1 全局光照

1.1 光线投射与光线追踪

1.1.1 光线投射

大部分环境中的光都不能被镜头所捕捉, 因此相比于考虑每个光源发出的每条光线, 我们不妨采取逆向思维, 考虑那些最终到达镜头的光线. 由于光路是可逆的, 因此我们从镜头向屏幕上的点连一条线, 该射线在场景中击中的第一个物体将决定该点的颜色.

定义 1.1 光线投射 光线投射 (Ray Casting) 是一种通过从观察点向场景中投射射线来确定可见表面和颜色的技术.

因此, 光线投射算法的基本流程如下:

1. 对于屏幕的每个像素点 (x, y), 从相机位置向场景中投射一条穿过 (x, y) 的射线 $\boldsymbol{r}(t) = \boldsymbol{o} + t\boldsymbol{d}$.
2. 计算射线 $\boldsymbol{r}(t)$ 与场景中的物体第一次发生相交的位置 $\boldsymbol{p}$.
3. 将交点 $\boldsymbol{p}$ 与所有光源相连, 分别得到一根 shadow ray. 判断 shadow ray 在到达光源之前是否与其他物体相交, 如果相交则该光源对点 $\boldsymbol{p}$ 不可见, 否则可见.
4. 在可见的情况下, 我们已经知道了指向光源的方向 $\boldsymbol{l}$, 指向观察者的方向 $-\boldsymbol{d}$, 以及表面法线 $\boldsymbol{n}$. 使用在光照与着色中介绍的各种光照模型即可计算光源对点 $\boldsymbol{p}$ 贡献的颜色.
5. 将所有光源的贡献累加, 得到最终的像素颜色.

上述过程的伪代码可以表示如下:

```
for each pixel (x, y) do
    # Generate primary ray from camera through pixel
    ray = generateRay(camera, x, y)
    # Find intersection with scene.
    hitInfo = intersectScene(ray)
    if hitInfo.hit then
        color = vec3(0, 0, 0)
        # For each light source, check visibility and compute lighting.
        for each light in scene.lights do
            shadowRay = generateShadowRay(hitInfo.position, light.position)
            if not intersectScene(shadowRay).hit then
                color += computeLighting(hitInfo, light)
        setPixelColor(x, y, color)
    else
        setPixelColor(x, y, backgroundColor)
```

1.1.2 光线追踪

在实际情况下, 物体接受的光照可能并不直接来自于光源, 而可能经过其它物体的反射/折射再到达该物体上. 为了考虑这些间接光照, 我们需要引入光线追踪技术.

定义 1.2 光线追踪 光线追踪 (Ray Tracing) 通过模拟光线在场景中的传播路径, 包括反射和折射, 来生成更加逼真的图像.

光线追踪算法的基本流程如下:

1. 对于屏幕的每个像素点 (x, y), 从相机位置向场景中投射一条穿过 (x, y) 的射线 $\boldsymbol{r}(t) = \boldsymbol{o} + t\boldsymbol{d}$.
2. 计算射线 $\boldsymbol{r}(t)$ 与场景中的物体第一次发生相交的位置 $\boldsymbol{p}$.
3. 按照光线投射中介绍的办法计算交点 $\boldsymbol{p}$ 处直接来自于光源形成的颜色.
4. 根据材质属性, 生成反射射线和折射射线, 递归地追踪这些射线以计算间接光照形成的颜色.
5. 将局部光照和间接光照形成的颜色累加, 得到最终的像素颜色.

上述过程的递归形式的伪代码可以表示如下:

```
# Define recursively ray tracing function.
function traceRay(ray, depth):
    if depth > maxDepth then
        return backgroundColor
    hitInfo = intersectScene(ray)
    if hitInfo.hit then
        color = computeLocalLighting(hitInfo)
        # Compute reflection
        if hitInfo.material.reflective then
            reflectRay = generateReflectRay(hitInfo)
            color += hitInfo.material.reflectivity * traceRay(reflectRay, depth + 1)
        # Compute refraction
        if hitInfo.material.refractive then
            refractRay = generateRefractRay(hitInfo)
            color += hitInfo.material.transparency * traceRay(refractRay, depth + 1)
        return color
    else
        return backgroundColor
for each pixel (x, y) do
    ray = generateRay(camera, x, y)
    color = traceRay(ray, 0)
    setPixelColor(x, y, color)
```

光线追踪算法能够生成高度逼真的图像, 但计算开销较大, 因此在实际应用中通常会结合其他算法进行优化. 我们将在之后讲到优化的方式. 在此之前, 我们先来了解光线投射/追踪的基本步骤, 即光线求交.

1.2 光线求交

可以看出, 在光线投射与光线追踪算法中, 计算射线与场景中物体的交点是一个核心步骤. 高效地实现光线求交对于提升渲染性能至关重要. 下面我们将介绍几种常见的几何体与光线求交点的办法. 在本节的推导中, 我们假定光线的方程为

$$\boldsymbol{r}(t) = \boldsymbol{o} + t\boldsymbol{d}, \quad 0 \leqslant t < +\infty$$

其中 $\boldsymbol{o}$ 意为 Origin, 即光线的起点, 而 $\boldsymbol{d}$ 意为 Direction, 即光线的方向向量. 求得的 t 即与物体的交点到光线起点的距离.

1.2.1 光线与球面求交

考虑球心为 $\boldsymbol{c}$, 半径为 r 的球面, 其隐式方程为

$$||\boldsymbol{p} - \boldsymbol{c}||^2 = r^2$$

代入光线方程, 我们有

$$||\boldsymbol{o} + t\boldsymbol{d} - \boldsymbol{c}||^2 = r^2$$

展开后得到关于 t 的二次方程

$$t^2\boldsymbol{d}\cdot\boldsymbol{d} + 2t\boldsymbol{d}\cdot(\boldsymbol{o}-\boldsymbol{c}) + (\boldsymbol{o}-\boldsymbol{c})\cdot(\boldsymbol{o}-\boldsymbol{c}) - r^2 = 0$$

设 $a = \boldsymbol{d}\cdot\boldsymbol{d}$, $b = 2\boldsymbol{d}\cdot(\boldsymbol{o}-\boldsymbol{c})$, $c = (\boldsymbol{o}-\boldsymbol{c})\cdot(\boldsymbol{o}-\boldsymbol{c}) - r^2$, 则方程的解为

$$t = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

根据判别式 $b^2 - 4ac$ 的值, 我们可以判断光线与球面的交点情况是相离, 相切还是相交, 然后根据求根公式即可求出交点.

1.2.2 光线与长方体求交

为了简化光线与物体的求交计算, 我们通常对物体创建一个规则的几何外形将其包围. 最常见的包围体是**轴对齐包围盒 (Axis-Aligned Bounding Box, AABB)**.

定义 1.3 轴对齐包围盒 轴对齐包围盒是指其边界与坐标轴平行的长方体, 通常由最小点 $\boldsymbol{min} = (x_{min}, y_{min}, z_{min})$ 和最大点 $\boldsymbol{max} = (x_{max}, y_{max}, z_{max})$ 定义.

这里介绍一种高效的光线与 AABB 求交的方法, 即 **Slabs Method**. 不失一般性地, 我们考虑长方体中平行于 xy 平面的表面. 设表面所在的方程分别为 $z = z_{min}$ 和 $z = z_{max}$, 则光线与这两个平面的交点 t 值分别为

$$t_{z_{min}} = \frac{z_{min} - o_z}{d_z}, \quad t_{z_{max}} = \frac{z_{max} - o_z}{d_z}$$