

VCL Lab3 Report

侯玉杰 2300013108

Task 1: Phong Illumination

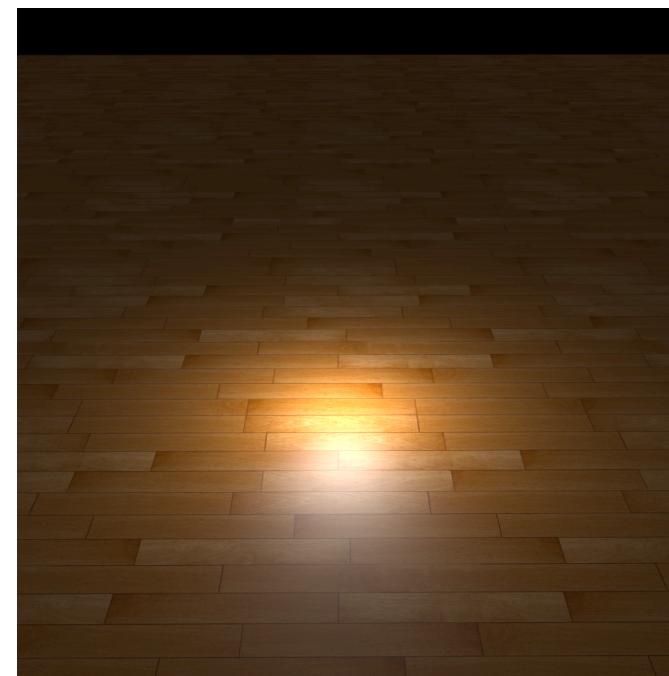
In phong.frag :

使用Blinn-Phong 光照模型时，计算 viewDir 和 lightDir 的半程向量 h 与法向量 normal 的夹角（接近程度）。使用Phong光照模型时，计算光照沿法向量 normal 的反射光线向量与 viewDir 的夹角（接近程度）。

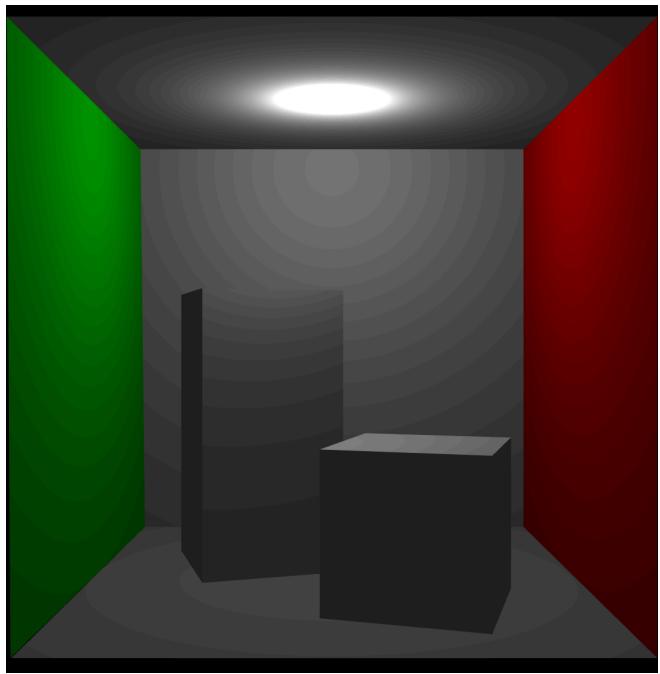
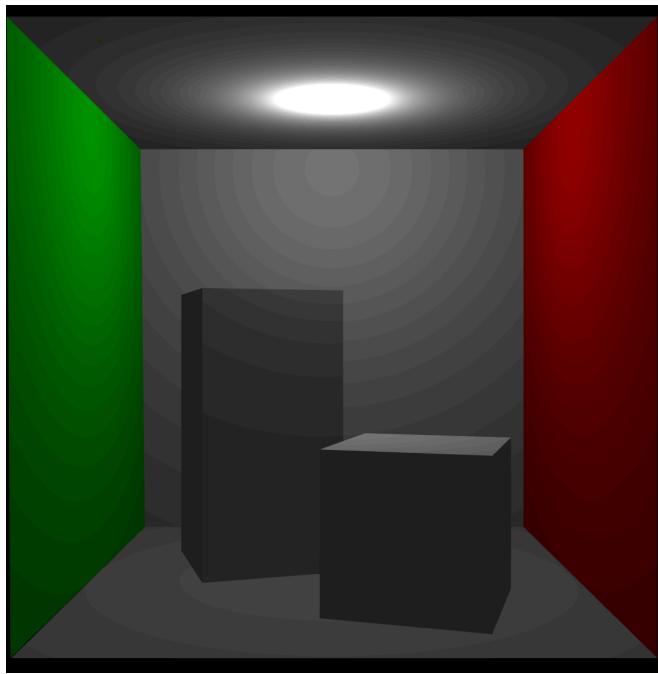
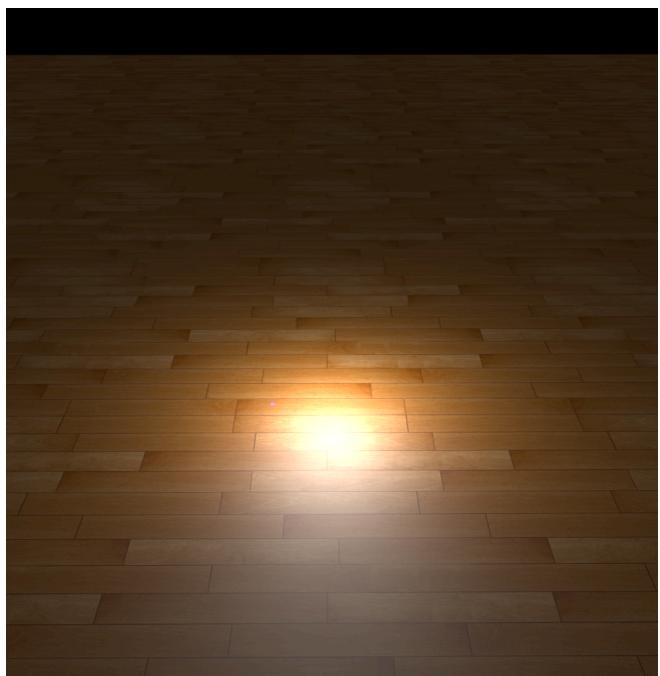
```
vec3 Shade(vec3 lightIntensity, vec3 lightDir, vec3 normal, vec3 viewDir,
           vec3 diffuseColor, vec3 specularColor, float shininess) {
    // your code here:
    vec3 res = vec3(0);
    if(!u_UseBlinn)
        res = diffuseColor*(lightIntensity * max(0,dot(normal, lightDir)))
        + specularColor*(lightIntensity * pow(max(0, dot(reflect(-lightDir, normal), viewDir)), shininess));
    else
        res = diffuseColor*(lightIntensity * max(0,dot(normal, lightDir)))
        + specularColor*(lightIntensity * pow(max(0, dot(normal,normalize(viewDir+lightDir))), shininess));
    return res;
}
```

Result (Shininess = 10.0, Ambient = 1.00x, Attenuation = 2)

Phong Model



Blinn-Phong Model

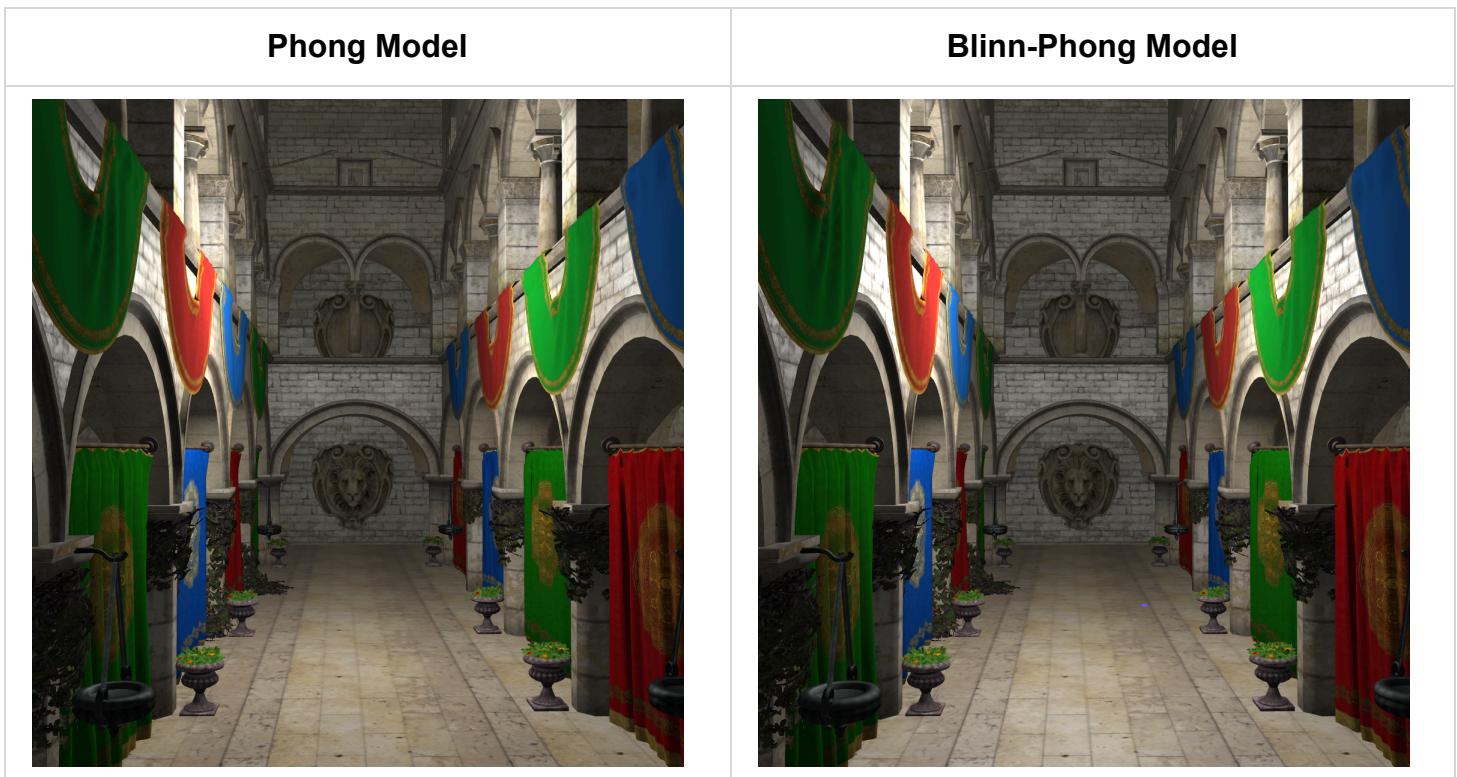


Phong Model



Blinn-Phong Model





问题回答

1.顶点着色器接收顶点数据，并将处理过的顶点参数作为输出，片段着色器则接收顶点着色器的输出作为输入，并输出颜色变量。这个过程需要通过 `in` , `out` 关键字，顶点着色器指定输出变量，片段着色器指定相应的输入变量，两个变量匹配成功后就可传递。

2.这一语句的目的是丢弃diffuse map颜色alpha值太低的像素或片段，确保近乎透明的像素最终不会被渲染，提升渲染效率。0.2是一个对于近乎透明像素alpha值的合理的阈值。, 然而，因为浮点数精准度问题alpha值等于0的像素可能在存储时并不完全等于0，而是 `0.0001` 等的数值，如果设置为要求alpha刚好等于0才丢弃，那么以上的像素并不会被过滤掉，最后依然被渲染。设为小于0.2则符合我们的需求。

Task 1 Bonus: Bump Mapping

In phong.frag :

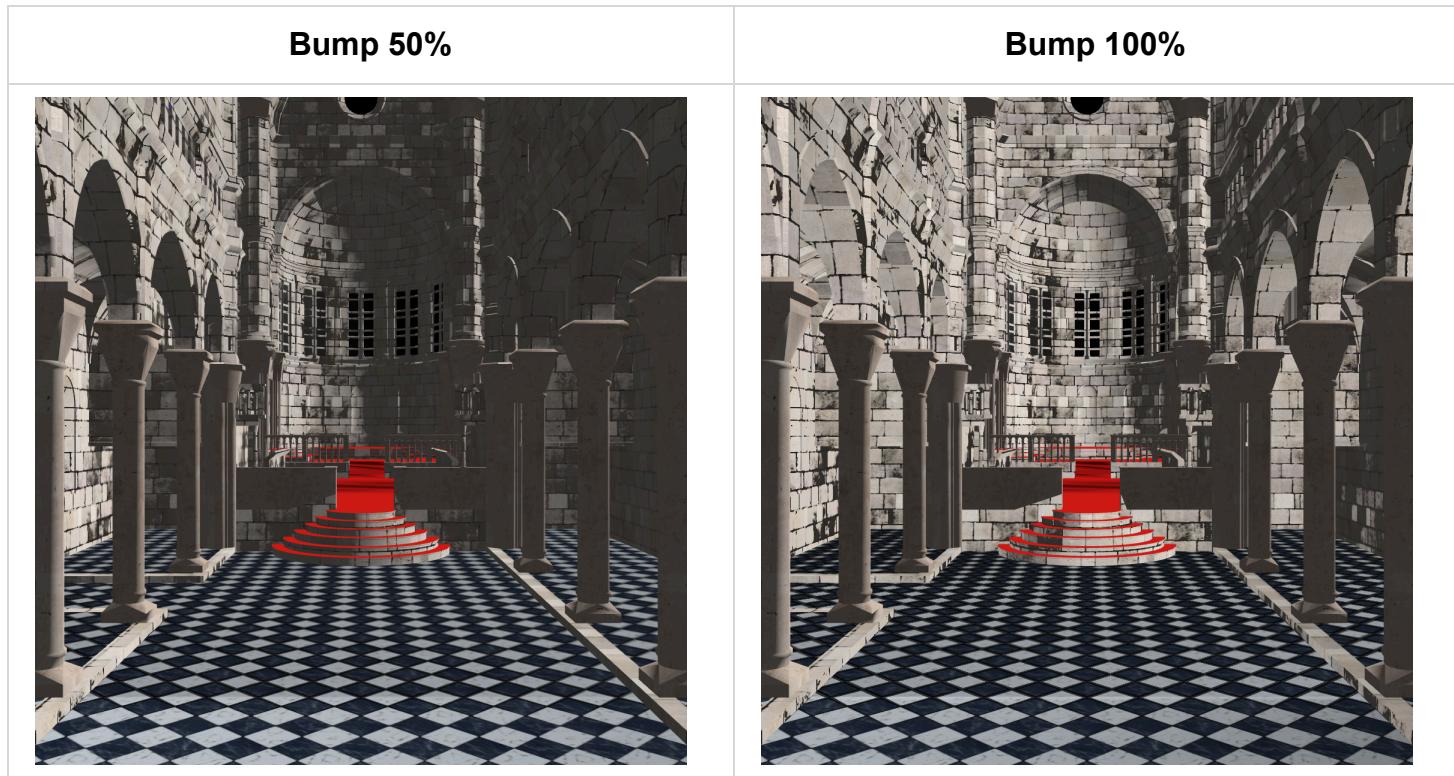
```
vec3 GetNormal() {
    // Bump mapping from paper: Bump Mapping Unparametrized Surfaces on the GPU
    vec3 vn = normalize(v_Normal);
    // your code here:

    vec3 bumpNormal = vn;
    vec3 posdx = dFdx(v_Position);
    vec3 posdy = dFdy(v_Position);
    vec3 r1 = cross(posdy, vn);
    vec3 r2 = cross(vn, posdx);
    float det = dot(posdx, r1);
    float Hll = texture(u_HeightMap, v_TexCoord).x;
    float Hlr = texture(u_HeightMap, dFdx(v_TexCoord.xy)).x;
    float Hull = texture(u_HeightMap, dFdy(v_TexCoord.xy)).x;
    vec3 surf_grad = sign(det) * (Hlr - Hll) * r1 + sign(det) * (Hull - Hll) * r2;
    bumpNormal = normalize(abs(det)*vn - surf_grad);

    return bumpNormal != bumpNormal ? vn : normalize(vn * (1. - u_BumpMappingBlend)
        + bumpNormal * u_BumpMappingBlend);
}
```

根据StackExchange上的回答，凹凸映射模拟了平面的位移，即着色器假设像素在法上要更靠后。接着我们可以通过比较邻点像素的在各个方向的高度差，计算斜率建立梯度，并计算此像素新的法向量，而获取邻点的高度可以通过 `texture()` 对 `u_HeightMap` 进行采样。注意到 `u_BumpMappingBlend` 即为可视化界面中可调整的Bump，以此调节表面凹凸程度或者平滑度。

Result



Task 2: Environment Mapping

In `skybox.vert` :

根据参考资料，首先计算位置数据并赋值给 `vec4` 型变量 `gl_Position`，这里我们将顶点位置数据乘以一个较大的常数，以放大其空间大小，从而在屏幕上形成“远近”的效果。根据透视线除法，为了标准化设备坐标，我们将 `gl_Position` 的 `xyz` 坐标除以 `w` 分量。

```
void main() {  
    v_TexCoord = a_Position;  
    // your code here  
    vec4 pos = u_Projection * u_View * vec4(a_Position*10000, 1.0);  
    gl_Position = pos.xyww;  
}
```

In `envmap.frag` :

根据参考资料，按照公式创建反射效果。

```
// Environment component  
// your code here  
vec3 R = reflect(-viewDir, normal); //反射向量  
vec3 envColor = texture(u_EnvironmentMap, R).rgb; //从天空盒采样  
total += envColor * u_EnvironmentScale;
```

Result

teapot	bunny
	

Task 3: Non-Photorealistic Rendering

In npr.frag :

```
vec3 Shade (vec3 lightDir, vec3 normal) {
    // your code here:
    float dotProduct = dot(normal, lightDir);
    float warm_weight = 0.0;
    float cool_weight = 0.0;
    if (dotProduct < 0.01) {
        warm_weight = 0.15;
        cool_weight = 0.85;
    } else if (dotProduct < 0.65) {
        warm_weight = 0.6;
        cool_weight = 0.4;
    } else {
        warm_weight = 1.0;
        cool_weight = 0.0;
    }
    vec3 res = u_WarmColor * warm_weight + u_CoolColor * cool_weight;
    //vec3 res = u_CoolColor*(1+dotProduct)*0.5 + u_WarmColor*(1-dotProduct)*0.5;
    return res;
}
```

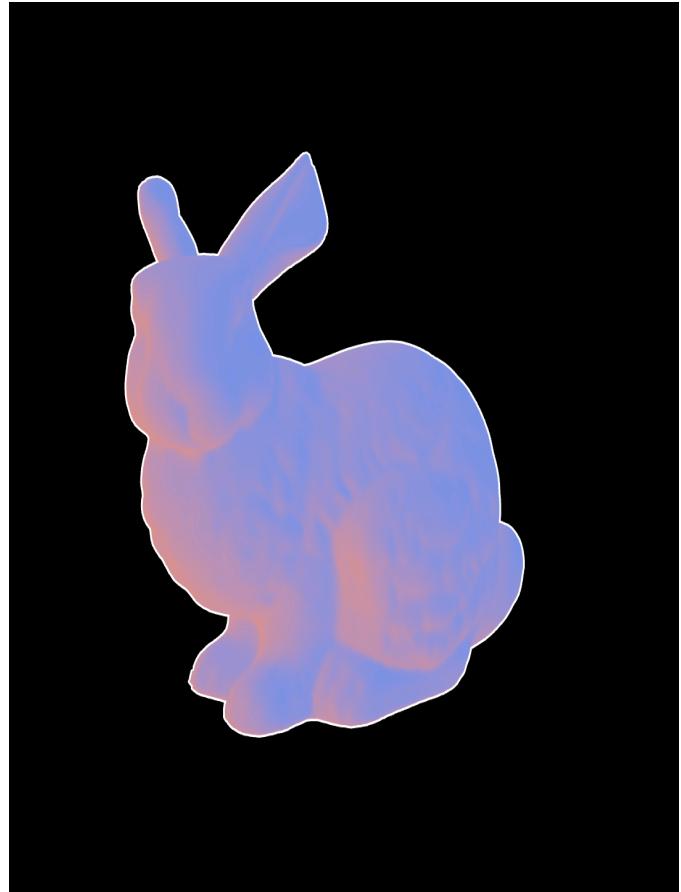
Result

若使用颜色插值得到渐变效果，即代码中注释掉的插值公式，效果如下：

teapot



bunny

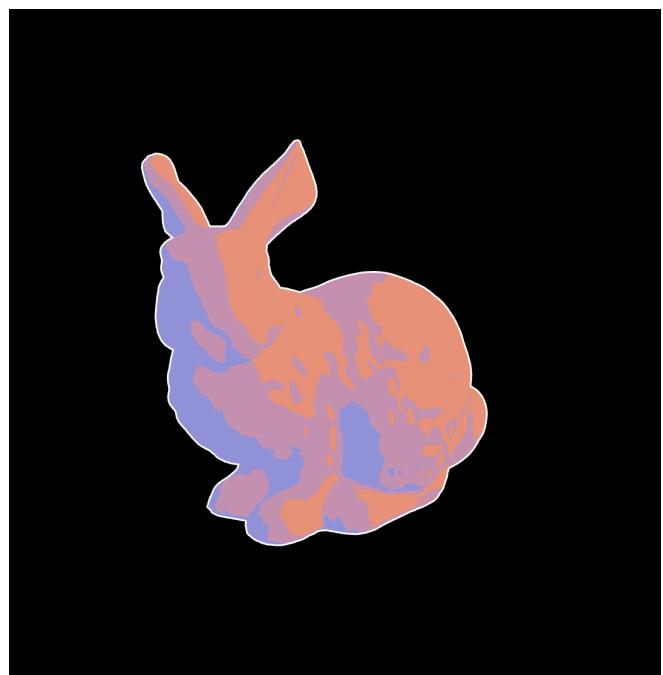


若为冷暖色的系数进行分段处理来离散化实现颜色分界，效果如下：

teapot



bunny



问题回答

1. 在 `OnRender` 中，

以下部分渲染了模型的反面 (back-facing facet) , 首先使用 `glCullFace(GL_FRONT)` 和 `glEnable(GL_CULL_FACE)` 启用将正面剔除, 然后通过逐面片进行反面纯白色渲染。

```
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() });
}
```

以下部分渲染了模型的正面 (front-facing facet) , 首先使用 `glCullFace(GL_BACK)` 将反面剔除, 然后通过逐面片进行正面的颜色渲染。

```
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() });
}
```

2. 首先, 沿着每个顶点移动的计算量太大, 导致渲染效率低下。其次, 这可能导致轮廓线偏离原始模型, 因为在世界坐标系中法线的精度可能不足以代表模型的轮廓, 沿着法向位移可能使模型的几何形状发生偏差。这也有可能导致深度误差, 如果沿法向量偏移顶点, 可能出现自相交, 导致深度计算错误, 发生透视现象。

Task 4: Shadow Mapping

In `phong-shadow.frag` :

在透视投影下取得最近点的深度。

```

float Shadow(vec4 lightSpacePosition, vec3 normal, vec3 lightDir) {
    ...
    // your code here: closestDepth = ?
    float closestDepth = 0;
    closestDepth = texture(u_ShadowMap, pos.xy).r;
    // your code end
    ...
}

vec3 Shade(vec3 lightIntensity, vec3 lightDir, vec3 normal, vec3 viewDir,
            vec3 diffuseColor, vec3 specularColor, float shininess) {
    // your code here:
    vec3 res = vec3(0);
    if(!u_UseBlinn)
        res = diffuseColor*(lightIntensity * max(0,dot(normal, lightDir)))
        + specularColor*(lightIntensity * pow(max(0, dot(reflect(-lightDir, normal), viewDir)), shininess));
    else
        res = diffuseColor*(lightIntensity * max(0,dot(normal, lightDir)))
        + specularColor*(lightIntensity * pow(max(0, dot(normal,normalize(viewDir+lightDir))), shininess));
    return res;
}

```

In phong-shadowcubemap.frag :

```

float Shadow(vec3 pos, vec3 lightPos) {
    // your code here: closestDepth = ?
    float closestDepth = 0;
    vec3 fragToLight = pos - lightPos;
    closestDepth = texture(u_ShadowCubeMap, fragToLight).r; // 从立方体纹理中采样
    closestDepth *= u_FarPlane; // 变换回原来的值
    // your code end
    ...
}

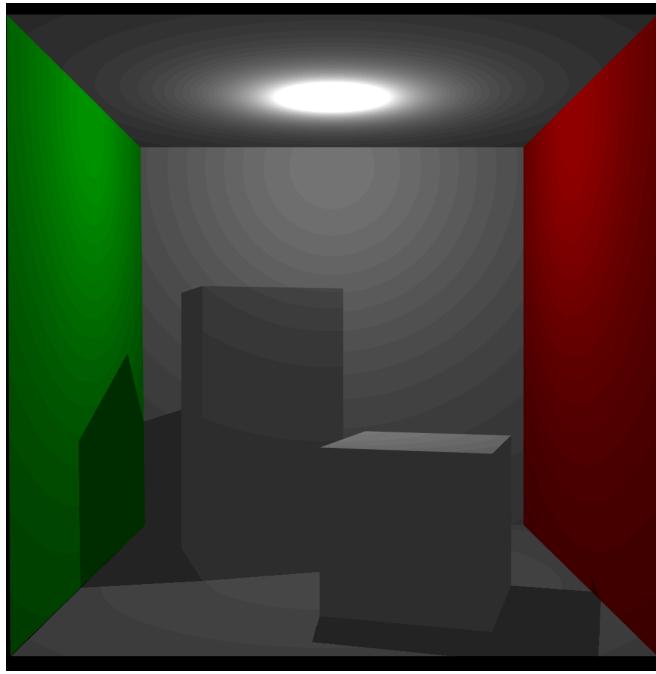
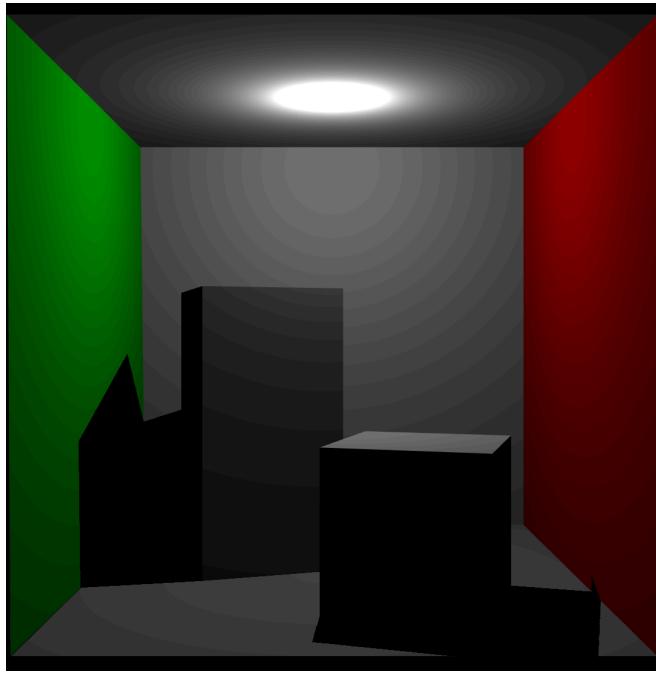
```

Result

0.00x Ambient



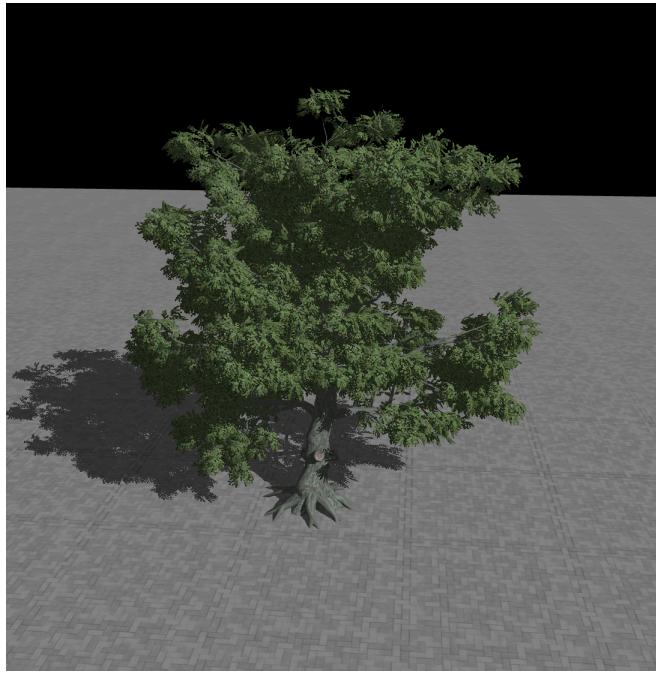
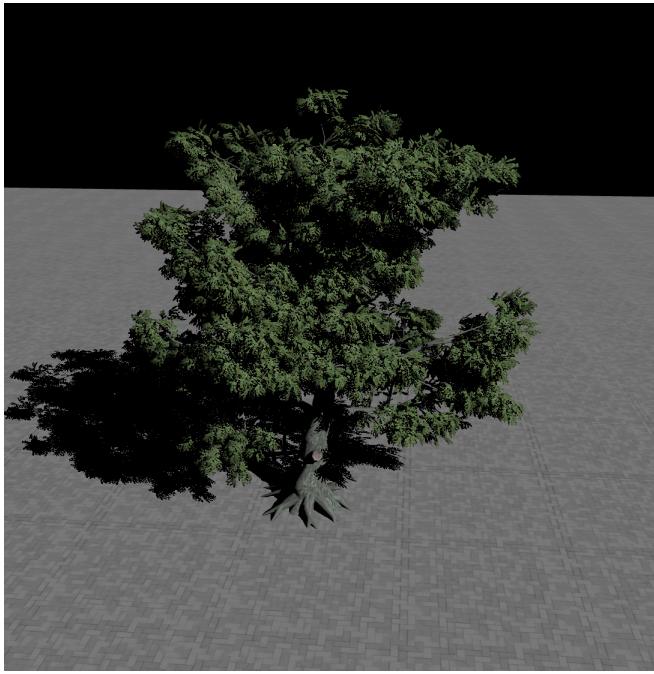
2.00x Ambient

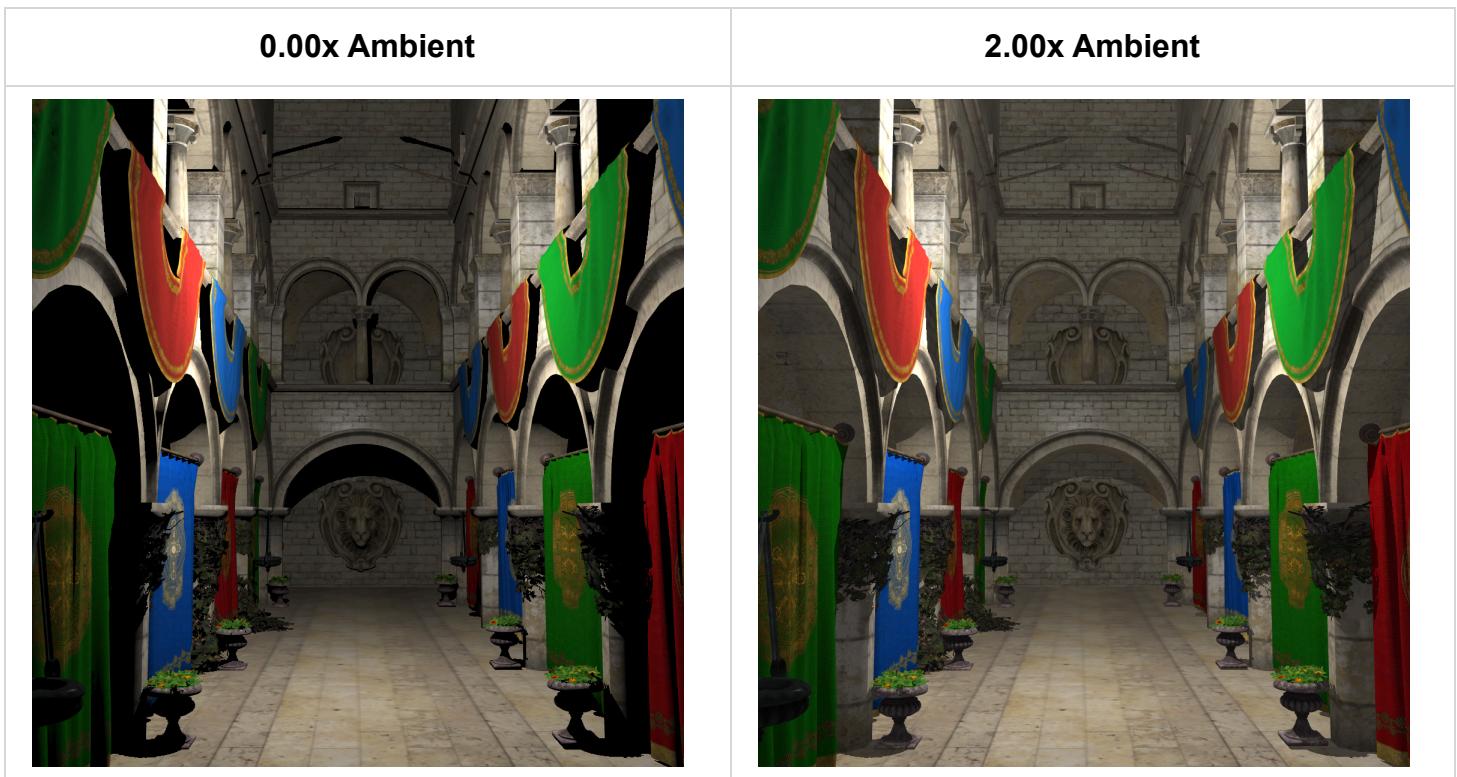


0.00x Ambient



2.00x Ambient





问题回答

1. 有向光源使用正交投影矩阵，点光源使用透视投影矩阵，而使用透视矩阵时需要先将非线性深度值转变为线性深度值。
2. 因为这两个着色器都使用了光的透视图所存储的深度信息，只用把顶点通过变换矩阵变换到光空间即可生成深度贴图，无需计算每个像素的深度。

Task 5: Whitted-Style Ray Tracing

In `IntersectTriangle()` :

根据讲义中推导的公式，我们有

$$\begin{bmatrix} t \\ u \\ v \end{bmatrix} = \frac{1}{\mathbf{P} \cdot \mathbf{E}_1} \begin{bmatrix} \mathbf{Q} \cdot \mathbf{E}_2 \\ \mathbf{P} \cdot \mathbf{T} \\ \mathbf{Q} \cdot \mathbf{D} \end{bmatrix}$$

$$\mathbf{P} = \mathbf{D} \times \mathbf{E}_2, \mathbf{Q} = \mathbf{E}_1 \times \mathbf{T}$$

$$\mathbf{E}_1 = \mathbf{V}_1 - \mathbf{V}_0, \mathbf{E}_2 = \mathbf{V}_2 - \mathbf{V}_0, \mathbf{T} = \mathbf{O} - \mathbf{V}_0$$

由公式可计算出 t, u, v , 检查计算得的三角形重心坐标 u, v 是否满足 $u \geq 0, v \geq 0, u + v \leq 1$ 且 $0 \leq t < \text{inf}$, 满足则设置 `output` 的 t, u, v 并返回 `true` 表示光线与三角形面片相交, 否则返回 `false`。

```
glm::vec3 edge1 = p2 - p1;
glm::vec3 edge2 = p3 - p1;
glm::vec3 T = ray.Origin - p1;
glm::vec3 P = glm::cross(ray.Direction, edge2);
glm::vec3 Q = glm::cross(T, edge1);
float a = glm::dot(edge1, P);
float f = 1.0f / a;
float u = f * glm::dot(T, P);
float v = f * glm::dot(ray.Direction, Q);
float t = f * glm::dot(edge2, Q);
if (u < 0.0f || v < 0.0f || u + v > 1.0f) {
    return false;
}
if (t >= 0.0f && t < INFINITY) {
    output.t = t;
    output.u = u;
    output.v = v;
    return true;
}
return false;
```

In RayTrace() :

这个函数的主循环遍历最大深度，处理每一层的反射折射。
在我们填充的部分，首先计算并添加环境光源。

```
glm::vec3 ambient = intersector.InternalScene->AmbientIntensity * kd;
result += ambient;
```

在对场景中光源的遍历中，当前光源若是点光源或有向光源，当阴影开启时，发射一条从交点指向光源的射线，若射线与其他物体相交而且alpha<0.2, 则认为该射线被遮挡，则继续发射另一条光线直到不被遮挡。若交点到物体的距离小于光源到交点的距离，则意味着光线的路径被物体遮挡，设置距离衰减为0，即当前点处于阴影中。

```

if (enableShadow) {
    // your code here
    Ray shadowRay(pos, glm::normalize(l)); // 射线
    auto hit = intersector.IntersectRay(shadowRay); //求交
    while (hit.IntersectState && hit.IntersectAlbedo.w < 0.2f) { // 被遮挡
        Ray secondRay(hit.IntersectPosition, glm::normalize(l));
        hit = intersector.IntersectRay(secondRay);
    }
    if (hit.IntersectState) {
        glm::vec3 intersectPt = hit.IntersectPosition - pos;
        if(glm::dot(intersectPt, intersectPt) < glm::dot(l, l)) // 距离
            attenuation = 0.0f;
    }
}

```

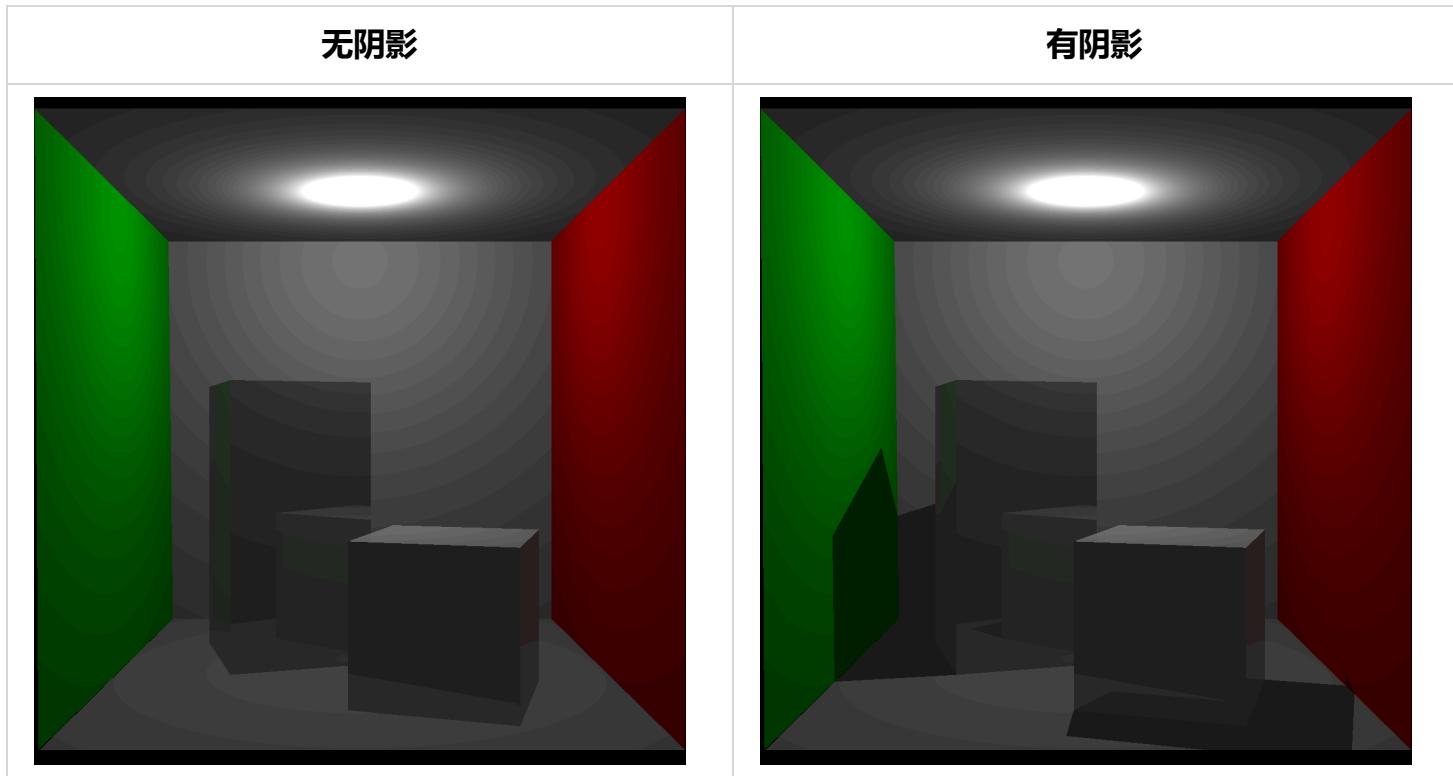
使用 *Phong* 着色模型，计算出漫反射以及镜面反射并添加到 `result` 中。

```

glm::vec3 viewDir = -ray.Direction;
glm::vec3 diffuse = kd * glm::max(glm::dot(glm::normalize(l),glm::normalize(n)), 0.0f)
                    * light.Intensity * attenuation;
glm::vec3 specular = ks * glm::pow(glm::max(0.0f, glm::dot(glm::normalize(n),
                glm::normalize(viewDir + glm::normalize(l)))), shininess) * light.Intensity
result += diffuse + specular;

```

Result (Sample Rate = 1, Max Depth = 3)



问题回答

1. 在模拟光照效果上，光线追踪与光栅化渲染效果相近，但光线追踪在模拟光照明影，镜面反射上效果更好，因为光线追踪通过模拟真实的光线传播，能对反射，折射等现象进行自然且快速的模拟，光栅化则在计算时较复杂。但是，光线追踪在渲染速度上要比光栅化要慢得多，因为要对巨量光线进行求交运算，光栅化则只用对每个像素进行处理。

End of Report.