Lab3 Report

2300012929 尹锦润

Task 1 Phong Illumination

Simple Part

对于该函数

```
vec3 Shade(vec3 lightIntensity, vec3 lightDir, vec3 normal, vec3 viewDir, vec3 diffuseColor,
vec3 specularColor, float shininess) {
```

对应变量：

- `lightIntensity`：I_d，漫反射光强
- `lightDir`：$\vec{l}$，光线方向
- `normal`：$\vec{n}$，像素法向量
- `viewDir`：$\vec{v}$，人眼方向
- `diffuseColor`：k_d，漫反射颜色
- `specularColor`：k_s，spotlight 颜色
- `shininess`：p，光泽度
- $I_a = 0$ 没有环境光。

对于 `Phong Illumination`，我们有公式（采用朴素方法进行计算）：

$$L_d = k_d(I_a + I_d \max(0, \mathbf{n} \cdot \mathbf{l})) + k_s I_s \max(0, \mathbf{v} \cdot \mathbf{reflect}(-\mathbf{l}, \mathbf{n}))$$

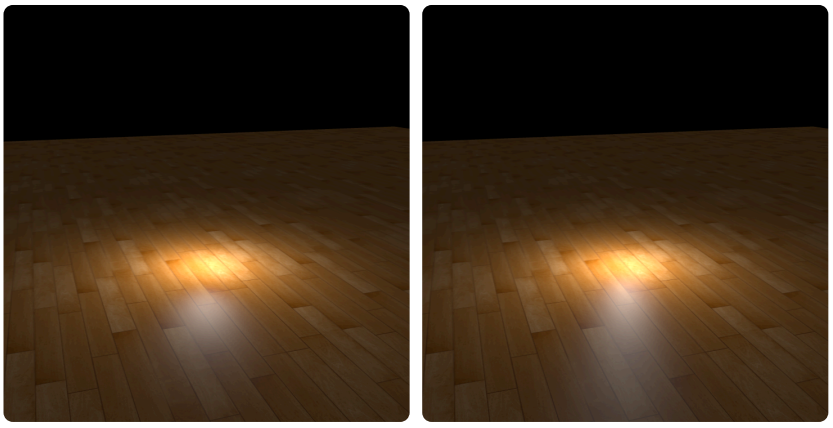
对于 `Blinn-Phong Illumination`，有公式：

$$L_d = k_d(I_a + I_d \max(0, \mathbf{n} \cdot \mathbf{l})) + k_s I_s \max(0, \mathbf{h} \cdot \mathbf{n}), \mathbf{h} = \text{normal}(\mathbf{v} + \mathbf{l})$$

代入即可写出代码：

```
vec3 Shade(vec3 lightIntensity, vec3 lightDir, vec3 normal, vec3 viewDir, vec3 diffuseColor,
vec3 specularColor, float shininess) {
    vec3 color = diffuseColor * lightIntensity * max(0, dot(normal, lightDir));
    color += specularColor * lightIntensity * pow(max(0, u_UseBlinn ? dot(normal,
normalize(lightDir + viewDir)) : dot(reflect(-lightDir, normal), viewDir)), shininess);
    return color;
}
```

效果：



> 顶点着色器和片段着色器的关系是什么样的？顶点着色器中的输出变量是如何传递到片段着色器当中的？

顶点着色器先运行，计算完顶点的信息（位置、法向量、颜色）等等后，再经过面片处理、光栅化等等操作后，将结果交给片段着色器处理。
顶点着色器中的输出变量用 `out` 声明，然后作为片段着色器的 `in` 关键字变量输入。

> 代码中的 `if (diffuseFactor.a < .2) discard;` 这行语句，作用是什么？为什么不能用 `if (diffuseFactor.a == 0.) discard;` 代替？

作用是，对于那些几乎透明的像素，直接删除，不再处理，在几乎不影响视觉效果情况下，较少性能占用。
因为插值存在，有很多 `diffuseFactor.a` 在 $[0,1]$ 之间，如果 `==0` 判断就会导致很多原本可以忽略的没有忽略。

Bonus Part

没写

Task2 Environment Mapping

对于 `skybox.vert`，按照惯例逻辑以及教程，只要

`gl_Position = (u_Projection * u_View * vec4(a_Position, 1.0));`

即可，但是有两个问题：

1. 仔细滑动 skybox，会发现到了模型里面，太小了，因此需要放大
2. 将其放大后，会出现深度原因导致覆盖，因此需要深度设置为 1

可以写出如下代码`gl_Position = (u_Projection * u_View * vec4(a_Position * 100000, 1.0)).xyww;`

对于`envmap.vert`，根据教程

```
vec3 I = normalize(Position - cameraPos);
vec3 R = reflect(I, normalize(Normal));
FragColor = vec4(texture(skybox, R).rgb, 1.0);
```

本质上我们就是将反射的代码迁移到 lab 中，不难写出：

```
vec3 I = normalize(u_ViewPosition - v_Position), R = reflect(I, normalize(normal));
total += texture(u_EnvironmentMap, R).rgb * u_EnvironmentScale;
```

效果：

Task 3 Non-Photorealistic Rendering

根据插值公式：

$$k = \left(\frac{1+\mathbf{l}\cdot\mathbf{n}}{2}\right)k_{cool} + \left(\frac{1-\mathbf{l}\cdot\mathbf{n}}{2}\right)k_{warm}$$

容易写出代码

```
vec3 Shade (vec3 lightDir, vec3 normal) {
    return (1 + dot(lightDir, normal)) / 2 * u_CoolColor + (1 - dot(lightDir, normal)) / 2 * u_WarmColor;
}
```

效果：

参考 `Labs/3-Rendering/CaseNonPhoto.cpp` 中的 `OnRender` 函数，代码是如何分别渲染模型的反面和正面的？

可以看到 OnRender 函数核心部分是：

```
glCullFace(GL_FRONT);
glEnable(GL_CULL_FACE);
for (auto const & model : _sceneObject.OpaqueModels) {
    auto const & material = _sceneObject.Materials[model.MaterialIndex];
    model.Mesh.Draw({ _backLineProgram.Use() });
}
glCullFace(GL_BACK);
```

```
glEnable(GL_DEPTH_TEST);

for (auto const & model : _sceneObject.OpaqueModels) {
    auto const & material = _sceneObject.Materials[model.MaterialIndex];
    model.Mesh.Draw({ material.Albedo.Use(), material.MetaSpec.Use(), _program.Use() });
}

glDisable(GL_DEPTH_TEST);
glDisable(GL_CULL_FACE);
```

而根据

```
_backLineProgram(
    Engine::GL::UniqueProgram({
        Engine::GL::SharedShader("assets/shaders/npr-line.vert"),
        Engine::GL::SharedShader("assets/shaders/npr-line.frag")})),
_program(
    Engine::GL::UniqueProgram({
        Engine::GL::SharedShader("assets/shaders/npr.vert"),
        Engine::GL::SharedShader("assets/shaders/npr.frag") })),
```

可以看到 `_program` 是 npr 画正面，`_backLineProgram`是 `npr-line` 画反面。

因此代码先渲染反面，再画正面，每次画的时候用 `glEnable(GL_CULL_FACE) / glDisable(GL_CULL_FACE)`启用剔除面的功能，用 `glCullFace` 先剔除正/反面。

> `npr-line.vert` 中为什么不简单将每个顶点在世界坐标中沿着法向移动一些距离来实现轮廓线的渲染？这样会导致什么问题？

因为近大远小，会导致出现轮廓线粗细不均的现象。

Task4 Shadow Mapping

在 `phong-shadow.frag` 中，我们通过前面计算的 `pos`，可以通过查询深度贴图获得深度坐标：

```
float closestDepth = texture(u_ShadowMap, pos.xy).r;
```

在 `phong-shadowcubemap.frag` 中，我们依然可以通过 `tolight` 来查询，但是结果要乘上 `u_Farplane`，因为有缩放：

```
float closestDepth = texture(u_ShadowCubeMap, toLight).r * u_FarPlane;
```

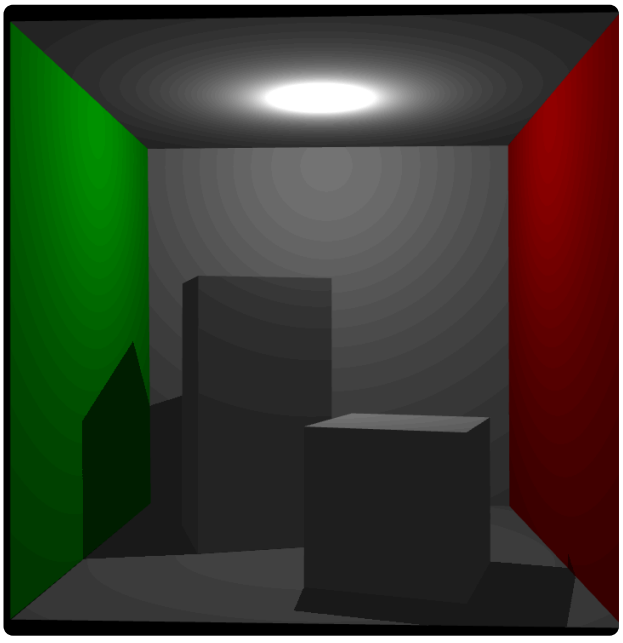
> 想要得到正确的深度，有向光源和点光源应该分别使用什么样的投影矩阵计算深度贴图？

有向光源使用了 `u_lightSpaceMatrix` 这个正交投影矩阵，点光源使用了 6 个 `u_LightMatrices[face]` 计算深度贴图。

> 为什么 phong-shadow.vert 和 phong-shadow.frag 中没有计算像素深度，但是能够得到正确的深度值？

因为通过乘上 `u_lightSpaceMatrix` 将像素坐标转换到光源坐标，已经能通过 `z` 分量计算出深度来了。

效果



Task5 Whitted-Style Ray Tracing

对于 shadow ray 判断时候，不断找到反射 hit position，直到不透明，然后判断是否处在阴影中（line light 类似）：

```
if (light.Type == Engine::LightType::Point) {
    l           = light.Position - pos;
    attenuation = 1.0f / glm::dot(l, l);
    if (enableShadow) {
        auto hit = intersector.IntersectRay(Ray(pos, glm::normalize(l)));
        if (hit.IntersectState && hit.IntersectAlbedo.w < 0.2) {
            hit = intersector.IntersectRay(Ray(hit.IntersectPosition, glm::normalize(l)));
```

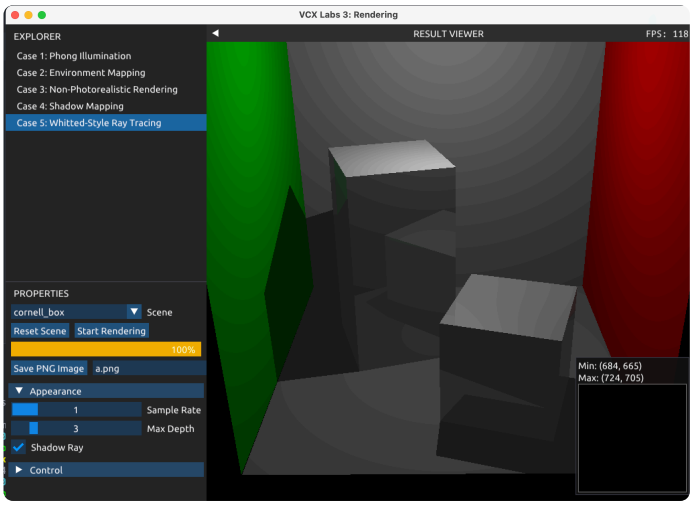
```
        }
        if (hit.IntersectState) {
            glm::vec3 p = hit.IntersectPosition - pos;
            if(glm::dot(p, p) < glm::dot(l, l)) attenuation = 0;
        }
    }
}
```

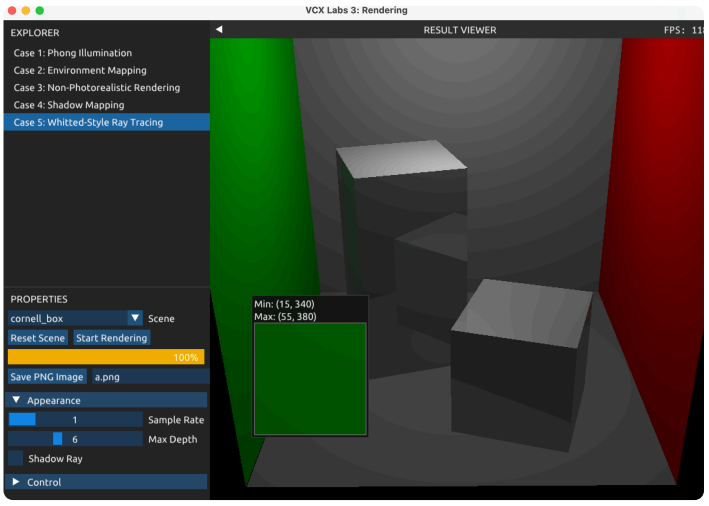
接着求出了 `attenuation` 后将 blinn-phong illumination 接上去即可。

```
glm::vec3 h = glm::normalize(-ray.Direction + glm::normalize(l));
result = kd * intersector.InternalScene->AmbientIntensity;
result += light.Intensity * attenuation * kd * std::max((float)0, glm::dot(n,
glm::normalize(l)));
result += light.Intensity * attenuation * ks * std::pow(std::max((float)0, glm::dot(h, n)),
shininess);
```

接下来的计算折射、反射已经在初始 task 中补全了。

效果：





光线追踪和光栅化的渲染结果有何异同？如何理解这种结果？

相同点：都能很好地展示 scene 的颜色，阴影等等信息，较为真实。

不同点：光栅化无法展示透明物质，无法处理反射、折射等现象，但是比较快。

理解：光线追踪更能反应物理过程，因此更加消耗资源但是效果更好。
