计算机图形学自选任务-光线追踪

shkaaa111

1. 项目简介

本项目基于`WebGL`实现了光线追踪的相关算法,实现了光线与球体,长方体,平面求交的逻辑,实现了光线追踪的过程,并实现了三个场景展示效果.核心的场景设置,光线追踪算法等均直接在片元着色器中实现,对应于`index.html`的`fshader`,为方面查看代码和修改,已将其中的代码复制到`fragment_shader.glsl`中,直接在编辑器中下载`glsl`的插件,可以清晰的编辑和阅读片元着色器的代码.

直接运行`index.html`即可. 因为三个场景都是动态的,并且对光源进行了9个点的采样,对着色器代码的编译工作量较大,因此加载可能较慢,本地测试大概运行后要10到20秒才能展示出画面.

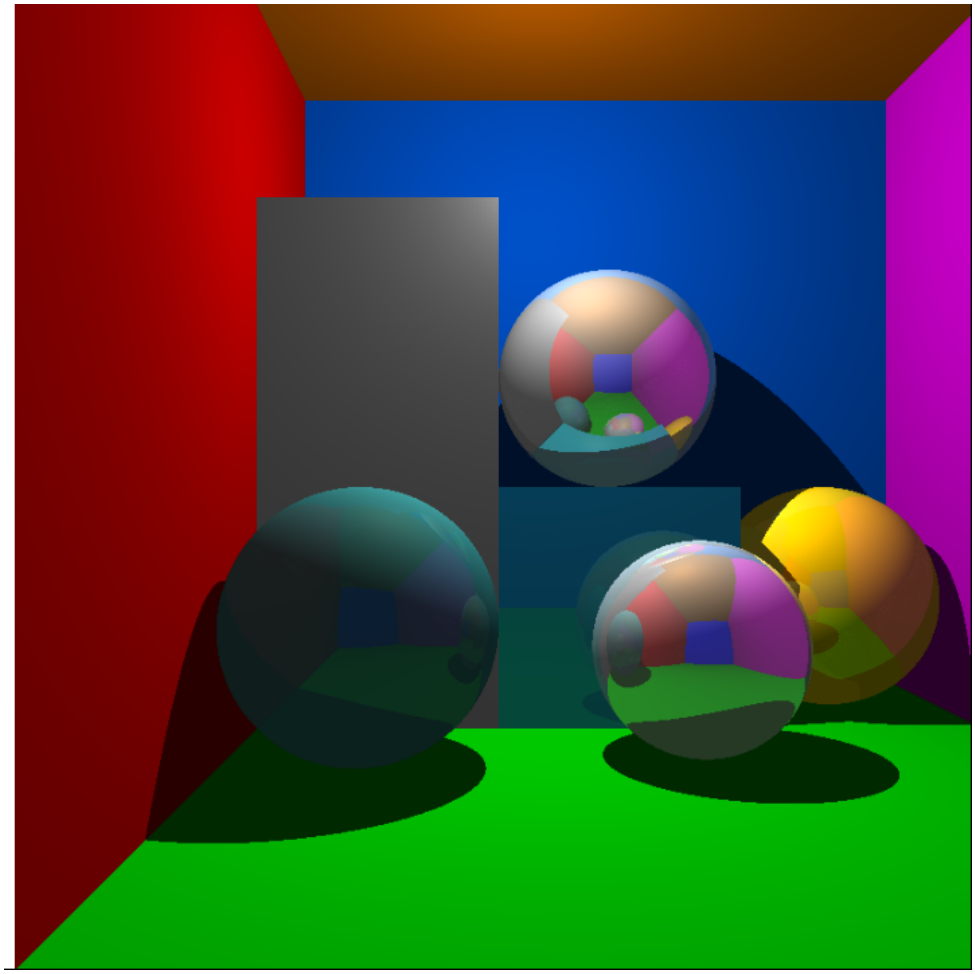
可能出现的问题:在使用外接显示器的时候画面基本流畅,但直接用笔记本电脑屏幕显示就会卡顿.解决方案:如果是windows可以更改显示卡设置,使得在运行chrome浏览器时,使用笔记本的独立显卡而不是集成显卡.

2. 项目效果展示

压缩包中有简单展示视频.可以选择三种不同的场景,以及设定光线反射的次数来查看效果.

2.1 场景1:

在一个正方体空间中有叠放的盒子球题体,不同物体的反射率不同,可以设置反射次数为1次或多次,从而查看效果的变化.

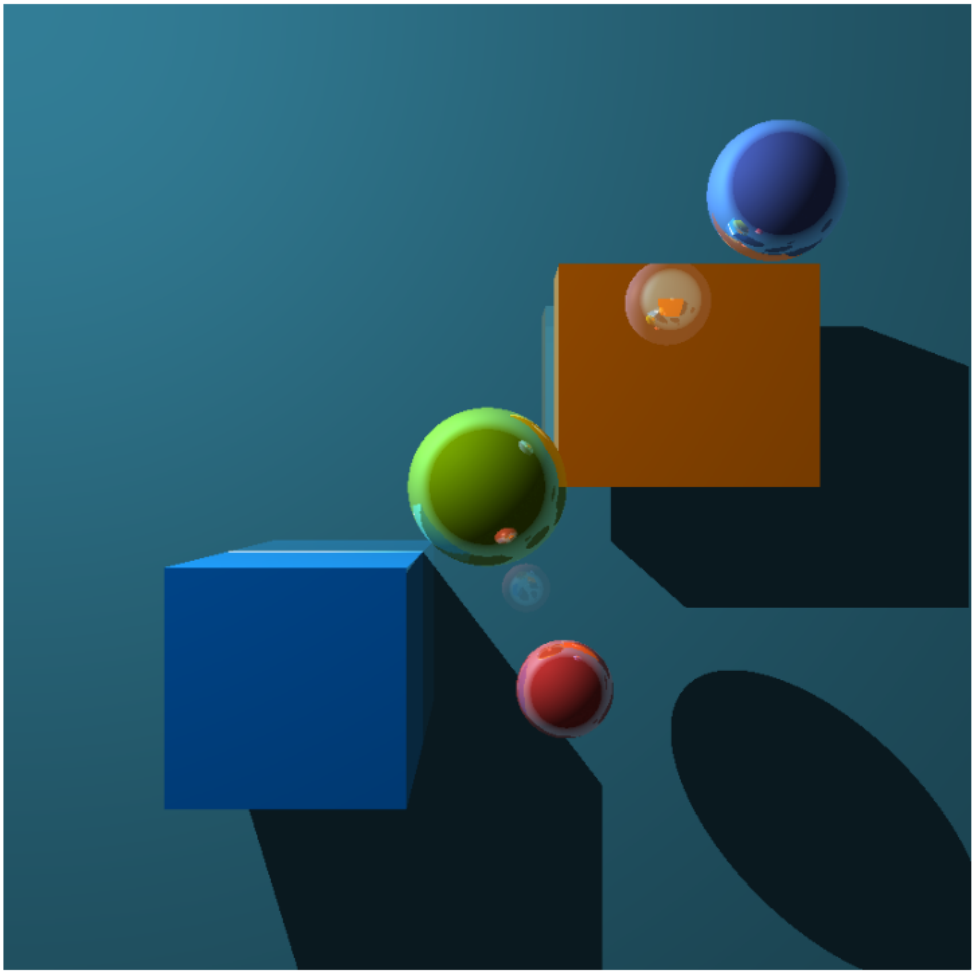


2.2 场景2:

场景二展示了多个立方体,球体以及地板之间的相互映衬.

2.3 场景3:

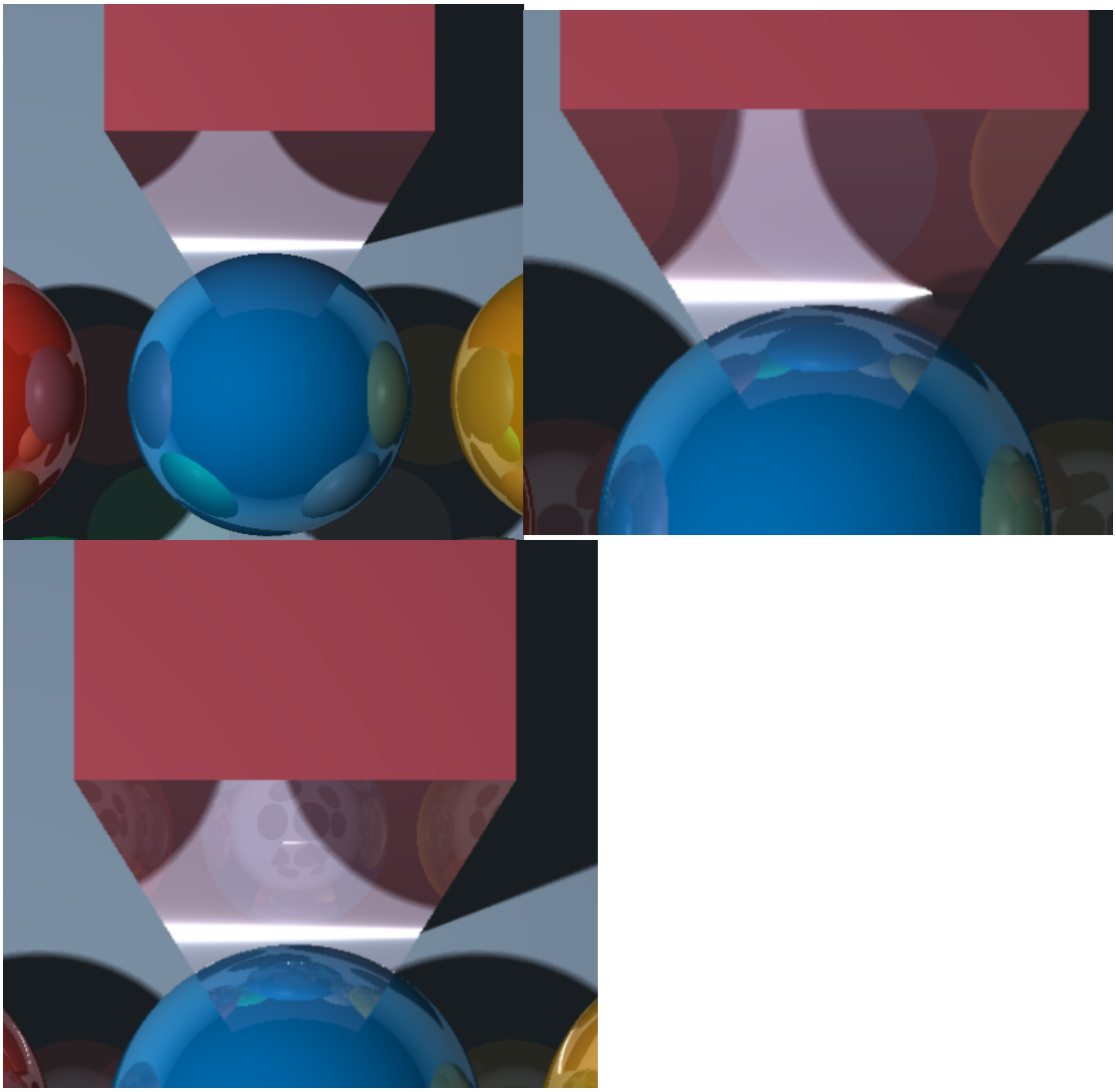
场景3让球体和长方体移动起来,可以查看其阴影的映衬,叠加,遮挡等效果.



2.4 反射周围环境的效果

通过设定反射率的形式,以此为权重叠加光线的颜色实现.从而模拟不锈钢材质的物体,反映周围环境. 通过设置光线反射的次数,可以看出明显的区别:

- 对比:从左到右反射1次.反射2次.反射10次

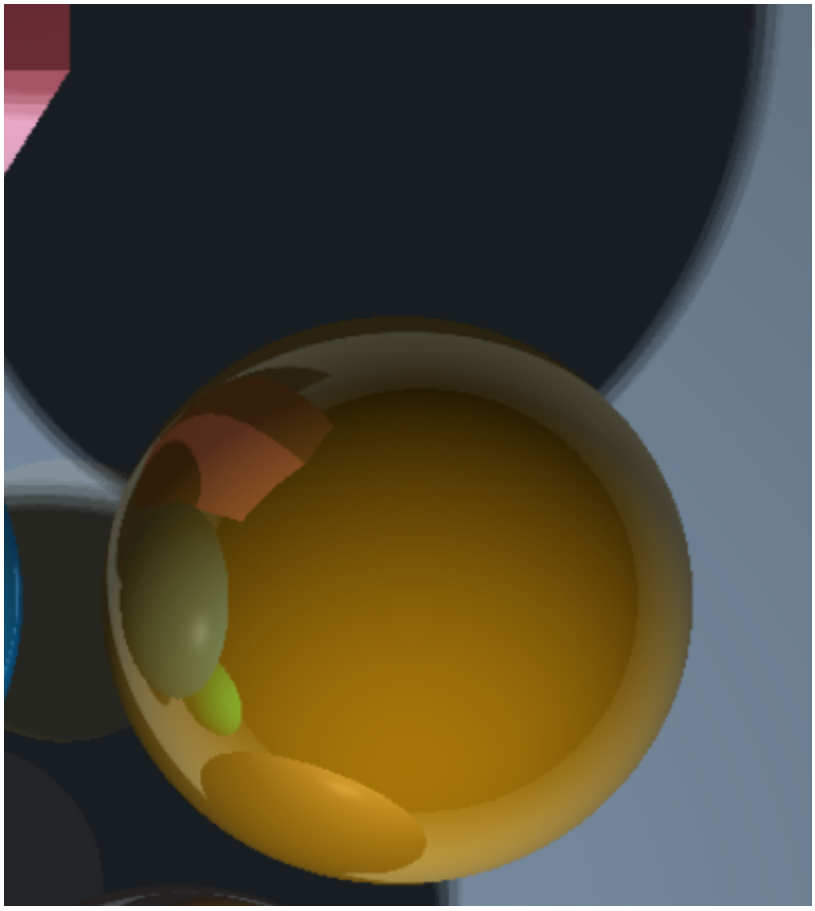


2.5 软阴影实现

由于本项目实在`WebGL`中直接用`glsl`语言完成的,与`OpenGL`相比缺少一些库的支持,实现光源采样较为复杂,而且受性能限制,**对光源的采样使用了附近的9个点进行模拟**.

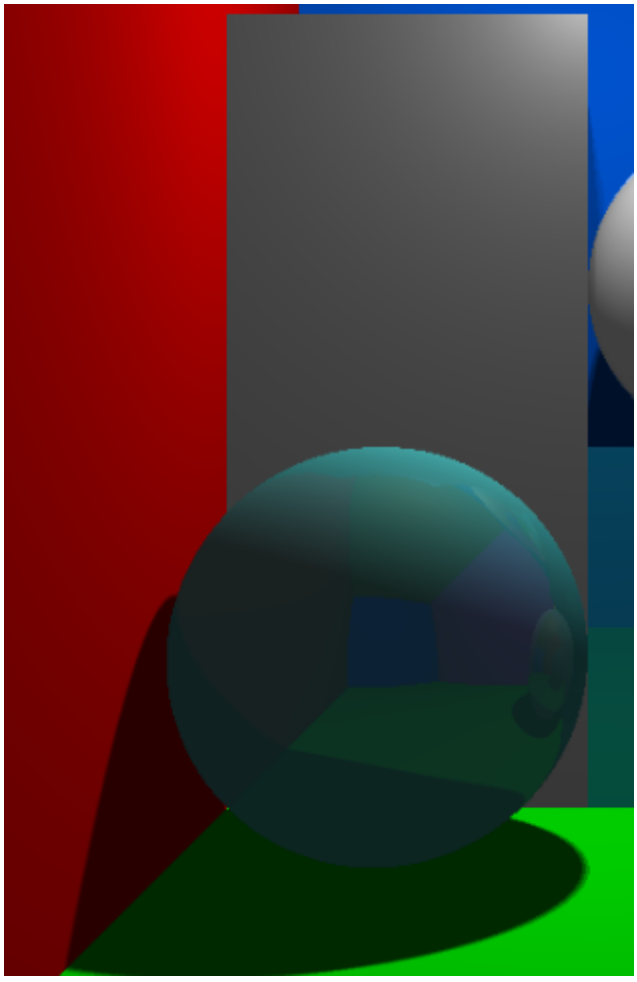
如果换一种实现语言,**可以用随机采样+更多次的采样数**来实现更好的效果.

- 如下图,阴影的边缘有一定的过渡,实现软阴影效果



- 如下图,光线强度模拟了**球面光源**,有一定的环状光晕,光强随着距离衰减,越靠近光源越亮.

$$w = w_0 * \frac{lightSize^2}{distance^2}$$



3. 源码介绍

3.1 基本架构

- `index.html`中直接写入了两个着色器(内容和`fragment_shader.glsl`文件中的一致),其余是网页的相关内容.
- `main.js`仅完成了最基本的渲染流程,传入一些参数,绘图全部在片元着色器中实现.
- `fragment.glsl`中为核心代码

3.2 结构体定义

- 定义了三种基本图形:球面,平面,长方体

```
struct Box {
    vec3 leftBottom;//左下角顶点
    vec3 rightTop;//右上角顶点
    vec3 hit;
    vec3 color;
    float reflectivity;
};
```

- 定义了光线和光源

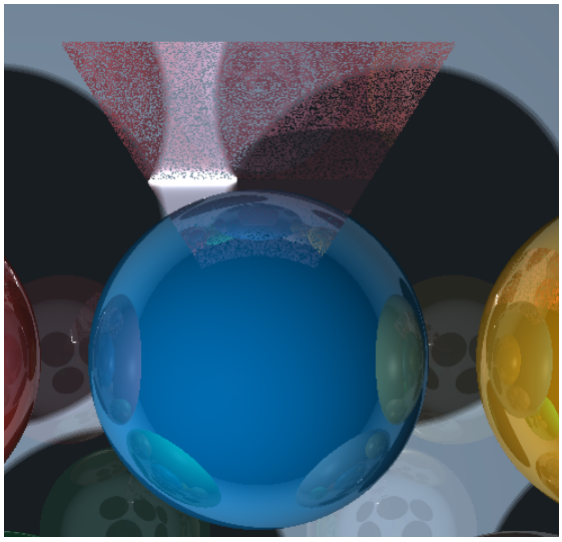
```
struct Ray {
    vec3 origin;// 源
    vec3 direction;// 方向
    float intensity;// 强度
};

struct Light {
    vec3 position;// 光源位置
    vec3 ambient;// 环境光
    vec3 diffuse;// 漫反射
    vec3 specular;// 镜面反射
    float lightsize;// 光源尺寸(球型光源,强度与距离的平方成反比)
};
```

3.3 场景定义

```
void secene1(){
    //设置光源属性
    ...
    //设置基本物体数量
    numSpheres = 4;
    numPlanes = 6;
    numBoxes = 2;
    //设置每个图形的具体信息
    boxes[0].color = vec3(0.1, 1.0, 1.1);
    boxes[0].reflectivity = 0.1;
    boxes[0].leftBottom = vec3(0.0, -1.0, -1.0);
    boxes[0].rightTop = vec3(1.0, -0.0, -0.0);
    ...
}
```

3.4 相交检测

- 设光线为$y = Dt + O$,D为单位向量,每一个相交测试的目的都是求出对应的t值并返回,如果$t < 0$证明没有交点. 如果$t > 0$证明有交点,并且t的值就代表着光线源头到该点的距离.
- 要注意浮点数判断相等的方式,不然显示出的正方形只有一些不连贯的点.如图所示:



```
//检测和长方体是否相交
float checkIntersectBox(inout Box box, Ray ray) {
    vec3 invdir = 1.0 / ray.direction;
    vec3 tmin = (box.leftBottom - ray.origin) * invdir;
    vec3 tmax = (box.rightTop - ray.origin) * invdir;
    vec3 t1 = min(tmin, tmax);
    vec3 t2 = max(tmin, tmax);
    float tNear = max(max(t1.x, t1.y), t1.z);
    float tFar = min(min(t2.x, t2.y), t2.z);
    if (tNear > tFar) {
        return -1.0;
    } else {
        // 可以确定相交
        vec3 dis = tNear * ray.direction;
        vec3 judge1 = abs(ray.origin + dis - box.leftBottom);
        vec3 judge2 = abs(ray.origin + dis - box.rightTop);
        box.hit = vec3(0.0, 0.0, 0.0);
        if (judge1.z < 0.00001) {
            box.hit = vec3(0.0, 0.0, -1.0);
        } else if (judge1.y < 0.00001) {
            box.hit = vec3(0.0, -1.0, 0.0);
        } else if (judge1.x < 0.00001) {
            box.hit = vec3(-1.0, 0.0, 0.0);
        } else if (judge2.z < 0.00001) {
            box.hit = vec3(0.0, 0.0, 1.0);
        } else if (judge2.y < 0.00001) {
            box.hit = vec3(0.0, 1.0, 0.0);
        } else if (judge2.x < 0.00001) {
            box.hit = vec3(1.0, 0.0, 0.0);
        } else {
            return -1.0;
        }
        return tNear;
    }
}
```

```
// 检测和平面是否相交
float checkIntersectPlane(Plane plane, Ray ray) {
    float up = dot(plane.point - ray.origin, plane.normal);
    float down = dot(ray.direction, plane.normal);
    if (down == 0.0 || down == -0.0) return 0.0;
    return up / down;
}
```

```
// 检测和球体是否相交
float checkIntersectSphere(Sphere sphere, Ray ray) {
    vec3 dis = (ray.origin - sphere.center);
    float B = -2.0 * dot(ray.direction, dis);
    float delta = B * B - 4.0 * (dot(dis, dis) - sphere.radius * sphere.radius);
    float t = 0.0;
    if (delta > 0.0) {
        float temp = abs(sqrt(delta));
        t = (B - temp) / 2.0;
    }
    if (delta == 0.0) {
        t = B / 2.0;
    }
    return t;
}
```

3.5 光线追踪

光线追踪分为以下三步:

- 初始化光线
- 做相交测试,找到最近的物体
- 取得物体相关的信息,得到反射点,反射率,反射点的法向量
- 遍历光源的采样(受性能限制,采取了固定采样9个点),每一次采样都有以下流程:
    1. 添加环境光`color += lights[i].ambient * objColor;`
    2. 计算阴影光线,检测阴影和当前光线的相交情况`intersectsBeforeLight(shadowRay, lights[i])`
    3. 如果不在阴影里,添加漫反射光和直射光,并考虑光强的衰减(平方反比)
- 计算反射光线,设置最终颜色并返回
- 当前光线设为反射光线

```
// 根据传入的反射次数决定迭代次数
vec3 color = vec3(0.0, 0.0, 0.0);
    for (int i = 0; i <= maxReflections; i++) {
        if (i > int(reflections)) break;
        RayAndColor rayTracer = traceRay(currRay);
        color += rayTracer.color;
        currRay = rayTracer.reflectedRay;
    }
```

4. 总结

光线追踪算法实现起来非常有趣,因为只需要一条一条追踪光线,其实逻辑比起光栅化更容易,写起来更顺畅. 在相交测试的部分,根据数学公式一步一步仿写出了相应的代码. 最困难的部分应该是实现软阴影和立方体的香蕉检测,由于`glsl`语言里面没有随机数功能,控制流也没法写的很复杂,因此最终采取了固定的几个点采样来模拟区域光源.