Task1 Phong Illumination

Coding

这一部分我们要计算的核心就是漫反射和镜面反射两部分。这里貌似不需要考虑环境光的影响。对于漫反射，Phong 和 Blinn-Phong的计算公式是一样的，均为$L_d = k_d I_L max(0, \boldsymbol{n} \cdot \boldsymbol{l})$，代码中即

```
vec3 Idiffuse = max(dot(normal, lightDir), 0.0) * lightIntensity * diffuseColor;
```

对于镜面反射的高光则稍稍复杂一些些。

对于标准Phong Illumination，定义$\boldsymbol{r}$为反射的光线，那么镜面反射强度的表达式就是$L_s = k_s I_d max(0, \boldsymbol{r} \cdot \boldsymbol{v})$，反射向量$\boldsymbol{r}$在GLSL中可以用函数`reflect`来计算，因此落实到代码上就是

```
vec3 reflectDir = reflect(-lightDir, normal);
float spec = pow(max(dot(viewDir, reflectDir), 0.0), shininess);
Ispecular = spec * lightIntensity * specularColor;
```

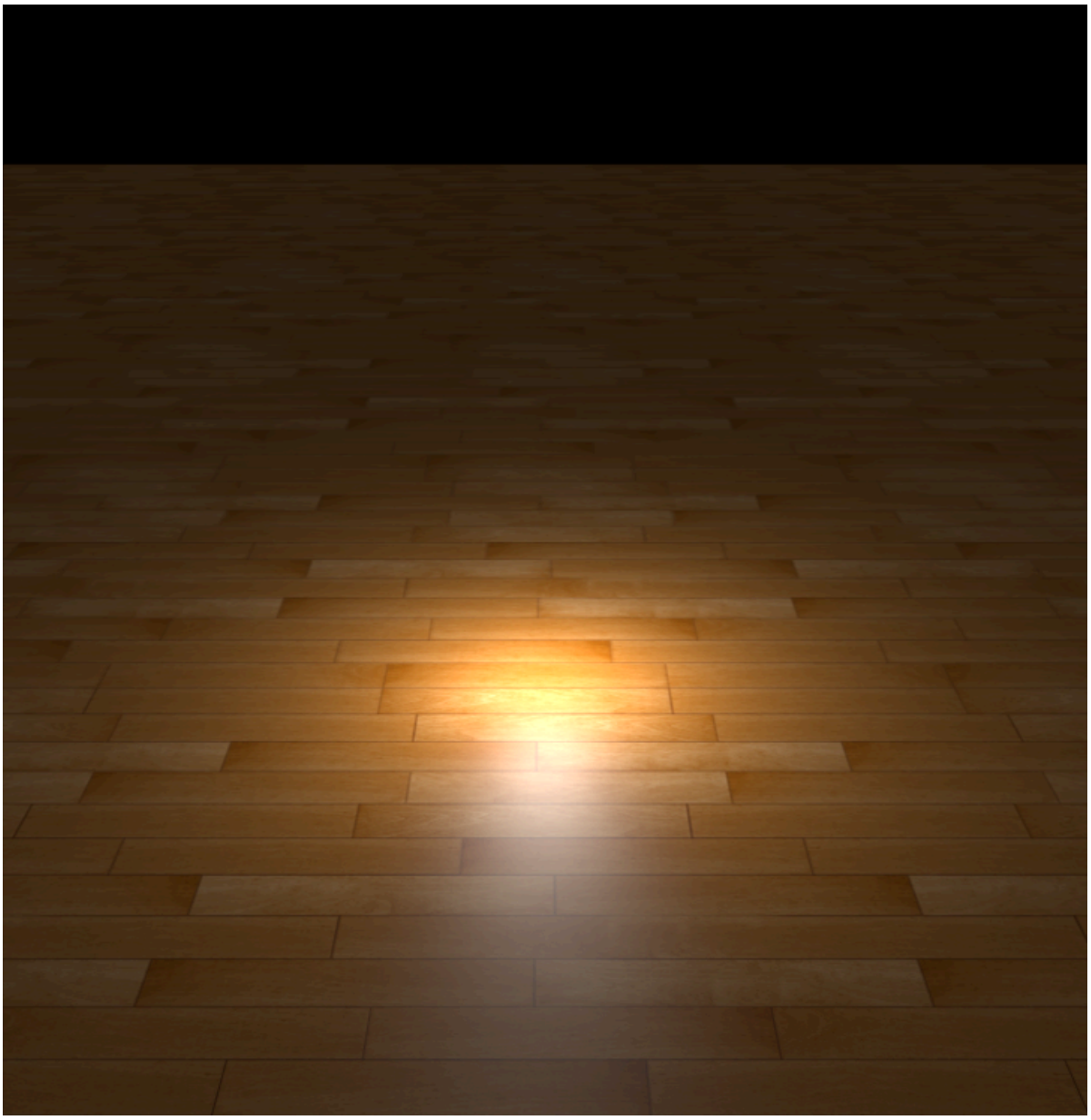
对于Blinn-Phong，我们计算一个半程向量$\boldsymbol{h} = \frac{\boldsymbol{l}+\boldsymbol{v}}{||\boldsymbol{l}+\boldsymbol{v}||}$，于是镜面反射强度$L_s = k_s I_d max(0, \boldsymbol{n} \cdot \boldsymbol{h})$，实现起来就是

```
vec3 halfDir = normalize(lightDir + viewDir);
float spec = pow(max(dot(normal, halfDir), 0.0), shininess);
Ispecular = spec * lightIntensity * specularColor;
```

最后将两个加起来，`return Idiffuse + Ispecular;`即可。

效果图如下：

Phong Illumination



Blinn-Phong Illumination

可以看到，比起标准Phong，Blinn-Phong的高光更加柔和均一。不过很难说哪个效果更好，私以为两者效果差的不多。

Questions

> 1. 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？

在渲染管线中，顶点着色器接收顶点数据，处理后输出数据并进行三角处理和光栅化，随后产生的数据才会送入片段着色器。它们角色不同，在渲染管线中前者是后者的前提，前者进行变换，输出光照强度等后续使用的属性，后者计算每个像素或片段的属性，它们的角色也不一样。

顶点着色器使用 `out` 关键字声明一个变量，并为它赋值。这个变量的数据会从顶点着色器中输出，随后在光栅化中GPU 会将顶点连接起来形成三角形等图元。在光栅化阶段，GPU 会在这些图元表面进行插值计算，对于渲染的每个像素（片段），GPU 会根据顶点坐标的插值计算出该片段的输出变量的值，最后在片段着色器中，使用 `in` 关键字声明一个同名的变量就可以接收这个插值后的数据。

> 2. 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？

相当于如果材质的不透明度过小（也就是材质比较透明），那么光线应当可以透过这个材质，我们就不需要也不应该渲染它的表面上的反射。

由于它是浮点数，因此用`==`来比较不太精确，容易产生精度误差，使得很多接近透明的地方也被绘制。使用`diffuseFactor.a<0.2`是更加稳妥的做法。

Task2 Environment Mapping

这一部分我极大地参照了给出的博客中的内容。博客中提到，

> 我们希望移除观察矩阵中的位移部分，让移动不会影响天空盒的位置向量。
>
> 你可能还记得在基础光照小节中，我们通过取4x4矩阵左上角的3x3矩阵来移除变换矩阵的位移部分。我们可以将观察矩阵转换为3x3矩阵（移除位移），再将其转换回4x4矩阵，来达到类似的效果。
>
> ```
> glm::mat4 view = glm::mat4(glm::mat3(camera.GetViewMatrix()));
> ```
>
> 这将移除任何的位移，但保留旋转变换，让玩家仍然能够环顾场景。
>
> ......
>
> 我们需要欺骗深度缓冲，让它认为天空盒有着最大的深度值1.0，只要它前面有一个物体，深度测试就会失败。
>
> 在坐标系统小节中我们说过，**透视除法**是在顶点着色器运行之后执行的，将gl_Position的`xyz`坐标除以w分量。我们又从深度测试小节中知道，相除结果的z分量等于顶点的深度值。使用这些信息，我们可以将输出位置的z分量等于它的w分量，让z分量永远等于1.0，这样子的话，当透视除法执行之后，z分量会变为`w / w = 1.0`。
>
> ```
> void main()
> {
>  TexCoords = aPos;
>  vec4 pos = projection * view * vec4(aPos, 1.0);
>  gl_Position = pos.xyww;
> }
> ```
>
> 最终的**标准化设备坐标**将永远会有一个等于1.0的z值：最大的深度值。结果就是天空盒只会在没有可见物体的地方渲染了（只有这样才能通过深度测试，其它所有的东西都在天空盒前面）。

这里直接照搬即可。

完成了环境贴图后，我们还需要根据环境贴图来改变物体表面的光照，也即进行环境映射。这里同样根据博客，

> 我们根据观察方向向量$\bar{I}$和物体的法向量$\bar{N}$，来计算反射向量$\bar{R}$。我们可以使用GLSL内建的`reflect`函数来计算这个反射向量。最终的$\bar{R}$向量将会作为索引/采样立方体贴图的方向向量，返回环境的颜色值。最终的结果是物体看起来反射了天空盒。
>
> 因为我们已经在场景中配置好天空盒了，创建反射效果并不会很难。我们将会改变箱子的片段着色器，让箱子有反射性：
>
> ```
> #version 330 core
> ```

```
out vec4 FragColor;

in vec3 Normal;
in vec3 Position;

uniform vec3 cameraPos;
uniform samplerCube skybox;

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

> 我们先计算了观察/摄像机方向向量`I`，并使用它来计算反射向量`R`，之后我们将使用`R`来从天空盒立方体贴图中采样。

博客中采用的是`vec4`，而我们的代码中`total`是`vec3`格式，所以需要做出一点点微调：

```
// your code here
vec3 I = normalize(v_Position - u_ViewPosition);
vec3 R = reflect(I, normalize(v_Normal));
vec3 Fragcolor=texture(u_EnvironmentMap, R).rgb;
total += Fragcolor * u_EnvironmentScale;
```

这样就完成了片段着色器的构建。效果图如下：

Task3 Non-Photorealistic Rendering

Coding

根据我们的课件上给出的公式

$$I = (\frac{1+\hat{\boldsymbol{l}}\cdot\hat{\boldsymbol{n}}}{2})k_{cool} + (1-\frac{1+\hat{\boldsymbol{l}}\cdot\hat{\boldsymbol{n}}}{2})k_{warm}$$

我们在代码中可以直接应用，也即

```
vec3 Shade (vec3 lightDir, vec3 normal) {
    // your code here:
    float NdotL=(1+dot(normal, -lightDir))/2.0;
    return NdotL*u_CoolColor+(1.0-NdotL)*u_WarmColor;
}
```

就可以了。效果图如下：

可以观察到，图中渲染结果中有白色的轮廓线，从冷色到暖色的过渡。由于代码中没有显式设置分界线，所以也没有颜色之间的分界线，不过总体上仍然实现了一种非真实的颜色渲染。

Questions

> 1. 代码中是如何分别渲染模型的反面和正面的？（答案在 `Labs/3-Rendering/CaseNonPhoto.cpp` 中的 `OnRender()` 函数中）

核心在于 `Onrender` 函数中的`glCullFace`和`glEnable(GL_CULL_FACE)`这两个 OpenGL 命令。

- `glEnable(GL_CULL_FACE)`启用面剔除功能。
- `glCullFace(mode)` 指定要剔除（即不渲染）的面。`mode` 可以是 `GL_FRONT`（正面）或 `GL_BACK`（背面）。

当只渲染背面时，首先执行`glCullFace(GL_FRONT);`剔除所有的正面，用`glEnable(GL_CULL_FACE);`激活这个规则，然后用`model.Mesh.Draw({ _backLineProgram.Use() });`进行渲染，它会调用npr-line.vert的顶点着色器渲染边缘。

渲染正面也是类似，首先执行`glCullFace(GL_BACK);`剔除所有的背面，用`glEnable(GL_DEPTH_TEST);`激活这个规则，同时启用深度测试，保证这一步绘制的模型能够正确地覆盖第一步绘制的轮廓线（因为模型的正面比轮廓线的背面**更靠近摄像机**）。然后用`model.Mesh.Draw({ material.Albedo.Use(), material.MetaSpec.Use(), _program.Use() });`渲染正面。`_program`会调用npr.frag着色器，绘制非真实颜色效果。

> 2. `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？

法线外扩是在**世界空间**进行的，但是我们渲染的时候只能在**屏幕空间**中绘制轮廓线。这两者之间有一个投影。具体来说，当物体靠近摄像机时，外扩距离会在屏幕上显得更粗；当物体远离摄像机时，外扩便会被削弱，轮廓线变细。这种非线性的空间关系导致轮廓线的粗细不一致。同时，在锐角转弯之类的尖锐边缘，会导致几何失真。

因此如果直接移动的话会造成轮廓线厚度随视距变化、几何破坏、轮廓不准确等问题。正确的做法还是要基于视角条件$\boldsymbol{v} \cdot \boldsymbol{n} \approx 0$来进行计算。

Task4 Shadow Mapping

Coding

点光源在投影的时候由于是在比较**真实世界中的深度**，所以我们需要`float closestDepth = texture(u_ShadowCubeMap, toLight).r * u_FarPlane;`；而平行光光源由于光源相当于在无限远处，使用的是普通的投影变换，因此我们的深度图记录的和进行比较的均为**已经归一化的数值**，因此只需要写`float closestDepth = texture(u_ShadowMap, pos.xy).r;`就可以了。

效果图如下：

Questions

> 1. 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：透视投影矩阵
- 点光源：正交投影矩阵

> 2. 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?

运行时，深度缓冲会被底层自动设置，没有必要使用显式设置，**除非需要手动修改默认行为。**

Task5 Whitted-Style Ray Tracing

Coding

首先是求光线与三角形的交点，这里我参考了给出的论文[Fast, minimum storage ray/triangle intersection]的算法：

由于三角形上的任意一点$T(u,v)=(1-u-v)p_1+up_2+vp_3$，我们想求解方程$O+tD=T(u,v)$。

整理方程并写成矩阵形式：

$$\begin{bmatrix} -D & p_2-p_1 & p_3-p_1 \end{bmatrix} \begin{bmatrix} t \\ u \\ v \end{bmatrix} = O-p_1$$

我们设$E_1=p_2-p_1, E_2=p_3-p_1, T=O-p_1$，代入即为

$$\begin{bmatrix} -D & E_1 & E_2 \end{bmatrix} \begin{bmatrix} t \\ u \\ v \end{bmatrix} = T$$

使用Cramer法则得到该线性方程组的解：

$$\begin{bmatrix} t \\ u \\ v \end{bmatrix} = \frac{1}{P\cdot E_1} \begin{bmatrix} Q\cdot E_2 \\ P\cdot T \\ E\cdot D \end{bmatrix}$$

其中$P=D\times E_2, Q=T\times E_1$。将如上的结果写成代码就是

```
bool IntersectTriangle(Intersection & output, Ray const & ray, glm::vec3 const &
p1, glm::vec3 const & p2, glm::vec3 const & p3) {
    // your code here
    auto O=ray.Origin, D=ray.Direction;
    auto E1=p2-p1, E2=p3-p1,T=O-p1;
    auto P=glm::cross(D,E2),Q=glm::cross(T,E1);
    auto det=glm::dot(E1,P);
    if (fabs(det)<1e-8) return false;
    float invDet=1.0f/det;
    float epsilon=1e-8f;
    auto u=dot(T,P)*invDet, v=dot(Q,D)*invDet, t=dot(Q,E2)*invDet;
    if (u<epsilon||v<epsilon||u+v>1.0f-epsilon||t<epsilon) return false;
    output.t=t, output.u=u, output.v=v;
    return true;
}
```

然后就是Ray-Tracing和Shadow ray。对于每一条发出的光线，我们计算它的`attenuation`，如果一个物体被遮挡，那么它应当处于阴影中，那么`attenuation`应该置零。

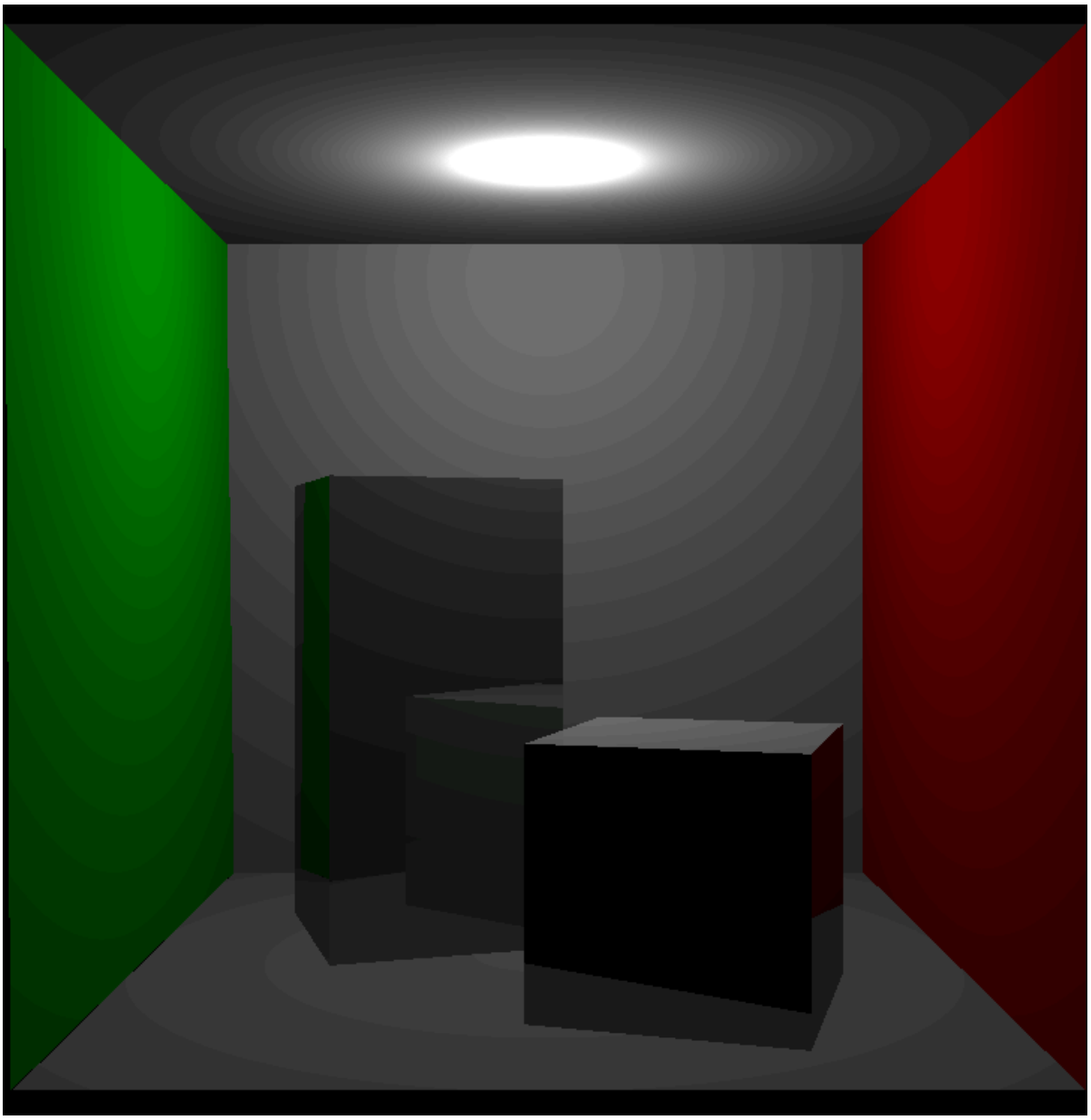
- 点光源情况下，初始的预设值`attenuation=1.0f/glm::dot(l,l);`，这正是点光源的衰减；考虑启用阴影的情况，我们计算光线击中物体的点到着色点的距离`glm::vec3 r=hit.IntersectPosition-pos;`，将这个向量的模和光源到着色点的距离作比较，如果`glm::dot(r,r)<dist*dist`，这就说明物体在光源前方，着色点被遮挡，`attenuation`置零。否则保持`1.0f/glm::dot(l,l)`即可。
- 平行光光源情况下，流程大致相同，区别在于由于平行光光源位于无限远处，所以**只要沿光线方向有任意一次命中**，那么着色点就会被遮挡，不需要判断距离大小。

最后我们采用Blinn-Phong光照模型对着色点进行渲染即可，这里和Task1的思路和代码实现几乎一致。

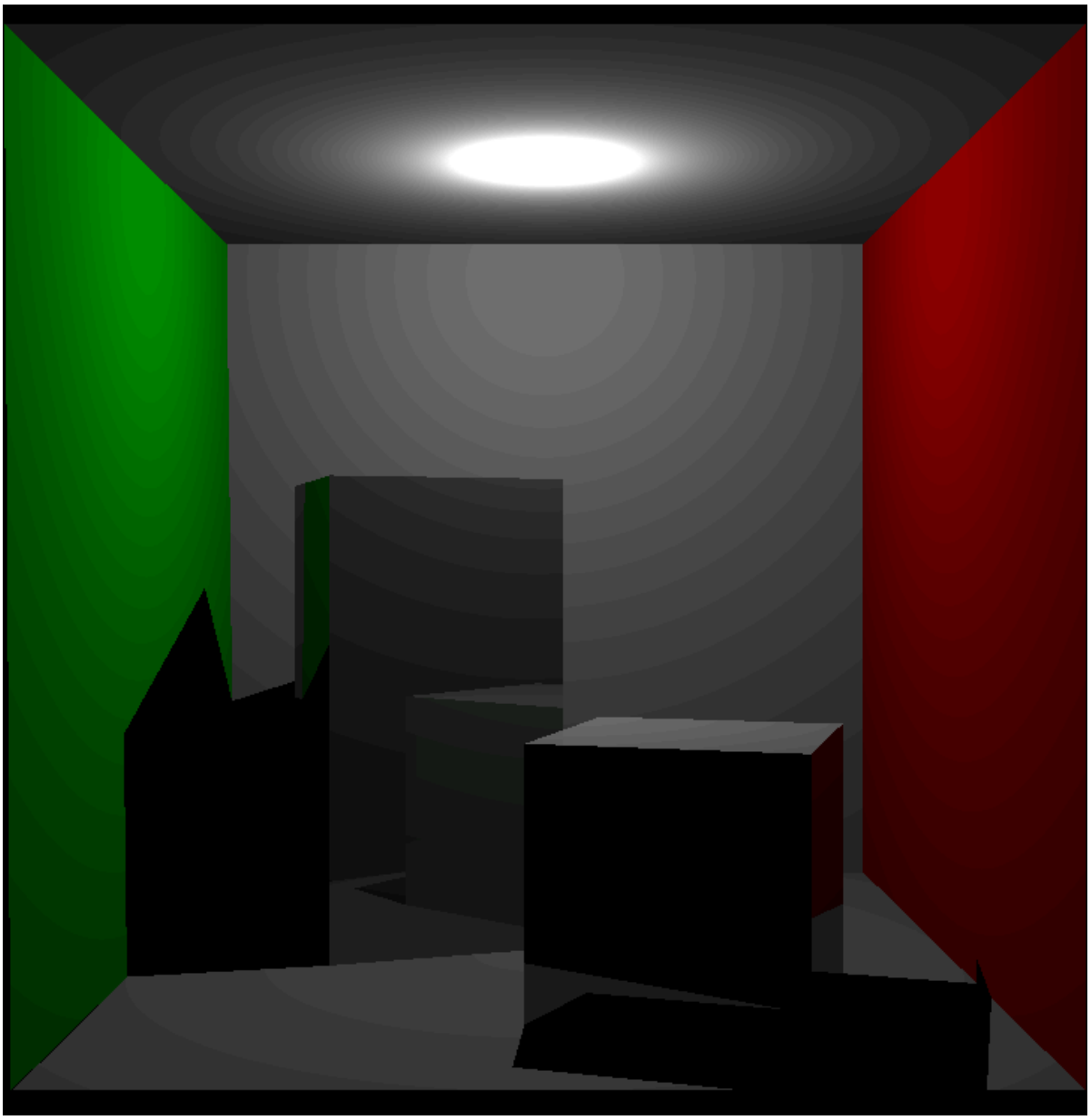
```
glm::vec3 h=glm::normalize(view_dir+glm::normalize(l));
float spec=glm::pow(glm::max(glm::dot(n,h),0.0f),shininess);
float diff=glm::max(glm::dot(n,glm::normalize(l)),0.0f);
result+=attenuation*light.Intensity*(kd*diff+ks*spec);
```

一些效果图如下：

Floor (depth=3)



Cornell Box without shadow (depth=3)



Cornell Box with shadow (depth=3)

Question

> 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？

两者都能大致渲染出一个物理场景，但光线追踪与光栅化的渲染结果的真实感差别很大。光栅化通过将三维场景投影到二维平面并逐像素着色，效率上更快，但其阴影、反射、折射等效果通常依赖近似算法。光线追踪则模拟光线的传播路径与物体交互，能自然产生阴影、全局光照与镜面反射等高真实度效果，但计算量巨大、渲染耗时较长。两者生成的图像主要区别在于光线追踪更接近真实，阴影和细节更丰富，而光栅化结果则显得更“计算机化”与锐利。区别的根源在于：光线追踪追求真实的能量传递过程，而光栅化追求视觉近似与性能平衡。