

可视计算与交互概论-Lab3

xTryer

2024/11/19

1 Task 1: Phong Illumination

1.1 Phong/Blinn-Phong

思路:反射模型中的光强分为两个部分,一个部分为 diffuse_reflection(漫反射),一个部分为 specular_reflection(镜面反射)。

其中漫反射公式为 $L_d = k_d(I_a + I_d \max(0, \mathbf{n} \cdot \mathbf{l}))$, (我们只需要计算后面漫反射平行光的部分, 因为前面的环境光已经在 main 函数中加入了)

而镜面反射公式根据选择 Phong 模型还是 Blinn-Phong 模型分为两种计算方法,

其中 Phong 模型: $L_s = k_s I_d \max(0, \mathbf{r} \cdot \mathbf{v})^p$, r 表示反射光线的单位向量, v 表示视线方向的单位向量

而对于 Blinn-Phong 模型: $L_s = k_s I_d \max(0, \mathbf{n} \cdot \mathbf{h})^p$, n 表示物体表面的法线方向的单位向量, h 表示由光线方向和视线方向计算得到的单位半程向量。

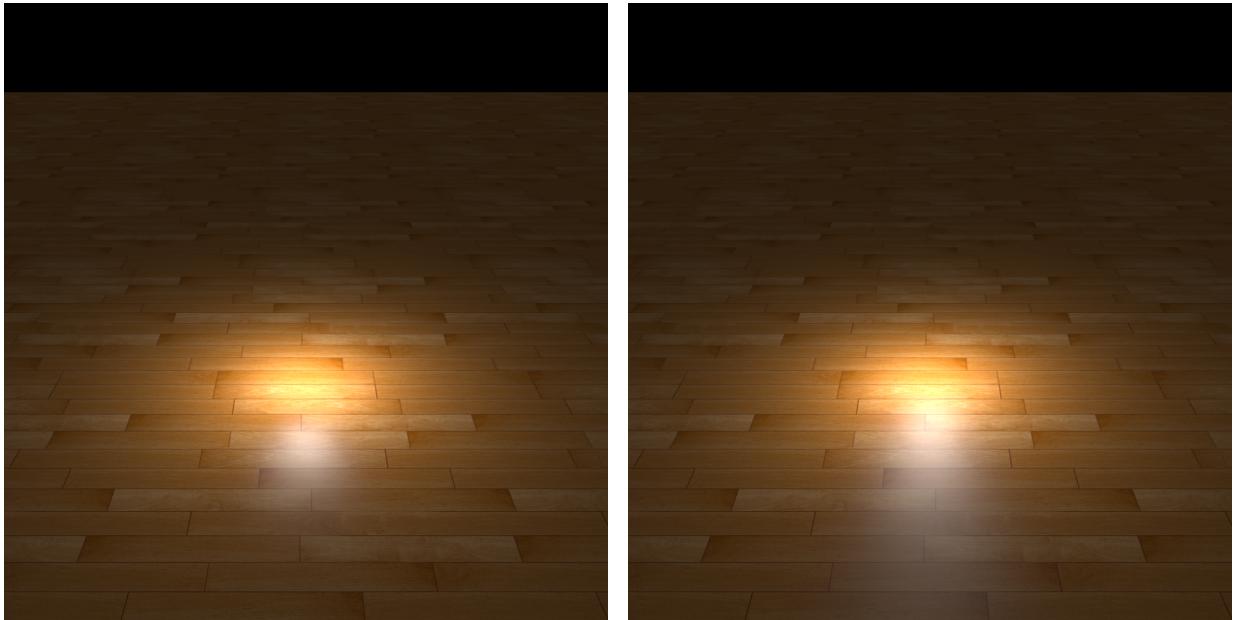
```
1 vec3 Shade(vec3 lightIntensity, vec3 lightDir, vec3 normal, vec3 viewDir, vec3 diffuseColor,
2             vec3 specularColor, float shininess) {
3                 // your code here:
4
5                 vec3 normalized_lightdir = normalize(lightDir);
6                 vec3 normalized_normaldir = normalize(normal);
7                 vec3 normalized_viewdir=normalize(viewDir);
8
9                 // diffuse_light
10                float diffuse_cos = max(0.0, dot(normalized_lightdir, normalized_normaldir));
11                vec3 diffuse_light = diffuseColor*(diffuse_cos*lightIntensity);
12
13                //specular_light
14                //Phong:
15                vec3 reflect_light_dir_Phong = dot(normalized_lightdir, normalized_normaldir)*
16                normalized_normaldir*2.0-normalized_lightdir;
17                //Blinn-Phong:
18                vec3 normalized_halfdir = normalize(normalized_lightdir+normalized_viewdir);
19
20                float specular_degree1 = pow(max(0.0, dot(reflect_light_dir_Phong, normalized_viewdir)), shininess);
21                float specular_degree2 = pow(max(0.0, dot(normalized_halfdir, normalized_normaldir)), shininess);
22
23                vec3 specular_light1 = specularColor*specular_degree1*lightIntensity;
24                vec3 specular_light2 = specularColor*specular_degree2*lightIntensity;
```

```

23
24
25
26     return (u_UseBlinn==true)?diffuse_light+specular_light2:diffuse_light+specular_light1;
27 }

```

获得结果图:



(a) floor_Phong(bump=0)

(b) floor_Blinn-Phong(bump=0)

图 1: Task 1: Phong Illumination

1.2 Task 1-Questions

1.2.1 Vertex Shader&Fragment Shader

顶点着色器输入每个顶点的数据，如顶点位置，顶点法线等，根据这些输入可以计算出经过变换的顶点位置，颜色，纹理坐标等，然后把计算得到的数据进行输出，传递给片段着色器进行进一步的计算，片段着色器根据由顶点着色输入的数据如插值后的顶点纹理坐标，顶点的法向量，顶点的位置和一些光照参数来计算最终片段的着色。

顶点着色器 phong.vert 可以通过在文件头部声明'out' 来对计算得到的数据进行输出，如

```

1 layout(location = 0) out vec3 v_Position;
2 layout(location = 1) out vec3 v_Normal;
3 layout(location = 2) out vec2 v_TexCoord;

```

而相应的，片段着色器 phong.frag 可以在文件头部声明'in' 来接收顶点着色器的输入，如

```

1 layout(location = 0) in vec3 v_Position;
2 layout(location = 1) in vec3 v_Normal;
3 layout(location = 2) in vec2 v_TexCoord;

```

1.2.2 discard with diffuseFactor.a

这行语句的作用是检查漫反射颜色的 alpha(透明度) 分量是否小于 0.2, 如果小于 0.2 那么这个像素片段就会被丢弃, 不会被写入颜色帧缓冲区中, 最终不会被渲染。

之所以不能使用 if (diffuseFactor.a == 0.) discard 代替, 是因为

1. 通过浮点计算得出恰好为 0. 的值很困难, ==0. 的判断是不安全的

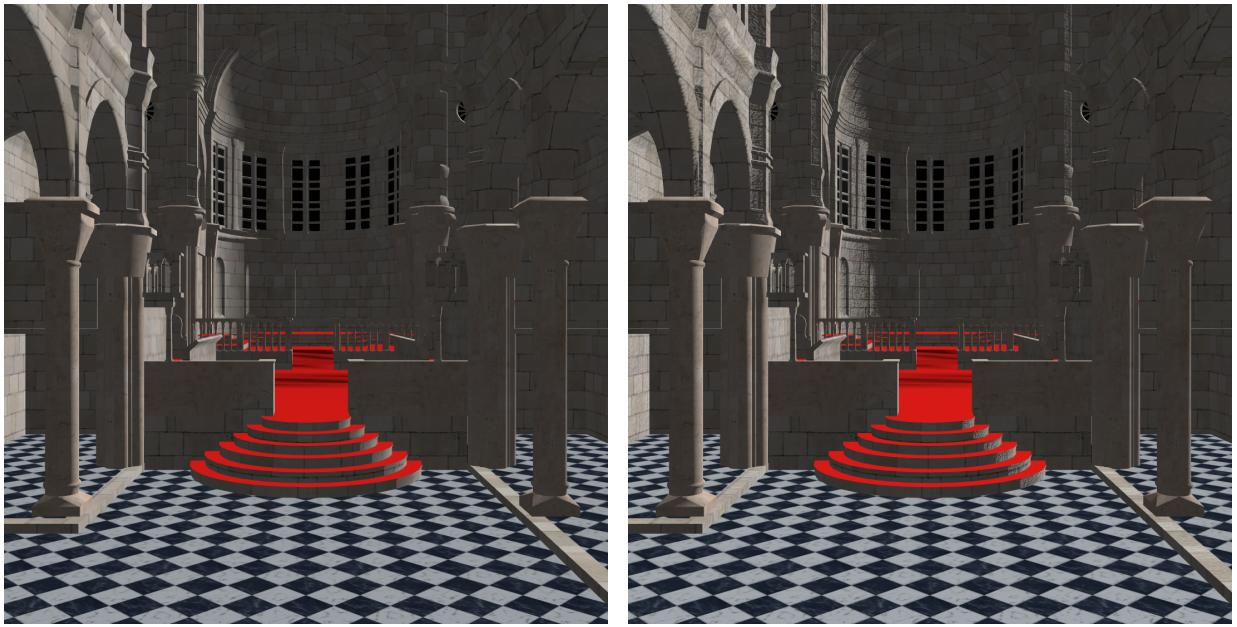
2. 在实际中, 当 alpha 很接近 0. 但不为 0. 时, 这个片段看上去就很透明了, 我们将其视为 0, 把 0.2 设为 threshold 是一个不错的选择。

1.3 Task 1-Bonus

思路: 设置 offset, 然后分别采样对应纹理坐标 left,right,down,up 四个点的高度值, 计算梯度, 其在 x 方向和 y 方向上的偏导, 然后将两个切线做叉积, 最后 normalize 作为 bumpnormal

```
1 vec3 GetNormal() {  
2     // Bump mapping from paper: Bump Mapping Unparametrized Surfaces on the GPU  
3     vec3 vn = normalize(v_Normal);  
4  
5     // your code here:  
6     vec3 bumpNormal = vn;  
7     float offset_ = 1.0/512.0;  
8     float heightL = texture(u_HeightMap, v_TexCoord - vec2(offset_, 0.0)).r;  
9     float heightR = texture(u_HeightMap, v_TexCoord + vec2(offset_, 0.0)).r;  
10    float heightD = texture(u_HeightMap, v_TexCoord - vec2(0.0, offset_)).r;  
11    float heightU = texture(u_HeightMap, v_TexCoord + vec2(0.0, offset_)).r;  
12  
13    float height_dx = (heightR - heightL);  
14    float height_dy = (heightU - heightD);  
15  
16    bumpNormal=normalize(vec3(-height_dx,-height_dy,2.0*offset_));  
17  
18    return bumpNormal != bumpNormal ? vn : normalize(vn * (1. - u_BumpMappingBlend) +  
19        bumpNormal * u_BumpMappingBlend);  
}
```

得到结果图:



(a) sibenik_bump=0

(b) sibenik_bump=10

图 2: Task 1: Phong Illumination-Bonus-sibenik



(a) Sponza_bump=0

(b) Sponza_bump=20

图 3: Task 1: Phong Illumination-Bonus-Sponza

2 Task 2: Environment Mapping

2.1 skybox

思路：通过取 4×4 的 view_matrix 矩阵的左上角 3×3 矩阵来去除掉变换矩阵的 transition 部分，这样就可以让移动不会影响到 skybox 的位置向量，实现了观察者移动但环境不会位移的效果，然后利用透视矩阵计算出 gl_position，在计算 position 时，我们将 xyz 中的 z 分量始终等于 w 分量，使得最终的设备坐标的 z 值永远为 1.0，使得 skybox 只在没有可见物体的地方进行渲染，通过深度测试。

```
1 void main() {
```

```

2     v_TexCoord = a_Position;
3
4 // your code here
5
6     mat4 View = mat4(mat3(u_View));
7     vec4 tmp_pos = u_Projection*View*vec4(a_Position, 1.0);
8     gl_Position = tmp_pos.xyww;
9 }
```

2.2 EnvMapping

思路：利用 reflect 函数计算出反射向量 (-viewDir 是由于 viewDir 的方向是由物体表面指向相机位置)，然后通过反射向量采样出环境纹理，再仿照 diffuseColor 的处理应用 gamma 校正到 envFactor，最终将环境映射得到的颜色添加到总光照中。

```

1 void main() {
2     float gamma          = 2.2;
3     vec4 diffuseFactor   = texture(u_DiffuseMap, v_TexCoord).rgba;
4     vec3 diffuseColor    = pow(diffuseFactor.rgb, vec3(gamma));
5     vec3 normal          = normalize(v_Normal);
6     vec3 viewDir         = normalize(u_ViewPosition - v_Position);
7 // Ambient component.
8     vec3 total = u_AmbientIntensity * u_AmbientScale * diffuseColor;
9 // Environment component
10
11 // your code here
12     vec3 reflectDir = reflect(-viewDir, normal);
13     vec3 envFactor = texture(u_EnvironmentMap, reflectDir).rgb;
14     vec3 envColor = pow(envFactor, vec3(gamma));
15     total+=u_EnvironmentScale*envColor;
16
17 // Iterate lights.
18     for (int i = 0; i < u_CntPointLights; i++) {
19         vec3 lightDir      = normalize(u_Lights[i].Position - v_Position);
20         float dist         = length(u_Lights[i].Position - v_Position);
21         float attenuation  = 1. / (dist * dist);
22         total             += Shade(u_Lights[i].Intensity, lightDir, normal, viewDir,
23         diffuseColor) * attenuation;
24     }
25     for (int i = u_CntPointLights + u_CntSpotLights; i < u_CntPointLights + u_CntSpotLights +
26     u_CntDirectionalLights; i++) {
27         total += Shade(u_Lights[i].Intensity, u_Lights[i].Direction, normal, viewDir,
28         diffuseColor);
29     }
// Gamma correction.
f_Color = vec4(pow(total, vec3(1. / gamma)), 1.);
```

得到结果图:



(a) teapot

(b) bunny

图 4: Task 2: Environment Mapping

3 Task 3: Non-Photorealistic Rendering

3.1 Gooch Shading

思路: 根据 Gooch 着色, 我们得到物体颜色的插值公式

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

其中 \mathbf{l} 为指向光源的方向, \mathbf{n} 为表面法线的方向, k_{cool}, k_{warm} 分别为冷色和暖色的颜色, 然后我们设定两个 threshold, 对 $\mathbf{l} \cdot \mathbf{n}$ 进行分段, 最后实现阶梯状的分段颜色

```
1 vec3 Shade ( vec3 lightDir , vec3 normal ) {
2     // your code here:
3     vec3 normal_lightDir = normalize( lightDir );
4     vec3 normal_normalDir = normalize( normal );
5     float cos_ = dot( normal_lightDir , normal_normalDir );
6     vec3 mix_color = vec3(0);
7
8     float threshold_1=0.25;
9     float threshold_2=0.7;
10    if(cos_>threshold_2)
11    {
12        mix_color=u_WarmColor;
13    }
14    else if(cos_>threshold_1)
15    {
16        mix_color=u_WarmColor*(1+cos_) / 2.0+u_CoolColor*(1-cos_) / 2.0;
17    }
18    else
```

```

19     {
20         mix_color=u_CoolColor;
21     }
22     return mix_color;
23 }
```

得到结果图:

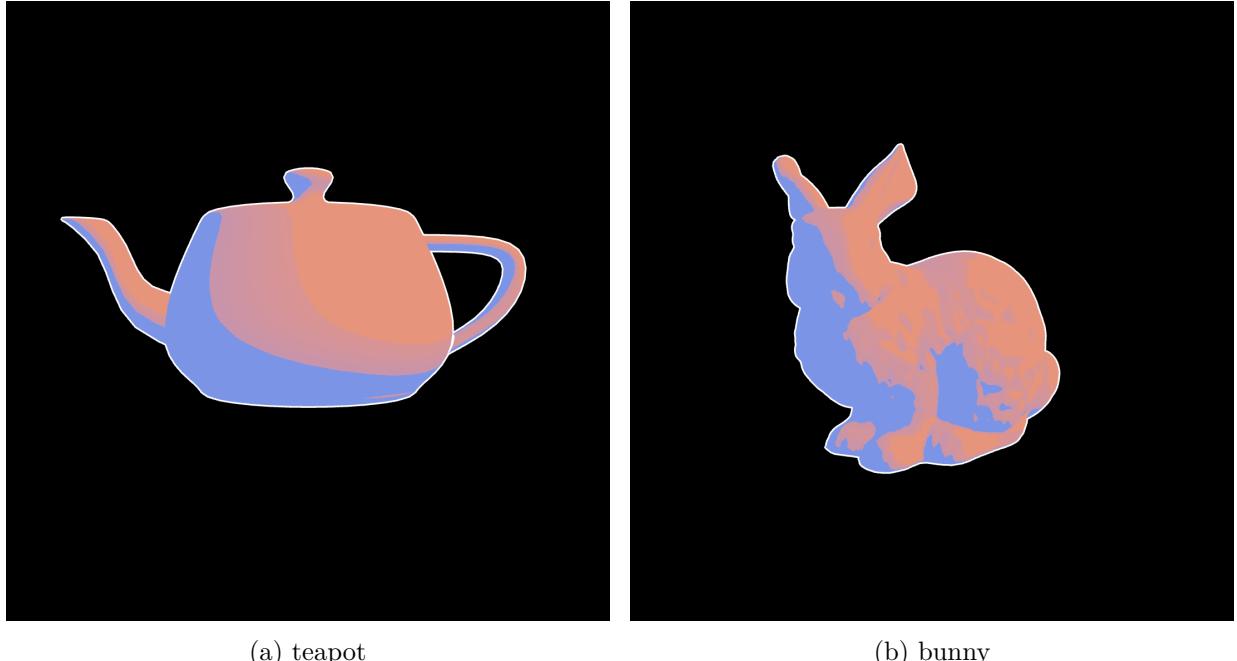


图 5: Task 3: Non-Photorealistic Rendering

3.2 Task 3-Questions

3.2.1 OnRender

OnRender 函数先渲染模型的反面然后渲染模型的正面，

渲染反面时先使用 OpenGL 的内置功能 `glCullFace(GL_FRONT)` 来设置将要剔除正面, 然后 `glEnable(GL_CULL_FACE)` 来启用 `GL_CULL_FACE`, 这样使得先前设置的 `GL_FRONT` 面被剔除, 不会被渲染, 然后遍历循环使用 `backLineProgram.Use()`, 来渲染未被剔除的面, 即渲染模型的背面。

渲染反面时, 同样的先将剔除面设为反面 `glCullFace(GL_BACK)` 然后启用 `GL_CULL_FACE`, 启用 `Depth_Test` `glEnable(GL_DEPTH_TEST)` 使得在渲染正面时, 先前渲染的背面能够被覆盖, 从而实现正面像素的正确渲染, 然后使用 `material.Albedo.Use()` 于 `material.MetaSpec.Use()` 来实现正面的渲染, 最后关闭深度测试和面剔除。

3.2.2 npr-line.vert

如果直接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, 由于透视变换会导致近处顶点的偏移量较大, 轮廓线的宽度可能会随观察者镜头的移动而改变, 造成轮廓线在视觉上不一致。且如果直接调整顶点的位置, 可能会导致经过透视计算后无法得到正确的空间位置, 导致物体的轮廓线出现奇怪的扭曲。

而我们如果在 clip 空间里做 offset 调整，可以确保轮廓线在屏幕上宽度始终保持一致，且得到正确的轮廓线空间位置。

```
1 void main() {  
2     vec4 clipPos = u_Projection * u_View * vec4(a_Position, 1.);  
3     vec3 clipNorm = mat3(u_Projection) * mat3(u_View) * a_Normal;  
4     vec2 offset = normalize(clipNorm.xy) / vec2(u_ScreenWidth, u_ScreenHeight) * u_LineWidth *  
5     clipPos.w * 2;  
6     clipPos.xy += offset;  
7     gl_Position = clipPos;  
}
```

4 Task 4: Shadow Mapping

4.1 phong-shadow

思路：先计算物体在光源空间的坐标，直接从 shadowMap 有向光源的阴影贴图中采样最接近纹理坐标的深度 closestDepth, curDepth 记录了当前物体的深度值，设 bias 为一个较小的偏移量，我们利用 shadow bias 的技巧避免出现 Shadow Acne 的问题，然后利用 bias 计算 curDepth-bias，若计算值大于 closestDepth 那么可以得知当前物体处于 shadow 中，将 shadow 设为 1.0。最后要判断当前物体是否超出了深度贴图的坐标范围，如果超出，那么认为其不在 shadow 中。

```
1 float Shadow(vec4 lightSpacePosition, vec3 normal, vec3 lightDir) {  
2     // return 1. if point in shadow, else return 0.  
3     vec3 pos = lightSpacePosition.xyz / lightSpacePosition.w;  
4     pos = pos * 0.5 + 0.5;  
5  
6     // your code here: closestDepth = ?  
7     float closestDepth = texture(u_ShadowMap, pos.xy).r;  
8     // your code end  
9  
10    float curDepth = pos.z;  
11    float bias = max(1e-3 * (1.0 - dot(normal, lightDir)), 1e-4);  
12    float shadow = (curDepth - bias > closestDepth ? 1.0 : 0.0);  
13    if (pos.z > 1.0 || pos.x < 0. || pos.x > 1. || pos.y < 0. || pos.y > 1.) shadow = 0.0;  
14    return shadow;  
15}
```

4.2 phong-shadowcubemap

思路：toLight 是物体到光源的向量，用它作为 shadowcubemap 的采样坐标，然后从 shadowCubeMap 点光源的阴影贴图中采样得到最近的深度 closestDepth, curDepth 记录了当前片元到光源的距离，设 bias 为一个较小的偏移量，我们利用 shadow bias 的技巧避免出现 Shadow Acne 的问题，当 curDepth-bias 大于 closestDepth 时，可以得到目标物体位于 shadow 中，将 shadow 设为 1.0

```

1 float Shadow( vec3 pos , vec3 lightPos ) {
2     // return 1. if point in shadow, else return 0.
3     vec3 toLight = pos - lightPos ;
4
5     // your code here: closestDepth = ?
6     float closestDepth = texture(u_ShadowCubeMap,toLight).r ;
7     // your code end
8
9     float curDepth = length(toLight) ;
10    float bias = 5. ;
11    float shadow = curDepth - bias > closestDepth ? 1.0 : 0.0 ;
12    return shadow ;
13 }
```

得到结果图:



(a) White Oak



(b) Sports Car

图 6: Task 4: Shadow Mapping

4.3 Task 4-Questions

4.3.1 Shadow Map

有向光源是从无限远处入射的平行光，可以直接使用正交投影矩阵计算深度贴图。

点光源是从一个点光源出发向所有方向发射光，需要使用透视投影矩阵来计算深度贴图。由于点光源需要覆盖 360° 的范围，可以利用 cubemap 对立方体贴图的六个面依次用透视投影矩阵计算深度贴图。

4.3.2 True Depth?

因为我们得到 shadow map 是通过在光源的透视图下来加上一个投影矩阵来渲染场景，最后把深度值的结果存储到纹理中。那么对于待渲染的一个点，我们将其变换到光源空间的坐标中，即 phong-shadow.vert 中” $v_LightSpacePosition = u_lightSpaceMatrix * vec4(v_Position, 1.);$ ”，在计算片元 pos 的时候，“ $vec3 pos =$

`lightSpacePosition.xyz/lightSpacePosition.w; pos = pos * 0.5 + 0.5;`,”都进行了归一化操作，深度值范围为0~1，那么这个点在光源空间中的z坐标就相当于一个深度，能够与shadow map中的深度值进行比较，最后判断这个片段是否处于阴影之中。

5 Task 5: Whitted-Style Ray Tracing

5.1 IntersectTriangle - Möller-Trumbore

思路：使用Möller-Trumbore算法，首先计算出两条边向量edge12与edge13，然后取edge12与光线方向做叉乘，再将叉乘结果与edge12做点积，若点积结果小于threshold，那么可以认为该射线与三角形平面几乎平行，没有交点。

接着利用射线原点到p1的向量tvec来确定u参数，根据u范围判断交点是否在三角形内部。

然后用tvec和edge12的叉乘计算qvec，用qvec和射线方向的点积来确定v参数，根据v和u的范围来判断交点是否在三角形内部。

最后利用qvec和edge13的点积来确定t参数。

```
1 bool IntersectTriangle(Intersection & output, Ray const & ray, glm::vec3 const & p1, glm::vec3
2   const & p2, glm::vec3 const & p3) {
3   // your code here
4
5   glm::vec3 edge12 = p2 - p1;
6   glm::vec3 edge13 = p3 - p1;
7
8   glm::vec3 normal_dir = normalize(ray.Direction);
9   glm::vec3 origin_point = ray.Origin;
10
11  glm::vec3 pvec = glm::cross(normal_dir, edge13);
12  double det = glm::dot(edge12, pvec);
13
14  if (abs(det) < 0.00005)
15  {
16    return false;
17  }
18  double inv_det = 1.0 / det;
19
20  glm::vec3 tvec = origin_point - p1;
21  double u_ = glm::dot(tvec, pvec)*inv_det;
22  if (u_<0.0 || u_>1.0)
23  {
24    return false;
25  }
26
27  glm::vec3 qvec = glm::cross(tvec, edge12);
28  double v_ = glm::dot(normal_dir, qvec)*inv_det;
29  if (v_<0.0 || u_ + v_>1.0)
30  {
31    return false;
32  }
```

```

33     double t_ = glm::dot(edge13, qvec)*inv_det;
34
35     output.t = t_;
36     output.u = u_;
37     output.v = v_;
38     return true;
39 }
```

5.2 RayTrace - Whitted-style RayTrace

思路：首先我们要判断什么时候 Light 发生了遮挡情况，然后对于未遮挡的情况再利用传入的参数对颜色进行计算

若 Light 的类型为点光源，那么我们先利用 intersector.IntersectRay 对光线求交，如果存在交点，且遮挡物的不透明度 $>=0.2$ ，那么我们初步判断发生了遮挡情况，接下来我们还需要考虑到 shadow ray 和遮挡物的交点可能在点光源的背后，但是这种情况应该被认为是没有发生遮挡，我们加一个特判：利用遮挡物交点到光源的向量与着色点到光源的向量做点积，若结果为正，说明交点在点光源和着色点之间，着色点位于阴影中，发生了遮挡情况，直接 continue

若 Light 的类型为有向光源，那么我们先利用 intersector.IntersectRay 对光线求交，如果存在交点，且遮挡物的不透明度 $>=0.2$ ，那么我们可以直接判断发生了遮挡情况，直接 continue

如果没有发生遮挡情况，我们利用 Blinn-Phong 模型来计算出着色点的颜色即可

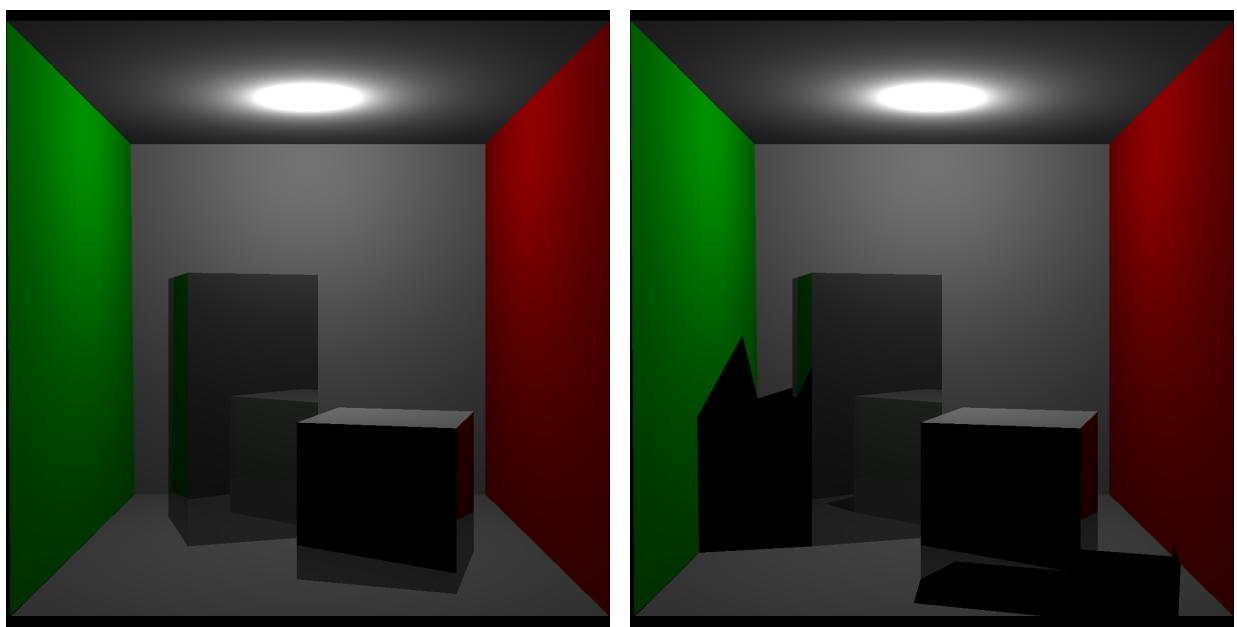
```

1 //***** 2. Whitted-style ray tracing *****/
2 // your code here
3
4 for (const Engine::Light & light : intersector.InternalScene->Lights) {
5     glm::vec3 l;
6     float attenuation;
7 //***** 3. Shadow ray *****/
8     if (light.Type == Engine::LightType::Point) {
9         l = light.Position - pos;
10        attenuation = 1.0f / glm::dot(l, l);
11        if (enableShadow) {
12            auto shadowHit_ = intersector.IntersectRay(Ray(pos, l));
13            if (shadowHit_.IntersectState==true && shadowHit_.IntersectAlbedo.w >= 0.2f) {
14
15                // consider light&scene ray cross point in the back
16                glm::vec3 shadow_pos = shadowHit_.IntersectPosition;
17                glm::vec3 shadow_light_dir = light.Position-shadow_pos;
18                if (glm::dot(shadow_light_dir, l)>0) continue;
19            }
20        }
21    } else if (light.Type == Engine::LightType::Directional) {
22        l = light.Direction;
23        attenuation = 1.0f;
24        if (enableShadow) {
25            auto shadowHit_ = intersector.IntersectRay(Ray(pos, l));
26            if (shadowHit_.IntersectState==true && shadowHit_.IntersectAlbedo.w >= 0.2f) {
27                continue;
28            }
29        }
30    }
31 }
```

```

28     }
29 }
30
31 ***** 2. Whitted-style ray tracing *****
32 // your code here
33
34
35     glm::vec3 normalize_normal = normalize(n);
36     glm::vec3 normalize_lightdir = normalize(l);
37
38 // diffuse_light
39     float diff_cos = glm::max(glm::dot(normalize_normal, normalize_lightdir), 0.0f);
40     glm::vec3 diffuse_light = kd * diff_cos * light.Intensity * attenuation;
41
42 // specular_light
43 // use_blinn-phong
44     glm::vec3 normalize_raydir = normalize(ray.Direction);
45     glm::vec3 normalize_halfdir = normalize(normalize_raydir + normalize_lightdir);
46     float spec_cos = pow(glm::max(0.0f, glm::dot(normalize_raydir, normalize_normal)), shininess);
47
48     glm::vec3 specular_light = ks * spec_cos * light.Intensity * attenuation;
49
50     result += (diffuse_light + specular_light);
51 }
```

得到结果图:



(a) Cornell Box-Noshadow

(b) Cornell Box-Shadow

图 7: Task 5: Whitted-Style Ray Tracing

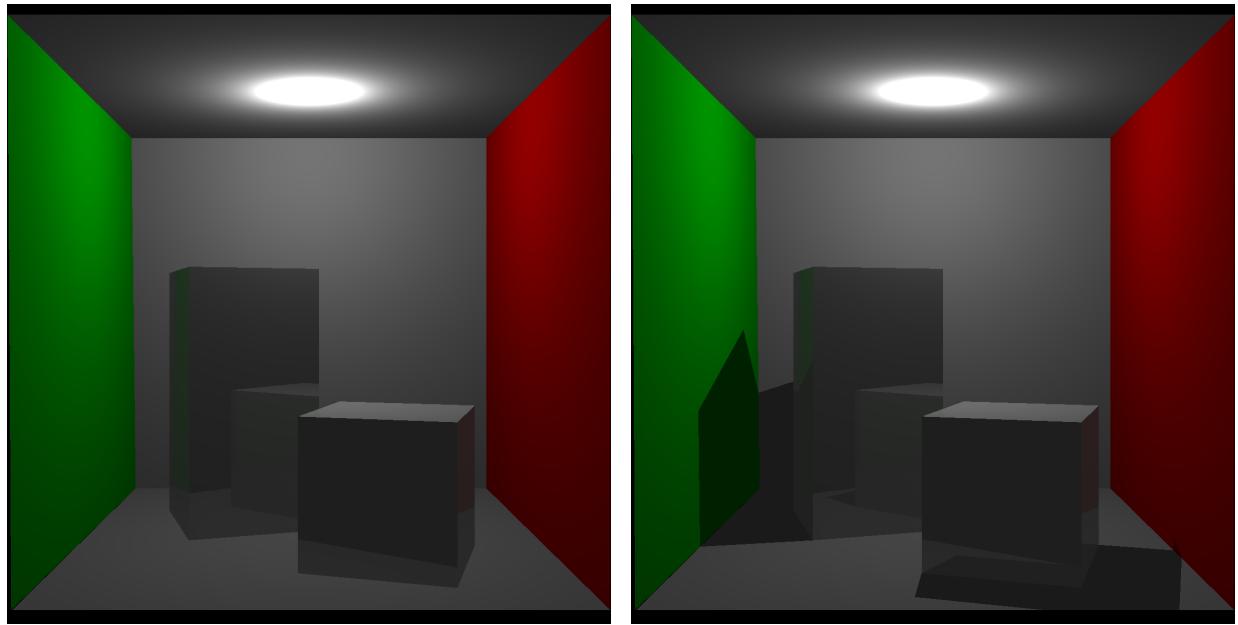
发现跟给出的 ground truth 有一些区别，阴影部分被处理为全黑，这是因为没有考虑到环境光，我们再给 result 加上环境光

```

1 // add ambient light
2 result += kd * intersector.InternalScene->AmbientIntensity;

```

再次得到结果图：



(a) Cornell Box-Noshadow-add ambient light

(b) Cornell Box-Shadow-add ambient light

图 8: Task 5: Whitted-Style Ray Tracing

5.3 Task 5-Questions

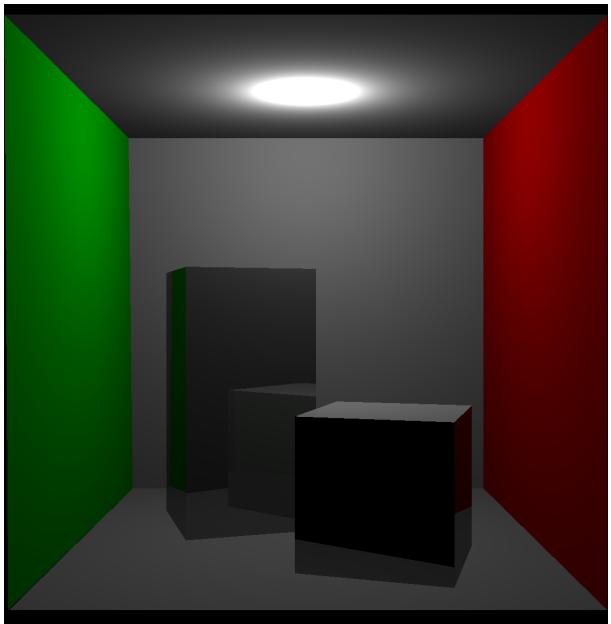
5.4 Ray Tracing vs Rasterization

光线追踪的渲染结果比光栅化的渲染结果增加了更多的细节，如对于 Cornell Box 这个场景，光线追踪的结果中长方体上有立方体的“影子”，涵盖了全局光照中的反射、折射等，更加具有真实感，但光线追踪的渲染时间要远高于光栅化的渲染时间。

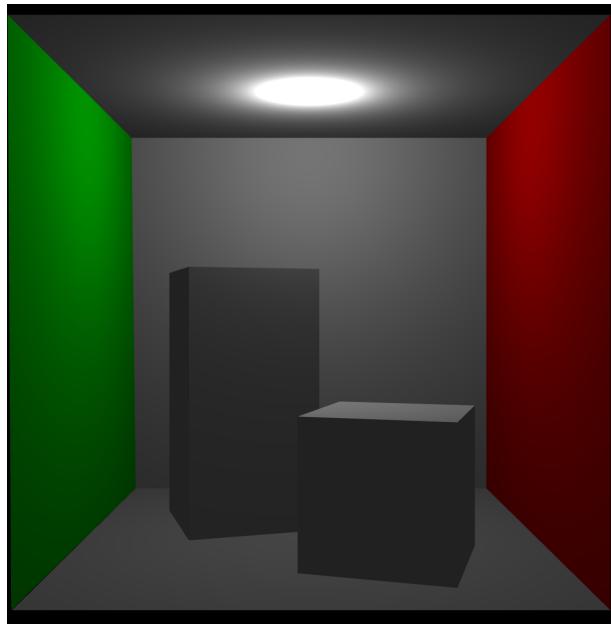
共同点在于两者在颜色计算上呈现出来的效果大致相似。

我的理解：

1. 光线追踪模拟光的行为来渲染图像，在全局上考虑了光的反射和折射，而光栅化则只考虑了局部的光照，利用 Blinn-Phong 模型对局部着色进行计算，未考虑光的全局传播。
2. 光线追踪需要处理全局光照效果，需要进行更为复杂的运算迭代，因此时间开销较大，而光栅化只处理局部，然后将 3D 模型转为 2D 图像，计算成本较低，渲染时间快。



(a) Cornell Box-RayTrace



(b) Cornell Box-Rasterization

图 9: Task 5: Whitted-Style Ray Tracing-Q1