光线追踪技术的理论和实践(面向对象)

Theory & Practice of Raytracing(Object Oriented)

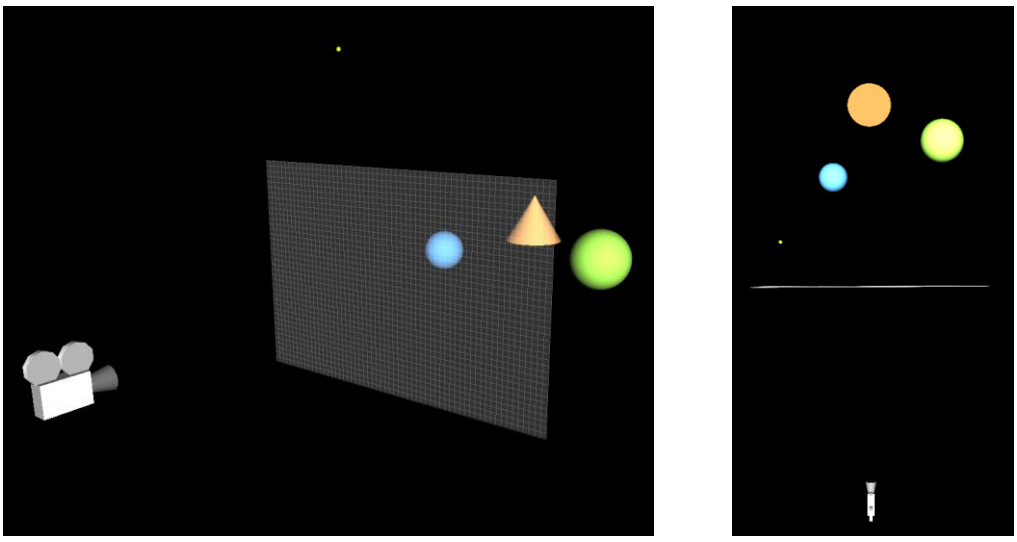
张 赐 zhangci226@hotmail.com

介绍

这篇文章将介绍光线追踪技术。在计算机图形领域中，这种技术被普遍应用于生成高质量的照片级图像。在为一个场景计算光照的时候，通过固定图形渲染管线可以计算 phong 光照模型，由于该模型的特征，使得渲染的物体看起来有塑料的质感。如果要渲染一个有金属质感且能反射周围环境的物体，phong 模型就无能为力了。和固定渲染管线相比，可编程图形渲染管线的力能要强的多，虽然可以实现很多逼真的光照效果，比如利用环境贴图来现实物体对环境的反射效果。但是这种环境反射只能反射出已经保存在 Cube Map 中的图像。在真实世界中，如果一个能反射周围环境的物体周围还有很多其他物体，它们就会相互反射。一般的环境贴图技术达不到这样的效果，于是在渲染照片级画面的时候，就要用到光线追踪的技术。文本还将利用 c++面向对象的方法来实现光线追踪。

原理

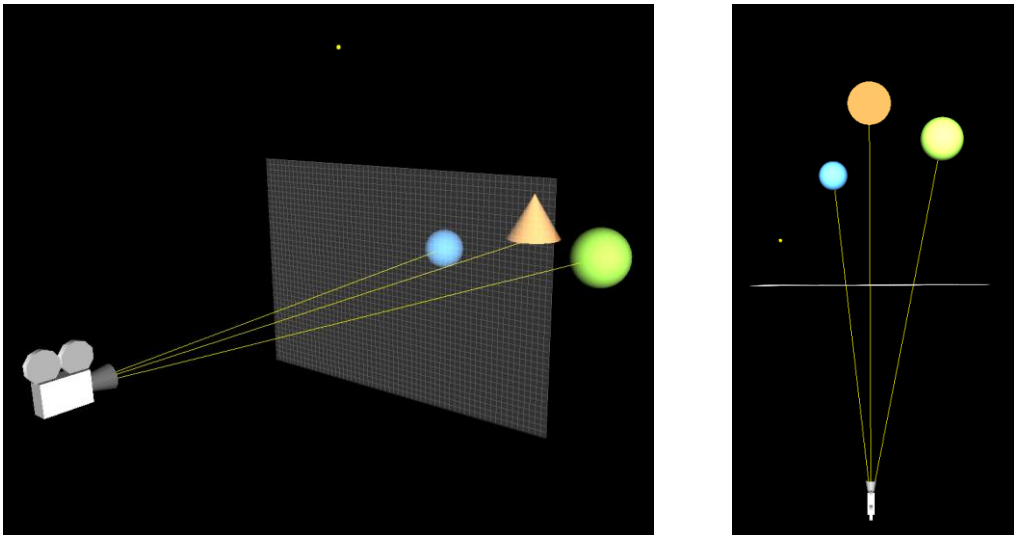
在介绍原理之前，先考虑一个问题：我们是怎样看到真实世界中的物体的？我们能看到物体，是因为该物体上有反射光线到达我们的眼睛。没有任何光线传入眼睛，我们就看不到任何东西。我们还经常看到一个物体表面能反射另一个物体。这也是因为被反射物体表面的反射光线到达该物体表面后，该物体继续将光线反射到我们的眼睛里，于是我们看到了该物体表面反射其他物体的效果。现在，我们将从物体表面出发最后到达眼睛的光线的方向反向。先来看看下面的 Fig1，在 Fig1 中是一个虚拟的场景，场景中有 2 个球和 1 个圆锥，白色的点代表光源，中间四边形就是虚拟屏幕，屏幕上一个一个的小方格就代表像素，相机的位置代表观察者眼睛的位置。



(a) **(b)**

Fig1 光线追踪场景

光线追踪的原理就是从相机的位置发出一条条通过每一个像素的射线，如果该射线和场景中的物体相交，那么就可以计算出该交点的颜色，这个颜色就是对应的像素的颜色。当然，计算像素颜色的时候首先要计算出交点处所有与光照计算相关量，比如法线，入射光线和反射光线等等。



(a) (b)

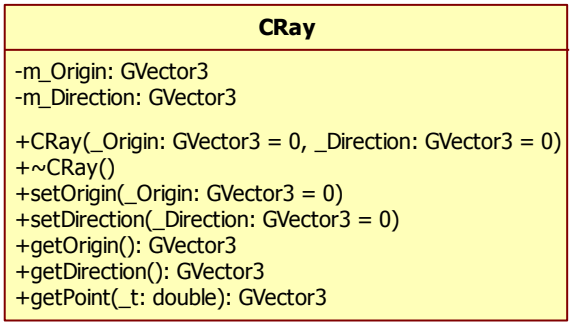
Fig2 光线和空间物体相交

在 Fig2 中可以看到，从相机出发的射线依次穿过每一个像素，图中显示出其中的三条。这些射线都与物体有交点，不同物体的交点计算方法也不一样。射线与平面的交点计算方法和射线与球的交点计算方法是截然不同的。为了计算方便，这里就只以球为例。如果一个物体可以反射周围的环境，那么当一条射线与该物体相交后，射线还会在该点产生反射和折射等。例如在 Fig2 中，当射线和蓝色球相交后，光线会反射，反射的光线又可能和橙色圆锥和绿色球相交，所以我们能在蓝色球的表面看到橙色的圆锥和绿色球。整个光线追踪的原理就是这么简单，但是实际操作起来又有很多要注意的地方。

实践

用面向对象的方法来实现光线追踪比使用面向结构要来的容易一些。因为在光线追踪的整个过程中，比较容易抽象出对象的共同特征，比如我们可以抽象出射线，物体，光源，材质等等。当然，最最基本的一个类就是向量类，在计算光照的时候向量很重要。在这里我们假设已经实现了一个三维向量类 GVector3，该类提供所有有关向量的操作。

除了向量，我们最先能想到一个关于射线的类，叫 CRay。



对于一条射线最基本的就是它的出发点和方向，所以在 CRay 的类图中，能看到两个私有成员变量

m_Origin 和 m_Direction，它们都是 GVector3 类型。由于类的设计原则要满足数据的封装性，既然射线的出发点和方向都是私有的，那么就要提供公共的成员方法来访问它们，于是我们还需要 set 和 get 方法。最后，getPoint(double)方法是通过向射线的参数方程传入参数 t 而获得在射线上的点。实现了射线 CRay 类后，那么在使用光线追踪计算每个像素颜色的时候，对于每一个像素都要创建一个 CRay 的实例。

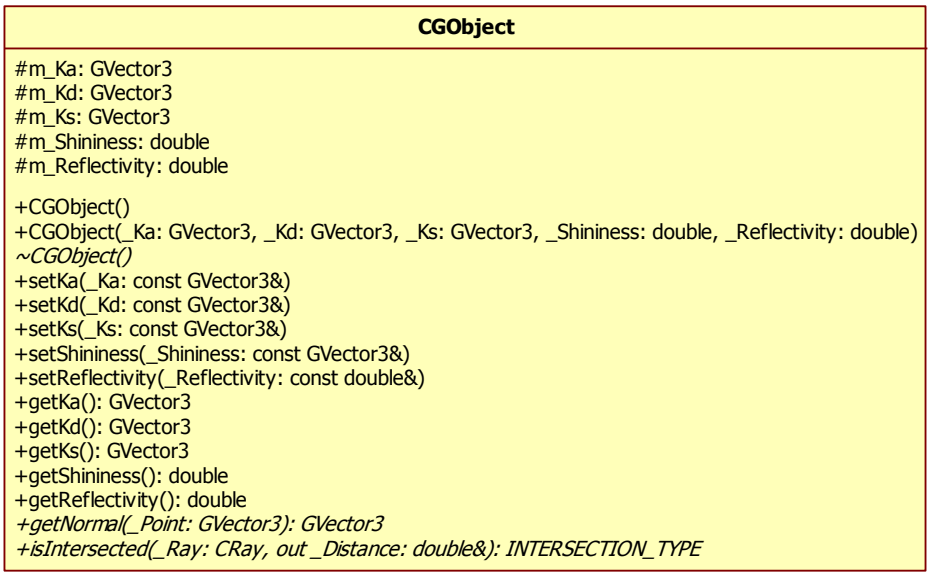
```
for(int y=0; y<=ImageHeight; y++)
{
     for(int x=0; x<=ImageWidth; x++)
     {
          double pixel_x = -20.0 +40.0/ImageWidth*x;
          double pixel_y = -15.0 +30.0/ImageHeight*y;

          GVector3 direction = GVector3(pixel_x, pixel_y,0)-CameraPosition;
          CRay ray(CameraPosition, direction);

          // call RayTracer function
     }
}
```

从上面的代码可以看到，两个 for 循环用于扫描每一个像素，然后在循环里计算出每个像素的位置。如果我们假设 Fig1 中，四边形屏幕处于 xy 平面，长和宽分别是 40 和 30，且左上顶点坐标和右下顶点坐标分别为(-20,15,0)和(20,-15,0)。为了将该屏幕映射到实际分辨率为 800*600 的窗口上，就要求出虚拟屏幕上每个像素的坐标 pixel_x 和 pixel_y。然后对每一个像素都用一条射线穿过它，射线的方向自然就是像素的位置和相机位置的差向量的方向。要注意一点，实际窗口的分辨率比例要和虚拟屏幕长宽比例保持一致，这样渲染出来的画面看起来长宽比例才正确。

现在我们来考虑在场景中的物体。一个物体可能有很多可以描述它的特征，比如形状，大小，颜色，材质等等。使用面向对象的方法，就需要将这些物体的共同特征抽象出来。下面是一个抽象出来的物体类 GCObject。



CGObject 类成员变量有五个，分别表示物体表面环境光反射系数(m_Ka)，漫反射系数(m_Kd)，镜面反射系数(m_Ks)，镜面反射强度(m_Shininess)和环境反射强度(m_Reflectivity)。前四个变量是计算光照所需要的最基本量，而环境反射强度表示该物体能反射环境的能力。这些成员变量都的类型都是protected，因为我们要把 CGObject 最为物体的基类，这些 protected 成员变量可以被该类的子类所继承。该类的所有 get 方法和 set 方法都能被子类继承，而且所有继承了该类的子类的方法都相同。该类还有两个虚成员函数，分别是 getNormal()和 isIntersected()。getNormal()函数的作用是获取物体表面一点的法线，它接受一个 GVector3 类型的参数_Point，并返回物体表面点_Point 处的法线。当然不同物体表面获得法线的方法是不一样的。比如，对于平面来说，平面上所有点的法线都是一样的。而对于球来说，球面上每一个的法线是球面上的该交点 $\boldsymbol{p}$ 和球心的 $\boldsymbol{c}$ 的差向量。

$$N_{Sphere} = p - c$$

所以将 getNormal()设置为虚成员函数就可以实现类的多态性，凡是继承了该方法的子类，都可以实现自己的 getNormal()方法。同样的道理，函数 isInserted 也是虚成员函数，该方法接受参数射线 CRay 和距离 Distance，CRay 是输入参数，用于判断射线和该物体的交点，Distance 是输出参数，如果物体和射线相交，则返回相机到该交点的距离。Distance 还应该有个很大初始值，表示在无限远处物体和射线相交，这种情况用于判断物体和射线没有交点。函数 isIntersected()还返回一个枚举类型 INTERSECTION_TYPE，定义如下：

```
enum INTERSECTION_TYPE {INTERSECTED_IN = -1, MISS = 0, INTERSECTED = 1};
```

其中 INTERSECTED_IN 表示射线从物体内部出发并和物体有交点，MISS 射线和物体没有交点，INTERSECTED 表示射线从物体外部出发并且和物体有交点。射线和不同物体交点的计算方法不同，于是该函数为虚函数，继承该函数的子类可以实现自己的 isIntersected()方法。下面的代码就可以判断一条射线和场景中所有物体的是否有交点，并且返回离相机最近的一个。

```
double distance = 1000000; // 初始化无限大距离
GVector3 Intersection;     // 交点
for(int i = 0; i<objects_numbers; i++)  // 遍历场景中每一个物体
{
     CGObject *obj = objects_list[i];
     if( obj->isIntersected(ray, distance) != MISS) // 判断是否有交点
     {
          Intersection = ray.getPoint(distance); //如果相交，求出交点保存到Intersection
     }
}
```

为了计算方便，这里就以球为例，创建一个 CSphere 的类，该类继承于 CGObject。

—

CSphere
-m_Center: GVector3 -m_Radius: double +CSphere() +CSphere(_Center: const GVector3&, _Radius: const double&) +CSphere(_copy: const CSphere&) ~CSphere() +setCenter(_Center: const GVector3&) +setRadius(_Radius: const double&) +getCenter(): GVector3 +getRadius(): double +isIntersected(_Ray: CRay, out _Distance: double&): INTERSECTION_TYPE +operator=(_copy: const CSphere&): CSphere&

作为球，只需要提供球心 Center 和半径 Radius 就可以决定它的几何性质。所以 CSphere 类只有两个私有成员变量。在所有成员函数中，我们重点来看看 isIntersected()方法。

```
INTERSECTION_TYPE CSphere::isIntersected(CRay _ray, double& _dist)
{
     GVector3 v = _ray.getOrigin() - m_Center;
     double b = -(v * _ray.getDirection());
     double det = (b * b) - v*v + m_Radius;
      INTERSECTION_TYPE retval = MISS;
     if (det > 0){
          det = sqrt(det);
          double t1 = b - det;
          double t2 = b + det;
          if (t2 > 0){
               if (t1 < 0) {
                    if (t2 < _dist) {
                         _dist = t2;
                         retval = INTERSECTED_IN;
                    }
               }
               else{
                    if (t1 < _dist){
                         _dist = t1;
                         retval = INTERSECTED;
                    }
               }
          }
     }
     return retval;
}
```

如果射线和球有交点，那么交点肯定在球面上。球面上的点 $\boldsymbol{P}$ 都满足下面的关系，

$$|\boldsymbol{P}-\boldsymbol{C}|=\boldsymbol{R}$$

很明显球面上的点和球心的差向量的大小等于球的半径。然后将射线的参数方程带入上面的公式，再利用求根公式判断解的情况。具体的方法这里就不详述了，有兴趣的同学可以参考另一篇文章“利用 OpenGL 实现 RayPicking”，这篇文章详细讲解了射线和球交点的计算过程。

现在我们实现了射线 CRay，球体 CSphere，还差一个重要的角色——光源。光源也是物体的一种，完全可以从我们的基类 CGObject 类继承。这里做一点区别，我们单独创建一个所有光源的基类 CLightSource，然后从它在派生出不同的光源种类，比如平行光源 DirectionalLight，点光源 CPointLight 和聚光源 CSpotLight。本文中只详细讲解平行光源的情况，其他两种光源有兴趣的同学可以自己实现。

CLightSource

#m_Position: GVector3
#m_Ka: GVector3
#m_Kd: GVector3
#m_Ks: GVector3

+CLightSource()
+CLightSource(_Position: GVector3&, _Ka: const GVector3&, _Kd: const GVector3&, _Ks: const GVector3&)
~CLightSource()
+setPosition(_Position: const GVector3&)
+setKa(_Ka: const GVector3&)
+setKd(_Kd: const GVector3&)
+setKs(_Ks: const GVector3&)
+getPosition(): GVector3
+getKa(): GVector3
+getKd(): GVector3
+getKs(): GVector3
+EvalAmbient(_material_Ka: GVector3): GVector3
+EvalDiffuse(_material_Kd: GVector3, _N: GVector3, _L: GVector3): GVector3
+EvalSpecular(_material_Ks: GVector3, _material_Shininess: double, _N: GVector3, _L: GVector3, _V: GVector3): GVector3

类 CLightSource 的成员变量有四个，分别表示光源的位置，光源的环境光成分，漫反射成分和镜面反射成分。同样地，所有的 set 和 get 方法都为该类的子类提供相同的功能。最后也有三个虚成员函数，EvalAmbient()，EvalDiffuse()和 EvalSpecular()，它们名字分别说明它们的功能，并且都返回 GVector3 类型的值——颜色。由于对于不同种类的光源，计算方法可能不同，于是将它们设置为虚函数为以后的扩展做准备。笔者这里将光照计算放在了光源类里面，当然你也可以放在物体类 CGObject 里，也可以单独写一个方法，将光源和物体作为参数传入，计算出颜色后最为返回值返回。具体使用哪一种好还是要根据具体情况具体分析。

CDirectionalLight

-m_Direction: GVector3

+CDirectionalLight()
+CDirectionalLight(_Direction: const GVector3&)
+CDirectionalLight(_copy: const CDirectionalLight&)
~CDirectionalLight()
+setDirection(_Direction: const GVector3&)
+getDirection(): GVector3
+EvalAmbient(_material_Ka: GVector3): GVector3
+EvalDiffuse(_material_Kd: GVector3, _N: GVector3, _L: GVector3): GVector3
+EvalSpecular(_material_Ks: GVector3, _material_Shininess: double, _N: GVector3, _L: GVector3, _V: GVector3): GVector3
+operator=(_copy: const CDirectionalLight&): CDirectionalLight&

上面的平行光源类 CDirectionalLight 是 CLightSource 的子类，它继承了父类三个虚函数方法。下面来看看这三个函数的具体实现。

环境光的计算是最简单的，将物体材质环境反射系数和光源的环境光成分相乘即可。

$$ambient = I_a \cdot K_a$$

计算环境光的代码如下

```
GVector3 CDirectionalLight::EvalAmbient(const GVector3& _material_Ka)
{
      return GVector3(m_Ka[0]*_material_Ka[0],
                      m_Ka[1]*_material_Ka[1],
                      m_Ka[2]*_material_Ka[2]);
}
```

漫反射的计算稍微比环境光复杂，漫反射的计算公式为

$$diffuse = I_d \cdot K_d \cdot (N \cdot L)$$

其中，I_d 是光源的漫反射成分，K_d 是物体的漫反射系数，N 是法线，L 是入射光向量。

```
GVector3 CDirectionalLight::EvalDiffuse(const GVector3& _N, const GVector3& _L, const GVector3& _material_Kd)
{
     GVector3 IdKd = GVector3( m_Kd[0]*_material_Kd[0],
                               m_Kd[1]*_material_Kd[1],
                               m_Kd[2]*_material_Kd[2]);

     double NdotL = MAX(_N*_L, 0.0);
     return IdKd*NdotL;
}
```

镜面反射的计算又比环境光要复杂，镜面反射的计算公式为

$$specular = I_s \cdot K_s \cdot (V \cdot R)^n$$

其中

$$R = 2(L \cdot N) \cdot N - L$$

I_s 是光源镜面反射成分，K_s 是物体的镜面反射系数，V 是相机方向向量，R 是反射向量，n 就反射强度 Shininess。为了提高计算效率，也可以利用 HalfVector H 来计算镜面反射。

$$specular = I_s \cdot K_s \cdot (N \cdot H)^n$$

其中

$$H = (L+V)/2$$

计算 H 要比计算反射向量 R 要快得多。

```
GVector3 CDirectionalLight::EvalSpecluar(const GVector3& _N, const GVector3& _L, const GVector3& _V,
                                         const GVector3& _material_Ks, const double& _shininess)
{
     GVector3 IsKs = GVector3( m_Ks[0]*_material_Ks[0],
                               m_Ks[1]*_material_Ks[1],
                               m_Ks[2]*_material_Ks[2]);

     GVector3 H = (_L+_V).Normalize();

     double NdotL = MAX(_N*_L, 0.0);
     double NdotH = pow(MAX(_N*H, 0.0), _shininess);
```

```
     if(NdotL<=0.0)
          NdotH = 0.0;

     return IsKs*NdotH;
}
```

分别计算出射线和物体交点处的环境光，漫反射和镜面反射后，那么该射线对应像素的颜色 ***c*** 为

$$C = ambient + diffuse + specular$$

于是，我们可以在代码中添加一个方法叫 Tracer()，该方法就是遍历场景中的每个物体，判断射线和物体的交点，然后计算交点的颜色。

```
GVector3 Tracer(CRay R)
{
     GVector3 color;
     for(/*遍历每一个物体*/)
     {
          if(/*如果有交点*/)
          {
               GVector3 p = R.getPoint(dist);
               GVector3 N = m_pObj[k]->getNormal(p);
               N.Normalize();
               for(/*遍历每一个光源*/)
               {

                    GVector3 ambient = m_pLight[m]->EvalAmbient(m_pObj[k]->getKa());
                    GVector3 L = m_pLight[m]->getPosition()-p;
                    L.Normalize();
                    GVector3 diffuse = m_pLight[m]->EvalDiffuse(N, L, m_pObj[k]->getKd());
                    GVector3 V = m_CameraPosition - p;
                    V.Normalize();
                    GVector3 specular = m_pLight[m]->EvalSpecluar(N, L, V, m_pObj[k]->getKs(),
m_pObj[k]->getShininess());

                    color = ambient + diffuse + specular;
               }
          }
      }
}
```

如果要渲染可以反射周围环境的物体，就需要稍微修改上面的 Tracer()方法，因为反射是一个递归的过程，一但一条射线被物体反射，那么同样的 Tracer()方法就要被执行一次来计算被反射光线和其他物体是否还有交点。于是，在 Tracer()方法中再传入一个代表递归迭代深度的参数 depth，它表示射线与物体相交后反射的次数，如果为 1，说明射线与物体相交后不反射，为 2 表示射线反射一次，以此类推。

```
Tracer(CRay R, int depth)
{
     GVector3 color;
     // 计算 C = ambient + diffuse + specular
     if(TotalTraceDepth == depth)
               return color;
     else
     {
          //计算射线和物体交点处的反射射线 Reflect;

          GVector3 c = Tracer(Reflect, ++depth);
          color += GVector3(color[0]*c[0],color[1]*c[1],color[2]*c[2]);
          return color;
     }
}
```

创建一个场景，然后执行代码，可以看到下面的效果。

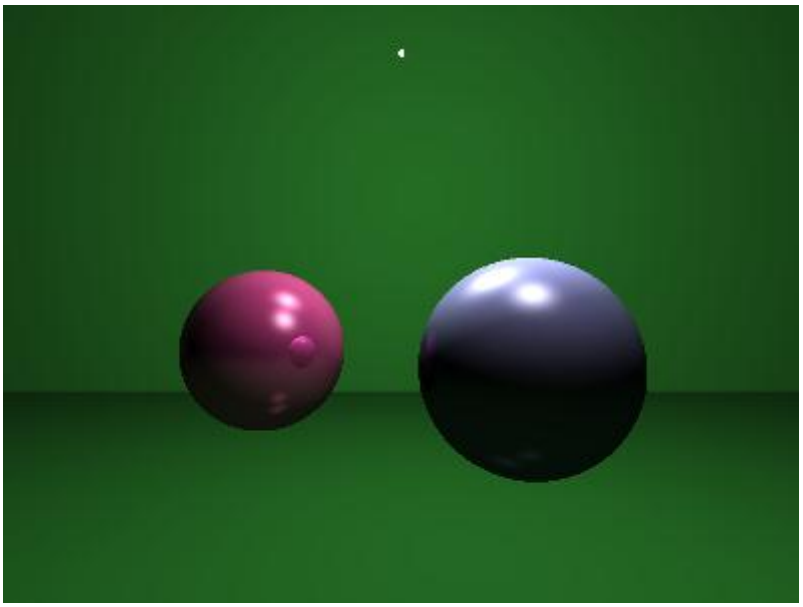


Fig3 光线追踪渲染的场景 1

如果设置 Tracer 的递归深度大于 2 的话，就可以看到两个球相互反射的情况。虽然这个光线追踪可以正常的执行，但是画面看起来总觉得缺少点什么。仔细观察你会发现画面虽然有光源，但是物体没有阴影，阴影可以增加场景的真实性。要计算阴影，我们应该从光源的出发，从光源出发的射线和物体如果有交点，而且这条射线与多个物体相交，那么除第一个交点外的后面所有交点都处于阴影中，这点很容易理解。于是，我们需要修改部分代码。

```
GVector3 Tracer(CRay R, int depth)
{
     GVector3 color;
     double shade = 1.0
     for(/*遍历每一个物体*/)
     {
          for(/*遍历每一个光源*/)
          {
```

```
                GVector3 L = pObj[k]->getCenter() - Intersection;
                double dist = norm(L);
                L *= (1.0f / dist);
                CRay r = CRay( Intersection,L );
                for ( /*遍历每一个物体*/ )
                {
                     CGObject* pr = pObj[s];
                     if (pr->isIntersected(r, dist)!=MISS)
                     {
                          shade = 0;
                          break;
                     }
                }
           }
     }
     if(shade>0)
     {
          // 计算C = ambient + diffuse + specular
          // 递归计算反射
     }
      return color*shade;
}
```

增加了阴影计算后，再运行程序，就能看到下面的效果。

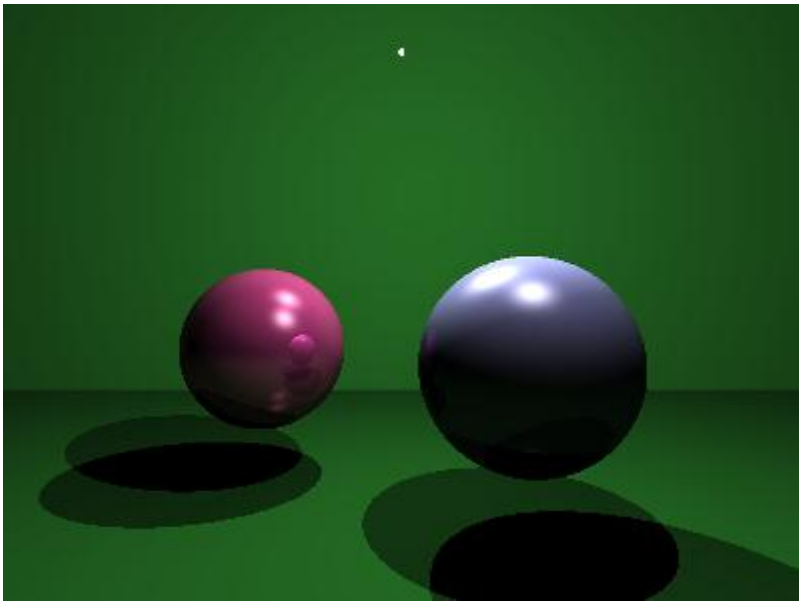


Fig4 光线追踪渲染的场景 2

最后我们也可以让地面反射物体，然后再墙上添加很多小球，让画面变得复杂一些，如下图。

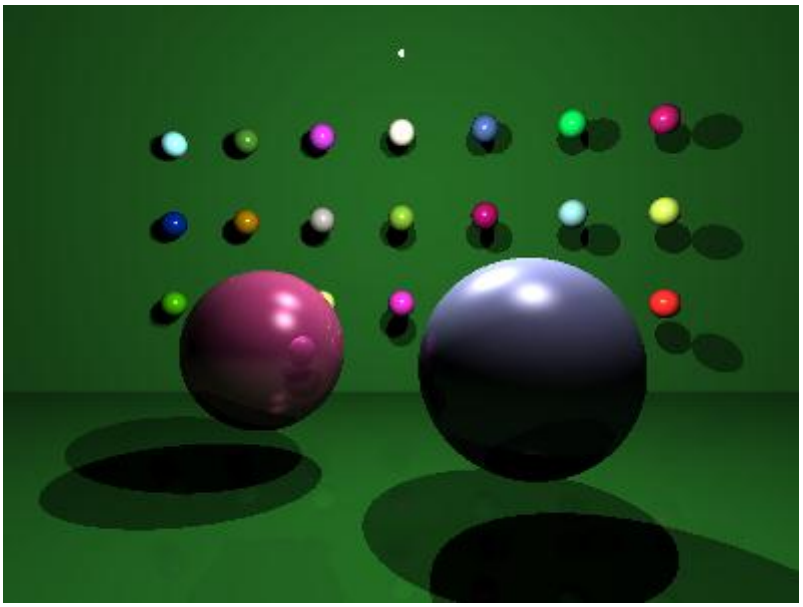
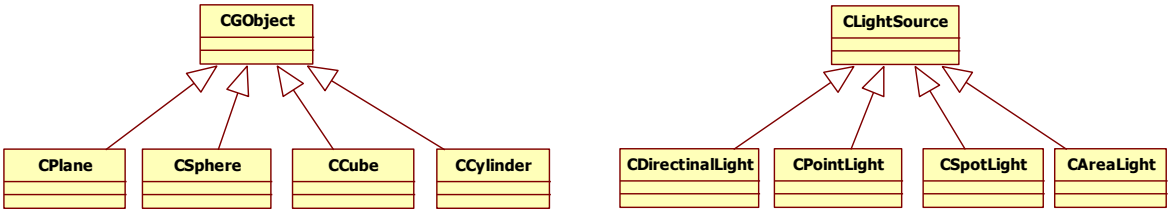


Fig5 光线追踪渲染的场景 3

总结

这篇文章通过利用面向对象的方法来实现了光线追踪渲染场景。利用面向对象的方法来实现光线追踪使程序的扩展性得到增强，渲染复杂的场景或者复杂的几何物体的时候，或者有很多光源和复杂光照计算的时候，只需要从基类继承，然后利用多态性来实现不同物体的不同渲染方法。



从上面的类图可以看到，利用面向对象的方式可以很容易扩展程序。而且，由于光线追踪的这种结构，不论添加多少物体在场景中，不论物体多么复杂，这种结构总能很好地渲染出正确的画面。

但是，对光线追踪来说，越复杂的场景需要的渲染时间越长。有的时候渲染一帧的画面甚至需要几天的时间。所以好的算法和程序结构对于光线追踪来说是很重要的，可以通过场景管理、使用 GPU 或 CUDA 等等技术来提高渲染效率。