XIAMEN UNIVERSITY

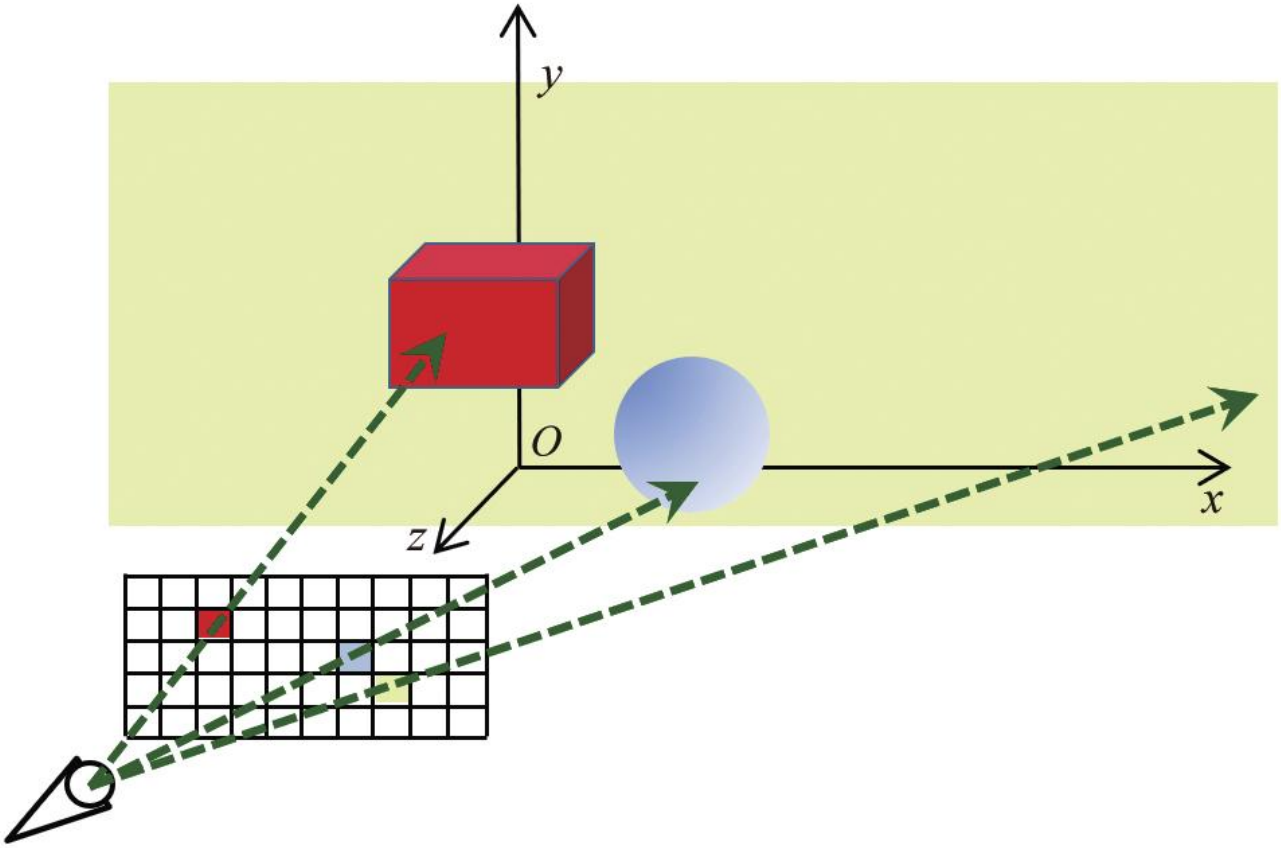
COMPUTER GRAPHICS

第十六章 光线追踪计算着色器

陈中贵
厦门大学信息学院
http://graphics.xmu.edu.cn

光线投射

- 1968 年 Arthur Appel 的文章，称为光线投射（ray casting）
- 1979 年 Foley和 Whitted 扩充了这个算法，加入了对每一道光线的递归式投射以模拟反射、阴影和折射[FW79]，并将这个过程叫作光线追踪（ray tracing）



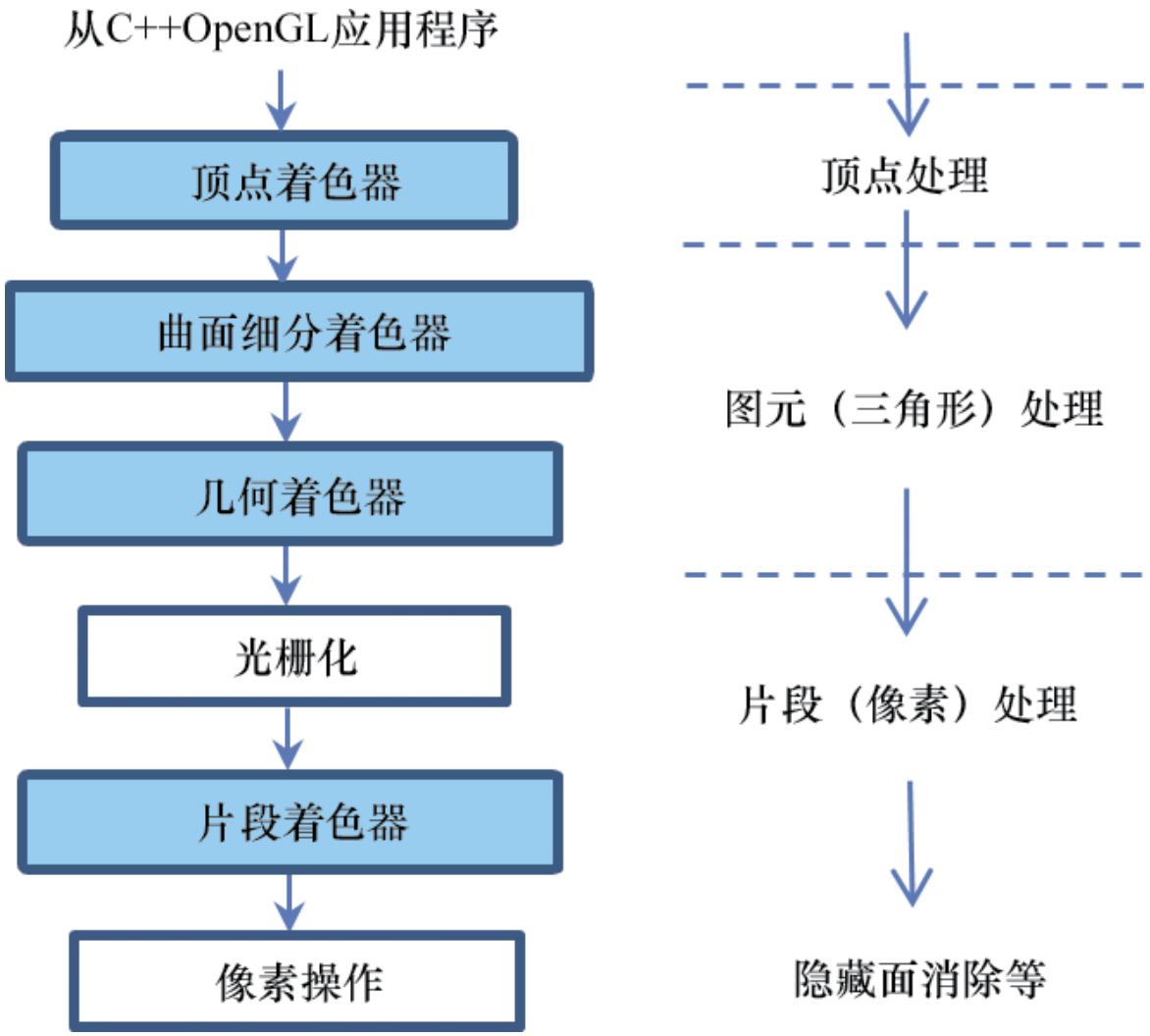
光栅化 vs 光线追踪

Rasterization vs Ray Tracing

- Objects defined w/ vertices
- Parallel processing:
  - vertices
  - triangles
  - rasterization
  - pixels

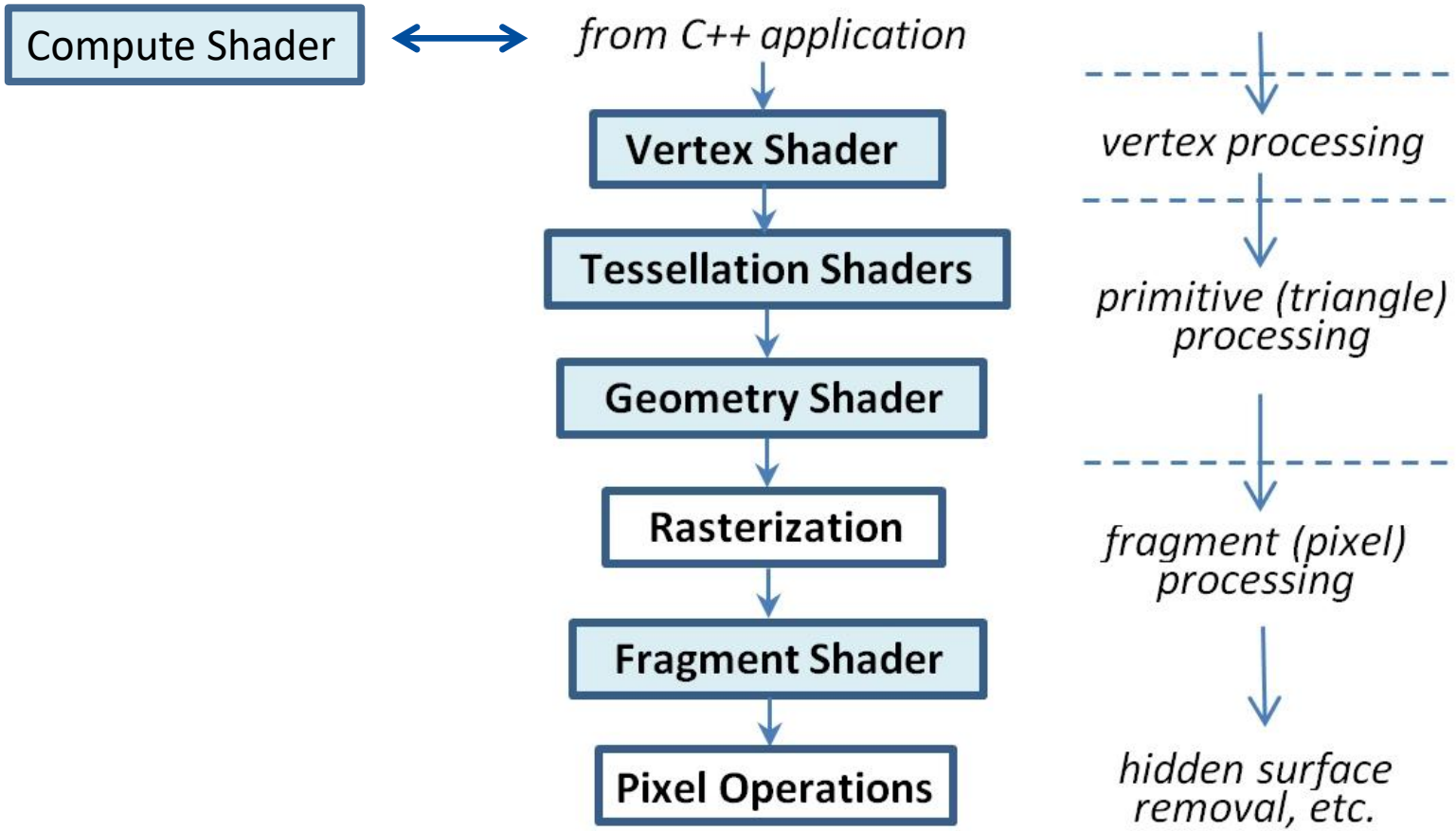
- Objects defined w/ math
- Parallel processing:
  - for each pixel:
    sequence of “rays”
  - compute collisions
    with the objects

OpenGL 图形管线概览



计算着色器

- 计算着色器与我们所见过的其他着色器类似，唯一的不同是它不作为图形管线的一部分，计算着色器独立运行



计算着色器示例

- 将两个一维矩阵相应元素相加

(shader side)

```
#version 430
buffer inputBuffer1 { int inVals1[ ]; };
buffer inputBuffer2 { int inVals2[ ]; };
buffer outputBuffer { int outVals[ ]; };
layout (local_size_x=1) in;
void main()
{   uint thisRun = gl_GlobalInvocationID.x;
    outVals[thisRun] = inVals1[thisRun] + inVals2[thisRun];
}
```

计算着色器示例

- 将两个一维矩阵相应元素相加

  (C++ side)

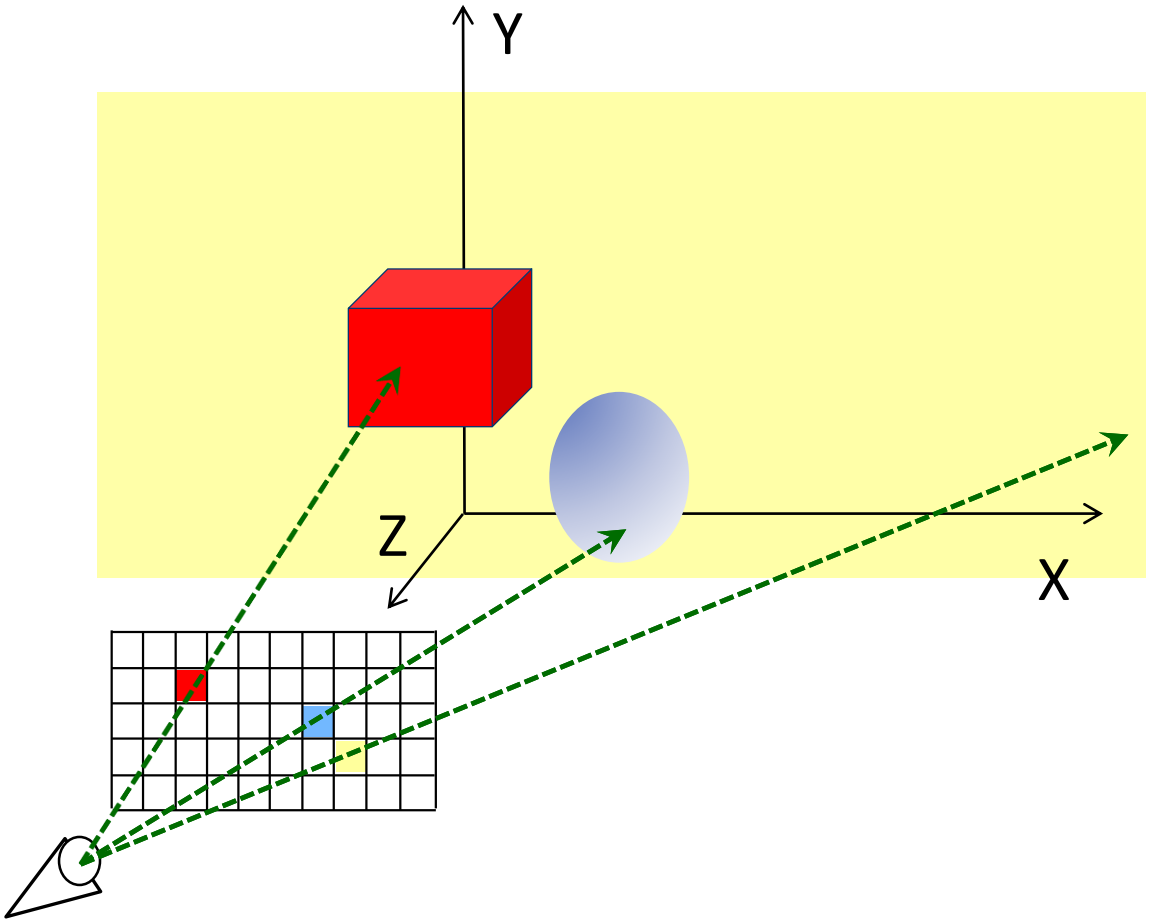
```
GLuint buffer[3];
GLuint computeShader;

int v1[ ] = { 10, 12, 16, 18, 50, 17};
int v2[ ] = { 30, 14, 80, 20, 51, 12 };
int res[6];

void init() {
    computeShader = Utils::createShaderProgram("simpleComputeShader.glsl");
    glGenBuffers(3, buffer);
    glBindBuffer(GL_SHADER_STORAGE_BUFFER, buffer[0]);
    glBufferData(GL_SHADER_STORAGE_BUFFER, sizeof(v1), v1, GL_STATIC_DRAW);
    glBindBuffer(GL_SHADER_STORAGE_BUFFER, buffer[1]);
    glBufferData(GL_SHADER_STORAGE_BUFFER, sizeof(v2), v2, GL_STATIC_DRAW);
    glBindBuffer(GL_SHADER_STORAGE_BUFFER, buffer[2]);
    glBufferData(GL_SHADER_STORAGE_BUFFER, sizeof(res), res, GL_STATIC_READ);
}
```

计算着色器示例

```
void computeSum() {
    glUseProgram(simpleComputeShader);
    glBindBufferBase(GL_SHADER_STORAGE_BUFFER, 0, buffer[0]);
    glBindBufferBase(GL_SHADER_STORAGE_BUFFER, 1, buffer[1]);
    glBindBufferBase(GL_SHADER_STORAGE_BUFFER, 2, buffer[2]);
    glDispatchCompute(6,1,1);
    glMemoryBarrier(GL_ALL_BARRIER_BITS);
    glBindBuffer(GL_SHADER_STORAGE_BUFFER, buffer[2]);
    glGetBufferSubData(GL_SHADER_STORAGE_BUFFER, 0, sizeof(res), res);
}
```

光线投射

- （1）实现光线追踪算法，将像素网格构建成图像；
  （2）渲染所得图像。



定义 2D 纹理图像

- 为纹理分配内存,并将每个像素都设置为粉红色所对应的颜色值(结果图像中残余的粉红色表明我们的实现中存在错误)
- 创建一个 OpenGL 纹理对象并将其与所分配的内存相关联
- 定义一个由两个三角形组成的矩形（或“四边形”），用于显示光线投射纹理图像
- 生成两个着色器程序:
  - 一个光线投射计算着色器程序;
  - 一个简单的着色器程序，用于在矩形上显示光线投射纹理图像

光线求交

- 光线与球面的交点
  给定射线的原点$\boldsymbol{r}_s$和方向$\boldsymbol{r}_d$，以及球心的位置$\boldsymbol{s}_p$和球半径s_r，
  t为射线原点到球面的距离
- 求解方程

$$\left(\boldsymbol{r}_{\mathrm{d}} \cdot \boldsymbol{r}_{\mathrm{d}}\right) t^2+\left(2 \boldsymbol{r}_{\mathrm{d}} \cdot\left(r_{\mathrm{s}}-s_{\mathrm{p}}\right)\right) t+\left(r_{\mathrm{s}}-s_{\mathrm{p}}\right)^2-s_{\mathrm{r}}{}^2=0$$

- 计算判别式

$$\Delta=\left(2\boldsymbol{r}_d \cdot\left|\boldsymbol{r}_s-\boldsymbol{s}_p\right|\right)^2-4\left|\boldsymbol{r}_d\right|^2\left(\left|\boldsymbol{r}_s-\boldsymbol{s}_p\right|^2-\boldsymbol{s}_r^2\right)$$

- 当Δ不小于 0 时，方程有两个解

$$t=\frac{\left(-2 r_d \bullet\left|\boldsymbol{r}_s-\boldsymbol{s}_p\right|\right) \pm \sqrt{\Delta}}{2\left|r_d\right|^2}$$

光线求交

- $\Delta > 0$时，t的两个值中较小和较大的分别表示为t_{near}和t_{far}
  - 当t_{near}和t_{far}都为负时，整个球体都在光线的背面，没有交点；
  - 当t_{near}为负且t_{far}为正时，光线从球体内部开始，第一个交点距起点t_{far}；
  - 当两者都为正时，第一个交点距起点t_{near}。

- 计算对应的碰撞点 collisionPoint

$$\text{collisionPoint} = r_s + t \cdot \boldsymbol{r}_d$$

- 碰撞点处的表面法线

$$\textbf{collisionNormal} = \text{normalize}(\text{collisionPoint} - s_p)$$

光线求交

- 光线与轴对齐的长方体的交点
  - 使用对角线上相对的两个顶点来定义立方体

$$\text{box}_{\text{mins}}:(x_{\min}, y_{\min}, z_{\min}) \text{ 和 } \text{box}_{\text{maxs}}:(x_{\max}, y_{\max}, z_{\max})$$

  - 使用 GLSL中针对 vec3 类型值的“/”操作找到光线与 6 个平面中的每一个相交的距离

$$\boldsymbol{t}_{\text{mins}} = (\text{box}_{\text{mins}} - r_{\text{s}}) / \boldsymbol{r}_{\text{d}}$$

$$\boldsymbol{t}_{\text{maxs}} = (\text{box}_{\text{maxs}} - r_{\text{s}}) / \boldsymbol{r}_{\text{d}}$$

  - $\boldsymbol{t}_{\text{mins}}$和$\boldsymbol{t}_{\text{maxs}}$向量分别包含光线起点距轴对齐立方体表面所在的三组平行面上的交点的最小和最大距离。然后，找到每个距离中的较小者和较大者：

$$\boldsymbol{t}_{\text{minDist}} = \min(\boldsymbol{t}_{\text{mins}}, \boldsymbol{t}_{\text{maxs}})$$

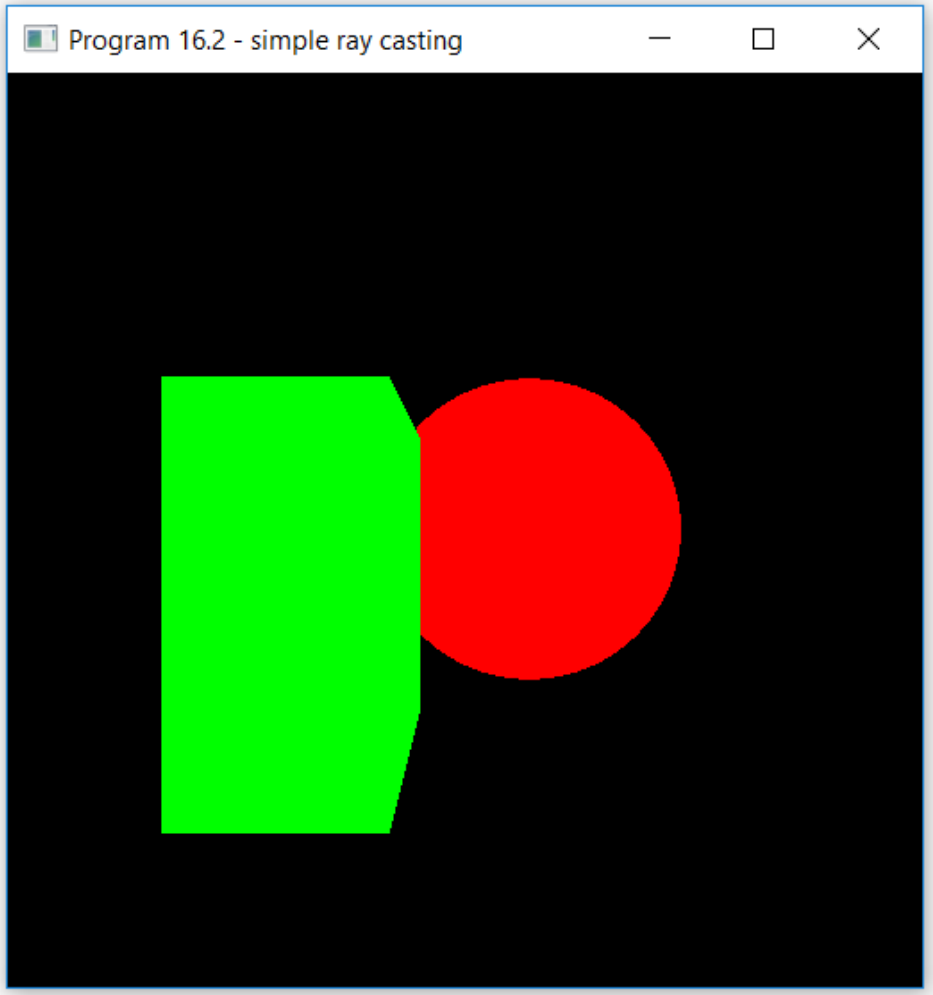
$$\boldsymbol{t}_{\text{maxDist}} = \max(\boldsymbol{t}_{\text{mins}}, \boldsymbol{t}_{\text{maxs}})$$

  - 射线原点到立方体面上的实际最近碰撞点的距离是$\boldsymbol{t}_{\text{minDist}}$中的最大值，而从射线原点到立方体面上最远碰撞点的距离是$\boldsymbol{t}_{\text{maxDist}}$中的最小值

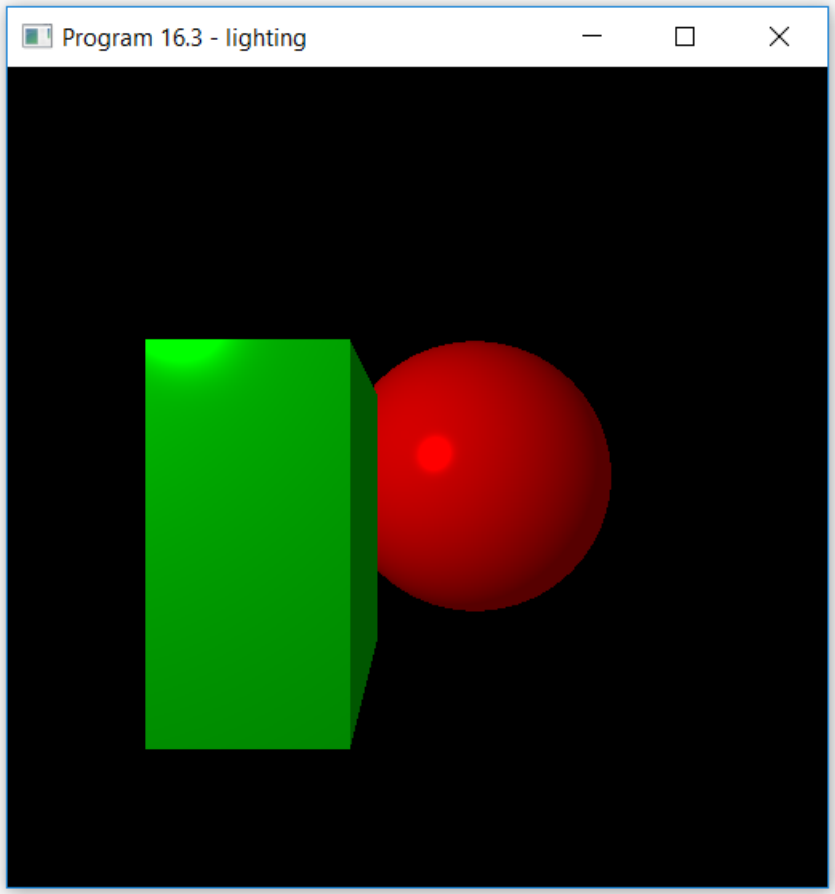
$$t_{\text{near}} = \max(t_{\text{minDist},x}, t_{\text{minDist},y}, t_{\text{minDist},z})$$

$$t_{\text{far}} = \min(t_{\text{maxDist},x}, t_{\text{maxDist},y}, t_{\text{maxDist},z})$$

无光照的简单光线投射的输出



添加 ADS 光照



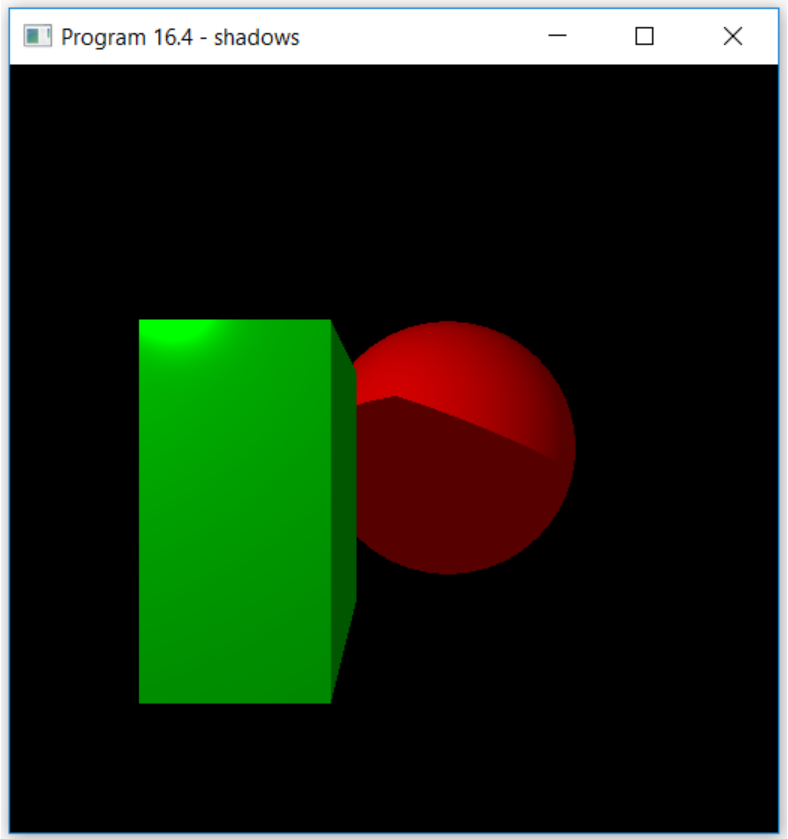
程序 16.3 添加光照

```
vec3 ads_phong_lighting(Ray r, Collision c)
{ // 通过全局环境光和位置光计算环境光分量
        vec4 ambient = global_ambient + pointLight_ambient * objMat_ambient;
// 计算光在表面的反射
        vec3 light_dir = normalize(pointLight_position - c.p);
        vec3 light_ref = normalize( reflect(-light_dir, c.n));
        float cos_theta = dot(light_dir, c.n);
        float cos_phi = dot( normalize(-r.dir), light_ref);
// 计算漫反射和镜面反射分量
        vec4 diffuse = pointLight_diffuse * objMat_diffuse * max(cos_theta, 0.0);
        vec4 specular = pointLight_specular * objMat_specular * pow( max( cos_phi, 0.0),
objMat_shininess);
        vec4 phong_color = ambient + diffuse + specular;
        return phong_color.rgb;
}
```

```
vec3 raytrace(Ray r)
{       Collision c = get_closest_collision(r);
        if (c.object_index == -1) return vec3(0.0); // 如果没有碰撞，返回黑色
        if (c.object_index == 1) return ads_phong_lighting(r,c) * sphere_color;
        if (c.object_index == 2) return ads_phong_lighting(r,c) * box_color;
}
```

添加阴影

- 定义从正在渲染的碰撞点指向灯光位置射线
- 使用求交计算，可以确定该射线的最近碰撞位置。如果它比位置光更近，那么在光和碰撞点之间必定有一个物体，且该碰撞点必定在阴影中

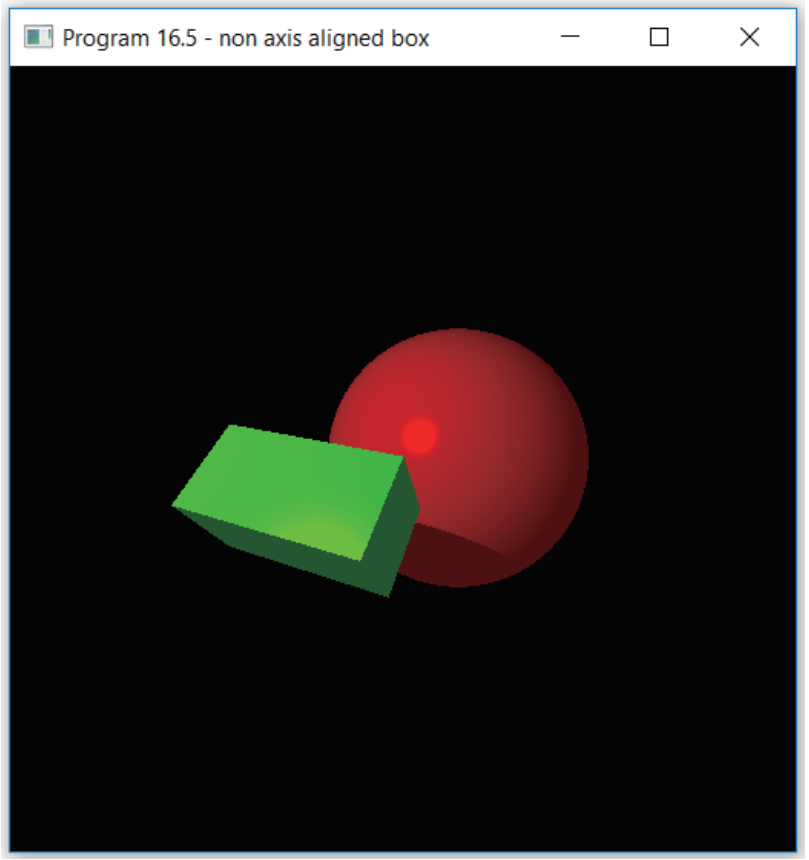


程序 16.4 添加阴影

```
// 将阴影感知射线投射进场景
Collision c_shadow = get_closest_collision(light_ray);
// 如果阴影感知射线碰到了物体，并且碰撞位置在光源与表面之间
if (c_shadow.object_index != -1 && c_shadow.t < length(pointLight_position–c.p))
{ in_shadow = true;
}
// 如果表面在阴影中，则不添加漫反射和镜面反射分量
if (in_shadow == false)
{ // 计算光在表面的反射
   vec3 light_dir = normalize(pointLight_position - c.p);
   vec3 light_ref = normalize( reflect(-light_dir, c.n));
   float cos_theta = dot(light_dir, c.n);
   float cos_phi = dot( normalize(-r.dir), light_ref);
  // 计算漫反射和镜面反射分量
   diffuse = pointLight_diffuse * objMat_diffuse * max(cos_theta, 0.0);
   specular = pointLight_specular * objMat_specular * pow( max( cos_phi, 0.0),objMat_shininess);
}
vec4 phong_color = ambient + diffuse + specular;


return phong_color.rgb;
```

光线与非轴对齐的长方体的交点

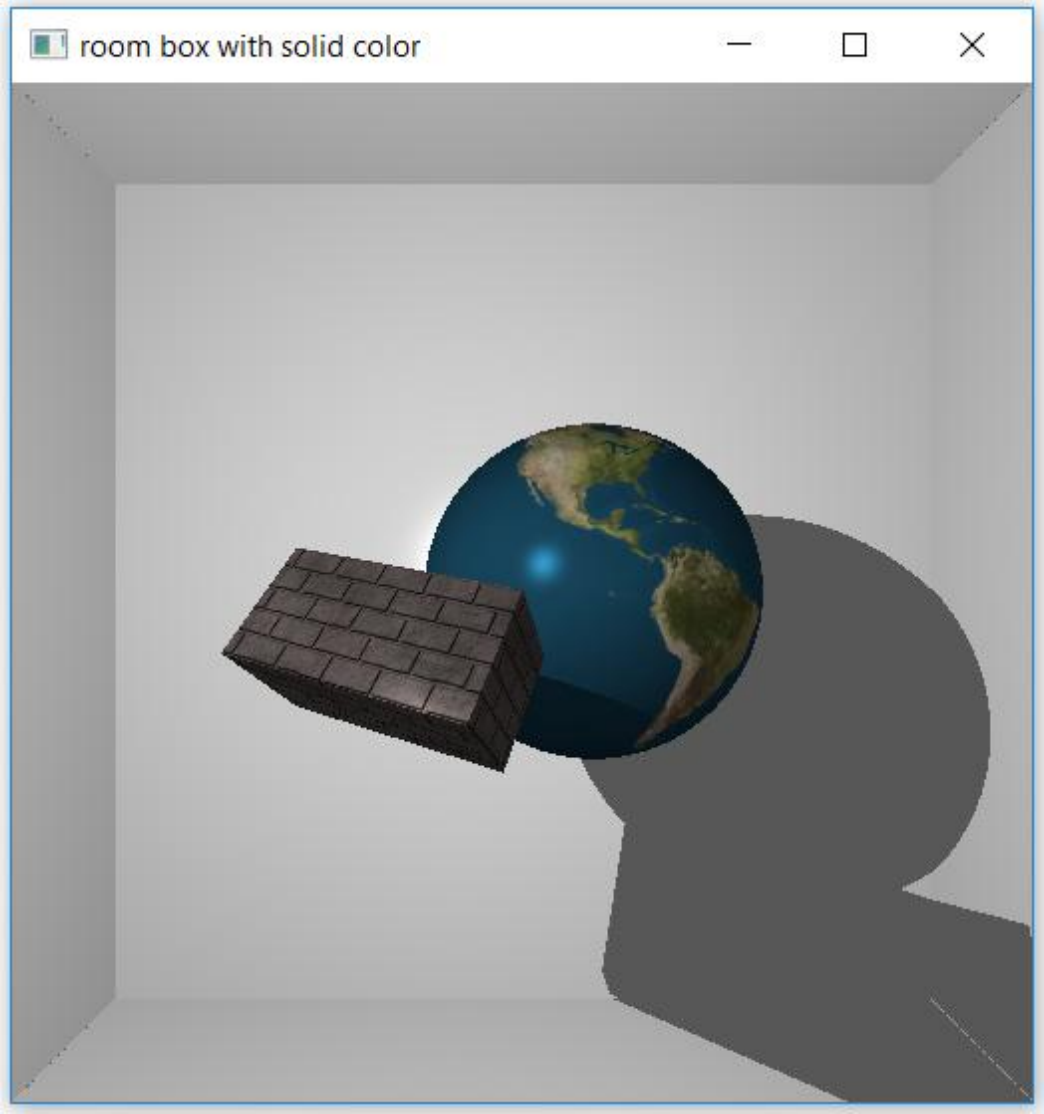
- 使用 box_mins 和 box_maxs 变量定义一个以原点为中心的长方体
- buildTranslate()函数和 buildRotate()函数构建平移变换矩阵和旋转变换矩阵
- 用这些矩阵的逆矩阵来修改射线的起点和方向
- 像前面轴对齐的长方体求加一样继续计算，以生成碰撞距离
- 根据实际的射线起点和方向它来计算全局坐标下的碰撞点位置



添加纹理

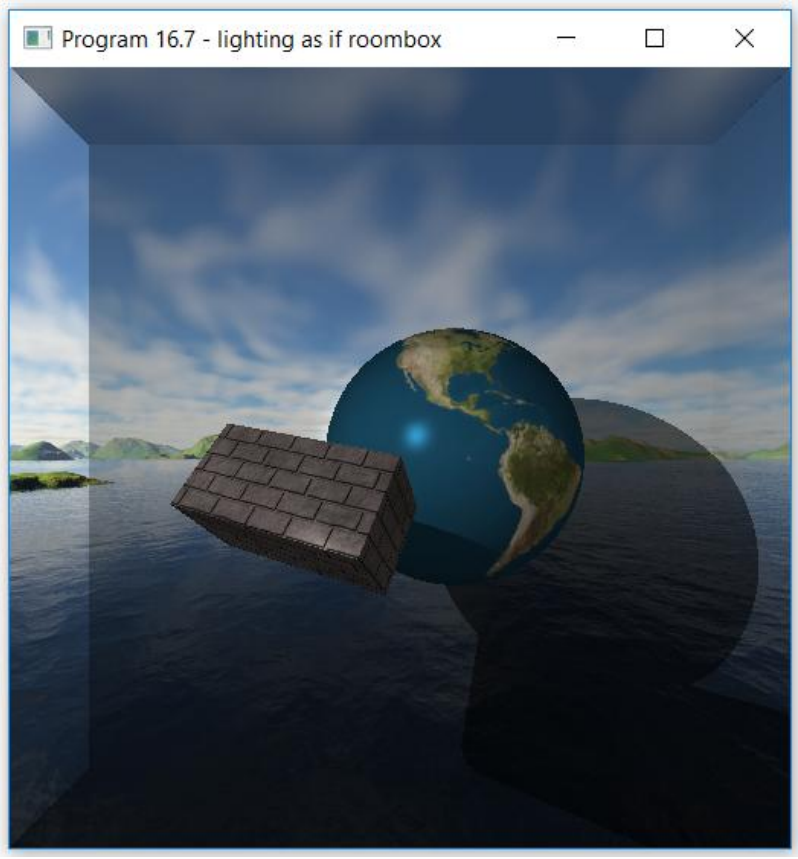
- 计算光线交点的纹理坐标

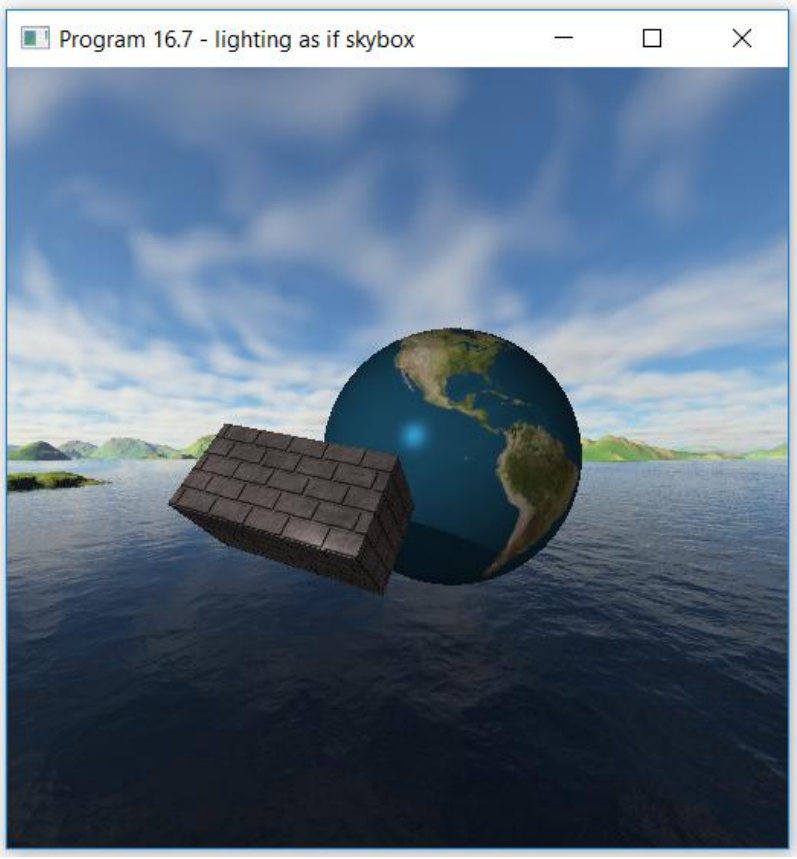
天空盒



天空盒

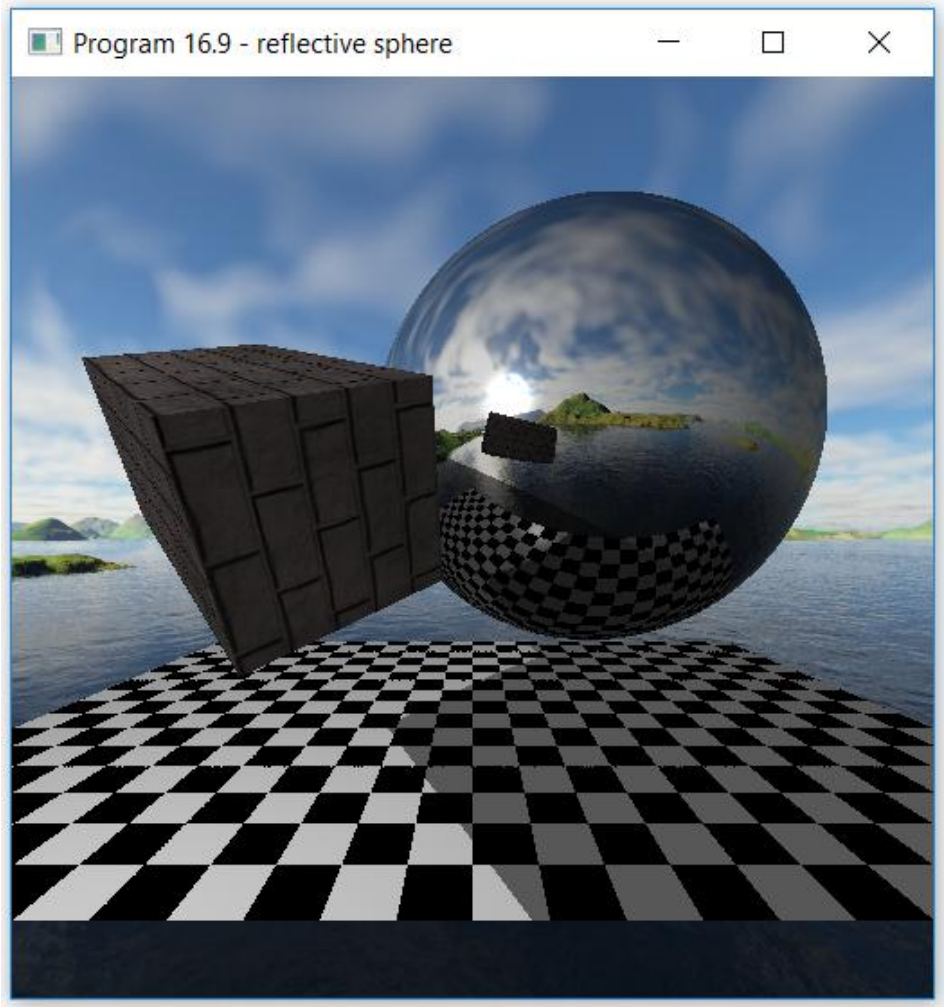
- 左右的区别?


Program 16.7 - lighting as if roombox

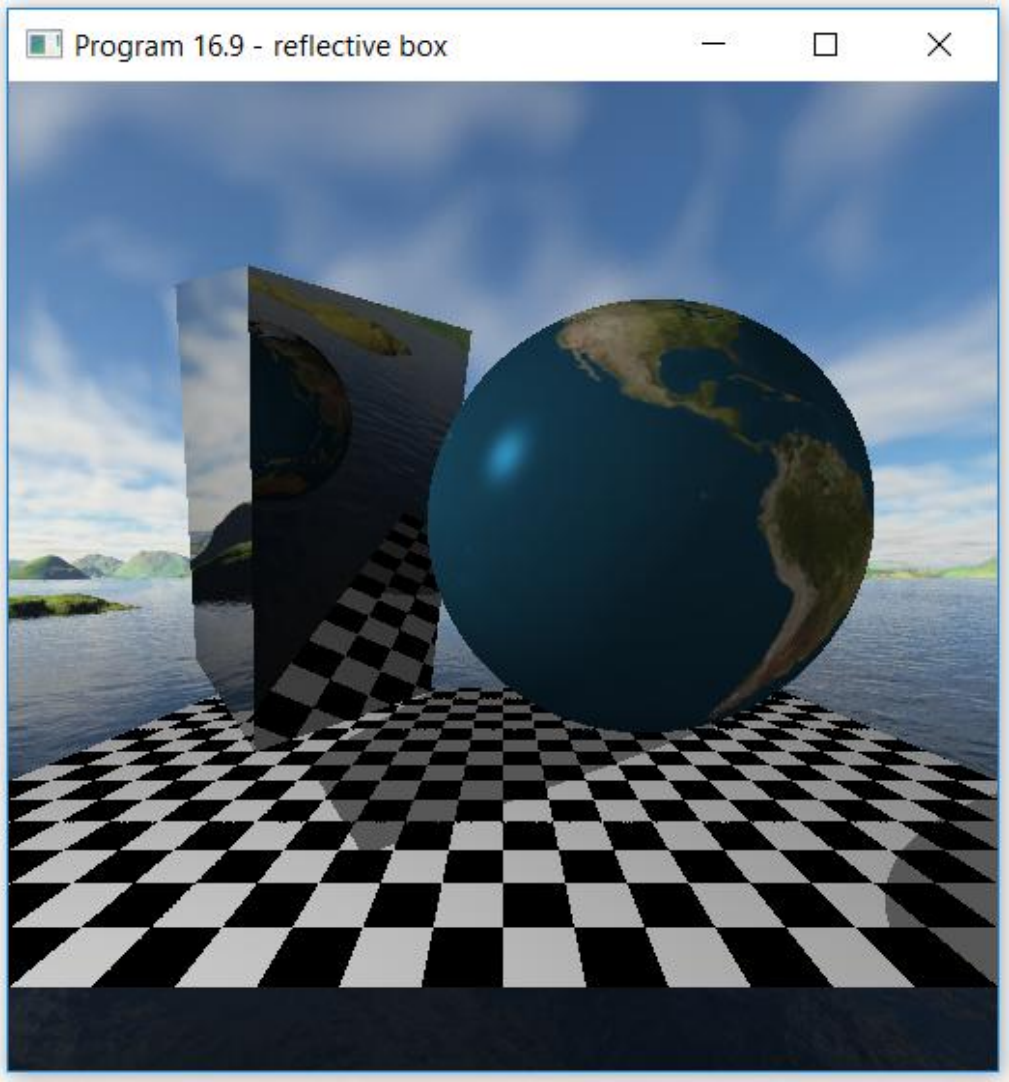

Program 16.7 - lighting as if skybox

光线追踪

- 从交点生成一条新射线（表面反射的光线，称为次级光线），查看新射线通向何处

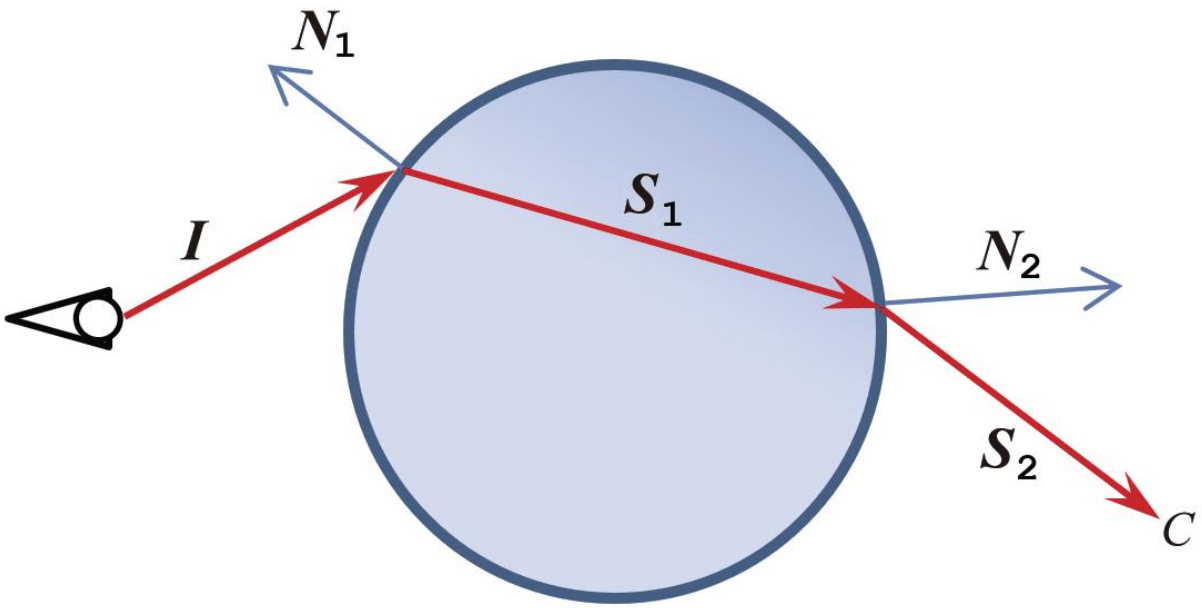


反射



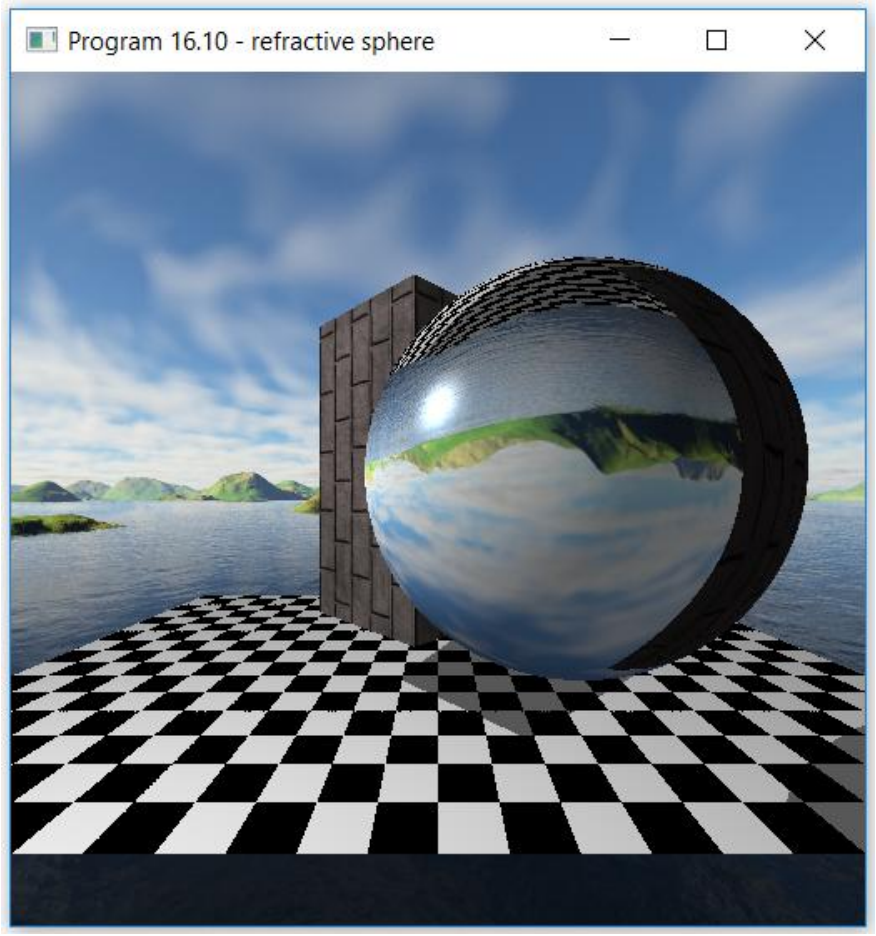
折射

- 用两束次级光线折射，穿过透明球体

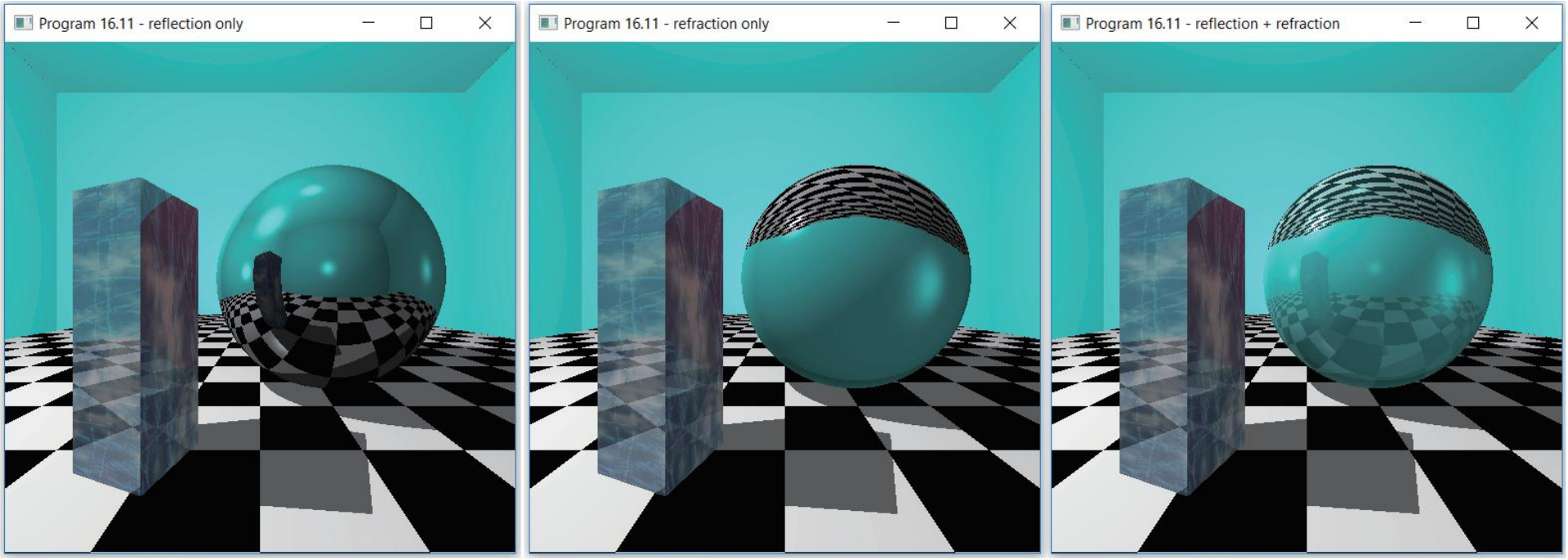


折射

- 通过实心透明球体的折射——两束次级光线



结合反射、折射和纹理



仅带反射的球体（左）、仅带折射的球体（中）、同时具有反射与折射的球体（右）

增加光线数

- 通过两个透明立方体查看星星。由于每个透明物体通常需要两束次级光线（一束进入物体，一束离开物体），因此我们总共需要 5 束光线（图中以细箭头表示）

需要多少光束?

GLSL 不支持递归