A2: Transformations & Additional Primitives

一、概述

1.目 标：

（1）新增基本模型 Plane，Triangle，并对它们应用仿射变换。
（2）创建一个 Perspective Camera
（3）使用新的渲染模式：材质中增加 diffuse 属性和法线可视化。

2.背景知识介绍：

（1）求射线和 Plane 的交点：

将射线的参数方程代入平面的点法式方程，求 t 即可。

$P(t) = R_o + t * R_d$　　$H(P) = n \cdot P + D = 0$

$n \cdot (R_o + t * R_d) + D = 0$　　$t = -(D + n \cdot R_o) / n \cdot R_d$

（2）求射线和 Triangle 的交点：

- *Use ray-polygon*

  此方法同样适用于其他多边形。先判断射线是否和三角形所在平面相交，若相交，再判断交点是否在三角形内。可以通过计算该点和三角形各边的夹角之和是否为内角和 180°。或者从该点发射射线，计算和三角形边的 Signed count，如果这个 count 总值为 0 则在三角形外部。

- *Use barycentric coordinates*

  使用重心坐标系。已知三角形 a,b,c 的顶点坐标和 P(Xp,Yp,Zp)点坐标，

  P(Xp, Yp, Zp) = Alpha*a + Beta*b + Gamma*c，且 Alpha + Beta + Gamma =1，（以下简写 Alpha 为 A，Beta 为 B，Gamma 为 G）如果 A,B,G 都满足大于 0 的条件，那么 P 点在三角形内部。

  设 A=1-B-G，则 P(t) = P(B, G) ，即：Ro + t * Rd = a + B(b-a) + G(c-a)。通过三阶行列式求出三元一次方程组的解，可得到 B,G,t 的值。判断 B,G，B+G 都大于 0，且 t 大于 tmin，则射线和三角形有交点。

（3）创建一个 Perspective Camera：origin 是定点，而 direction 根据输入的 image 像素的坐标在 virtual space 中变化。而 virtual space 的 size 的大小和 0.5*angle 的正切值成正比，因此可以算出 direction。

（4）diffuse 属性： $c_{pixel} = c_{ambient} * c_{object} + SUM_i [\ clamped(\mathbf{L_i} \cdot \mathbf{N}) * c_{lighti} * c_{object}\ ]$

（5）仿射变换的应用：

对所有物体进行变换的运算是复杂和高成本的，而直接将摄像机向相反方向变换也可以得到同样的画面效果。所以获得 objects 的 transform 之后，可以对摄像机的 origin 和 dir 进行逆向变换（左乘 transform 矩阵的逆矩阵）。此外，还要对物体的法线进行变换，可以通过变换切线的方式来间接变换，最后得到的公式是：$n_{WS} = (\mathbf{M}^{-1})^{\mathbf{T}}\, n_{OS}$

二、实现细节

1. 摄像机：

（1）PerspectiveCamera 的 GenerateRay 函数：

```
Ray PerspectiveCamera::generateRay (Vec2f point){
    Vec3f origin = center;
    //映射到虚拟image上，然后确定dir
    float  k_size = tan (angle_radians / 2.0f);  if (k_size < 0)k_size = 0;
```

```
    Vec3f end = center + (-0.5 + point.x ())*horizontal*k_size
        + (-0.5 + point.y ())*up*k_size+direction*0.5f   ;
    Vec3f dir = end - center;  dir.Normalize ();
    Ray ray (origin, dir);
    return  ray;
}
```

（2）diffuse 和背面渲染：

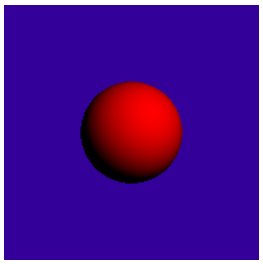
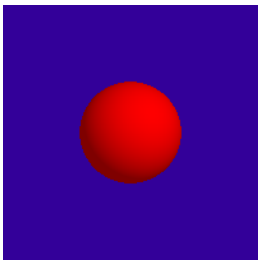
```
if (scene->getGroup ()->intersect (ray, hit, scene->getCamera ()->getTMin ())){
    Vec3f pixelColor(0.0,0.0,0.0), shadeColor(0.0,0.0,0.0);
    for (int i = 0; i < numLights; i++) {//遍历lights数组
    Vec3f lightColor, lightDir;
    scene->getLight (i)->getIllumination (ray.pointAtParameter (hit.getT ()),
lightDir, lightColor);
    Vec3f normal = hit.getNormal ();
    float wrong = ray.getDirection ().Dot3 (normal);
      if (wrong > 0){
       if (shadeBack) {//渲染背面，就反向法线
          normal *= -1.0f;   hasAmbient = true;}
       else {hasAmbient = false;}
      } else   hasAmbient = true;
    //diffuse
    float d = normal.Dot3 (lightDir);   if (d < 0)d = 0;
    shadeColor = lightColor * hit.getMaterial ()->getDiffuseColor ()*d;
    pixelColor += shadeColor;
    }
    if(hasAmbient) pixelColor += (ambient*hit.getMaterial()->getDiffuseColor());
                image->SetPixel (x, y, pixelColor);
}
```

（3）仿射变换的应用：

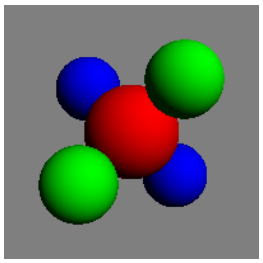
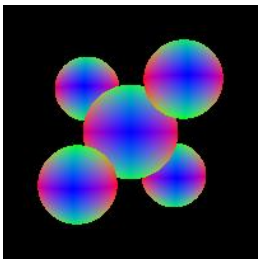
```
bool Transform::intersect (const Ray &r, Hit &h, float tmin) {
    //对发出的射线进行逆向变换
    Vec4f o4 (r.getOrigin (),1.0f);          Vec4f d4 (r.getDirection (),0.0f);
    Vec4f o4_trans= matrix.inverse()*o4;     Vec4f d4_trans = matrix.inverse()*d4;
    Vec3f o3_trans = o4_trans.xyz();         Vec3f d3_trans = d4_trans.xyz();
    Ray newRay (o3_trans, d3_trans);
    bool intersect = false;
    if (obj->intersect (newRay, h, tmin)) { //   对法线进行变换
        Vec4f normalTrans4 = Vec4f (h.getNormal (), 0.0f);
        Vec4f normal4 = (this->matrix.inverse ().transposed ()*normalTrans4);
        Vec3f normal = normal4.xyz ().Normalized();
        h.set (h.getT (), h.getMaterial (), normal,r);   intersect = true;
    }
    return intersect;}
```

三、结果展示

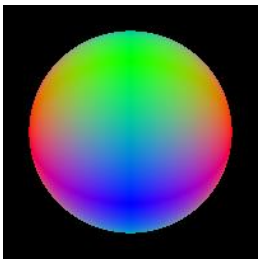
1.diffuse 和 ambient 测试：

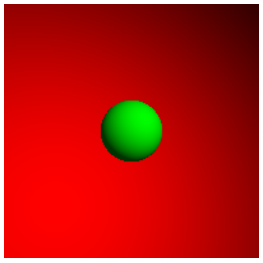
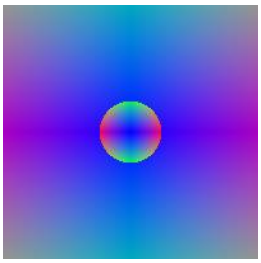
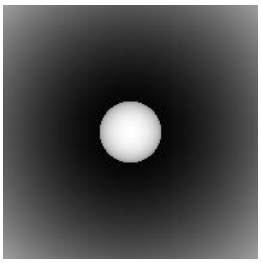
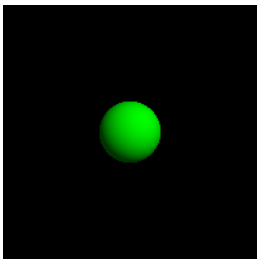
2.Colored lights 和法线可视化测试：

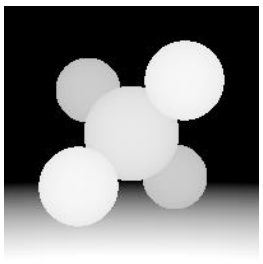
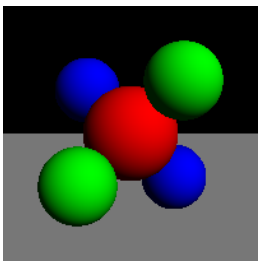
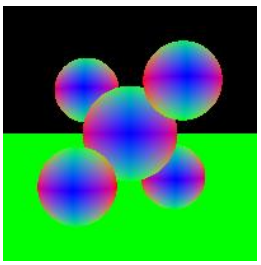
3.Perspective Camera 测试：

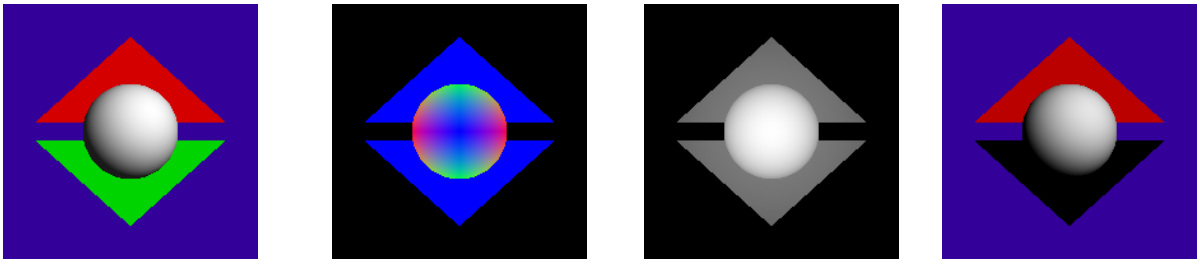
4.shadeback 和 noback 测试：

5.Plane 测试：

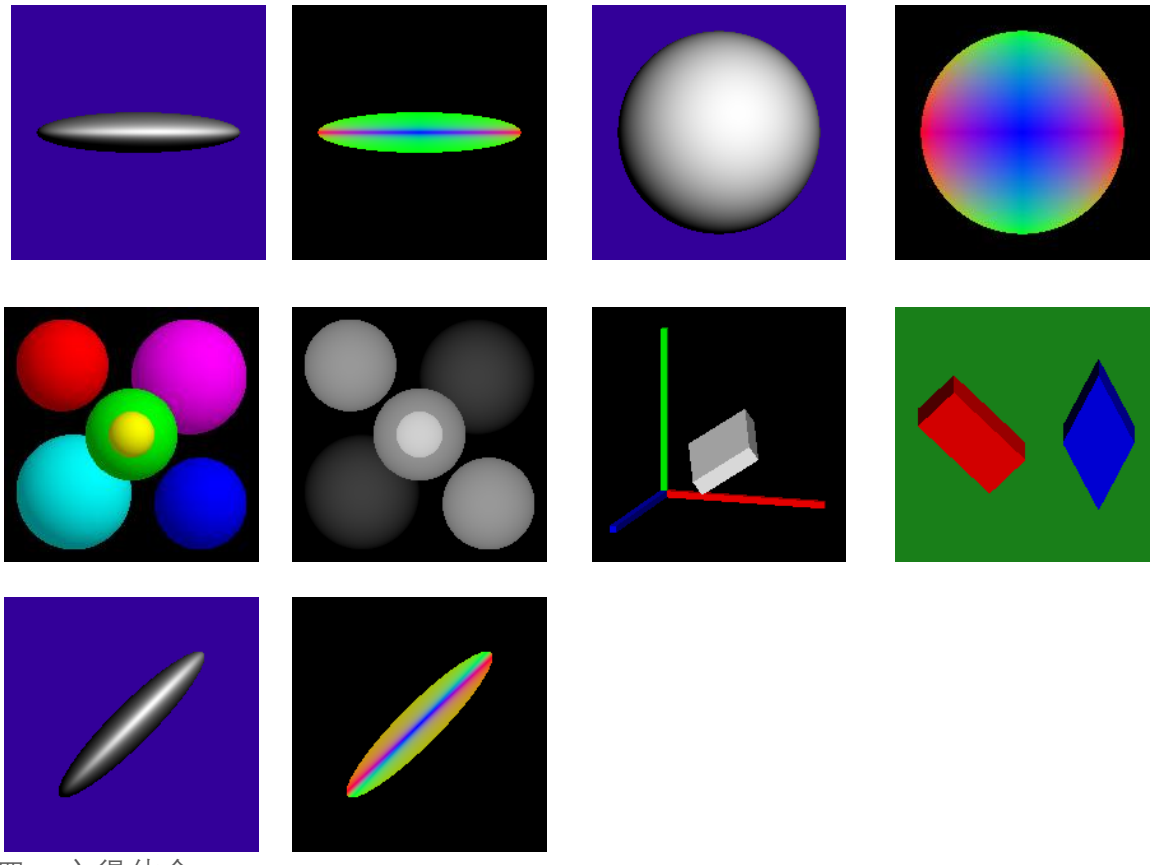




6.三角形测试:



7.obj 测试:

8.Transform 测试:



四、心得体会

测试样例很多的时候想要把每一张图画对是比较困难的，每个不一致之处面对的问题可能都不一样。这次通过对光线投射的实现，理解了两种不同的摄像机和图元的交互方式，基本捋清楚从三维物体变成一张 image 上的像素的过程。在学习的过程中发现图形学中有很多巧思，也真是一直在印证那句话：在计算机图形学中，看起来是对的，就是对的。