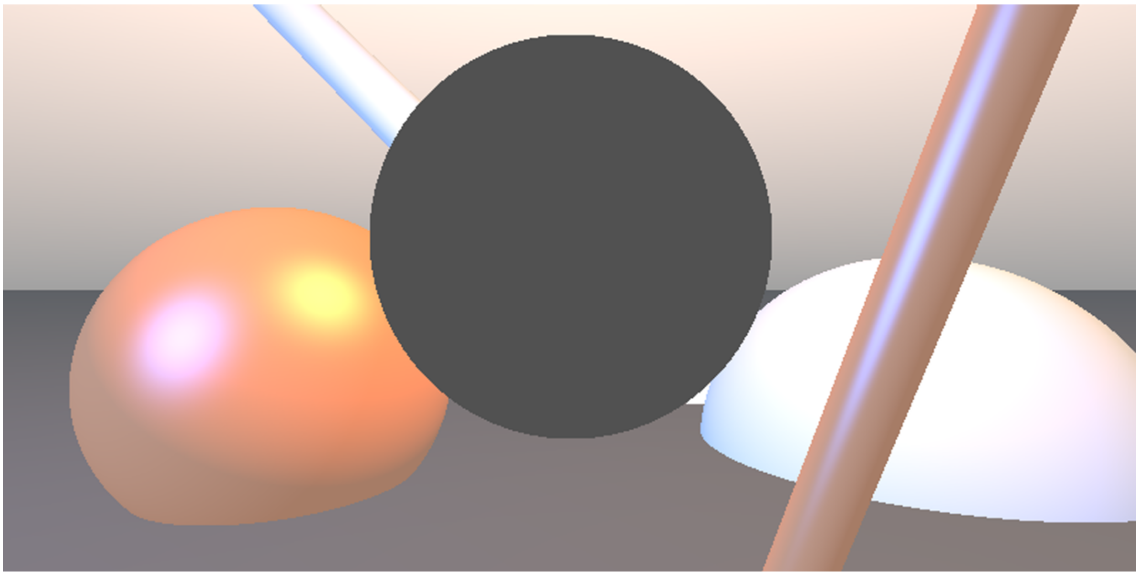
CG Coursework1指导

L.C. 2024.10

实验1难度不高，但是AI对5的理解可能有问题所以很可能会得到错误的结果，需要自己理解探索修改代码。本人分数95+，代码仅供参考。

实验要求

1. 新基元：平面和圆柱体（20分）



   - 要求：我们已经在场景中添加了平面（由法线n和到原点的距离r定义）和圆柱体（由方向o和半径r定义）的定义。请你编写射线-平面（5分）和射线-圆柱体（10分）的相交代码，并解释它们是如何工作的（5分）。注意观察球体是如何进行相交计算的。对于平面和圆柱体，请保持与Sphere相同的函数签名，包括HitInfo、法线和材质。
   - 主要任务：
     a. 实现射线-平面相交（5分）

```
// 1a. 计算射线与平面的相交
HitInfo intersectPlane(const Ray ray,const Plane plane, const float
tMin, const float tMax) {
#ifdef SOLUTION_CYLINDER_AND_PLANE
    // 计算射线方向与平面法线的点积
    float denom = dot(ray.direction, plane.normal);

    // 如果点积接近零，射线平行于平面，无相交
    if (abs(denom) < 0.0001) {
        return getEmptyHit();
    }

    // 计算从射线原点到平面的距离
    float t = -(dot(ray.origin, plane.normal) - plane.d) / denom;

    // 检查交点是否在有效范围内
    if (t < tMin || t > tMax) {
        return getEmptyHit();
    }
```

```
    // 计算交点位置
    vec3 hitPosition = ray.origin + t * ray.direction;

    // 确定法线方向（始终指向射线来源）
    vec3 normal;
    if (denom < 0.0) {
        normal = plane.normal;
    } else {
        normal = -plane.normal;
    }

    // 返回相交信息
    return HitInfo(
        true,           // 发生相交
        t,              // 相交距离
        hitPosition,    // 相交位置
        normal,         // 相交点法线
        plane.material, // 平面材质
        denom < 0.0     // 是否进入物体（对平面来说总是true）
    );
#endif
        return getEmptyHit();
}

float lengthSquared(vec3 x) {
    return dot(x, x);
}
```

b. 实现射线-圆柱体相交（10分）

```
// 1b. 计算射线与圆柱体的相交
HitInfo intersectCylinder(const Ray ray, const Cylinder cylinder, const
float tMin, const float tMax) {
#ifdef SOLUTION_CYLINDER_AND_PLANE
    // 将射线变换到圆柱体的局部空间
    vec3 ro = ray.origin - cylinder.position;
    vec3 rd = ray.direction;
    vec3 ca = cylinder.direction;

    // 计算二次方程系数
    float a = dot(rd, rd) - pow(dot(rd, ca), 2.0);
    float b = 2.0 * (dot(ro, rd) - dot(ro, ca) * dot(rd, ca));
    float c = dot(ro, ro) - pow(dot(ro, ca), 2.0) - cylinder.radius *
cylinder.radius;

    // 计算判别式
    float discriminant = b * b - 4.0 * a * c;

    if (discriminant < 0.0) {
        return getEmptyHit(); // 无相交
    }

    // 计算相交点
    float t1 = (-b - sqrt(discriminant)) / (2.0 * a);
    float t2 = (-b + sqrt(discriminant)) / (2.0 * a);
```

```
    // 确保 t1 < t2
    if (t1 > t2) {
        float temp = t1;
        t1 = t2;
        t2 = temp;
    }

    float t;
    bool entering;

    if (t1 >= tMin && t1 <= tMax) {
        t = t1;
        entering = true;
    } else if (t2 >= tMin && t2 <= tMax) {
        t = t2;
        entering = false;
    } else {
        return getEmptyHit(); // 相交点不在有效范围内
    }

    // 计算相交点位置
    vec3 hitPosition = ray.origin + t * ray.direction;

    // 计算相交点法线
    vec3 normal = normalize(hitPosition - cylinder.position - ca *
dot(hitPosition - cylinder.position, ca));

    if (!entering) {
        normal = -normal;
    }

    // 返回相交信息
    return HitInfo(
        true,               // 发生相交
        t,                  // 相交距离
        hitPosition,        // 相交位置
        normal,             // 相交点法线
        cylinder.material,  // 圆柱体材质
        entering            // 是否进入物体
    );
#endif
    return getEmptyHit();
}
```

c. 解释这些相交计算的原理（5分）

```
平面与射线相交的原理：
1. 射线方程：P(t) = O + tD
   其中 O 是射线起点，D 是射线方向，t 是参数

2. 平面方程：dot(N, P) - d = 0
   其中 N 是平面法向量，P 是平面上任意点，d 是平面到原点的有符号距离

3. 求解步骤：
```

```
   - 将射线方程代入平面方程：dot(N, O + tD) - d = 0
   - 展开：dot(N, O) + t * dot(N, D) - d = 0
   - 求解 t：t = (d - dot(N, O)) / dot(N, D)
   - 当 dot(N, D) 接近0时，射线平行于平面，无交点
   - 检查 t 是否在有效范围内 [tMin, tMax]

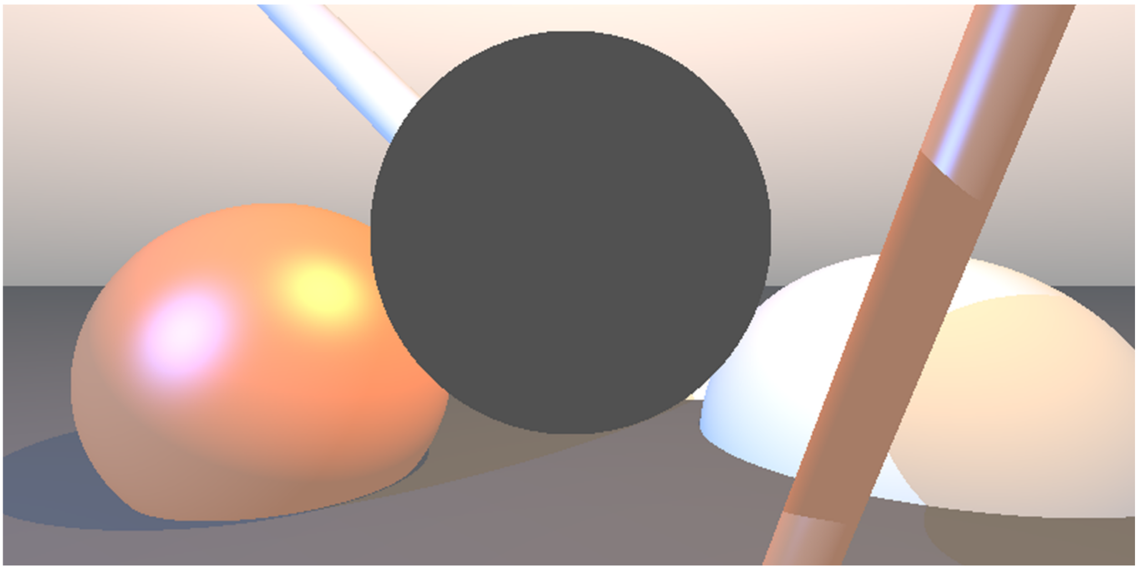
圆柱体与射线相交的原理：
1. 圆柱体可以描述为：到中心轴的距离等于半径r的点的集合
   - 中心轴由点 P 和方向向量 D 定义

2. 求解步骤：
   - 将问题转换到圆柱体的局部坐标系
   - 射线到轴线的距离可以用向量叉乘和点乘表示
   - 构建二次方程：at² + bt + c = 0
     其中：
     a = dot(rd, rd) - pow(dot(rd, ca), 2)
     b = 2(dot(ro, rd) - dot(ro, ca)dot(rd, ca))
     c = dot(ro, ro) - pow(dot(ro, ca), 2) - r²
     rd 是射线方向，ro 是射线原点（相对圆柱体位置），ca 是圆柱体轴向
   - 求解二次方程得到交点参数 t
   - 检查 t 是否在有效范围内

注意事项：
1. 所有交点计算都需要考虑数值精度问题，使用小偏移量避免自相交
2. 需要正确计算交点处的法向量，这对后续的光照计算很重要
3. 对于圆柱体，需要特别注意轴向向量的归一化
4. 相交测试要考虑物体内部和外部的情况，这对折射计算很重要
```

- 解决方法：
  - 射线-平面相交：使用平面方程和射线方程，求解交点。
  - 射线-圆柱体相交：将问题分解为射线与无限长圆柱体的相交，然后检查交点是否在有限长度内。
  - 解释时，关注几何原理和数学推导。
- 细节指导：
  - 在 `intersectPlane` 和 `intersectCylinder` 函数中实现。
  - 对于平面，使用射线方程和平面方程求解交点。
  - 对于圆柱体，将问题分解为射线与无限长圆柱体的相交，然后检查交点是否在有限长度内。

2. 投射阴影（10分）



- 要求：我们已经讨论了阴影、反射和折射的工作原理，以及如何在某些条件下将递归展开为循环。框架中包含了这种遍历的骨架，但没有阴影、反射和折射的代码。首先，在着色中添加阴影测试（10分）。

```
// 2. 在着色中添加阴影测试

// 计算来自单个光源的着色
vec3 shadeFromLight(
  const Scene scene,
  const Ray ray,
  const HitInfo hit_info,
  const PointLight light)
{
    // ... (计算漫反射和镜面反射)
  vec3 hitToLight = light.position - hit_info.position;

  vec3 lightDirection = normalize(hitToLight);
  vec3 viewDirection = normalize(hit_info.position - ray.origin);
  vec3 reflectedDirection = reflect(viewDirection, hit_info.normal);
  float diffuse_term = max(0.0, dot(lightDirection, hit_info.normal));
  float specular_term  = pow(max(0.0, dot(lightDirection,
reflectedDirection)), hit_info.material.glossiness);

#ifdef SOLUTION_SHADOW
    // 创建从交点到光源的阴影射线
    Ray shadowRay;
    shadowRay.origin = hit_info.position + hit_info.normal * 0.001; // 避
免自遮挡
    shadowRay.direction = normalize(hitToLight);

    // 检查射线到光源的距离内是否有遮挡物
    float distanceToLight = length(hitToLight);
    HitInfo shadowHit = intersectScene(scene, shadowRay, 0.001,
distanceToLight - 0.001);

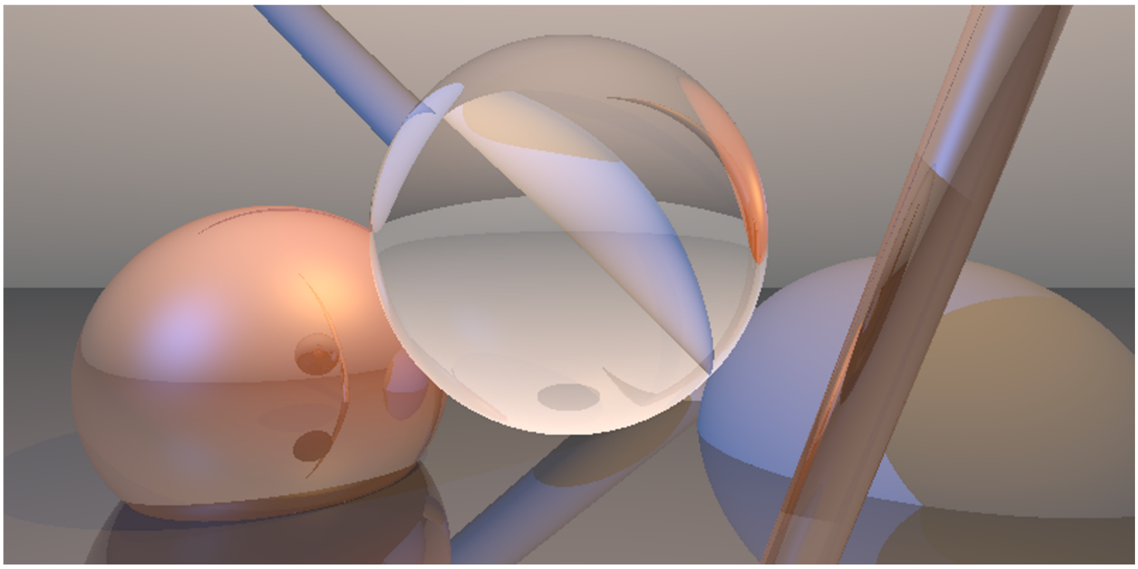
    // 如果有物体遮挡，则该点在阴影中
    float visibility = shadowHit.hit ? 0.0 : 1.0;
#else
    float visibility = 1.0;
#endif
```

```
    return    visibility *
              light.color * (
              specular_term * hit_info.material.specular +
              diffuse_term * hit_info.material.diffuse);
}
```

主要任务：实现阴影检测算法

- 解决方法：
    - 实现阴影射线：从交点向光源发射射线。
    - 检查阴影射线是否与场景中的其他物体相交。
    - 如果相交，说明该点在阴影中，调整其亮度。
- 阴影实现：
    - 在 `shadeFromLight` 函数中的 `SOLUTION_SHADOW` 宏定义块内实现。
    - 从交点向光源发射阴影射线，检查是否与其他物体相交。

3. 添加反射和折射（24分）



- 要求：代码已经包含了迭代执行射线遍历的循环，剩下要添加的是计算反射方向（12分）和折射方向的代码。折射方向分为两个任务：1）反弹射线的主要代码流程，包括折射常数（10分）；2）正确使用enteringPrimitive标志来跟踪折射率IOR（2分），总计12分。注意玻璃球后面足够远的物体如何呈现镜像效果。虽然这里没有显示，但在玻璃球后面且靠近的物体只会显得扭曲。
- 主要任务：
  a. 计算反射方向（12分）

```
// 反射权重计算
#ifdef SOLUTION_REFLECTION_REFRACTION
        // 检查材质是否有反射特性
        if(currentHitInfo.material.reflection > 0.0) {
            reflectionWeight *= currentHitInfo.material.reflection;
        } else {
            break;  // 如果没有反射,直接退出循环
        }
#else
        reflectionWeight *= 0.5;
#endif
```

```
        //  菲涅耳效应
#ifdef SOLUTION_FRESNEL
        float fresnelFactor = fresnel(normalize(currentRay.direction),
                                      currentHitInfo.normal,
                                      sourceIOR,
                                      destIOR);
        reflectionWeight *= fresnelFactor;
#else
        reflectionWeight *= 0.5;
#endif

      Ray nextRay;

      // 计算反射射线
#ifdef SOLUTION_REFLECTION_REFRACTION
        // 计算反射射线
        nextRay.origin = currentHitInfo.position + currentHitInfo.normal
* 0.001;  // 略微偏移以避免自相交
        nextRay.direction = reflect(currentRay.direction,
currentHitInfo.normal);
#endif
```

b. 计算折射方向（12分，包括折射率处理）

```
//  折射权重计算
#ifdef SOLUTION_REFLECTION_REFRACTION
        if(currentHitInfo.material.refraction > 0.0) {
            refractionWeight *= currentHitInfo.material.refraction;
        } else {
            break;  // 如果没有折射,直接退出循环
        }
#else
        refractionWeight *= 0.5;
#endif

    //  菲涅耳效应对折射的影响
#ifdef SOLUTION_FRESNEL
        float fresnelFactor = fresnel(normalize(currentRay.direction),
                                      currentHitInfo.normal,
                                      sourceIOR,
                                      destIOR);
        refractionWeight *= (1.0 - fresnelFactor);
#endif

      Ray nextRay;

    //  计算折射射线
#ifdef SOLUTION_REFLECTION_REFRACTION
        // 确定折射率
        if(currentHitInfo.enteringPrimitive) {
            sourceIOR = 1.0;  // 空气的折射率
            destIOR = currentHitInfo.material.ior;
            currentIOR = destIOR;
        } else {
            sourceIOR = currentIOR;
```

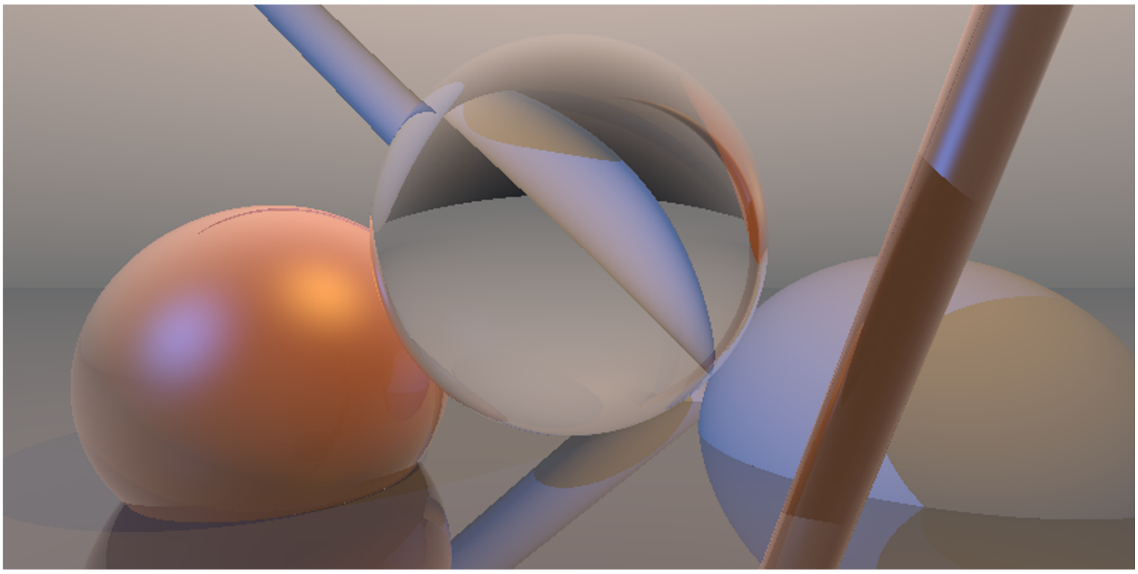
```
            destIOR = 1.0;  // 回到空气中
            currentIOR = destIOR;
        }

        // 计算折射方向
        vec3 n = currentHitInfo.normal;
        float eta = sourceIOR / destIOR;
        float c1 = -dot(currentRay.direction, n);
        float c2 = sqrt(1.0 - eta * eta * (1.0 - c1 * c1));

        nextRay.direction = normalize(eta * currentRay.direction + (eta
* c1 - c2) * n);
        nextRay.origin = currentHitInfo.position - n * 0.001;  // 略微偏移
以避免自相交
        currentRay = nextRay;
#endif
```

- 解决方法：
  - 反射：使用入射方向和表面法线计算反射方向。
  - 折射：使用Snell定律计算折射方向，考虑入射角和两种介质的折射率。
  - 实现递归或迭代的光线追踪算法，处理多次反射和折射。
- 反射和折射实现：
  - 在 `colorForFragment` 函数中的 `SOLUTION_REFLECTION_REFRACTION` 宏定义块内实现。
  - 计算反射方向：reflect(入射方向, 法线)
  - 计算折射方向：使用Snell定律

4. 菲涅耳效应（10分）



- 要求：带有反射和折射的图像看起来还不错，但玻璃可能看起来不像玻璃，因为它在整个球体上都显示相等强度的反射和折射。上面显示了这样一幅图像。在这里和现实中，反射和折射强度是变化的，使得wrefect和wrefract的和为1或更小，所以这里我们简单假设wrefect = 1 - wrefract。这种权重取决于视线方向和击中点的法线。反射通常在接近球体边缘的掠射角度上最强，而折射在中心最强。做一些研究，找出如何计算这些权重，告诉我们你使用了什么方法（5分），并在fresnel函数中实现它（5分）。使用单个点积或Schlick近似可能是一个好的起点。上面的图像是使用点积解决方案生成的，Schlick方法可能看起来不同。

- 主要任务：
  a. 研究并说明如何计算反射和折射的权重（5分）

```
在光学中，当光线从一种介质进入另一种介质时，会同时发生反射和折射。反射和折射的强度比例（权重）可以通过以下几种方法计算：

1. 点积（Cosine）方法：
这是最简单的近似方法，基本原理是反射强度与入射角余弦成反比
计算方法：R = 1 - |dot(V, N)|，
其中：V 是视线方向向量（归一化），N 是表面法线向量（归一化），R 是反射比例，1-R 则为折射比例

优点：计算简单，效果还不错
缺点：物理准确性不高


2. Schlick近似：
这是对Fresnel方程的一个实用近似
计算方法：R = R0 + (1 - R0)(1 - cos θ)^5，
其中：R0 = ((n1 - n2)/(n1 + n2))²，n1，n2 是两种介质的折射率，θ 是入射角

优点：比点积方法更准确，计算量适中
缺点：仍是近似，在某些极端情况下可能不够准确


3. 完整Fresnel方程：
这是最准确的物理方法，直接基于电磁波理论推导。对于非极化光：
Rs = |((n1 * cos(θi) - n2 * cos(θt)) / (n1 * cos(θi) + n2 * cos(θt)))|²
Rp = |((n2 * cos(θi) - n1 * cos(θt)) / (n2 * cos(θi) + n1 * cos(θt)))|²
R = (Rs + Rp) / 2
其中：n1, n2 是两种介质的折射率，θi 是入射角，θt 是折射角，可以通过斯涅尔定律计算：n1 * sin(θi) = n2 * sin(θt)，Rs 是s偏振光的反射率，Rp 是p偏振光的反射率，R 是总反射率

优点：物理上最准确，在所有入射角度都能给出正确结果，并能正确处理全反射情况

缺点：计算复杂，且需要进行较多的三角函数运算
```

  b. 在fresnel函数中实现计算（5分）

```
// 4b. 计算菲涅耳效应
float fresnel(const vec3 viewDirection, const vec3 normal, const float sourceIOR, const float destIOR) {
#ifdef SOLUTION_FRESNEL
    // 方法1：点积方法（与参考图片一致）
    vec3 V = normalize(-viewDirection);
    vec3 N = normalize(normal);
    float cosTheta = abs(dot(V, N));
    return 1.0 - cosTheta;

    /*
    // 方法2：Schlick近似方法
    vec3 V = normalize(-viewDirection);
    vec3 N = normalize(normal);
```

```
    float cosTheta = abs(dot(V, N));

    // 计算基础反射率R0
    float r0 = pow((sourceIOR - destIOR) / (sourceIOR + destIOR), 2.0);

    // Schlick近似公式
    return r0 + (1.0 - r0) * pow(1.0 - cosTheta, 5.0);

    // 方法3：完整Fresnel方程
    vec3 V = normalize(-viewDirection);
    vec3 N = normalize(normal);

    // 计算入射角的余弦值
    float cosTheta_i = abs(dot(V, N));

    // 使用斯涅尔定律计算折射角
    float sinTheta_i = sqrt(1.0 - cosTheta_i * cosTheta_i);
    float sinTheta_t = (sourceIOR / destIOR) * sinTheta_i;

    // 检查是否发生全反射
    if(sinTheta_t >= 1.0) {
        return 1.0; // 全反射
    }

    // 计算折射角的余弦值
    float cosTheta_t = sqrt(1.0 - sinTheta_t * sinTheta_t);

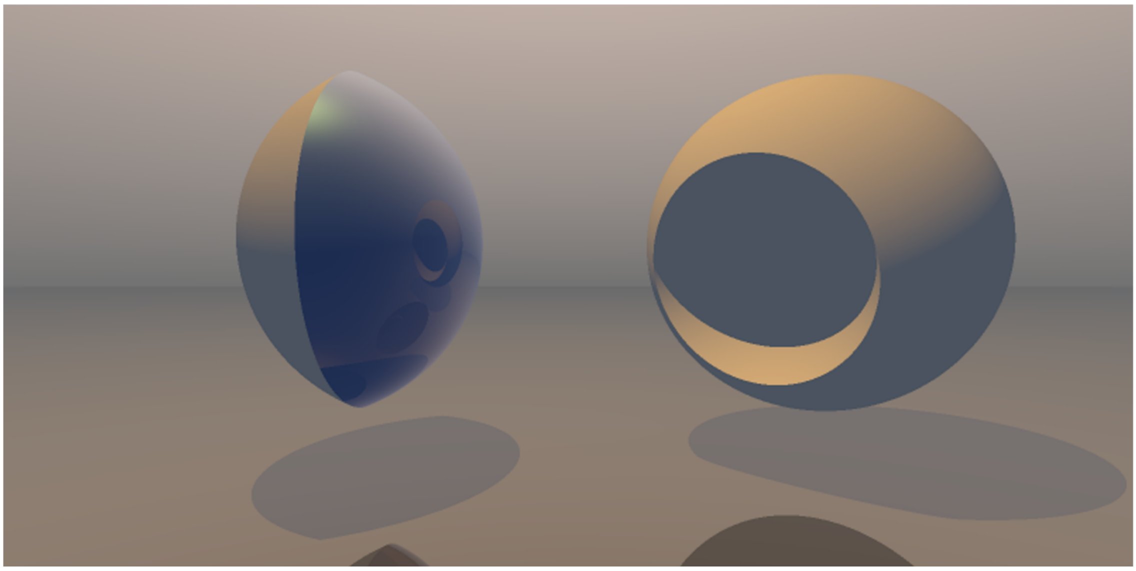
    // 计算s偏振的反射率
    float Rs_num = sourceIOR * cosTheta_i - destIOR * cosTheta_t;
    float Rs_den = sourceIOR * cosTheta_i + destIOR * cosTheta_t;
    float Rs = (Rs_num * Rs_num) / (Rs_den * Rs_den);

    // 计算p偏振的反射率
    float Rp_num = destIOR * cosTheta_i - sourceIOR * cosTheta_t;
    float Rp_den = destIOR * cosTheta_i + sourceIOR * cosTheta_t;
    float Rp = (Rp_num * Rp_num) / (Rp_den * Rp_den);

    // 返回平均反射率
    return (Rs + Rp) * 0.5;
    */
#else
    return 1.0;
#endif
}
```

- 解决方法：
  - 研究菲涅耳方程或其近似（如Schlick近似）。
  - 实现所选方法，计算反射和折射的权重。
  - 在渲染过程中应用这些权重，调整反射和折射的贡献。
- 菲涅耳效应实现：
  - 在 `fresnel` 函数中的 `SOLUTION_FRESNEL` 宏定义块内实现。
  - 使用Schlick近似或完整的菲涅耳方程计算反射系数。

5. 布尔运算：月亮悬挂在UCL城市上空（36分）



- 要求：我们现在将实现一个高级基元：形成两个形状A和B组合的布尔运算。组合可以是逻辑"与"或"减"。实现时，将specialScene设置为true以获得一个干净的场景。对于逻辑"与"，形状被定义为同时在A和B内部的所有点的集合（如图左侧所示）。在intersectBoolean中实现这个相交代码（14分）。对于"减"运算，形状被定义为在B上但不在A内的所有点，加上在A上且在B内的点，即B-A（图右侧）。修改intersectBoolean以处理这两种类型的布尔运算，并将类型翻转为"减"来测试（16分）。不允许更改定义块之外的代码。在intersectBoolean中添加注释，解释如何更改HitInfo结构以使其更适用于布尔运算，并说明原因（6分）。
- 主要任务：
  a. 实现"与"运算的相交代码（14分）

```
#ifdef SOLUTION_BOOLEAN
    // 获取射线与两个球体的相交信息
    // Get intersection information for both spheres
    HitInfo hitA = intersectSphere(ray, boolean.spheres[0], tMin, tMax);
    HitInfo hitB = intersectSphere(ray, boolean.spheres[1], tMin, tMax);

    if (boolean.mode == BOOLEAN_MODE_AND) {
        // 处理交集运算 / Handle intersection operation
        if (!hitA.hit || !hitB.hit) {
            return getEmptyHit();
        }

        // 取较远的入射点作为实际交点
        float t = max(hitA.t, hitB.t);

        // 检查是否在有效范围内
        if (!isTInInterval(t, tMin, tMax)) {
            return getEmptyHit();
        }

        // 使用距离较远的那个球体的表面信息
        vec3 hitPoint = ray.origin + ray.direction * t;
        if (abs(t - hitA.t) < 0.001) {
            return HitInfo(true, t, hitPoint, hitA.normal,
hitA.material, true);
        } else {
            return HitInfo(true, t, hitPoint, hitB.normal,
hitB.material, true);
```

```
        }
    }
```

b. 扩展代码以处理"减"运算（16分）

```
// MINUS操作（差集）
    else if (boolean.mode == BOOLEAN_MODE_MINUS) {
        // 处理差集运算 B-A / Handle subtraction operation B-A
        if (!hitB.hit) {
            return getEmptyHit();
        }

        bool startInA = inside(ray.origin, boolean.spheres[0]);
        vec3 hitBPos = ray.origin + ray.direction * hitB.t;
        bool hitBInA = inside(hitBPos, boolean.spheres[0]);

        // 如果起点在A内且射线和A有交点
        if (startInA && hitA.hit) {
            // 先要从A出来
            vec3 exitAPos = ray.origin + ray.direction * hitA.t;
            // 确保出射点在B内部
            if (inside(exitAPos, boolean.spheres[1])) {
                return HitInfo(
                    true,
                    hitA.t,
                    exitAPos,
                    -hitA.normal,  // 从A内部出来，法线朝外
                    boolean.spheres[1].material,
                    true
                );
            }
            return getEmptyHit();
        }

        // 如果射线与B的交点在A外部，保留B的交点
        if (!hitBInA) {
            if (hitA.hit && hitA.t < hitB.t) {
                // 如果先和A相交，这个交点需要在B内部
                vec3 hitAPos = ray.origin + ray.direction * hitA.t;
                if (inside(hitAPos, boolean.spheres[1])) {
                    // 使用A的法线表示切割面
                    return HitInfo(
                        true,
                        hitA.t,
                        hitAPos,
                        hitA.normal,
                        boolean.spheres[1].material,  // 使用B的材质
                        true
                    );
                }
            }
            // 直接返回B的交点
            return hitB;
        }
```

```
        // 如果B的交点在A内，需要找到从A出来的点
        if (hitA.hit) {
            Ray exitRay = Ray(hitBPos + ray.direction * 0.001,
ray.direction);
            HitInfo exitA = intersectSphere(exitRay, boolean.spheres[0],
0.001, tMax);
            if (exitA.hit && inside(exitRay.origin + exitRay.direction *
exitA.t, boolean.spheres[1])) {
                vec3 exitPos = exitRay.origin + exitRay.direction *
exitA.t;
                return HitInfo(
                    true,
                    hitB.t + exitA.t,
                    exitPos,
                    exitA.normal,
                    boolean.spheres[1].material,
                    true
                );
            }
        }

        return getEmptyHit();
    }
#endif
    return getEmptyHit();
}
```

c. 解释如何改进HitInfo结构以更好地支持布尔运算（6分）

````
改进建议:添加多重交点信息:
```glsl
struct HitInfo {
    bool hit;
    float t;                // 第一个交点的距离
    float t_exit;           // 出射点距离（对于透明物体和布尔运算很重要）
    vec3 position;          // 入射点位置
    vec3 exit_position;     // 出射点位置
    vec3 normal;
    Material material;
    bool enteringPrimitive;
    int objectID;           // 标识与哪个物体相交
    int operationType;      // 布尔运算类型（AND、MINUS等）
};
```

改进理由:
1. 多重交点跟踪:
   - 布尔运算经常需要同时知道射线与物体的入射点和出射点
   - 当前结构只能记录单个交点，导致需要多次射线追踪计算
   - 添加t_exit和exit_position可以一次性存储完整的相交信息

2. 物体身份识别:
   - 添加objectID可以跟踪射线究竟与哪个物体相交
   - 在复杂的布尔运算中，这对于确定使用哪个物体的材质和属性很重要
   - 有助于处理多层嵌套的布尔运算
````

```
3. 运算类型记录：
   - operationType字段可以记录当前交点涉及的布尔运算类型
   - 这对于处理复杂的布尔运算组合很有帮助
   - 可以更容易地实现嵌套的布尔运算

这些改进的好处：
1. 性能提升：
   - 减少重复的相交计算
   - 避免多次射线追踪
   - 更高效地处理复杂的布尔运算场景

2. 更好的健壮性：
   - 更容易处理边界情况
   - 减少数值精度问题
   - 提供更完整的相交信息

3. 扩展性：
   - 更容易支持复杂的布尔运算组合
   - 为未来添加新的布尔运算类型提供基础
   - 便于实现更高级的渲染效果
```

解决方法：

- 实现"与"运算：检查射线是否同时与两个形状相交，并取最远的入射点和最近的出射点。
- 实现"减"运算：检查射线与B的相交，然后检查这些交点是否在A内部。
- 优化HitInfo结构：考虑添加多个交点信息，以便更好地处理复杂的布尔运算。

◦ 布尔运算实现：
  - 在 `intersectBoolean` 函数中的 `SOLUTION_BOOLEAN` 宏定义块内实现。
  - 对于"与"运算，找到两个形状相交部分的入射点和出射点。
  - 对于"减"运算，检查射线与B的相交点是否在A内部。

完整代码部分(无注释)

```
#define SOLUTION_CYLINDER_AND_PLANE
#define SOLUTION_SHADOW
#define SOLUTION_REFLECTION_REFRACTION
#define SOLUTION_FRESNEL

#define SOLUTION_BOOLEAN
// These are preprocessor directives used to control code compilation
// When you complete the corresponding parts, you can uncomment them to enable
the respective code blocks
// They define macros that can be replaced with specific values during
compilation to control program behavior
// For example, if you uncomment #define SOLUTION_SHADOW, all occurrences of
SOLUTION_SHADOW will be replaced with empty space during compilation
// This is typically used to implement feature toggles for different functional
modules
```

```
precision highp float;
uniform ivec2 viewport;

// Defines a point light with position and color
struct PointLight {
    vec3 position;
    vec3 color;
};

// Defines material properties including diffuse, specular, and optical
characteristics
struct Material {
    vec3  diffuse;
    vec3  specular;
    float glossiness;
    float reflection;
    float refraction;
    float ior;
};

// Defines a sphere primitive with position, radius, and material
struct Sphere {
    vec3 position;
    float radius;
    Material material;
};

// Defines a plane primitive with normal, distance from origin, and material
struct Plane {
    vec3 normal;
    float d;
    Material material;
};

// Defines a cylinder primitive with position, direction, radius, and material
struct Cylinder {
    vec3 position;
    vec3 direction;
    float radius;
    Material material;
};

// Boolean operation modes for combining primitives
const int BOOLEAN_MODE_AND = 0;         // and
const int BOOLEAN_MODE_MINUS = 1;       // minus

// Defines boolean operations between two spheres
struct Boolean {
    Sphere spheres[2];
    int mode;
};

// Scene configuration constants
const int lightCount = 2;
const int sphereCount = 3;
```

```
const int planeCount = 1;
const int cylinderCount = 2;
const int booleanCount = 2;

// Defines the complete scene with lights and primitives
struct Scene {
    vec3 ambient;
    PointLight[lightCount] lights;
    Sphere[sphereCount] spheres;
    Plane[planeCount] planes;
    Cylinder[cylinderCount] cylinders;
    Boolean[booleanCount] booleans;
};

// Defines a ray with origin and direction
struct Ray {
    vec3 origin;
    vec3 direction;
};

// Contains all information pertaining to a ray/object intersection
struct HitInfo {
    bool hit;
    float t;
    vec3 position;
    vec3 normal;
    Material material;
    bool enteringPrimitive;
};

// Returns an empty hit info structure
HitInfo getEmptyHit() {
    return HitInfo(
        false,
        0.0,
        vec3(0.0),
        vec3(0.0),
        Material(vec3(0.0), vec3(0.0), 0.0, 0.0, 0.0, 0.0),
        false);
}

// Sorts the two t values such that t1 is smaller than t2
void sortT(inout float t1, inout float t2) {
    // Make t1 the smaller t
    if(t2 < t1)  {
        float temp = t1;
        t1 = t2;
        t2 = temp;
    }
}

// Tests if t is in an interval
bool isTInInterval(const float t, const float tMin, const float tMax) {
    return t > tMin && t < tMax;
}
```

```
// Get the smallest t in an interval.
bool getSmallestTInInterval(float t0, float t1, const float tMin, const float
tMax, inout float smallestTInInterval) {

    sortT(t0, t1);

    // As t0 is smaller, test this first
    if(isTInInterval(t0, tMin, tMax)) {
        smallestTInInterval = t0;
        return true;
    }

    // If t0 was not in the interval, still t1 could be
    if(isTInInterval(t1, tMin, tMax)) {
        smallestTInInterval = t1;
        return true;
    }

    // None was
    return false;
}

// Calculates the intersection between a ray and a sphere
// Returns HitInfo containing intersection details if hit occurs, otherwise
returns empty hit
// Parameters:
//   ray: The ray to test intersection with
//   sphere: The sphere to test against
//   tMin: Minimum distance along ray to consider for intersection
//   tMax: Maximum distance along ray to consider for intersection
HitInfo intersectSphere(const Ray ray, const Sphere sphere, const float tMin,
const float tMax) {

    vec3 to_sphere = ray.origin - sphere.position;

    float a = dot(ray.direction, ray.direction);
    float b = 2.0 * dot(ray.direction, to_sphere);
    float c = dot(to_sphere, to_sphere) - sphere.radius * sphere.radius;
    float D = b * b - 4.0 * a * c;
    if (D > 0.0)
    {
        float t0 = (-b - sqrt(D)) / (2.0 * a);
        float t1 = (-b + sqrt(D)) / (2.0 * a);

        float smallestTInInterval;
        if(!getSmallestTInInterval(t0, t1, tMin, tMax, smallestTInInterval)) {
          return getEmptyHit();
        }

        vec3 hitPosition = ray.origin + smallestTInInterval * ray.direction;


        //Checking if we're inside the sphere by checking if the ray's origin is
inside. If we are, then the normal
        //at the intersection surface points towards the center. Otherwise, if we
are outside the sphere, then the normal
```

```
        //at the intersection surface points outwards from the sphere's center.
This is important for refraction.
        vec3 normal =
            length(ray.origin - sphere.position) < sphere.radius + 0.001?
            -normalize(hitPosition - sphere.position):
            normalize(hitPosition - sphere.position);

        //Checking if we're inside the sphere by checking if the ray's origin is
inside,
        // but this time for IOR bookkeeping.
        //If we are inside, set a flag to say we're leaving. If we are outside,
set the flag to say we're entering.
        //This is also important for refraction.
        bool enteringPrimitive =
            length(ray.origin - sphere.position) < sphere.radius + 0.001 ?
            false:
            true;

        return HitInfo(
            true,
            smallestTInInterval,
            hitPosition,
            normal,
            sphere.material,
            enteringPrimitive);
    }
    return getEmptyHit();
}

// Experiment 1
// 1a. Calculate intersection between ray and plane
HitInfo intersectPlane(const Ray ray,const Plane plane, const float tMin, const
float tMax) {
#ifdef SOLUTION_CYLINDER_AND_PLANE
    // Calculate dot product between ray direction and plane normal
    float denom = dot(ray.direction, plane.normal);

    // If dot product is close to zero, ray is parallel to plane - no
intersection
    if (abs(denom) < 0.0001) {
        return getEmptyHit();
    }

    // Calculate distance from ray origin to plane
    float t = -(dot(ray.origin, plane.normal) - plane.d) / denom;

    // Check if intersection point is within valid range
    if (t < tMin || t > tMax) {
        return getEmptyHit();
    }

    // Calculate intersection position
    vec3 hitPosition = ray.origin + t * ray.direction;

    // Determine normal direction (always pointing towards ray origin)
    vec3 normal;
```

```
    if (denom < 0.0) {
        normal = plane.normal;
    } else {
        normal = -plane.normal;
    }

    // Return intersection information
    return HitInfo(
        true,           // Intersection occurred
        t,              // Distance to intersection
        hitPosition,    // Intersection position
        normal,         // Surface normal at intersection
        plane.material, // Plane material
        denom < 0.0     // Whether entering object (always true for planes)
    );
#endif
    return getEmptyHit();
}

float lengthSquared(vec3 x) {
    return dot(x, x);
}

// 1b. Calculate intersection between ray and cylinder
HitInfo intersectCylinder(const Ray ray, const Cylinder cylinder, const float
tMin, const float tMax) {
#ifdef SOLUTION_CYLINDER_AND_PLANE
    // Transform ray to cylinder's local space
    vec3 ro = ray.origin - cylinder.position;
    vec3 rd = ray.direction;
    vec3 ca = cylinder.direction;

    // Calculate quadratic equation coefficients
    float a = dot(rd, rd) - pow(dot(rd, ca), 2.0);
    float b = 2.0 * (dot(ro, rd) - dot(ro, ca) * dot(rd, ca));
    float c = dot(ro, ro) - pow(dot(ro, ca), 2.0) - cylinder.radius *
cylinder.radius;

    // Calculate discriminant
    float discriminant = b * b - 4.0 * a * c;

    if (discriminant < 0.0) {
        return getEmptyHit(); // No intersection
    }

    // Calculate intersection points
    float t1 = (-b - sqrt(discriminant)) / (2.0 * a);
    float t2 = (-b + sqrt(discriminant)) / (2.0 * a);

    // Ensure t1 < t2
    if (t1 > t2) {
        float temp = t1;
        t1 = t2;
        t2 = temp;
    }
```

```
    float t;
    bool entering;

    if (t1 >= tMin && t1 <= tMax) {
        t = t1;
        entering = true;
    } else if (t2 >= tMin && t2 <= tMax) {
        t = t2;
        entering = false;
    } else {
        return getEmptyHit(); // Intersection points outside valid range
    }

    // Calculate intersection position
    vec3 hitPosition = ray.origin + t * ray.direction;

    // Calculate surface normal at intersection point
    vec3 normal = normalize(hitPosition - cylinder.position - ca *
dot(hitPosition - cylinder.position, ca));

    if (!entering) {
        normal = -normal;
    }

    // Return intersection information
    return HitInfo(
        true,                // Intersection occurred
        t,                   // Distance to intersection
        hitPosition,         // Intersection position
        normal,              // Surface normal at intersection
        cylinder.material,   // Cylinder material
        entering            // Whether entering object
    );
#endif
    return getEmptyHit();
}

// 1c. Principles of Intersection Calculations
/*
Principles of Ray-Plane Intersection:
1. Ray equation: $P(t) = O + tD$
   where O is ray origin, D is ray direction, t is parameter

2. Plane equation: $N \cdot P - d = 0$
   where N is plane normal, P is any point on plane, d is signed distance from
plane to origin

3. Solution steps:
   - Substitute ray equation into plane equation: $N \cdot (O + tD) - d = 0$
   - Expand: $N \cdot O + t(N \cdot D) - d = 0$
   - Solve for t: $t = \frac{d - N \cdot O}{N \cdot D}$
   - When dot(N, D) is close to 0, ray is parallel to plane, no intersection
   - Check if t is within valid range [tMin, tMax]

Principles of Ray-Cylinder Intersection:
1. Cylinder can be described as: set of points with distance r from central axis
```

```
   - Central axis defined by point P and direction vector D

2. Solution steps:
   - Transform problem to cylinder's local coordinate system
   - Distance from ray to axis can be expressed using vector cross and dot
products
   - Construct quadratic equation: $at^2 + bt + c = 0$
     where:
     $a = rd \cdot rd - (rd \cdot ca)^2$
     $b = 2(ro \cdot rd - (ro \cdot ca)(rd \cdot ca))$
     $c = ro \cdot ro - (ro \cdot ca)^2 - r^2$
     rd is ray direction, ro is ray origin (relative to cylinder position), ca is
cylinder axis
   - Solve quadratic equation to get intersection parameter t
   - Check if t is within valid range
*/

// Check if a point is inside a sphere
bool inside(const vec3 position, const Sphere sphere) {
    return length(position - sphere.position) < sphere.radius;
}

// Experiment 5
// 5a. "AND" operation
// 5b. "MINUS" operation
// 5c. Explanation of how to improve HitInfo structure to better support boolean
operations

/*
Here's my thought process on improving the HitInfo structure for boolean
operations:

Proposed Enhanced Structure:

struct HitInfo {
    bool hit;
    float t;                 // Distance to first intersection
    float t_exit;           // Distance to exit point
    vec3 position;          // Entry point position
    vec3 exit_position;     // Exit point position
    vec3 normal;
    Material material;
    bool enteringPrimitive;
    int objectID;           // Which object we hit
    int operationType;      // Type of boolean operation
};

Key Improvements and Justification:

1. Multiple Intersection Tracking
   - Added t_exit and exit_position fields
   - Benefits:
     * Enables single-pass intersection calculations
     * Reduces redundant ray tracing operations
     * Essential for accurate boolean operation handling
```

```
2. Object Identity Management
   - Added objectID field
   - Benefits:
     * Enables proper material attribution in complex operations
     * Facilitates nested boolean operations
     * Improves intersection point classification

3. Operation Type Tracking
   - Added operationType field
   - Benefits:
     * Supports operation precedence in nested booleans
     * Enables complex operation combinations
     * Improves debugging capabilities

Technical Impact:

1. Performance Enhancement
   - Reduces intersection calculations
   - Minimizes redundant ray tracing
   - Optimizes memory access patterns

2. Numerical Stability
   - Better handling of edge cases
   - Reduced floating-point error accumulation
   - More reliable boolean operations

3. Implementation Flexibility
   - Supports future operation types
   - Enables advanced rendering techniques
   - Facilitates operation composition
*/

HitInfo intersectBoolean(const Ray ray, const Boolean boolean, const float tMin,
const float tMax) {
#ifdef SOLUTION_BOOLEAN
    // 获取射线与两个球体的相交信息
    // Get intersection information for both spheres
    HitInfo hitA = intersectSphere(ray, boolean.spheres[0], tMin, tMax);
    HitInfo hitB = intersectSphere(ray, boolean.spheres[1], tMin, tMax);

    if (boolean.mode == BOOLEAN_MODE_AND) {
        // 处理交集运算 / Handle intersection operation (AND)
        if (!hitA.hit || !hitB.hit) {
            return getEmptyHit();
        }

        // Take the farther entry point as the actual intersection
        float t = max(hitA.t, hitB.t);

        // Check if within valid range
        if (!isTInInterval(t, tMin, tMax)) {
            return getEmptyHit();
        }

        // Use surface information from the sphere with the farther intersection
        vec3 hitPoint = ray.origin + ray.direction * t;
```

```
        if (abs(t - hitA.t) < 0.001) {
            return HitInfo(true, t, hitPoint, hitA.normal, hitA.material, true);
        } else {
            return HitInfo(true, t, hitPoint, hitB.normal, hitB.material, true);
        }
    }

    else if (boolean.mode == BOOLEAN_MODE_MINUS) {
        // 处理差集运算 / Handle subtraction operation B-A
        if (!hitB.hit) {
            return getEmptyHit();
        }

        bool startInA = inside(ray.origin, boolean.spheres[0]);
        vec3 hitBPos = ray.origin + ray.direction * hitB.t;
        bool hitBInA = inside(hitBPos, boolean.spheres[0]);

        // If starting point is inside A and ray intersects A
        // 如果起点在A内且射线和A有交点
        if (startInA && hitA.hit) {
            // Must exit A first
            vec3 exitAPos = ray.origin + ray.direction * hitA.t;
            // Ensure exit point is inside B
            if (inside(exitAPos, boolean.spheres[1])) {
                return HitInfo(
                    true,
                    hitA.t,
                    exitAPos,
                    -hitA.normal,  // Normal points outward when exiting A
                    boolean.spheres[1].material,
                    true
                );
            }
            return getEmptyHit();
        }

        // If B's intersection point is outside A, keep B's intersection
        // 如果射线与B的交点在A外部，保留B的交点
        if (!hitBInA) {
            if (hitA.hit && hitA.t < hitB.t) {
                // If intersecting A first, this point must be inside B
                vec3 hitAPos = ray.origin + ray.direction * hitA.t;
                if (inside(hitAPos, boolean.spheres[1])) {
                    // Use A's normal for the cut surface
                    return HitInfo(
                        true,
                        hitA.t,
                        hitAPos,
                        hitA.normal,
                        boolean.spheres[1].material,  // Use B's material
                        true
                    );
                }
            }
            // Return B's intersection directly
            return hitB;
```

```
        }

        // If B's intersection point is inside A, need to find exit point from A
        // 如果B的交点在A内，需要找到从A出来的点
        if (hitA.hit) {
            Ray exitRay = Ray(hitBPos + ray.direction * 0.001, ray.direction);
            HitInfo exitA = intersectSphere(exitRay, boolean.spheres[0], 0.001,
tMax);
            if (exitA.hit && inside(exitRay.origin + exitRay.direction * exitA.t,
boolean.spheres[1])) {
                vec3 exitPos = exitRay.origin + exitRay.direction * exitA.t;
                return HitInfo(
                    true,
                    hitB.t + exitA.t,
                    exitPos,
                    exitA.normal,
                    boolean.spheres[1].material,
                    true
                );
            }
        }

        return getEmptyHit();
    }
#endif
    return getEmptyHit();
}

// Time variable (for animation effects)
uniform float time;

// Compare two HitInfos and return the closer one
HitInfo getBetterHitInfo(const HitInfo oldHitInfo, const HitInfo newHitInfo) {
    if(newHitInfo.hit)
        if(newHitInfo.t < oldHitInfo.t)  // No need to test for the interval,
this has to be done per-primitive
          return newHitInfo;
    return oldHitInfo;
}

// Calculate intersection between ray and entire scene
// 计算射线与整个场景的相交
HitInfo intersectScene(const Scene scene, const Ray ray, const float tMin, const
float tMax) {
    HitInfo bestHitInfo;
    bestHitInfo.t = tMax;
    bestHitInfo.hit = false;

    // Iterate through all objects and find the nearest intersection point
    for (int i = 0; i < booleanCount; ++i) {
        bestHitInfo = getBetterHitInfo(bestHitInfo, intersectBoolean(ray,
scene.booleans[i], tMin, tMax));
    }

    for (int i = 0; i < planeCount; ++i) {
```

```
        bestHitInfo = getBetterHitInfo(bestHitInfo, intersectPlane(ray,
scene.planes[i], tMin, tMax));
    }
    for (int i = 0; i < sphereCount; ++i) {
        bestHitInfo = getBetterHitInfo(bestHitInfo, intersectSphere(ray,
scene.spheres[i], tMin, tMax));
    }
    for (int i = 0; i < cylinderCount; ++i) {
        bestHitInfo = getBetterHitInfo(bestHitInfo, intersectCylinder(ray,
scene.cylinders[i], tMin, tMax));
    }

    return bestHitInfo;
}

// Experiment 2
// 2. Add shadow testing to shading
// Calculate shading from a single light source
vec3 shadeFromLight(
  const Scene scene,
  const Ray ray,
  const HitInfo hit_info,
  const PointLight light)
{
  // Calculate diffuse and specular reflection
  // 计算漫反射和镜面反射
  vec3 hitToLight = light.position - hit_info.position;

  vec3 lightDirection = normalize(hitToLight);
  vec3 viewDirection = normalize(hit_info.position - ray.origin);
  vec3 reflectedDirection = reflect(viewDirection, hit_info.normal);
  float diffuse_term = max(0.0, dot(lightDirection, hit_info.normal));
  float specular_term  = pow(max(0.0, dot(lightDirection, reflectedDirection)),
hit_info.material.glossiness);

#ifdef SOLUTION_SHADOW
    // Create shadow ray from intersection point to light source
    Ray shadowRay;
    shadowRay.origin = hit_info.position + hit_info.normal * 0.001; // Avoid
self-shadowing
    shadowRay.direction = normalize(hitToLight);

    // Check for occluders between intersection point and light
    float distanceToLight = length(hitToLight);
    HitInfo shadowHit = intersectScene(scene, shadowRay, 0.001, distanceToLight -
0.001);

    // Point is in shadow if there's an occluding object
    float visibility = shadowHit.hit ? 0.0 : 1.0;
#else
    float visibility = 1.0;
#endif
  return    visibility *
            light.color * (
            specular_term * hit_info.material.specular +
            diffuse_term * hit_info.material.diffuse);
```

```
}

// Calculate background color
vec3 background(const Ray ray) {
  // A simple implicit sky that can be used for the background
  return vec3(0.2) + vec3(0.8, 0.6, 0.5) * max(0.0, ray.direction.y);
}

// Calculate overall shading
// It seems to be a WebGL issue that the third parameter needs to be inout instea
dof const on Tobias' machine
vec3 shade(const Scene scene, const Ray ray, inout HitInfo hitInfo) {
    // Combine ambient light and contributions from all light sources
    // 综合环境光和所有光源的贡献
    if(!hitInfo.hit) {
        return background(ray);
    }

    vec3 shading = scene.ambient * hitInfo.material.diffuse;
    for (int i = 0; i < lightCount; ++i) {
        shading += shadeFromLight(scene, ray, hitInfo, scene.lights[i]);
    }
    return shading;
}

// Generate ray based on pixel coordinates
Ray getFragCoordRay(const vec2 frag_coord) {
    float sensorDistance = 1.0;
    vec2 sensorMin = vec2(-1, -0.5);
    vec2 sensorMax = vec2(1, 0.5);
    vec2 pixelSize = (sensorMax- sensorMin) / vec2(viewport.x, viewport.y);
    vec3 origin = vec3(0, 0, sensorDistance);
    vec3 direction = normalize(vec3(sensorMin + pixelSize * frag_coord, -
sensorDistance));
    // Calculate ray origin and direction
    return Ray(origin, direction);
}

// Experiment 4
// 4a. Understanding Reflection and Refraction Weight Calculations

/*
When light hits the boundary between two materials, it splits into reflected and
refracted rays. Here's how we can calculate their relative strengths:

1. Simple Cosine Method:
A quick way to approximate the reflection ratio based on the viewing angle.
How it works: $R = 1 - |V \cdot N|$
Where:
- V = view direction (normalized)
- N = surface normal (normalized)
- R = reflection ratio
- (1-R) = refraction ratio
Quick and easy, but not physically accurate.

2. Schlick's Approximation:
```

```
A clever approximation of the full Fresnel equations that's widely used in
graphics.
Formula: $R = R_0 + (1 - R_0)(1 - \cos \theta)^5$
Where:
- $R_0 = (\frac{n_1 - n_2}{n_1 + n_2})^2$
- n1, n2 = refractive indices
- $\theta$ = incident angle
Gives good results while being reasonably fast to compute.

3. Full Fresnel Equations:
The physically accurate way to calculate reflection, based on electromagnetic
theory:
$R_s = |\frac{n_1 \cos\theta_i - n_2 \cos\theta_t}{n_1 \cos\theta_i + n_2
\cos\theta_t}|^2$
$R_p = |\frac{n_2 \cos\theta_i - n_1 \cos\theta_t}{n_2 \cos\theta_i + n_1
\cos\theta_t}|^2$
$R = \frac{R_s + R_p}{2}$

Where:
- $n_1, n_2$ = refractive indices
- $\theta_i$ = incident angle
- $\theta_t$ = transmission angle (from Snell's law: $n_1 \sin\theta_i = n_2
\sin\theta_t$)
- Rs = reflection ratio for s-polarized light
- Rp = reflection ratio for p-polarized light
Most accurate but computationally expensive.
*/

// 4b. Calculate Fresnel effect
// 计算菲涅耳效应
float fresnel(const vec3 viewDirection, const vec3 normal, const float sourceIOR,
const float destIOR) {
#ifdef SOLUTION_FRESNEL
    // Method 1: Cosine method (matches reference image)
    // 方法1：点积方法（与参考图片一致）
    vec3 V = normalize(-viewDirection);
    vec3 N = normalize(normal);
    float cosTheta = abs(dot(V, N));
    return 1.0 - cosTheta;

    /*
    // Method 2: Schlick's approximation
    // 方法2：Schlick近似方法
    vec3 V = normalize(-viewDirection);
    vec3 N = normalize(normal);
    float cosTheta = abs(dot(V, N));
    // Calculate base reflectivity R0
    float r0 = pow((sourceIOR - destIOR) / (sourceIOR + destIOR), 2.0);
    // Schlick's approximation formula
    return r0 + (1.0 - r0) * pow(1.0 - cosTheta, 5.0);

    // Method 3: Complete Fresnel equations
    // 方法3：完整Fresnel方程
    vec3 V = normalize(-viewDirection);
    vec3 N = normalize(normal);
    // Calculate cosine of incident angle
```

```
    float cosTheta_i = abs(dot(V, N));
    // Use Snell's law to get transmission angle
    float sinTheta_i = sqrt(1.0 - cosTheta_i * cosTheta_i);
    float sinTheta_t = (sourceIOR / destIOR) * sinTheta_i;
    // Check for total internal reflection
    if(sinTheta_t >= 1.0) {
        return 1.0; // 全反射
    }
    // Calculate cosine of transmission angle
    float cosTheta_t = sqrt(1.0 - sinTheta_t * sinTheta_t);
    // Calculate reflection for s-polarized light
    float Rs_num = sourceIOR * cosTheta_i - destIOR * cosTheta_t;
    float Rs_den = sourceIOR * cosTheta_i + destIOR * cosTheta_t;
    float Rs = (Rs_num * Rs_num) / (Rs_den * Rs_den);
    // Calculate reflection for p-polarized light
    float Rp_num = destIOR * cosTheta_i - sourceIOR * cosTheta_t;
    float Rp_den = destIOR * cosTheta_i + sourceIOR * cosTheta_t;
    float Rp = (Rp_num * Rp_num) / (Rp_den * Rp_den);
    // Return average reflectivity
    return (Rs + Rp) * 0.5;
    */
#else
    return 1.0;
#endif
}

// Experiment 3

// Calculate color for each pixel
// 为每个像素计算颜色
vec3 colorForFragment(const Scene scene, const vec2 fragCoord) {
    // Main rendering logic, including reflection and refraction calculations
    Ray initialRay = getFragCoordRay(fragCoord);
    HitInfo initialHitInfo = intersectScene(scene, initialRay, 0.001, 10000.0);
    vec3 result = shade(scene, initialRay, initialHitInfo);

    Ray currentRay;
    HitInfo currentHitInfo;

    // Compute the reflection
    currentRay = initialRay;
    currentHitInfo = initialHitInfo;

    // The initial strength of the reflection
    float reflectionWeight = 1.0;

    // The initial medium is air
    float currentIOR = 1.0;

    float sourceIOR = 1.0;
    float destIOR = 1.0;

    // 3a. Reflection loop
    const int maxReflectionStepCount = 2;
    for(int i = 0; i < maxReflectionStepCount; i++) {
```

```
      if(!currentHitInfo.hit) break;

// Calculate reflection weight
// 反射权重计算
#ifdef SOLUTION_REFLECTION_REFRACTION
        // Check if material has reflective properties
        if(currentHitInfo.material.reflection > 0.0) {
            reflectionWeight *= currentHitInfo.material.reflection;
        } else {
            break;  // Exit loop if no reflection
        }
#else
        reflectionWeight *= 0.5;
#endif

// Fresnel effect
//  菲涅耳效应
#ifdef SOLUTION_FRESNEL
        float fresnelFactor = fresnel(normalize(currentRay.direction),
                                    currentHitInfo.normal,
                                    sourceIOR,
                                    destIOR);
        reflectionWeight *= fresnelFactor;
#else
        reflectionWeight *= 0.5;
#endif

      Ray nextRay;

// Calculate reflection ray
// 计算反射射线
#ifdef SOLUTION_REFLECTION_REFRACTION
        nextRay.origin = currentHitInfo.position + currentHitInfo.normal * 0.001;
 // Small offset to avoid self-intersection
        nextRay.direction = reflect(currentRay.direction, currentHitInfo.normal);
#endif

      currentRay = nextRay;

      currentHitInfo = intersectScene(scene, currentRay, 0.001, 10000.0);

      result += reflectionWeight * shade(scene, currentRay, currentHitInfo);
    }

    // Compute the refraction
    currentRay = initialRay;
    currentHitInfo = initialHitInfo;

    // The initial strength of the refraction.
    float refractionWeight = 1.0;

    // 3b. Refraction loop
    const int maxRefractionStepCount = 2;
    for(int i = 0; i < maxRefractionStepCount; i++) {

// Calculate refraction weight
```

```
//  折射权重计算
#ifdef SOLUTION_REFLECTION_REFRACTION
        if(currentHitInfo.material.refraction > 0.0) {
            refractionWeight *= currentHitInfo.material.refraction;
        } else {
            break;  // Exit loop if no refraction
        }
#else
        refractionWeight *= 0.5;
#endif

// Fresnel effect
//  菲涅耳效应
#ifdef SOLUTION_FRESNEL
        float fresnelFactor = fresnel(normalize(currentRay.direction),
                                      currentHitInfo.normal,
                                      sourceIOR,
                                      destIOR);
        refractionWeight *= (1.0 - fresnelFactor);
#endif

      Ray nextRay;

// Calculate refraction ray
//  计算折射射线
#ifdef SOLUTION_REFLECTION_REFRACTION
        // Determine refractive indices
        if(currentHitInfo.enteringPrimitive) {
            sourceIOR = 1.0;  // Air refractive index
            destIOR = currentHitInfo.material.ior;
            currentIOR = destIOR;
        } else {
            sourceIOR = currentIOR;
            destIOR = 1.0;  // Back to air
            currentIOR = destIOR;
        }

        // Calculate refraction direction using Snell's law
        vec3 n = currentHitInfo.normal;
        float eta = sourceIOR / destIOR;
        float c1 = -dot(currentRay.direction, n);
        float c2 = sqrt(1.0 - eta * eta * (1.0 - c1 * c1));

        nextRay.direction = normalize(eta * currentRay.direction + (eta * c1 -
c2) * n);
        nextRay.origin = currentHitInfo.position - n * 0.001;  // Small offset to
avoid self-intersection
        currentRay = nextRay;
        currentRay = nextRay;
#endif
      currentHitInfo = intersectScene(scene, currentRay, 0.001, 10000.0);

      result += refractionWeight * shade(scene, currentRay, currentHitInfo);

      if(!currentHitInfo.hit) break;
    }
```

```
  return result;
}

// Common materials
// IOR data comes from https://refractiveindex.info/, otherwise zero
// When reflection and refraction set to zero, it is turned off

Material getDefaultMaterial() {
  return Material(vec3(0.3), vec3(0), 0.0, 0.0, 0.0, 0.0);
}

Material getPaperMaterial() {
  return Material(vec3(0.7, 0.7, 0.7), vec3(0, 0, 0), 5.0, 0.0, 0.0, 0.0);
}

Material getPlasticMaterial() {
    return Material(vec3(0.9, 0.3, 0.1), vec3(1.0), 10.0, 0.9, 0.0, 1.5);
}

Material getGlassMaterial() {
    return Material(vec3(0.0), vec3(0.0), 5.0, 1.0, 1.0, 1.5);
}

Material getSteelMirrorMaterial() {
    return Material(vec3(0.1), vec3(0.3), 20.0, 0.8, 0.0, 2.9);
}

Material getMetaMaterial() {
    return Material(vec3(0.1, 0.2, 0.5), vec3(0.3, 0.7, 0.9), 20.0, 0.8, 0.0,
0.0);
}

vec3 tonemap(const vec3 radiance) {
  const float monitorGamma = 2.0;
  return pow(radiance, vec3(1.0 / monitorGamma));
}


void main() {
    // Setup scene
    Scene scene;
    scene.ambient = vec3(0.12, 0.15, 0.2);
    scene.lights[0].position = vec3(5, 15, -5);
    scene.lights[0].color    = 0.5 * vec3(0.9, 0.5, 0.1);

    scene.lights[1].position = vec3(-15, 5, 2);
    scene.lights[1].color    = 0.5 * vec3(0.1, 0.3, 1.0);

    // Primitives
    bool specialScene = false;

    // Set specialScene to true to implement the task in the below ifdef block
#ifdef SOLUTION_BOOLEAN
    specialScene = true; // 启用特殊场景
#endif
```

```
    if (specialScene) {
        // Boolean scene
        scene.booleans[0].mode = BOOLEAN_MODE_MINUS;

        // sphere A
        scene.booleans[0].spheres[0].position       = vec3(3, 0, -10);
        scene.booleans[0].spheres[0].radius         = 2.75;
        scene.booleans[0].spheres[0].material       = getPaperMaterial();

        // sphere B
        scene.booleans[0].spheres[1].position       = vec3(6, 1, -13);
        scene.booleans[0].spheres[1].radius         = 4.0;
        scene.booleans[0].spheres[1].material       = getPaperMaterial();


        scene.booleans[1].mode = BOOLEAN_MODE_AND;

        scene.booleans[1].spheres[0].position       = vec3(-3.0, 1, -12);
        scene.booleans[1].spheres[0].radius         = 4.0;
        scene.booleans[1].spheres[0].material       = getPaperMaterial();

        scene.booleans[1].spheres[1].position       = vec3(-6.0, 1, -12);
        scene.booleans[1].spheres[1].radius         = 4.0;
        scene.booleans[1].spheres[1].material       = getMetaMaterial();


        scene.planes[0].normal                  = normalize(vec3(0, 0.8, 0));
        scene.planes[0].d                       = -4.5;
        scene.planes[0].material                = getSteelMirrorMaterial();

        scene.lights[0].position = vec3(-5, 25, -5);
        scene.lights[0].color    = vec3(0.9, 0.5, 0.1);

        scene.lights[1].position = vec3(-15, 5, 2);
        scene.lights[1].color    = 0.0 * 0.5 * vec3(0.1, 0.3, 1.0);
    }
    else {
        // normal scene
        scene.spheres[0].position               = vec3(10, -5, -16);
        scene.spheres[0].radius                 = 6.0;
        scene.spheres[0].material               = getPaperMaterial();

        scene.spheres[1].position               = vec3(-7, -2, -13);
        scene.spheres[1].radius                 = 4.0;
        scene.spheres[1].material               = getPlasticMaterial();

        scene.spheres[2].position               = vec3(0, 0.5, -5);
        scene.spheres[2].radius                 = 2.0;
        scene.spheres[2].material               = getGlassMaterial();

        scene.planes[0].normal                  = normalize(vec3(0, 1.0, 0));
        scene.planes[0].d                       = -4.5;
        scene.planes[0].material                = getSteelMirrorMaterial();

        scene.cylinders[0].position             = vec3(-1, 1, -26);
        scene.cylinders[0].direction            = normalize(vec3(-2, 2, -1));
```

```
        scene.cylinders[0].radius                = 1.5;
        scene.cylinders[0].material              = getPaperMaterial();

        scene.cylinders[1].position              = vec3(4, 1, -5);
        scene.cylinders[1].direction             = normalize(vec3(1, 4, 1));
        scene.cylinders[1].radius                = 0.4;
        scene.cylinders[1].material              = getPlasticMaterial();
    }

    // Compute color for fragment
    gl_FragColor.rgb = tonemap(colorForFragment(scene, gl_FragCoord.xy));
    gl_FragColor.a = 1.0;

}
```

完整代码部分(带注释)

```
#define SOLUTION_CYLINDER_AND_PLANE
#define SOLUTION_SHADOW
//#define SOLUTION_REFLECTION_REFRACTION
//#define SOLUTION_FRESNEL

//#define SOLUTION_BOOLEAN

// 这些是预处理指令，用于控制代码的编译。当你完成相应的部分时，可以取消注释来启用相应的代码块。
// 它们的作用是定义了一些宏，这些宏可以在编译时被替换成特定的值，从而控制程序的行为。
// 比如，如果取消注释了#define SOLUTION_SHADOW，那么在编译时，所有包含SOLUTION_SHADOW的地方都会被替换为空。
// 这通常用于实现不同功能模块的开关控制。
precision highp float;
// 设置浮点数精度为高精度
// 在OpenGL着色器中，我们经常需要进行大量的浮点数计算。为了保证计算的精度，这里将浮点数的精度设置为高精度。
uniform ivec2 viewport;
// 定义一个uniform变量，表示视口的大小
// uniform变量是OpenGL中一种特殊的变量，它的值可以在CPU端设置，然后在GPU端使用。
// 这里的viewport变量表示当前渲染窗口的大小，它是一个二维整数向量，分别表示窗口的宽度和高度。

// 以下是各种数据结构的定义
// 这些数据结构定义了场景中各种元素的属性，比如光源、材质、球体、平面、圆柱体等。

// 定义点光源结构体，包含光源的位置和颜色。
struct PointLight {
    vec3 position;
    vec3 color;
};

// 定义材质结构体，包含漫反射颜色、镜面反射颜色、光泽度、反射系数、折射系数和折射率。
struct Material {
    vec3  diffuse;
    vec3  specular;
```

```
    float glossiness;
    float reflection;
    float refraction;
    float ior;
};

// 定义球体结构体，包含球心位置、半径和材质。
struct Sphere {
    vec3 position;
    float radius;
    Material material;
};

// 定义平面结构体，包含平面的法向量、到原点的距离和材质。
struct Plane {
    vec3 normal;
    float d;
    Material material;
};

// 定义圆柱体结构体，包含圆柱体中心位置、方向向量、半径和材质。
struct Cylinder {
    vec3 position;
    vec3 direction;
    float radius;
    Material material;
};

// ... (其他结构体的定义)
const int BOOLEAN_MODE_AND = 0;          // and
const int BOOLEAN_MODE_MINUS = 1;        // minus

struct Boolean {
    Sphere spheres[2];
    int mode;
};

// 场景中物体的数量常量
// 定义了场景中各种物体数量的常量，方便后续使用。
const int lightCount = 2;
const int sphereCount = 3;
const int planeCount = 1;
const int cylinderCount = 2;
const int booleanCount = 2;

// 场景结构体定义
// 定义场景结构体，包含环境光、多个光源、多个球体、多个平面、多个圆柱体等。
struct Scene {
    vec3 ambient;
    PointLight[lightCount] lights;
    Sphere[sphereCount] spheres;
    Plane[planeCount] planes;
    Cylinder[cylinderCount] cylinders;
    Boolean[booleanCount] booleans;
};
```

```
// 射线结构体
// 定义射线结构体，包含射线的起点和方向。
struct Ray {
    vec3 origin;
    vec3 direction;
};

// 射线与物体相交的信息结构体
// 定义射线与物体相交的信息结构体，包含是否发生碰撞、碰撞点到射线起点的距离、碰撞点的位置、法向量、材质等信息。
// Contains all information pertaining to a ray/object intersection
struct HitInfo {
    bool hit;
    float t;
    vec3 position;
    vec3 normal;
    Material material;
    bool enteringPrimitive;
};

// 创建一个空的HitInfo结构体
// 射线追踪算法需要计算射线与多个物体相交的距离。这些距离可以用参数 t 来表示。通过比较这些 t 值，可以找到最近的交点，也就是射线首先击中的物体。
HitInfo getEmptyHit() {
    return HitInfo(
        false,  // 没有击中
        0.0,    // 距离为0
        vec3(0.0),  // 位置在原点
        vec3(0.0),  // 法线为零向量
        Material(vec3(0.0), vec3(0.0), 0.0, 0.0, 0.0, 0.0),  // 空材质
        false); // 不进入物体
}

// 对两个t值进行排序，确保t1小于t2
// Sorts the two t values such that t1 is smaller than t2
void sortT(inout float t1, inout float t2) {
    // Make t1 the smaller t
    if(t2 < t1)  {
        float temp = t1;
        t1 = t2;
        t2 = temp;
    }
}

// 检查t是否在给定的区间内
// Tests if t is in an interval
bool isTInInterval(const float t, const float tMin, const float tMax) {
    return t > tMin && t < tMax;
}

// 获取区间内最小的t值
// 给定两个浮点数 t0 和 t1，以及一个区间 [tMin, tMax]，找出这两个数中哪个在区间内，并将其赋值给 smallestTInInterval。如果两个数都在区间内，就取较小的那个。函数返回一个布尔值，表示是否找到了在区间内的值。
// Get the smallest t in an interval.
```

```
bool getSmallestTInInterval(float t0, float t1, const float tMin, const float
tMax, inout float smallestTInInterval) {

    sortT(t0, t1);

    // As t0 is smaller, test this first
    if(isTInInterval(t0, tMin, tMax)) {
        smallestTInInterval = t0;
        return true;
    }

    // If t0 was not in the interval, still t1 could be
    if(isTInInterval(t1, tMin, tMax)) {
        smallestTInInterval = t1;
        return true;
    }

    // None was
    return false;
}

// 计算射线与球体的相交
HitInfo intersectSphere(const Ray ray, const Sphere sphere, const float tMin,
const float tMax) {
    // 定义一个函数 intersectSphere，用于计算射线与球体的交点。
    // 计算射线起点到球心之间的向量
    vec3 to_sphere = ray.origin - sphere.position;

    // 计算射线与球体的相交方程的系数 a, b, c，以及判别式 D。
    float a = dot(ray.direction, ray.direction);
    float b = 2.0 * dot(ray.direction, to_sphere);
    float c = dot(to_sphere, to_sphere) - sphere.radius * sphere.radius;
    float D = b * b - 4.0 * a * c;
    // 如果判别式 D 大于 0，则说明射线与球体相交，计算两个交点 t0 和 t1。
    if (D > 0.0)
    {
        float t0 = (-b - sqrt(D)) / (2.0 * a);
        float t1 = (-b + sqrt(D)) / (2.0 * a);

        // 调用 getSmallestTInInterval 函数，找到在给定区间 [tMin, tMax] 内的最小交点
smallestTInInterval。如果找不到，则返回空交点信息。
        float smallestTInInterval;
        if(!getSmallestTInInterval(t0, t1, tMin, tMax, smallestTInInterval)) {
          return getEmptyHit();
        }

        // 根据最小交点 smallestTInInterval 计算交点的位置。
        vec3 hitPosition = ray.origin + smallestTInInterval * ray.direction;


        // 计算交点处的法向量和进入标志，并返回一个 HitInfo 结构体，表示射线与球体的交点信息。
        //Checking if we're inside the sphere by checking if the ray's origin is
inside. If we are, then the normal
        //at the intersection surface points towards the center. Otherwise, if we
are outside the sphere, then the normal
```

```
        //at the intersection surface points outwards from the sphere's center.
This is important for refraction.
        vec3 normal =
            length(ray.origin - sphere.position) < sphere.radius + 0.001?
            -normalize(hitPosition - sphere.position):
            normalize(hitPosition - sphere.position);

        //Checking if we're inside the sphere by checking if the ray's origin is
inside,
        // but this time for IOR bookkeeping.
        //If we are inside, set a flag to say we're leaving. If we are outside,
set the flag to say we're entering.
        //This is also important for refraction.
        bool enteringPrimitive =
            length(ray.origin - sphere.position) < sphere.radius + 0.001 ?
            false:
            true;

        return HitInfo(
            true,
            smallestTInInterval,
            hitPosition,
            normal,
            sphere.material,
            enteringPrimitive);
    }
    return getEmptyHit();
}

// 实验1
// 1a. 计算射线与平面的相交
HitInfo intersectPlane(const Ray ray,const Plane plane, const float tMin, const
float tMax) {
#ifdef SOLUTION_CYLINDER_AND_PLANE
    // 计算射线方向与平面法线的点积
    float denom = dot(ray.direction, plane.normal);

    // 如果点积接近零，射线平行于平面，无相交
    if (abs(denom) < 0.0001) {
        return getEmptyHit();
    }

    // 计算从射线原点到平面的距离
    float t = -(dot(ray.origin, plane.normal) - plane.d) / denom;

    // 检查交点是否在有效范围内
    if (t < tMin || t > tMax) {
        return getEmptyHit();
    }

    // 计算交点位置
    vec3 hitPosition = ray.origin + t * ray.direction;

    // 确定法线方向（始终指向射线来源）
    vec3 normal;
    if (denom < 0.0) {
```

```
        normal = plane.normal;
    } else {
        normal = -plane.normal;
    }

    // 返回相交信息
    return HitInfo(
        true,            // 发生相交
        t,               // 相交距离
        hitPosition,     // 相交位置
        normal,          // 相交点法线
        plane.material,  // 平面材质
        denom < 0.0      // 是否进入物体（对平面来说总是true）
    );
#endif
        return getEmptyHit();
}

float lengthSquared(vec3 x) {
    return dot(x, x);
}

// 1b. 计算射线与圆柱体的相交
HitInfo intersectCylinder(const Ray ray, const Cylinder cylinder, const float
tMin, const float tMax) {
#ifdef SOLUTION_CYLINDER_AND_PLANE
    // 将射线变换到圆柱体的局部空间
    vec3 ro = ray.origin - cylinder.position;
    vec3 rd = ray.direction;
    vec3 ca = cylinder.direction;

    // 计算二次方程系数
    float a = dot(rd, rd) - pow(dot(rd, ca), 2.0);
    float b = 2.0 * (dot(ro, rd) - dot(ro, ca) * dot(rd, ca));
    float c = dot(ro, ro) - pow(dot(ro, ca), 2.0) - cylinder.radius *
cylinder.radius;

    // 计算判别式
    float discriminant = b * b - 4.0 * a * c;

    if (discriminant < 0.0) {
        return getEmptyHit(); // 无相交
    }

    // 计算相交点
    float t1 = (-b - sqrt(discriminant)) / (2.0 * a);
    float t2 = (-b + sqrt(discriminant)) / (2.0 * a);

    // 确保 t1 < t2
    if (t1 > t2) {
        float temp = t1;
        t1 = t2;
        t2 = temp;
    }

    float t;
```

```
    bool entering;

    if (t1 >= tMin && t1 <= tMax) {
        t = t1;
        entering = true;
    } else if (t2 >= tMin && t2 <= tMax) {
        t = t2;
        entering = false;
    } else {
        return getEmptyHit(); // 相交点不在有效范围内
    }

    // 计算相交点位置
    vec3 hitPosition = ray.origin + t * ray.direction;

    // 计算相交点法线
    vec3 normal = normalize(hitPosition - cylinder.position - ca *
dot(hitPosition - cylinder.position, ca));

    if (!entering) {
        normal = -normal;
    }

    // 返回相交信息
    return HitInfo(
        true,                  // 发生相交
        t,                     // 相交距离
        hitPosition,           // 相交位置
        normal,                // 相交点法线
        cylinder.material,     // 圆柱体材质
        entering               // 是否进入物体
    );
#endif
    return getEmptyHit();
}

// 1c. 相交计算的原理
/*
平面与射线相交的原理：
1. 射线方程：P(t) = O + tD
   其中 O 是射线起点，D 是射线方向，t 是参数

2. 平面方程：dot(N, P) - d = 0
   其中 N 是平面法向量，P 是平面上任意点，d 是平面到原点的有符号距离

3. 求解步骤：
   - 将射线方程代入平面方程：dot(N, O + tD) - d = 0
   - 展开：dot(N, O) + t * dot(N, D) - d = 0
   - 求解 t：t = (d - dot(N, O)) / dot(N, D)
   - 当 dot(N, D) 接近0时，射线平行于平面，无交点
   - 检查 t 是否在有效范围内 [tMin, tMax]

圆柱体与射线相交的原理：
1. 圆柱体可以描述为：到中心轴的距离等于半径r的点的集合
   - 中心轴由点 P 和方向向量 D 定义
```

```
2. 求解步骤:
   - 将问题转换到圆柱体的局部坐标系
   - 射线到轴线的距离可以用向量叉乘和点乘表示
   - 构建二次方程: at² + bt + c = 0
     其中:
     a = dot(rd, rd) - pow(dot(rd, ca), 2)
     b = 2(dot(ro, rd) - dot(ro, ca)dot(rd, ca))
     c = dot(ro, ro) - pow(dot(ro, ca), 2) - r²
     rd 是射线方向, ro 是射线原点(相对圆柱体位置), ca 是圆柱体轴向
   - 求解二次方程得到交点参数 t
   - 检查 t 是否在有效范围内

注意事项:
1. 所有交点计算都需要考虑数值精度问题,使用小偏移量避免自相交
2. 需要正确计算交点处的法向量,这对后续的光照计算很重要
3. 对于圆柱体,需要特别注意轴向向量的归一化
4. 相交测试要考虑物体内部和外部的情况,这对折射计算很重要
*/

// 检查点是否在球体内部
bool inside(const vec3 position, const Sphere sphere) {
    return length(position - sphere.position) < sphere.radius;
}

// 实验5

// 5a. "与"运算
// 5b. "减"运算
// 5c. 解释如何改进HitInfo结构以更好地支持布尔运算
/*
改进建议:添加多重交点信息:
```glsl
struct HitInfo {
    bool hit;
    float t;                    // 第一个交点的距离
    float t_exit;              // 出射点距离(对于透明物体和布尔运算很重要)
    vec3 position;             // 入射点位置
    vec3 exit_position;        // 出射点位置
    vec3 normal;
    Material material;
    bool enteringPrimitive;
    int objectID;              // 标识与哪个物体相交
    int operationType;         // 布尔运算类型(AND、MINUS等)
};
```

改进理由:
1. 多重交点跟踪:
   - 布尔运算经常需要同时知道射线与物体的入射点和出射点
   - 当前结构只能记录单个交点,导致需要多次射线追踪计算
   - 添加t_exit和exit_position可以一次性存储完整的相交信息

2. 物体身份识别:
   - 添加objectID可以跟踪射线究竟与哪个物体相交
   - 在复杂的布尔运算中,这对于确定使用哪个物体的材质和属性很重要
   - 有助于处理多层嵌套的布尔运算
```

```
3. 运算类型记录：
   - operationType字段可以记录当前交点涉及的布尔运算类型
   - 这对于处理复杂的布尔运算组合很有帮助
   - 可以更容易地实现嵌套的布尔运算

这些改进的好处：
1. 性能提升：
   - 减少重复的相交计算
   - 避免多次射线追踪
   - 更高效地处理复杂的布尔运算场景

2. 更好的健壮性：
   - 更容易处理边界情况
   - 减少数值精度问题
   - 提供更完整的相交信息

3. 扩展性：
   - 更容易支持复杂的布尔运算组合
   - 为未来添加新的布尔运算类型提供基础
   - 便于实现更高级的渲染效果
*/

HitInfo intersectBoolean(const Ray ray, const Boolean boolean, const float tMin,
const float tMax) {
#ifdef SOLUTION_BOOLEAN
    // 获取射线与两个球体的相交信息
    // Get intersection information for both spheres
    HitInfo hitA = intersectSphere(ray, boolean.spheres[0], tMin, tMax);
    HitInfo hitB = intersectSphere(ray, boolean.spheres[1], tMin, tMax);

    if (boolean.mode == BOOLEAN_MODE_AND) {
        // 处理交集运算 / Handle intersection operation
        if (!hitA.hit || !hitB.hit) {
            return getEmptyHit();
        }

        // 取较远的入射点作为实际交点
        float t = max(hitA.t, hitB.t);

        // 检查是否在有效范围内
        if (!isTInInterval(t, tMin, tMax)) {
            return getEmptyHit();
        }

        // 使用距离较远的那个球体的表面信息
        vec3 hitPoint = ray.origin + ray.direction * t;
        if (abs(t - hitA.t) < 0.001) {
            return HitInfo(true, t, hitPoint, hitA.normal, hitA.material, true);
        } else {
            return HitInfo(true, t, hitPoint, hitB.normal, hitB.material, true);
        }
    }

    else if (boolean.mode == BOOLEAN_MODE_MINUS) {
        // 处理差集运算 B-A / Handle subtraction operation B-A
```

```
        if (!hitB.hit) {
            return getEmptyHit();
        }

        bool startInA = inside(ray.origin, boolean.spheres[0]);
        vec3 hitBPos = ray.origin + ray.direction * hitB.t;
        bool hitBInA = inside(hitBPos, boolean.spheres[0]);

        // 如果起点在A内且射线和A有交点
        if (startInA && hitA.hit) {
            // 先要从A出来
            vec3 exitAPos = ray.origin + ray.direction * hitA.t;
            // 确保出射点在B内部
            if (inside(exitAPos, boolean.spheres[1])) {
                return HitInfo(
                    true,
                    hitA.t,
                    exitAPos,
                    -hitA.normal,  // 从A内部出来，法线朝外
                    boolean.spheres[1].material,
                    true
                );
            }
            return getEmptyHit();
        }

        // 如果射线与B的交点在A外部，保留B的交点
        if (!hitBInA) {
            if (hitA.hit && hitA.t < hitB.t) {
                // 如果先和A相交，这个交点需要在B内部
                vec3 hitAPos = ray.origin + ray.direction * hitA.t;
                if (inside(hitAPos, boolean.spheres[1])) {
                    // 使用A的法线表示切割面
                    return HitInfo(
                        true,
                        hitA.t,
                        hitAPos,
                        hitA.normal,
                        boolean.spheres[1].material,  // 使用B的材质
                        true
                    );
                }
            }
            // 直接返回B的交点
            return hitB;
        }

        // 如果B的交点在A内，需要找到从A出来的点
        if (hitA.hit) {
            Ray exitRay = Ray(hitBPos + ray.direction * 0.001, ray.direction);
            HitInfo exitA = intersectSphere(exitRay, boolean.spheres[0], 0.001,
tMax);
            if (exitA.hit && inside(exitRay.origin + exitRay.direction * exitA.t,
boolean.spheres[1])) {
                vec3 exitPos = exitRay.origin + exitRay.direction * exitA.t;
                return HitInfo(
```

```
                    true,
                    hitB.t + exitA.t,
                    exitPos,
                    exitA.normal,
                    boolean.spheres[1].material,
                    true
                );
            }
        }

        return getEmptyHit();
    }
#endif
    return getEmptyHit();
}

// 时间变量（用于动画效果）
uniform float time;

// 比较两个HitInfo，返回更近的一个
HitInfo getBetterHitInfo(const HitInfo oldHitInfo, const HitInfo newHitInfo) {
    if(newHitInfo.hit)
        if(newHitInfo.t < oldHitInfo.t)  // No need to test for the interval,
this has to be done per-primitive
          return newHitInfo;
    return oldHitInfo;
}

// 计算射线与整个场景的相交
HitInfo intersectScene(const Scene scene, const Ray ray, const float tMin, const
float tMax) {
    HitInfo bestHitInfo;
    bestHitInfo.t = tMax;
    bestHitInfo.hit = false;

    // ... (遍历所有物体并找到最近的交点)
    for (int i = 0; i < booleanCount; ++i) {
        bestHitInfo = getBetterHitInfo(bestHitInfo, intersectBoolean(ray,
scene.booleans[i], tMin, tMax));
    }

    for (int i = 0; i < planeCount; ++i) {
        bestHitInfo = getBetterHitInfo(bestHitInfo, intersectPlane(ray,
scene.planes[i], tMin, tMax));
    }
    for (int i = 0; i < sphereCount; ++i) {
        bestHitInfo = getBetterHitInfo(bestHitInfo, intersectSphere(ray,
scene.spheres[i], tMin, tMax));
    }
    for (int i = 0; i < cylinderCount; ++i) {
        bestHitInfo = getBetterHitInfo(bestHitInfo, intersectCylinder(ray,
scene.cylinders[i], tMin, tMax));
    }

    return bestHitInfo;
}
```

```
// 实验2
// 2. 在着色中添加阴影测试

// 计算来自单个光源的着色
vec3 shadeFromLight(
  const Scene scene,
  const Ray ray,
  const HitInfo hit_info,
  const PointLight light)
{
    // ... (计算漫反射和镜面反射)
  vec3 hitToLight = light.position - hit_info.position;

  vec3 lightDirection = normalize(hitToLight);
  vec3 viewDirection = normalize(hit_info.position - ray.origin);
  vec3 reflectedDirection = reflect(viewDirection, hit_info.normal);
  float diffuse_term = max(0.0, dot(lightDirection, hit_info.normal));
  float specular_term  = pow(max(0.0, dot(lightDirection, reflectedDirection)),
hit_info.material.glossiness);

#ifdef SOLUTION_SHADOW
    // 创建从交点到光源的阴影射线
    Ray shadowRay;
    shadowRay.origin = hit_info.position + hit_info.normal * 0.001; // 避免自遮挡
    shadowRay.direction = normalize(hitToLight);

    // 检查射线到光源的距离内是否有遮挡物
    float distanceToLight = length(hitToLight);
    HitInfo shadowHit = intersectScene(scene, shadowRay, 0.001, distanceToLight -
0.001);

    // 如果有物体遮挡，则该点在阴影中
    float visibility = shadowHit.hit ? 0.0 : 1.0;
#else
    float visibility = 1.0;
#endif
  return    visibility *
            light.color * (
            specular_term * hit_info.material.specular +
            diffuse_term * hit_info.material.diffuse);
}

// 计算背景颜色
vec3 background(const Ray ray) {
  // A simple implicit sky that can be used for the background
  return vec3(0.2) + vec3(0.8, 0.6, 0.5) * max(0.0, ray.direction.y);
}

// 计算整体着色
// It seems to be a WebGL issue that the third parameter needs to be inout instea
dof const on Tobias' machine
vec3 shade(const Scene scene, const Ray ray, inout HitInfo hitInfo) {
    // ... (综合环境光和所有光源的贡献)
    if(!hitInfo.hit) {
```

```
        return background(ray);
    }

    vec3 shading = scene.ambient * hitInfo.material.diffuse;
    for (int i = 0; i < lightCount; ++i) {
        shading += shadeFromLight(scene, ray, hitInfo, scene.lights[i]);
    }
    return shading;
}


// 根据像素坐标生成射线
Ray getFragCoordRay(const vec2 frag_coord) {
    float sensorDistance = 1.0;
    vec2 sensorMin = vec2(-1, -0.5);
    vec2 sensorMax = vec2(1, 0.5);
    vec2 pixelSize = (sensorMax- sensorMin) / vec2(viewport.x, viewport.y);
    vec3 origin = vec3(0, 0, sensorDistance);
    vec3 direction = normalize(vec3(sensorMin + pixelSize * frag_coord, -
sensorDistance));
    // ... (计算射线原点和方向)
    return Ray(origin, direction);
}

//实验4

// 4a. 研究并说明如何计算反射和折射的权重
/*
在光学中，当光线从一种介质进入另一种介质时，会同时发生反射和折射。反射和折射的强度比例（权重）可以
通过以下几种方法计算：

1. 点积（Cosine）方法：
这是最简单的近似方法，基本原理是反射强度与入射角余弦成反比
计算方法：R = 1 - |dot(V, N)|，
其中：V 是视线方向向量（归一化），N 是表面法线向量（归一化），R 是反射比例，1-R 则为折射比例

优点：计算简单，效果还不错
缺点：物理准确性不高


2. Schlick近似：
这是对Fresnel方程的一个实用近似
计算方法：R = R0 + (1 - R0)(1 - cos θ)^5，
其中：R0 = ((n1 - n2)/(n1 + n2))²，n1，n2 是两种介质的折射率，θ 是入射角

优点：比点积方法更准确，计算量适中
缺点：仍是近似，在某些极端情况下可能不够准确


3. 完整Fresnel方程：
这是最准确的物理方法，但计算复杂，在实时渲染中较少使用，需要考虑光的偏振状态
*/

// 4b. 计算菲涅耳效应
float fresnel(const vec3 viewDirection, const vec3 normal, const float sourceIOR,
const float destIOR) {
```

```
#ifdef SOLUTION_FRESNEL
    // 方法1：点积方法（与参考图片一致）
    vec3 V = normalize(-viewDirection);
    vec3 N = normalize(normal);
    float cosTheta = abs(dot(V, N));
    return 1.0 - cosTheta;

    /*
    // 方法2：Schlick近似方法
    vec3 V = normalize(-viewDirection);
    vec3 N = normalize(normal);
    float cosTheta = abs(dot(V, N));

    // 计算基础反射率R0
    float r0 = pow((sourceIOR - destIOR) / (sourceIOR + destIOR), 2.0);

    // Schlick近似公式
    return r0 + (1.0 - r0) * pow(1.0 - cosTheta, 5.0);
    */
#else
    return 1.0;
#endif
}

// 实验3

// 为每个像素计算颜色
vec3 colorForFragment(const Scene scene, const vec2 fragCoord) {
    // ... (主要的渲染逻辑，包括反射和折射的计算)
    Ray initialRay = getFragCoordRay(fragCoord);
    HitInfo initialHitInfo = intersectScene(scene, initialRay, 0.001, 10000.0);
    vec3 result = shade(scene, initialRay, initialHitInfo);

    Ray currentRay;
    HitInfo currentHitInfo;

    // Compute the reflection
    currentRay = initialRay;
    currentHitInfo = initialHitInfo;

    // The initial strength of the reflection
    float reflectionWeight = 1.0;

    // The initial medium is air
    float currentIOR = 1.0;

    float sourceIOR = 1.0;
    float destIOR = 1.0;

    // 3a. 反射循环 (12分)
    const int maxReflectionStepCount = 2;
    for(int i = 0; i < maxReflectionStepCount; i++) {

      if(!currentHitInfo.hit) break;

      // 反射权重计算
```

```
#ifdef SOLUTION_REFLECTION_REFRACTION
        // 检查材质是否有反射特性
        if(currentHitInfo.material.reflection > 0.0) {
            reflectionWeight *= currentHitInfo.material.reflection;
        } else {
            break;  // 如果没有反射,直接退出循环
        }
#else
        reflectionWeight *= 0.5;
#endif

        //  菲涅耳效应
#ifdef SOLUTION_FRESNEL
        float fresnelFactor = fresnel(normalize(currentRay.direction),
                                      currentHitInfo.normal,
                                      sourceIOR,
                                      destIOR);
        reflectionWeight *= fresnelFactor;
#else
        reflectionWeight *= 0.5;
#endif

      Ray nextRay;

      // 计算反射射线
#ifdef SOLUTION_REFLECTION_REFRACTION
        // 计算反射射线
        nextRay.origin = currentHitInfo.position + currentHitInfo.normal * 0.001;
 // 略微偏移以避免自相交
        nextRay.direction = reflect(currentRay.direction, currentHitInfo.normal);
#endif
      currentRay = nextRay;

      currentHitInfo = intersectScene(scene, currentRay, 0.001, 10000.0);

      result += reflectionWeight * shade(scene, currentRay, currentHitInfo);
    }

    // Compute the refraction
    currentRay = initialRay;
    currentHitInfo = initialHitInfo;

    // The initial strength of the refraction.
    float refractionWeight = 1.0;

    // 3b. 折射循环(12分)

    const int maxRefractionStepCount = 2;
    for(int i = 0; i < maxRefractionStepCount; i++) {

    //  折射权重计算
#ifdef SOLUTION_REFLECTION_REFRACTION
        if(currentHitInfo.material.refraction > 0.0) {
            refractionWeight *= currentHitInfo.material.refraction;
        } else {
            break;  // 如果没有折射,直接退出循环
```

```
        }
#else
        refractionWeight *= 0.5;
#endif

    //  菲涅耳效应对折射的影响
#ifdef SOLUTION_FRESNEL
        float fresnelFactor = fresnel(normalize(currentRay.direction),
                                     currentHitInfo.normal,
                                     sourceIOR,
                                     destIOR);
        refractionWeight *= (1.0 - fresnelFactor);
#endif

      Ray nextRay;

    //  计算折射射线
#ifdef SOLUTION_REFLECTION_REFRACTION
        // 确定折射率
        if(currentHitInfo.enteringPrimitive) {
            sourceIOR = 1.0;  // 空气的折射率
            destIOR = currentHitInfo.material.ior;
            currentIOR = destIOR;
        } else {
            sourceIOR = currentIOR;
            destIOR = 1.0;  // 回到空气中
            currentIOR = destIOR;
        }

        // 计算折射方向
        vec3 n = currentHitInfo.normal;
        float eta = sourceIOR / destIOR;
        float c1 = -dot(currentRay.direction, n);
        float c2 = sqrt(1.0 - eta * eta * (1.0 - c1 * c1));

        nextRay.direction = normalize(eta * currentRay.direction + (eta * c1 -
c2) * n);
        nextRay.origin = currentHitInfo.position - n * 0.001;  // 略微偏移以避免自相
交
        currentRay = nextRay;
#endif

        currentHitInfo = intersectScene(scene, currentRay, 0.001, 10000.0);
        result += refractionWeight * shade(scene, currentRay, currentHitInfo);

        if(!currentHitInfo.hit) break;
    }
    return result;
}

// 定义常用材质
// Common materials
// IOR data comes from https://refractiveindex.info/, otherwise zero
// When reflection and refraction set to zero, it is turned off

Material getDefaultMaterial() {
```

```
  return Material(vec3(0.3), vec3(0), 0.0, 0.0, 0.0, 0.0);
}

Material getPaperMaterial() {
  return Material(vec3(0.7, 0.7, 0.7), vec3(0, 0, 0), 5.0, 0.0, 0.0, 0.0);
}

Material getPlasticMaterial() {
    return Material(vec3(0.9, 0.3, 0.1), vec3(1.0), 10.0, 0.9, 0.0, 1.5);
}

Material getGlassMaterial() {
    return Material(vec3(0.0), vec3(0.0), 5.0, 1.0, 1.0, 1.5);
}

Material getSteelMirrorMaterial() {
    return Material(vec3(0.1), vec3(0.3), 20.0, 0.8, 0.0, 2.9);
}

Material getMetaMaterial() {
    return Material(vec3(0.1, 0.2, 0.5), vec3(0.3, 0.7, 0.9), 20.0, 0.8, 0.0,
0.0);
}

// 色调映射函数
vec3 tonemap(const vec3 radiance) {
  const float monitorGamma = 2.0;
  return pow(radiance, vec3(1.0 / monitorGamma));
}


void main() {
    // 设置场景
    // Setup scene
    Scene scene;
    // 创建一个场景对象，用于存储场景中的各种元素，如光源、物体等。
    scene.ambient = vec3(0.12, 0.15, 0.2);
    // 设置场景的环境光强度，用于模拟全局照明。
    scene.lights[0].position = vec3(5, 15, -5);
    scene.lights[0].color    = 0.5 * vec3(0.9, 0.5, 0.1);

    // 设置第一个光源的位置和颜色。
    scene.lights[1].position = vec3(-15, 5, 2);
    scene.lights[1].color    = 0.5 * vec3(0.1, 0.3, 1.0);

    // ... (设置光源、物体等)
    // Primitives
    bool specialScene = false;

    // Set specialScene to true to implement the task in the below ifdef block
#ifdef SOLUTION_BOOLEAN
    specialScene = true; // 启用特殊场景
#endif

    if (specialScene) {
        // 设置第一个布尔运算的模式为减法。
```

```
        // Boolean scene
        scene.booleans[0].mode = BOOLEAN_MODE_MINUS;

        // 设置第一个布尔运算中的第一个球体的属性。
        // sphere A
        scene.booleans[0].spheres[0].position       = vec3(3, 0, -10);
        scene.booleans[0].spheres[0].radius         = 2.75;
        scene.booleans[0].spheres[0].material       = getPaperMaterial();

        // 设置第一个布尔运算中的第二个球体的属性。
        // sphere B
        scene.booleans[0].spheres[1].position       = vec3(6, 1, -13);
        scene.booleans[0].spheres[1].radius         = 4.0;
        scene.booleans[0].spheres[1].material       = getPaperMaterial();

        // 设置第二个布尔运算的模式为与运算。
        scene.booleans[1].mode = BOOLEAN_MODE_AND;

        // 设置第二个布尔运算中的第一个球体的属性
        scene.booleans[1].spheres[0].position       = vec3(-3.0, 1, -12);
        scene.booleans[1].spheres[0].radius         = 4.0;
        scene.booleans[1].spheres[0].material       = getPaperMaterial();

        // 设置第二个布尔运算中的第二个球体的属性
        scene.booleans[1].spheres[1].position       = vec3(-6.0, 1, -12);
        scene.booleans[1].spheres[1].radius         = 4.0;
        scene.booleans[1].spheres[1].material       = getMetaMaterial();

        // 设置第一个平面的属性
        scene.planes[0].normal                  = normalize(vec3(0, 0.8, 0));
        scene.planes[0].d                       = -4.5;
        scene.planes[0].material                = getSteelMirrorMaterial();

        // 设置第一个光源的属性
        scene.lights[0].position = vec3(-5, 25, -5);
        scene.lights[0].color    = vec3(0.9, 0.5, 0.1);

        // 设置第二个光源的属性
        scene.lights[1].position = vec3(-15, 5, 2);
        scene.lights[1].color    = 0.0 * 0.5 * vec3(0.1, 0.3, 1.0);
    }
    else {// 根据 specialScene 的值选择不同的场景配置。
        // normal scene

        // 设置第一个球体的属性
        scene.spheres[0].position               = vec3(10, -5, -16);
        scene.spheres[0].radius                 = 6.0;
        scene.spheres[0].material               = getPaperMaterial();

        // 设置第二个球体的属性
        scene.spheres[1].position               = vec3(-7, -2, -13);
        scene.spheres[1].radius                 = 4.0;
        scene.spheres[1].material               = getPlasticMaterial();

        // 设置第三个球体的属性
        scene.spheres[2].position               = vec3(0, 0.5, -5);
```

```
        scene.spheres[2].radius                      = 2.0;
        scene.spheres[2].material                    = getGlassMaterial();

        // 设置第一个平面的属性
        scene.planes[0].normal                       = normalize(vec3(0, 1.0, 0));
        scene.planes[0].d                            = -4.5;
        scene.planes[0].material                     = getSteelMirrorMaterial();

        // 设置第一个圆柱体的属性
        scene.cylinders[0].position                  = vec3(-1, 1, -26);
        scene.cylinders[0].direction                 = normalize(vec3(-2, 2, -1));
        scene.cylinders[0].radius                    = 1.5;
        scene.cylinders[0].material                  = getPaperMaterial();

        // 设置第二个圆柱体的属性
        scene.cylinders[1].position                  = vec3(4, 1, -5);
        scene.cylinders[1].direction                 = normalize(vec3(1, 4, 1));
        scene.cylinders[1].radius                    = 0.4;
        scene.cylinders[1].material                  = getPlasticMaterial();
    }

    // 计算当前像素的最终颜色，并将其写入输出颜色缓冲区。
    // Compute color for fragment
    gl_FragColor.rgb = tonemap(colorForFragment(scene, gl_FragCoord.xy));
    gl_FragColor.a = 1.0;

}
```