vs_out.FragPos, 1.0);
    vs_out.TexCoord = aTexCoord;
    vs_out.TBN = mat3(T, B, N);
}

```

利用MVP矩阵得到裁剪空间坐标，同时需要计算TBN矩阵

在过去只有单光源时，我将点在光源中的位置也放到了顶点着色器的输出中，但是后来改为多光源时，我直接将光源的VP矩阵设置为光的属性了

## PBR着色

```

float ShadowCalculation(Light dirLight, vec3 normal, vec4 fragPosLightSpace)
{
    vec3 projCoords = fragPosLightSpace.xyz / fragPosLightSpace.w;
    projCoords = projCoords * 0.5 + 0.5;
    float closestDepth = texture(dirLight.shadowMap, projCoords.xy).r;
    float currentDepth = projCoords.z;
    vec3 lightDir = -normalize(dirLight.direction);
    float bias = max(0.001 * (1.0 - dot(normal, lightDir)), 0.0005);
    float shadow = 0.0;
    vec2 texelSize = 1.0 / textureSize(dirLight.shadowMap, 0);
    for(int x = -1; x <= 1; ++x)
    {
        for(int y = -1; y <= 1; ++y)
        {
            float pcfDepth = texture(dirLight.shadowMap, projCoords.xy + vec2(x, y) * texelSize).r;
            shadow += currentDepth - bias > pcfDepth ? 1.0 : 0.0;
        }
    }
    shadow /= 9.0;

    return shadow;
}

```

将片段对应的世界坐标位置变化到光源的视口坐标中，判断是否在阴影中，并设置阴影值

需要注意的是，为了解决阴影失真与锯齿现象，需要增加bias，和多次采样

```

float DistributionGGX(vec3 N, vec3 H, float roughness)
{

```

```

        float a      = roughness*roughness;
        float a2     = a*a;
        float NdotH  = max(dot(N, H), 0.0);
        float NdotH2 = NdotH*NdotH;

        float nom   = a2;
        float denom = (NdotH2 * (a2 - 1.0) + 1.0);
        denom = PI * denom * denom;

        return nom / denom;
    }

float GeometrySchlickGGX(float NdotV, float k)
{
    float nom   = NdotV;
    float denom = NdotV * (1.0 - k) + k;

    return nom / denom;
}

float GeometrySmith(vec3 N, vec3 V, vec3 L, float k)
{
    float NdotV = max(dot(N, V), 0.0);
    float NdotL = max(dot(N, L), 0.0);
    float ggx1 = GeometrySchlickGGX(NdotV, k);
    float ggx2 = GeometrySchlickGGX(NdotL, k);

    return ggx1 * ggx2;
}

vec3 fresnelSchlick(float cosTheta, vec3 F0)
{
    return F0 + (1.0 - F0) * pow(1.0 - cosTheta, 5.0);
}

```

上述函数为Cook-Torrance镜面反射模型需要的几个函数，已在实验原理中阐明

```

vec3 lightShade(vec3 normal, vec3 viewDir, Light light, vec3 albedo, vec3
metallic, float roughness, vec3 F0)
{
    vec4 FragPosLightSpace = light.lightSpaceMatrix * vec4(fs_in.FragPos, 1.0);

    float shadow = ShadowCalculation(light,normal, FragPosLightSpace);

    vec3 lightDir = -normalize(light.direction);

    vec3 halfVec = normalize(viewDir + lightDir);

    float attenuation = 1.0;

    vec3 F = fresnelSchlick(max(dot(halfVec, viewDir), 0.0), F0);

    vec3 Lo = vec3(0.0);

    float NDF = DistributionGGX(normal, halfVec, roughness);
    float G   = GeometrySmith(normal, viewDir, lightDir, roughness);
    vec3 nominator   = NDF * G * F;
    float denominator = 4.0 * max(dot(normal, viewDir), 0.0) * max(dot(normal,
    lightDir), 0.0) + 0.001;

```

```

vec3 specular      = nominator / denominator;

vec3 ks = F;
vec3 kD = vec3(1.0) - ks;

kD *= 1.0 - metallic;

float NdotL = max(dot(normal, lightdir), 0.0);

vec3 radiance = light.lightcolor * attenuation;

Lo += (kD * albedo / PI + specular) * radiance * NdotL;

vec3 result     = Lo;

if(shadowOn)
{
    result = (1-shadow)*Lo;
}

return result;
}

```

通过光源信息、视线信息和其他参数进行着色，需要注意的一点是，由于菲涅尔方程已经计算得到镜面反射占比，所以不用再乘一次 $K_s$

```

#version 330 core
#define RENDER 0
#define NORMAL 1
#define AO 2
#define ALBEDO 3
#define SPECULAR 4
#define ROUGHNESS 5
#define MODEL 0
const float PI = 3.1415926;
struct Material {
    vec3 diffuse;
    vec3 specular;
    float shininess;
    sampler2D texture_diffuse1;
    sampler2D texture_diffuse2;
    sampler2D texture_specular1;
    sampler2D texture_specular2;
    sampler2D texture_normal1;
    sampler2D texture_normal2;
    sampler2D texture_AO1;
    sampler2D texture_AO2;
    sampler2D texture_roughness1;
    sampler2D texture_roughness2;
};
struct Light {
    vec3 direction;
    vec3 lightColor;
    mat4 lightSpaceMatrix;
    sampler2D shadowMap;
};
in VS_OUT {

```

```

vec3 FragPos;
vec4 FragPosLightSpace;
vec2 TexCoord;
mat3 TBN;
} fs_in;
out vec4 FragColor;
uniform Light light_PBR;
uniform Light light_PBR1;
uniform Material material;
uniform bool shadowOn;
uniform bool gammaOn;
uniform bool HDROn;
uniform bool sRGBTexture;
uniform vec3 viewPos;
uniform int renderMode;
uniform int HDRMode;
uniform float exposure;
void main()
{
    vec3 normal = texture(material.texture_normal1, fs_in.TexCoord).rgb;
    normal = normalize(normal * 2.0 - 1.0);
    normal = normalize(fs_in.TBN * normal);
    vec3 viewDir = normalize(viewPos-fs_in.FragPos);

    float shadow = ShadowCalculation(light_PBR, normal,
fs_in.FragPosLightSpace);

    vec3 ao = texture(material.texture_AO1,fs_in.TexCoord).rrr;
    vec3 albedo = texture(material.texture_diffuse1,fs_in.TexCoord).rgb;
    if(sRGBTexture) albedo.rgb=pow(albedo.rgb,vec3(2.2));
    vec3 metallic = texture(material.texture_specular1,fs_in.TexCoord).rrr;
    float roughness = texture(material.texture_roughness1,fs_in.TexCoord).r;

    vec3 F0 = vec3(0.04);
    F0      = mix(F0, albedo, metallic);

    vec3 Lo = vec3(0.0);

    vec3 ambient = vec3(0.03) * albedo * ao;

    Lo += Lightshade(normal, viewDir, light_PBR, albedo, metallic, roughness,
F0);
    Lo += Lightshade(normal, viewDir, light_PBR1, albedo, metallic, roughness,
F0);

    vec3 result = ambient + Lo;

    if(HDROn)
    {
        if(HDRMode == 0)
        {
            result = result / (result + vec3(1.0));
        }
        else if(HDRMode == 1)
        {
            result = vec3(1.0) - exp(-result * exposure);
        }
    }
}

```

```

if(gammaOn)
{
    float gamma = 2.2;
    result.rgb = pow(result.rgb, vec3(1.0/gamma));
}
if(renderMode==RENDER)
    FragColor = vec4(result,1.f);
else if(renderMode==NORMAL)
    FragColor = vec4((normal+1)/2,1.f);
else if(renderMode==AO)
    FragColor = vec4(texture(material.texture_AO1, fs_in.TexCoord).rrr,1);
else if(renderMode==ALBEDO)
    FragColor = vec4(texture(material.texture_diffuse1,
fs_in.TexCoord).rgb,1);
else if(renderMode==SPECULAR)
    FragColor = vec4(texture(material.texture_specular1,
fs_in.TexCoord).rgb,1);
else if(renderMode==ROUGHNESS)
    FragColor = vec4(texture(material.texture_roughness1,
fs_in.TexCoord).rrr,1);
}

```

主函数与一些uniform参数定义，根据传入的参数来选择不同的渲染模式和渲染方法

## 主函数

仅讲解渲染循环部分

```

float currentFrame = glfwGetTime();
deltaTime = currentFrame - lastFrame;
lastFrame = currentFrame;
//Input
processInput(window);

```

处理输入输出

```

//ImGui
{
    ImGui_ImplOpenGL3_NewFrame();
    ImGui_ImplGlfw_NewFrame();
    ImGui::NewFrame();
    //Scene Editor
    {
        ImGui::Begin("Scene Editor", 0, ImGuiWindowFlags_AlwaysAutoResize);
        ImGui::BulletText("PBRLight Attribute");
        ImGui::Checkbox("pwhite Light (Around Y-axis)", &PBRLight_arr.white);
        ImGui::SliderFloat("pRadius ", &(PBRLight_arr.radius), 0.f, 10.f);
        ImGui::SliderFloat("pDegree ", &(PBRLight_arr.degree), 0.f, 360.f);
        ImGui::SliderFloat("pHeight ", &(PBRLight_arr.height), 0.f, 10.f);
        ImGui::DragFloat3("pColor ", PBRLight_arr.color, 0.5f, 0.f, 30.f);
        ImGui::SliderFloat("pFlux ", &(PBRLight_arr.flux), 0.f, 30.f);
        ImGui::BulletText("PBRLight1 Attribute (Around Z-axis)");
        ImGui::Checkbox("p1white Light", &PBRLight_arr1.white);
        ImGui::SliderFloat("p1Radius ", &(PBRLight_arr1.radius), 0.f, 10.f);
        ImGui::SliderFloat("p1Degree ", &(PBRLight_arr1.degree), 0.f, 360.f);
        ImGui::SliderFloat("p1Height ", &(PBRLight_arr1.height), 0.f, 10.f);
    }
}

```

```

ImGui::DragFloat3("p1Color ", PBRlight_arr1.color, 0.5f, 0.f, 30.f);
ImGui::SliderFloat("p1Flux ", &(PBRlight_arr1.flux), 0.f, 30.f);
ImGui::BulletText("Display Attribute");
ImGui::Checkbox("Gamma Correction ", &gamma_on);
ImGui::Checkbox("HDR ", &HDR_on);
if (ImGui::Selectable("Reinhard ", HDR_mode == 0))
    HDR_mode = 0;
if (ImGui::Selectable("By Exposure ", HDR_mode == 1))
    HDR_mode = 1;
ImGui::SliderFloat("Exposure ", &(exposure), 0.f, 10.f);
ImGui::Checkbox("Shadow ", &shadow_on);
ImGui::BulletText("Render Mode");
{
    if (ImGui::Selectable("Render", render_mode == RENDER))
        render_mode = RENDER;
    if (ImGui::Selectable("Albedo", render_mode == ALBEDO))
        render_mode = ALBEDO;
    if (ImGui::Selectable("Metallic", render_mode == SPECULAR))
        render_mode = SPECULAR;
    if (ImGui::Selectable("Roughness", render_mode == ROUGHNESS))
        render_mode = ROUGHNESS;
    if (ImGui::Selectable("Normal", render_mode == NORMAL))
        render_mode = NORMAL;
    if (ImGui::Selectable("AO", render_mode == AO))
        render_mode = AO;
    if (ImGui::Selectable("Mesh", render_mode == MESH))
        render_mode = MESH;
    if (ImGui::Selectable("Shadow Map 1", render_mode == SHADOW1))
        render_mode = SHADOW1;
    if (ImGui::Selectable("Shadow Map 2", render_mode == SHADOW2))
        render_mode = SHADOW2;
}
ImGui::Text("Application average %.3f ms/frame (%.1f FPS)", 1000.0f /
ImGui::GetIO().Framerate, ImGui::GetIO().Framerate);
ImGui::End();
}

//Model Editor
{
    ImGui::Begin("Model Editor", 0, ImGuiwindowFlags_AlwaysAutoResize);
    ImGui::Checkbox("Flip Y-axis", &flip_y);
    ImGui::Checkbox("SRGB Texture", &sRGB_texture);
    if (ImGui::Button("Model"))
    {
        fileDialog.Open();
        model_choose = MODEL;
    }
    if (ImGui::Button("Albedo Map"))
    {
        fileDialog.Open();
        model_choose = ALBEDO;

    }
    if (ImGui::Button("Normal Map"))
    {
        fileDialog.Open();
        model_choose = NORMAL;

    }
}

```

```

if (ImGui::Button("Metallic Map"))
{
    fileDialog.Open();
    model_choose = SPECULAR;
}
if (ImGui::Button("Roughness Map"))
{
    fileDialog.Open();
    model_choose = ROUGHNESS;

}
if (ImGui::Button("AO Map"))
{
    fileDialog.Open();
    model_choose = AO;

}
fileDialog.Display();
if (fileDialog.HasSelected())
{
    string ab_path = fileDialog.GetSelected().string();
    switch (model_choose)
    {
        case MODEL :
        {
            ourModel->loadModel(ab_path);
            break;
        }
        case ALBEDO:
        {
            texture_albedo.id = TextureFromFile(ab_path);
            break;
        }
        case METALLIC:
        {
            texture_metallic.id = TextureFromFile(ab_path);
            break;
        }
        case NORMAL:
        {
            texture_normal.id = TextureFromFile(ab_path);
            break;
        }
        case AO:
        {
            texture_AO.id = TextureFromFile(ab_path);
            break;
        }
        case ROUGHNESS :
        {
            texture_roughness.id = TextureFromFile(ab_path);
            break;
        }
    }
    fileDialog.ClearSelected();
}
stbi_set_flip_vertically_on_load(flip_y);
}

```

```
ImGui::Render();  
}
```

设置ImGui，设置不同的参数，加载模型、贴图等

```
glClearColor(0.1f, 0.1f, 0.1f, 1.0f);  
glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
```

清理颜色缓冲与深度缓冲

```
{  
    PBRlight_arr.position = vec3(PBRlight_arr.radius *  
        cos(glm::radians(PBRlight_arr.degree)),  
            PBRlight_arr.height,  
            -1 * PBRlight_arr.radius *  
        sin(glm::radians(PBRlight_arr.degree)));  
    vec3 d = vec3(0.f) - PBRlight_arr.position;  
  
    PBRlight_arr1.position = vec3(PBRlight_arr1.height,  
        PBRlight_arr1.radius *  
        sin(glm::radians(PBRlight_arr1.degree)),  
            PBRlight_arr1.radius *  
        cos(glm::radians(PBRlight_arr1.degree)));  
    vec3 d1 = vec3(0.f) - PBRlight_arr1.position;  
  
    if (PBRlight_arr.white)  
    {  
        PBRlight_arr.color[0] = 1.f;  
        PBRlight_arr.color[1] = 1.f;  
        PBRlight_arr.color[2] = 1.f;  
    }  
    if (PBRlight_arr1.white)  
    {  
        PBRlight_arr1.color[0] = 1.f;  
        PBRlight_arr1.color[1] = 1.f;  
        PBRlight_arr1.color[2] = 1.f;  
    }  
  
    PBRlight_arr.direction[0] = d.x;  
    PBRlight_arr.direction[1] = d.y;  
    PBRlight_arr.direction[2] = d.z;  
  
    PBRlight_arr1.direction[0] = d1.x;  
    PBRlight_arr1.direction[1] = d1.y;  
    PBRlight_arr1.direction[2] = d1.z;  
  
    projection = glm::perspective(glm::radians(camera.Zoom), (float)SCR_WIDTH /  
        (float)SCR_HEIGHT, 0.1f, 100.0f);  
    view = camera.GetViewMatrix();  
}
```

处理ImGui中获得的参数

```
{  
    glBindTexture(GL_TEXTURE_2D, depthMap);  
    glviewport(0, 0, SHADOW_WIDTH, SHADOW_HEIGHT);
```

```

glBindFramebuffer(GL_FRAMEBUFFER, depthMapFBO);
glClear(GL_DEPTH_BUFFER_BIT);
model = identity;
depthShader.use();
glm::mat4 lightProjection = glm::ortho(-ortho_length, ortho_length, -
ortho_length, ortho_length, near_plane, far_plane);
glm::mat4 lightView = glm::lookAt(//camera.Position,
- light_radius * glm::normalize(vec3(PBRlight_arr.direction[0],
PBRlight_arr.direction[1], PBRlight_arr.direction[2])),
vec3(0.f),
vec3(0.f, 1.f, 0.f));
lightSpaceMatrix = lightProjection * lightView;
depthShader.setMat4("lightSpaceMatrix", lightSpaceMatrix);
depthShader.setMat4("model", model);
ourModel->Draw(depthShader);
glBindFramebuffer(GL_FRAMEBUFFER, 0);
glBindTexture(GL_TEXTURE_2D, depthMap1);
glViewport(0, 0, SHADOW_WIDTH, SHADOW_HEIGHT);
glBindFramebuffer(GL_FRAMEBUFFER, depthMapFB01);
glClear(GL_DEPTH_BUFFER_BIT);
model = identity;
depthShader.use();
glm::mat4 lightProjection1 = glm::ortho(-ortho_length, ortho_length, -
ortho_length, ortho_length, near_plane, far_plane);
glm::mat4 lightView1 = glm::lookAt(//camera.Position,
- light_radius * glm::normalize(vec3(PBRlight_arr1.direction[0],
PBRlight_arr1.direction[1], PBRlight_arr1.direction[2])),
vec3(0.f),
vec3(0.f, 1.f, 0.f));
lightSpaceMatrix1 = lightProjection1 * lightView1;
depthShader.setMat4("lightSpaceMatrix", lightSpaceMatrix1);
depthShader.setMat4("model", model);
ourModel->Draw(depthShader);
glBindFramebuffer(GL_FRAMEBUFFER, 0);
glviewport(0, 0, SCR_WIDTH, SCR_HEIGHT);
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
}

```

## 阴影贴图相关

```

tureshader->use();
{
{
tureshader->setVec3("viewPos", camera.Position);

tureshader->setMat4("light_PBR1.lightSpaceMatrix", lightSpaceMatrix1);
tureShader->setVec3("light_PBR1.direction", PBRlight_arr1.direction[0],
PBRlight_arr1.direction[1], PBRlight_arr1.direction[2]);
tureShader->setVec3("light_PBR1.lightColor", PBRlight_arr1.flux*
vec3(PBRlight_arr1.color[0], PBRlight_arr1.color[1], PBRlight_arr1.color[2]));
tureShader->setBool("light_PBR1.point", false);

tureShader->setMat4("light_PBR.lightSpaceMatrix", lightSpaceMatrix);
tureShader->setVec3("light_PBR.direction", PBRlight_arr.direction[0],
PBRlight_arr.direction[1], PBRlight_arr.direction[2]);
tureShader->setVec3("light_PBR.lightColor", PBRlight_arr.flux*
vec3(PBRlight_arr.color[0], PBRlight_arr.color[1], PBRlight_arr.color[2]));

```

```

tureShader->setBool("light_PBR.point", false);

tureShader->setBool("sRGBTexture", sRGB_texture);
tureShader->setBool("gammaOn", gamma_on);
tureShader->setBool("HDROn", HDR_on);
tureShader->setInt("HDRMode", HDR_mode);
tureShader->setFloat("exposure", exposure);
tureShader->setBool("shadowOn", shadow_on);
tureShader->setInt("renderMode", render_mode);
// view/projection transformations
projection = glm::perspective(glm::radians(camera.Zoom),
(float)SCR_WIDTH / (float)SCR_HEIGHT, 0.1f, 100.0f);
view = camera.GetViewMatrix();
tureShader->setMat4("projection", projection);
tureShader->setMat4("view", view);
// render the loaded model
model = identity;
model = glm::translate(model, glm::vec3(0.0f, 0.0f, 0.0f)); // translate
it down so it's at the center of the scene
model = glm::scale(model, glm::vec3(1.f, 1.f, 1.f)); // it's a bit
too big for our scene, so scale it down
tureShader->setMat4("model", model);
tureShader->setMat4("normal_mat", transpose(inverse(model)));
}

{
glActiveTexture(GL_TEXTURE12);
tureShader->setInt("light_PBR.shadowMap", 12);
glBindTexture(GL_TEXTURE_2D, depthMap);

glActiveTexture(GL_TEXTURE13);
tureShader->setInt("light_PBR1.shadowMap", 13);
glBindTexture(GL_TEXTURE_2D, depthMap1);

glActiveTexture(GL_TEXTURE0 + ALBEDO);
tureShader->setInt("material.texture_diffuse1", ALBEDO);
glBindTexture(GL_TEXTURE_2D, texture_albedo.id);

glActiveTexture(GL_TEXTURE0 + METALLIC);
tureShader->setInt("material.texture_specular1", METALLIC);
glBindTexture(GL_TEXTURE_2D, texture_metallic.id);

glActiveTexture(GL_TEXTURE0 + ROUGHNESS);
tureShader->setInt("material.texture_roughness1", ROUGHNESS);
glBindTexture(GL_TEXTURE_2D, texture_roughness.id);

glActiveTexture(GL_TEXTURE0 + NORMAL);
tureShader->setInt("material.texture_normal1", NORMAL);
glBindTexture(GL_TEXTURE_2D, texture_normal.id);

glActiveTexture(GL_TEXTURE0 + AO);
tureShader->setInt("material.texture_AO1", AO);
glBindTexture(GL_TEXTURE_2D, texture_AO.id);
}
}

```

设置Shader中的变量

```

if (render_mode == MESH)
{
    ourModel->Draw(*tureShader, GL_LINES);
}
else if (render_mode == SHADOW1)
{
    postShader.use();
    postShader.setInt("depthMap", 12);
    postShader.setFloat("near_plane", near_plane);
    postShader.setFloat("far_plane", far_plane);
    glBindVertexArray(quad_VAO);
    glDrawArrays(GL_TRIANGLES, 0, 6);
}
else if (render_mode == SHADOW2)
{
    postShader.use();
    postShader.setInt("depthMap", 13);
    postShader.setFloat("near_plane", near_plane);
    postShader.setFloat("far_plane", far_plane);
    glBindVertexArray(quad_VAO);
    glDrawArrays(GL_TRIANGLES, 0, 6);
}
else
{
    ourModel->Draw(*tureShader);
}

```

根据选择的绘制模式的不同进行不同的绘制

**注：**绘图函数被包装到Mesh类中了，如下

```

void setupMesh()
{
    // create buffers/arrays
    glGenVertexArrays(1, &VAO);
    glGenBuffers(1, &VBO);
    glGenBuffers(1, &EBO);

    glBindVertexArray(VAO);
    // Load data into vertex buffers
    glBindBuffer(GL_ARRAY_BUFFER, VBO);
    // A great thing about structs is that their memory layout is sequential for
    // all its items.
    // The effect is that we can simply pass a pointer to the struct and it
    // translates perfectly to a glm::vec3/2 array which
    // again translates to 3/2 floats which translates to a byte array.
    glBufferData(GL_ARRAY_BUFFER, vertices.size() * sizeof(vertex),
    &vertices[0], GL_STATIC_DRAW);

    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, EBO);
    glBufferData(GL_ELEMENT_ARRAY_BUFFER, indices.size() * sizeof(unsigned int),
    &indices[0], GL_STATIC_DRAW);

    // set the vertex attribute pointers
    // vertex Positions
    glEnableVertexAttribArray(0);
}

```

```

glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, sizeof(Vertex), (void*)0);
// vertex normals
glEnableVertexAttribArray(1);
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, sizeof(Vertex),
(void*)offsetof(Vertex, Normal));
// vertex texture coords
glEnableVertexAttribArray(2);
glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, sizeof(Vertex),
(void*)offsetof(Vertex, TexCoords));
// vertex tangent
glEnableVertexAttribArray(3);
glVertexAttribPointer(3, 3, GL_FLOAT, GL_FALSE, sizeof(Vertex),
(void*)offsetof(Vertex, Tangent));
// vertex bitangent
glEnableVertexAttribArray(4);
glVertexAttribPointer(4, 3, GL_FLOAT, GL_FALSE, sizeof(Vertex),
(void*)offsetof(Vertex, Bitangent));
// ids
glEnableVertexAttribArray(5);
glVertexAttribIPointer(5, 4, GL_INT, sizeof(Vertex), (void*)offsetof(Vertex,
m_BoneIDs));
// weights
glEnableVertexAttribArray(6);
glVertexAttribPointer(6, 4, GL_FLOAT, GL_FALSE, sizeof(Vertex),
(void*)offsetof(Vertex, m_Weights));
 glBindVertexArray(0);
}

```

建立VAO

```

void Draw(Shader& shader, GLenum mode = GL_TRIANGLES)
{
    // draw mesh
    glBindVertexArray(VAO);
    glDrawElements(mode, indices.size(), GL_UNSIGNED_INT, 0);
    glBindVertexArray(0);

    // always good practice to set everything back to defaults once configured.
    glActiveTexture(GL_TEXTURE0);
}

```

绑定VAO，绘制

在渲染循环函数的最后：

```

ImGui_ImplOpenGL3_RenderDrawData(ImGui::GetDrawData());
// Poll Events and Swap
glfwPollEvents();
glfwSwapBuffers(window);

```

绘制GUI，处理实践，交换Buffer

## 实验结果

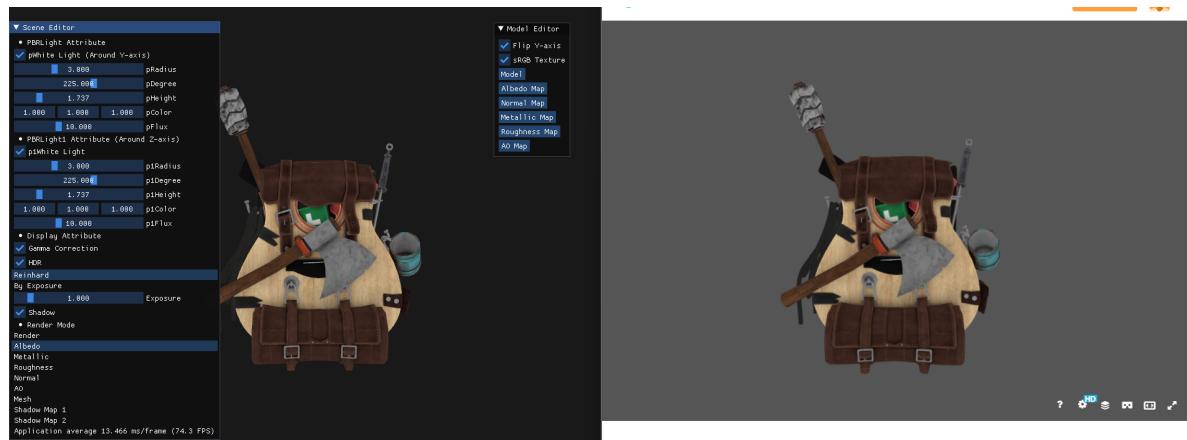
我采用了SketchFab上一个比较复杂的模型来检测自己的渲染器

## Survival Guitar Backpack (Low Poly).

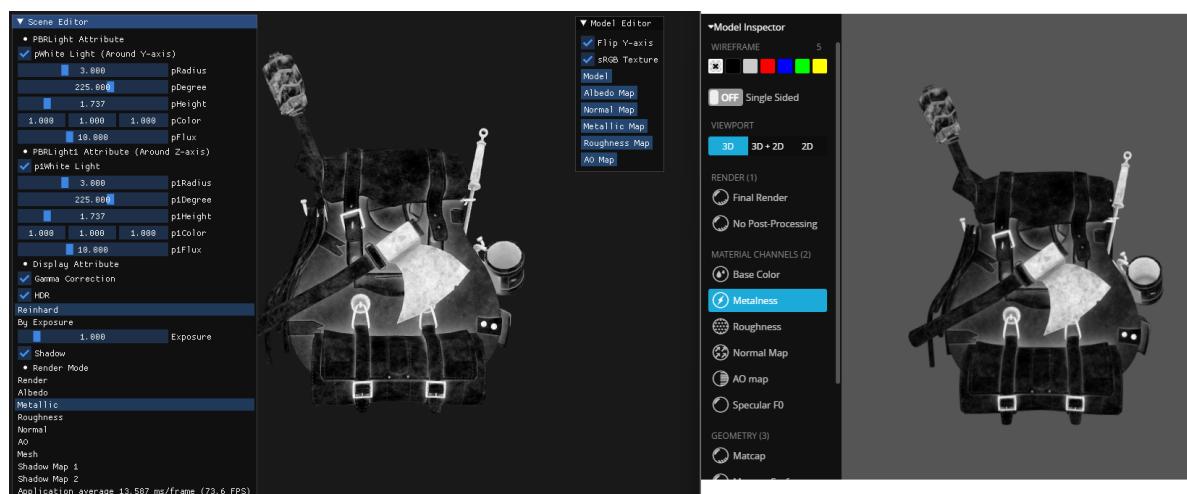


可以看到，该物品有阴影，也有PBR工作流的所有贴图，很适合测试我们的渲染器

- Base Color/Albedo

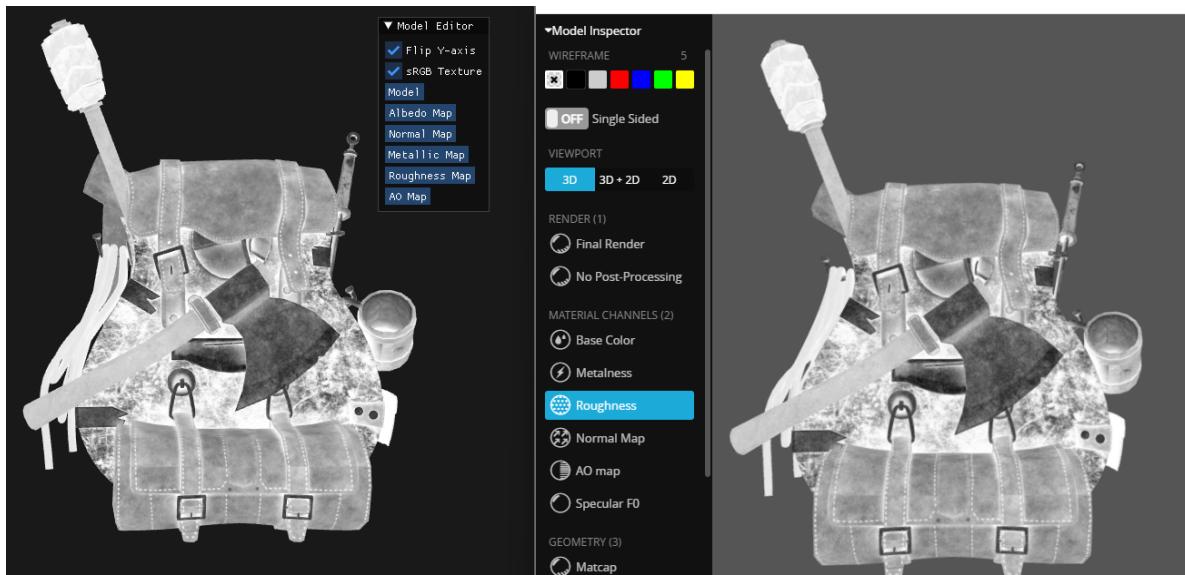


- Metallic

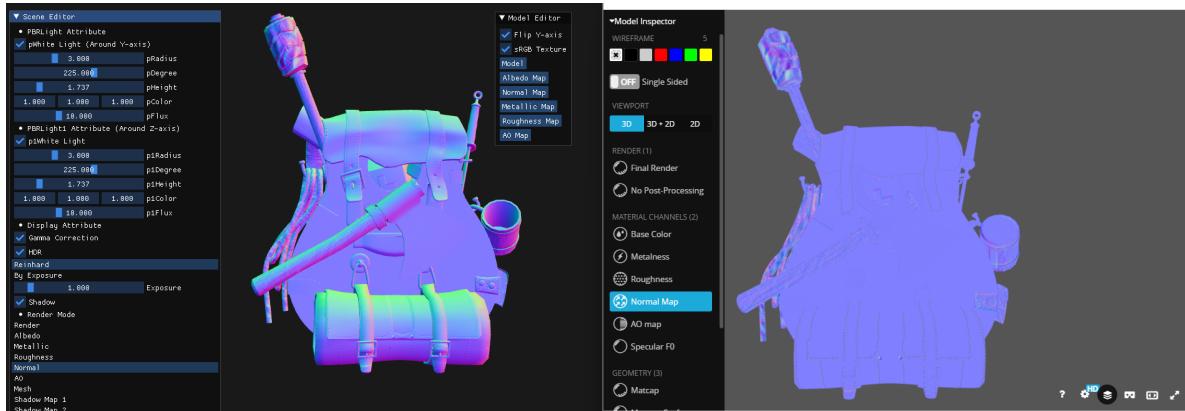


(可以发现，金属材质两者有些许色差，但我采用的确实是下载的贴图，片段着色器中的也是直接通过记录的纹理坐标进行采样来显示的，不清楚为什么会出现这种差异，这也略微影响了最终渲染的质量)

- Roughness

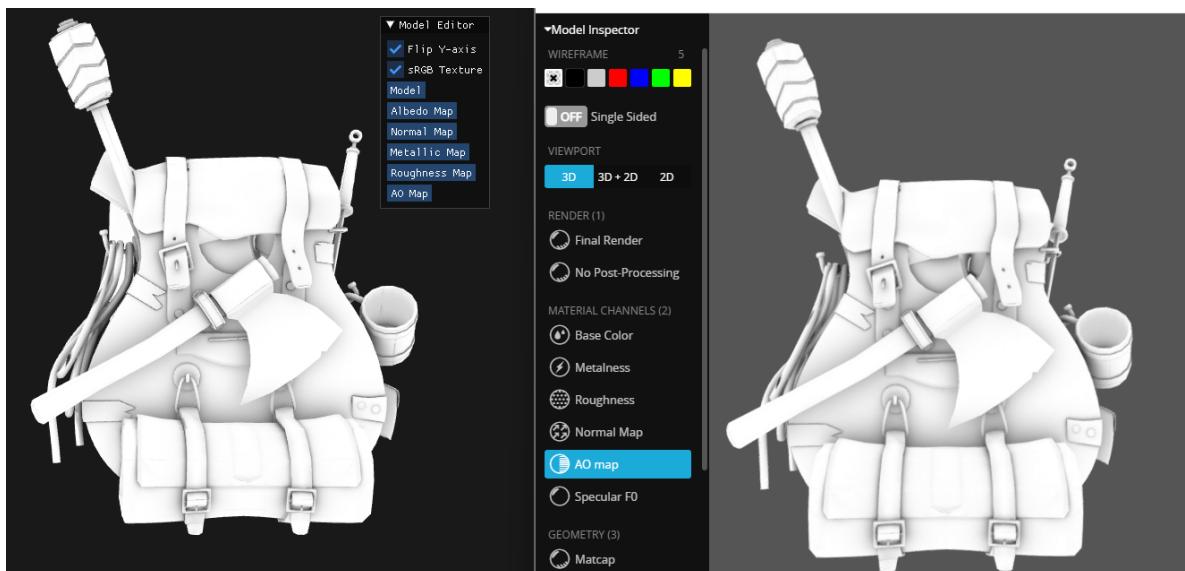


- Normal



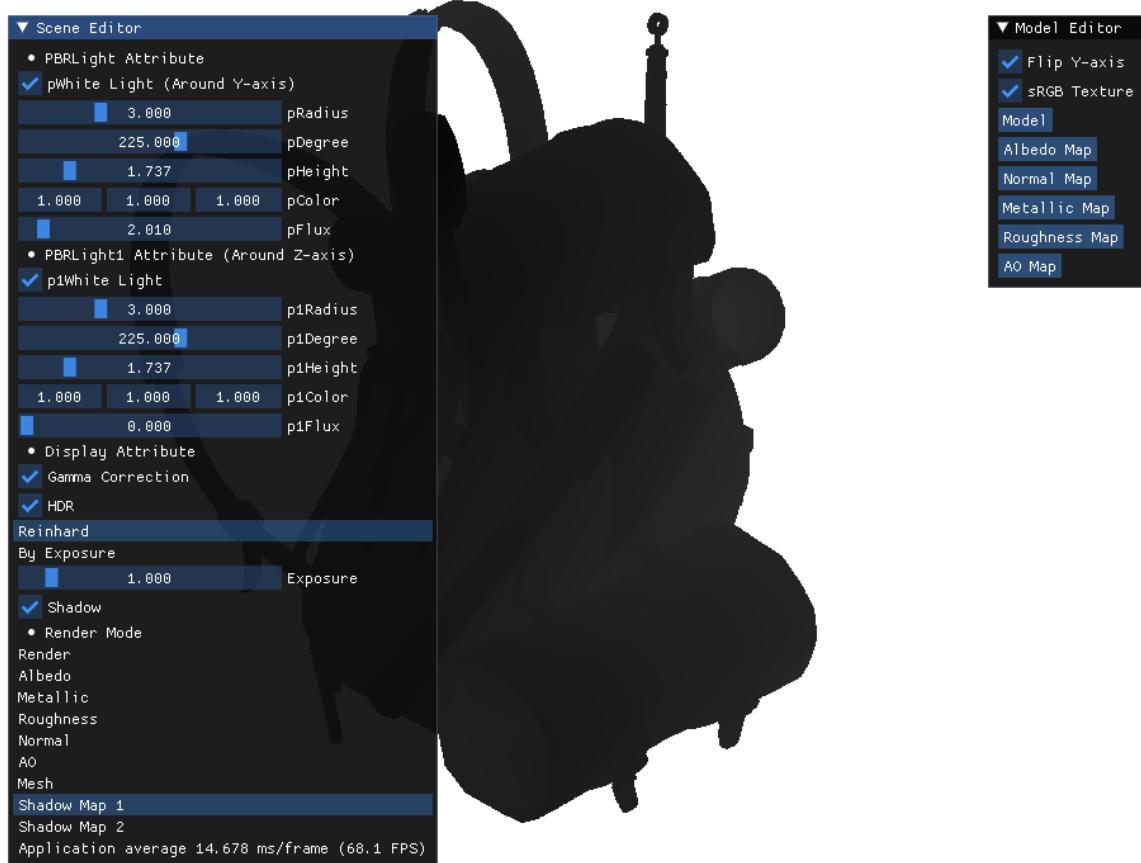
可以看到SketchFab是直接显示了法线贴图，这没有任何意义，我显示的是处理后的法线，更有用

- AO

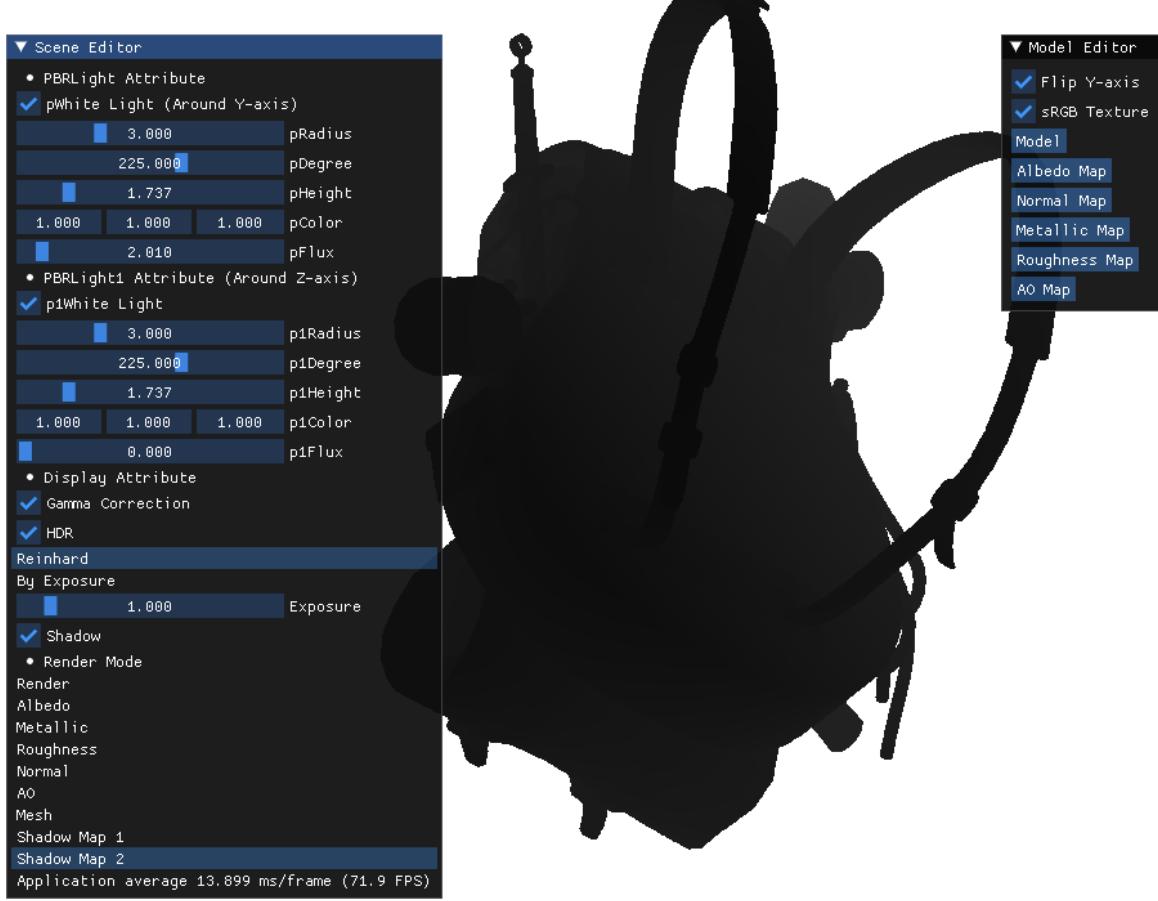


需要注意的是，SketchFab上渲染结果的阴影较淡（经过我自己测试，在没有灯光时，阴影部分颜色是比SketchFab上更深的，这说明SketchFab上不止有一个光源，但是PBR材质模型的表现是极其依赖光源的，在真实的渲染中，光源都会产生阴影，但是SketchFab上只有一个方向的阴影，所以我怀疑它增加了不会产生阴影的环境光，但这就与真实感渲染相悖了...所以我实现了两个可以产生阴影的定向光，实现更加真实的渲染）

- Shadow Map 1



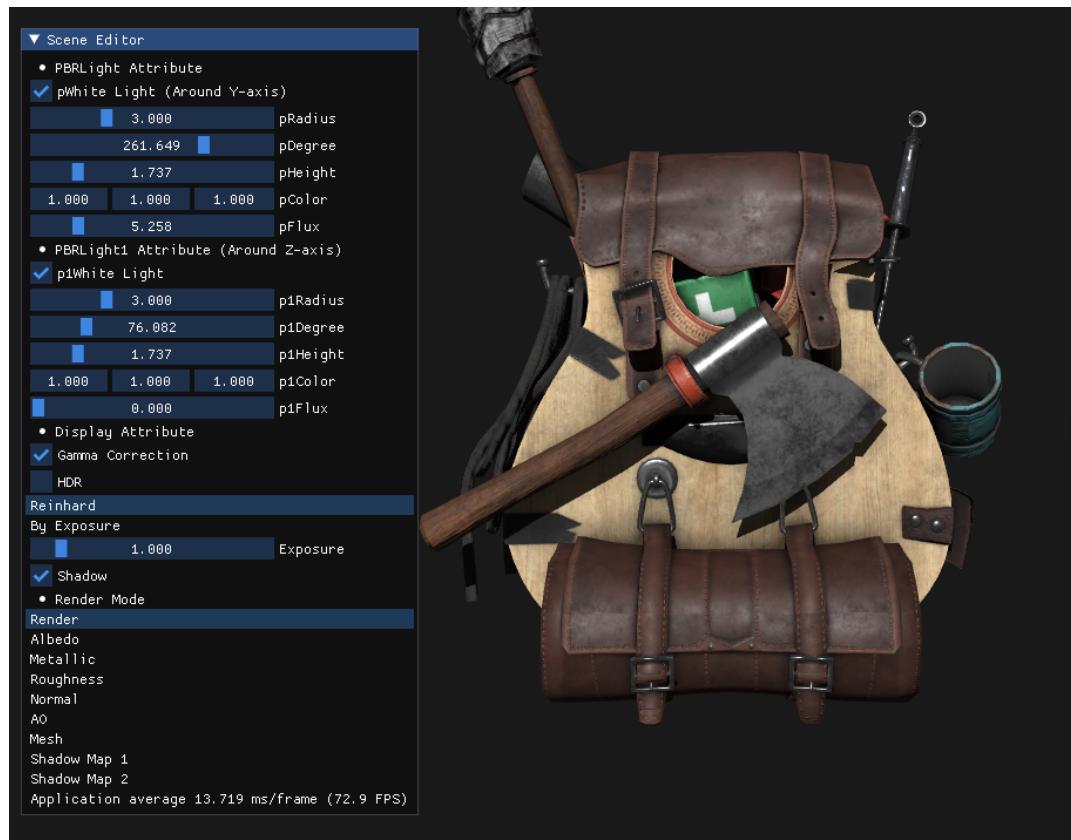
- Shadow Map 2

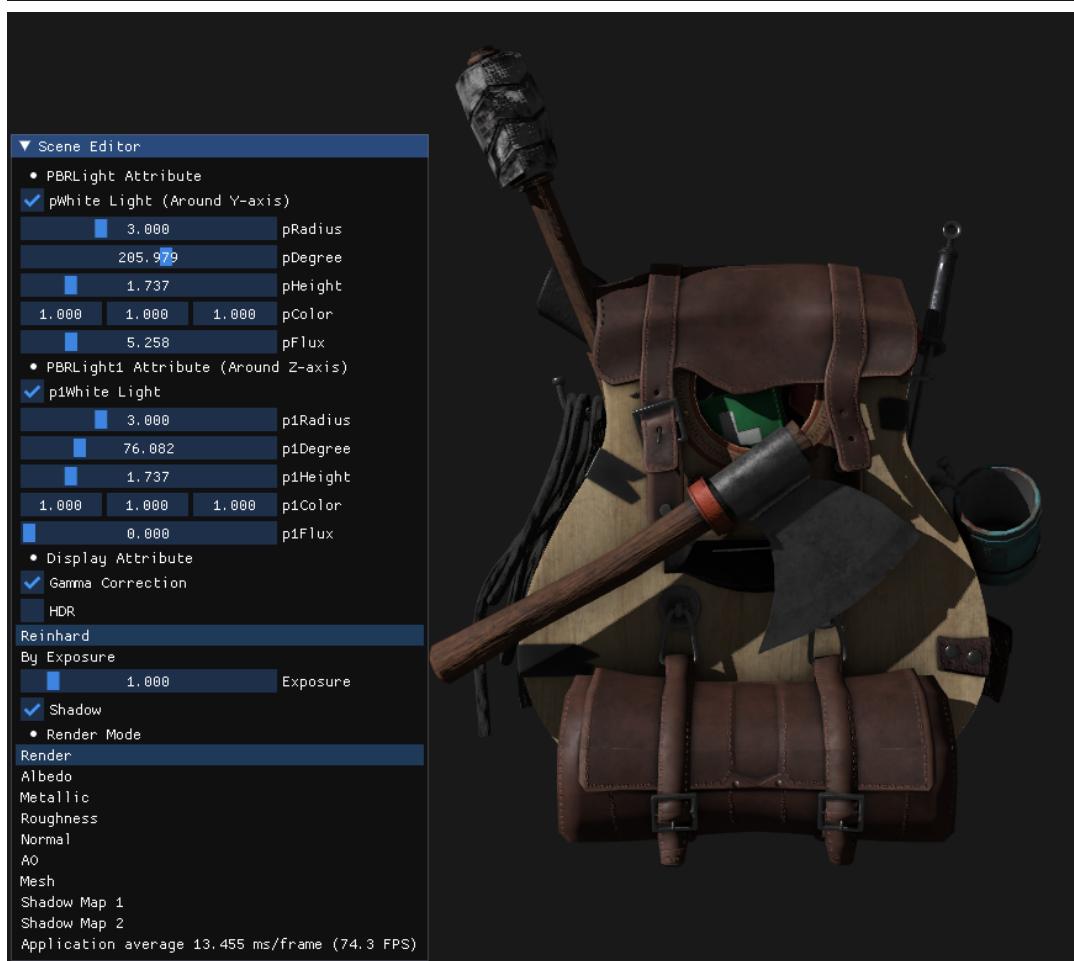
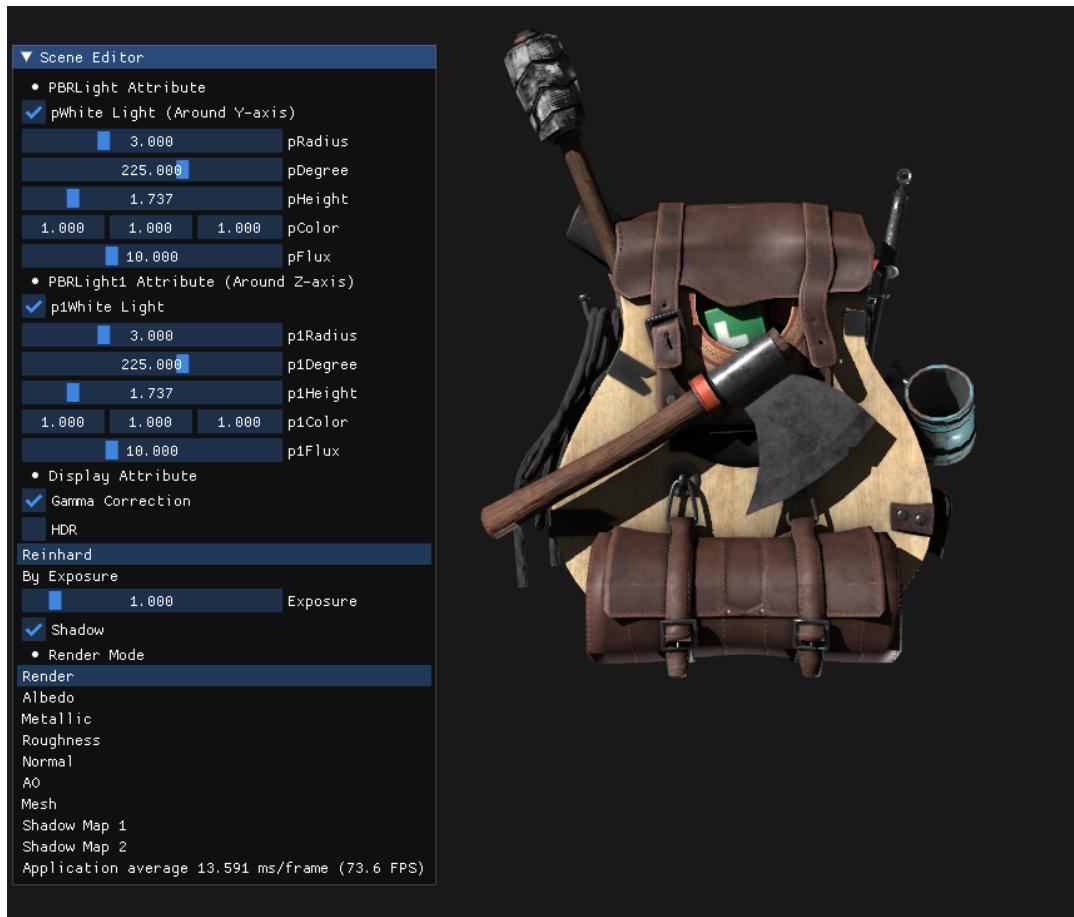


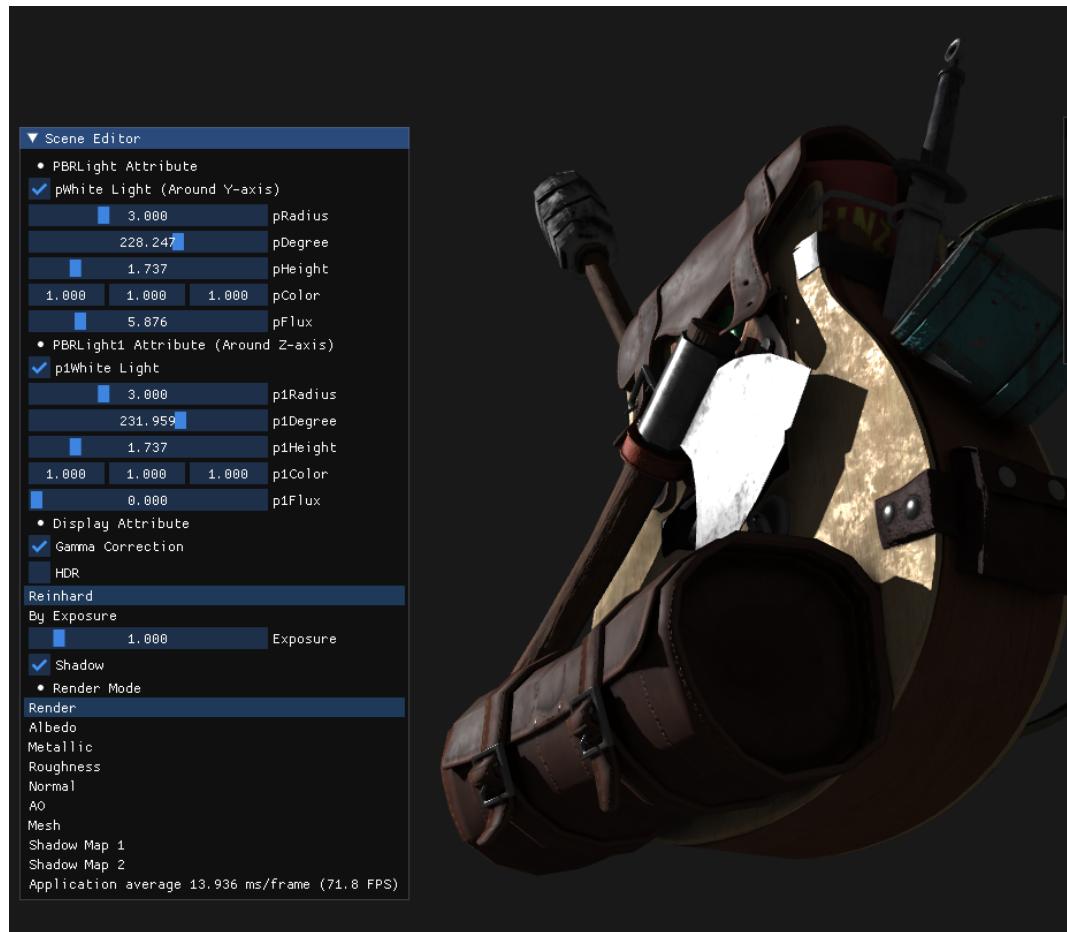
## • 最终结果

光照设置如图所示，可进行调整

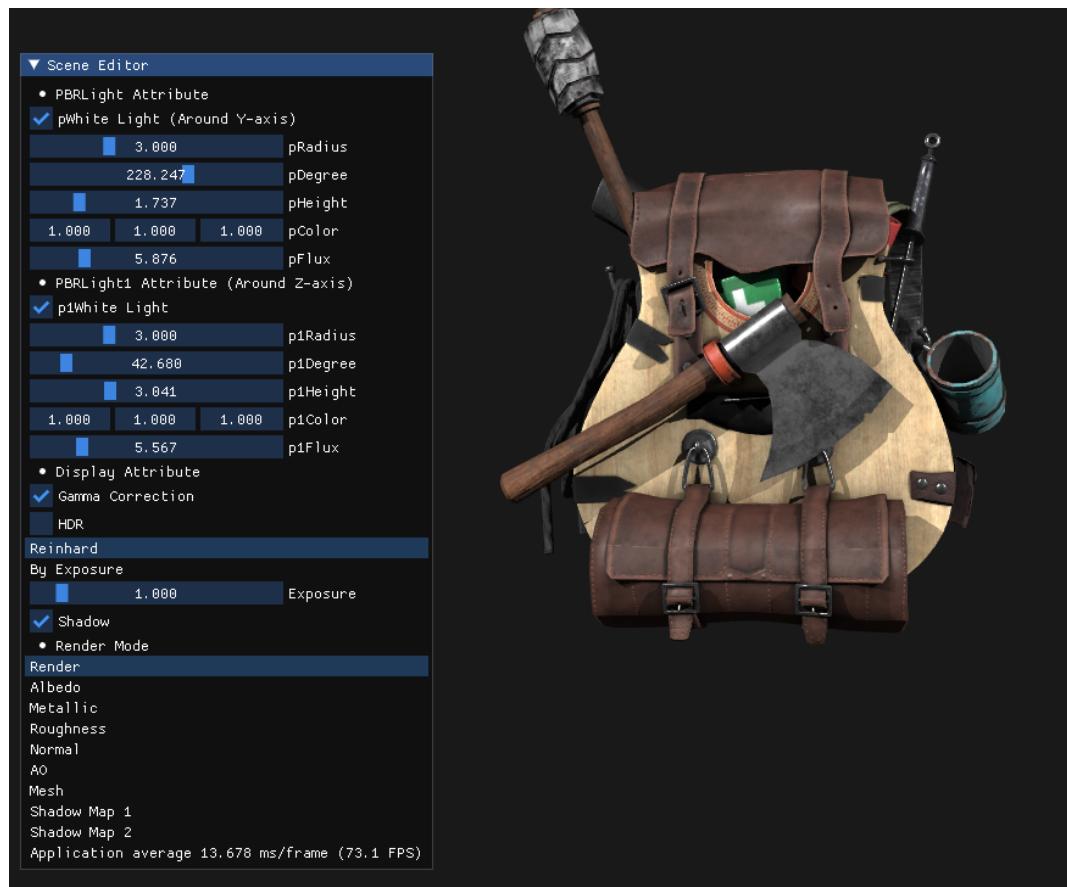
### ◦ 单光源







### ◦ 双光源



可以看到，出现了更加符合真实情况的双阴影



## 个人总结

本次实验，我一以贯之学习了现代OpenGL的绘制方法（过去在CG课上，我使用的是立即渲染模式的旧OpenGL），虽然说实验目的是写Shader，但我感觉占据我时间最多的还是复习图形学知识，学习现代OpenGL，学习各种各样的代码组织方式、Debug方法，还有在网络上寻找合适的轮子，Shader编写本身反倒不是一件很困难的事情。

在做完本次实验后，我也拥有了一个非常基础的PBR渲染器，和之前MIT 6.837课程相比，光栅化渲染是目前绝大多数实时渲染器/视频游戏采用的渲染方式，本次对现代OpenGL的尝试也让我更进一步接触到了现代图形学工业界的“事实标准”，加深了我对图形学，尤其是渲染方向的理解。