Project 2 实验报告

詹佳豪 22307140116

1. 光线投射

(1) GetIllumination()

对于点光源与面光源（即光线平行射入）的不同光源，该函数需要计算得出指向光源的方向矢量、照明强度、场景点到达光源之间的距离。我们主要实现点光源的情况，距离与方向矢量直接利用私有变量_position 与传入的 p（指定点的坐标），进行向量运算即可得到。而光强则由公式：

$$L(x_{\text{surf}}) = \frac{I}{\alpha d^2}$$

确定。该公式表明，距离光源越远的点，其光强越小，而其程度由衰减因子来进行控制。

(2) Phong 光照模型

Phong Lighting 是一种光照模型，其将光分为三种：环境光、慢反射光、镜面光照。三者叠加形成最后的光照图。

环境光在 scene 中有定义，渲染时直接调用即可。

慢反射光与镜面光照主要在 shade 函数中实现。Shade 函数的作用在于根据相机视角、照射点、光源信息来输出对应的慢反射光与镜面光照。慢反射光的计算如下公式所示，慢反射光是会射向四面八方，所以不需要考虑 camera 的角度。我们只需要计算接触面法向与光源方向的内积，即两个方向的相似度，将其作为 scale 调整光强即可，最后再根据具体材料调整反射率即可。

对于镜面光照，我们需要计算理想 camera 射线的反射方向与光源方向之间的相似度。仍然是将其作为 scale 去调整光强。公式如下，可以发现，clamp 上方有个系数次方，这是由于：在方向达到 45 度时，其实 scale 仍然很大（0.7），这与我们先是中的感觉肯定是不吻合的，镜面的反射只有在相差几度时会比较明显，所以为了实现这种真实物理世界的感觉，通过 s 次方强行在大角度时压低 scale，这是结合实际情况的重要应用。

$$\text{clamp}(\mathbf{L},\mathbf{N}) = \begin{cases} \mathbf{L}\cdot\mathbf{N} & \text{if } \mathbf{L}\cdot\mathbf{N} > 0 \\ 0 & \text{otherwise} \end{cases} \qquad I_{\text{diffuse}} = \text{clamp}(\mathbf{L}\cdot\mathbf{N}) * L * k_{\text{diffuse}}$$

$$I_{\text{specular}} = \text{clamp}(\mathbf{L},\mathbf{R})^s * L * k_{specular}$$

Shade 函数只处理一个光源的情况，但我们很容易想到，实际的图形学问题中可能存在多个光源的情况，只需要利用循环反复调用 shade，最后累加漫反射与镜面反射即可，最后再加上环境光，否则环境光会被重复计算。

2. 光线投射

我们前面对第一个图片的渲染调用了 intersect 函数，这个函数对于球体代码已经实现了，所以我们可以渲染图片一，对于后面不同的形状，都没有实现 intersect，所以需要我们手动实现。

具体使用面向对象的技术，首先 Object3D 是一个大的父类，包含了所有物体，换句话说，不同的物体类型如三角形、球体都是其的特殊子类。基于这个父类，我们需要在不同物体中分别实现他的 intersect 函数。Intersect 函数接受光线、返回 t 值以及接触面的信息（包括法向）。

Sphere 类中的 intersect 函数已经实现完成。

Plane 类的 intersect 函数：我们已知平面在三维空间中可以表示为 Pn=dist，而光线也可以写为方程 p=o+t*d 的形式，所以简单来说，联立这两个方程，即可求出 t 值，而接触面的信息直接用平面信息更新即可。

Triangle 类的 intersect 函数：类似于平面类的实现，需要注意的是用重心法判断交点是否在三角形中间。这里直接采用线性方程组的求解方法，矩阵求逆来求解就行。在更新 h 法向量时需要注意，为了三角形组成的物体可以更加平滑，我们在法向量上要进行插值处理，具体方法是根据前面求出的顶点加权系数来加权三个顶点上的法向量，最后得到较为平滑的方向：

```
n=_normals[0]*param.x()+_normals[1]*param.y()+_normals[2]*param.z();
```

比较复杂的是 transform 类的实现，这个类并非一个具体物体形状，而是针对那些经过仿射变换后的物体，如果我们把所有物体都变换回全局坐标系，这样计算量过大，比较实用的方法是将光线变换到局部坐标系上，在局部坐标系中求出对应的信息，再变换回全局坐标系。

这里需要额外注意，对于球体的变换不能完全遵循“先变换到局部坐标系，求出特征变换到全局坐标系”这个流程，因为其法向量要特殊处理。所以我在代码中的做法是，先检测球体，利用 dynamic_cast 如果是球体对象就会返回对应指针，不是的话就是空指针，由此控制球体的特殊处理。之后照常将光线变换到局部坐标系，求出特征，尤其是法向量。之后，我们需要对变换矩阵 M_inv 进行矩阵分解，M=RST，R 为旋转矩阵、S 为伸缩矩阵、T 为平移矩阵，这样我们就可以对 S 进行特殊处理，对于球体，变换回全局坐标系，S 不需要求逆，而 TR 求逆即可，所以得到下面的代码。

```
h.set(h.getT(),h.getMaterial(),VecUtils::transformDirection(T.inverse(),VecUtil
s::transformDirection(S,VecUtils::transformDirection(R.inverse(),n))));
```

这里我还想就矩阵分解的问题展开，对于 T 的求取很同意，直接去除最右边一列即可。而求出了 R 其实求逆再乘上 M 就会得到 S，所以我们的目标就是求 R。这里可以利用下面递推公式（极分解），不断递推，知道变换很小的时候停止，就得到 R。这个公式其实很自然，如果 M 本身就是旋转矩阵，可以发现其根本是不会变换的。而具体的证明 Shoemake 和 Duff（1992）已经讨论过了。

$$\mathbf{M}_{i+1} = \frac{1}{2}\left(\mathbf{M}_i + (\mathbf{M}_i^T)^{-1}\right)$$

总的代码如下：

```
bool Transform::intersect(const Ray &r, float tmin, Hit &h) const
```

```
{
    // TODO implement
    //std::cout<<"The old ray is :"<<r.getDirection().x()<<"
"<<r.getDirection().y()<<" "<<r.getDirection().z()<<std::endl;

    Matrix4f M_inv= M.inverse();
    Vector3f o=r.getOrigin();
    Vector3f d=r.getDirection();
    Vector3f orig=VecUtils::transformPoint(M_inv,o);
    Vector3f dir=VecUtils::transformDirection(M_inv,d);
    Ray new_r(orig,dir);
    //std::cout<<"The new ray is :"<<new_r.getDirection().x()<<"
"<<new_r.getDirection().y()<<" "<<new_r.getDirection().z()<<std::endl;
    //std::cout<<_object->getType()<<std::endl;
    bool result =_object->intersect(new_r,tmin,h);
    //we should conduct matrix decomposition
    //std::cout<<"The result is :"<<result<<std::endl;
    Matrix4f T,S,R;
```

```
    if(result){
        Vector3f n=h.getNormal().normalized();
        decomposeMatrix(M_inv,T,S,R);
        Vector3f n_global = VecUtils::transformDirection(M,n);
        //std::cout<<"the type:"<<this->_object->type<<std::endl;
        //R.print();
        if(dynamic_cast<Sphere*>(_object)){
            h.set(h.getT(),h.getMaterial(),VecUtils::transformDirection(T.invers
e(),VecUtils::transformDirection(S,VecUtils::transformDirection(R.inverse(),n)))
);
            //std::cout<<"The normal is :"<<h.getNormal().x()<<"
"<<h.getNormal().y()<<" "<<h.getNormal().z()<<std::endl;
        }else
        h.set(h.getT(),h.getMaterial(),n_global.normalized());
        //std::cout<<"The normal is :"<<h.getNormal().x()<<"
"<<h.getNormal().y()<<" "<<h.getNormal().z()<<std::endl;
    }

    return result;
}
```

3. 光线追踪与阴影投射

前面我们利用 Phong 模型确实可以反映光照情况，但是现实场景中，一个物体受到的光照不止来源与光源，光源打在物体上可能会映射到别的物体上，举个简单的例子，一个白色小球放置在红色背景幕布下，会显示为红色小球，但光源其实是白的。为了实现这个效果，我们就要进行光线追踪。

具体的做法也很容易，我们多给到一个 bounces 的参数，不断减 1，来控制反射次数（由于计算量的限制，我们肯定做不到无数次的反射）。在 Renderer::traceRay 中，每次调用都在最后判断 bounces 是否到达 0，如果没有到达，就递归调用自己。此时对于递归调用的函数，ray 由反射光线组成，而 bounces 减 1，由此得到反射的光照信息，进行累加，最后再添加环境光即可，这里注意环境光只需要加一次即可，如果每次递归都加会导致小兔子底部很亮。这里还需要注意，对于最后一次递归 bounces 为 0 的时候以及本身就压根没交到物体的情况，都返回环境颜色。

阴影的实现可以大大提升成像的真实感。只需要从交点，向光源引出射线作为 ray，再考察是否与其他物体 intersect 就可以判断该点是否被遮挡。

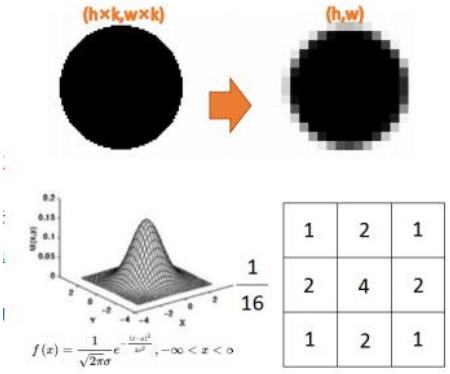
```
if(_scene.getGroup()->intersect(shadowRay,tmin,h_)){
                    shadow=true;
                }
```

4. 抗锯齿问题

我们通过扩大采样频率、抖动采样、最后高斯滤波的方式解决锯齿问题。首先，我们相较于原本的显示采样频率，多采样三倍，这样可以反映出更多细节，当然也需要更多的计算开销，在实时性上有欠缺，在真实上有提升。这也就是颜老师上课说的实时、真实、交互的相互 tradeoff。

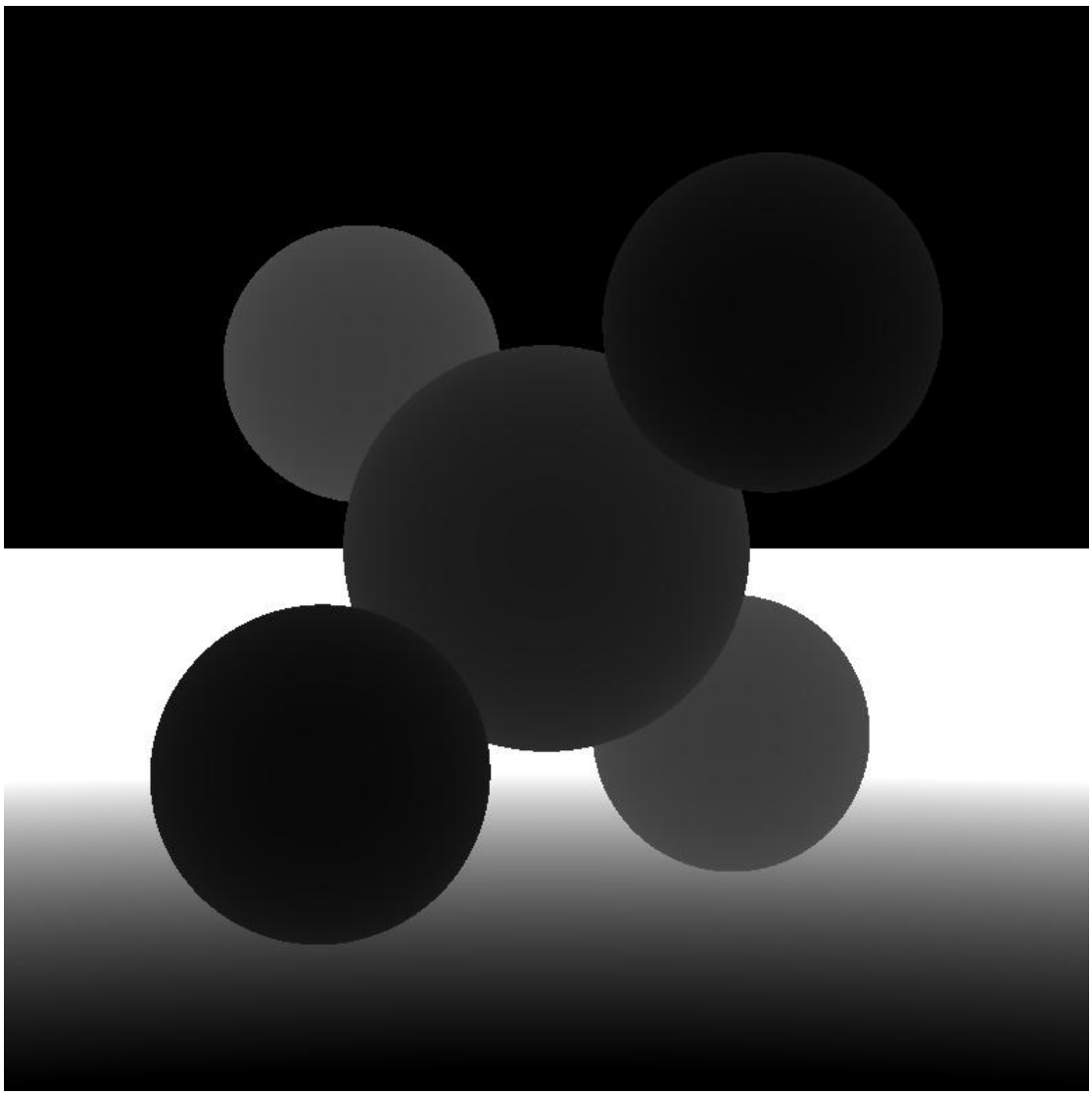
对于已经扩大的每个像素点，都抖动采样 16 次，也就是说选取一个随机数，然后加在像素点上，使得其偏离一点中心位置，这样可以采样到对应像素周围的全貌，而不只是一个点。

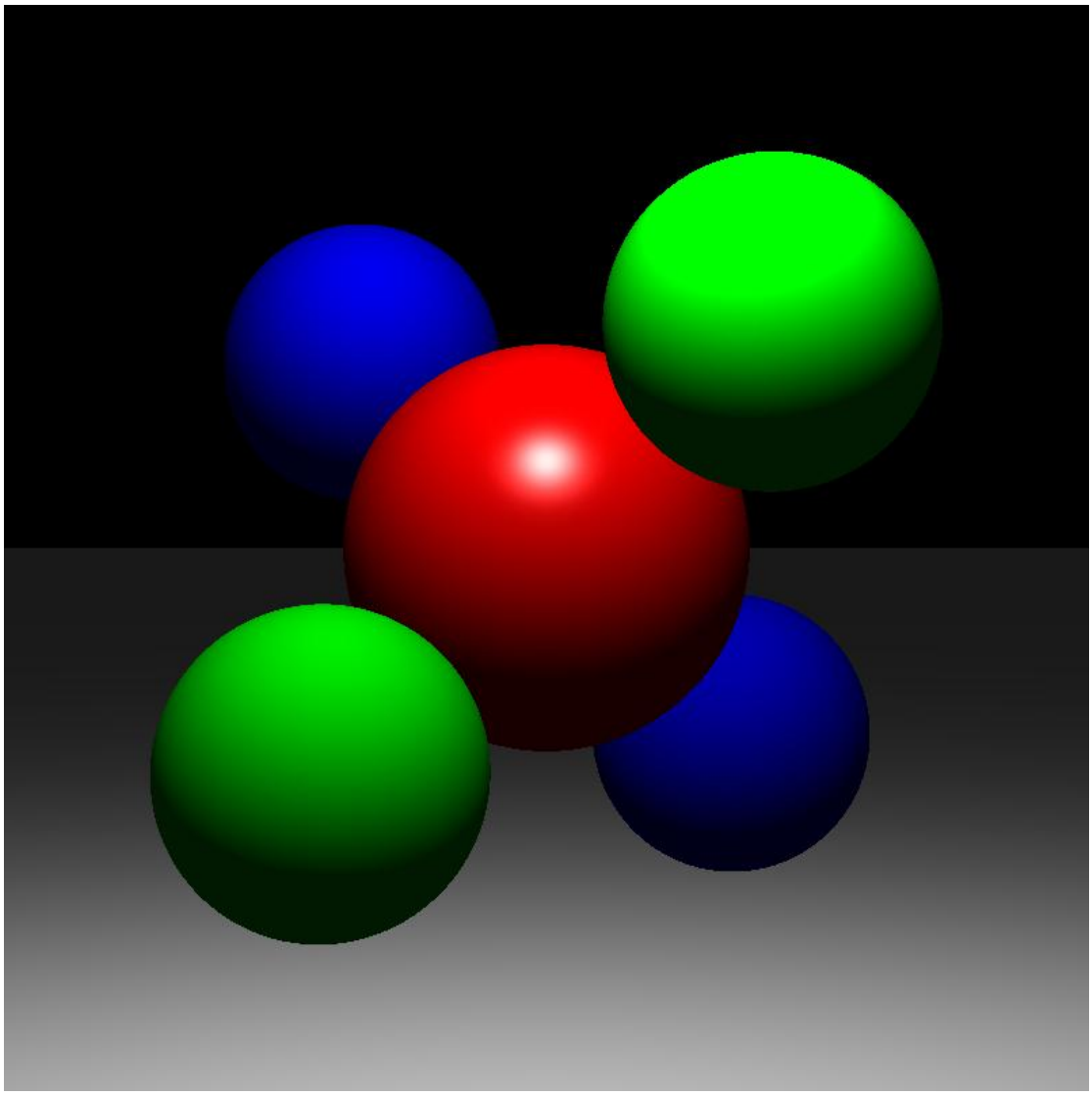
最后做 downsampling，这里采用高斯滤波，类似卷积的模式，按照二维正态分布加权像素信息，最后得到原本像素上一个点的显示信息，这样该点综合了局部的所有信息，会更加真实。

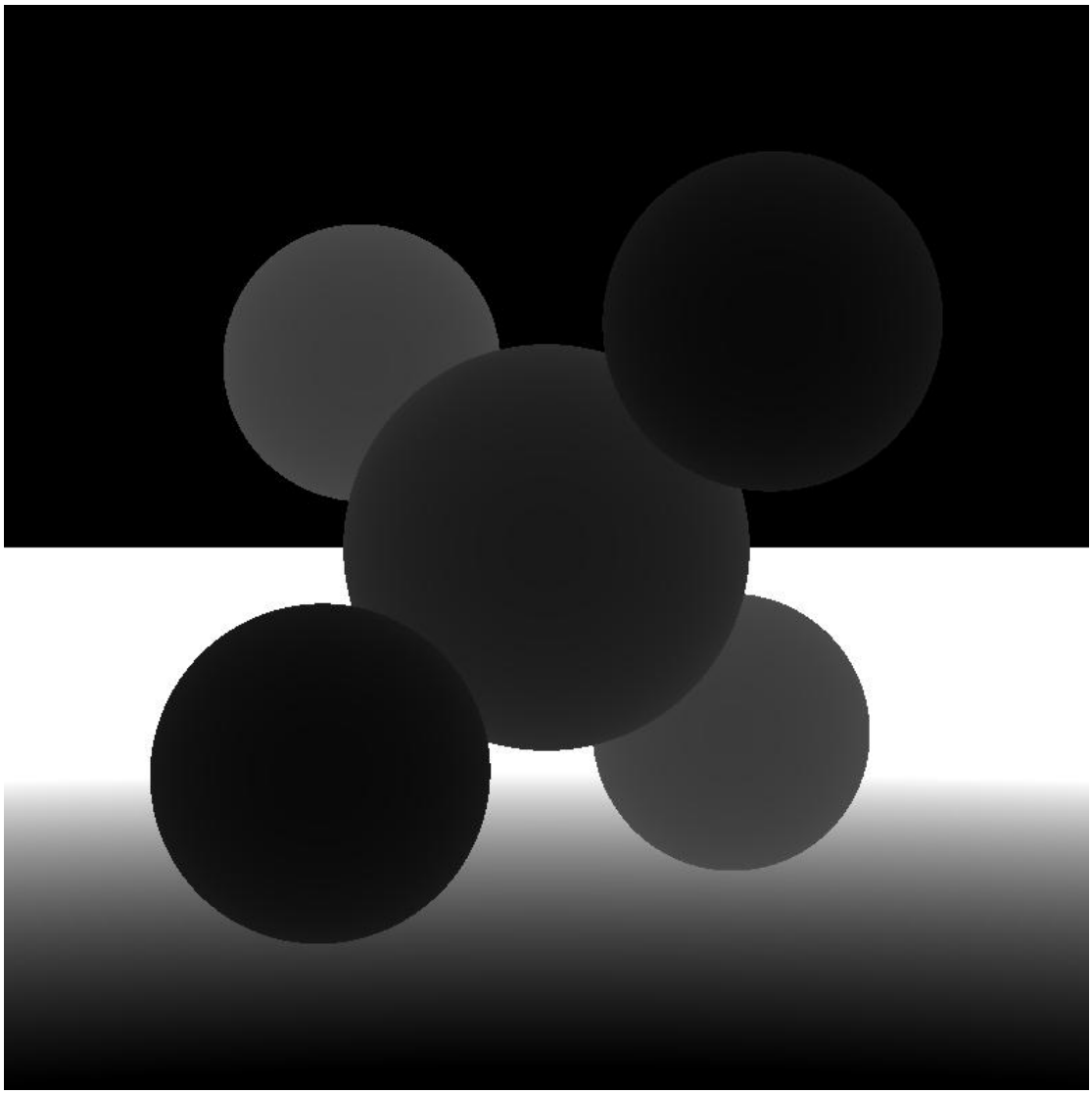


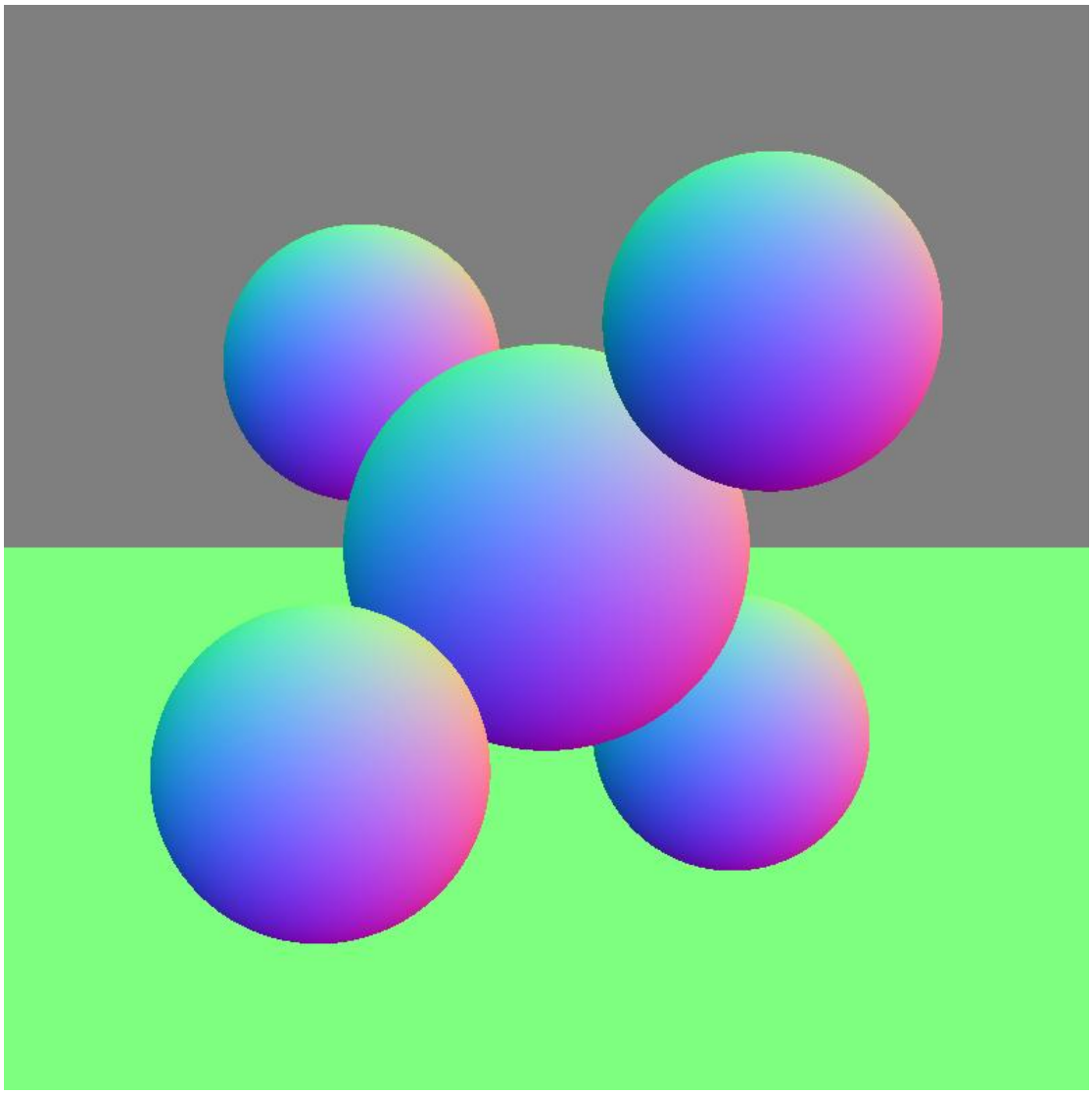
5. 实验结果

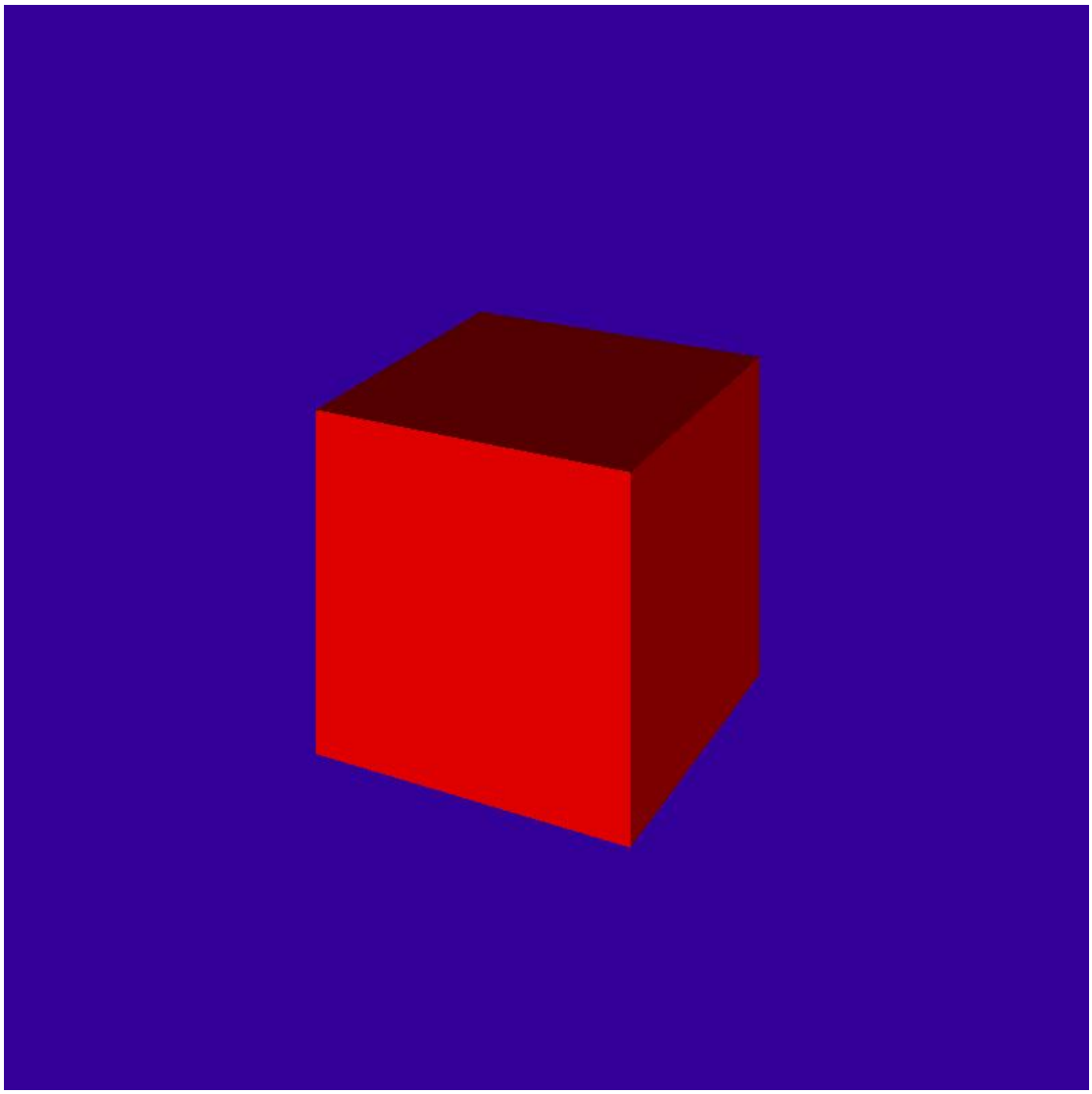
6.

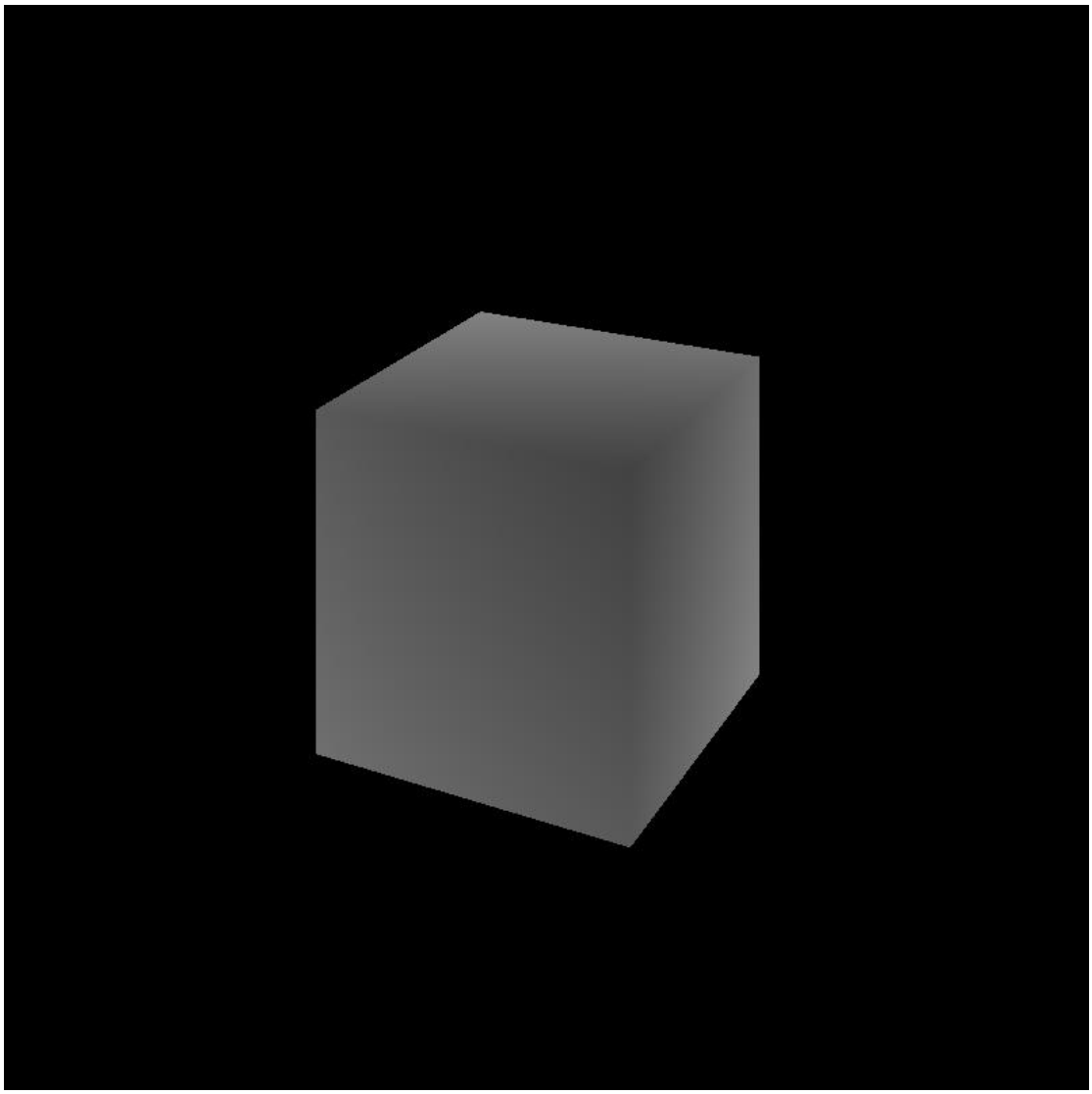


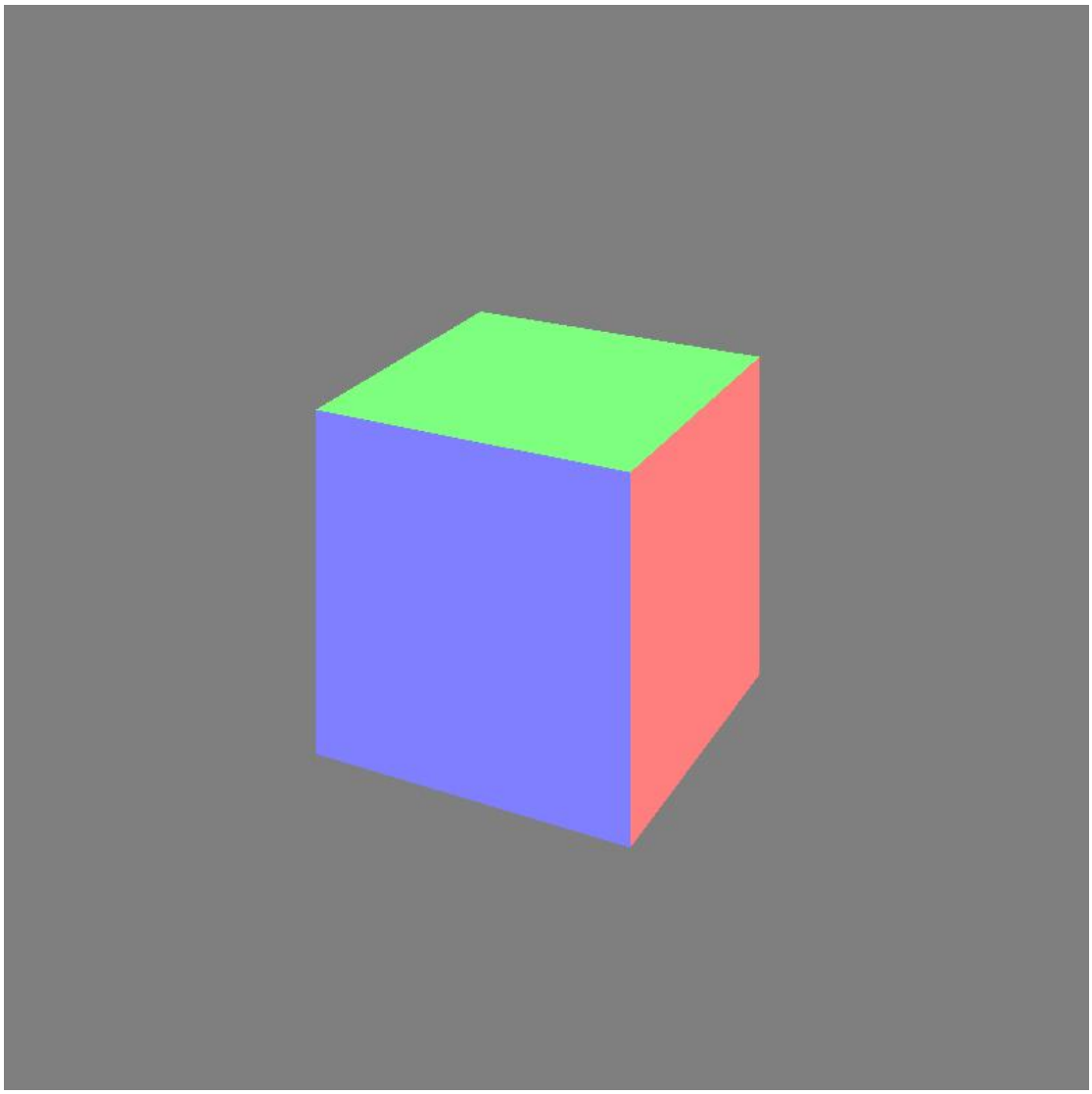


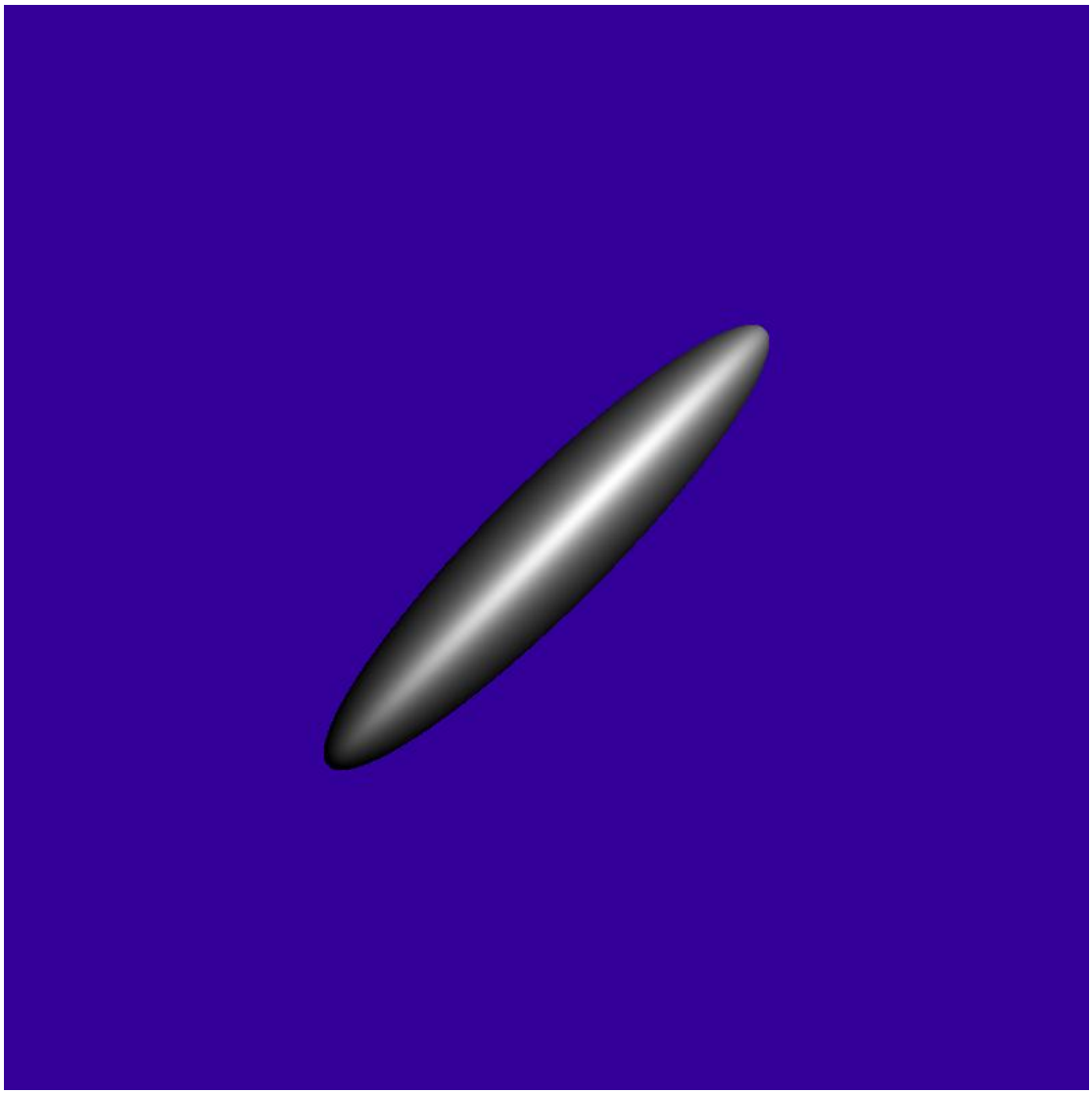


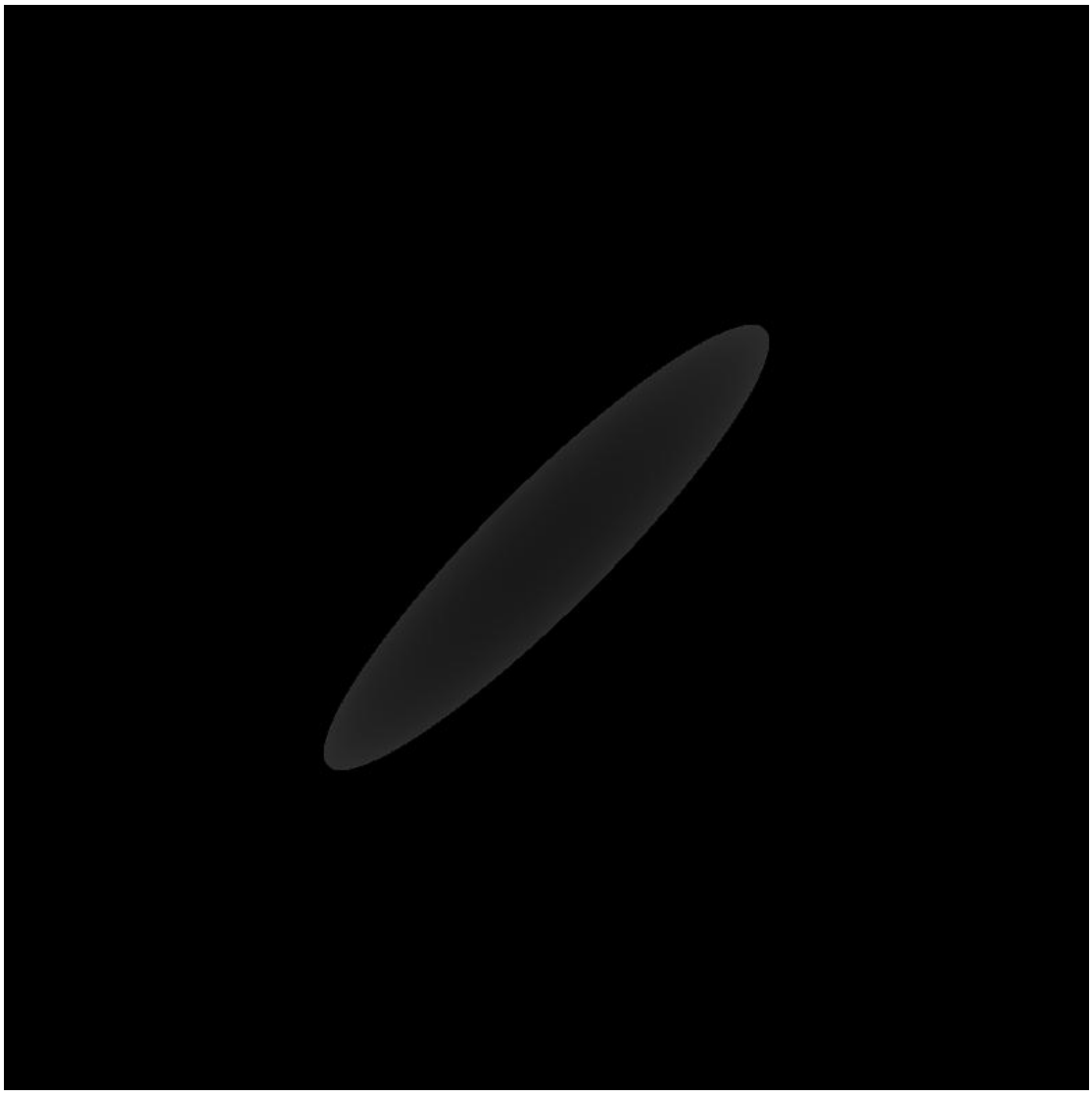


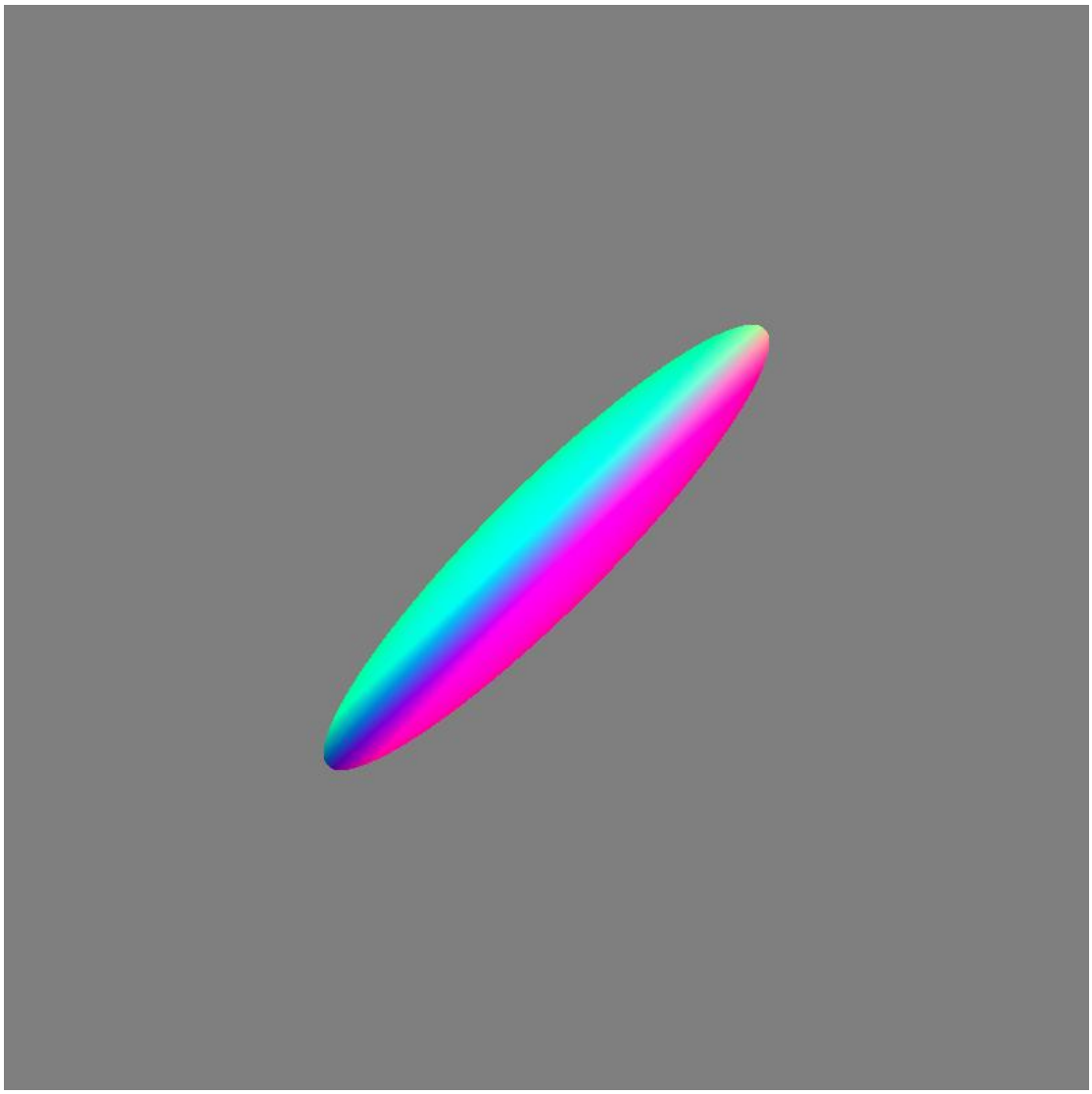


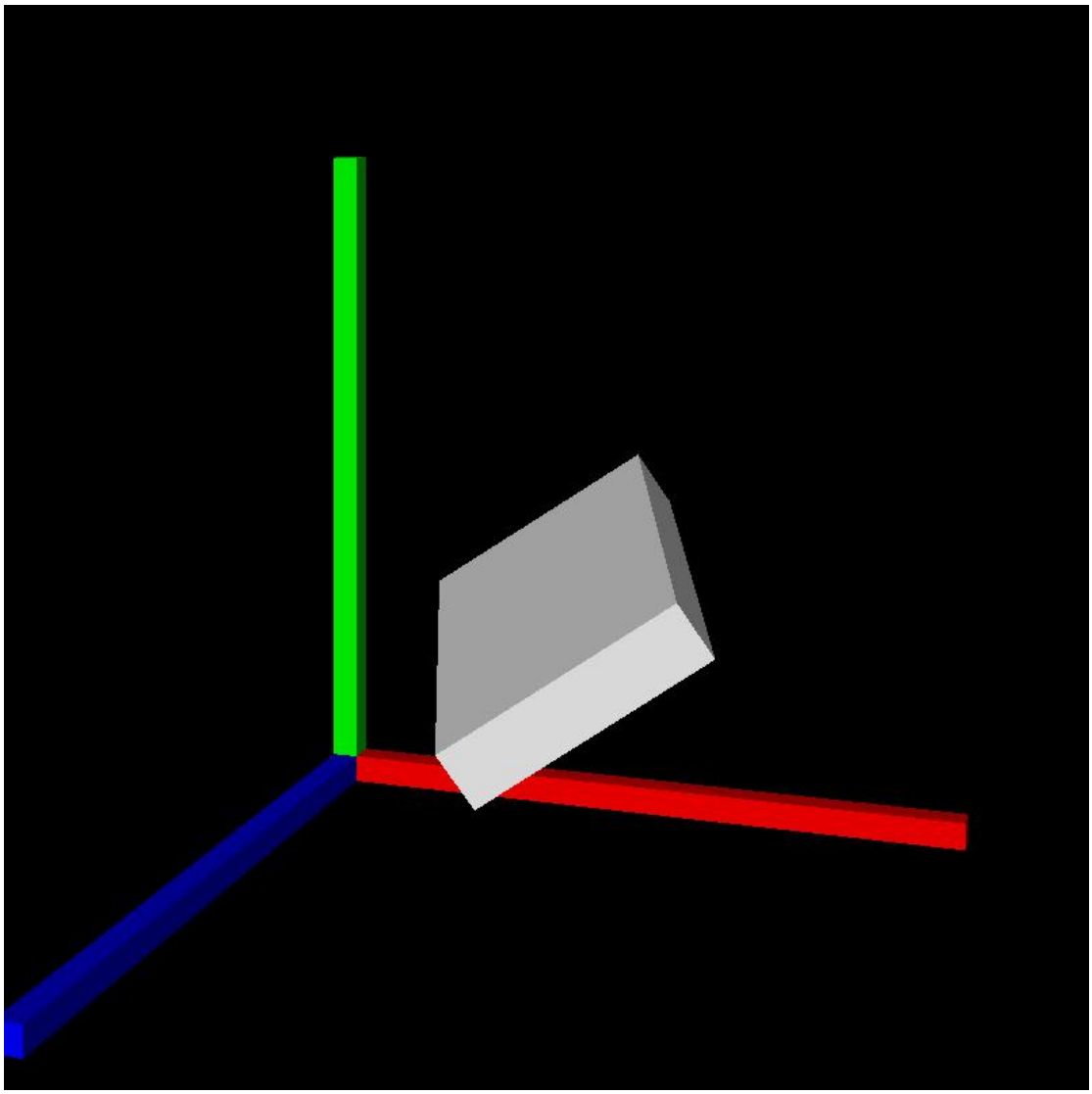


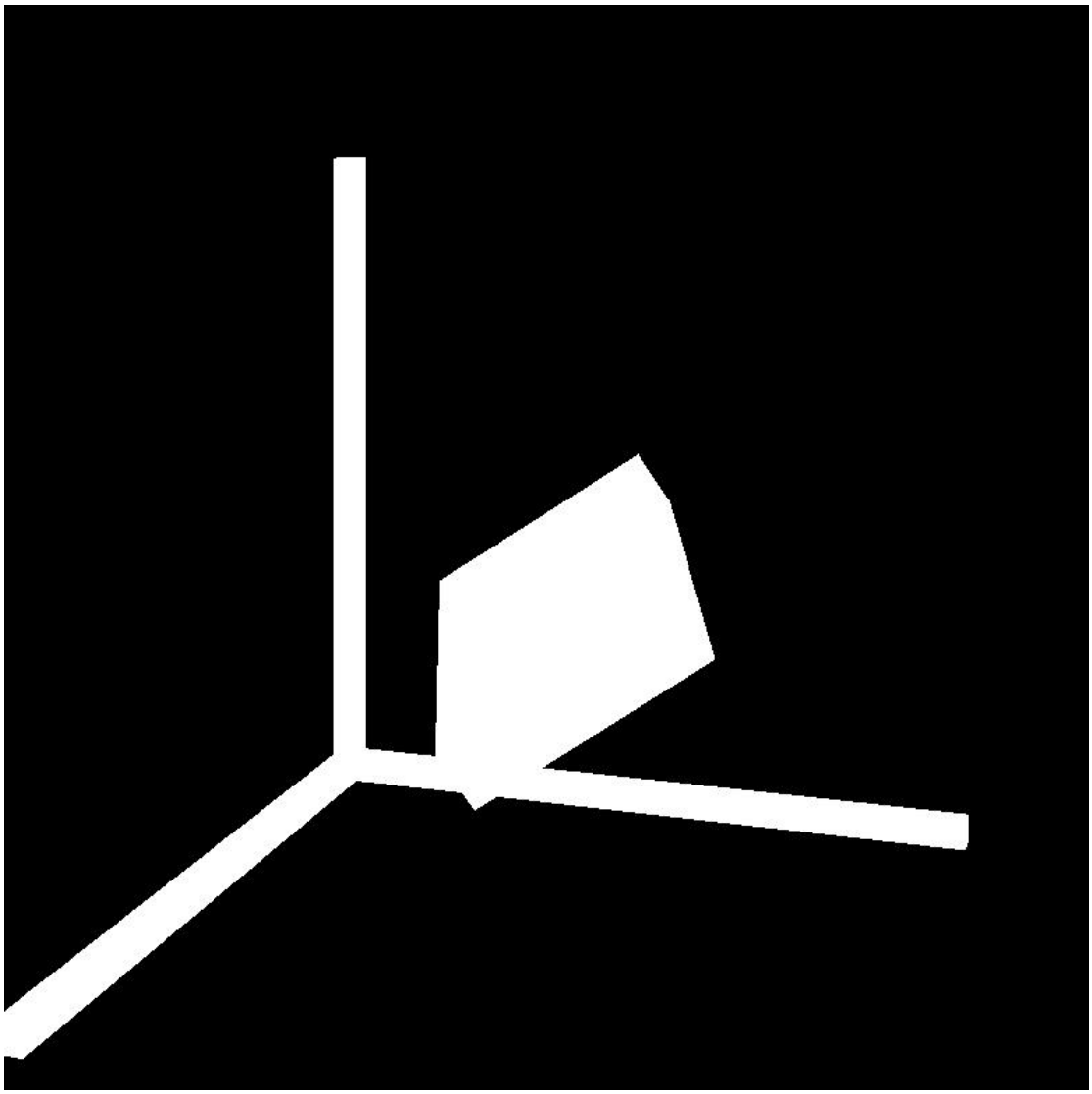


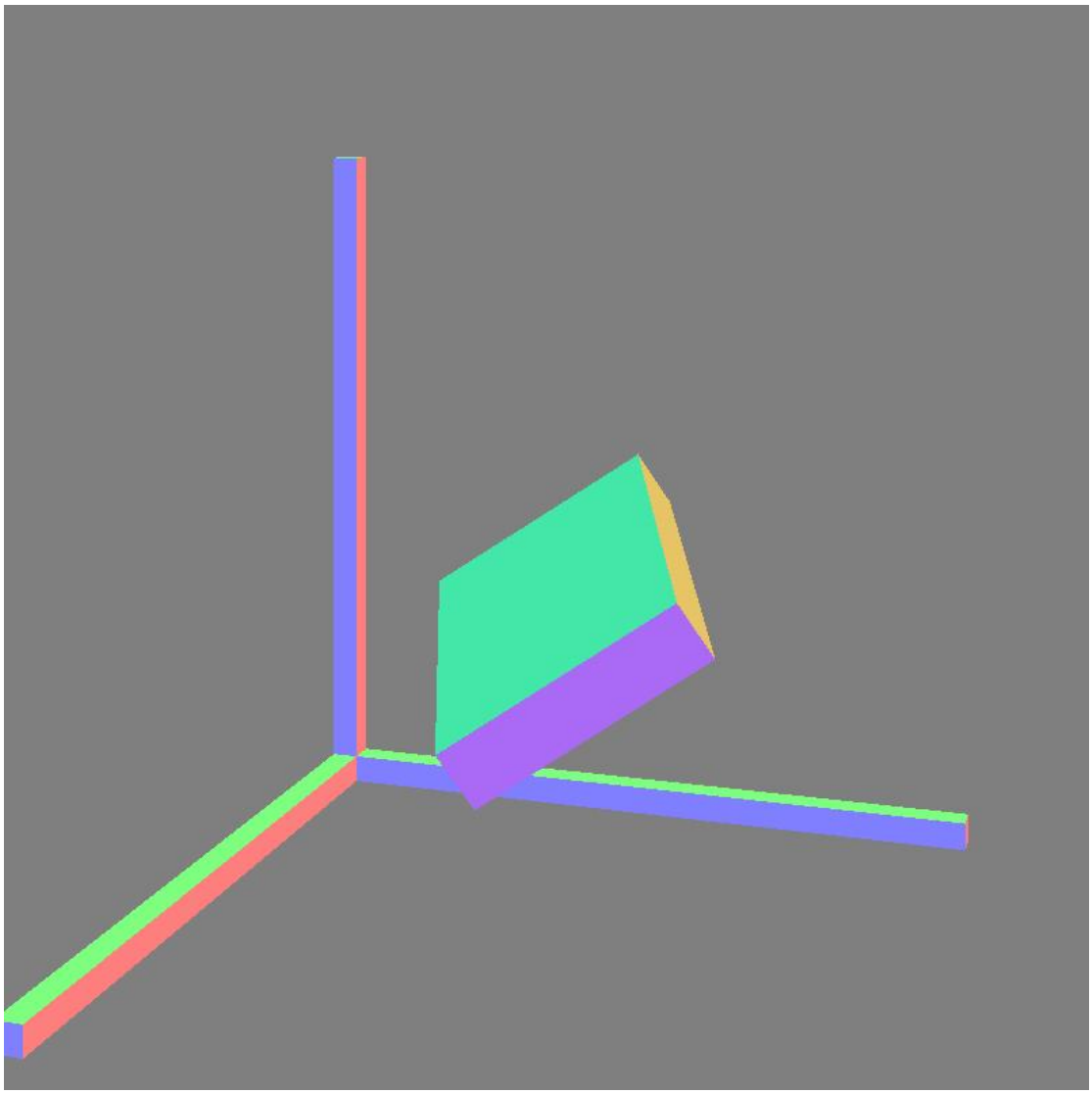


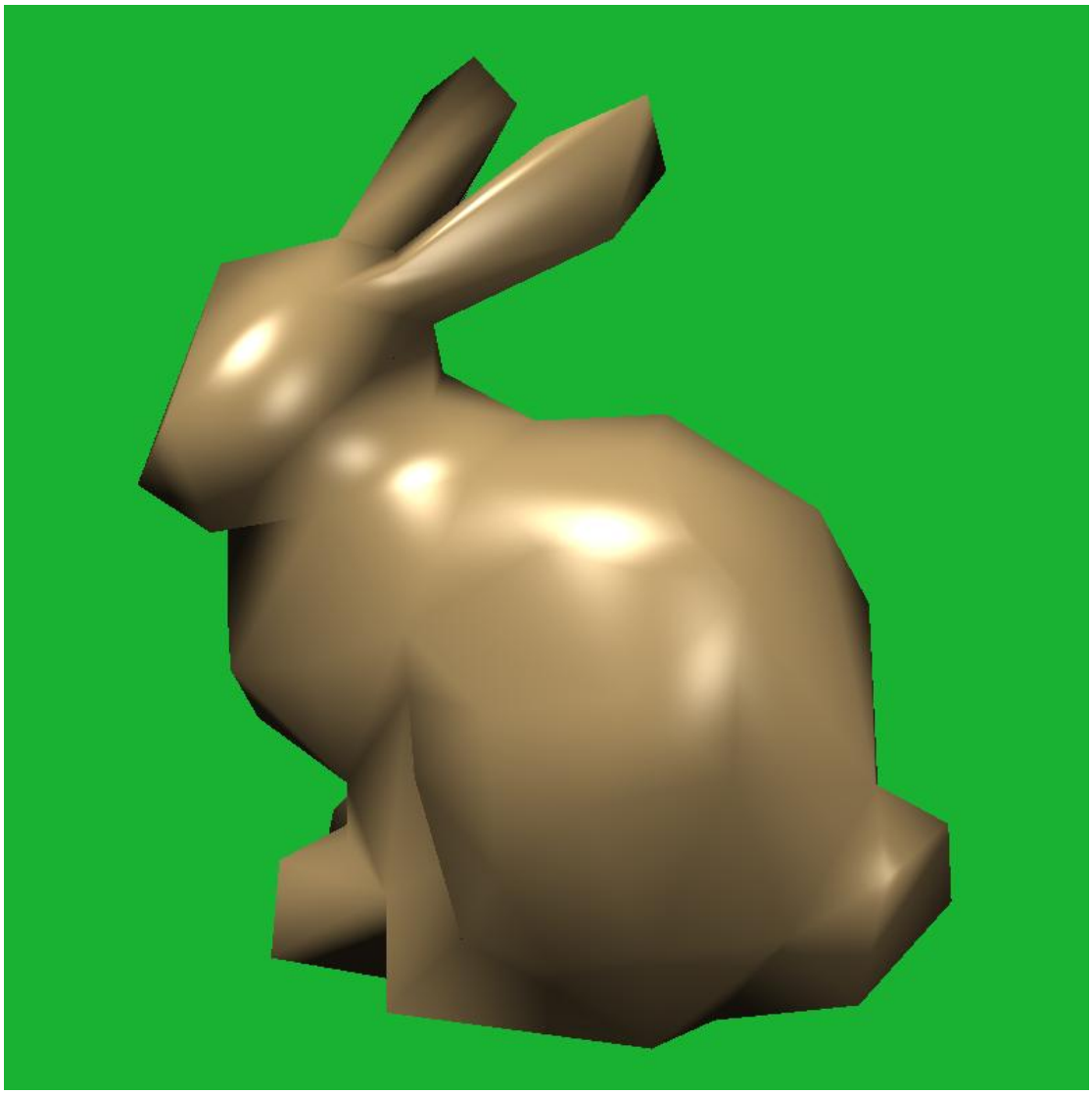


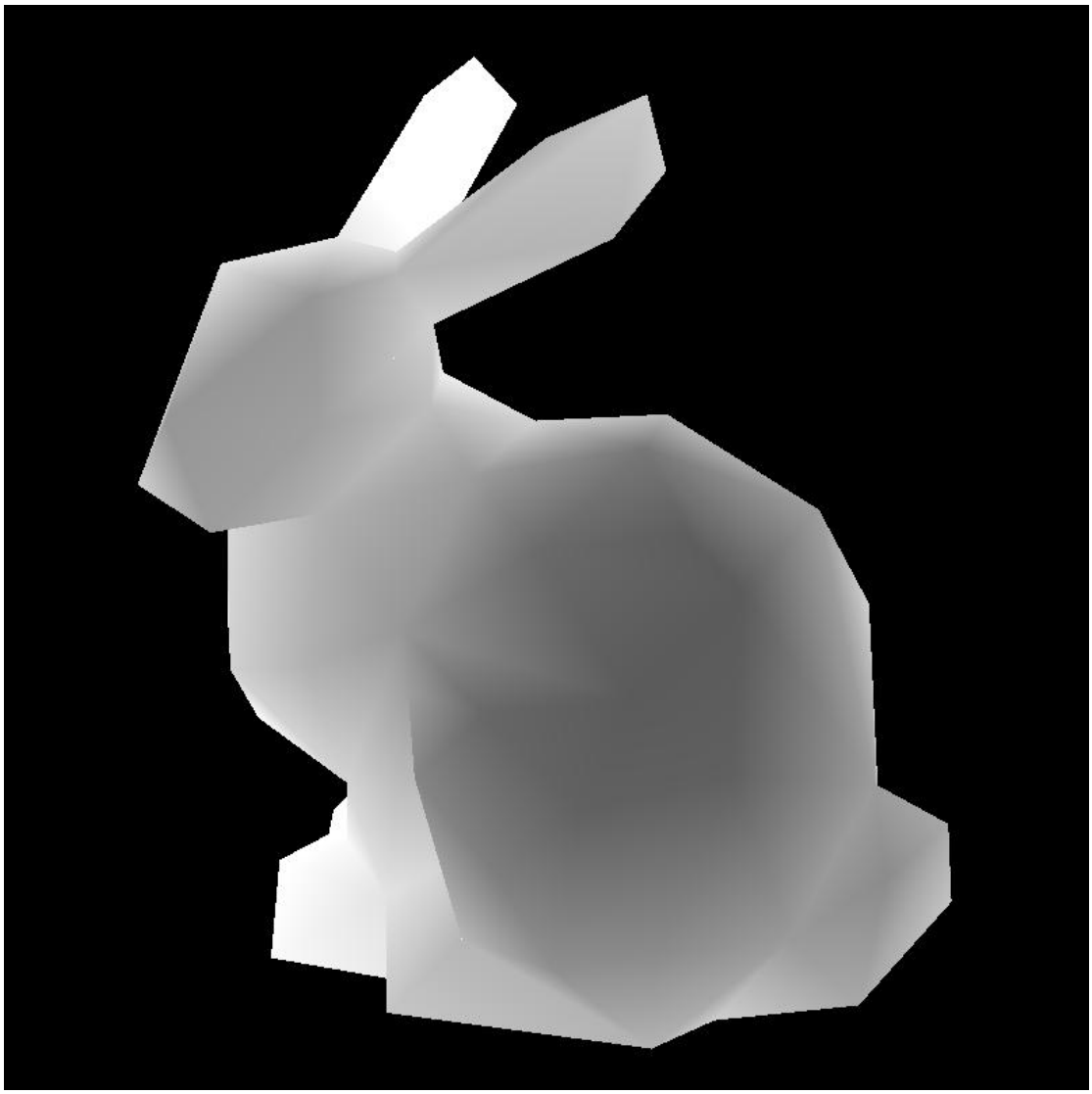


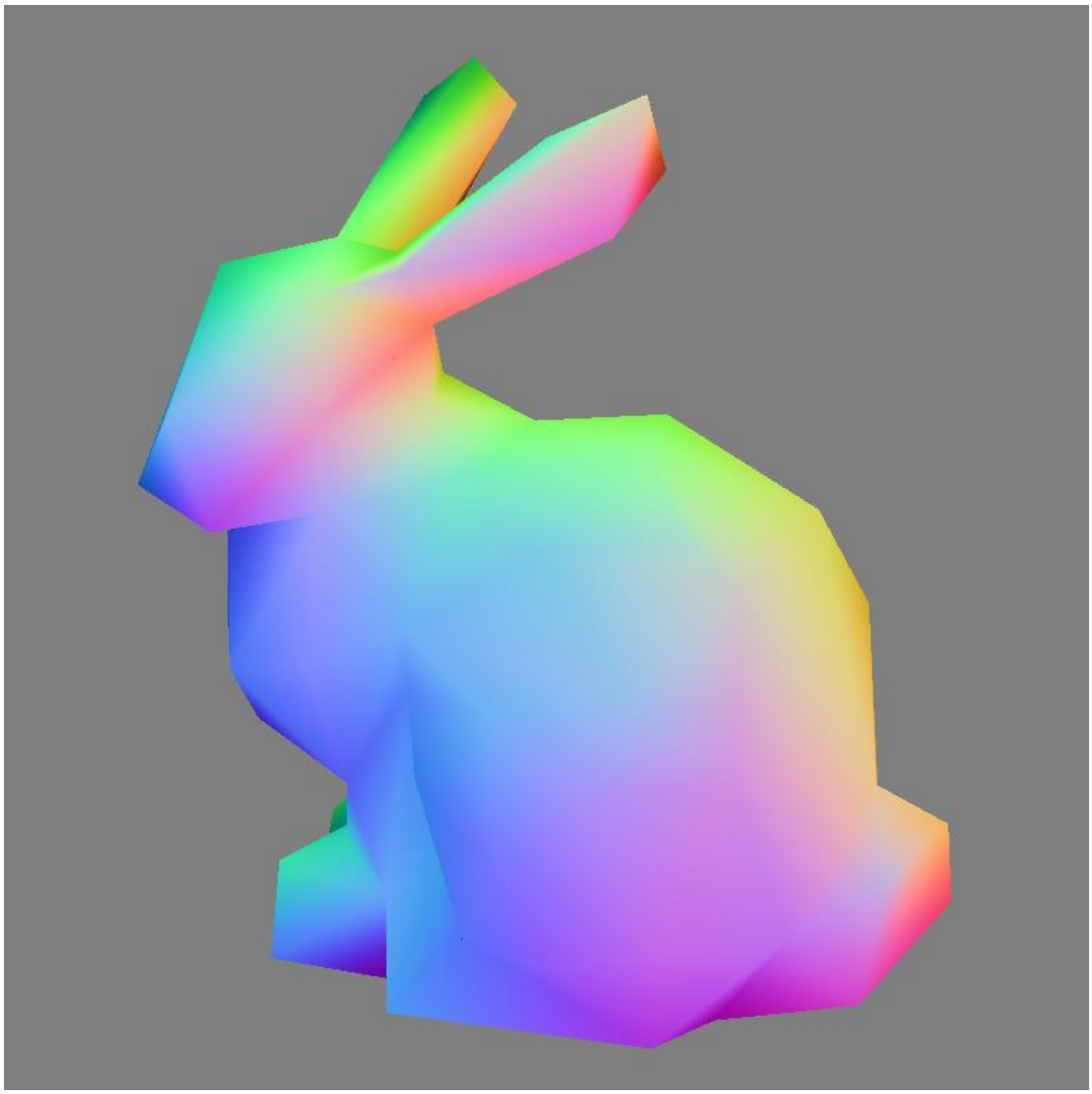


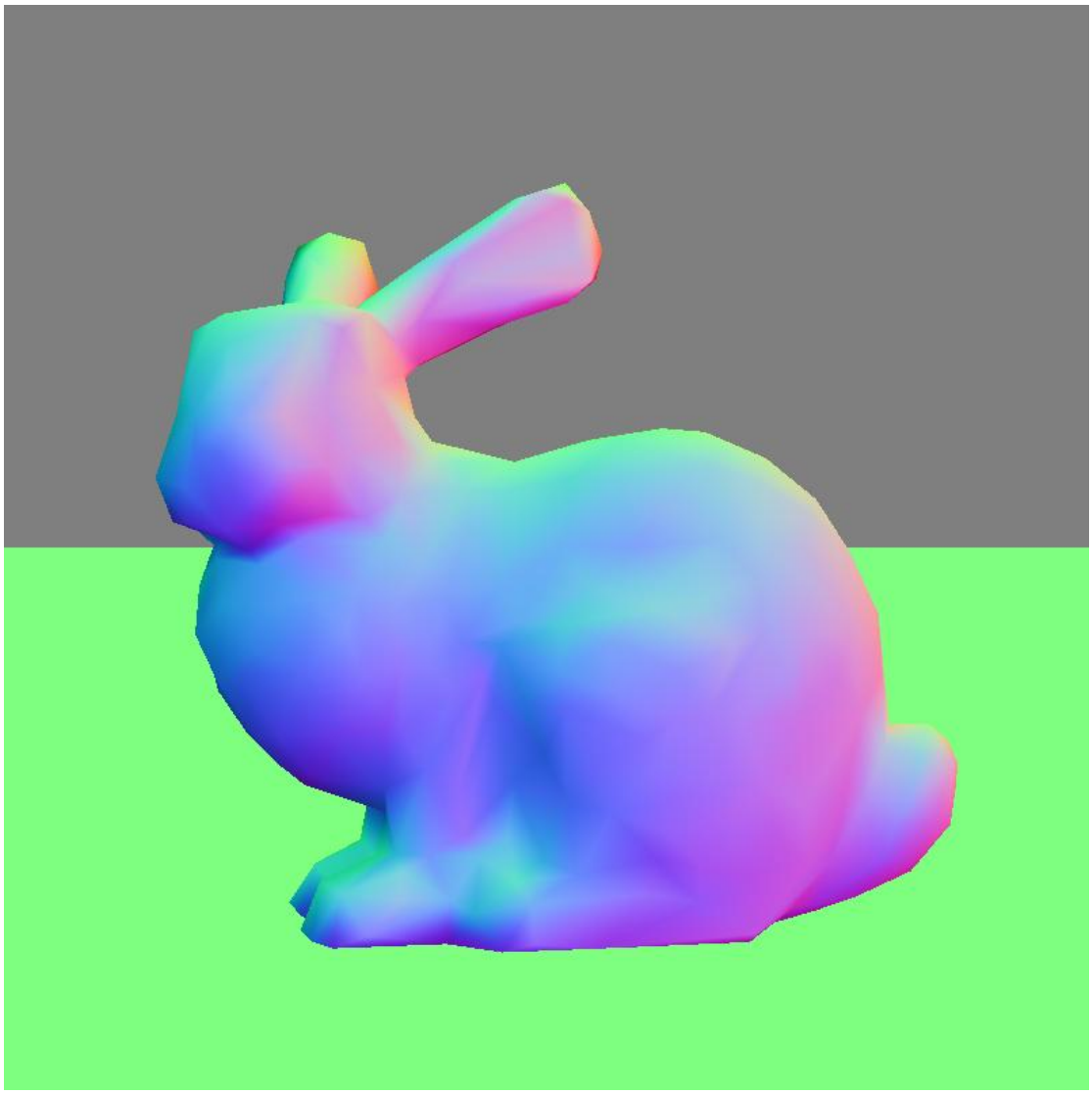


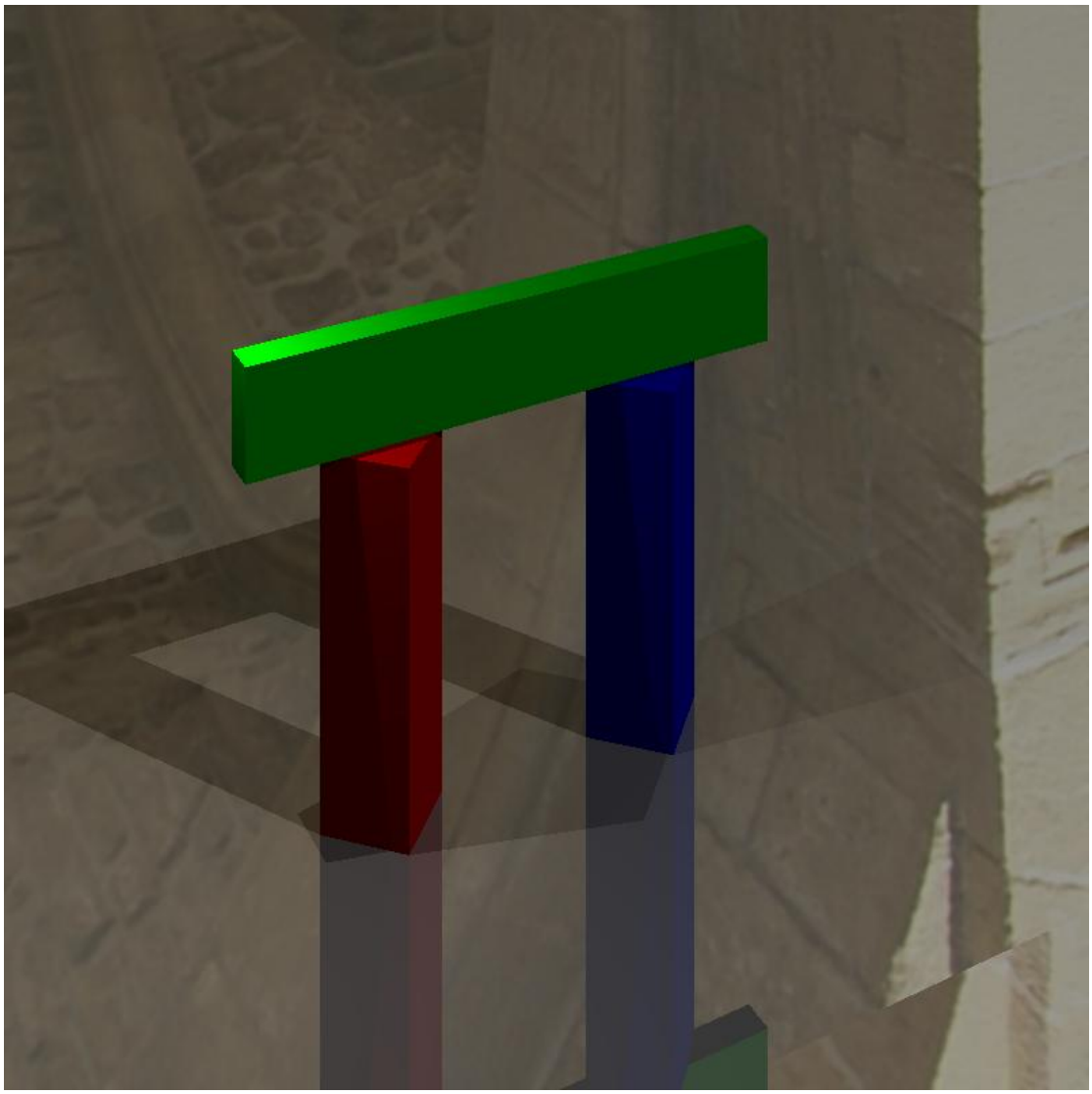


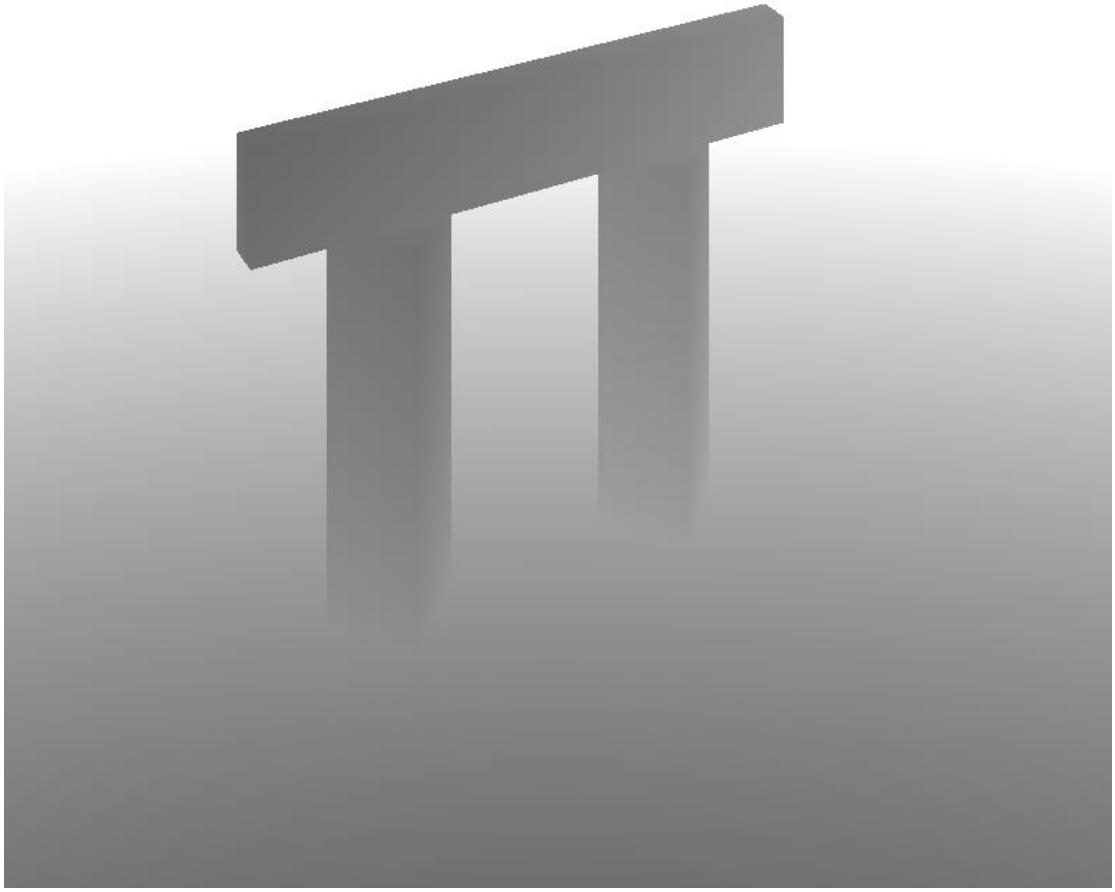


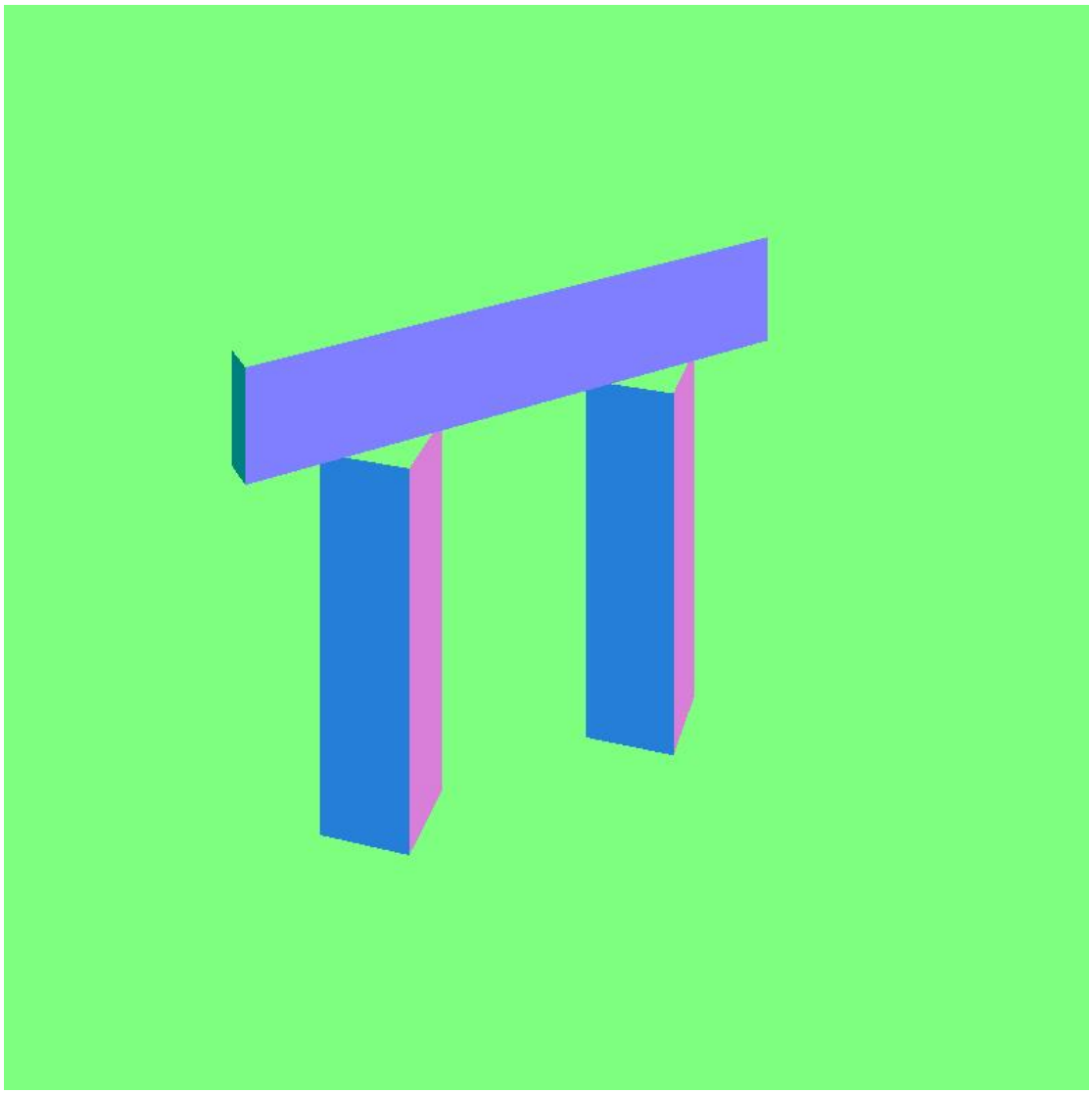












接下来是防锯齿后的效果：