

Ray Tracing in one Weekend

关于本教程

此教程的原版是 Peter Shirley 的 "["Ray Tracing in One Weekend"](#)" . 他把Ray tracing讲的深入浅出，水平很高，用来入门是很好的教程。但是原文是英文的，书中的例子是用C++写的，所以需要一定的C++和英文基础。本教程是用Unity把原教程的例子用C#重新写了一遍，所以特别适合Unity 3D的程序员。其实也可不用Unity，主要是这样运行例子比较简单，省的安装一些其他类库。

本教程是用markdown写的。因为嵌入了tex，可能不同的浏览器看到的不大一样。如果看到数学公式解析有问题，就看README.pdf，在readme.md的同级目录里面。

关于代码

运行：项目代码包含在Unity的工程里面，可以用Unity打开在Raytracing菜单,点击相应的章节就可以运行代码。

代码冗余的说明：不同章节的代码里有些冗余的类，是因为在不同的章节中可能会少有改动。就让不同的章节都包含了一份完整的代码。

代码运行时间：后面几章的代码运行时间会比较久，尤其是最后一张可能会有几个小时。如果想快速看到结果，可以修改生成图片的大小和采样数。

Unity版本：对Unity版本没有特殊要求，在Unity2018.4 和Unity2018.3上测试过，其他版本应该问题不大

关于Ray tracing

Ray tracing是一种计算机渲染图像的算法，特点是计算量大但是效果好，之前主要用于电影或者动画的渲染。Nvidia的显卡支持了实时的Ray tracing，现在也可以在实时渲染的地方用了。之前实时的图形学使用的是光栅化算法。关于光栅化的算法介绍，推荐一个[网站](#)上面讲的很详细。

第一章 输出一张图片

这章讲一下，计算机中图片，图片是由像素构成的，每个像素，可以被编码成一个[R,G,B]的向量。关于RGB和颜色空间，也推荐一个[网页](#)上面有比较详细的介绍。看下面这段代码生成了一张图片

```

public class Chapter1
{
    [MenuItem("Raytracing/Chapter1")]
    public static void Main()
    {
        int nx = 1280;
        int ny = 720;

        Texture2D tex = ImageHelper.CreateImg(nx, ny);
        for (int j = ny - 1; j >= 0; --j)
        {
            for (int i = 0; i < nx; ++i)
            {
                float r = (float)(i) / (float)(nx);
                float g = (float)(j) / (float)(ny);
                float b = 0.2f;
                ImageHelper.SetPixel(tex, i, j, r, g, b);
            }
        }

        ImageHelper.SaveImg(tex, "img\\chapter1.png");
    }
}

public static class ImageHelper
{
    public static Texture2D CreateImg(int width ,int height)
    {
        Texture2D tex = new Texture2D(width, height, TextureFormat.RGB24, false);
        return tex;
    }

    public static void SetPixel(Texture2D tex,int x,int y,float r,float g,float b)
    {
        tex.SetPixel(x, y, new Color(r, g, b));
    }

    public static void SetPixel(Texture2D tex,int x,int y,Vector3 color)
    {
        tex.SetPixel(x, y, new Color(color.x, color.y, color.z));
    }

    public static void SaveImg(Texture2D tex, string path)
    {
        var bytes = tex.EncodeToPNG();
        File.WriteAllBytes(Path.Combine(Application.dataPath, path), bytes);
    }
}

```

在Unity的菜单中运行后，得到如下结果，保存在Img文件夹chapter1.png。



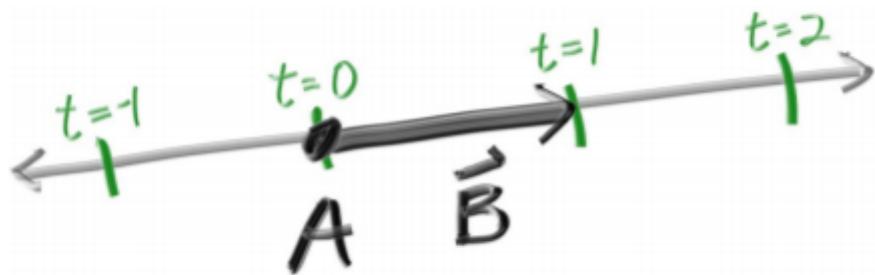
。ImageHelper直接使用了Unity中保存图片的接口。Main函数中就是对一张1280 X 720的每个像素点赋值。这样我们就得到了一张用算法生成的图片。

第二章 向量类

这章主要介绍向量的基本知识。如果有不熟悉的，请在网上查一下。这里就不细讲了。因为使用了Unity，里面有自带的向量类Vector3，所以在此教程中就不自己实现了。

第三章 射线，简单的摄像机和背景

ray tracing算法，有翻译为光线追踪的，也有翻译为射线追踪，所以肯定要用到射线。本书里面的射线是由参数方程表示的，射线是由一个端点和一个方向向量组成的 $p(t) = A + t * \vec{B}$ 。 A 就是三维空间中的一个点， \vec{B} 是射线的方向是一个向量， t 是一个实数。当 t 从0变化到 $+\infty$ 时， $p(x)$ 取遍射线上所有的点。如果 t 取负数，那么 $p(t)$ 可以理解为跟原来射线同原点但是方向相反的射线。上一张原教程中的图：



要实现一个Ray的类很简单，而且Unity中实现了一个Ray的类，就不自己实现了。

```
namespace UnityEngine
{
    public struct Ray
    {
        public Ray(Vector3 origin, Vector3 direction);

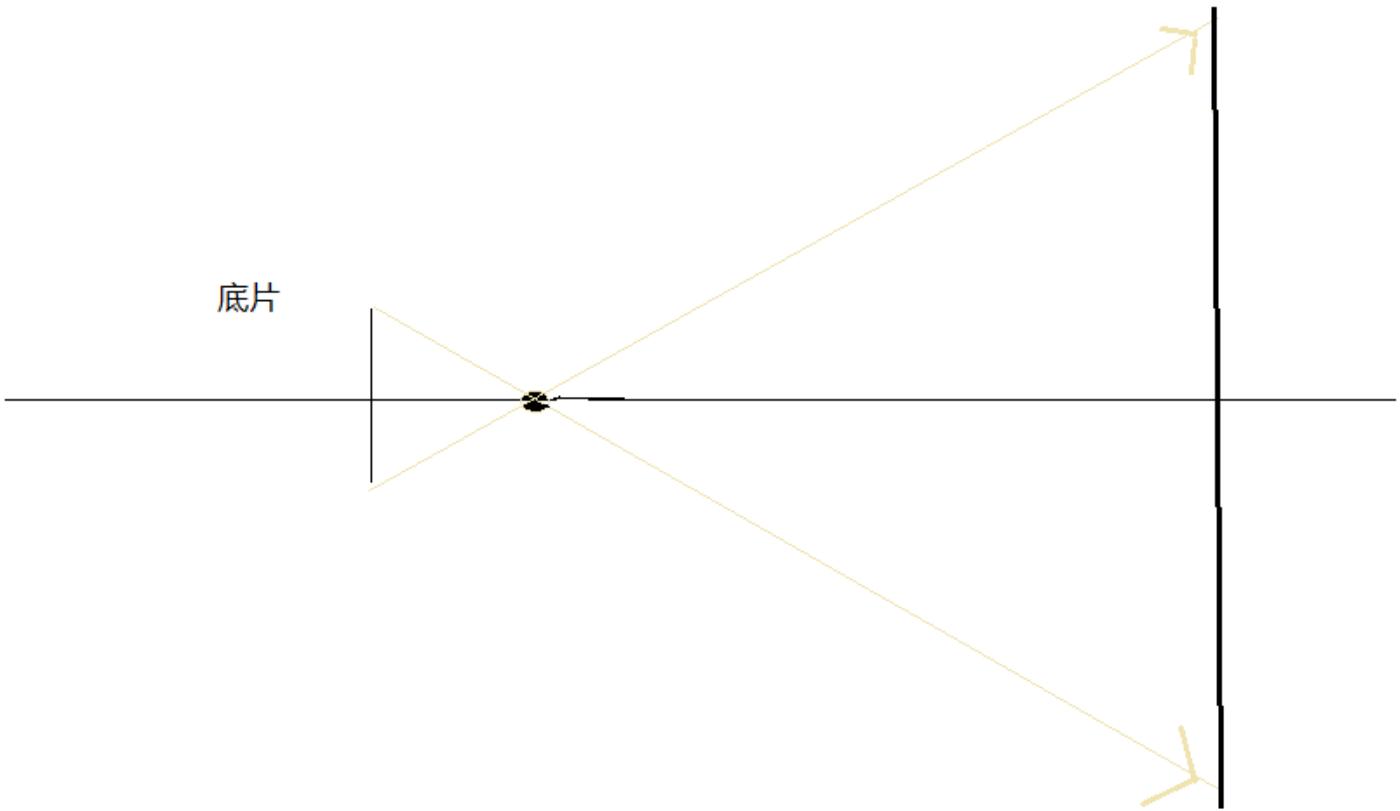
        public Vector3 origin { get; set; }

        public Vector3 direction { get; set; }

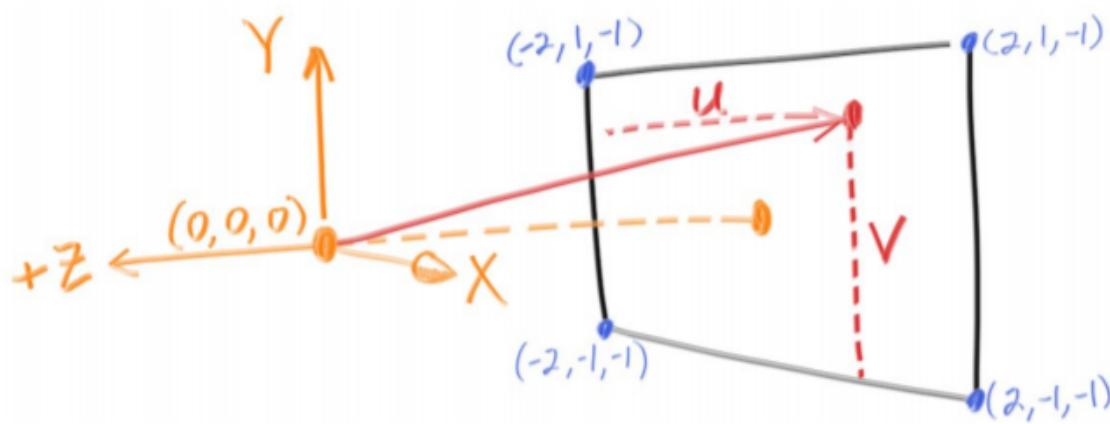
        //
        // 摘要:
        //     Returns a point at distance units along the ray.
        //
        // 参数:
        //     distance:
        public Vector3 GetPoint(float distance);
    }
}
```

要得到射线指向的点，即射线的终点，我们需要计算 $p(t)$ 就是使用GetPoint方法， t 可以理解为从原点发射出去的距离distance。有兴趣的可以自己写一个Ray的类实现一下，看看跟Unity的计算结果是否相同。

有了这些基础知识，我们现在就可以写一个简单的Ray tracer了。先让我们来想一下，相机是如何拍照的，假设镜头是一个小孔，无数的光线从这里穿过，最终在底片上留下的痕迹，形成照片。因为光线是可逆的，我们可以假想，光线是从最后的照片的某个像素出发，然后穿过镜头，与真实世界里面的物体发生作用，通过遇到的物体来决定改像素的颜色，这个就是Ray tracer的算法的核心思想。由于相机的参数一旦确定，如下图：



我们就可以忽略掉底片，假想光线是从相机直接投在前面的幕布上的。然后我们来写第一个光线追踪算法：



假如我们的以摄像机所在的位置作为原点，用右手坐标系，幕布就在Z轴的负方向上，且与XY屏幕平行。幕布的左上角在空间的 $(-2, 1, -1)$ ，右下角在 $(2, -1, -1)$ （这里我们假设，幕布的分辨率是1280 X 720的）现在我们从相机发射一道射线，指向幕布上的一个像素点，这样就得到了条射线，然后我们根据射线的方向向量，来插值两个颜色，最终得到像素的颜色。等射线跑遍了整个幕布，这是我们

就得到了一张图。这里用射线的Y方向做差值，图像就是从上到下渐变的，没有什么道理只是单纯的图片不那么单调。代码如下：

```
public class Chapter3
{
    private static Vector3 topColor = Vector3.one;
    private static Vector3 bottomColor = new Vector3(0.5f, 0.7f, 1.0f);

    public static Vector3 RayCast(Ray ray)
    {
        Vector3 unit_direction = ray.direction.normalized;
        float t = 0.5f * (unit_direction.y + 1.0f);
        return Vector3.Lerp(topColor, bottomColor, t);
    }

    [MenuItem("Raytracing/Chapter3")]
    public static void Main()
    {
        int nx = 1280;
        int ny = 720;

        Vector3 lower_left_corner = new Vector3(-2.0f, -1.0f, -1.0f);
        Vector3 horizontal = new Vector3(4.0f, 0.0f, 0.0f);
        Vector3 vertical = new Vector3(0.0f, 2.0f, 0.0f);
        Vector3 origin = Vector3.zero;

        Texture2D tex = ImageHelper.CreateImg(nx, ny);
        for (int j = ny - 1; j >= 0; --j)
        {
            for (int i = 0; i < nx; ++i)
            {
                float u = (float)(i) / (float)(nx);
                float v = (float)(j) / (float)(ny);

                Ray r = new Ray(origin, lower_left_corner + u * horizontal + v * vertical);
                Vector3 color = RayCast(r);

                ImageHelper.SetPixel(tex, i, j, color);
            }
        }

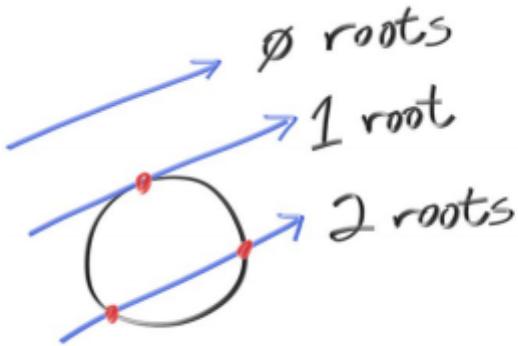
        ImageHelper.SaveImg(tex, "Assets/Img/chapter3.png");
    }
}
```

最终渲染的图片如下：



第四章 画个圆

这次我们在场景里添加个圆，为啥是个圆呢，因为判断射线跟圆是否相交最简单。圆的方程： $(x - cx)^2 + (y - cy)^2 + (z - cz)^2 = R^2$ 这里 (cx, cy, cz) 是圆的中心点坐标， R 是半径。其实这个方程就是描述了圆的几何性质：圆上任意一点到圆心的距离是相等的。如果写成向量的形式 $\vec{C} = (cx, cy, cz)$ 是圆心点， \vec{P} 是圆上任意一点， R 是半径，那么圆的方程可以写成 $\|\vec{P} - \vec{C}\| = R^2$ ，也就是 $\text{dot}(\vec{P} - \vec{C}, \vec{P} - \vec{C}) = R^2$ 这里 dot 代表向量的点乘。还记得射线的向量形式么 $p(t) = A + t * \vec{B}$ 。如果射线与圆有交点的话，也就是射线上存在一个点，可以满足圆的方程式，因为那个点也在圆上。所以这里我们把射线的方程带入到圆面方程里面，得到 $\text{dot}(p(t) - \vec{C}, p(t) - \vec{C}) = R^2$ 也就是 $\text{dot}(\vec{A} + t * \vec{B} - \vec{C}, \vec{A} + t * \vec{B} - \vec{C}) = R^2$ 这里把点乘展开，并整理可得 $t * t * \text{dot}(\vec{B}, \vec{B}) + 2 * t * \text{dot}(\vec{B}, \vec{A} - \vec{C}) + \text{dot}(\vec{A} - \vec{C}, \vec{A} - \vec{C}) - R * R = 0$ 这里看上去有点复杂，其实除了 t 以外都是常量，假如让 $a = \text{dot}(\vec{B}, \vec{B})$, $b = 2 * \text{dot}(\vec{B}, \vec{A} - \vec{C})$, $c = \text{dot}(\vec{A} - \vec{C}, \vec{A} - \vec{C}) - R * R$ 这样替换以后得到 $at^2 + bt + c = 0$ ，是不是很眼熟，就是一元二次方程，这里 a, b, c 都是常数，可以通过上面的式子计算得到。这样一个射线与圆相交的问题，就变成了一个一元二次方程根的求解，如下图：



如果方程有两个根那就是射线与圆相交，如果有一个根那就是射线与圆相切，如果一个根都没有就是不相交。还记得一元二次方程根的公式么， $x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$ ，有没有根就看 $b^2 - 4ac \geq 0$

这样我们在场景里面添加一个圆，位置为(0,0,-1) 半径是0.5的圆。做Raycast的时候，如果射线跟圆相交，直接返回红色，不相交还是跟上一章一样。下面就是与上一章不同的代码部分，完整的代码在chapter4.cs中。

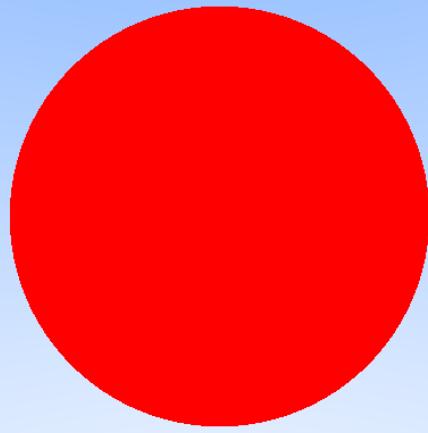
```

public static bool Hit_sphere(Vector3 center, float radius, Ray ray)
{
    Vector3 oc = ray.origin - center;
    float a = Vector3.Dot(ray.direction, ray.direction);
    float b = 2.0f * Vector3.Dot(oc, ray.direction);
    float c = Vector3.Dot(oc, oc) - radius * radius;
    float d = b * b - 4 * a * c;
    return d > 0;
}

public static Vector3 RayCast(Ray ray)
{
    if (Hit_sphere(center, radius, ray))
    {
        return ballColor;
    }
    Vector3 unit_direction = ray.direction.normalized;
    float t = 0.5f * (unit_direction.y + 1.0f);
    return Vector3.Lerp(topColor, bottomColor, t);
}

```

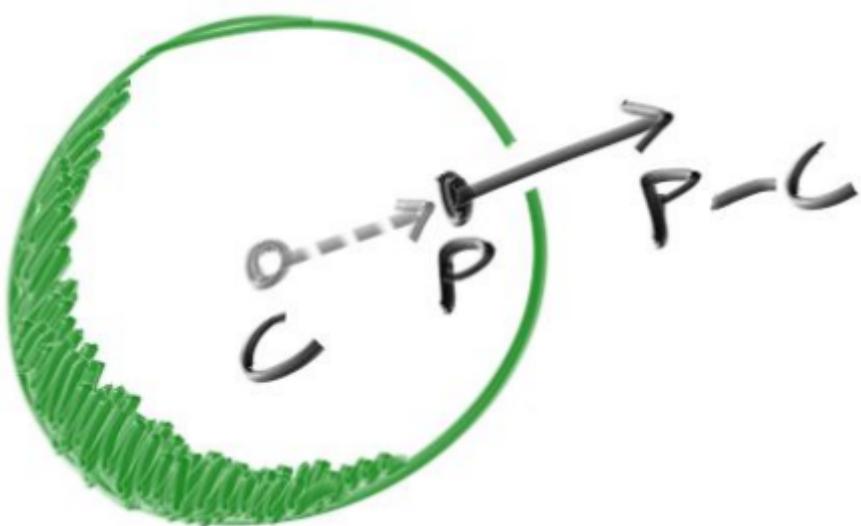
渲染结果如下：



虽然很丑，但是确实是渲染出来东西了。这里只计算了是否碰到了球，就统一给了个颜色，没有做着色，也没有计算光线遇到物体后的反射。后面让我们一点一点加上。

第五章 圆的表面法线和多个圆的渲染

首先，我们来讲一下什么是物体的法线，法线就是一个垂直于物体表面的向量。圆的法线很容易计算，用圆上任意一点的坐标，减去圆心的坐标，得到的向量就是该点的法向量。在应用中我们会使用法向量的单位向量，（单位向量就是方向和法向量一样，但是长度是1的向量）这样可以避免计算中的很多数值问题。如下图：



有了法向量，我们把它用在上次渲染的地方，现在的问题就是如果把单位的法向量变成一个颜色值。单位法向量的三个分量取值范围都是[-1,1]之间，这样我们把三个分量分别加上1，然后再乘以0.5,这样的三个分量就落在了[0,1]的区间，又正好是三个分量，我们直接把变换后的x,y,z 三个分量对于 r,g,b 的颜色分量，这样就把一个单位向量编码成了一个颜色值。看上去是不是有点随意，其实实时渲染中的 Normal Map 就是这样编码的。代码在 chapter5_1.cs 中，这里只贴跟上次不一样的地方。之前 Hit_sphere 函数只是返回圆是否和射线相交，这里会返回一个真正的根，这样就可以得到交点的具体坐标，从而计算出交点的法向量，然后用上面提到的方法，编码成一个颜色向量。

```
public static float Hit_sphere(Vector3 center, float radius, Ray ray)
{
    Vector3 oc = ray.origin - center;
    float a = Vector3.Dot(ray.direction, ray.direction);
    float b = 2.0f * Vector3.Dot(oc, ray.direction);
    float c = Vector3.Dot(oc, oc) - radius * radius;
    float d = b * b - 4 * a * c;
    if (d < 0)
    {
        return -1;
    }
    else
    {
        return (-b - Mathf.Sqrt(d)) / (2 * a);
    }
}

public static Vector3 RayCast(Ray ray)
{
    var t = Hit_sphere(center, radius, ray);
    if (t > 0)
    {
        Vector3 N = (ray.GetPoint(t) - new Vector3(0, 0, -1)).normalized;
        return 0.5f * (N + Vector3.one);
    }
    Vector3 unit_direction = ray.direction.normalized;
    t = 0.5f * (unit_direction.y + 1.0f);
    return Vector3.Lerp(topColor, bottomColor, t);
}
```

渲染结果如下：



下面我们来解决如何渲染多个圆的问题,其实也很简单,做法比较简单粗暴,就是用射线跟每一个球做碰撞,找出那个交点最近就行了。

```

public struct Hit_record
{
    public float t;
    public Vector3 hitpoint;
    public Vector3 normal;
};

public interface Hitable
{
    bool Hit(Ray r, ref float t_min, ref float t_max, out Hit_record record);
};

public class Sphere : Hitable
{
    public Vector3 center;
    public float radius;
    public Sphere()
    {
        center = Vector3.zero;
        radius = 1;
    }
    public Sphere(Vector3 c, float r)
    {
        center = c;
        radius = r;
    }
    public bool Hit(Ray ray, ref float t_min, ref float t_max, out Hit_record record)
    {
        //具体内容见chapter5_2.cs
    }
}

```

这里**Hit_record**用来记录交点的信息。**Hitable**是个抽象类，为了将来场景里面能渲染别的东西。我们把圆的一下方法类封装了变成一个**Sphere**类，这里判断碰撞的时候，传入了参数的最大最小范围，如果不范围内，这样就当是没有交点那样处理

```

public class HitList : Hitable
{
    private List<Hitable> list = new List<Hitable>();

    public HitList()
    {
    }

    public int GetCount()
    {
        return list.Count;
    }

    public void Add(Hitable item)
    {
        list.Add(item);
    }

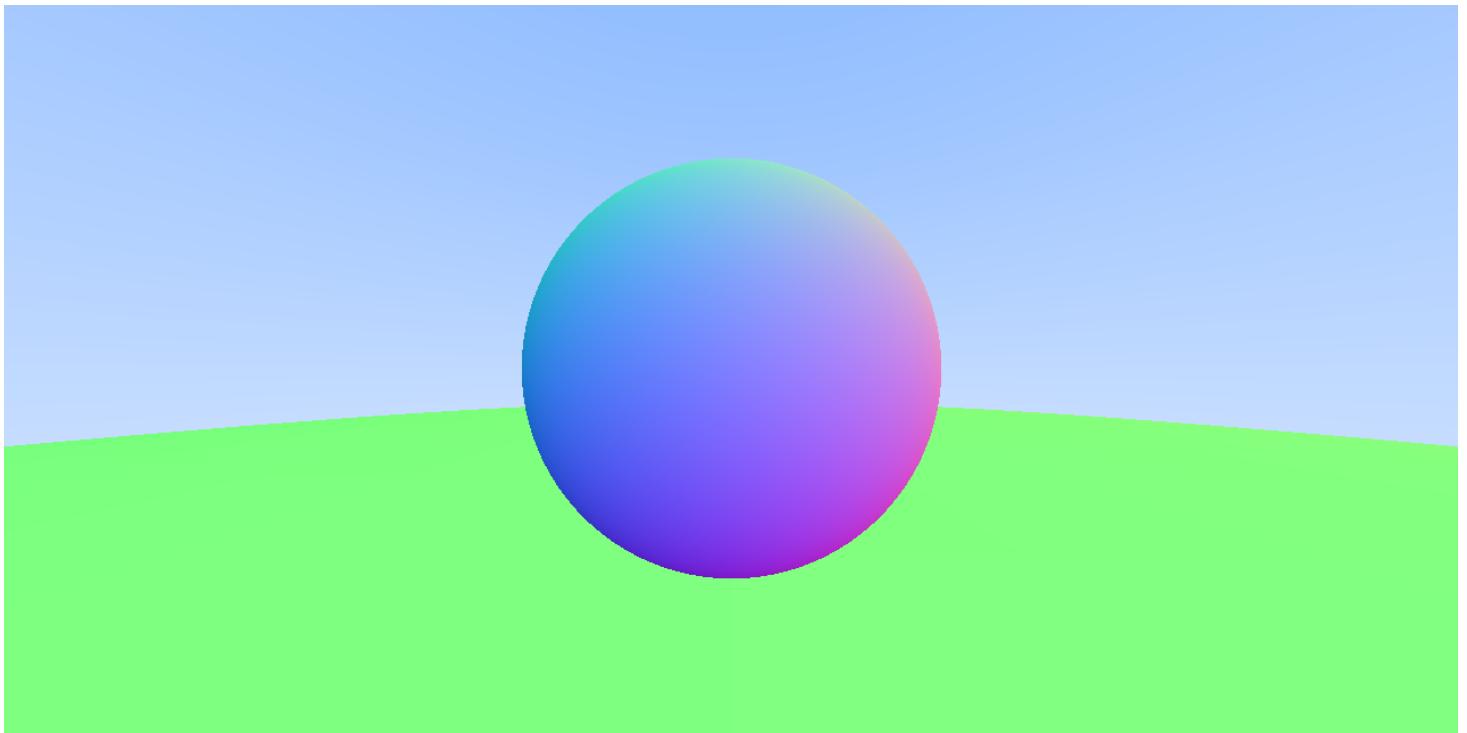
    public bool Hit(Ray r, ref float t_min, ref float t_max, out Hit_record record)
    {
        Hit_record temp_rec = new Hit_record();
        record = temp_rec;
        bool hit_anything = false;
        float closest_so_far = t_max;

        for (int i = 0; i < list.Count; ++i)
        {
            if (list[i].Hit(r, ref t_min, ref closest_so_far, out temp_rec))
            {
                hit_anything = true;
                closest_so_far = temp_rec.t;
                record = temp_rec;
            }
        }

        return hit_anything;
    }
}

```

HitList 用来存放场景里面所有可以渲染的物体，每次做射线检测的时候，就遍历一遍场景里面的物体，找出最近的碰撞点。剩下的代码可以参加chapter5_2.cs
最后渲染的结果为



下面那个绿色的地面，其实是一个半径很大的圆。

第六章 反走样

将上一张最后的图片放大，可以看到球的边缘，有很多锯齿。这个是因为，我们是用一根光线碰撞的结果代表了整个像素的值，实际上一个像素上可能是无数光线共同作用的结果。所以这里我们模拟这个过程，用近似的方法出来每个像素。我们在该像素上，随机产生一点偏离，保证光线还在像素格子里，这个发射多根光线，最好对结果做一个平均值。具体的代码在chapter6.cs中

```

public static void Main()
{
    int nx = 1280;
    int ny = 640;
    int ns = 64;

    RayCamera camera = new RayCamera();

    HitList list = new HitList();
    list.Add(new Sphere(new Vector3(0, 0, -1), 0.5f));
    list.Add(new Sphere(new Vector3(0, -100.5f, -1), 100));

    Texture2D tex = ImageHelper.CreateImg(nx, ny);
    for (int j = ny - 1; j >= 0; --j)
    {
        for (int i = 0; i < nx; ++i)
        {
            Vector3 color = Vector3.zero;
            for (int k = 0; k < ns; ++k)
            {
                float u = (float)(i + Random.Range(-1f, 1f)) / (float)(nx);
                float v = (float)(j + Random.Range(-1f, 1f)) / (float)(ny);

                Ray r = camera.GetRay(u, v);
                color += RayCast(r, list);
            }
            color = color / (float)(ns);
            ImageHelper.SetPixel(tex, i, j, color);
        }
    }
}

```

这里`ns`的值就是一个像素采样的次数，当然采样的次数越多，效果越好，不过计算量也越大。运行这段代码的时候，需要等一会。可以算一下计算量有多大，这里一共有 1280×640 的像素，一个像素需要采样64次，一次采样需要解2个一元二次方程组。也就是说，一共需要解 $1280 \times 640 \times 64 \times 2 = 104857600$ 个一元二次方程组。如果不想等那么久可以简单把图片改小一点。这次我们用到了`RayCamera`这个类

具体的实现在`camera.cs`中，也就是简单的封装了一下之前的方法。通过幕布的位置确定了摄像机，封装了`GetRay`方法，通过幕布的uv，来获得一根射线。

```

public class RayCamera
{
    private Vector3 origin;
    private Vector3 lower_left_corner;
    private Vector3 horizontal;
    private Vector3 vertical;

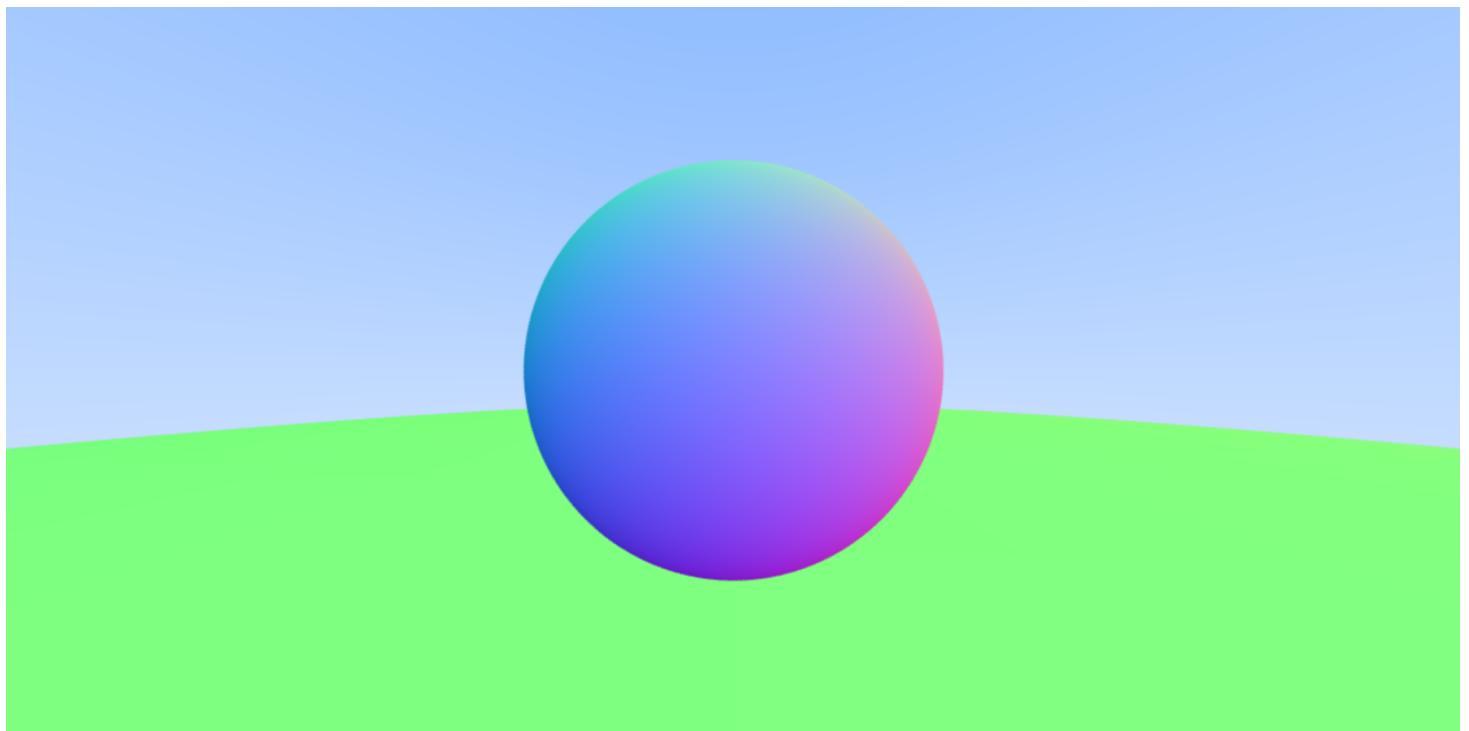
    public RayCamera()
    {
        origin = Vector3.zero;
        lower_left_corner = new Vector3(-2.0f, -1.0f, -1.0f);
        horizontal = new Vector3(4, 0, 0);
        vertical = new Vector3(0, 2, 0);
    }

    public RayCamera(Vector3 ori,Vector3 corner,Vector3 h,Vector3 v)
    {
        origin = ori;
        lower_left_corner = corner;
        horizontal = h;
        vertical = v;
    }

    public Ray GetRay(float u,float v)
    {
        return new Ray(origin, lower_left_corner + u * horizontal + v * vertical - origin);
    }
}

```

最终的效果如下：

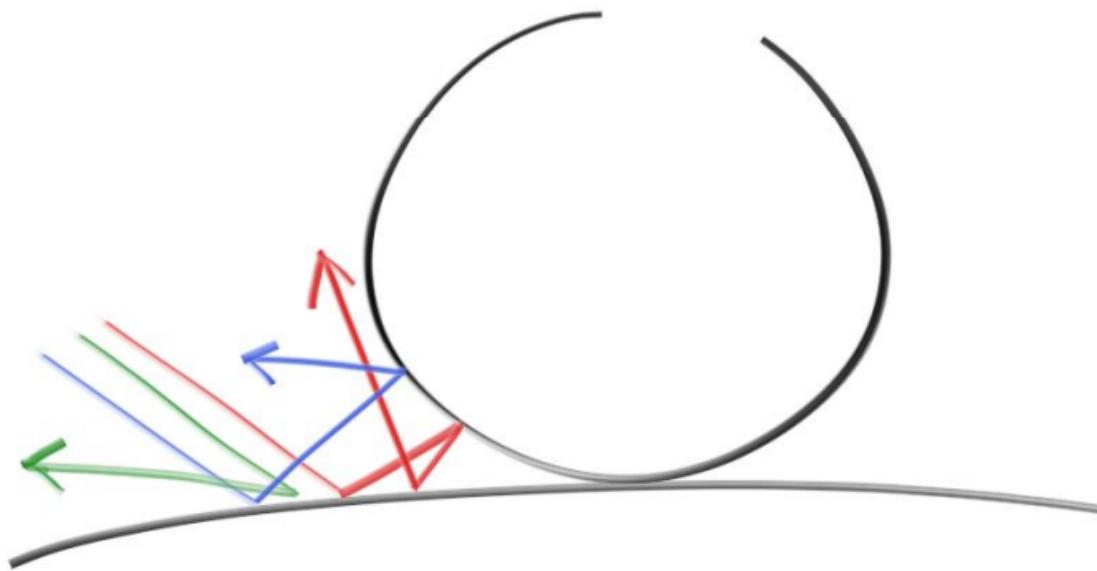


放大看边缘的对比，还是很明显的。

第七章 漫反射材质 (Diffuse Materials)

我们可以渲染多个物体并且每个像素可以用多个射线采样了，现在我们来尝试一下渲染一些看起来真实一点的材质。我们就先从漫反射（亚光）材质开始吧。在实现中有一个问题：我们应该把材质和物体分开，让一个物体可以有不同的材质。或者把材质和物体绑定在一起，在一些几何体和材质绑定的过程纹理实现比较方便。本教程采用的是把材质和物体分开的方式，也是大多数渲染器采用的方式，但是也要注意这种方式的限制。

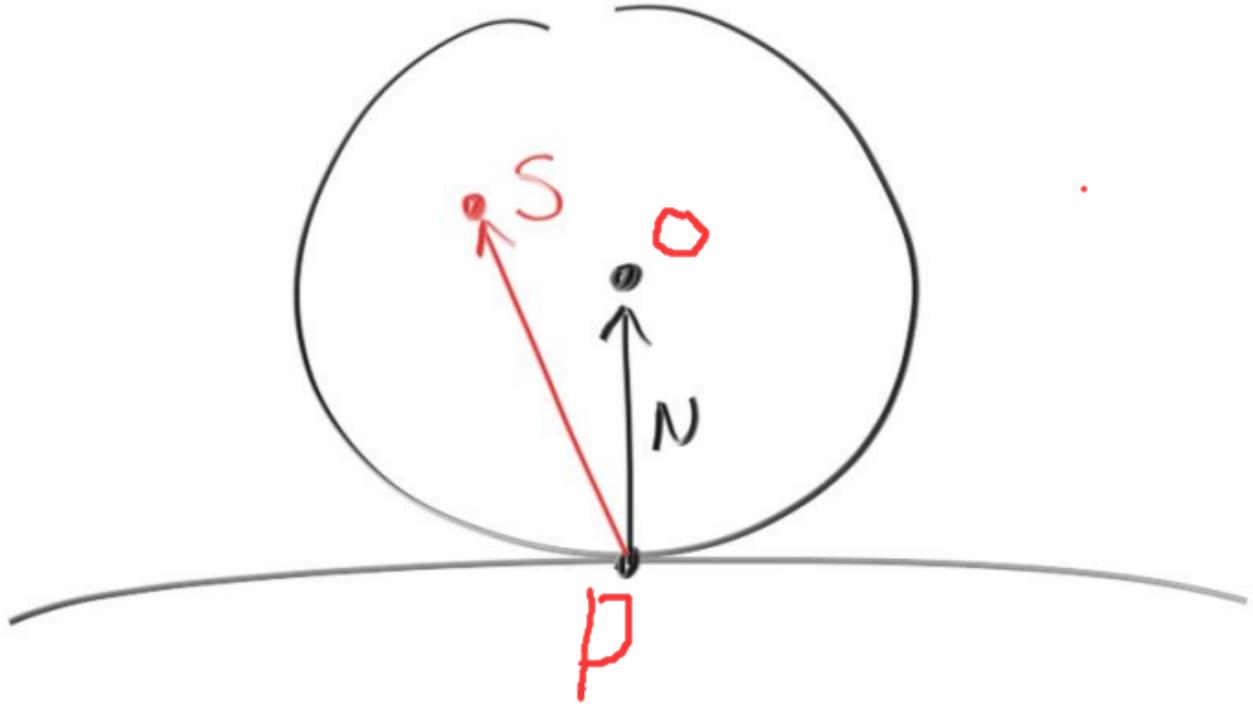
漫反射，是投射在粗糙表面上的光向各个方向反射的现象。拥有这种特性的材质，表面粗糙，不发光，他们在光线的照射下会体现出本身固有的颜色。因为反射光线的方向是随机的，所以如下图，三束平行的光线照射反射的结果非常不同：



到达漫反射的表面光线被吸收的要比被反射的多。因此颜色越深的物体，吸收的光线越多，这也是他们颜色深的原因。真随机方向算法，可以让物体表面看起来像磨砂的物体一样。

下面我们来实现这种随机的算法。在射线与圆相交的地方，做一个单位圆（半径是1的）并与之前的圆相切，随机在单位圆上取一点S，这样 $S-P$ 这个向量就可以看成是反射的方向。

$S = O + \vec{R}$ 这里 \vec{R} 就是 x,y,z 在 $[-1,1]$ 内随机在单位化一下（这里的实现跟原书不大一样，实测效果差不多）。所以 $S - P = P + \vec{N} + \vec{R} - P$, $S - P = \vec{N} + \vec{R}$



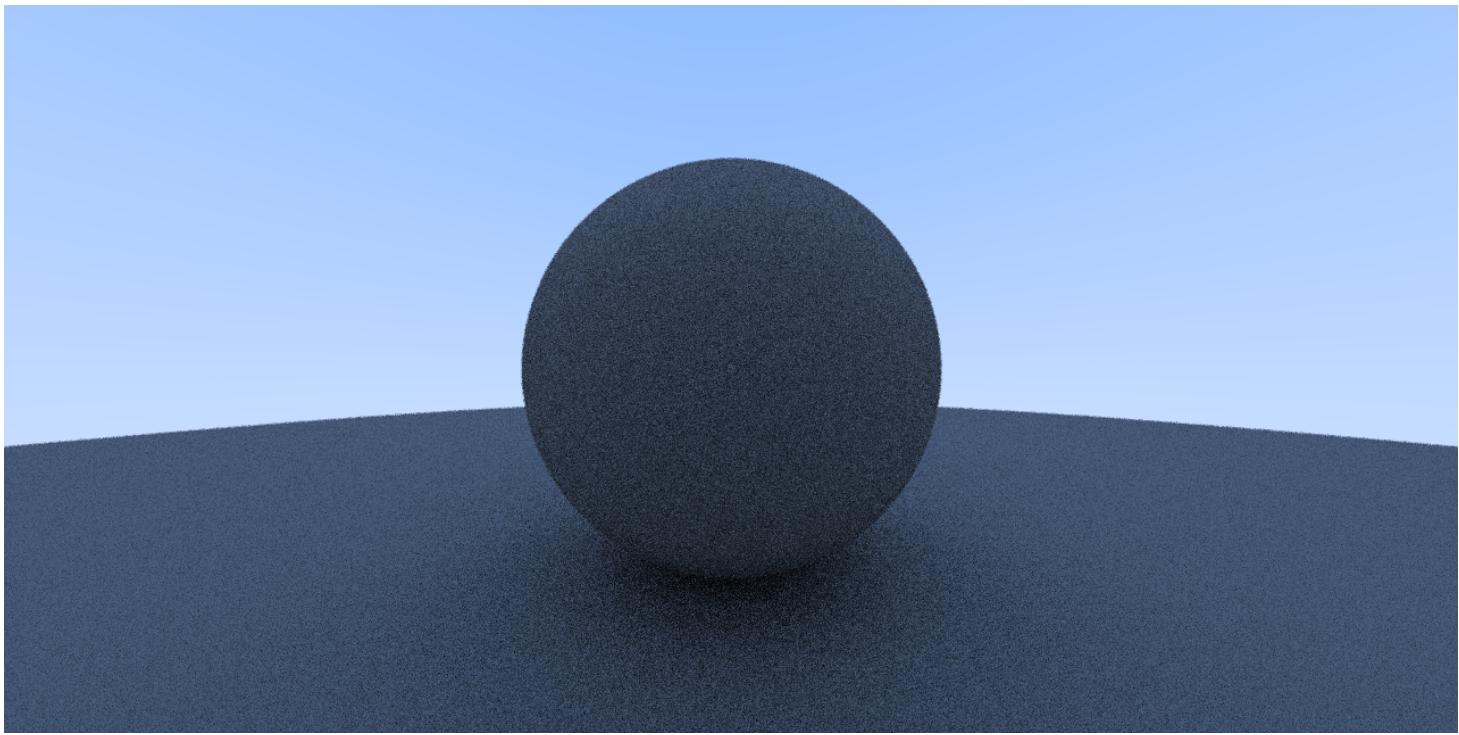
```

public static Vector3 RayCast(Ray ray, Hitable world)
{
    Hit_record rec;
    float min = 0;
    float max = float.MaxValue;
    if (world.Hit(ray, ref min, ref max, out rec))
    {
        var target = rec.normal.normalized +
            new Vector3(Random.Range(-1, 1f), Random.Range(-1f, 1f), Random.Range(-1f, 1f)).norrr

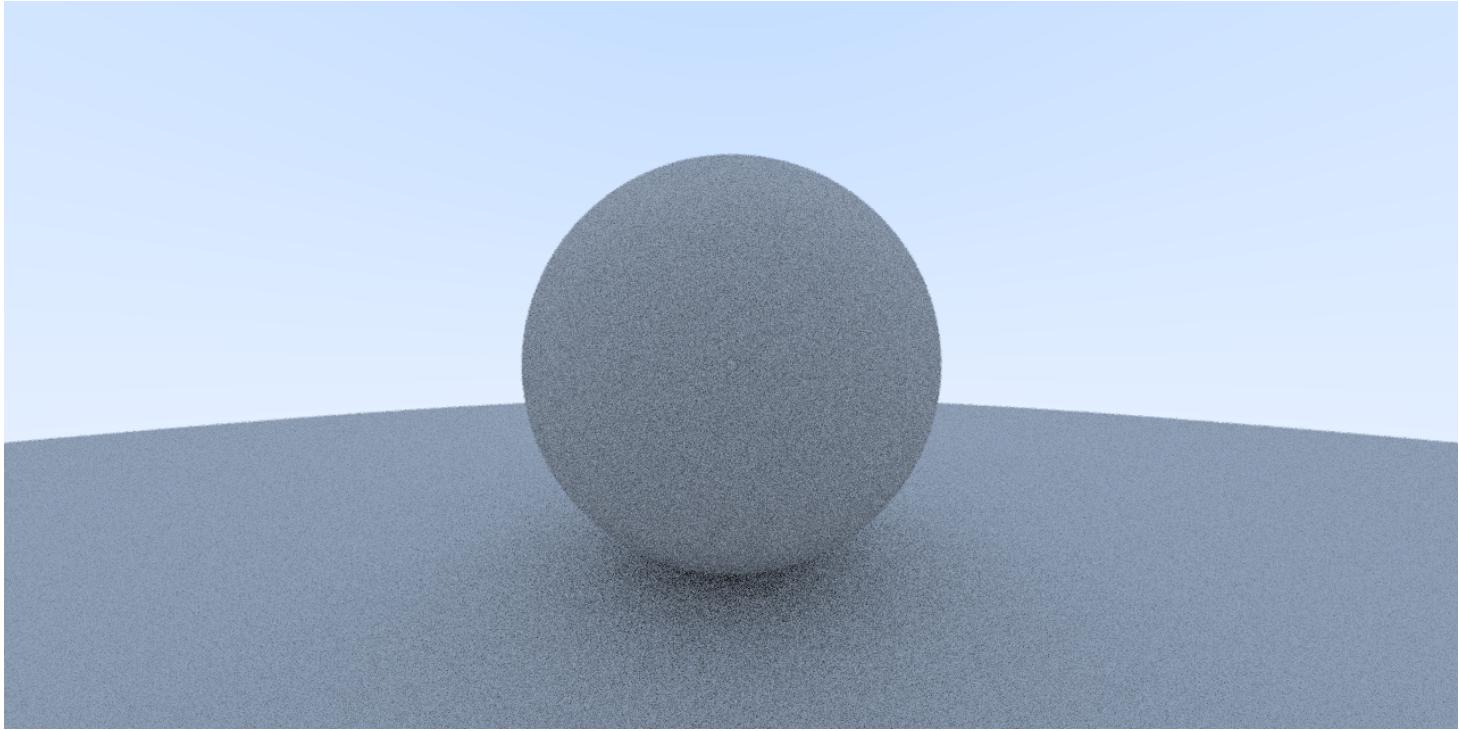
        return 0.5f * RayCast(new Ray(rec.hitpoint, target), world);
    }
    else
    {
        Vector3 unit_direction = ray.direction.normalized;
        float t = 0.5f * (unit_direction.y + 1.0f);
        return Vector3.Lerp(topColor, bottomColor, t);
    }
}

```

完整代码见chapter7.cs，给反射光线乘以0.5是假设光线有一半被吸收了。所以被光线反射次数越多的地方越暗，渲染结果如下图：



下图是我们如果给图像做gamma矫正后的情况，gamma矫正是因为人眼对光强的变化不是线性的，根据gamma曲线做了矫正。



第八章 金属材质

如果我们想要物体用不同的材质，这里需要给物体一个接口，就是设置其材质的接口。为此抽象一个材质的接口：

```

public interface IMaterial
{
    bool Scatter(ref Ray r, ref Hit_record rec, ref Vector3 attenuation, ref Ray scattered);
};

```

光接触到物体表面，有一部分被吸收，有一部分被反射，这里的Scatter函数主要用来表述不同的材质，对光的作用，有多少吸收的，有多少是反射了，反射的方向又是怎样的。

之前我们的Hit_record也需要交点表面的材质信息

```

public struct Hit_record
{
    public float t;
    public Vector3 hitpoint;
    public Vector3 normal;
    public IMaterial mat; // 添加的材质信息
};

```

把上一章的漫反射材质，封装一下写一个类：

```

public class Lambertian : IMaterial
{
    public Vector3 albedo;
    public float reflect;

    public Lambertian(Vector3 a, float r)
    {
        albedo = a;
        reflect = r;
    }

    public bool Scatter(ref Ray r, ref Hit_record rec, ref Vector3 attenuation, ref Ray scattered)
    {
        Vector3 target = rec.normal.normalized +
            new Vector3(Random.Range(-1, 1f), Random.Range(-1f, 1f), Random.Range(-1f, 1f)).norm

        scattered.origin = rec.hitpoint;
        scattered.direction = target;
        attenuation = albedo * reflect;
        return true;
    }
}

```

这里比原书在对反射光的处理上多了一个系数reflect，就是假设有一部分光会被吸，反射的光线变暗了，这样效果更真实一点。

现在我们来讨论一下光滑的金属材质：光在遇到光滑的金属表面后几乎都被反射了，反射服从镜面反射定律，因此对金属材质来说，只要求的反射光线的方向就可以了

原书的讲解不是很细致：这里放个链接感觉讲的要细一些：[反射向量](#)。因为我们用了Unity中的Vector3这个类，里面有方法直接求的反射向量。这里就直接用了。

```
public class MetalNoFuzz : IMaterial
{
    public Vector3 albedo;

    public MetalNoFuzz(Vector3 a)
    {
        albedo = a;
    }

    public bool Scatter(ref Ray r, ref Hit_record rec, ref Vector3 attenuation, ref Ray scat
    {
        Vector3 reflected = Vector3.Reflect(r.direction.normalized, rec.normal.normalized);

        scattered.origin = rec.hitpoint;
        scattered.direction = reflected;
        attenuation = albedo;
        return Vector3.Dot(scattered.direction, rec.normal) > 0;
    }
}
```

最后我们在场景里面添加4个球，2个金属材质的，2个Lambertian材质的，做渲染：

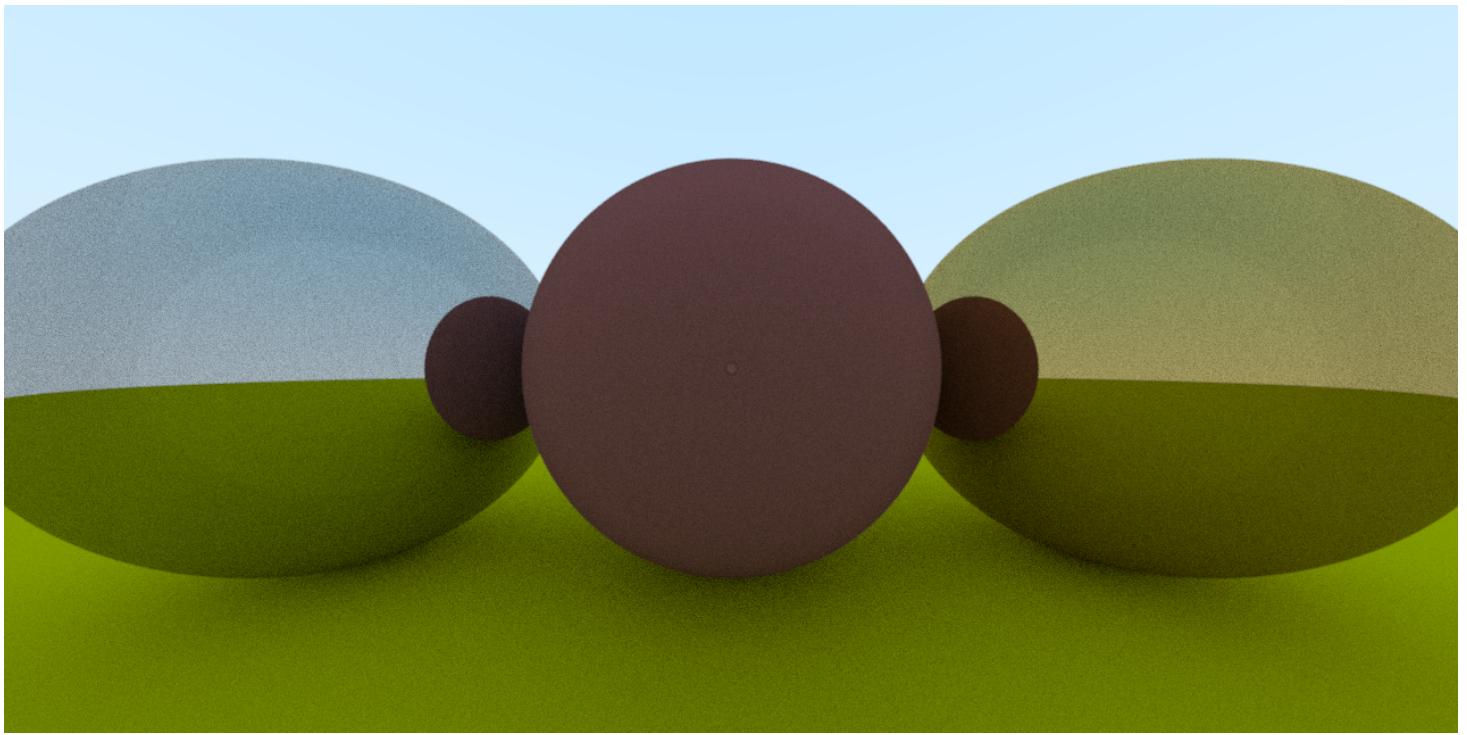
这里限定的光线最多反射50次

```
private static int MaxDepth = 50;
public static Vector3 RayCast(Ray ray, Hitable world, int depth)
{
    Hit_record rec;
    float min = 0;
    float max = float.MaxValue;
    if (world.Hit(ray, ref min, ref max, out rec))
    {
        Ray scattered = new Ray();
        Vector3 attenuation = Vector3.one;

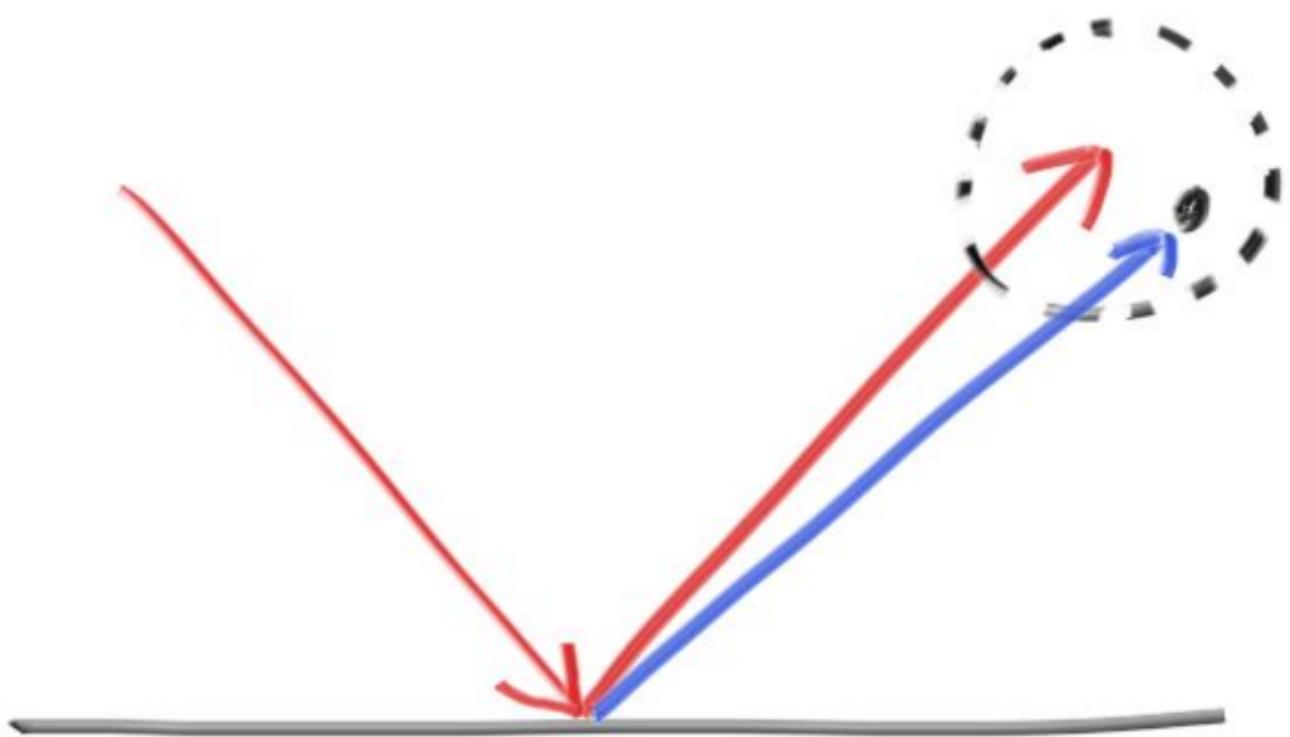
        if (depth < MaxDepth && rec.mat.Scatter(ref ray, ref rec, ref attenuation, ref scatterer))
        {
            var color = RayCast(scattered, world, depth + 1);
            attenuation.x *= color.x;
            attenuation.y *= color.y;
            attenuation.z *= color.z;
            return attenuation;
        }
        else
        {
            return Vector3.zero;
        }
    }
    else
    {
        Vector3 unit_direction = ray.direction.normalized;
        float t = 0.5f * (unit_direction.y + 1.0f);
        return Vector3.Lerp(topColor, bottomColor, t);
    }
}
```

完整代码在chapter8.cs中

渲染结果如下：



上面我们的金属材质完全是遵循了镜面反射，这样看上去像有点不真实。我们在反射的时候添加一个随机的偏移：



```

public class Metal : IMaterial
{
    public Vector3 albedo;
    public float fuzz;

    public Metal(Vector3 a, float f)
    {
        albedo = a;
        fuzz = f;
    }

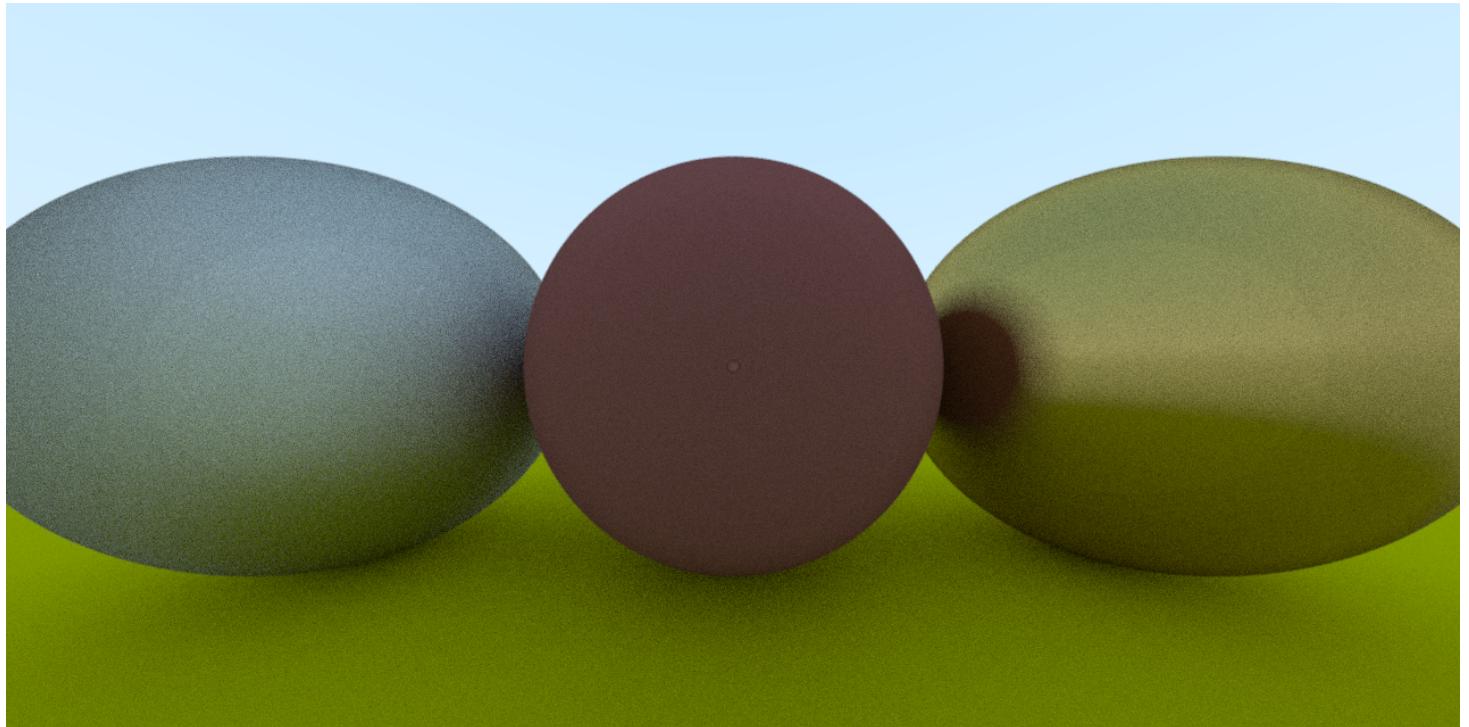
    private Vector3 Random_in_unit_sphere()
    {
        return new Vector3(Random.Range(-1f, 1f), Random.Range(-1, 1f), Random.Range(-1f, 1f));
    }

    public bool Scatter(ref Ray r, ref Hit_record rec, ref Vector3 attenuation, ref Ray scattered)
    {
        Vector3 reflected = Vector3.Reflect(r.direction.normalized, rec.normal.normalized);
        reflected = reflected + fuzz * Random_in_unit_sphere();

        scattered.origin = rec.hitpoint;
        scattered.direction = reflected;
        attenuation = albedo;
        return Vector3.Dot(scattered.direction, rec.normal) > 0;
    }
}

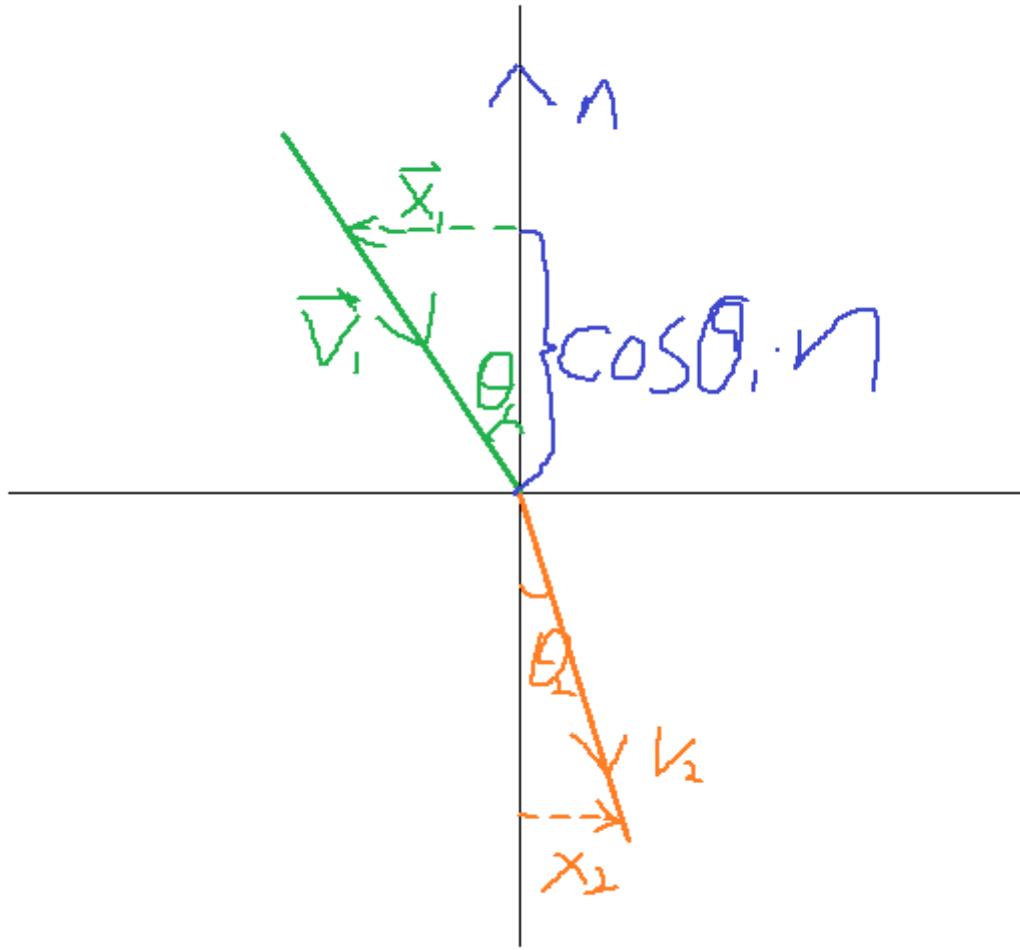
```

渲染结果如下：



第九章 透明材质

透明的材质比如水，玻璃，钻石都是Dielectrics（绝缘体），这类材质的特点是，光线通过的时候，一部分被反射，一部分被折射。折射的光线遵从斯涅尔定律： $n_1 \sin \theta_1 = n_2 \sin \theta_2$ 其中， n_1, n_2 分别是两种介质的折射率， θ_1, θ_2 分别是入射光、折射光与界面法线的夹角，分别叫做“入射角”、“折射角”。如下图



\vec{V}_1 是入射方向向量， \vec{V}_2 是折射方向， \vec{N} 是法向量。这里假设他们都是单位向量。已知的是： \vec{V}_1 和 \vec{N} 的方向， $\text{dot}(-\vec{V}_1, \vec{N})$ 就是 $\cos\theta_1$ 的值，还知道 $\frac{\sin\theta_1}{\sin\theta_2} = \frac{n_2}{n_1}$ ，现在我们来求 \vec{V}_2 ，（原书中省略了推导过程，这里我补全一下）由图像可知 $\vec{V}_2 = \cos\theta_2(-\vec{N}) + \vec{X}_2$ ，法向量 \vec{N} 已知，所以只要求的 \vec{X}_2 和 $\cos\theta_2$ 就可以了。先来求 \vec{X}_2 ，看上图可知 \vec{X}_2 与 \vec{X}_1 方向相反。向量的长度比： $\frac{\|\vec{X}_1\|}{\|\vec{V}_1\|} = \sin\theta_1$ ，因为 \vec{V}_1 是单位向量，长度是 1。由此可得 $\|\vec{X}_1\| = \sin\theta_1$ ，同理可得 $\|\vec{X}_2\| = \sin\theta_2$ ，因此有 $\frac{\|\vec{X}_2\|}{\|\vec{X}_1\|} = \frac{\sin\theta_2}{\sin\theta_1}$ 。现在来求 \vec{X}_1 ，根据向量的减法可得 $\vec{X}_1 = -\vec{V}_1 - \cos\theta_1 \vec{N}$ ，因为 \vec{X}_2 的方向与 \vec{X}_1 正好相反，所以可得 $\vec{X}_2 = -\vec{X}_1 * \frac{\sin\theta_2}{\sin\theta_1}$ ， $\vec{X}_2 = (\vec{V}_1 + \cos\theta_1 \vec{N}) * \frac{\sin\theta_2}{\sin\theta_1}$ ，带

入点积 $\vec{X}_2 = (\vec{V}_1 + \text{dot}(-\vec{V}_1, \vec{N})\vec{N}) * \frac{\sin\theta_2}{\sin\theta_1}$ 。这个式子里面就都是已知量了，现在来求 $\cos\theta_2$ 。
由 $\frac{\sin\theta_1}{\sin\theta_2} = \frac{n_2}{n_1}$ 可得 $\frac{\sqrt{1-\cos^2\theta_1}}{\sqrt{1-\cos^2\theta_2}} = \frac{n_2}{n_1}$ ，两边平方后整理可得 $\cos^2\theta_2 = 1 - (\frac{n_1}{n_2})^2(1 - \cos^2\theta_1)$ 。
 $\vec{V}_2 = \vec{X}_2 + \cos\theta_2 * (-\vec{N})$, $\vec{V}_2 = \vec{X}_2 - \cos\theta_2\vec{N}$ 只要上面的 \vec{X}_2 和 $\cos\theta_2$ 带入即可。

```
private bool Refract(Vector3 v, Vector3 n, float ni_over_nt, out Vector3 refracted)
{
    v.Normalize();
    n.Normalize();
    float dt = Vector3.Dot(-v.normalized, n.normalized);
    float discriminant = 1.0f - ni_over_nt * ni_over_nt * (1.0f - dt * dt);
    if (discriminant > 0)
    {
        refracted = ni_over_nt * (v.normalized + n * dt) - n * Mathf.Sqrt(discriminant);
        return true;
    }
    refracted = Vector3.one;
    return false;
}
```

注释：这里的实现跟原书有第一点区别，是在求 dt 的时候，原书直接求的 \vec{V}_1 和 \vec{N} 直接的夹角，这个夹角正好是 $\pi - \theta_1$ ，因此原书的 $dt = \cos(\pi - \theta_1)$ ，又因为 $\cos(\pi - \theta) = -\cos\theta$ ，所以我们这里的 dt 跟原书的正好差一个符号，所以我们这里的计算`refracted`向量的时候， $n * dt$ 的地方是跟原书差一个符号的。

下面是Dielectric材质的散射函数，这里我们先假设，光线要么反射要么折射，当然这是不科学的，除非发生全反射。这里我们为了简化，就先这样假定吧。先计算入射光线跟法线的夹角，如果大于90度，那么就是光线从玻璃到空气，如果小于90度就是从空气入射到玻璃。然后我们计算Refract的方向，注意这里Refract函数有个返回值，这个值就是看是否会发生全反射，如果发生了全反射，那么就不会有折射光线。

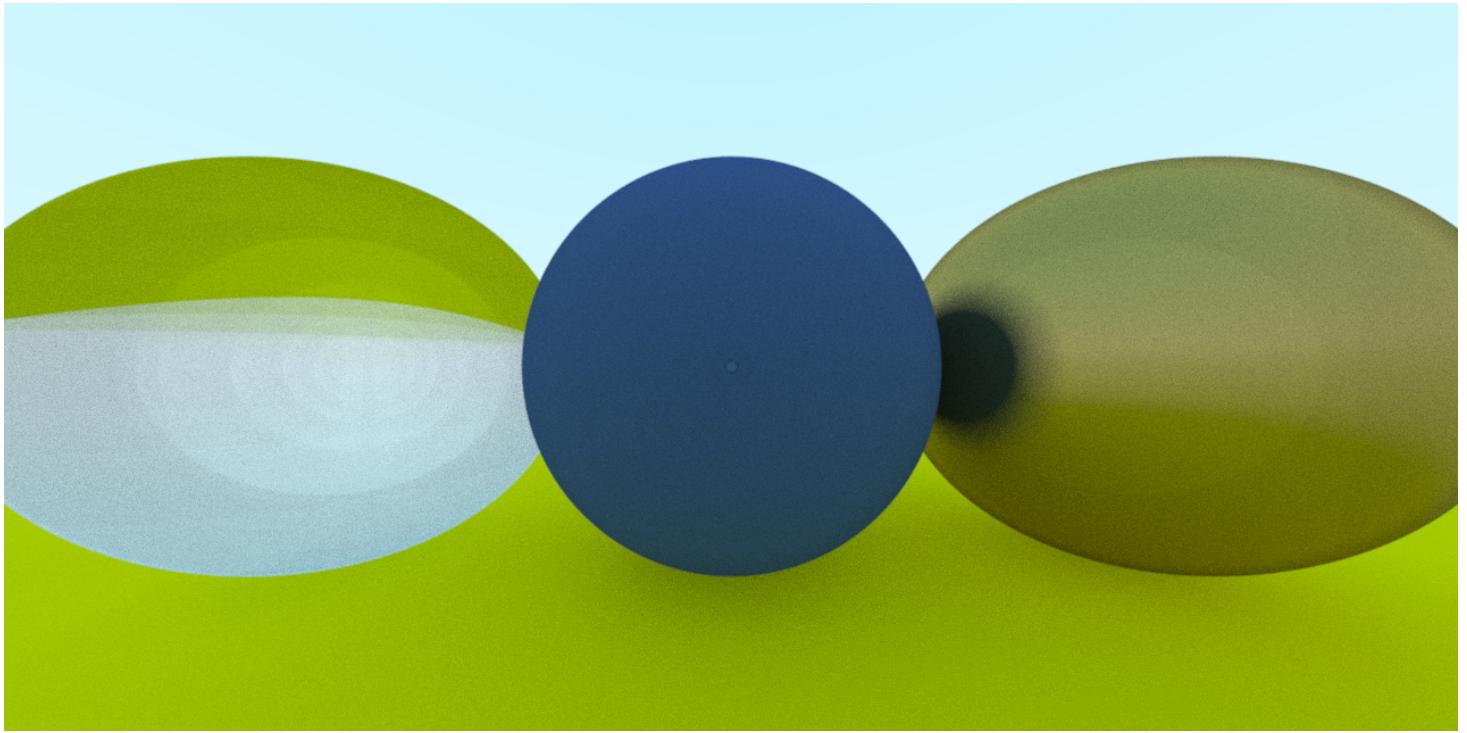
```

public bool Scatter(ref Ray r, ref Hit_record rec, ref Vector3 attenuation, ref Ray scattered)
{
    Vector3 outward_normal = Vector3.zero;
    Vector3 reflected = Vector3.Reflect(r.direction.normalized, rec.normal.normalized);
    float ni_over_nt = 0f;
    attenuation.x = 1.0f;
    attenuation.y = 1.0f;
    attenuation.z = 1.0f;
    Vector3 refracted;
    if (Vector3.Dot(r.direction, rec.normal) > 0)
    {
        outward_normal = -rec.normal;
        ni_over_nt = ref_idx;
    }
    else
    {
        outward_normal = rec.normal;
        ni_over_nt = 1.0f / ref_idx;
    }

    if (Refract(r.direction, outward_normal, ni_over_nt, out refracted))
    {
        scattered.origin = rec.hitpoint;
        scattered.direction = refracted;
        return true;
    }
    else
    {
        scattered.origin = rec.hitpoint;
        scattered.direction = reflected;
        return false;
    }
}

```

最终图片如下：



现在来让我们修正一下之前的假设，让材质看起来更真实。之前我们假设是光线要么反射要么折射，但是如果没有发生全反射的时候，其实是折射和反射同时发生的，也就是说光线入射玻璃之后，有一部分光反射了，有一部分光折射了。光线反射的比例一般跟入射的角度和物体本身的折射率有关。有人给出了一个拟合的公式： $R(\theta) = R_0 + (1 - R_0)(1 - \cos\theta)^5$ ，这里 $R_0 = (\frac{n_1 - n_2}{n_1 + n_2})^2$ $n_1 n_2$ 分别是两种介质的折射率。这样我们可以通过计算反射的概率来决定光线是折射还是反射。

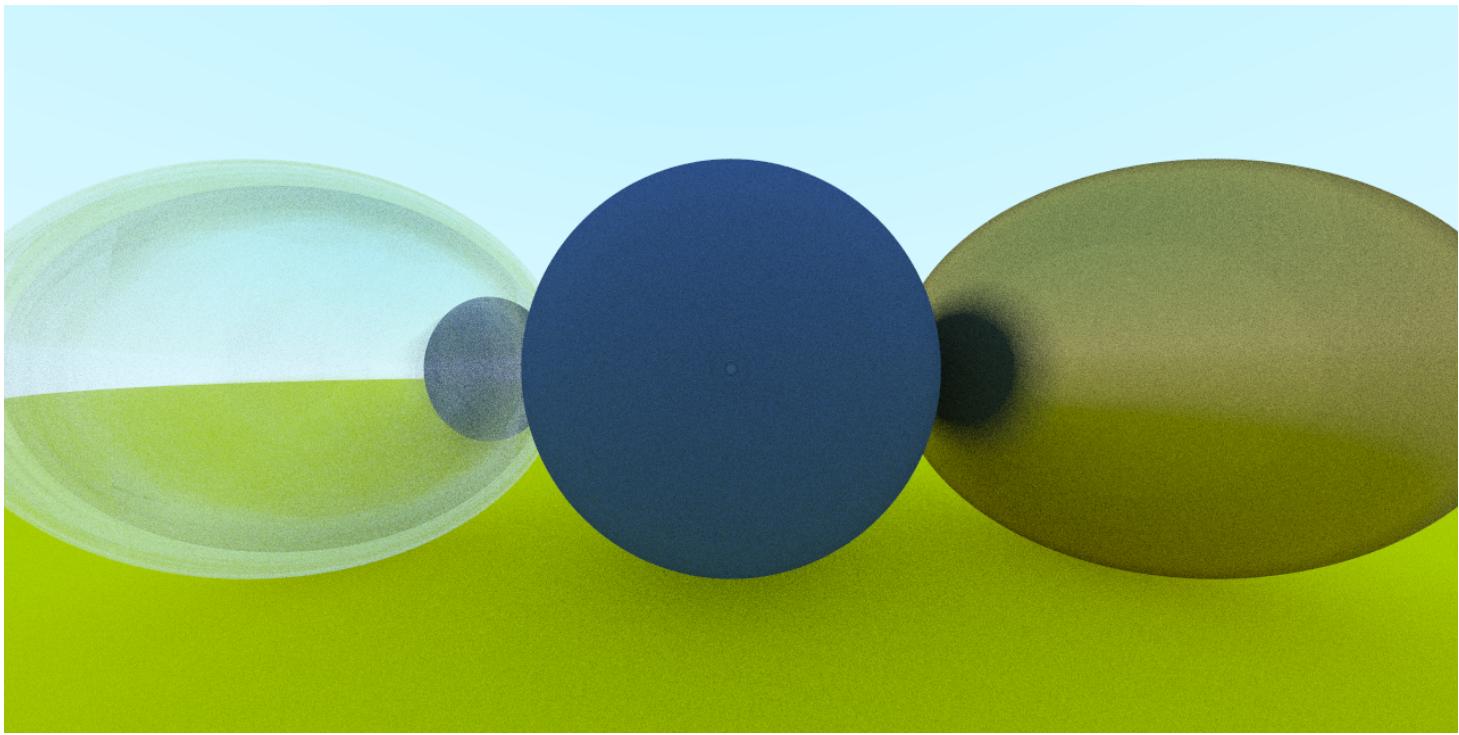
```

public bool Scatter(ref Ray r, ref Hit_record rec, ref Vector3 attenuation, ref Ray scattered)
{
    Vector3 outward_normal = Vector3.zero;
    Vector3 reflected = Vector3.Reflect(r.direction.normalized, rec.normal.normalized);
    float ni_over_nt = 0f;
    attenuation.x = 1.0f;
    attenuation.y = 1.0f;
    attenuation.z = 1.0f;
    Vector3 refracted;
    float reflect_prob;
    float cosine;
    if (Vector3.Dot(r.direction, rec.normal) > 0)
    {
        outward_normal = -rec.normal;
        ni_over_nt = ref_idx;
        cosine = ref_idx * Vector3.Dot(r.direction, rec.normal) / r.direction.magnitude;
    }
    else
    {
        outward_normal = rec.normal;
        ni_over_nt = 1.0f / ref_idx;
        cosine = -Vector3.Dot(r.direction.normalized, rec.normal) / r.direction.magnitude;
    }

    var bRefracted = Refract(r.direction, outward_normal, ni_over_nt, out refracted);
    if (bRefracted)
    {
        reflect_prob = Schlick(cosine, ref_idx);
    }
    else
    {
        scattered.origin = rec.hitpoint;
        scattered.direction = reflected;
        reflect_prob = 1.0f;
    }
    if (Random.Range(0, 1) < reflect_prob)
    {
        scattered.origin = rec.hitpoint;
        scattered.direction = reflected;
    }
    else
    {
        scattered.origin = rec.hitpoint;
        scattered.direction = refracted;
    }
    return true;
}

```

这里我们做两个玻璃球套在一起的，渲染结果如下：



第十章 可放置的摄像机

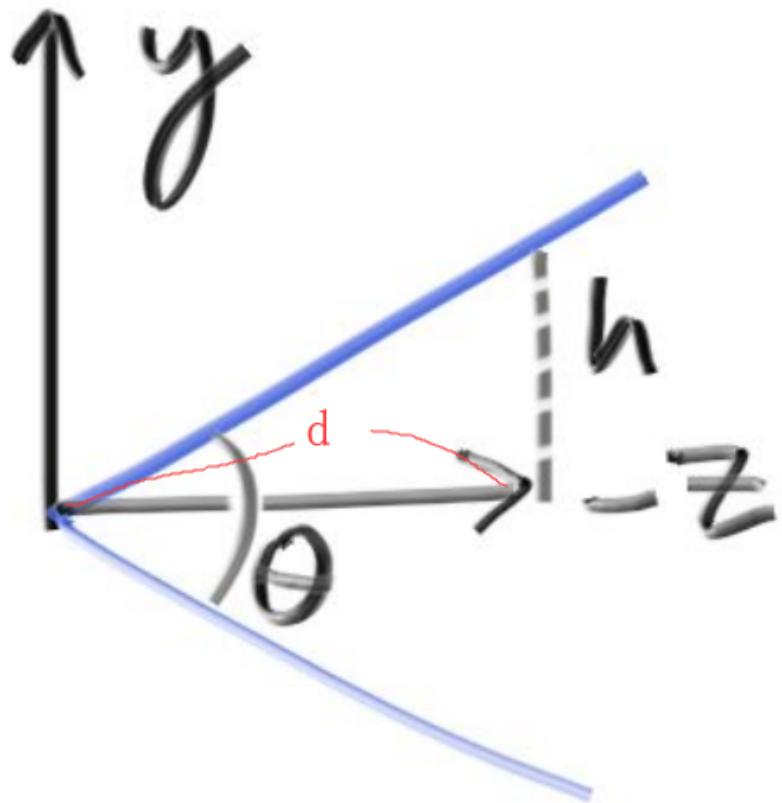
摄像机，跟dielectrics（透明的材质）一样，出bug特别不好调试。所以我常常是采用增量的方式开发。

首先，我们来做一个可以调节

fov（视野-field of view）的摄像机。其实**fov**是有分水平的和垂直的，这里我们用垂直的**fov**，来决定图像的高，用指定图像的长宽比，来确定图像的宽。这个只是个人的选择问题，没有特殊的道理，就跟坐标的左右手系一样。

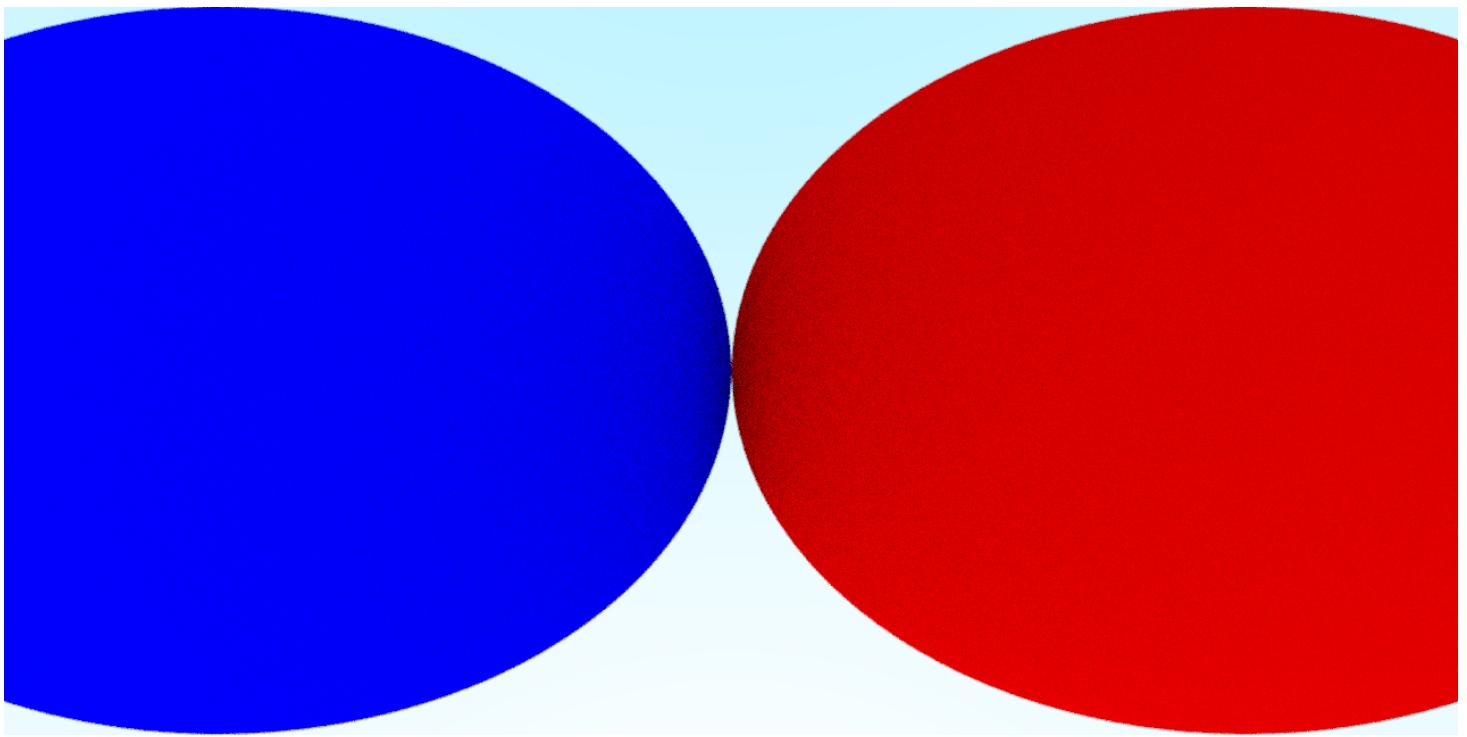
fov是射线画面从射线最高和最低的夹角，给定的**fov**角度如 θ ,那么画布最后的高 $\frac{h}{2} = d * \tan(\frac{\theta}{2})$ 这里我们固定 $d = 1$, 根据长宽比，就可以计算出w。

如下图所示：

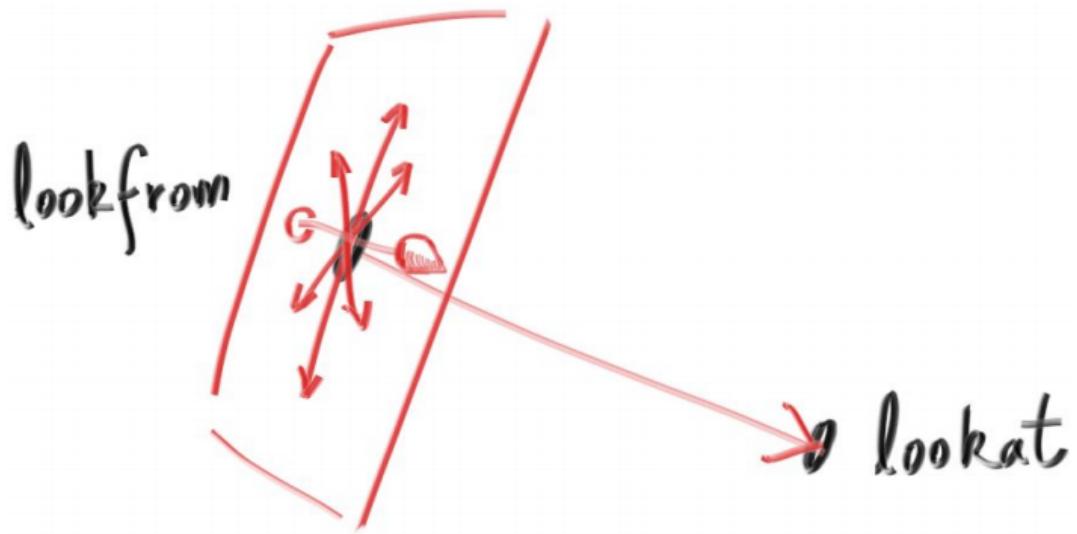


```
public RayCamera(float fov, float aspect)
{
    float theta = Mathf.Deg2Rad * fov;
    float half_height = Mathf.Tan(theta * 0.5f);
    float half_width = aspect * half_height;
    lower_left_corner = new Vector3(-half_width, -half_height, -1.0f);
    horizontal = new Vector3(2 * half_width, 0, 0);
    vertical = new Vector3(0, 2 * half_height, 0);
    origin = Vector3.zero;
}
```

现在我们来构造一个简单的场景，代码参见chapter10中Main方法：

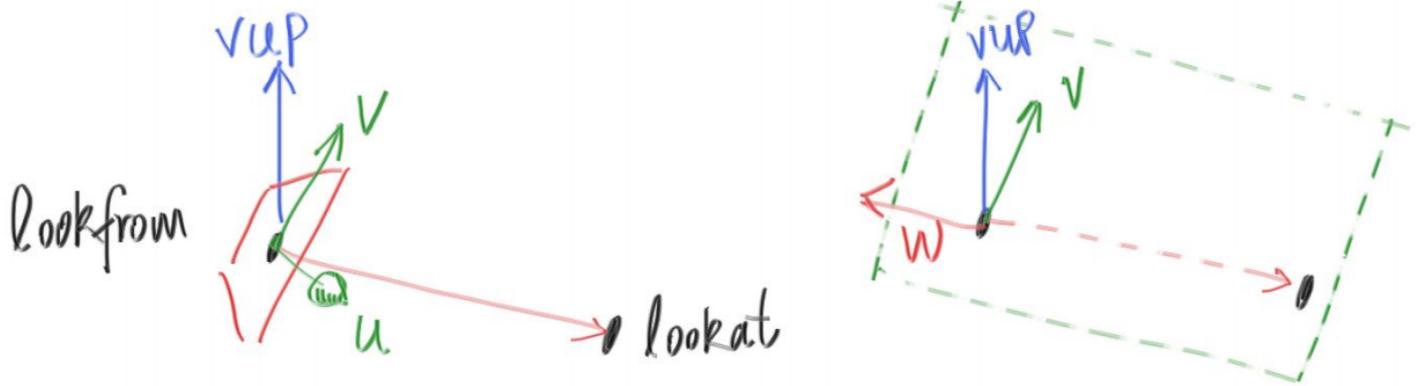


现在获得了可以修改fov的摄像机，有兴趣的可以改改摄像机参数，重新渲染一下，看看结果。但是我们的摄像机还是固定在原点，朝向z轴的负方向。我们现在来做一个可以在更加自由的摄像机，可以放置在任何一个地方，也可以朝向任何一个地方。这里我们假定摄像机所在的点叫lookfrom，摄像机的视点叫lookat，如下图：



这里虽然确定了摄像机的位置和朝向，但是摄像机本身可以在其所在平面内任意旋转，这样我们就需要规定，摄像机向上的方向。把摄像机固定在红色的平面内。我们指定相机朝上的方向vup，因为vup和v在同一平面内，所以我们对w 和vup做叉乘，就可以得到向量u，在用向量w和u做叉乘就可以得到向量

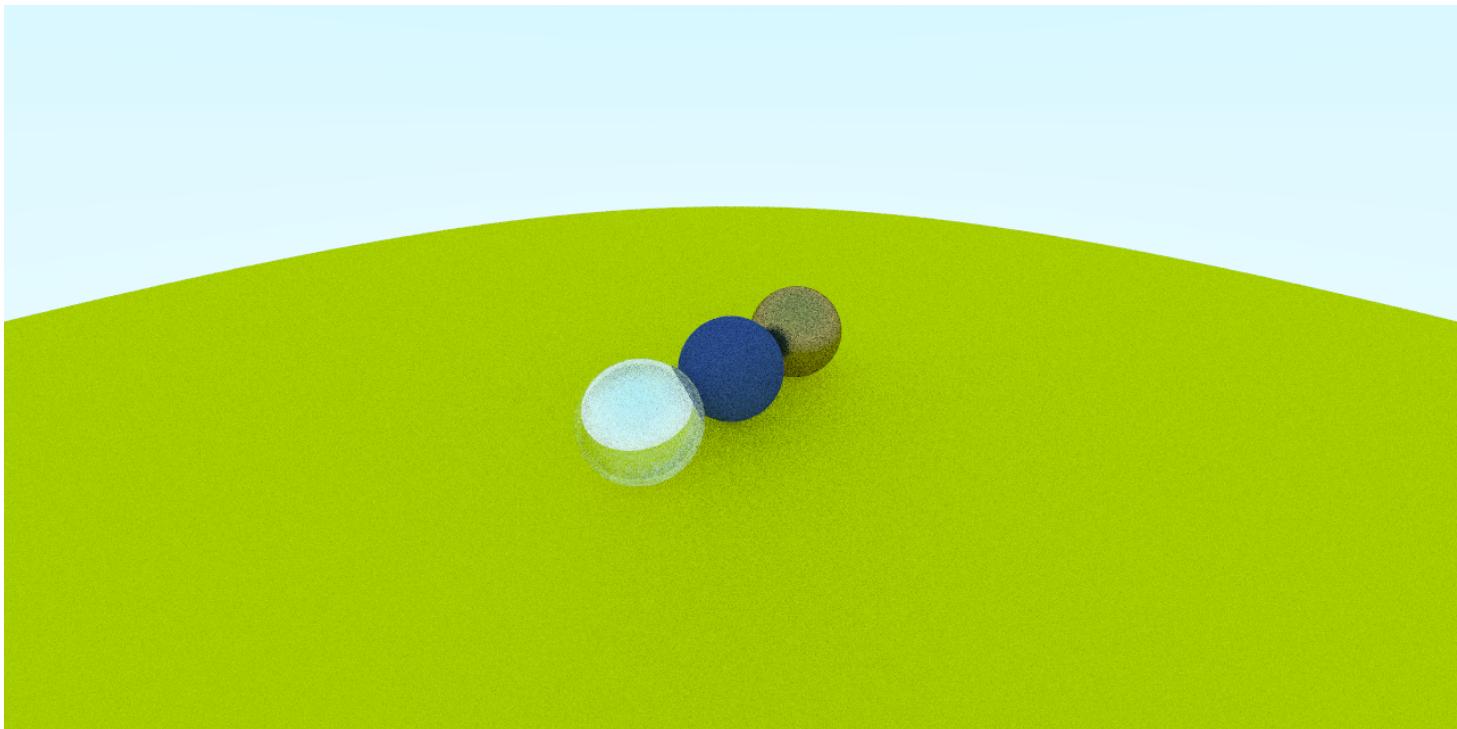
v。向量w可以通过`lookfrom - lookat`获得。补充一下向量叉乘的几何意义，对u,v叉乘，就是过uv的交点，做一个向量通过其交点又可以垂直于uv所在的平面，方向应该服从右手法则。参加下图：



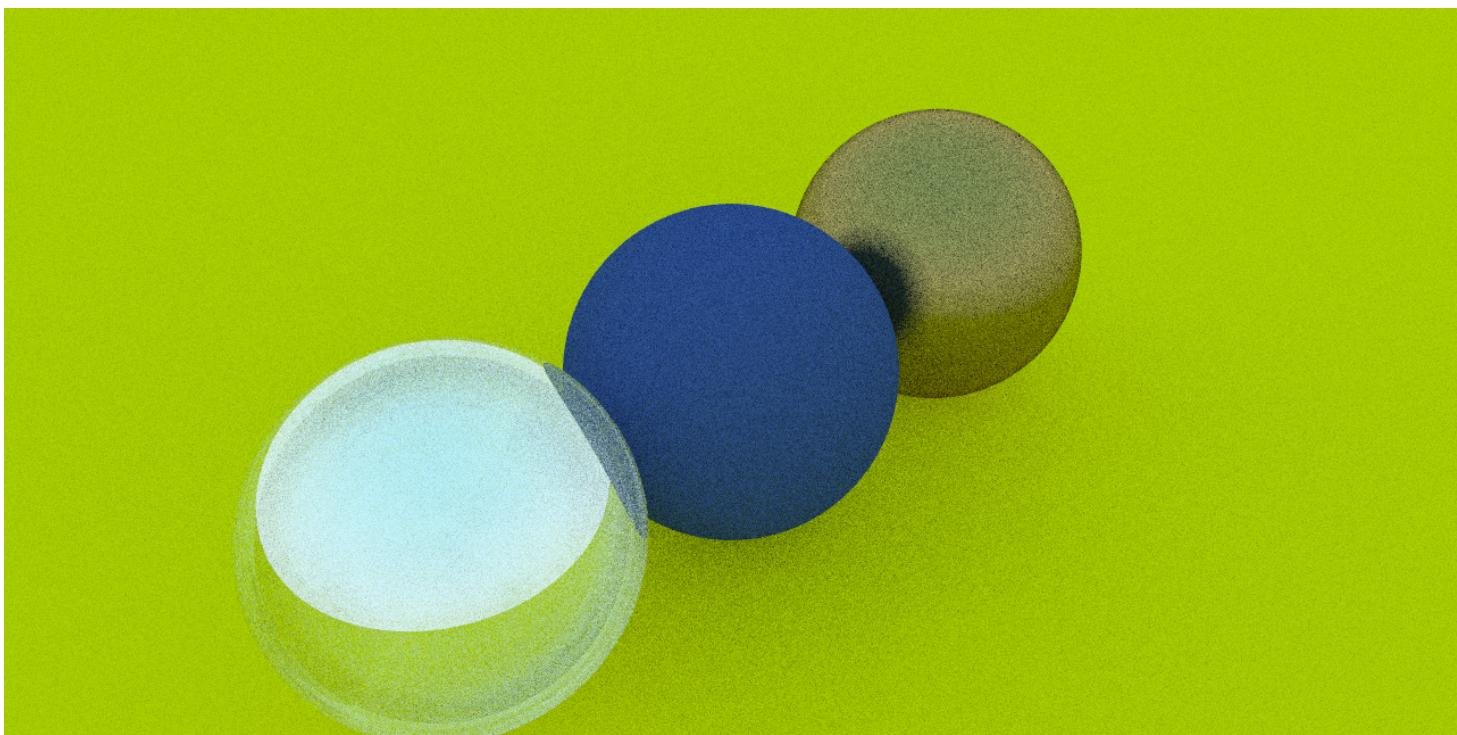
```
public RayCamera(Vector3 lookfrom,Vector3 lookat,Vector3 vup, float fov, float aspect)
{
    Vector3 u, v, w;
    float theta = Mathf.Deg2Rad * fov;
    float half_height = Mathf.Tan(theta * 0.5f);
    float half_width = aspect * half_height;
    origin = lookfrom;
    w = (lookfrom - lookat).normalized;
    u = Vector3.Cross(vup, w).normalized;
    v = Vector3.Cross(w, u);

    lower_left_corner = new Vector3(-half_width, -half_height, -1.0f);
    lower_left_corner = origin - half_width * u - half_height * v - w;
    horizontal = 2 * half_width * u;
    vertical = 2 * half_height * v;
}
```

代码参见chapter10中Main2方法：



修改fov后可得：



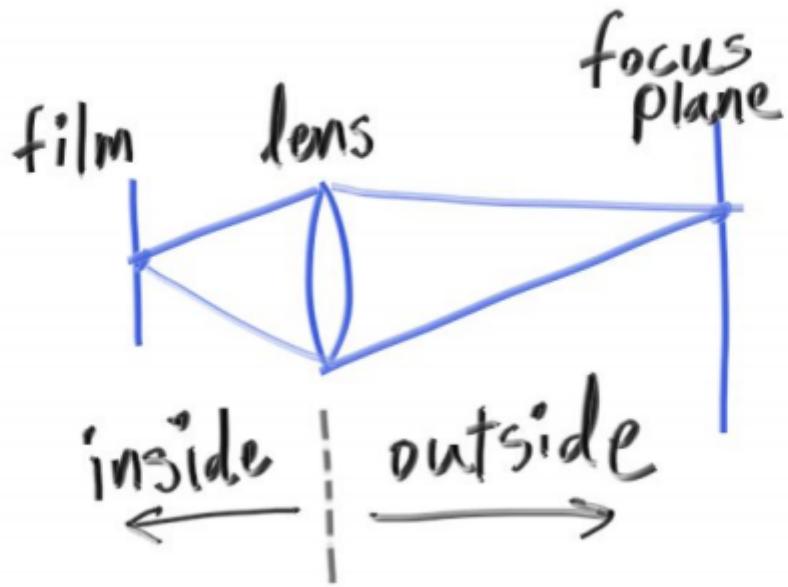
第十一章 景深(Defocus Blur)

渲染中**Defocus Blur**, 直译为焦外模糊, 平常大家都不这么说这里就用摄像中的一个词就是景深代替。景深效果的产生是因为在现实世界中的像机采集光线的时候, 需要一个大一点的孔, 而不是我们假设的一个小小的孔, 这样就会让所有的东西都模糊。如果我们在光线通过的地方放置一个凸透镜的话, 就可以让处在特定距离上的物体聚焦。这个距离是透镜和底片直接的距离共同决定的。光圈 (aperture) 就

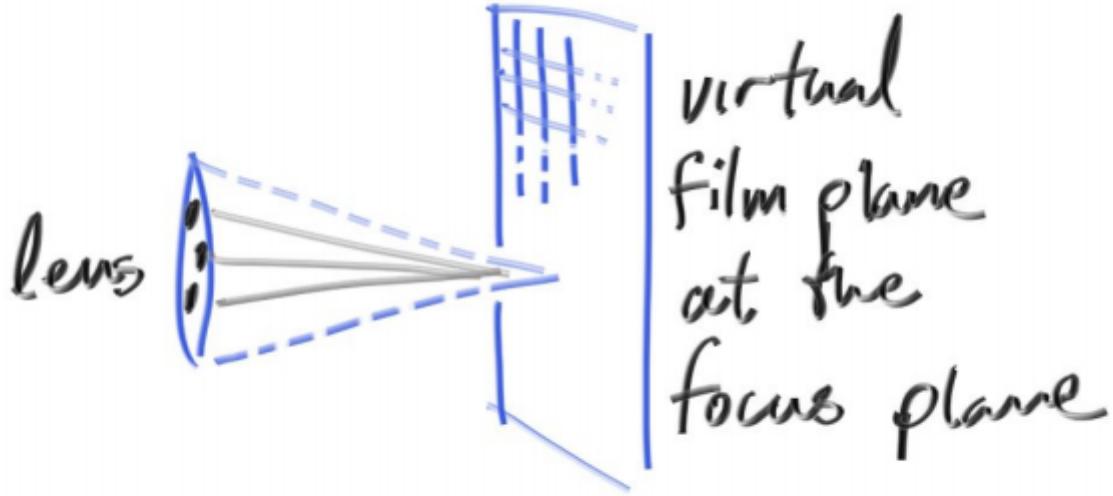
是光线通过的孔，孔越大单位时间内可以通过的光线也就越多。对真正的相机来说，光圈越大，那么景深效果越明显。（就是背景虚化）。对于我们渲染中的虚拟摄像机来说，虚拟的感光元器件足够的灵敏，不需要把孔做的比较大就可以清晰的成像，因此其实虚拟摄像机只有在需要模拟景深效果的时候，才需要一个光圈大小的参数。

可变焦的镜头其实是有多组透镜组成的，一般来说比较复杂。代码模拟的时候只考虑：底片，透镜，光圈大小

并且需要找出从哪里发射光线还有计算反转才能得到图片（因为透镜会在底片上产生上下颠倒的图）。做图形的人通常会用一个很薄的透镜近似。



我们不需要模拟相机内部的情况。因为渲染的图片的时候，光线追踪都是发生在摄像机外面的，模拟相机内部对渲染来说太复杂也没必要。所以我通常从透镜上随机的一点发射光线到一个虚拟的幕布上，通过投影来确定在焦距上的物体是清晰的。



代码在camera.cs中

```

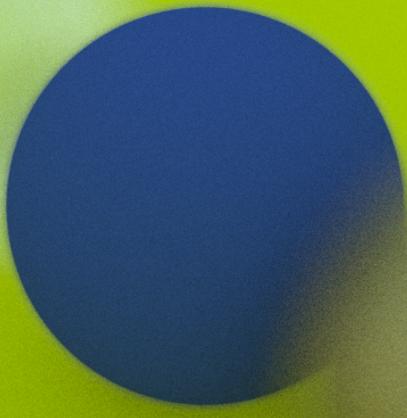
public MotionBlurRayCamera(Vector3 lookfrom, Vector3 lookat, Vector3 vup, float fov, float aspect
{
    lens_radius = aperture / 2;
    float theta = Mathf.Deg2Rad * fov;
    float half_height = Mathf.Tan(theta * 0.5f);
    float half_width = aspect * half_height;
    origin = lookfrom;
    w = (lookfrom - lookat).normalized;
    u = Vector3.Cross(vup, w).normalized;
    v = Vector3.Cross(w, u);
    focus = focus_dist;

    lower_left_corner = origin - half_width * focus_dist * u - half_height * focus_dist * v - focus;
    horizontal = 2 * half_width * focus_dist * u;
    vertical = 2 * half_height * focus_dist * v;
}

public Ray GetRay(float s, float t)
{
    Vector2 rd = lens_radius * Random_in_unit_disk();
    Vector3 offset = u * rd.x + v * rd.y;
    return new Ray(origin + offset, lower_left_corner + s * horizontal + t * vertical - origin -
}

```

在chapter11中，我们用光圈是2.0的相机拍摄场景，效果如下：



第十二章 接下来呢

让我们把前面学到的东西，放在一起，做一个有很多随机球的场景,代码在chapter12.cs中

```

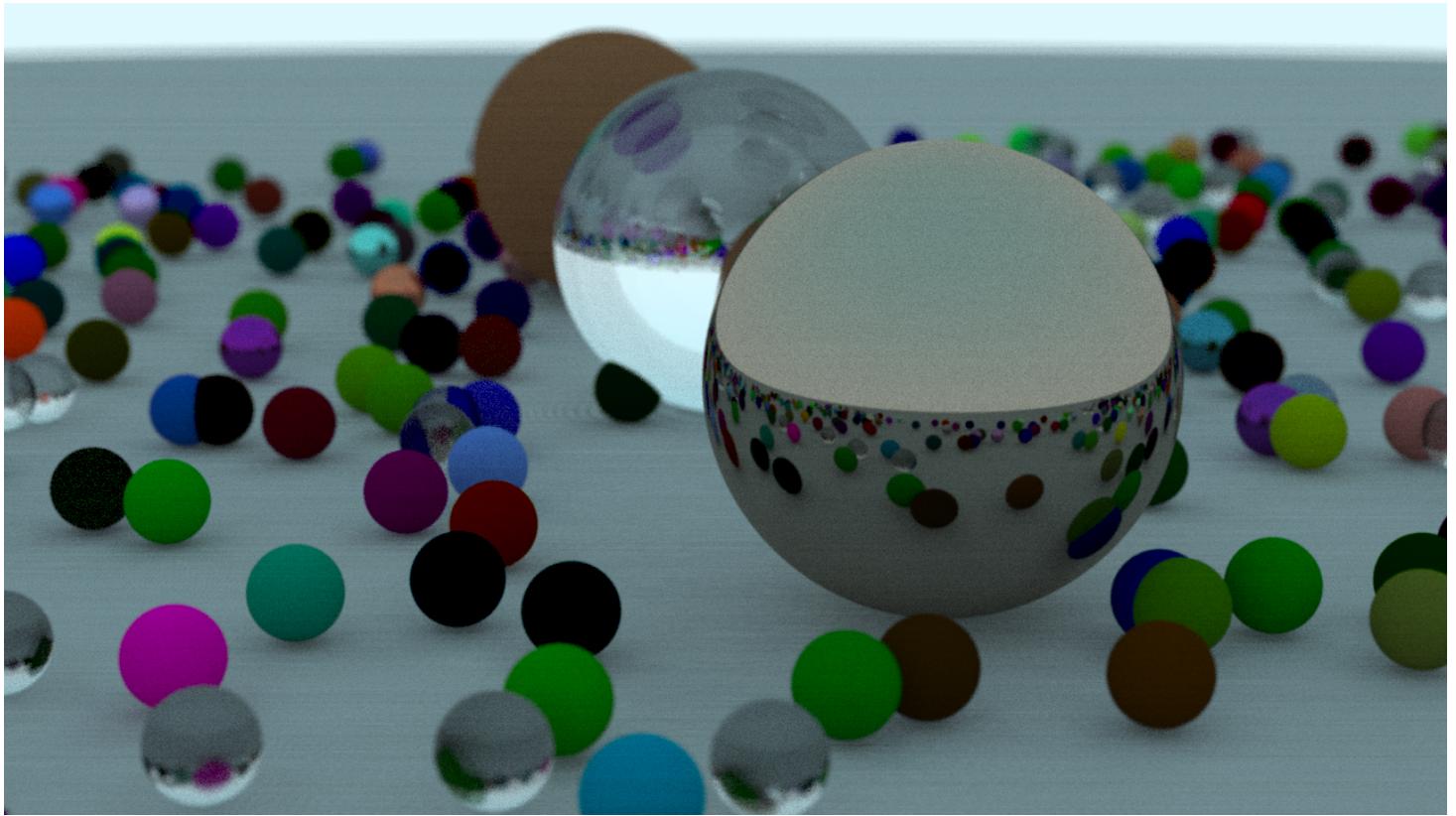
private static void RandomScene(ref HitList list)
{
    list.Add(new Sphere(new Vector3(0, -1000, 0), 1000f, new Lambertian(new Vector3(0.5f, 0.5f,
for (int i = -11; i < 11; ++i)
{
    for (int j = -11; j < 11; ++j)
    {
        Vector3 center = new Vector3(i + 0.9f * RandomFloat01(), 0.2f, j + 0.9f * RandomFloat01());
        Vector3 baseCenter = new Vector3(4, 0.2f, 0);
        float choose_mat = RandomFloat01();
        if ((center-baseCenter).magnitude > 0.9)
        {
            if(choose_mat < 0.8f)
            {
                list.Add(new Sphere(center, 0.2f, new Lambertian(new Vector3(RandomFloat01()
                    RandomFloat01() * RandomFloat01(), RandomFloat01() * RandomFloat01())));
            }
            else if (choose_mat < 0.95f)
            {
                list.Add(new Sphere(center, 0.2f, new Metal(new Vector3(0.5f * (1 + RandomFloat01())
                    0.5f * (1 + RandomFloat01()),
                    0.5f * (1 + RandomFloat01()))));
            }
            else
            {
                list.Add(new Sphere(center, 0.2f, new Dielectric(1.5f)));
            }
        }
    }
}
}

list.Add(new Sphere(new Vector3(0, 1, 0), 1f, new Dielectric(1.5f)));
list.Add(new Sphere(new Vector3(-4, 1, 0), 1f, new Lambertian(new Vector3(0.4f, 0.2f, 0.1f))));
list.Add(new Sphere(new Vector3(4, 1, 0), 1f, new Metal(new Vector3(0.7f, 0.6f, 0.5f), 0.0f)));
}

```

在教程的代码中，这里用多线程加速了一下，不然太慢了。

结果如下：



现在你有了一个很酷的ray tracer! 接下来呢?

后面推荐该作者的另外两本教程:

"Ray Tracing in One Weekend"

"Ray Tracing: The Rest of Your Life"

上面的教程里面会教你往Ray Tracer里面添加光线, 贴图, 体积效果等等。

周末愉快!