

Lab 2 报告

蒋锦豪 2400011785

2025 年 11 月 14 日

Task 1: Phong Illumination

原理与代码实现

Phong 光照模型的计算公式为

$$L = k_a I_a + k_d \left(I_d + \frac{I_p}{r^2} \right) \max(\mathbf{l} \cdot \mathbf{n}, 0) + k_s \left(I_d + \frac{I_p}{r^2} \right) \max(\mathbf{v} \cdot \mathbf{r}, 0)^\alpha$$

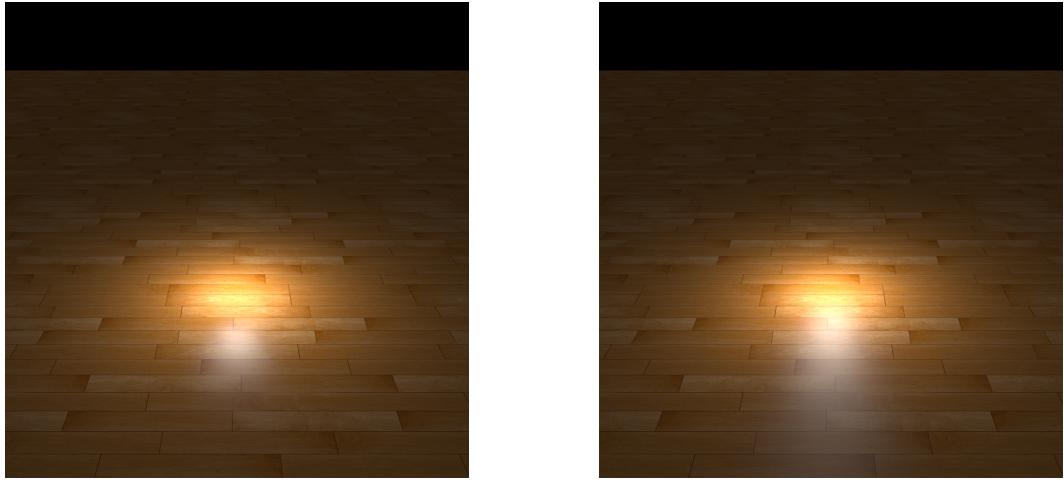
Blinn-Phong 光照模型的计算公式为

$$L = k_a I_a + k_d \left(I_d + \frac{I_p}{r^2} \right) \max(\mathbf{l} \cdot \mathbf{n}, 0) + k_s \left(I_d + \frac{I_p}{r^2} \right) \max(\mathbf{n} \cdot \mathbf{h}, 0)^\alpha$$

在着色器中实现的代码如下 (这里已经将光强统一记为 `lightIntensity`, 合并了距离衰减等因素):

```
1 vec3 Shade(vec3 lightIntensity, vec3 lightDir, vec3 normal, vec3 viewDir, vec3 diffuseColor, vec3
2   specularColor, float shininess) {
3   // your code here:
4   vec3 l = normalize(lightDir);
5   vec3 n = normalize(normal);
6   vec3 v = normalize(viewDir);
7   vec3 h = normalize(l + v);
8   if (!u_UseBlinn) {
9     return lightIntensity * (diffuseColor * max(dot(n, l), 0.0) +
10    specularColor * pow(max(dot(v, reflect(-l, n)), 0.0), shininess));
11 } else{
12   return lightIntensity * (diffuseColor * max(dot(n, l), 0.0) +
13   specularColor * pow(max(dot(h, n), 0.0), shininess));
14 }
```

实现效果



(a) Phong 光照模型的渲染效果

(b) Blinn-Phong 光照模型的渲染效果

图 1: 对 `floor` 的渲染结果

问题回答

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?

答: 顶点着色器负责处理顶点数据, 进行坐标变换等操作; 片段着色器负责计算每个像素的最终颜色; 在 GLSL 中, 变量传递依靠着色器的 `in` 和 `out` 关键字. 顶点着色器中使用 `out` 声明的变量会传递给片段着色器中使用 `in` 声明的同名变量.

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?

答: 起到透明度测试的作用, 忽略较为透明的片段. 如果用后面的代码替代, 那么只有完全透明的像素会被丢弃, 而很多比较透明的像素不会被丢弃, 效率降低, 同时可能造成较大的精度误差.

Task 1 Bonus: Bump Mapping

原理与代码实现

考虑模型的法线 \mathbf{n} . 希望通过凹凸贴图对法向量施加一个扰动 $\Delta\mathbf{n}$, 即

$$\mathbf{n}' = \mathbf{n} + \Delta\mathbf{n}$$

现在的目标是求出法线 \mathbf{n} . 对于模型上任意一点, 将其位置 \mathbf{p} 写作参数方程 $\mathbf{p} = \mathbf{p}(s, t)$. 此处的两个切向量分别为

$$\mathbf{t} = \frac{\partial \mathbf{p}}{\partial s}, \quad \mathbf{b} = \frac{\partial \mathbf{p}}{\partial t}$$

法线应当与所有切向量垂直, 于是

$$\mathbf{n} = \frac{\mathbf{t} \times \mathbf{b}}{\|\mathbf{t} \times \mathbf{b}\|}$$

设每个点的高度由纹理函数 $h(s, t)$ 给出, 那么该点处真实的高度即为模型上的点沿模型法向量 \mathbf{n} 方向移动 $h(s, t)$ 长度, 于是新的位置为

$$\mathbf{p}'(s, t) = \mathbf{p}(s, t) + h(s, t)\mathbf{n}$$

对 \mathbf{p}' 求偏导数, 可得新的切线 \mathbf{t}' 为

$$\mathbf{t}' = \frac{\partial \mathbf{p}'}{\partial s} = \frac{\partial \mathbf{p}}{\partial s} + \frac{\partial h}{\partial s}\mathbf{n} + \frac{\partial \mathbf{n}}{\partial s}h = \mathbf{t} + \frac{\partial h}{\partial s}\mathbf{n} + \frac{\partial \mathbf{n}}{\partial s}h$$

类似地可得新的副切线 \mathbf{b}' 为

$$\mathbf{b}' = \mathbf{b} + \frac{\partial h}{\partial t}\mathbf{n} + \frac{\partial \mathbf{n}}{\partial t}h$$

一般而言, 法向量 \mathbf{n} 的变化是比较小的, 因此可以忽略上式中带有 $\frac{\partial \mathbf{n}}{\partial s, t}$ 的项. 于是新的法向量为

$$\begin{aligned} \mathbf{n}' &= \mathbf{t}' \times \mathbf{b}' = \left(\mathbf{t} + \frac{\partial h}{\partial s}\mathbf{n}\right) \times \left(\mathbf{b} + \frac{\partial h}{\partial t}\mathbf{n}\right) \\ &= \mathbf{t} \times \mathbf{b} + \frac{\partial h}{\partial s}\mathbf{n} \times \mathbf{b} + \frac{\partial h}{\partial t}\mathbf{t} \times \mathbf{n} \end{aligned}$$

最后归一化即可得到新的法向量:

$$\mathbf{n}' = \text{normalize}\left(\mathbf{t} \times \mathbf{b} - \frac{\partial h}{\partial s}\mathbf{b} \times \mathbf{n} - \frac{\partial h}{\partial t}\mathbf{n} \times \mathbf{t}\right) = \mathbf{n} + \frac{\frac{\partial h}{\partial s}\mathbf{n} \times \mathbf{b} + \frac{\partial h}{\partial t}\mathbf{t} \times \mathbf{n}}{\mathbf{n} \cdot (\mathbf{t} \times \mathbf{b})}$$

既然参数化方法 $\mathbf{p}(s, t)$ 是任意选取的, 那么可以令 s, t 分别为屏幕空间的 x, y 坐标. 这样, 可以在屏幕空间中通过微分或差分的方式直接计算上述式子中需要的参量 $\mathbf{t}, \mathbf{b}, \frac{\partial h}{\partial s}, \frac{\partial h}{\partial t}$. 这就是基于表面梯度的凹凸贴图的实现原理, 其代码实现如下:

```

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 pDx = dFdx(v_Position);
7     vec3 pDy = dFdy(v_Position);
8     vec3 r1 = cross(pDy, vn);
9     vec3 r2 = cross(vn, pDx);
10    float d = dot(pDx, r1);
11    float h = texture(u_HeightMap, v_TexCoord).r;
12    float h1 = texture(u_HeightMap, v_TexCoord + dFdx(v_TexCoord)).r;
13    float h2 = texture(u_HeightMap, v_TexCoord + dFdy(v_TexCoord)).r;
14    vec3 bumpNormal = normalize(abs(d) * vn - (sign(d) * ((h1 - h) * r1 + (h2 - h) * r2)));

```

```

15
16     return bumpNormal != bumpNormal ? vn : normalize(vn * (1. - u_BumpMappingBlend) + bumpNormal *
17     u_BumpMappingBlend);
}

```

其中 pDx 和 pDy 分别对应 t 和 b , $r1$ 和 $r2$ 分别对应 $b \times n$ 和 $n \times t$, $h1 - h$ 和 $h2 - h$ 分别对应 $\frac{\partial h}{\partial s}$ 和 $\frac{\partial h}{\partial t}$ 通过有限差分求得的近似值.

实现效果

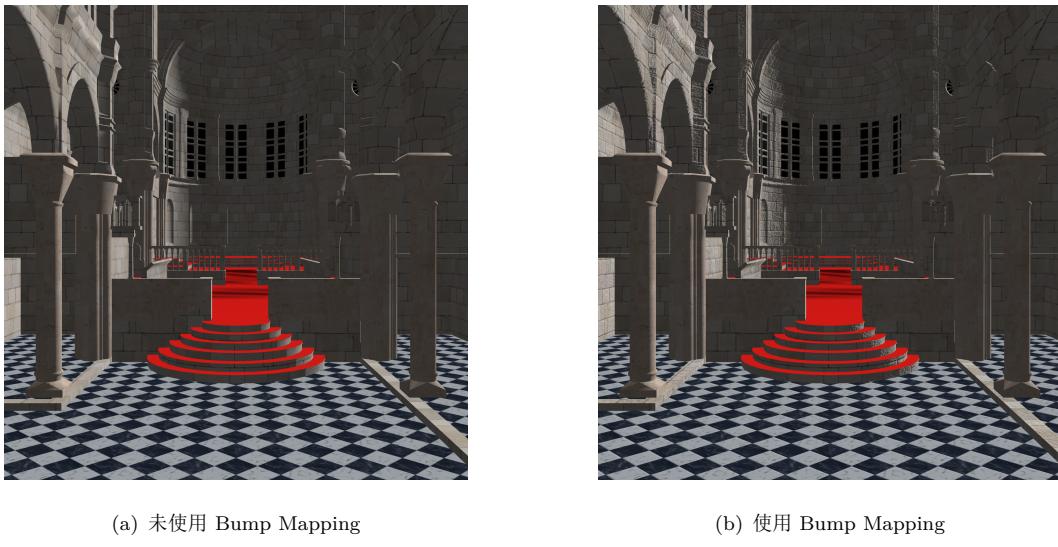


图 2: 对 Sibenik 使用 Bump Mapping 与否渲染结果对比

Task 2: Environment Mapping

原理与代码实现

环境映射的基本原理是利用一个预先生成的立方体贴图 (一般可称之为天空盒) 来表示环境, 相机移动时天空盒不动, 但相机旋转时天空盒随之旋转. 此外, 天空盒总是位于无限远处, 不会产生透视缩放. 其代码实现的过程和原理如下:

首先, 直接把立方体的方向向量 `a_Position` 传给 `v_TexCoord`, 用作立方体贴图的方向向量:

```
1 v_TexCoord = a_Position;
```

为了保留相机的旋转而消除平移, 将视图矩阵 `u_View` 的平移部分 (即最后一列) 置零:

```
1 mat4 view = mat4(mat3(u_View));
```

然后经过投影变换:

```
1 vec4 pos = u_Projection * view * vec4(a_Position, 1.0);
```

最后, 为了强制使得天空盒位于最远处, 需要在透视线除法后使得 z 坐标为 1. 这可以通过令顶点的 z 坐标等于 w 坐标来实现:

```
1 gl_Position = pos.xyww;
```

整体的代码如下:

```
1 void main() {
2     v_TexCoord = a_Position;
3     // your code here
4     mat4 view = mat4(mat3(u_View));
5     vec4 pos = u_Projection * view * vec4(a_Position, 1.0);
6     gl_Position = pos.xyww;
7 }
```

然后是片段着色器的实现. 其余部分已经给出, 只需要补充天空盒的着色部分即可. 考虑到环境贴图的采样方向为视线方向关于法线的反射方向, 因此用 `reflect` 函数来计算反射方向并在 `u_EnvironmentMap` 中采样, 然后乘以环境光系数 `u_EnvironmentScale` 后累加到最终颜色 `total`:

```
1 void main() {
2     ...
3     // your code here
4     total += texture(u_EnvironmentMap, reflect(viewDir, normal)).xyz * u_EnvironmentScale;
5     ...
6 }
```

实现效果



(a) 对 Teapot 场景的环境映射效果

(b) 对 Bunny 场景的环境映射效果

图 3: 环境映射的渲染效果

Task 3: Non-Photorealistic Rendering

原理与代码实现

按照 Gooch 着色模型, 取定冷色 k_c 和暖色 k_w , 物体上各点的颜色通过插值得到:

$$k = \left(\frac{1 + \mathbf{l} \cdot \mathbf{n}}{2} \right) k_c + \left(\frac{1 - \mathbf{l} \cdot \mathbf{n}}{2} \right) k_w$$

其中 \mathbf{l} 是物体上的点指向光源的方向, \mathbf{n} 是该点表面法线的方向. 为了实现分段的效果, 需要对系数进行离散化处理, 代码如下:

```

1 vec3 Shade (vec3 lightDir, vec3 normal) {
2     // your code here:
3     lightDir = normalize(lightDir);
4     normal = normalize(normal);
5     float t = dot(-lightDir, normal);
6     float t_Cool = (1.0 + t) / 2.0;
7     if (t_Cool > 0.6) t_Cool = 1.0;
8     else if (t_Cool < 0.25) t_Cool = 0.0;
9     else t_Cool = 0.5;
10    float t_Warm = 1 - t_Cool;
11    return t_Cool * u_CoolColor + t_Warm * u_WarmColor;
12 }
13

```

```

14 void main() {
15     // your code here:
16     float gamma = 2.2;
17     vec3 total = Shade(u_Lights[0].Direction, v_Normal);
18     f_Color = vec4(pow(total, vec3(1. / gamma)), 1.);
19 }
```

实现效果



图 4: Gooch 着色模型的渲染效果

问题回答

- 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?

答: `OnRender` 函数进行了两次渲染; 第一次渲染时启用了面剔除去掉了模型的正面, 然后渲染模型的背面; 第二次渲染时剔除了模型的背面, 并启用深度测试, 按照正常的着色办法渲染模型的正面.

- `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?

答: 由于世界坐标到屏幕的映射是非线性的, 因此在世界坐标中移动相同距离, 可能在屏幕坐标上移动的距离就有差别, 这会导致绘制的轮廓线粗细不均.

Task 4: Shadow Mapping

原理与代码实现

只需在正常的渲染流程中增加阴影贴图的采样和比较即可。由于深度贴图的值是存储在 $[0, 1]$ 区间内的，因此需要在比较之前将其还原至真实值。代码如下 (`Shade` 函数采用 Blinn-Phong 模型，与前面的相同，这里不再给出。):

```

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 * u_FarPlane;
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 }
```

```

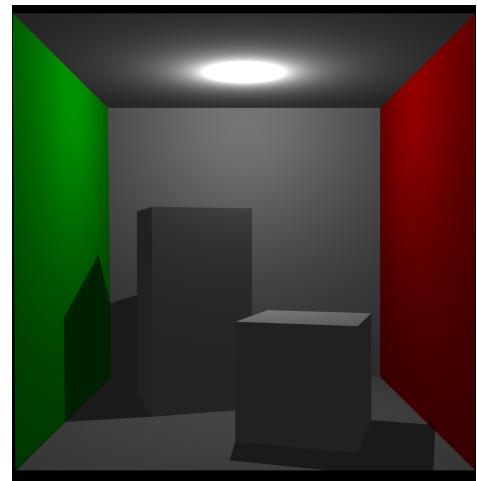
1 float Shadow(vec4 lightSpacePosition, vec3 normal, vec3 lightDir) {
2     // return 1. if point in shadow, else return 0.
3     vec3 pos = lightSpacePosition.xyz / lightSpacePosition.w;
4     pos = pos * 0.5 + 0.5;
5
6     // your code here: closestDepth = ?
7     float closestDepth = texture(u_ShadowMap, 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 }
```

注意到在普通阴影贴图中已经通过 `vec3 pos = lightSpacePosition.xyz / lightSpacePosition.w;`
`pos = pos * 0.5 + 0.5;` 语句对深度归一化，因此后续直接用深度贴图的值比较；在立方体的阴影贴图中，深度值没有经过归一化，因此深度贴图的值需要乘以远裁剪面距离 `u_FarPlane` 再与真实深度比较。

实现效果



(a) 对 `teapot` 场景进行阴影映射的效果



(b) 对 `box` 场景进行阴影映射的效果



(c) 对 `oak` 场景进行阴影映射的效果



(d) 对 `sponza` 场景进行阴影映射的效果

图 5: 阴影映射的效果

问题回答

- 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？

答：有向光源是平行光，因此应该使用正交投影矩阵计算深度贴图；点光源的光线汇聚于一点，因此应该使用透视投影矩阵计算深度贴图。

- 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？

答：通过将顶点坐标乘以光源视角投影矩阵，把每个顶点的坐标转换到以光源视角下的标准设备坐标，已经求得了深度值。

Task 5: Whitted-Style Ray Tracing

原理与代码实现

首先是光线与三角形求交的实现, 原理如下: 对于三角形上任意一点 $\mathbf{t}(u, v)$, 其满足

$$\mathbf{t}(u, v) = (1 - u - v)\mathbf{v}_0 + u\mathbf{v}_1 + v\mathbf{v}_2$$

这里的 u, v 即为点在三角形中的重心坐标, 满足 $u \geq 0, v \geq 0$ 且 $u + v \leq 1$. 将上述方程与光线方程联立, 可得

$$\mathbf{o} + t\mathbf{d} = (1 - u - v)\mathbf{v}_0 + u\mathbf{v}_1 + v\mathbf{v}_2$$

整理后得到

$$-t\mathbf{d} + u(\mathbf{v}_1 - \mathbf{v}_0) + v(\mathbf{v}_2 - \mathbf{v}_0) = \mathbf{o} - \mathbf{v}_0$$

这一线性方程组的矩阵形式为

$$\begin{bmatrix} -\mathbf{d} & \mathbf{v}_1 - \mathbf{v}_0 & \mathbf{v}_2 - \mathbf{v}_0 \end{bmatrix} \begin{bmatrix} t \\ u \\ v \end{bmatrix} = \mathbf{o} - \mathbf{v}_0$$

令

$$\mathbf{D} = \mathbf{d}, \quad \mathbf{E}_1 = \mathbf{v}_1 - \mathbf{v}_0, \quad \mathbf{E}_2 = \mathbf{v}_2 - \mathbf{v}_0, \quad \mathbf{T} = \mathbf{o} - \mathbf{v}_0$$

根据 Cramer 法则可知线性方程组的解为

$$\begin{bmatrix} t \\ u \\ v \end{bmatrix} = \frac{1}{\det \begin{bmatrix} -\mathbf{D} & \mathbf{E}_1 & \mathbf{E}_2 \end{bmatrix}} \begin{bmatrix} \det \begin{bmatrix} \mathbf{T} & \mathbf{E}_1 & \mathbf{E}_2 \end{bmatrix} \\ \det \begin{bmatrix} -\mathbf{D} & \mathbf{T} & \mathbf{E}_2 \end{bmatrix} \\ \det \begin{bmatrix} -\mathbf{D} & \mathbf{E}_1 & \mathbf{T} \end{bmatrix} \end{bmatrix}$$

即

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

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

$$t = \frac{\mathbf{Q} \cdot \mathbf{E}_2}{\mathbf{P} \cdot \mathbf{E}_1}, \quad u = \frac{\mathbf{P} \cdot \mathbf{T}}{\mathbf{P} \cdot \mathbf{E}_1}, \quad v = \frac{\mathbf{Q} \cdot \mathbf{D}}{\mathbf{P} \cdot \mathbf{E}_1}$$

最后判断 $u \geq 0, v \geq 0, u + v \leq 1$ 且 $t \geq 0$ 即可确定光线与三角面片是否相交, 如果相交则交点为 $\mathbf{r}(t) = \mathbf{o} + t\mathbf{d}$, 在三角面片上的重心坐标为 (u, v) .

上述过程的算法流程如下:

1. 按照定义计算辅助向量 $\mathbf{D}, \mathbf{E}_1, \mathbf{E}_2, \mathbf{T}, \mathbf{P}, \mathbf{Q}$.
2. 判断行列式 $\mathbf{P} \cdot \mathbf{E}_1$ 是否接近于 0, 如果是则光线与三角面片平行, 不相交.

3. 计算参数 t, u, v 的值.

4. 判断 $u \geq 0, v \geq 0, u + v \leq 1$ 且 $t \geq 0$ 是否成立, 如果成立则光线与三角面片相交, 否则不相交.

上述过程的代码实现如下:

```

1 bool IntersectTriangle(Intersection & output, Ray const & ray, glm::vec3 const & p1, glm::vec3 const
2   & p2, glm::vec3 const & p3) {
3   // your code here
4   glm::vec3 O = ray.Origin, D = ray.Direction;
5   glm::vec3 T = O - p1, E1 = p2 - p1, E2 = p3 - p1;
6   glm::vec3 P = glm::cross(D, E2), Q = glm::cross(T, E1);
7   float d = glm::dot(P, E1);
8   if (d < 0.000001f && d > -0.000001f) return false;
9   float u = glm::dot(T, P) / d, v = glm::dot(D, Q) / d, t = glm::dot(E2, Q) / d;
10  if (u < 0 || u > 1 || v < 0 || u + v > 1) return false;
11  else {
12    output.t = t;
13    output.u = u;
14    output.v = v;
15    return true;
16  }
}

```

然后是光线追踪的实现. 首先需要在遍历各光源之前对结果加上环境光的贡献, 然后在遍历光源时按照 Blin-Phong 模型计算漫反射和镜面反射的贡献. 代码实现如下:

```

1 result += kd * intersector.InternalScene->AmbientIntensity;
2 for (const Engine::Light & light : intersector.InternalScene->Lights) {
3   glm::vec3 l;
4   float attenuation;
5   ...
6   // **** 2. Whitted-style ray tracing ****
7   // your code here
8   if (inShadow) continue;
9   glm::vec3 V = glm::normalize(-ray.Direction);
10  glm::vec3 N = glm::normalize(n);
11  glm::vec3 L = glm::normalize(l);
12  glm::vec3 H = glm::normalize(L + V);
13  result += attenuation * light.Intensity * (kd * glm::max(glm::dot(N, L), 0.0f) + ks * glm::pow(
14    glm::max(glm::dot(H, N), 0.0f), shininess));
}

```

然后是 Shadow Ray 的实现, 在计算漫反射和镜面反射之前, 需要先发射一条 Shadow Ray 来判断该点是否被遮挡. 如果该点发出的 Shadow Ray 与场景中的物体相交, 并且该物体的透明度 $\alpha > 0.2$, 那么认为被遮挡; 否则就循环地以该物体为起点继续发射 Shadow Ray, 直到没有击中任何物体为止 (对于点光源, 只要距离大于光源距离即可终止). 代码实现如下:

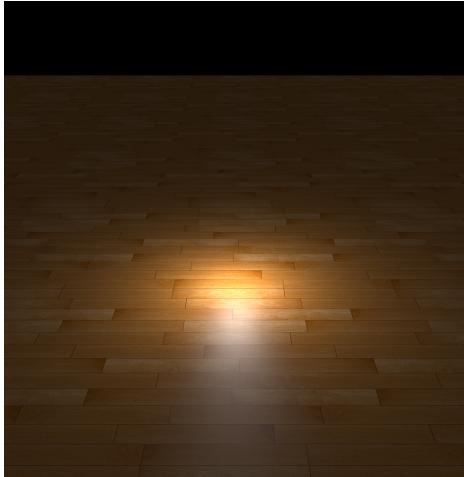
```

1  for (const Engine::Light & light : intersector.InternalScene->Lights) {
2      glm::vec3 l;
3      float    attenuation;
4      bool     inShadow = false;
5      /***** 3. Shadow ray *****/
6      if (light.Type == Engine::LightType::Point) {
7          l        = light.Position - pos;
8          float length = glm::dot(l, l);
9          attenuation = 1.0f / glm::dot(l, l);
10         if (enableShadow) {
11             // your code here
12             auto shadowRayHit = intersector.IntersectRay(Ray(pos, glm::normalize(l)));
13             while (shadowRayHit.IntersectState && shadowRayHit.IntersectAlbedo.w < 0.2)
14                 shadowRayHit = intersector.IntersectRay(Ray(shadowRayHit.IntersectPosition,
15                                                 glm::normalize(l)));
16             if (shadowRayHit.IntersectState) {
17                 glm::vec3 sh = shadowRayHit.IntersectPosition - pos;
18                 if (glm::dot(sh, sh) < length)
19                     attenuation = 0.0f;
20             }
21         }
22     } else if (light.Type == Engine::LightType::Directional) {
23         l        = light.Direction;
24         attenuation = 1.0f;
25         if (enableShadow) {
26             // your code here
27             auto shadowRayHit = intersector.IntersectRay(Ray(pos, glm::normalize(l)));
28             while (shadowRayHit.IntersectState && shadowRayHit.IntersectAlbedo.w < 0.2)
29                 shadowRayHit = intersector.IntersectRay(Ray(shadowRayHit.IntersectPosition,
30                                                 glm::normalize(l)));
31             if (shadowRayHit.IntersectState)
32                 attenuation = 0.0f;
33         }
34     }
35     /***** 2. Whitted-style ray tracing *****/

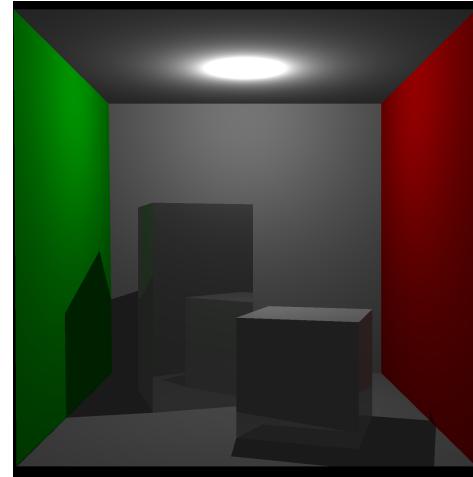
```

```
36     ...
37 }
```

实现效果



(a) 对 `floor` 场景进行光线追踪的效果



(b) 对 `box` 场景进行光线追踪的效果

图 6: 光线追踪的效果

问题回答

1. 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?

答: 光线追踪能够自然地模拟光的传播路径, 因此能够更真实地表现反射, 折射和阴影等效果; 而光栅化则是通过对场景进行投影和像素着色来生成图像, 在处理复杂光照效果时可能不够准确. 因此光线追踪的渲染结果通常更接近真实世界的视觉效果, 而光栅化则更适合实时渲染应用.

Task 5 Bonus: Spatial Acceleration

原理与代码实现

这里采用 BVH 树进行光线求交的空间加速. 首先定义了三个结构体 `Triangle`, `AABB` 和 `BVHNode`, 分别表示三角形, 轴对齐包围盒和 BVH 树节点, 以存储接下来需要用到的各种数据.

```
1 struct Triangle {
2     glm::vec3 p1, p2, p3, centroid;
3     int      modelIndex, triangleIndex;
4 };
5 struct AABB {
```

```

6   glm::vec3 min;
7   glm::vec3 max;
8   AABB():
9     min(FLT_MAX), max(-FLT_MAX) {}
10  AABB(const glm::vec3 & min_, const glm::vec3 & max_):
11    min(min_), max(max_) {}
12  void expand(const Triangle & t) {
13    min = glm::min(min, t.p1, t.p2, t.p3);
14    max = glm::max(max, t.p1, t.p2, t.p3);
15  }
16};
17 struct BVHNode {
18   AABB bound;
19   int left = -1, right = -1, start = 0, end = 0;
20   bool isLeaf() const { return left == -1 && right == -1; }
21 };
22 inline bool IntersectAABB(Ray const & ray, AABB const & box) {
23   float tmin = 0, tmax = 1e10f;
24   for (int i = 0; i < 3; ++i) {
25     float t1 = (box.min[i] - ray.Origin[i]) / ray.Direction[i];
26     float t2 = (box.max[i] - ray.Origin[i]) / ray.Direction[i];
27     if (t1 > t2) std::swap(t1, t2);
28     tmin = std::max(tmin, t1);
29     tmax = std::min(tmax, t2);
30     if (tmin > tmax) return false;
31   }
32   return true;
33 }
```

然后实现了 IntersectAABB 函数用于判断光线与 AABB 是否相交.

```

1 inline bool IntersectAABB(Ray const & ray, AABB const & box) {
2   float tmin = 0, tmax = 1e10f;
3   for (int i = 0; i < 3; ++i) {
4     float t1 = (box.min[i] - ray.Origin[i]) / ray.Direction[i];
5     float t2 = (box.max[i] - ray.Origin[i]) / ray.Direction[i];
6     if (t1 > t2) std::swap(t1, t2);
7     tmin = std::max(tmin, t1);
8     tmax = std::min(tmax, t2);
9     if (tmin > tmax) return false;
10 }
```

```

11     return true;
12 }
```

然后是结构体 `BVHIntersector` 的实现. 其中主要包含两个函数: `BuildBVH` 函数递归地计算给定起终点的三角形集合的整体包围盒, 并沿最长轴按质心中值划分 (使用 `nth_element` 函数) 为左右两部分, 生成子节点, 重复此过程直至叶节点 (三角形数较少), 最终构建出用于加速光线求交的 BVH 结构, 其代码如下所示:

```

1 int BuildBVH(int start, int end) {
2     BVHNode node;
3     AABB bound;
4     for (int i = start; i < end; ++i) bound.expand(Triangles[i]);
5     node.bound = bound;
6     int n      = end - start;
7     if (n <= 3) {
8         node.start = start;
9         node.end   = end;
10        int idx   = Nodes.size();
11        Nodes.push_back(node);
12        return idx;
13    }
14    glm::vec3 ex   = bound.max - bound.min;
15    int      axis = (ex.x > ex.y && ex.x > ex.z) ? 0 : ((ex.y > ex.z) ? 1 : 2);
16    int midIdx = start + n / 2;
17    std::nth_element(
18        Triangles.begin() + start,
19        Triangles.begin() + midIdx,
20        Triangles.begin() + end,
21        [axis](const Triangle & a, const Triangle & b) {
22            return a.centroid[axis] < b.centroid[axis];
23        });
24    int left   = BuildBVH(start, midIdx);
25    int right  = BuildBVH(midIdx, end);
26    node.left  = left;
27    node.right = right;
28    int idx   = Nodes.size();
29    Nodes.push_back(node);
30    return idx;
31 }
```

`IntersectRay` 函数则递归地遍历 BVH 树, 首先判断光线与当前节点的 AABB 是否相交, 如果不相交则直接返回; 如果相交且当前节点为叶节点, 则遍历该节点内的所有三角形进行求交; 如果当前节点不是叶节点, 则递

归地对其左右子节点进行求交，最终返回最近的交点。找到最近的交点后插值计算交点的颜色等性质。其代码如下所示：

```

1 RayHit IntersectRay(Ray const& ray) const {
2     RayHit result;
3     result.IntersectState = false;
4     float tmin = 1e7, umin, vmin;
5     int modelIdx, meshIdx;
6     if (Nodes.empty()) return result;
7     std::stack<int> boundStack;
8     boundStack.push((int) Nodes.size() - 1);
9     Intersection its;
10    while (! boundStack.empty()) {
11        int idx = boundStack.top(); boundStack.pop();
12        BVHNode const & node = Nodes[idx];
13        if (! IntersectAABB(ray, node.bound)) continue;
14        if (node.isLeaf()) {
15            for (int i = node.start; i < node.end; ++i) {
16                const Triangle & t = Triangles[i];
17                if (! IntersectTriangle(its, ray, t.p1, t.p2, t.p3)) continue;
18                if (its.t < EPS1 || its.t > tmin) continue;
19                tmin = its.t, umin = its.u, vmin = its.v;
20                modelIdx = t.modelIndex, meshIdx = t.triangleIndex;
21            }
22        } else {
23            boundStack.push(node.left);
24            boundStack.push(node.right);
25        }
26    }
27    if (tmin == 1e7) {
28        result.IntersectState = false; return result;
29    }
30    // caculate intersection info
31    return result;
32 }
```

实现效果

渲染采用的设备为 GeForce RTX 5070.

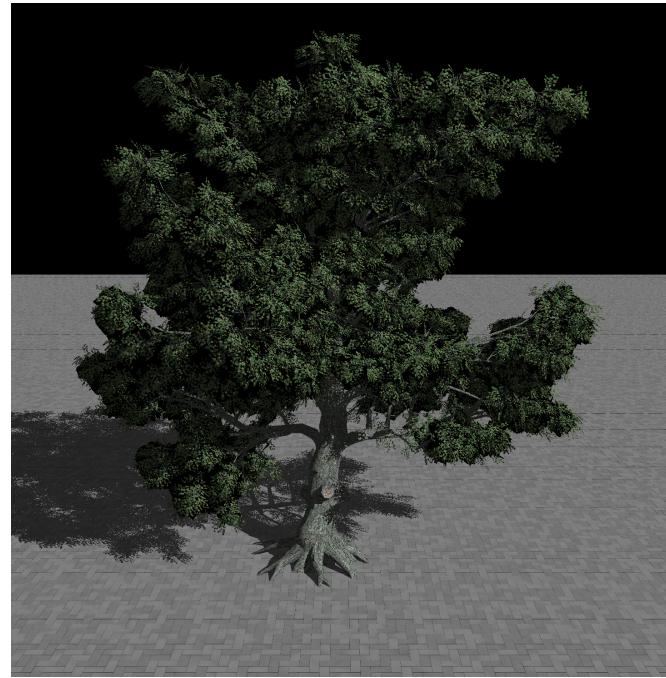


图 7: 对 `oak` 场景进行光线追踪的效果 (Sample Rate = 1, Max Depth = 5), 渲染时间: 5 min

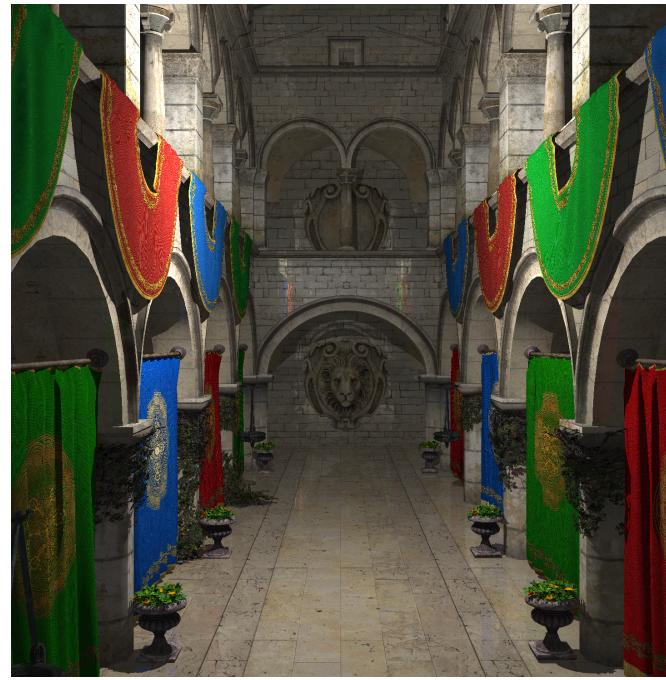


图 8: 对 `sponza` 场景进行光线追踪的效果 (Sample Rate = 1, Max Depth = 5), 渲染时间: 15 min



图 9: 对 `breakfast-room` 场景进行光线追踪的效果 (Sample Rate = 2, Max Depth = 5), 渲染时间: 20 min