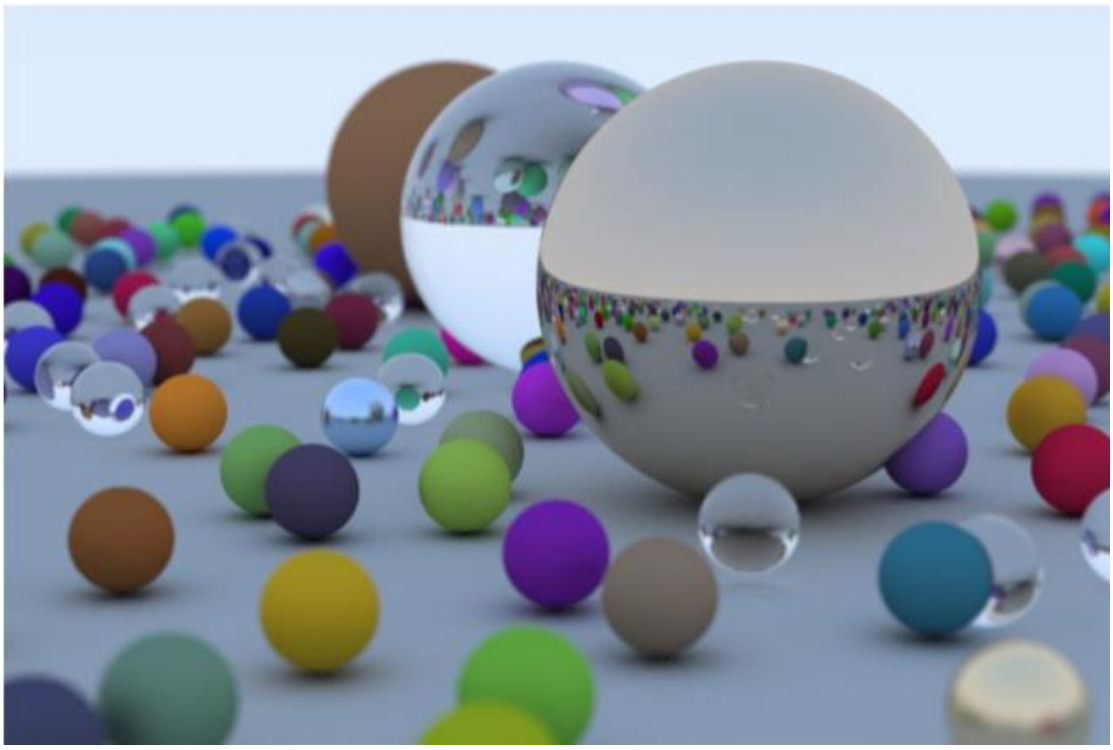


光线追踪教辅

罗付强

2020-02-28

光线追踪入门

步骤 1：渲染结果载体

光线追踪渲染器需要一个载体来展现最后的渲染结果，本课程用 C#作为基础语言（一般光线追踪是以 C++为基础语言，因为它效率高，C#相对效率会低很多），所以用 Bitmap 来作为最后结果渲染的载体。

关于 Bitmap：

Bitmap 的构造方法、属性、方法有很多，这里一般只用到这些：

Bitmap（int 32， int32） 构造方法，用指定大小初始化 Bitmap 类的实例
Height 属性，位图的高（以像素为单位）
Width 属性，位图的宽（以像素为单位）
SetPixel(Int32, Int32, Color) 方法，设置指定像素点的颜色
Save(String, ImageFormat) 方法，如果你想把渲染好的照片用文件存下来就用这个方法

基本上在这个简单的光线追踪里就只用的到上面的这些写方法。

看个实例：

```
int nx = 200;   //宽度
int ny = 100;   //高度
Bitmap bitmap = new Bitmap(nx, ny);
for (int i = 0 ; i < nx; i++)
{
    for (int j = 0; j < ny; j++)
    {
        double r = (double)(i) / (double)(nx);
        double g = (double)(j) / (double)(ny);
        double b = 0.2;
        int ir = (int)(255.99 * r);
        int ig = (int)(255.99 * g);
```

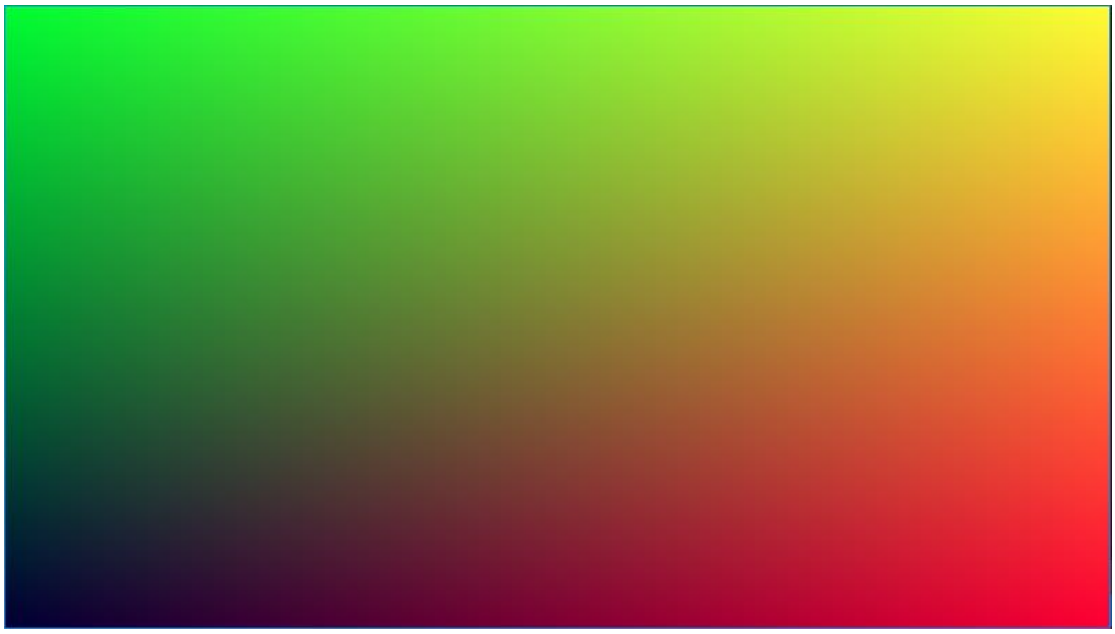
```
            int ib = (int)(255.99 * b);
            bitmap.SetPixel(i, ny - j - 1, Color.FromArgb(ir, ig, ib));
        }
    }
```

注：

1、 每个红色/绿色/蓝色的值范围都是从0.0 到 1.0，随后会把它放宽到rgb上

2、 R值从左到右逐渐怎大，G值从下到上逐渐增大，B值不变，所以看到的效果应该是左下角由黑色向右上角的黄色过度，因为红色和绿色混合到一起就是黄色。

最后得到的bitmap效果可以展示出来：



调个参数b = 0.5试试：

换成b = 1再试试：

至于`bitmap.SetPixel(i, ny - j - 1, Color.FromArgb(ir, ig, ib));`这里为什么要这样用，可以自己探索一下。

为什么要设置
```
double r = (double)(i) / (double)(nx);
double g = (double)(j) / (double)(ny);
double b = 0.5;
```
成这样，这个仅仅是为了演示，设置位图的每个像素点最后可以输出一个什么样的图片。

为什么要
```
int ir = (int)(255.99 * r);
int ig = (int)(255.99 * g);
int ib = (int)(255.99 * b);
```
这样写，因为Color的rgb每个值不能超过255。

步骤 2：一些基础类

图形程序需要一些类来作为向量、颜色、点载体。在大部分系统中这些矢量都是思维的（三维加上几何的齐次坐标，RGB 加上颜色的都明通道 A），但对于这个简单的光线追踪，我们就用三维的就行了。

向量：

新建一个名为 Vector 的类

一个三维向量需要有 x、y、z 三个属性：

```
private double x;  //对应向量x的值
private double y;  //对应向量y的值
private double z;  //对应向量 z 的值
public double X { get => x; set => x = value; }
public double Y { get => y; set => y = value; }
public double Z { get => z; set => z = value; }
```

注：get，set 也可以这样写：

```
public double X
{
    get{ return x; } set{ x = value; }
}
public double Y
{
    get{ return y; } set{ x = value; }
}
public double Z
{
    get{ return z; }set{ z = value;  }
}
```

为了方便我会采用第一种方法。

我给该向量类定义了三个构造函数：

```
public Vector3() { }
public Vector3(double x_, double y_, double z_)
{
    this.x = x_;
    this.y = y_;
    this.z = z_;
```

```
}
public Vector3(Vector3 v_)
{
    this.x = v_.x;
    this.y = v_.y;
    this.z = v_.z;
}
```

接下来是对该向量类进行运算符的重构:

+:

```
/// <summary>
/// 重载+运算符，向量的加法
/// </summary>
/// <param name="a">向量加数a</param>
/// <param name="b">向量加数b</param>
/// <returns>向量a和向量b相加的结果</returns>
public static Vector3 operator +(Vector3 a, Vector3 b)
{
    return new Vector3(a.X + b.X, a.Y + b.Y, a.Z + b.Z);
}


/// <summary>
/// 点与向量相加左
/// </summary>
/// <param name="a">点a</param>
/// <param name="b">向量b</param>
/// <returns>结果是一个向量</returns>
public static Vector3 operator +(Point3D a, Vector3 b)
{
    return new Vector3(a.X + b.X, a.Y + b.Y, a.Z + b.Z);
}


/// <summary>
/// 点与向量相加右
/// </summary>
/// <param name="a">向量a</param>
/// <param name="b">点b</param>
/// <returns>结果是一个向量</returns>
```

```
public static Vector3 operator +(Vector3 a, Point3D b)
{
    return new Vector3(a.X + b.X, a.Y + b.Y, a.Z + b.Z);
}
```

-：

```
/// <summary>
/// 重载-运算符，向量的减法
/// </summary>
/// <param name="a">向量被减数a</param>
/// <param name="b">向量减数b</param>
/// <returns>向量a和向量b相减的结果</returns>
public static Vector3 operator -(Vector3 a, Vector3 b)
{
    return new Vector3(a.X - b.X, a.Y - b.Y, a.Z - b.Z);
}


/// <summary>
/// 重载-运算符，向量的取负
/// </summary>
/// <param name="b">要取负的向量b</param>
/// <returns>向量b取负的结果</returns>
public static Vector3 operator -(Vector3 b)
{
    return new Vector3(-b.X, -b.Y, -b.Z);
}


/// <summary>
/// 点与向量相减左
/// </summary>
/// <param name="a">点a</param>
/// <param name="b">向量b</param>
/// <returns>结果是一个向量</returns>
public static Vector3 operator -(Point3D a, Vector3 b)
{
    return new Vector3(a.X - b.X, a.Y - b.Y, a.Z - b.Z);
}


/// <summary>
```

```
/// 点与向量相减右
/// </summary>
/// <param name="a">向量a</param>
/// <param name="b">点b</param>
/// <returns>结果是一个向量</returns>
public static Vector3 operator -(Vector3 a, Point3D b)
{
    return new Vector3(a.X - b.X, a.Y - b.Y, a.Z - b.Z);
}
```

***:**

```
/// <summary>
/// 重载*运算符，向量左乘以一个数
/// </summary>
/// <param name="a">向量a</param>
/// <param name="b">数b</param>
/// <returns>向量a和数b相乘的结果</returns>
public static Vector3 operator *(Vector3 a, double b)
{
    return new Vector3(a.X * b, a.Y * b, a.Z * b);
}


/// <summary>
/// 重载*运算符，向量右乘以一个数
/// </summary>
/// <param name="a">数a</param>
/// <param name="b">向量b</param>
/// <returns>数a和向量b相乘的结果</returns>
public static Vector3 operator *(double a, Vector3 b)
{
    return new Vector3(a * b.X, a * b.Y, a * b.Z);
}


/// <summary>
/// 重载*运算符,向量对应的坐标相乘
/// </summary>
/// <param name="a">向量a</param>
/// <param name="b">向量b</param>
/// <returns></returns>
public static Vector3 operator *(Vector3 a, Vector3 b)
```

```
{
    return new Vector3(a.X * b.X, a.Y * b.Y, a.Z * b.Z);
}
```

/:

```
/// <summary>
/// 重载/运算符，向量除以一个数
/// </summary>
/// <param name="a">向量a</param>
/// <param name="b">除数b</param>
/// <returns>向量a和数b相除的结果</returns>
public static Vector3 operator /(Vector3 a, double b)
{
    return new Vector3(a.X / b, a.Y / b, a.Z / b);
}
```

然后是向量的一些运算:

```
/// <summary>
/// 得到该向量的模
/// </summary>
/// <returns>返回向量的模</returns>
public double Length()
{
    return Math.Sqrt(x * x + y * y + z * z);
}


/// <summary>
/// 向量模的平方
/// </summary>
/// <returns></returns>
public double SquaredLength()
{
    return x * x + y * y + z * z;
}
```

```
/// <summary>
/// 使该向量单位化
/// </summary>
public void MakeUnitVector()
{
    double k = 1 / Length();
    x *= k;
    y *= k;
    z *= k;
}


/// <summary>
/// 向量的点积
/// </summary>
/// <param name="a">向量a</param>
/// <param name="b">向量b</param>
/// <returns>向量a和向量b的点积结果</returns>
public static double DotProduct(Vector3 a, Vector3 b)
{
    return a.X * b.X + a.Y * b.Y + a.Z * b.Z;
}


/// <summary>
/// 向量的叉积
/// </summary>
/// <param name="a">向量a</param>
/// <param name="b">向量b</param>
/// <returns>向量a和向量b的差积结果</returns>
public static Vector3 CrossProduct(Vector3 a, Vector3 b)
{
    return new Vector3(a.Y * b.Z - a.Z * b.Y, a.Z * b.X - a.X * b.Z,
a.X * b.Y - a.Y * b.X);
}


/// <summary>
/// 单位化向量
/// </summary>
/// <param name="v">需要进行单位化的向量v</param>
/// <returns>单位化向量结果</returns>
public static Vector3 UnitVector(Vector3 v)
{
```

```
    return v / v.Length();
}
```

一个完整的三维向量就齐活了。

三维的点：

新建一个名为 Point3D 的类

一个三维点和三维向量一样需要有 x、y、z 三个属性:

```
private double x;  //对应点x的值
private double y;  //对应点y的值
private double z;  //对应点 z 的值
public double X { get => x; set => x = value; }
public double Y { get => y; set => y = value; }
public double Z { get => z; set => z = value; }
```

两个构造函数:

```
public Point3D() { }

public Point3D(double x, double y, double z)
{
    this.x = x;
    this.y = y;
    this.z = z;
}
```

运算符重构:

```
/// <summary>
```

```
/// 两个点相减得到这两个点构成的向量
/// </summary>
/// <param name="a">末尾的点a</param>
/// <param name="b">开始的点b</param>
/// <returns>两个点组成的向量</returns>
public static Vector3 operator -(Point3D a, Point3D b)
{
    return new Vector3(a.x - b.x, a.y - b.y, a.z - b.z);
}


/// <summary>
/// 两个点相加得到向量
/// </summary>
/// <param name="a"></param>
/// <param name="b"></param>
/// <returns></returns>
public static Vector3 operator +(Point3D a, Point3D b)
{
    return new Vector3(a.x + b.x, a.y + b.y, a.z + b.z);
}
```

储存颜色相应值得颜色类

新建一个名为 Color3D 的类

三个属性，分别代表未放宽的 r、g、b 值:

```
private double r;  //未放宽的r值(0~1)
private double g;  //未放宽的g值(0~1)
private double b;  //未放宽的 z 值(0~1)
public double R { get => r; set => r = value; }
public double G { get => g; set => g = value; }
public double B { get => b; set => b = value; }
```

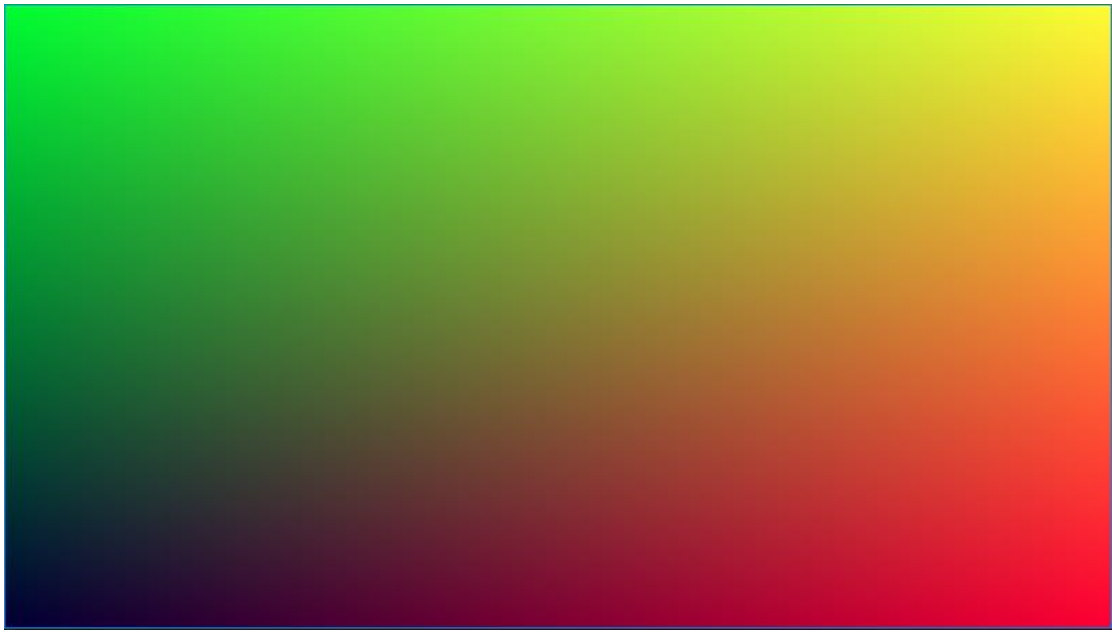
两个构造函数:

```
public Color3D() { }

public Color3D(double r, double g, double b)
{
    this.r = r;
    this.g = g;
    this.b = b;
}
```

基本需要的一些类就是这些了，当然，所有的这些类都可以用 Vector 一个类就可以完成，但是为了使逻辑清晰点我就不厌其烦的多定义两个类。

简单的看看效果：

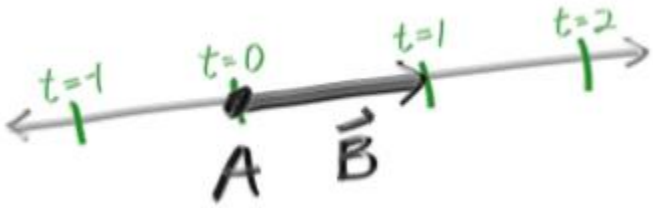
```
int nx = 200;   //宽度
int ny = 100;   //高度
Bitmap bitmap = new Bitmap(nx, ny);
Color3D color3D = new Color3D();
for (int i = 0; i < nx; i++)
{
    for (int j = 0; j < ny; j++)
    {
        color3D.R = (double)i / (double)nx;
        color3D.G = (double)j / (double)ny;
        color3D.B = 0.2;
        int ir = (int)(255.99 * color3D.R);
        int ig = (int)(255.99 * color3D.G);
        int ib = (int)(255.99 * color3D.B);
        bitmap.SetPixel(i, ny - j - 1, Color.FromArgb(ir, ig, ib));
    }
}
```



一样的效果

步骤 3：光线 Ray

光线追踪当然少不了光线：
光线就是一条射线，我们可以大体的写出他的函数 **p**(t) = A + t * **B**;其中 A 代表起点，B 代表方向，t 代表射线参数（t 不同射线移动的成都不同）。
注：上述表达式其实是错误的，点是不能和向量相运算的，不过为了表示就暂时认为正确。



来看看光线类：

```
class Ray
{
    private Point3D origin;
    private Vector3 direction;
```

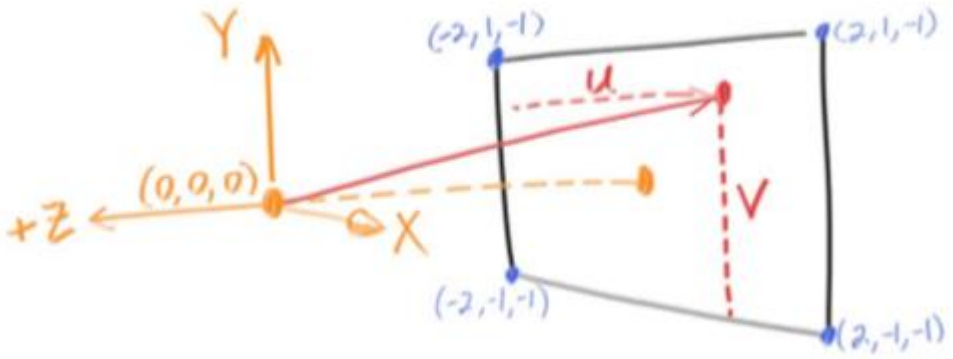
```
    public Ray () { }

    public Ray (Point3D origin, Vector3 direction)
    {
        this.origin = origin;
        this.direction = direction;
    }

    internal Point3D Origin { get => origin; set => origin = value; }
    internal Vector3 Direction { get => direction; set => direction =
value; }

    public Vector3 PointAtPara(double t)
    {
        return origin + new Point3D(t * direction.X, t * direction.Y,
t * direction.Z);
    }
}
```

其中方法 PointAtPara 就是用来表示前面那个光线函数的。

接下来我要利用 Ray 来简单的模拟一个根据光线变化颜色渐变的简单光线追踪：

光线追踪核心：让光线通过（击中）像素点并计算出这些光线方向上能看到什么颜色。方法就是，计算哪条光线从出发点到像素面（理解为屏幕），计算出这条光线通过（击中）的像素点，然后计算像素点颜色。(假设光线是从眼睛向外发射的，也可以理解为相机)

这里用 200*100 的图像，把眼睛位置放在（0,0,0），y 轴向上，x 轴向右，遵从右手坐标系，进入屏幕的为负 z 轴。这里从屏幕左下角开始遍历屏幕，使用屏幕两侧两个的位移矢量（**u** 和 **v**）来控制光线打到屏幕上的点。下图图示：



先把每个像素点的颜色解决:

```
public static Color3D Color(Ray r)
{
    Vector3 unitDirection = Vector3.UnitVector(r.Direction);
    double t = 0.5 * (unitDirection.Y + 1.0);
    return new Color3D((1.0 - t) + t * 0.5, (1.0 - t) + t * 0.7, (1.0
- t) + t * 1);
}
```

解释：这个函数是根据 y 坐标的上下限来线性混合白色和蓝色。首先把光线的方向向量单位化（这里光线的起点是（0,0,0），其实就是对光线的单位化），让 -1.0 < y < 1.0,然后再扩展到 0.0 < t < 1.0,t=0.0 白色，t=1.0 蓝色。两者之间混合就用到了线性混合，具体算法 **blended_value = (1-t)*start_value + t*end_value**，函数里 start_value = Color3D(1.0,1.0,1.0),end_value = Color3D(0.5,0.7,1.0)。

来看看效果：

```
int nx = 200;   //宽度
int ny = 100;   //高度
Bitmap bitmap = new Bitmap(nx, ny);

Point3D lowerLeftCorner = new Point3D (-2, -1, -1);
Vector3 horizontal = new Vector3(4, 0, 0);
Vector3 vertical = new Vector3(0, 2, 0);
Point3D origin = new Point3D (0, 0, 0);
for (int i = 0; i < nx; i++)
{
    for (int j = 0; j < ny; j++)
```

```
    {
        double u = (double)i / (double)nx;
        double v = (double)j / (double)ny;
        Ray r = new Ray(origin, lowerLeftCorner + u * horizontal + v
* vertical);
        Color3D col = RTUtils.Color(r);
        int ir = (int)(255.99 * col.R);
        int ig = (int)(255.99 * col.G);
        int ib = (int)(255.99 * col.B);
        bitmap.SetPixel(i, ny - j - 1 , Color.FromArgb(ir, ig, ib));
    }
}
```

最后的结果：

步骤 4：加入最简单的立体图形，球（sphere）

光线追踪去只有背景屏幕是么有任何乐趣的，先现在我们就来加入一点简单的物体，我们就选择球，因为它最简单。

球体方程 x*x + y*y + z*z = R*R,球心（0,0,0）、半径 R。我们都知道对于任何一个点（x,y,z）满足球的方程 x*x + y*y + z*z = R*R，那这个点就在球面上，反之就不在球面上。球心位于（cx,cy,cz）时，球面方程就是(x-cx)*(x-cx) + (y-cy)*(y-cy) + (z-cz)*(z-cz)= R*R

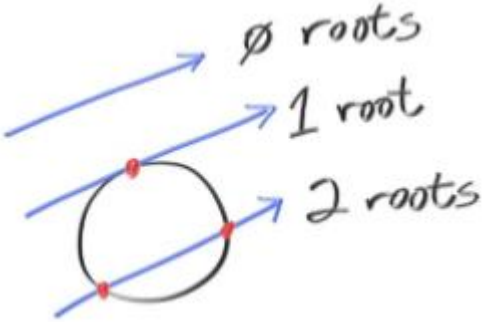
但是在图形学中我们得用向量来表示这个球，球心 C（cx,cy,cz）到点 P=（x,y,z）的向量为**(P - C)**，有 dot（**(P - C)**，**(P - C)**） = (x-cx)*(x-cx) + (y-cy)*(y-cy) + (z-cz)*(z-cz)= R*R，

dot 表示向量的点积。

光线那一步我们讲了 **p**(t) = A + t * **B**，所以有：
dot(**(p (t) - c)**,**(p (t) - c)**) = R*R
dot(**(A + t* B - C)**,**(A + t* B - C)**) = R*R

最后得到：
t*t*dot(**B** , **B**) + 2*t*dot(**B**,**A -C**) + dot(**A-C**,**A -C**) - R*R = 0

这个方程里边只有 t 是未知的，并且这个方程是二次方程就可以求出 t，有三种情况：



简单的二维图像

简单的做个二维，我们把这个方程转换成代码，我们假定打中球的点就用红色来表示：

```
public static bool Hit(center, double radius, Ray r)
{
    Vector3 oc = r.Direction - center;
    double a = Vector3.DotProduct(r.Direction, r.Direction);
    double b = 2.0 * Vector3.DotProduct(oc, r.Direction);
    double c = Vector3.DotProduct(oc, oc) - radius * radius;
    double discriminant = b * b - 4 * a * c;
    return (discriminant > 0);
}

public static Color3D Color(Ray r)
{
    if (Hit(new Point3D(0, 0, -1), 0.5, r))
    {
        return new Color3D(1, 0, 0);
    }
    Vector3 unitDirection = Vector3.UnitVector(r.Direction);
    double t = 0.5 * (unitDirection.Y + 1.0);
    return new Color3D((1.0 - t) + t * 0.5, (1.0 - t) + t * 0.7, (1.0
- t) + t * 1);
}
```

看看效果：

```
int nx = 200;   //宽度
int ny = 100;   //高度
Bitmap bitmap = new Bitmap(nx, ny);
Point3D lowerLeftCorner = new Point3D(-2, -1, -1);
Vector3 horizontal = new Vector3(4, 0, 0);
Vector3 vertical = new Vector3(0, 2, 0);
Point3D origin = new Point3D(0, 0, 0);
for (int i = 0; i < nx; i++)
{
    for (int j = 0; j < ny; j++)
    {
        double u = (double)i / (double)nx;
        double v = (double)j / (double)ny;
        Ray r = new Ray(origin, lowerLeftCorner + u * horizontal + v
* vertical);
        Color3D col = RTUtils.Color(r);
        int ir = (int)(255.99 * col.R);
        int ig = (int)(255.99 * col.G);
        int ib = (int)(255.99 * col.B);
        bitmap.SetPixel(i, ny - j - 1, Color.FromArgb(ir, ig, ib));
    }
}
```

结果：

现在只只是一个简单的二维成像，缺少很多东西，比如阴影和反射光线，多个球。

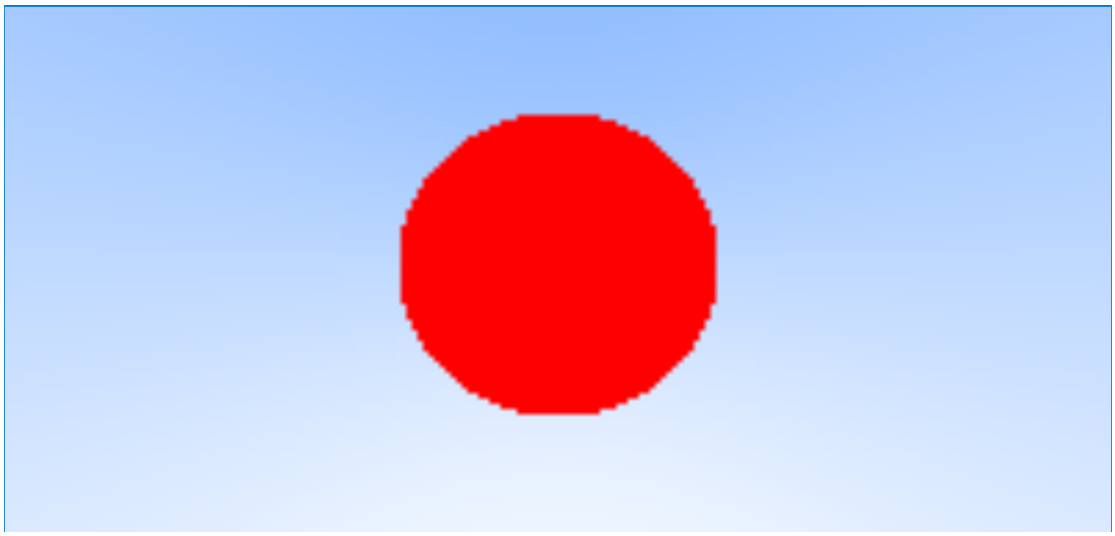
加入三维立体感

想一下，如果使 z = 1，得到的图像是哪种?

我们试一试

```
public static Color3D Color(Ray r)
{
    if (Hit(new Point3D(0, 0, 1), 0.5, r))
    {
        return new Color3D(1, 0, 0);
    }
    Vector3 unitDirection = Vector3.UnitVector(r.Direction);
    double t = 0.5 * (unitDirection.Y + 1.0);
    return new Color3D((1.0 - t) + t * 0.5, (1.0 - t) + t * 0.7, (1.0
- t) + t * 1);
}
```

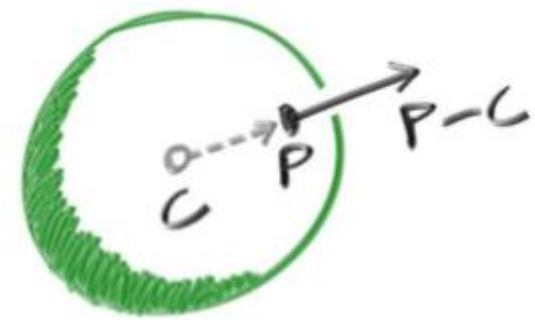
得到:



和 z=-1 是一样的，这个结果和你想的是不是一样的?
其实就是我们看到了球的背面。

接下来我们就是要处理背面阴影的部分。

我们先要得到一个表面法线，这样我们才能进行阴影处理。法线是一个垂直于曲面的向量。



想一下，地球中心到你的矢量是否是指向正上方的。
接着我们来添加阴影，由于我们现在我们没有 light（灯光）或者其他光源，我们就简单的把每根法线想象成一个颜色。然后我们要可视化法线，假设 N 是单位长度就很简单也很直观，故每个分量都会在-1 和 1 之间，这里有个技巧就是将每个分量都映射到 0 到 1 之间，然后再将 x/y/z 映射到 r/g/b。一般情况下我们要的不仅仅是是否集中，还要命中点。我们就要最近的命中点，也就是 t 最小的那个命中点。

看代码:

```
public static double Hit(Point3D center, double radius, Ray r)
{
    Vector3 oc = r.Origin - center;
    double a = Vector3.DotProduct(r.Direction, r.Direction);
    double b = 2.0 * Vector3.DotProduct(oc, r.Direction);
    double c = Vector3.DotProduct(oc, oc) - radius * radius;
    double discriminant = b * b - 4 * a * c;
    if(discriminant < 0)
    {
        return -1.0;
    }
    else
    {
        return (-b - Math.Sqrt(discriminant)) / (2.0 * a);
    }
}
```

没打中就返回-1，打中了就返回最近的那个击中点对对应的 t，(-b - Math.Sqrt(discriminant)) / (2.0 * a);

```
public static Color3D Color(Ray r)
{
    double t = Hit(new Point3D(0, 0, -1), 0.5, r);
    if(t > 0.0)
    {
        Vector3 N = Vector3.UnitVector(r.PointAtPara(t) - new
Point3D(0, 0, -1));
```

```
        return new Color3D(0.5 * (N.X + 1), 0.5 * (N.Y + 1), 0.5 *
(N.Z + 1));
    }
    Vector3 unitDirection = Vector3.UnitVector(r.Direction);
    t = 0.5 * (unitDirection.Y + 1.0);
    return new Color3D((1.0 - t) + t * 0.5, (1.0 - t) + t * 0.7, (1.0
- t) + t * 1);
}
```

t>0,我们有了 t 就有了最近那个击中点，有了那个击中点就有了球心到击中点的的法线 Vector3 N = Vector3.UnitVector(r.PointAtPara(t) - new Point3D(0, 0, -1))，然后将法线映射到 0 到 1 这个区间，再映射到 R/G/B 上；t<=0,就和前面一样让颜色在白色和蓝色之间渐变。

效果：

步骤 5：要是有几个球呢？

大家首先想到的肯定是用数组来装，其实数组不是最好的解决方案。有个更好地方法是我们为光线能够集中地任何一个物体创建一个父类，并且同时创建球体和球体列表。这里就用 Hitable 来作为类的名字（其实用 Object 更形象，但是面向对象编程里边，Object 有特殊的含义）。

Hitable 类和 HitRecord 类

Hitable 类会有一个接收光线的 Hit 函数 ， 对于一个初始光线 t>0, t 有个有效范围 tMin ~ tMax，也就是在此范围内击中才有效计数。这里设计的关键问题是：如果我们撞到了某个东西，是否应该计算法线。来看下解决方案：

```
class HitRecord
{
    private double t;
    private Vector3 p;  //光线向量
    private Vector3 normal; // 法线

    public double T { get => t; set => t = value; }
    internal Vector3 P { get => p; set => p = value; }
    internal Vector3 Normal { get => normal; set => normal = value; }
}
```

```
class Hitable
{
    public virtual bool Hit(Ray r, double tMin, double tMax, ref
HitRecord rec)
    {
        return false;
    }
}
```

关于 HitRecord 我们就简单的理解为接收记录或者击中记录。

在 Hitable 中可以看到有个参数前面有个 ref，这是因为我们需要在这个函数里边改变这个参数的一些值（可以去了解一下 ref 相关的知识，简单解释一下，就和指针有点一样，但不相同）

Sphere 类（球体）

都进入球体这个步骤这么久了，我们还是没看到一个有关球体的实体类，接下来就来了：

```
class Sphere : Hitable
{
    private double radius;
    private Point3D center;
```

```
    public double Radius { get => radius; set => radius = value; }
    internal Point3D Center { get => center; set => center = value; }
    public Sphere() { }
    public Sphere(Point3D cen, double r)
    {
        this.center = cen;
        this.radius = r;
    }
    public override bool Hit(Ray r, double tMin, double tMax, ref
HitRecord rec)
    {
        Vector3 oc = r.Origin - center;
        double a = Vector3.DotProduct(r.Direction, r.Direction);
        double b = Vector3.DotProduct(oc, r.Direction);
        double c = Vector3.DotProduct(oc, oc) - radius * radius;
        double discriminant = b * b - a * c;
        if(discriminant > 0)
        {
            double temp = (-b - Math.Sqrt(b * b - a * c)) / a;
            if(temp < tMax && temp > tMin)
            {
                rec.T = temp;
                rec.P = r.PointAtPara(rec.T);
                rec.Normal = (rec.P - center) / radius;
                return true;
            }
            temp = temp = (-b + Math.Sqrt(b * b - a * c)) / a;
            if (temp < tMax && temp > tMin)
            {
                rec.T = temp;
                rec.P = r.PointAtPara(rec.T);
                rec.Normal = (rec.P - center) / radius;
                return true;
            }
        }
        return false;
    }
}
```

可以看到一个球体有两个两个属性分别是球心 center 和半径 radius。球体类继承于 Hitable，我们重写了 Hit 方法。

可以看到
```
Vector3 oc = r.Origin - center;
double a = Vector3.DotProduct(r.Direction, r.Direction);
double b = Vector3.DotProduct(oc, r.Direction);
double c = Vector3.DotProduct(oc, oc) - radius * radius;
double discriminant = b * b - a * c;
```
少了两个 2*，其实这个道理是一样的知识把 2*抵消了，如果要坚持用前面的也
可以
```
Vector3 oc =  r.Direction - center;
double a = Vector3.DotProduct(r.Direction, r.Direction);
double b = 2.0 * Vector3.DotProduct(oc, r.Direction);
double c = Vector3.DotProduct(oc, oc) - radius * radius;
double discriminant = b * b - 4 * a * c;
```

HitableList 类（接收列表）

```
class HitableList : Hitable
{
    private List<Hitable> list;
    private int listSize;

    public HitableList(){}

    public HitableList(List<Hitable> list, int listSize)
    {
        this.list = list;
        this.listSize = listSize;
    }

    public int ListSize { get => listSize; set => listSize = value; }
    internal List<Hitable> List { get => list; set => list = value; }

    public override bool Hit(Ray r, double tMin, double tMax, ref
HitRecord rec)
    {
        HitRecord tempRec = new HitRecord();
        bool hitAnything = false;
        double closestSoFar = tMax;
        for (int i = 0; i < listSize; i++)
        {
            if (list[i].Hit(r, tMin, closestSoFar, ref tempRec))
            {
```

```
                hitAnything = true;
                closestSoFar = tempRec.T;
                rec = tempRec;
            }
        }
        return hitAnything;
    }
}
```

可以看到 HitableList 有两个属性，list 物体列表，listSize 物体个数。同样 HitableList 也继承与 Hitable，并且重写了 Hit 方法，这里重写 Hit 方法实现的功能就是判断光线 r 是否击中了某个物体。

Color 函数也要发生变化

```
public static Color3D Color(Ray r, Hitable world)
{
    HitRecord rec = new HitRecord();
    if (world.Hit(r, 0.0, double.MaxValue, ref rec))
    {
        return new Color3D(0.5 * (rec.Normal.X + 1), 0.5 *
(rec.Normal.Y + 1), 0.5 * (rec.Normal.Z + 1));
    }
    else
    {
        Vector3 unitDirection = Vector3.UnitVector(r.Direction);
        double t = 0.5 * (unitDirection.Y + 1.0);
        return new Color3D((1.0 - t) + t * 0.5, (1.0 - t) + t * 0.7,
(1.0 - t) + t * 1);
    }
}
```

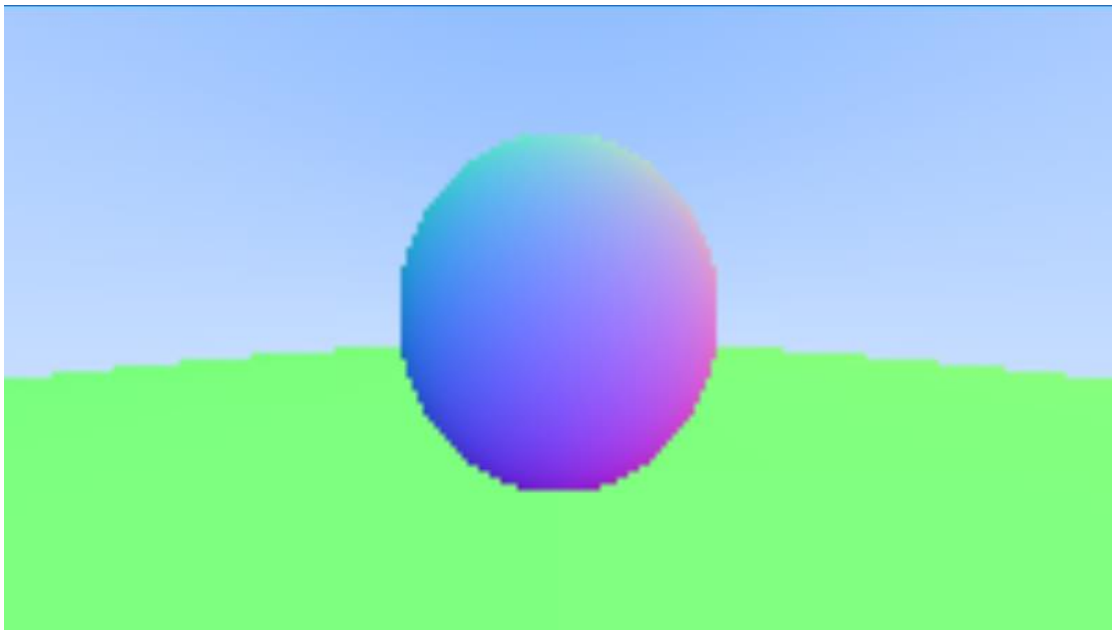
这个方法的思想和前面的几乎是一致的，就不在赘述。

看看结果：

```
int nx = 200;   //宽度
int ny = 100;   //高度
Bitmap bitmap = new Bitmap(nx, ny);
```

```
Point3D lowerLeftCorner = new Point3D(-2, -1, -1);
Vector3 horizontal = new Vector3(4, 0, 0);
Vector3 vertical = new Vector3(0, 2, 0);
Point3D origin = new Point3D(0, 0, 0);
List<Hitable> list = new List<Hitable>();
list.Add(new Sphere(new Point3D(0, 0, -1), 0.5));
list.Add(new Sphere(new Point3D(0, -100.5, -1), 100));
HitableList world = new HitableList(list, 2);
for (int i = 0; i < nx; i++)
{
    for (int j = 0; j < ny; j++)
    {
        double u = (double)i / (double)nx;
        double v = (double)j / (double)ny;
        Ray r = new Ray(origin, lowerLeftCorner + u * horizontal + v
* vertical);
        Vector3 p = r.PointAtPara(2.0);
        Color3D col = RTUtils.Color(r, world);
        int ir = (int)(255.99 * col.R);
        int ig = (int)(255.99 * col.G);
        int ib = (int)(255.99 * col.B);
        bitmap.SetPixel(i, ny - j - 1, Color.FromArgb(ir, ig, ib));
    }
}
```

得到图片



从结果图来看可以看出途中有两个球，小球放在大球上面，大球很大所以只能看到一部分。

步骤 6：抗锯齿

大家都知道，相机在拍照的时通常是没有锯齿在物体边缘的。边缘部分的像素是一些前景色和背景色的混合，我们通过对每个像素内的一堆样本求平均，也能得到了上述混合的效果。其实在我们这个简单的光线追踪中对于这个抗锯齿我们得到的效果不是特别明显，反而他会产生更多代码，而且还会使效率大打折扣。用处还是有，图片会变清晰，但是是用效率作为交换。

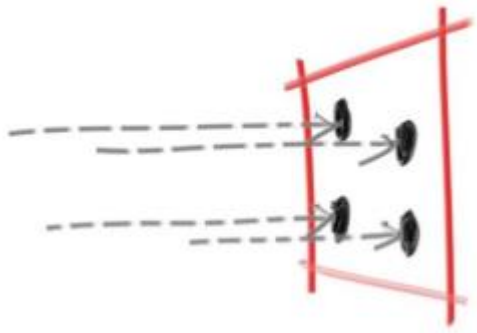
我们需要抽象出一个 Camera 类

C#里的 Random

在抗锯齿中我们会用到随机数生成，我们选择而简单的 Random 类。
Random rd = new Random（）;
rd.NextDouble()　//返回 0-1.0 之间的随机数，我们就是需要这个方法
注：时间间隔如果太短，创建的多个不同的 Ramdom 对象将具有相同的默认种子值，因而会产生几组相同的随机数。使用单个 Random 对象生成所有随机数可避免此问题。

实现方法

对于一个给定的像素，我们要有几个样本在该像素内，并且发送光线通过每个样本，然后对光线颜色球平均：



Camera 类:

```
class Camera
{
    private Point3D lowerLeftCorner ;
    private Vector3 horizontal ;
    private Vector3 vertical ;
    private Point3D origin ;

    public Camera()
    {
        LowerLeftCorner = new Point3D(-2, -1, -1);
        Horizontal = new Vector3(4, 0, 0);
        Vertical = new Vector3(0, 2, 0);
        Origin = new Point3D(0, 0, 0);
    }

    internal Point3D LowerLeftCorner { get => lowerLeftCorner; set =>
lowerLeftCorner = value; }
    internal Vector3 Horizontal { get => horizontal; set =>
horizontal = value; }
    internal Vector3 Vertical { get => vertical; set => vertical =
value; }
    internal Point3D Origin { get => origin; set => origin = value; }

    public Ray GetRay(double u, double v)
    {
        return new Ray(origin, lowerLeftCorner + u * horizontal + v *
vertical - origin);
    }
}
```

将图片生成程序里边的

```
Point3D lowerLeftCorner = new Point3D(-2, -1, -1);
Vector3 horizontal = new Vector3(4, 0, 0);
Vector3 vertical = new Vector3(0, 2, 0);
Point3D origin = new Point3D(0, 0, 0);
```

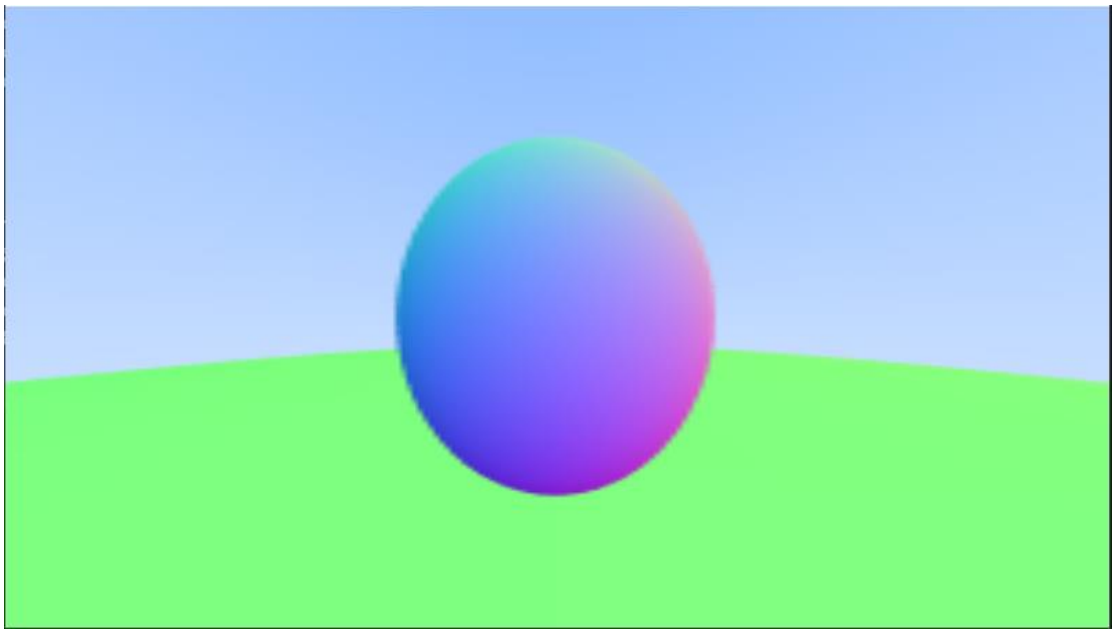
提取出来了，加了一个新方法 GetRay，顾名思义，得到光线向量。

看结果

```
int nx = 200;   //宽度
int ny = 200;   //高度
int ns = 100;
Bitmap bitmap = new Bitmap(nx, ny);
Random rd = new Random();
List<Hitable> list = new List<Hitable>();
list.Add(new Sphere(new Point3D(0, 0, -1), 0.5));
list.Add(new Sphere(new Point3D(0, -100.5, -1), 100));
HitableList world = new HitableList(list, 2);
Camera cam = new Camera();
for (int i = 0; i < nx; i++)
{
    for (int j = 0; j < ny; j++)
    {
        Color3D col = new Color3D();
        for (int s = 0; s < ns; s++)
        {
            double u = (double)(i + rd.NextDouble()) / (double)nx;
            double v = (double)(j + rd.NextDouble()) / (double)ny;
            Ray r = cam.GetRay(u, v);
            Color3D colTemp = RTUtils.Color(r, world);
            col.R += colTemp.R;
            col.G += colTemp.G;
            col.B += colTemp.B;
        }
        col.R /= ns;
        col.G /= ns;
```

```
            col.B /= ns;
            int ir = (int)(255.99 * col.R);
            int ig = (int)(255.99 * col.G);
            int ib = (int)(255.99 * col.B);
            bitmap.SetPixel(i, ny - j - 1, Color.FromArgb(ir, ig, ib));
        }
    }
```

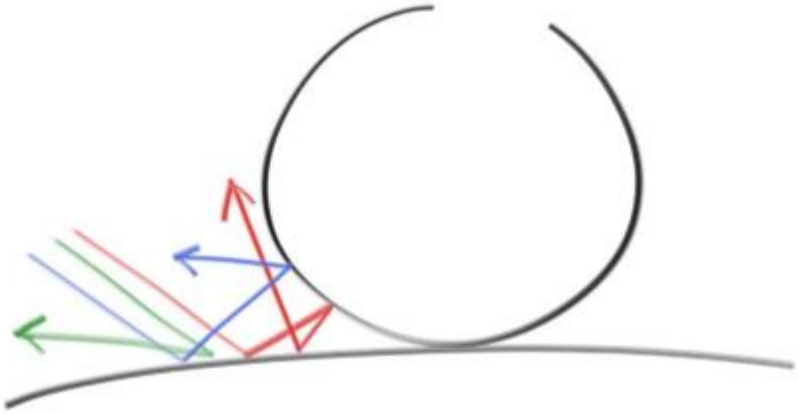
得到图片：



可以明显看到，照片还是清晰了不少，但是你如果把代码写出来运行，你会感觉速度变慢了。

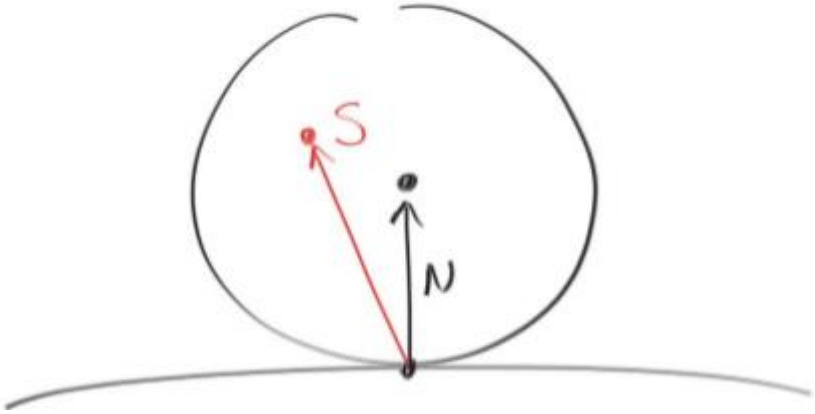
步骤 7：漫反射材质

不发光的漫射物体只是呈现出它们周围环境的颜色，但它们通过自身固有的颜色来调节这种颜色。被漫反射表面反射的光的方向是随机的。所以，如果我们把三束光线射入两个漫反射表面之间的缝隙，它们会有不同的随机行为（漫反射要其实要比折射简单）：



它们也可能被吸收而不是反射。表面越黑，越容易被吸收。任何随机方向的算法都会产生粗糙的表面。一种最简单的方法是完全理想的漫反射面（全部吸收，就是黑色）。

从单位半径球中选取一个与命中点相切的随机点 s，从命中点 p 向随机点 s 发送一条射线。那个球有球心(p + N)（这里是点）：



我们还需要一种方法在以原点为中心的单位半径球体中随机选取一个点。我们将使用通常最简单的算法:拒绝法。首先，我们在单位立方体中随机选择一个点，其中 x、y 和 z 的取值范围都是-1 到+1。如果这个点在球面外，我们拒绝这个点，然后再试一次。do/while 结构非常适合:

```
private static Random rd = new Random();
public static Vector3 RandomInUnitSphere()
{
    Vector3 p = new Vector3();
    do
    {
        p = 2.0 * new Vector3(rd.NextDouble(), rd.NextDouble(),
rd.NextDouble()) - new Point3D(1, 1, 1);
    } while (p.SquaredLength() >= 1.0);
```

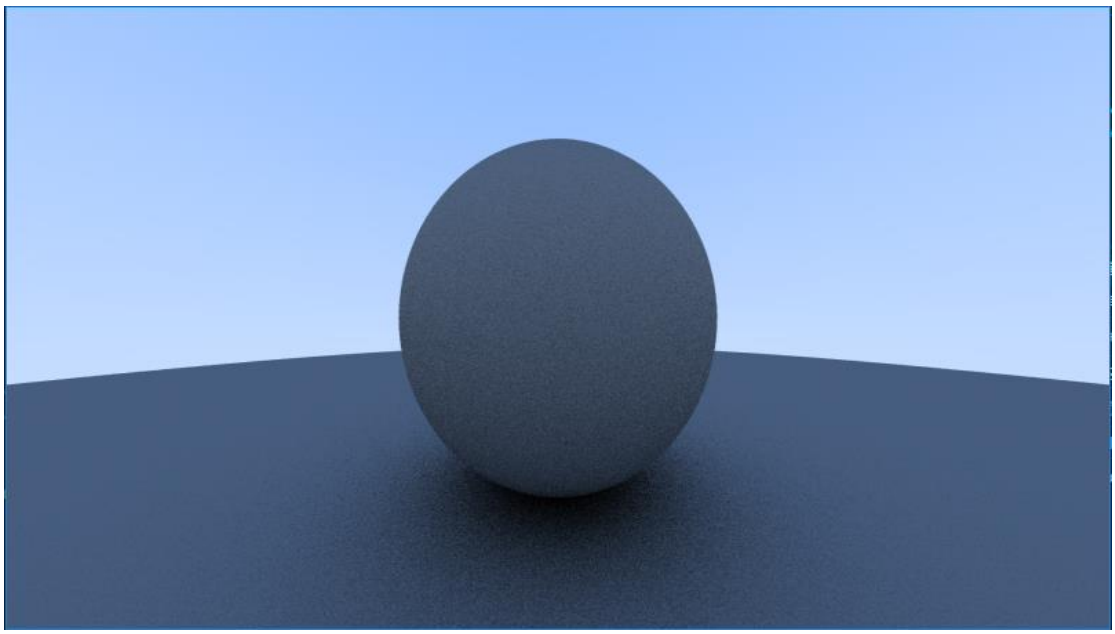
```
    return p;
}
```

代码中-new Point3D(1, 1, 1)但是前面剩了2，这样做的目的是为了避免偶然情况发生提高随机性。待会儿可以用p = new Vector3(rd.NextDouble(), rd.NextDouble(), rd.NextDouble())试试看是什么效果。

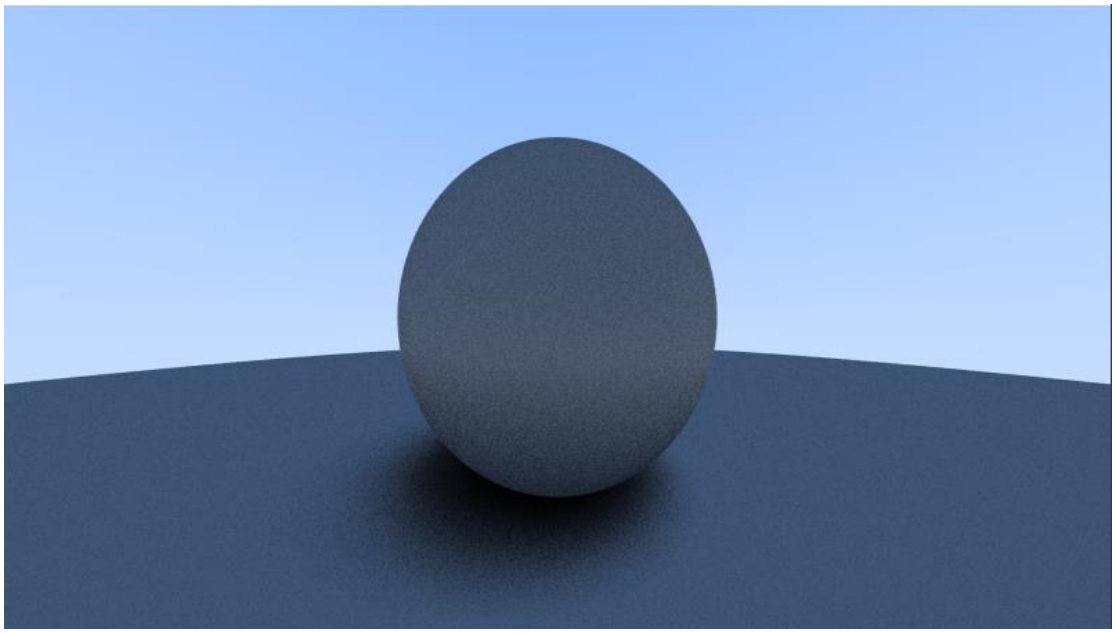
```
public static Color3D Color(Ray r, Hitable world)
{
    HitRecord rec = new HitRecord();
    if (world.Hit(r, 0.001, double.MaxValue, ref rec))
    {
        Vector3 target = rec.P + rec.Normal + RandomInUnitSphere();
        Color3D colorTemp = Color(new Ray(new Point3D(rec.P.X,
rec.P.Y, rec.P.Z), target - rec.P), world);
        colorTemp.R *= 0.5;
        colorTemp.G *= 0.5;
        colorTemp.B *= 0.5;
        return colorTemp;
    }
    else
    {
        Vector3 unitDirection = Vector3.UnitVector(r.Direction);
        double t = 0.5 * (unitDirection.Y + 1.0);
        return new Color3D((1.0 - t) + t * 0.5, (1.0 - t) + t * 0.7,
(1.0 - t) + t * 1);
    }
}
```

Vector3 target = rec.P + rec.Normal + RandomInUnitSphere()这句代码就是“从单位半径球中选取一个与命中点相切的随机点 s，从命中点 p 向随机点 s 发送一条射线。那个球有球心(p + N)（这里是点）”的翻译

得到图片：



把 p = 2.0 * new Vector3(rd.NextDouble(), rd.NextDouble(), rd.NextDouble()) - new Point3D(1, 1, 1)换成 p = new Vector3(rd.NextDouble(), rd.NextDouble(), rd.NextDouble())试试:

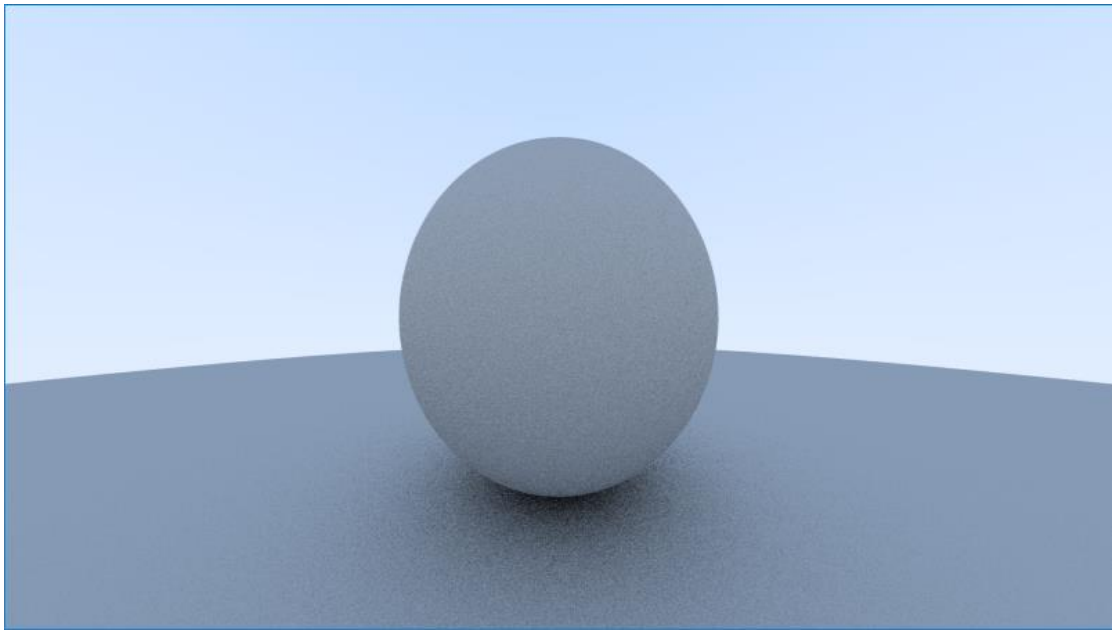


两张照片差距不是很大。

注意球体下面的阴影。这张照片很暗，但是我们的球体每次反弹只吸收一半的能量，所以它们是 50%的反射器。如果你看不到阴影，不要担心，我们现在就会修复它。这些球体看起来应该很轻(在现实生活中，是浅灰色)。这样做的原因是，几乎所有的图像查看器都假定图像经过了“gamma 校正”，这意味着 0 到 1 的值在存储为字节之前进行了一些转换。我们只需要了解它，知道有这个事情就行了。我们可以用“gamma2”，也就是提升颜色 1/gamma，就是开根号：

```
col.R /= ns;
col.G /= ns;
col.B /= ns;
col = new Color3D(Math.Sqrt(col.R), Math.Sqrt(col.G), Math.Sqrt(col.B));
```

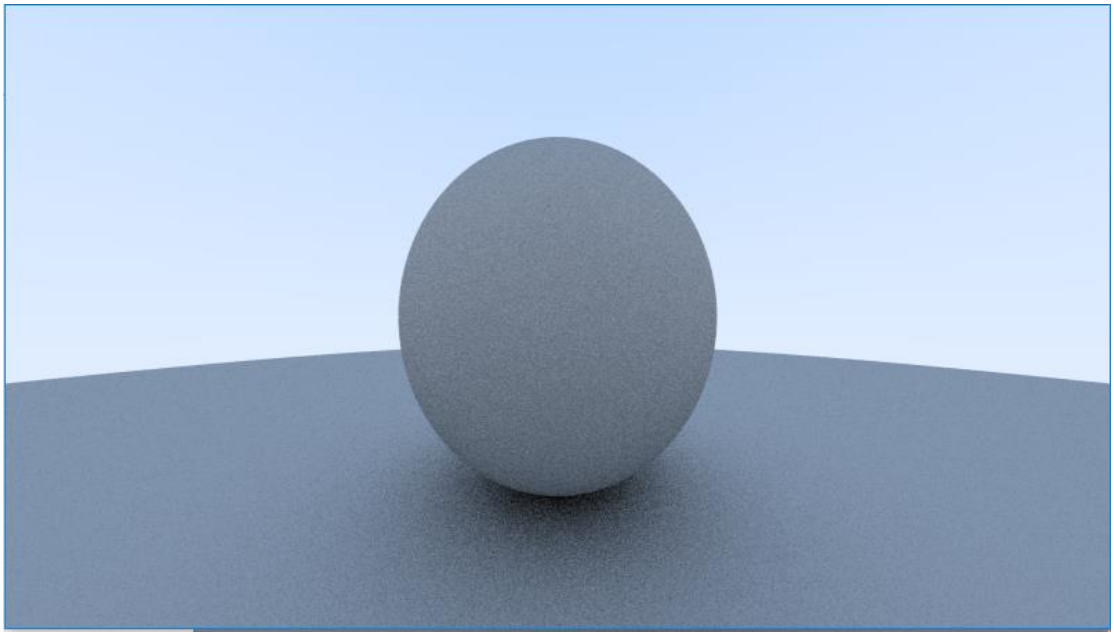
结果图片：



注：这里面还有一个小问题。有些反射的光线并不是在 t=0 时，而是在 t=-0.0000001 或 t=0.00000001 或球面扇形间给出的任何浮点近似值。所以我们需要忽略接近零的点：

```
if (world.Hit(r, 0.001, double.MaxValue, ref rec))
```

如果`if (world.Hit(r, 0.0, double.MaxValue, ref rec))`这样，结果是：



步骤 8：金属材质

Material 类

有了不同的材质过后就要考虑怎么讲材质和物体绑定起来，我们这里用一个 Material 类来封装材质，然后以属性的形式绑定到物体上去。
对于我们的这个项目，材料要有两种功能：
1. 产生散射光(或者说吸收入射光)
2。如果散射，说明光线应该衰减多少

Material 类：

```
class Material
{
    public virtual bool Scatter(Ray rIn,HitRecord rec, ref Vector3 attenuation, ref Ray scattered)
    {
        return false;
    }
}
```

HitRecord 类的变化

```
class HitRecord
{
    private double t;
    private Vector3 p;
    private Vector3 normal;
    private Material material ;

    public double T { get => t; set => t = value; }
    internal Vector3 P { get => p; set => p = value; }
    internal Vector3 Normal { get => normal; set => normal = value; }
    internal Material Material { get => material; set => material =
value; }
}
```

这样做，材料会告诉我们光线是如何与表面相互作用的。Hitecord 是一个用几个参数组成的的类，这样我们就可以把它们作为一个组发送。当射线击中一个表面(例如一个特定的球体)时，HitRecord 中的材质指针将被设置为指向我们开始时在 main()中设置球体时给出的材质指针。当 color()例程获得 HitRecord 时，它可以调用材质指针的成员函数来找出哪条射线(如果有的话)是分散的。

漫反射材质（Lambertian 类）

对于我们已经有的朗伯(漫反射)情况，它要么总是散射，然后通过反射系数 R 衰减，要么它可以散射而不衰减，但吸收 1-R 部分的射线。也可能是这些策略的混合。对于朗伯材料，我们得到了这样一个简单的类:

```
class Lambertian : Material
{
    private Vector3 albedo;  //漫反射系数

    public Lambertian(Vector3 albedo)
    {
        this.Albedo = albedo;
    }

    internal Vector3 Albedo { get => albedo; set => albedo = value; }

    public override bool Scatter(Ray rIn, HitRecord rec, ref Vector3
attenuation, ref Ray scattered)
    {
```

```
            Vector3 target = rec.P + rec.Normal +
RTUtils.RandomInUnitSphere();
            scattered = new Ray(rec.P, target - rec.P);
            attenuation = albedo;
            return true;
        }
    }
```

这里有个漫反射系数 albedo，我为了方便就暂时用的 Vector 这个类来表示，你可以自己再去新建一个类来表示它。
什么是漫反射系数？

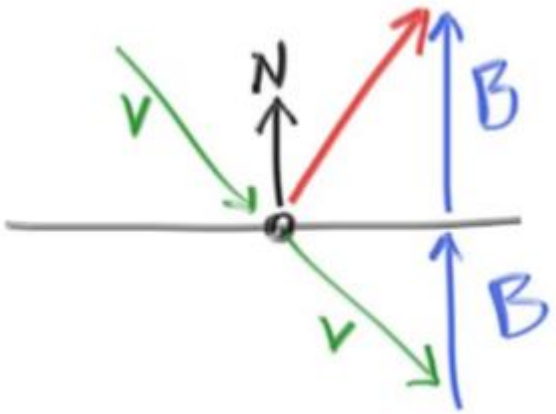
漫反射系数是指光照在一个物体的表面时，若该物体的表面为理想的平直面，除了被物体吸收的光通量外，其它光通量都被这个理想平面反射出来，此时称为全反射或纯反射。一般有漫反射系数的平方与镜面反射系数平方的和为一。

但实际的物体表面，从微观角度看，再平直的表面都存在凹凸不平，因此就存在光向四周漫射的现象。向四周漫射的光通量与总的反射光通量之比称为：漫射系数或漫反射系数。

光照在一个物体表面时，其能量分布与光源发光有关，若光源发出的光是均匀的（不一定要求要平行），则光在物体表面的能量分布也是均匀的。但物体表面反射出来的光的能量（光通量）就不一定是均匀的（除了理想平面外，从微观的角度看应该都是不均匀的）。

金属材质（Metal 类）

对于光滑金属，射线不会随机散射。关键的数学问题是:光线如何从金属表面反射的?



红色的反射光线方向就是(**v** + 2**B**),在我们的设计中，**N** 是单位向量，但 **v** 可能不是。**B** 的长度应该是点乘(**v**, **N**)。因为 **v** 的指向，我们需要一个负号来表示:

```
public static Vector3 Reflect(Vector3 v,Vector3 n)
{
    return v - 2 * Vector3.DotProduct(v, n) * n;
}
```
反射算法，不过多解释，画出图比划一下很容易理解的。

然后 Metal 类：
```
class Metal : Material
{
    private Vector3 albedo;   //漫反射率

    public Metal(Vector3 albedo)
    {
        this.Albedo = albedo;
    }
    internal Vector3 Albedo { get => albedo; set => albedo = value; }

    public override bool Scatter(Ray rIn, HitRecord rec, ref Vector3
attenuation, ref Ray scattered)
    {
        Vector3 reflected =
RTUtils.Reflect(Vector3.UnitVector(rIn.Direction), rec.Normal);
        scattered = new Ray(new Point3D(rec.P.X, rec.P.Y, rec.P.Z),
reflected);
        attenuation = albedo;
        return (Vector3.DotProduct(scattered.Direction, rec.Normal) >
0);
    }
}
```

Reflected 就是反射过后的光线。

改变 Color 方法

```
public static Color3D Color(Ray r, Hitable world, int depth)
{
    HitRecord rec = new HitRecord();
    if (world.Hit(r, 0.001, double.MaxValue, ref rec))
    {
        Ray scattered = new Ray();
        Vector3 attenuation = new Vector3();
```

```
        if (depth < 50 && rec.Material.Scatter(r, rec, ref
attenuation, ref scattered))
        {
            Color3D colorTemp = Color(scattered, world, depth + 1);
            colorTemp.R *= attenuation.X;
            colorTemp.G *= attenuation.Y;
            colorTemp.B *= attenuation.Z;
            return colorTemp;
        }
        else
        {
            return new Color3D(0, 0, 0);
        }
    }
    else
    {
        Vector3 unitDirection = Vector3.UnitVector(r.Direction);
        double t = 0.5 * (unitDirection.Y + 1.0);
        return new Color3D((1.0 - t) + t * 0.5, (1.0 - t) + t * 0.7,
(1.0 - t) + t * 1);
    }
}
```

这里计算反射光线采用的是递归方法来解决的，我们有个参数 depth 就是来控制反射次数的。每次反射过后就会衰减，所以剩以 attenuation。

改变球体 Shere 类

```
class Sphere : Hitable
{
    private double radius;
    private Point3D center;
    private Material material;

    public double Radius { get => radius; set => radius = value; }
    internal Point3D Center { get => center; set => center = value; }
    internal Material Material { get => material; set => material =
value; }

    public Sphere() { }

    public Sphere(Point3D center,double radius, Material material)
```

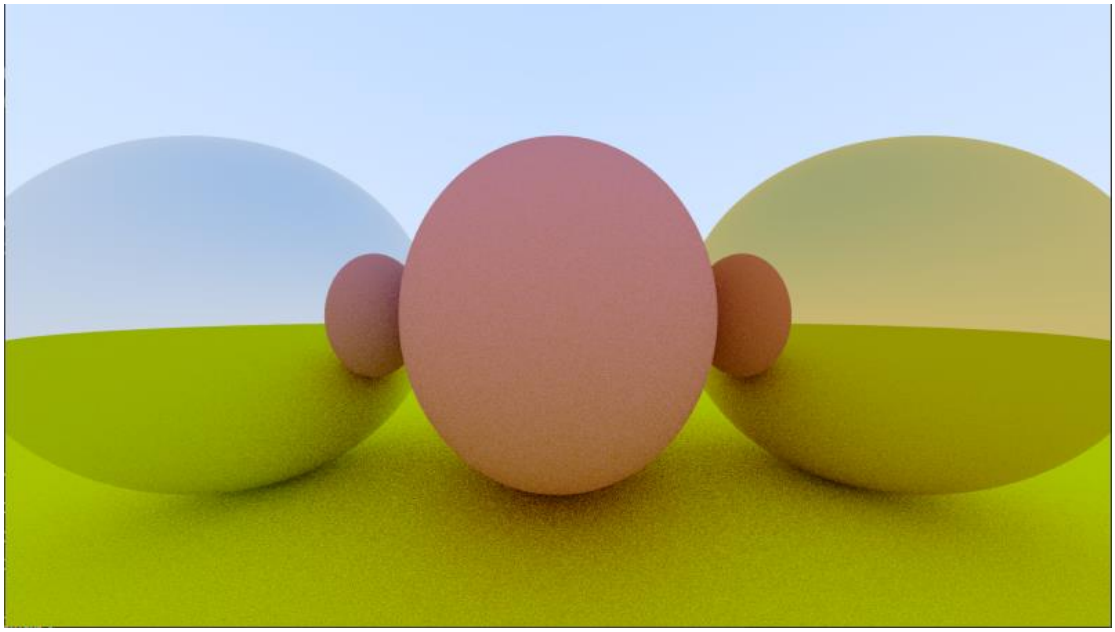
```
    {
        this.radius = radius;
        this.center = center;
        this.material = material;
    }

    public override bool Hit(RayTemp r, double tMin, double tMax, ref
HitRecord rec)
    {
        Vector3 oc = r.Origin - center;
        double a = Vector3.DotProduct(r.Direction, r.Direction);
        double b = Vector3.DotProduct(oc, r.Direction);
        double c = Vector3.DotProduct(oc, oc) - radius * radius;
        double discriminant = b * b - a * c;
        if(discriminant > 0)
        {
            double temp = (-b - Math.Sqrt(b * b - a * c)) / a;
            if(temp < tMax && temp > tMin)
            {
                rec.T = temp;
                rec.P = r.PointAtPara(rec.T);
                rec.Normal = (rec.P - center) / radius;
                rec.Material = material;
                return true;
            }
            temp = temp = (-b + Math.Sqrt(b * b - a * c)) / a;
            if (temp < tMax && temp > tMin)
            {
                rec.T = temp;
                rec.P = r.PointAtPara(rec.T);
                rec.Normal = (rec.P - center) / radius;
                rec.Material = material;
                return true;
            }
        }
        return false;
    }
}
```

就是增加了一个 Material 材料属性。

改变主函数：

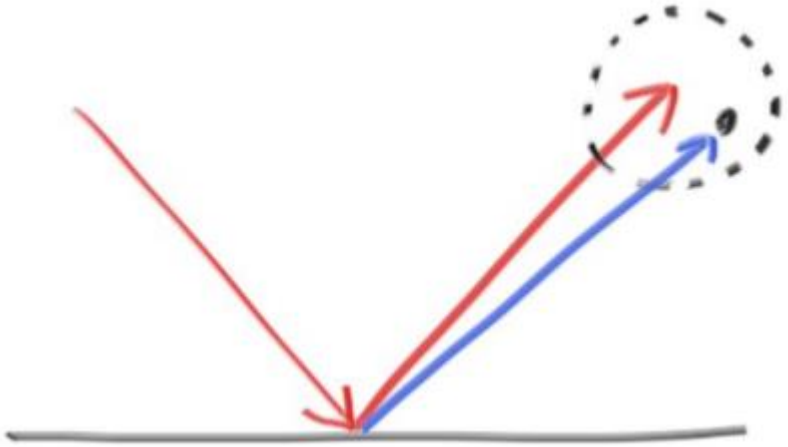
```
int nx = 1080;   //宽度
int ny = 1080;   //高度
int ns = 10;
Bitmap bitmap = new Bitmap(nx, ny);

Random rd = new Random();
List<Hitable> list = new List<Hitable>();
list.Add(new Sphere(new Point3D(0, 0, -1), 0.5, new Lambertian(new
Vector3(0.8, 0.3, 0.3))));
list.Add(new Sphere(new Point3D(0, -100.5, -1), 100, new
Lambertian(new Vector3(0.8, 0.8, 0.0))));
list.Add(new Sphere(new Point3D(1, 0, -1), 0.5, new Metal(new
Vector3(0.8, 0.6, 0.2))));
list.Add(new Sphere(new Point3D(-1, 0, -1), 0.5, new Metal(new
Vector3(0.8, 0.8, 0.8))));
HitableList world = new HitableList(list, 4);
Camera cam = new Camera();
for (int i = 0; i < nx; i++)
{
    for (int j = 0; j < ny; j++)
    {
        Color3D col = new Color3D();
        for (int s = 0; s < ns; s++)
        {
            double u = (double)(i + rd.NextDouble()) / (double)nx;
            double v = (double)(j + rd.NextDouble()) / (double)ny;
            Ray r = cam.GetRay(u, v);
            Color3D colTemp = RTUtils.Color(r, world, 0);
            col.R += colTemp.R;
            col.G += colTemp.G;
            col.B += colTemp.B;
        }
        col.R /= ns;
        col.G /= ns;
        col.B /= ns;
        col = new Color3D(Math.Sqrt(col.R), Math.Sqrt(col.G),
Math.Sqrt(col.B));
        int ir = (int)(255.99 * col.R);
        int ig = (int)(255.99 * col.G);
        int ib = (int)(255.99 * col.B);
        bitmap.SetPixel(i, ny - j - 1, Color.FromArgb(ir, ig, ib));
        this.BackgroundImage = bitmap;
    }
}
```

得到图片 1：



改进 Metal 类

我们也可以通过使用一个小球体和为光线选择一个新的端点来随机化反射方向：



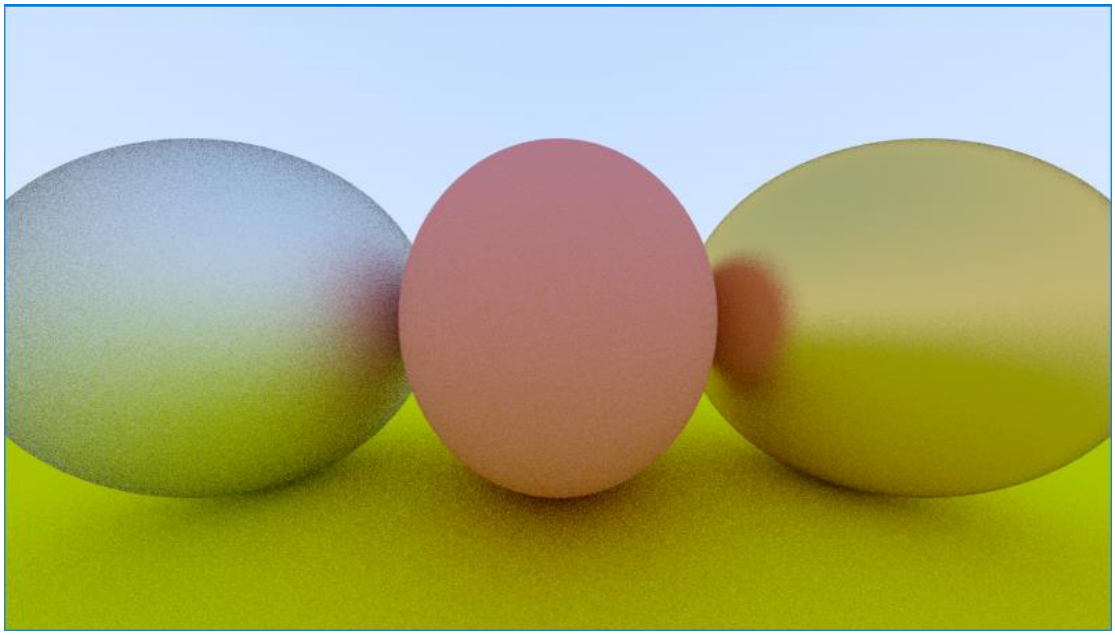
球体越大，反射就越模糊。我们添加一个模糊参数，它只是球体的半径（所以 0 不是扰动）。捕捉到的是，对于大球体或放牧射线，我们可能会散布在地表以下。我们可以让表面吸收这些。我们将在球的半径上加上 1 的最大值：

```
class Metal : Material
{
    private Vector3 albedo;   //漫反射率
    private double fuzz;   //模糊参数

    public Metal(Vector3 albedo, double fuzz)
    {
        this.Albedo = albedo;
        this.fuzz = fuzz;
    }

    public double Fuzz { get => fuzz; set => fuzz = value; }
    internal Vector3 Albedo { get => albedo; set => albedo = value; }

    public override bool Scatter(RayTemp rIn, HitRecord rec, ref
Vector3 attenuation, ref RayTemp scattered)
    {
        Vector3 reflected =
RTUtils.Reflect(Vector3.UnitVector(rIn.Direction), rec.Normal);
        scattered = new RayTemp(new Point3D(rec.P.X, rec.P.Y,
rec.P.Z), reflected + fuzz * RTUtils.RandomInUnitSphere());
        attenuation = albedo;
        return (Vector3.DotProduct(scattered.Direction, rec.Normal) >
0);
    }
}
```

我们可以尝试在金属中加入 0.3 和 1.0 的模糊度（fuzz）：

```
list.Add(new Sphere(new Point3D(1, 0, -1), 0.5, new Metal(new Vector3(0.8, 0.6, 0.2), 0.3)));
list.Add(new Sphere(new Point3D(-1, 0, -1), 0.5, new Metal(new Vector3(0.8, 0.8, 0.8), 1)));
```

得到图片 2：

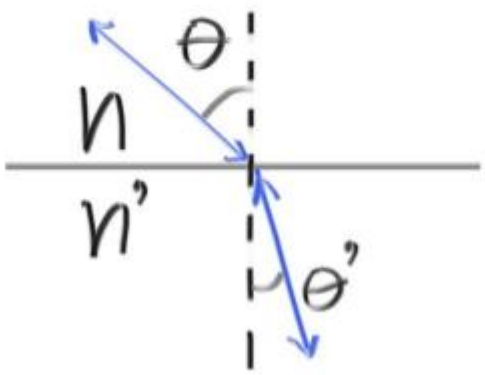


步骤 9：类玻璃材质（Dielectrics）

透明材料如水、玻璃和钻石都是电介质。当光线击中它们时，它分裂成反射光线和折射（透射）光线。我们将通过在反射或折射之间随机选择并在每次交互中仅生成一个散射光线来处理该问题。

涅尔定律

折射用斯涅尔定律来描述：
n sin(theta) = n' sin(theta')
其中 n 和 n'为折射率(通常空气= 1，玻璃= 1.3-1.7，钻石= 2.4)，几何描述为下图：



注：有种特殊情况——全反射（全内反射），基于这种情况，折射代码会比反射复杂：

```
public static bool Refract(Vector3 v,Vector3 n, double niOverNt, ref
Vector3 refracted)
{
    Vector3 uv = Vector3.UnitVector(v);
    double dt = Vector3.DotProduct(uv, n);
    double discriminant = 1.0 - niOverNt * niOverNt * (1 - dt * dt);
    if (discriminant > 0)
    {
        refracted = niOverNt * (uv - n * dt) - n *
Math.Sqrt(discriminant);
        return true;
    }
    else
        return false;
}
```

V 为光线，n 为法线，niOverNt 为折射率，refracted 折射光线。

Dielectrics 类

```
class DielectricsTemp : Material
{
    private double refIdx;  //相对折射率

    public DielectricsTemp(double refIdx)
```

```
    {
        this.RefIdx = refIdx;
    }

    public double RefIdx { get => refIdx; set => refIdx = value; }
    public override bool Scatter(Ray rIn, HitRecord rec, ref Vector3
attenuation, ref Ray scattered)
    {
        Vector3 outwardNormal = new Vector3();
        Vector3 reflected = RTUtils.Reflect(rIn.Direction,
rec.Normal);
        double niOverNt;
        attenuation = new Vector3(1, 1, 1);   //衰减率
        Vector3 refracted = new Vector3();
        if (Vector3.DotProduct(rIn.Direction, rec.Normal) > 0)
        {
            outwardNormal = rec.Normal * -1;
            niOverNt = refIdx;
        }
        else
        {
            outwardNormal = rec.Normal;
            niOverNt = 1.0 / refIdx;
        }

        if (RTUtils.Refract(rIn.Direction, outwardNormal, niOverNt,
ref refracted))
        {
            scattered = new Ray(rec.P, refracted);
        }
        else
        {
            scattered = new Ray(rec.P, reflected);
            return false;
        }
        return true;
    }
}
```

Scatter 方法还是做好他的工作：

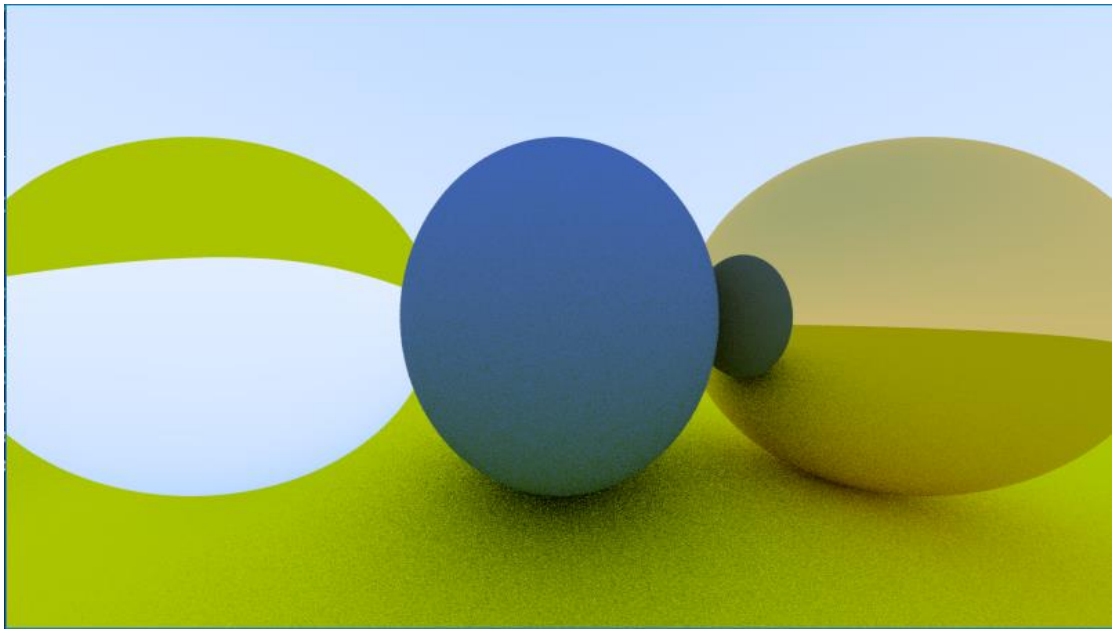
1. 产生散射光(或者说吸收入射光)
2. 如果散射，说明光线应该衰减多少

得到图片 1

```
int nx = 1080;   //宽度
int ny = 1080;   //高度
int ns = 10;
Bitmap bitmap = new Bitmap(nx, ny);

Random rd = new Random();
List<Hitable> list = new List<Hitable>();
list.Add(new Sphere(new Point3D(0, 0, -1), 0.5, new Lambertian(new
Vector3(0.1, 0.2, 0.5))));
list.Add(new Sphere(new Point3D(0, -100.5, -1), 100, new
Lambertian(new Vector3(0.8, 0.8, 0.0))));
list.Add(new Sphere(new Point3D(1, 0, -1), 0.5, new Metal(new
Vector3(0.8, 0.6, 0.2), 0.0)));
list.Add(new Sphere(new Point3D(-1, 0, -1), 0.5, new
Dielectrics(1.5)));
HitableList world = new HitableList(list, 4);
Camera cam = new Camera();
for (int i = 0; i < nx; i++)
{
    for (int j = 0; j < ny; j++)
    {
        Color3D col = new Color3D();
        for (int s = 0; s < ns; s++)
        {
            double u = (double)(i + rd.NextDouble()) / (double)nx;
            double v = (double)(j + rd.NextDouble()) / (double)ny;
            Ray r = cam.GetRay(u, v);
            Color3D colTemp = RTUtils.Color(r, world, 0);
            col.R += colTemp.R;
            col.G += colTemp.G;
            col.B += colTemp.B;
        }
        col.R /= ns;
        col.G /= ns;
        col.B /= ns;
        col = new Color3D(Math.Sqrt(col.R), Math.Sqrt(col.G),
Math.Sqrt(col.B));
        int ir = (int)(255.99 * col.R);
        int ig = (int)(255.99 * col.G);
        int ib = (int)(255.99 * col.B);
        bitmap.SetPixel(i, ny - j - 1, Color.FromArgb(ir, ig, ib));
```

```
    }
}
```

得到图片：



Dielectrics 类改进

真正的玻璃具有随角度变化的反射率——以一个陡峭的角度看一个窗口，它就会变成一面镜子。有一个不是很让人喜欢的公式，但几乎每个人都使用 Christophe Schlick 简单而令人惊讶的简单多项式近似：

Christophe Schlick 公式

```
public static double Schlick(double cosine,double refIdx)
{
    double r0 = (1 - refIdx) / (1 + refIdx);
    r0 = r0 * r0;
    return r0 + (1 - r0) * Math.Pow((1 - cosine), 5);
}
```

Dielectrics 类：

```
class Dielectrics : Material
{
    private double refIdx;  //相对折射率

    public Dielectrics(double refIdx)
    {
        this.RefIdx = refIdx;
    }

    public double RefIdx { get => refIdx; set => refIdx = value; }
    public override bool Scatter(RayTemp rIn, HitRecord rec, ref
Vector3 attenuation, ref RayTemp scattered)
    {
        Vector3 outwardNormal = new Vector3();
        Vector3 reflected = RTUtils.Reflect(rIn.Direction,
rec.Normal);
        double niOverNt;
        attenuation = new Vector3(1, 1, 1);   //衰减率
        Vector3 refracted = new Vector3();
        double reflect_prob;
        double cosine;
        if(Vector3.DotProduct(rIn.Direction,rec.Normal) > 0)
        {
            outwardNormal = rec.Normal * -1;
            niOverNt = refIdx;
            cosine = refIdx * Vector3.DotProduct(rIn.Direction,
rec.Normal) / rIn.Direction.Length();
        }
        else
        {
            outwardNormal = rec.Normal;
            niOverNt = 1.0 / refIdx;
            cosine = -Vector3.DotProduct(rIn.Direction, rec.Normal) /
rIn.Direction.Length();
        }

        if (RTUtils.Refract(rIn.Direction, outwardNormal, niOverNt,
ref refracted))
        {
            reflect_prob = RTUtils.Schlick(cosine, refIdx);
        }
```

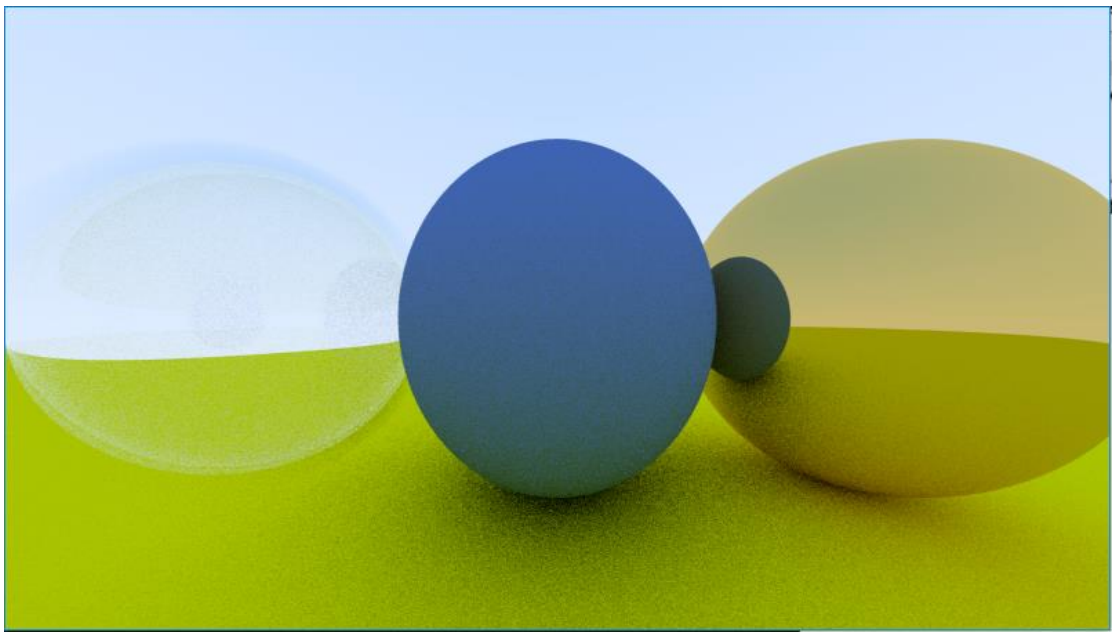
```
        else
        {
            scattered = new RayTemp(new Point3D(rec.P.X, rec.P.Y,
rec.P.Z), reflected);
            reflect_prob = 1.0;
        }
        if(RTUtils.rd.NextDouble() < reflect_prob)
        {
            scattered = new RayTemp(new Point3D(rec.P.X, rec.P.Y,
rec.P.Z), reflected);
        }
        else
        {
            scattered = new RayTemp(new Point3D(rec.P.X, rec.P.Y,
rec.P.Z), refracted);
        }
        return true;
    }
}
```

Scanner 的作用就不在赘述，大家根据前面的反射和公式结合着代码多多理解。

得到图片 2

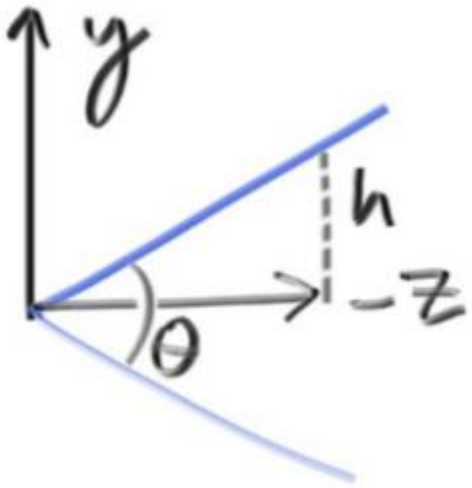
```
list.Add(new Sphere(new Point3D(0, 0, -1), 0.5, new Lambertian(new Vector3(0.1, 0.2, 0.5))));
list.Add(new Sphere(new Point3D(0, -100.5, -1), 100, new Lambertian(new Vector3(0.8, 0.8, 0.0))));
list.Add(new Sphere(new Point3D(1, 0, -1), 0.5, new Metal(new Vector3(0.8, 0.6, 0.2), 0.0)));
list.Add(new Sphere(new Point3D(-1, 0, -1), 0.5, new Dielectrics(1.5)));
list.Add(new Sphere(new Point3D(-1, 0, -1), -0.45, new Dielectrics(1.5)));
```

得到图片：



步骤 10：可移动相机（Positionable camera）

相机，就像电介质一样，调试起来很麻烦。所以我们慢慢来。首先，让我们允许一个可调视野（field of view ，fov），就是通过入口看到的角度。因为我们的图像不是正方形的，所以视场在水平和垂直方向上是不同的。我们先用垂直视野。
保持光线从原点射向 z=-1 平面。我们可以把它变成 z=-2 平面，或者其他，只要我们能得到 h 和这个距离的比值。如下设置：



有 h=tan（θ/2）。我们的相机现在变成：

```
class Camera
{
    private Point3D lowerLeftCorner;
    private Vector3 horizontal;
```

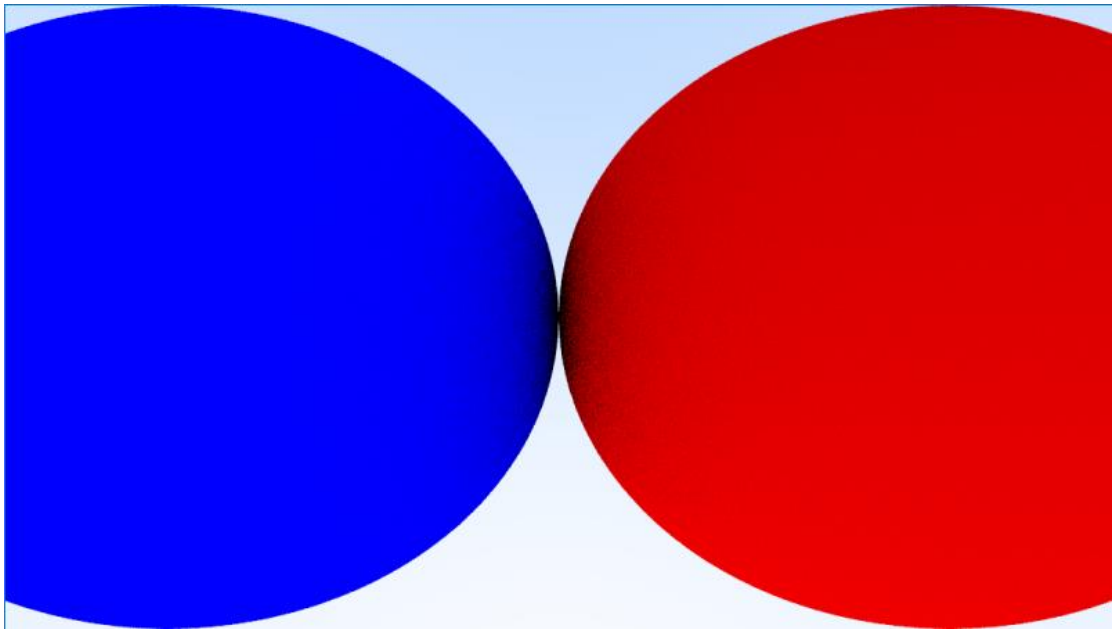
```
    private Vector3 vertical;
    private Point3D origin;

    public Camera(double vfov, double aspect)
    {
        double thete = vfov * Math.PI / 180;
        double halfHeight = Math.Tan(thete / 2);
        double halfWidth = aspect * halfHeight;
        lowerLeftCorner = new Point3D(-halfWidth, -halfHeight, -1.0);
        horizontal = new Vector3(2 * halfWidth, 0.0, 0.0);
        vertical = new Vector3(0.0, 2 * halfHeight, 0.0);
        origin = new Point3D(0.0, 0.0, 0.0);
    }

    internal Point3D LowerLeftCorner { get => lowerLeftCorner; set =>
lowerLeftCorner = value; }
    internal Vector3 Horizontal { get => horizontal; set =>
horizontal = value; }
    internal Vector3 Vertical { get => vertical; set => vertical =
value; }
    internal Point3D Origin { get => origin; set => origin = value; }

    public Ray GetRay(double s, double t)
    {
        return new Ray(origin, lowerLeftCorner + s * horizontal + t *
vertical - origin);
    }
}
```

参数 vfov，从上到下的度数

修改主函数

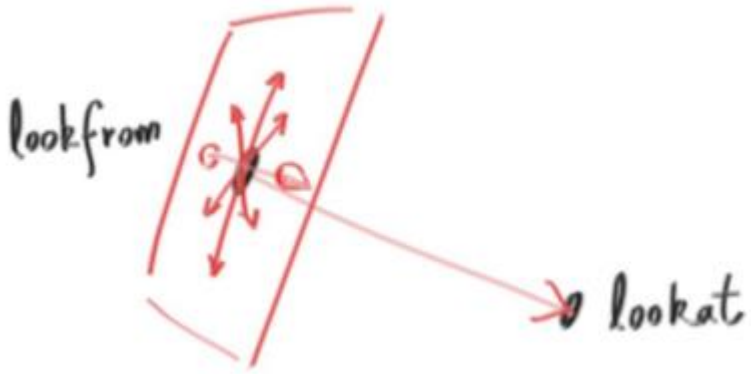
```
double R = Math.Cos(Math.PI / 4);
list.Add(new Sphere(new Point3D (-R, 0, -1), R, new Lambertian(new
Vector3(0, 0, 1))));
list.Add(new Sphere(new Point3D  (R, 0, -1), R, new Lambertian(new
Vector3(1, 0, 0))));
HitableList world = new HitableList(list, list.Count);
Camera cam = new Camera( 90, (double)(nx) / (double)(ny));
```

得到图片：

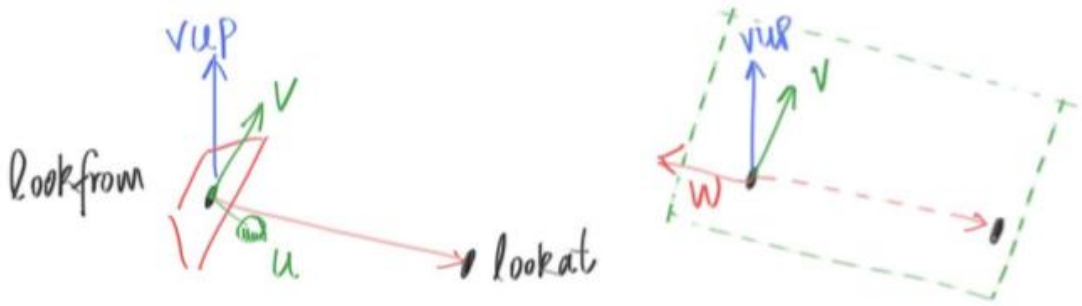


我们将放置摄像头的位置记作，我们查看的点为 lookat。

我们还需要一种方法来指定摄像机的侧倾或侧倾，绕着轴的旋转。另一种思考的方式是，即使你保持注视和注视不变，你仍然可以绕着鼻子转动你的头。我们需要的是一种为相机指定上方向向量的方法。注意，我们已经有了一个平面，上方向向量应该在其中，这个平面与视图方向垂直。



我们可以用任何向上的向量，简单地投影到这个平面上得到相机的向上的向量。我用“view up”(vup)命名。有几个叉积，并且我们有一个完整的正交基（u，v，w）来描述我们相机的方向。



记住 vup、v 和 w 都在同一平面上。任意视图相机面对-w，就像之前我们的固定相机面对-Z 一样，。不必使用 world up（0,1,0）来指定 vup，这很方便，也自然会保持相机水平，除非你尝试无理的相机角度。

修改 Camera 类:

```
class Camera
{
    private Point3D lowerLeftCorner;
    private Vector3 horizontal ;
    private Vector3 vertical ;
    private Point3D origin ;

    public Camera(Point3D lookfrom, Point3D lookat, Vector3 vup,
double vfov, double aspect)
    {
        Vector3 u = new Vector3();
        Vector3 v = new Vector3();
        Vector3 w = new Vector3();
        double thete = vfov * Math.PI / 180;
        double halfHeight = Math.Tan(thete / 2);
        double halfWidth = aspect * halfHeight;
        origin = lookfrom;
        w = Vector3.UnitVector(lookfrom - lookat);
        u = Vector3.UnitVector(Vector3.CrossProduct(vup, w));
        v = Vector3.CrossProduct(w, u);
        lowerLeftCorner = new Point3D(-halfWidth, -halfHeight, -1.0);
        Vector3 lowerLeftCornerTemp = origin - halfWidth * u -
halfHeight * v - w;
        lowerLeftCorner = new Point3D(lowerLeftCornerTemp.X,
lowerLeftCornerTemp.Y, lowerLeftCornerTemp.Z);
        Horizontal = 2 * halfWidth * u;
        Vertical = 2 * halfHeight * v;
    }

    internal Point3D LowerLeftCorner { get => lowerLeftCorner; set =>
```

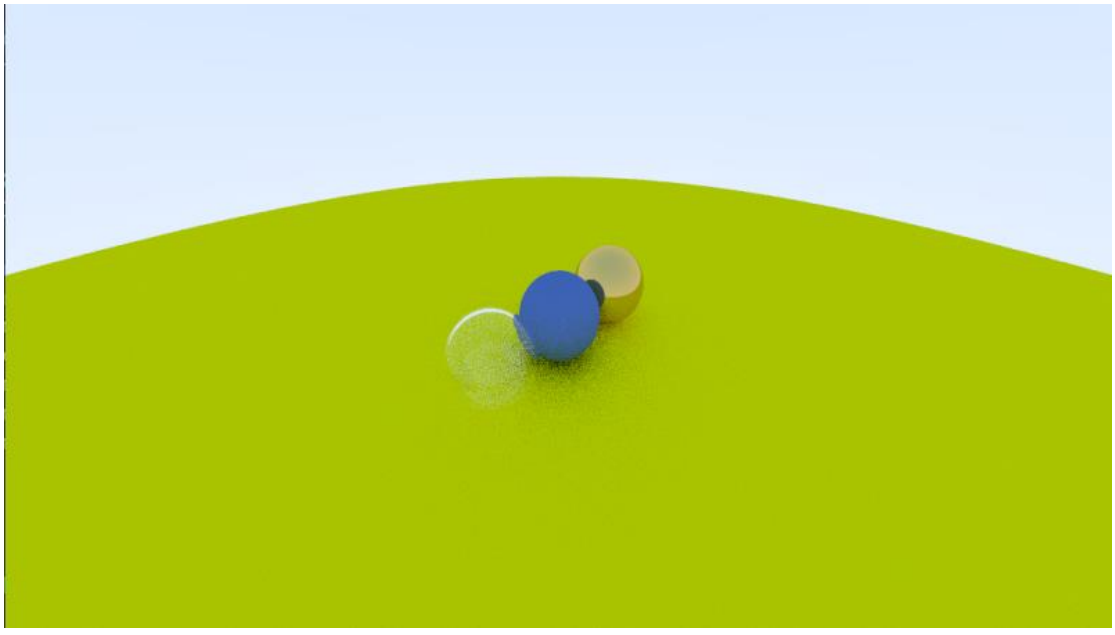
```
lowerLeftCorner = value; }
    internal Vector3 Horizontal { get => horizontal; set =>
horizontal = value; }
    internal Vector3 Vertical { get => vertical; set => vertical =
value; }
    internal Point3D Origin { get => origin; set => origin = value; }

    public Ray GetRay(double s, double t)
    {
        return new Ray(origin, lowerLeftCorner + s * horizontal + t *
vertical - origin);
    }
}
```

结合前面的讲解看代码。

修改主函数：

```
Camera cam = new Camera(new Point3D(-2, 2, 1), new Point3D(0, 0, -1), new
Vector3(0, 1, 0), 90, (double)(nx) / (double)(ny));
```

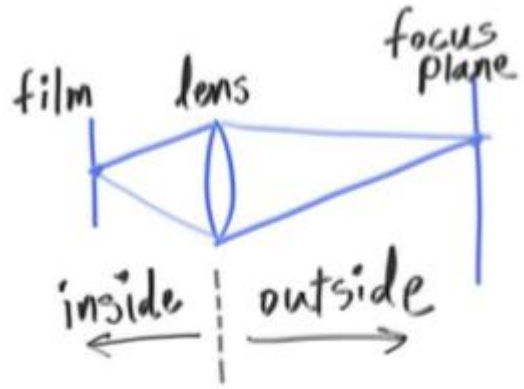
得到图像：



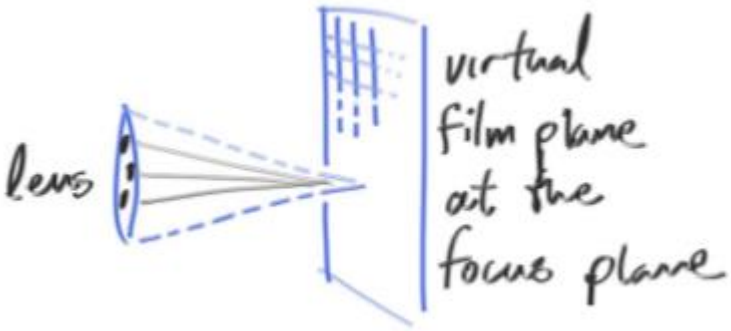
步骤 11：离焦模糊（Defocus Blur）

在真正的相机中，我们使焦距模糊的原因是因为它们需要一个大洞（而不仅仅是一个针孔）来收集光线。这会使所有的东西都散焦，但是如果我们把一个镜头插进洞里，就会有一定的距离，所有的东西都会聚焦。到物体聚焦平面的距离由透镜和胶片/传感器之间的距离控制。这就是为什么当你改变焦点时，镜头会相对相机移动。“光圈”是一个控制镜头有效大小的孔。对于一个真正的相机，如果你需要更多的光线，你会使光圈更大，会得到更多的离焦模糊。对于我们的虚拟相机，我们可以有一个完美的传感器，不需要更多的光线，所以我们只有一个光圈，当我们想要离焦模糊。

真正的相机有一个复杂的复合镜头。对于我们的代码，我们可以模拟顺序：传感器，然后是镜头，然后是光圈，计算出光线的发送位置，并在计算后翻转图像（图像被倒投影到胶片上）。图形用户通常使用薄透镜近似。



我们不管相机的内部结构，只需要模拟透镜，光线从透镜出来，然后通过找到胶片在聚焦平面上的投影（在距离教具处）将他发送到虚拟胶片平面。



为此，我们只需要让光线的原点在一个盘上，而不是从一个点看：

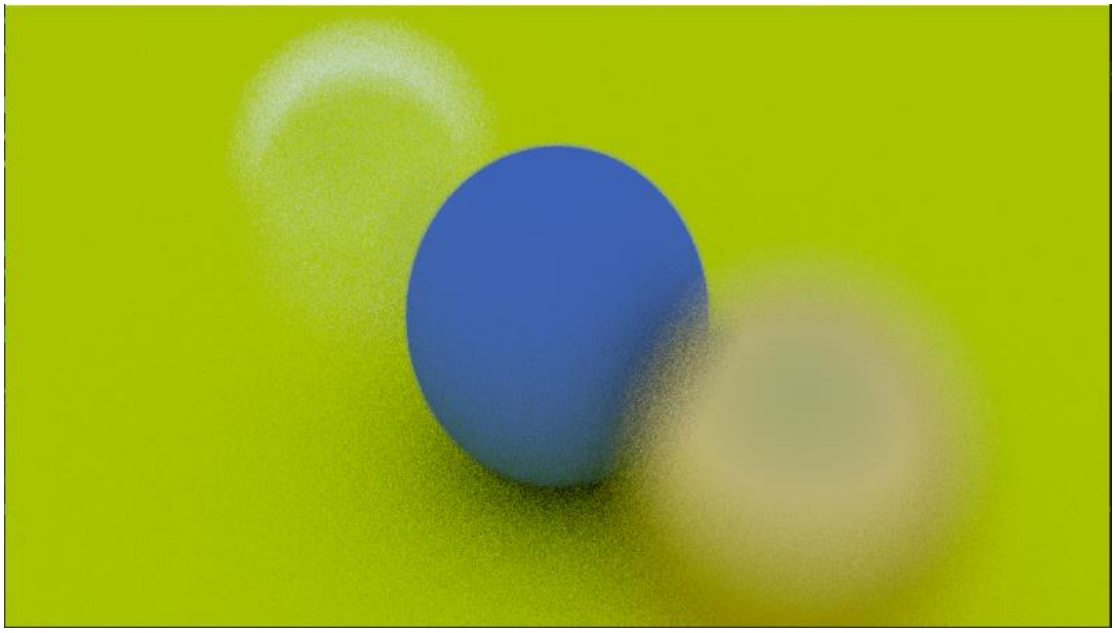
修改 Camera 类：

```
class Camera
```

```
{
    private Point3D lowerLeftCorner;
    private Vector3 horizontal ;
    private Vector3 vertical ;
    private Point3D origin ;
    private Vector3 u = new Vector3();
    private Vector3 v = new Vector3();
    private Vector3 w = new Vector3();
    private double lensRadius;

    public Camera(Point3D lookfrom, Point3D lookat, Vector3 vup,
double vfov, double aspect, double aperture, double focusDist)
    {
        lensRadius = aperture / 2;
        double theta = vfov * Math.PI / 180;
        double halfHeight = Math.Tan(theta / 2);
        double halfWidth = aspect * halfHeight;
        origin = lookfrom;
        w = Vector3.UnitVector(lookfrom - lookat);
        u = Vector3.UnitVector(Vector3.CrossProduct(vup, w));
        v = Vector3.CrossProduct(w, u);
        Vector3 lowerLeftCornerTemp = origin - halfWidth * focusDist * u -
halfHeight * focusDist * v - w * focusDist;
        lowerLeftCorner = new Point3D(lowerLeftCornerTemp.X,
lowerLeftCornerTemp.Y, lowerLeftCornerTemp.Z);
        horizontal = 2 * halfWidth * u * focusDist;
        Vertical = 2 * halfHeight * v * focusDist;
    }

    internal Point3D LowerLeftCorner { get => lowerLeftCorner; set =>
lowerLeftCorner = value; }
    internal Vector3 Horizontal { get => horizontal; set =>
horizontal = value; }
    internal Vector3 Vertical { get => vertical; set => vertical =
value; }
    internal Point3D Origin { get => origin; set => origin = value; }

    public Ray GetRay(double s, double t)
    {
        Vector3 rd = lensRadius * RTUtils.RandomInUnitDisk();
        Vector3 offset = u * rd.X + v * rd.Y;  //偏移量，为了方便用
Vector表示
        return new Ray(new Point3D(origin.X+offset.X, origin.Y+
offset.Y origin.Z + offset.Z), lowerLeftCorner + s * horizontal + t *
```

```
vertical - origin - offset);
    }
```

得到图片：



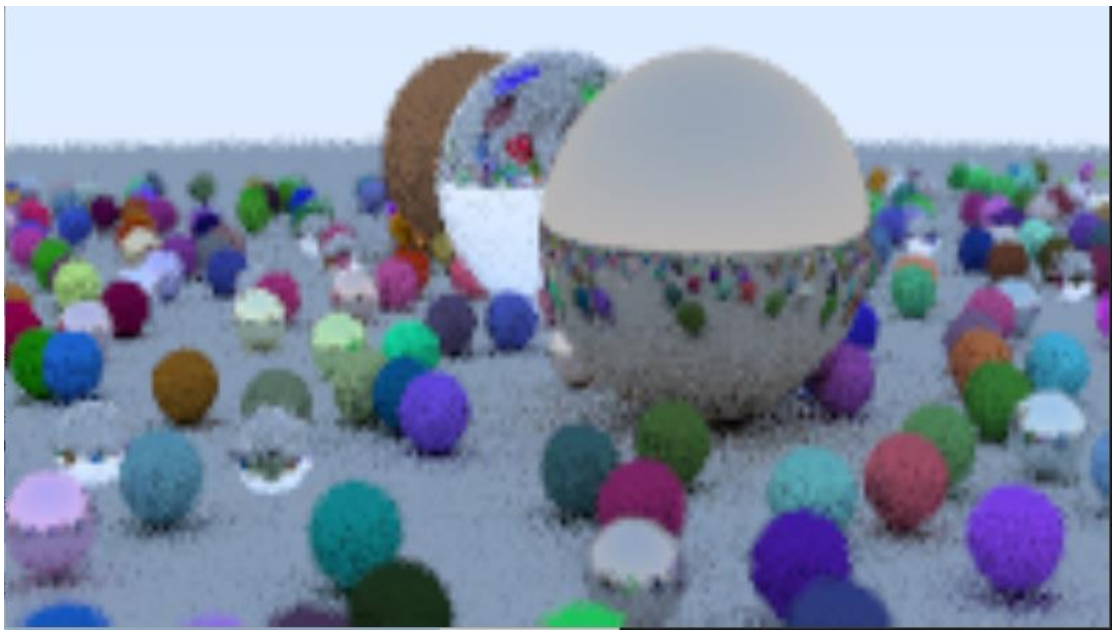
步骤 12：入门成品：

随机物体生成代码：

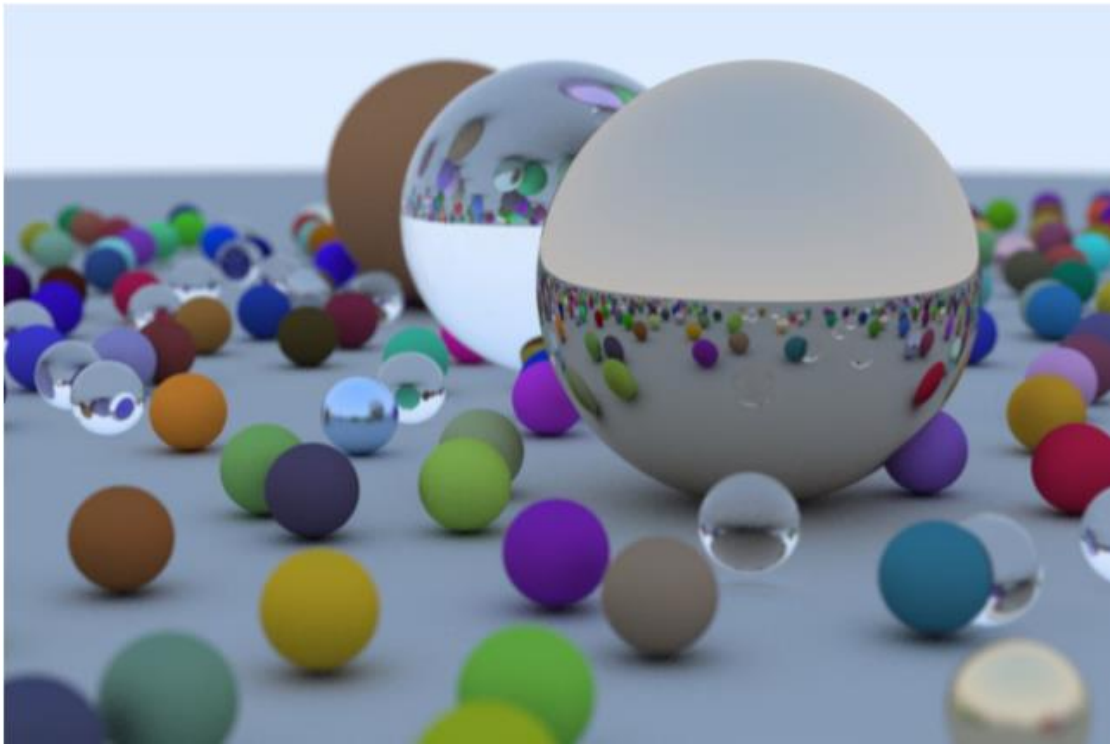
```
public static Hitable RandomScene()
{
    List<Hitable> list = new List<Hitable>();
    list.Add(new Sphere(new Vector3(0, -1000, 0), 1000, new
Lambertian(new Vector3(0.5, 0.5, 0.5))));
    for (int a = -11; a < 11; a++)
    {
        for(int b = -11; b< 11; b++)
        {
            double chooseMat = rd.NextDouble();
            Vector3 center = new Vector3(a + 0.9 * rd.NextDouble(),
0.2, b + 0.9 * rd.NextDouble());
            if ((center - new Vector3(4, 0.2, 0)).Length() > 0.9)
            {
                if(chooseMat < 0.8)
                {
```

```
                    list.Add(new Sphere(center, 0.2, new
Lambertian(new Vector3(rd.NextDouble() * rd.NextDouble(),
rd.NextDouble() * rd.NextDouble(), rd.NextDouble() *
rd.NextDouble()))));
                }
                else if(chooseMat < 0.95)
                {
                    list.Add(new Sphere(center, 0.2, new Metal(new
Vector3(0.5 * (1 + rd.NextDouble()), 0.5 * (1 + rd.NextDouble()), 0.5
* (1 + rd.NextDouble())), 0.5 * rd.NextDouble())));
                }
                else
                {
                    list.Add(new Sphere(center, 0.2, new
Dielectrics(1.5)));
                }
            }
        }
    }
    list.Add(new Sphere(new Vector3(0, 1, 0), 1.0, new
Dielectrics(1.5)));
    list.Add(new Sphere(new Vector3(-4, 1, 0), 1.0, new
Lambertian(new Vector3(0.4, 0.2, 0.1))));
    list.Add(new Sphere(new Vector3(4, 1, 0), 1.0, new Metal(new
Vector3(0.7, 0.6, 0.5),0)));
    HitableList lists = new HitableList(list, list.Count);
    return lists;
}
```

得到图片：



提高像素过后应该是这样的:



光线追踪入门结束。

本入门教程引用《Ray Tracing in One Weekend》 作者：Peter Shirley