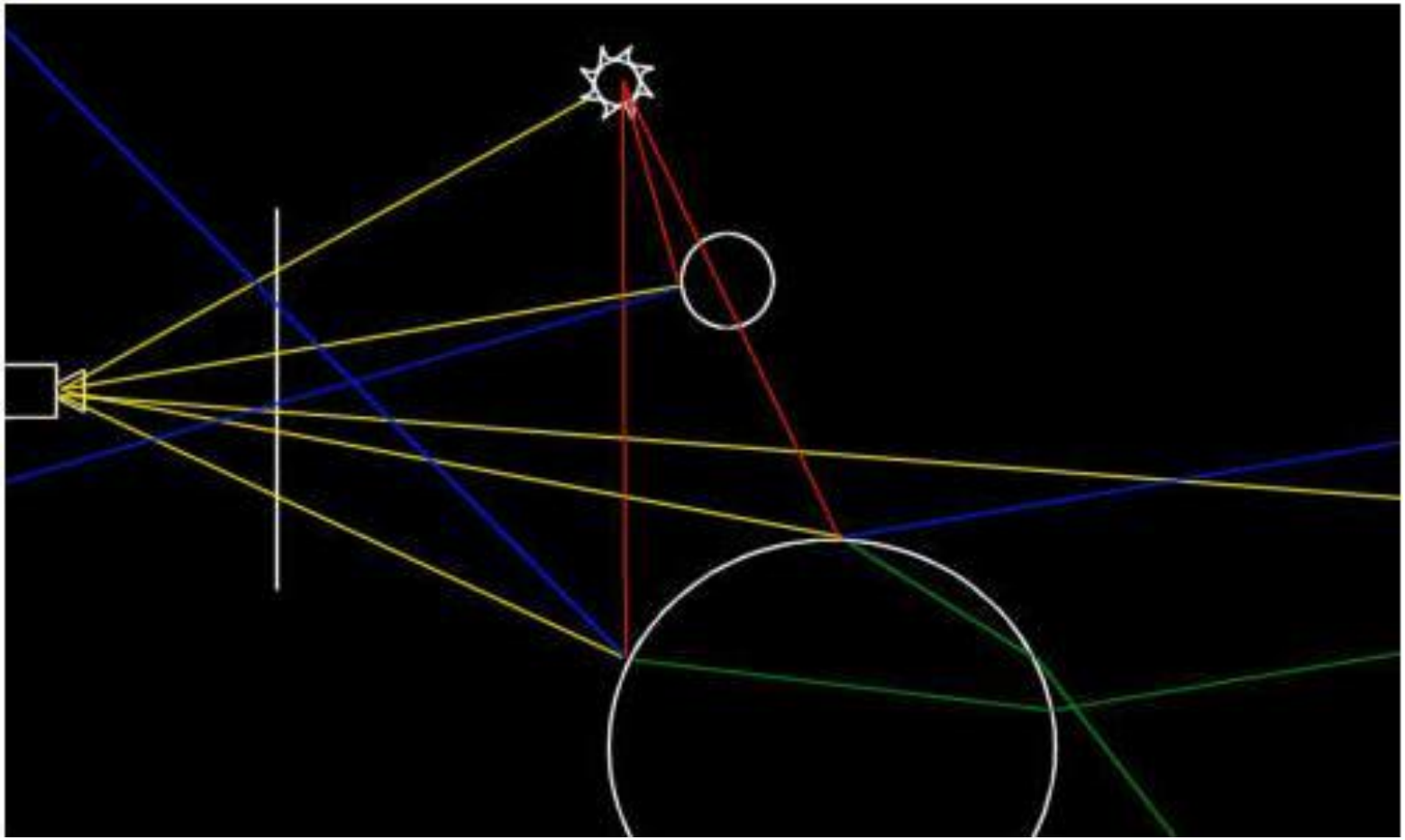
《高等计算机图形学》习题课

光线跟踪实现中的一些细节

张方略



- 跟踪从照相机中发出的黄线的话，每条黄线都可以产生出一系列 secondary rays：一条反射光线，一条折射光线，并为每个光源产生一条阴影线。

- 这些光线产生后（除了阴影线外）都可以被视为普通的光线。

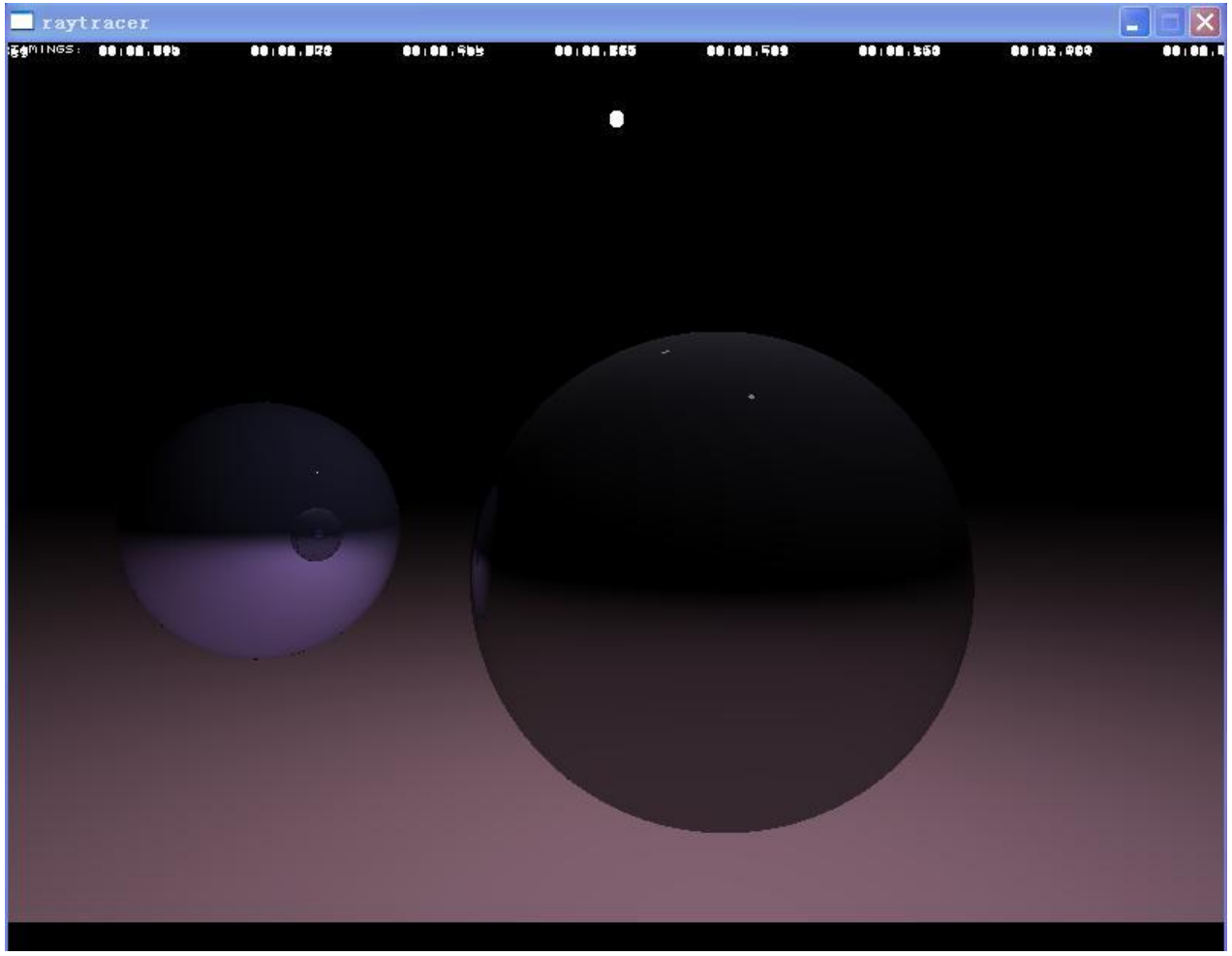
这意味着一条反射光线可以再被反射和折射，这种方法叫做"递归光线跟踪"。每条新产生的光线都增加了它先前光线聚集的地方的颜色，最终每条光线都对最开始由 primary ray 穿过的像素点的颜色做出了自己的贡献。

反射光线的跟踪

为了对于一个已知的平面法向量时对一条光线进行反射，采用下面的方法

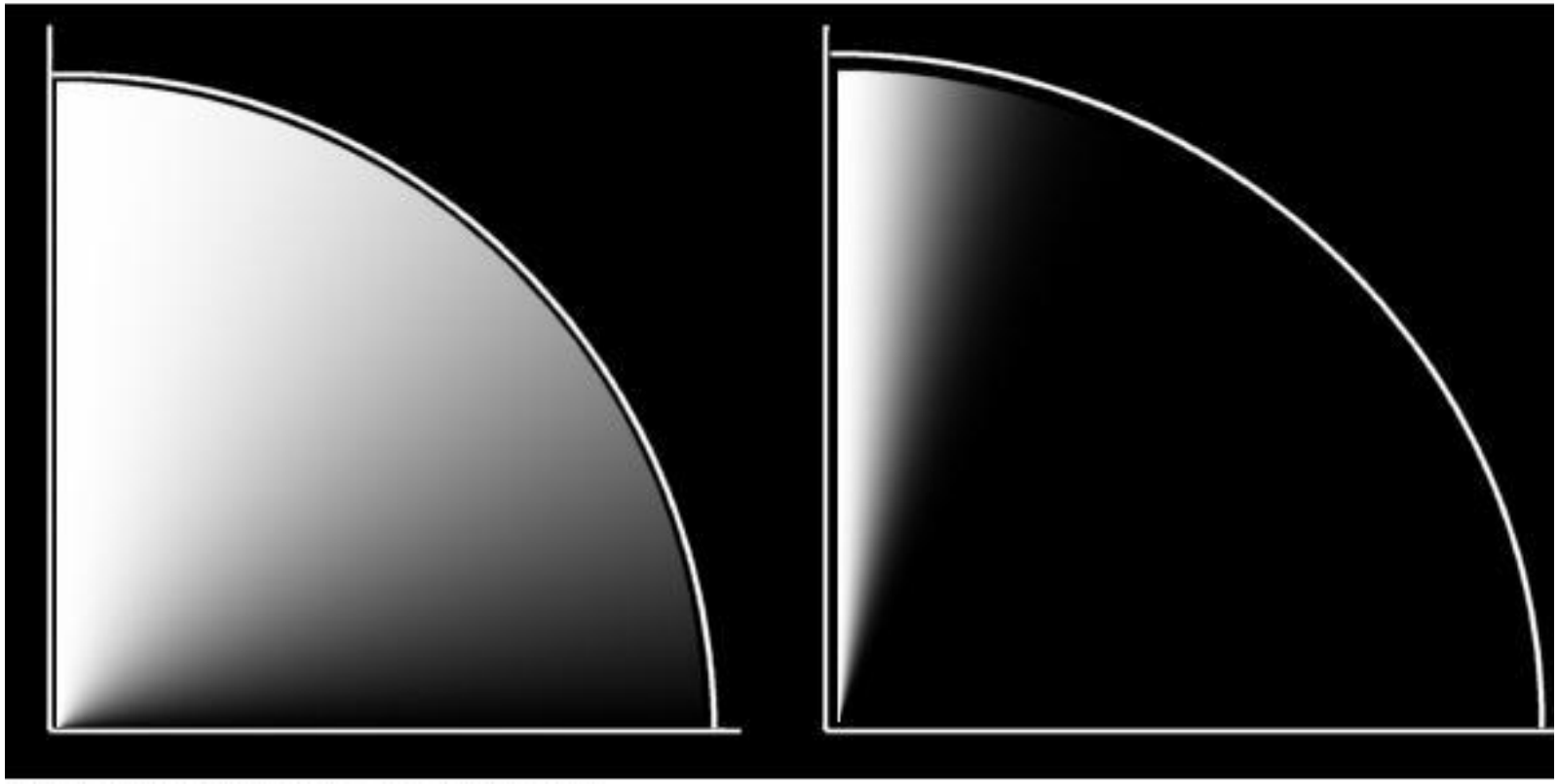
R--被反射的向量，V--入射光向量，N--平面法向量

```
        // calculate reflection
        float refl = prim->GetMaterial()->GetReflection();
        if (refl > 0.0f)
        {//几何体材质的反射度大于 0
            vector3 N = prim->GetNormal( pi );//几何体的法向量
            vector3 R = a_Ray.GetDirection() - 2.0f * DOT( a_Ray.GetDirection(), N ) * N;
            if (a_Depth < TRACEDEPTH)
            {//层数还没到上限
                Color rcol( 0, 0, 0 );
                float dist;
                Raytrace( Ray( pi + R * EPSILON, R ), rcol, a_Depth + 1, a_RIndex, dist );
                a_Acc += refl * rcol * prim->GetMaterial()->GetColor();//颜色值中加入反射光的贡献值
            }
        }
```



注意两个球彼此之间是会互相反射的，而且球也会反射地表平面。

Phong 光照模型



镜面高亮区是对光源的散射性反射。Phong 于是提出的光照模型，用来产生高光的效果。

$$intensity = diffuse * (L \cdot N) + specular * (V \cdot R)^n$$

L----从相交点到光源的向量，

N----平面法向量

V----视线方向

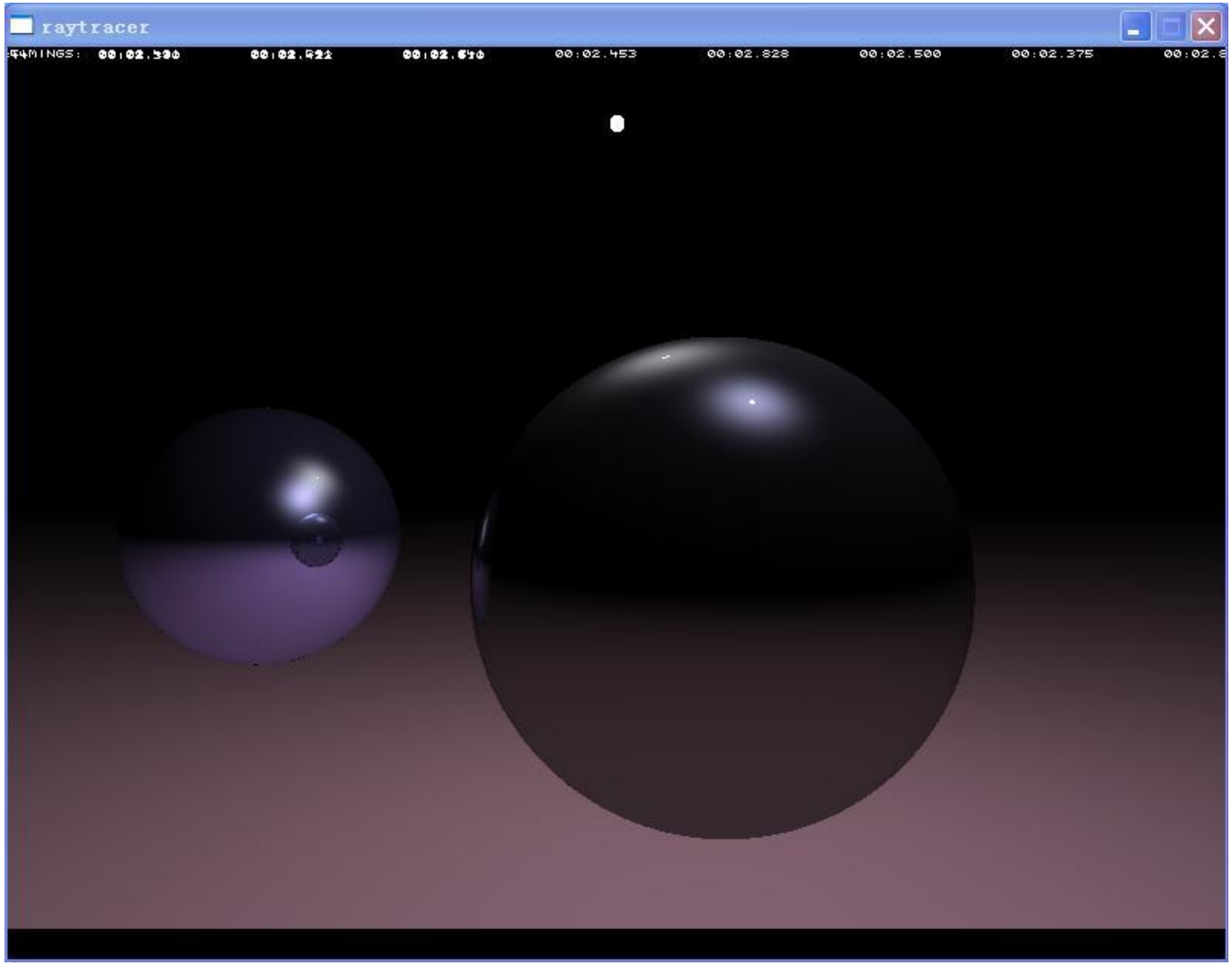
R----L 在表面上的反射向量

注意这个公式包含了散射和镜面反射光。

```
vector3 V = a_Ray.GetDirection();//光线方向
vector3 R = L -2.0f * DOT( L, N ) * N;

float dot = DOT( V, R );
if (dot > 0)
{
    float spec = powf( dot, 20 ) * prim->GetMaterial()->GetSpecular() * shade;
    // add specular component to ray color
    a_Acc += spec * light->GetMaterial()->GetColor();
}
```

增加了 Phong 光照模型的计算后，产生的结果如下图：

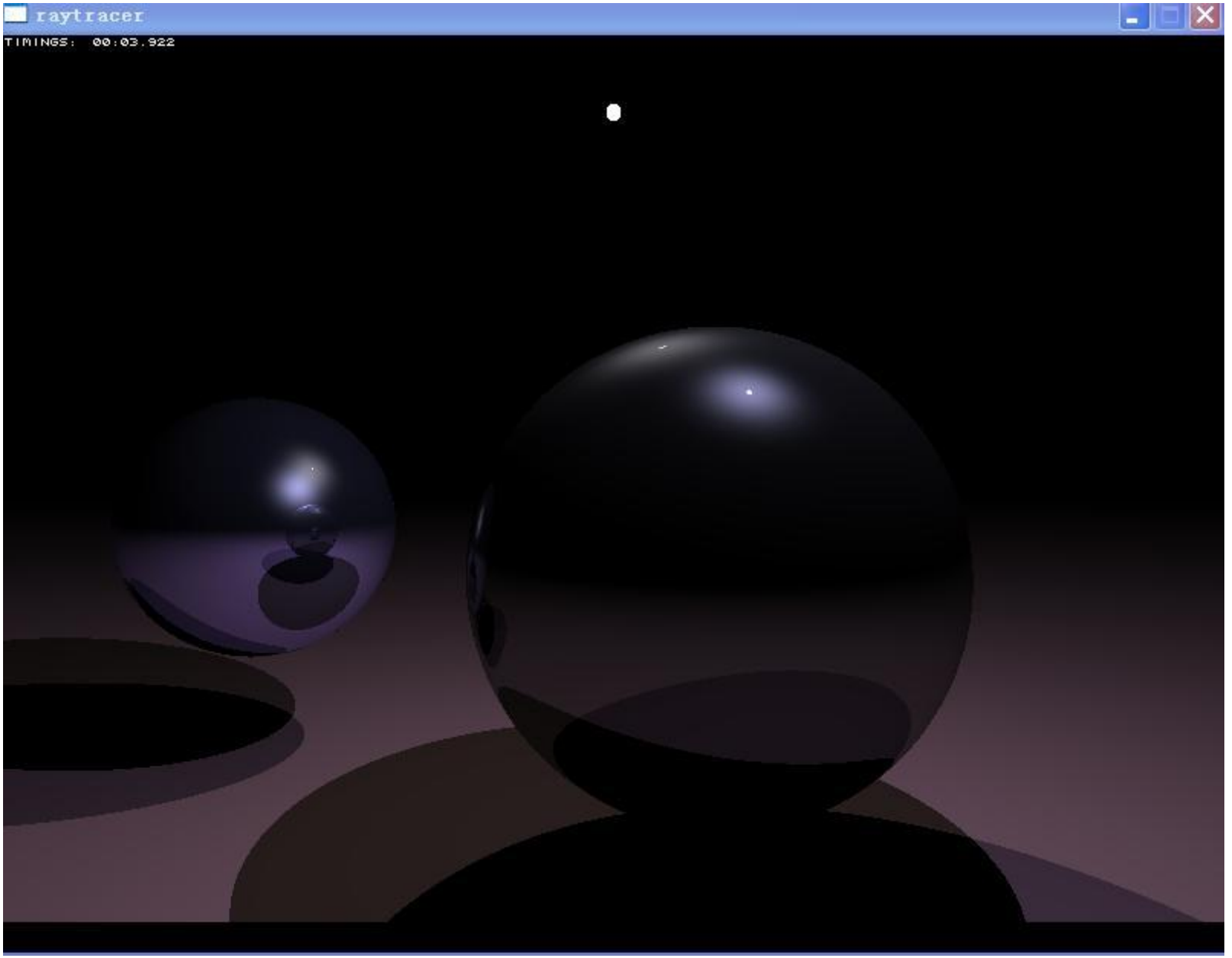


Phong 模型的效果

阴影

阴影线和其他的不同：它对于产生它的光线的颜色没有贡献；相反，它们经常用来判段一个光源是否可以“看见”一个相交点。

```
// handle point light source
 float shade = 1.0f;
 if (light->GetType() == Primitive::SPHERE)
{
    vector3 L = ((Sphere*)light)->GetCentre() - pi;
    float tdist = LENGTH( L );
    L *= (1.0f / tdist);
    Ray r = Ray( pi + L * EPSILON, L );
    for ( int s = 0; s < m_Scene->GetNrPrimitives(); s++ )
    {
       Primitive* pr = m_Scene->GetPrimitive( s );
       if ((pr != light) && (pr->Intersect( r, tdist )))
       {
          shade = 0;
          break;
       }
    }
}
```


raytracer
TIMINGS: 00:03.922

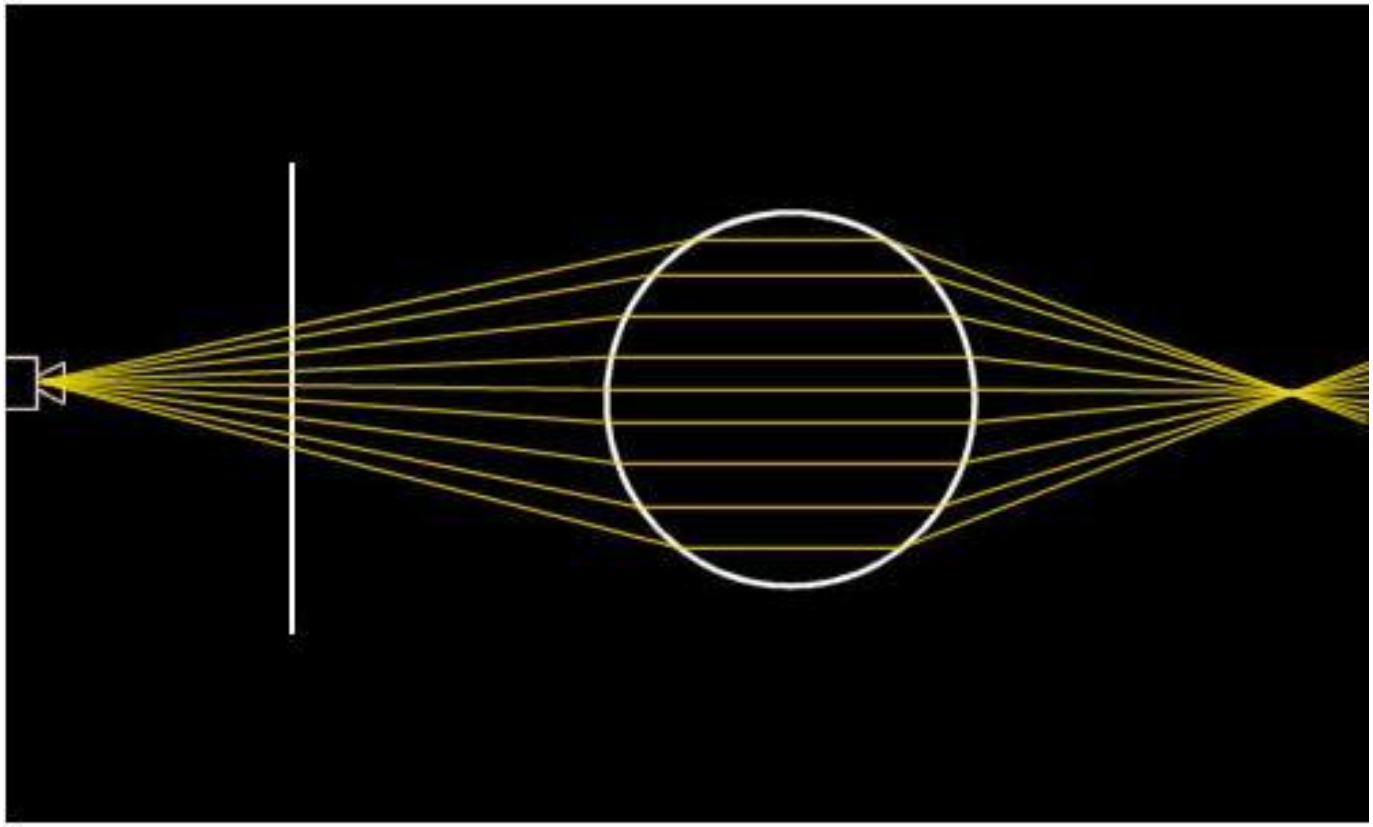
```
Primitive* Engine::Raytrace( Ray& a_Ray, Color& a_Acc, int a_Depth, float a_RIndex, float& a_Dist )
{
    if (a_Depth > TRACEDEPTH) return 0;
    // trace primary ray
    a_Dist = 1000000.0f;
    vector3 pi;
    Primitive* prim = 0;
    int result;
    // find the nearest intersection
    for ( int s = 0; s < m_Scene->GetNrPrimitives(); s++ )
    {
        Primitive* pr = m_Scene->GetPrimitive( s );
        int res;
        if (res = pr->Intersect( a_Ray, a_Dist ))
        {
            prim = pr;
            result = res; // 0 = miss, 1 = hit, -1 = hit from inside primitive
        }
    }
    // no hit, terminate ray
    if (!prim) return 0;
    // handle intersection
    if (prim->IsLight())
    {
        // we hit a light, stop tracing
        a_Acc = Color( 1, 1, 1 );
    }
```

```
    else
    {
        // determine color at point of intersection
        pi = a_Ray.GetOrigin() + a_Ray.GetDirection() * a_Dist;
        // trace lights
        for ( int l = 0; l < m_Scene->GetNrPrimitives(); l++ )
        {
            Primitive* p = m_Scene->GetPrimitive( l );
            if (p->IsLight())
            {
                Primitive* light = p;
                // handle point light source
                float shade = 1.0f;
                if (light->GetType() == Primitive::SPHERE)
                {
                    vector3 L = ((Sphere*)light)->GetCentre() - pi;
                    float tdist = LENGTH( L );
                    L *= (1.0f / tdist);
                    Ray r = Ray( pi + L * EPSILON, L );
                    for ( int s = 0; s < m_Scene->GetNrPrimitives(); s++ )
                    {
                        Primitive* pr = m_Scene->GetPrimitive( s );
                        if ((pr != light) && (pr->Intersect( r, tdist )))
                        {
                            shade = 0;
                            break;
                        }
                    }
```

```
            }
            // calculate diffuse shading
            vector3 L = ((Sphere*)light)->GetCentre() - pi;
            NORMALIZE( L );
            vector3 N = prim->GetNormal( pi );
            if (prim->GetMaterial()->GetDiffuse() > 0)
            {
               float dot = DOT( L, N );
               if (dot > 0)
               {
                  float diff = dot * prim->GetMaterial()->GetDiffuse() * shade;
                  // add diffuse component to ray color
                  a_Acc += diff * light->GetMaterial()->GetColor() * prim->GetMaterial()->GetColor();
               }
            }
            // determine specular component
            if (prim->GetMaterial()->GetSpecular() > 0)
            {
               // point light source: sample once for specular highlight
               vector3 V = a_Ray.GetDirection();
               vector3 R = L - 2.0f * DOT( L, N ) * N;
               float dot = DOT( V, R );
               if (dot > 0)
               {
                  float spec = powf( dot, 20 ) * prim->GetMaterial()->GetSpecular() * shade;
                  // add specular component to ray color
                  a_Acc += spec * light->GetMaterial()->GetColor();
               }
```

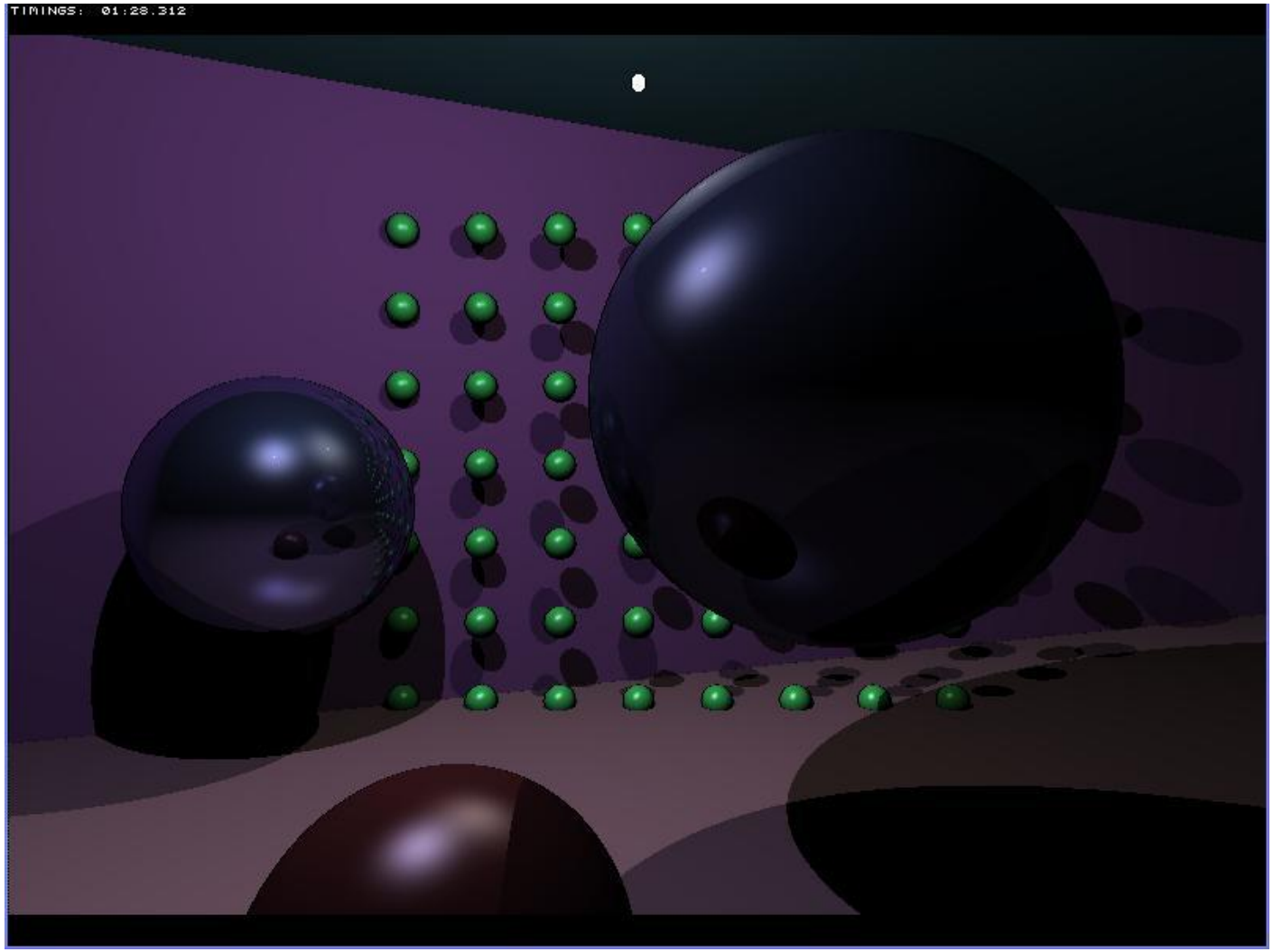
```
            }
        }
    }
    // calculate reflection
    float refl = prim->GetMaterial()->GetReflection();
    if (refl > 0.0f)
    {//几何体材质的反射度大于 0
        vector3 N = prim->GetNormal( pi );//几何体的法向量
        vector3 R = a_Ray.GetDirection() - 2.0f * DOT( a_Ray.GetDirection(), N ) * N;
        if (a_Depth < TRACEDEPTH)
        {//层数还没到上限
            Color rcol( 0, 0, 0 );
            float dist;
            Raytrace( Ray( pi + R * EPSILON, R ), rcol, a_Depth + 1, a_RIndex, dist );
            a_Acc += refl * rcol * prim->GetMaterial()->GetColor();//颜色值中加入反射光的贡献值
        }
    }
  }
  // return pointer to primitive hit by primary ray
  return prim;
}
```

折射



```
// calculate refraction
	float refr = prim->GetMaterial()->GetRefraction();
	if ((refr > 0) && (a_Depth < TRACEDEPTH))
	{
		float rindex = prim->GetMaterial()->GetRefrIndex();
		float n = a_RIndex / rindex;
		vector3 N = prim->GetNormal( pi ) * (float)result;
		float cosI = -DOT( N, a_Ray.GetDirection() );
		float cosT2 = 1.0f - n * n * (1.0f - cosI * cosI);
		if (cosT2 > 0.0f)
		{
			vector3 T = (n * a_Ray.GetDirection()) + (n * cosI - sqrtf( cosT2 )) * N;
			Color rcol( 0, 0, 0 );
			float dist;
			Raytrace( Ray( pi + T * EPSILON, T ), rcol, a_Depth + 1, rindex, dist );
			a_Acc += rcol * transparency;
		}
	}
```

加入折射的效果图：



Lambert-Beer 定律

现在让我们来想象有这么一个水池，里面充满了带颜色的物质（比如说水里面混合着蓝墨水）。池子的浅处大概 10cm 深，深处呢有 1 米深。如果你从上往底下看，很明显可以看到在较深的那端的底部受到颜色的影响会比浅处的要大。这种效应就叫 Beer 定律。

Beer 定律可以用下列公式表示：

```
light_out = light_in * e(e * c * d)
```

这个公式主要是用来计算溶解在水中的物质的光吸收度的。e 是一个常量，表明溶剂的吸收度（准确来说，是单位为 L/mol*cm 的摩尔吸光率）。c 是溶质的数量，单位 mol/L. d 是光线的路径长度。一般我们可以简化公式如下：

```
light_out = light_in * e(d * C)
```

d 是路径长度，C 是一个常量，表示物质的密度

```
// apply Beer's law
Color absorbance = prim->GetMaterial()->GetColor() * 0.15f * -dist;
Color transparency = Color( expf( absorbance.r ), expf( absorbance.g ), expf( absorbance.b ) );
a_Acc += rcol * transparency;
```

效果图