计算机图形学 PJ2

姓名: 陈锐林, 学号:21307130148

2024 年 5 月 3 日

2.1： Phong 光照模型：

2.1.1: PointLight 的 getIllumination 实现

(1) 实现思路：

阅读 "*light.h*" 和 "*light.cpp*" 后可知，DirectionalLight 和 PointLight 都是基类 Light 的一个派生类，都继承了 getIllumination 这个虚函数。其目的都是将类的数据复制到输入的这几个引用内。

具体到 PointLight 类，其有 _position,_color,_falloff 三个私有成员。分别计算三个输出，tolight，即点到光源的距离矢量，取为 p−_position，根据 PPT 提示，要进行正则化；距离 distTolight 就是 |p − _position|；最后计算光强，用公式 $L = \frac{I}{\alpha d^2} = \frac{\text{\_color}}{\text{\_falloff(distToLight)}^2}$ 得到输出 intensity。

(2) 具体代码：

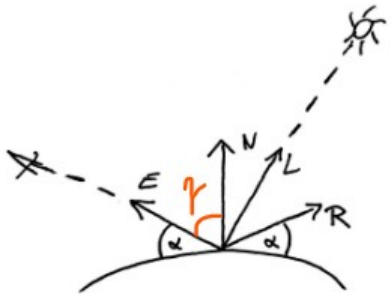
```
void PointLight::getIllumination(const Vector3f &p,
                                 Vector3f &tolight,
                                 Vector3f &intensity,
                                 float &distToLight) const
{
    // TODO Implement point light source
    // tolight, intensity, distToLight are outputs
    tolight = (_position - p).normalized();
    distToLight = (_position - p).abs();
    intensity = _color / (_falloff * distToLight * distToLight);
}
```

2.1.2: Material::shade() 实现

(1) 实现思路：

先看总体要求，我们要生成 Phong 光照模型的光，其由三部分组成：$I_{phong} = I_{ambient} + I_{diffuse} + I_{specular}$，其中 $I_{ambient}$ 由其他类给出，只需要考虑后两个。注意到他们的计算都用到了一个函数 $\mathbf{champ}(x, y)$，其实就是 $\max(0, xy)$。然后公式分别是 $I_{diffuse} = \mathbf{champ}(\mathbf{L}, \mathbf{N}) * L * k_{diffuse}$ 和 $I_{specular} = \mathbf{champ}(\mathbf{L}, \mathbf{R})^s * L * k_{specular}$。这里加粗的 $\mathbf{L}$ 是方向，而 L 是光强 (一不小心就弄混了)。

具体实现上，该函数输入了 $\mathbf{L}, \mathbf{N}$ 和光强 L，就是 dirToLight,hit 和 lightIntensity。根据提示和阅读 "$Material.h$"，$k_{diffuse}, k_{specular}$，分别由 $\_diffuseColor, \_specularColor$ 表示，系数 s 由 $\_shininess$ 表示。所以只需要求出理想反射矢量 $\mathbf{R}$ 即可。观察下图，$\mathbf{E}$ 和 $\mathbf{R}$ 关于 $\mathbf{R}$ 对称，假设 $\mathbf{N}$ 上的单位向量是 $\mathbf{e}$，而 $<\mathbf{R}, \mathbf{N}>=\gamma$。那么有：$\mathbf{R}\cdot\mathbf{e}=\cos\gamma$，$\mathbf{R}+\mathbf{E}=2\mathbf{e}(\cos\gamma)$，所以 $\mathbf{R}=2(\mathbf{E}\cdot\mathbf{e})\mathbf{e}-\mathbf{E}$。



(2) 具体代码：

先把各个变量表示出来，需要注意的是 hit 直接 getNormal 就是单位向量了，然后根据 "$Ray.h$" 的定义，getDirection 后取反才是我们上面的 $\mathbf{E}$。最后带入计算即可。

```
Vector3f N = hit.getNormal();
float clampLN = std::max(0.0f, Vector3f::dot(dirToLight, N));
Vector3f diffuse = clampLN * lightIntensity * _diffuseColor;
Vector3f E = -ray.getDirection();
Vector3f R = 2.0 * Vector3f::dot(E, N) * N - E;
float clampLR = std::max(0.0f, Vector3f::dot(dirToLight, R));
Vector3f specular = pow(clampLR, _shininess) * lightIntensity * _specularColor;
return diffuse + specular;
```

2.1.3: Renderer::traceRay() 初步实现

(1) 实现思路：

这个函数就是一个对前面的总结；要返回所有的光的叠加值。根据 PPT 的提示，光由 _scene.lights 得到；但有个特例就是前面被我们省略的环境光 ambient，要通过 _scene.getAmbientLight() 得到。对于由 _scene.lights 得到的每一束光，我们都去计算其漫反射和镜面反射的光照，就是上面的 diffuse 和 specular。

更具体地，首先我们要调用 2.1.1 的 getIllumination() 得到 tolight，intensity 和 disToLight 这几个量，之后输入到函数 shade() 计算。

(2) 具体代码：

如上所述的做法，用一个 for 循环控制光强的汇总即可。这里的变量 TLight 表示总的光照，根据 PPT，就只是简单相加即可，不需要考虑其他做法。

* 所有的测试结果会放在最后一起展示 *

```
Vector3f TLight(0, 0, 0);
TLight += _scene.getAmbientLight();
for (int i = 0; i < _scene.lights.size(); i++)
    {
        Vector3f dirToLight;
        Vector3f lightIntensity;
        float distToLight;
        _scene.lights[i]->getIllumination(r.pointAtParameter(h.getT()),
            dirToLight, lightIntensity, distToLight);
        TLight += h.getMaterial()->shade(r, h, dirToLight, lightIntensity);
    }
return TLight;
```

2.2：光线投射：

2.2.0：学习 Sphere 类：

首先，Sphere 类继承自 Object3D，并有了自己的私有成员：球心坐标和半径。并且具体实现了自己的 intersect 函数；intersect 函数通过联立方程来计算出交点坐标（或者无解），并取出大于 tmin 的最小根，当有必要时修正 hit 的 t/normal/-material。

2.2.1：完成 Plane 类：

(1) 具体思路：

平面的定义是 $P\mathrm{n} = \mathrm{dist}$，分别是点、法线、偏移。

首先补上 Plane 类的私有成员，根据其构造函数可知，分别是 **_normal**,_d；而 **_material** 已经在基类给出。

根据类似的垂直原理（PPT 直接给出），求出 $t = \dfrac{(\mathbf{p}' - \mathbf{o}) \cdot \mathbf{N}}{\mathbf{d} \cdot \mathbf{N}}$。其中 **N** 就是 **_normal**，**d** 就是光线的方向向量 (r.getDirection())，**o** 是光线源头 (r.getOrigin())，最后的 $\mathbf{p}'$ 就是 d**_normal**。

紧接着继续仿照 sphere 判断根，t<tmin 直接返回 false，之后更新 hit 的信息（当 $t < h.getT()$）。

(2) 具体代码：

```
const Vector3f &o = r.getOrigin();
const Vector3f &d = r.getDirection();
Vector3f T = _d * _normal - o;
float t = Vector3f::dot(T, _normal) / Vector3f::dot(d, _normal);
if (t < tmin)
    return false;
else if (t < h.getT()){
    h.set(t, this->material, _normal);
    return true;
}
return false;
```

2.2.2：完成 Triangle 类：

(1) 具体思路：

注意到 Triangle 类的定义已经完成，补充 intersect 函数就可以了。

但三角形可以通过判断和三个点的关系就得到结果。按照 PPT 的指引，直接使用 Möller-Trumbore 算法（参考了point here）。直接解方程 $\begin{bmatrix}\mathbf{B}-\mathbf{A} & \mathbf{C}-\mathbf{A} & -\mathbf{D}\end{bmatrix}\begin{bmatrix}u\\v\\t\end{bmatrix}=\begin{bmatrix}\mathbf{O}-\mathbf{A}\end{bmatrix}$。这里的 $\mathbf{A},\mathbf{B},\mathbf{C}$ 就是三角形内的三个顶点，而 $\mathbf{D}$ 就是光线的射线方向向量，$\mathbf{O}$ 则是光线起点，与前面一样，只是配合书写而大写了。

解方程后回到条件判断，首先 $u,v,(1-u-v)$ 都应在 $(0,1)$ 内；其次如果要更新 hit，$t \geq$tmin 且 $t < h.getT()$ 是不可或缺的；更新时我们用 AP 来更新 normal。由此完成这一部分。

(2) 具体代码：

```
Vector3f D = r.getDirection(), O = r.getOrigin();
Matrix3f T(_v[1] - _v[0], _v[2] - _v[0], -D);
Vector3f uvt = T.inverse() * (O - _v[0]);
float u = uvt[0], v = uvt[1], t = uvt[2];
if (u > 0 && v > 0 && u + v < 1 && t >= tmin && t < h.getT())
{
    Vector3f normal = (1 - u - v) * _normals[0] + u * _normals[1] + v * _normals[2];
    normal = normal.normalized();
    h.set(t, this->material, normal);
    return true;
}
return false;
```

2.2.3：完成 Transform 类：

(1) 实现思路：

注意到 Transform 类的构造函数中有 4 阶矩阵 m，所以我们要为其添上私有成员 $\_m$；并且在 "*Object3D.cpp*" 中要先完成构造函数（其实为啥不写前面呢 233，三个类两个放.cpp，一个放.h）。

根据 PPT，$\_m$ 的作用是从局部坐标系变到世界坐标系，所以反过来的变化矩阵就是 $\_m^{-1}$，或者是 $m.inverse()$。接着我们要判断 intersect，但我们并不知道这个旋转体到底是怎样的形状? 回到 Transform 类的构造函数，会发现其还有一个 Object3D 成员，这应该才是真正的类型，所以我们直接调用该类型的 intersect 函数（可能是前面的 Sphere 或者 Plane 或者 Triangle）。

在调用其他类的 Object3D 时，我们应该：(i) 变换光线 Ray,(ii) 变换 tmin（坐标系换了之后原先的 tmin 自然不准确）。若我们求解的 t(其他类的 intersect 完成) 是符合要求的，我们要更新 hit；此时就要将 normal 使用 m 变换回去。

还是能很清晰地感觉到因为 Transform 它不是指代的一个具体的类（如球、三角），所以实现上有很多不一样的地方，也感觉 OOP 的思想真的很重要。

(2) 具体代码：

这里的 Im 就是 inverse _m，在局部坐标系下的变量都以 Trans 前缀表示。

```
Matrix4f Im = _m.inverse();
Vector3f D = r.getDirection(),O = r.getOrigin();
Vector3f TransDir((Im * Vector4f(D, 0)).xyz());
float TransTmin = tmin * TransDir.abs();
Ray TransRay((Im * Vector4f(O, 1)).xyz(), TransDir);
if (_object->intersect(TransRay, TransTmin, h))
{
    Vector3f normal = (Im.transposed() * Vector4f(h.getNormal(), 0)).xyz();
    normal = normal.normalized();
    h.set(h.getT(), h.getMaterial(), normal);
    return true;
}
return false;
```

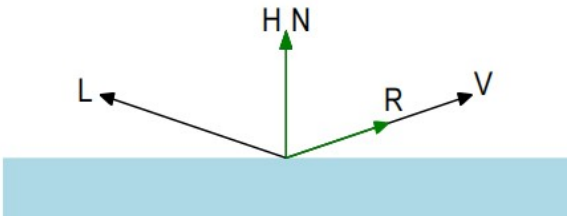
2.3：光线追踪与阴影投射：

2.3.1：光线追踪：

(1) 实现思路：

这里是在任务一的 traceRay() 上做修改，所以应该放在 for 循环内对每束光线都考虑多次反射。

根据 PPT 的意思理解，光线是向 **R** 的。根据示意图，这里的方向求法和前面任务一是一样的，不再赘述；之后我们需要定义下一层递归需要的 Hit 和 Ray，这里随机初始化一个 hit 就可以；但是 Ray 需要给新值，注意到 Ray 的函数 pointAtParameter，就是返回射线起点 + 偏置，这里配合当前 hit 就能得到。更简单来说，因为我们是在考虑多次反射，反射光线的起点就是这个交点，所以要调整 Ray 达到这个目的。借鉴阴影的投射，特地稍微更改了反射光线的起点。



(2) 具体代码：

```
if (bounces > 0)
{
    Vector3f D = r.getDirection();
    Vector3f N = h.getNormal().normalized();
    Vector3f R = (D - (2 * Vector3f::dot(D, N) * N)).normalized();
    Hit rHit = Hit();
    Ray rRay(r.pointAtParameter(h.getT()) + 0.005 * R, R);
    TLight += (h.getMaterial()->getSpecularColor()) * traceRay(rRay, 0.0f, bounces - 1, rHit);
}
```

2.3.2：阴影投射：

(1) 具体思路：

首先，这个功能要在调用 shade() 函数前；所以 traceRay() 的逻辑是：对每束光，先制造阴影；再调用 shade()；再考虑多次反射。

根据提示，我们需要从交点反向射光给光源，判断是否能达到；不行就不对这点作渲染。有了光线追踪的前例，我们知道，我们需要声明新的 Ray（类似地，起点要与当前交点作区别，稍作偏置），Ray 的方向就是和来时一样。说回到具体怎么判断是否需要阴影，我们知道，hit 的 t 是用一个很大的初始值初始化的，所以如果我们得到的 t 变小了，那就是不行的；同时，我们要考虑光源可能是碰到了很远很远的一个点，所以我们要将新的交点和射线起点的距离做一个判断。所以综上所述，shadow 当且仅当 t 值变小且这个点是在初始光源投射的长度之内。

(2) 具体代码：

```
Vector3f newOrigin = r.pointAtParameter(h.getT()) + 0.01 * dirToLight;
Ray sRay(newOrigin, dirToLight);
Hit sHit = Hit();
Vector3f sTrace = traceRay(sRay, 0.0f, 0, sHit);
bool isShadow = sHit.getT() < std::numeric_limits<float>::max();
float newDist = (sRay.pointAtParameter(sHit.getT()) - newOrigin).abs();
if (isShadow && newDist < distToLight)
    continue;
```

2.4：抗锯齿化：

2.4.0：学习已有的 Render 函数：

首先这里是先用输入的 h 和 w 初始化了三个 Image 对象(image,nimage,dimage)，用于放置图像的颜色、法线、深度。之后对每一个像素做处理，这里对坐标做了归一化；即 “ndcx = 2 * (x / (w - 1.0f)) - 1.0f”，“2 * (y / (h - 1.0f)) - 1.0f”；根据 x 和 y 的变化，我们知道，ndcx 和 ndcy 都取到 $[-1, 1]$。

之后根据我们前面实现的功能，用 generateRay 产生光线；用 traceRay 完成追踪。之后把结果用 Image 的 setPixel 设置每个像素。最后保存图片即可。

2.4.1：总体思路：

首先根据 PPT 的指引，我们要做的操作，按顺序是：（在 “___args.filter” 的指令下）上采样 3 倍分辨率；（在 “___args.jitter” 的指令下）进行 16 次充分抖动采样（jitter）；（在 “___args.filter” 的指令下）进行 σ 为 1 的高斯滤波。

对于操作 1 和 3，我们设变量 k 来表示是否上采样和滤波；k 的初值为 1，如果要滤波则改为 3。另取三个 Image 变量（kimage,knimage,kdimage），其长宽由 k×w,k×h 决定，存储上采样后的图，对其进行采样设置；在操作 3 时进行高斯滤波或者直接赋值回去。这样保证了输出图的形状是正确的，（$k = 1$：h×w 的

kimage→(直接赋值)h×w 的 image；$k = 3$：3h×3w 的 kimage→(高斯滤波)h×w 的 image)。

而对于操作 2，在原先的采样过程上拓展。首先我们进行 16 次一样的操作（如上 2.4.0 所述）：用随机的偏移去更新 ndcx 和 ndcy，生成新的光线 r1，得到对应的 color。对于这 16 次操作我们要对 color 取均值，并且注意到我们后面 setPixel 时要用到 hit 的 normal 和 t，所以这两个值也要保存并且取均值。

2.4.2：抖动采样：

(1) 实现思路：

聚焦在需要抖动的情况，首先解决偏移的生成。我们要生成的偏移在 $[0, 1]$，于是只要用 rand()/RAND_MAX 就能取到；并且这个偏移要归一化（类似 ndcx 和 ndcy)。

其次是剩余的代码逻辑。用 Vector3f 的 normal 和 colors、float 的 t 存储后续 setPixel 要用到的参数。执行操作 16 次并且各自取均值。

(2) 具体代码：

```
Vector3f normal = Vector3f(0,0,0);
Vector3f colors = Vector3f(0,0,0);
float t = 0.0;
if(_args.jitter){
    for(int i = 0;i != 16;i++){
        srand(time(NULL));
        double random_x = (double)rand() / RAND_MAX;
        double random_y = (double)rand() / RAND_MAX;
        random_x = ndcx + random_x * 2 / (kw - 1.0f);
        random_y = ndcy + random_y * 2 / (kh - 1.0f);
        Ray r1 = cam->generateRay(Vector2f(ndcx, ndcy));
        Hit hit1;
        colors += traceRay(r1, cam->getTMin(), _args.bounces, hit1);
        normal += hit1.getNormal();
        t += hit1.getT();
    }
    colors = colors / 16;
    normal = normal / 16;
    t = t / 16;
}
```

2.4.3：高斯滤波：

(1) 实现思路：

遍历 image 的每个像素，对其进行高斯滤波。用变量 i（-1,0,1）和 j（-1,0,1）控制滤波，那么根据绝对值来看就是 $[[2, 1, 2], [1, 0, 1], [2, 1, 2]]$，那么根据绝对值对其分类赋权值即可。注意到我们 getPixel 的 x 和 y 要记得补 1，免得越界。

(2) 具体代码:

```
for (int y = 0; y < h; ++y) {
    for (int x = 0; x < w; ++x) {
        Vector3f color(0.0f, 0.0f, 0.0f);
        Vector3f normal(0.0f, 0.0f, 0.0f);
        Vector3f depth(0.0f, 0.0f, 0.0f);
        for (int j = -1; j <= 1; ++j) {
            for (int i = -1; i <= 1; ++i) {
                float weight = 1.0f;
                if (abs(i) + abs(j) == 1) weight = 2.0f;
                else if (abs(i) + abs(j) == 0) weight = 4.0f;
                int neighbor_x = x * k + i + 1;
                int neighbor_y = y * k + j + 1;
                color += kimage.getPixel(neighbor_x, neighbor_y) * weight;
                normal += knimage.getPixel(neighbor_x, neighbor_y) * weight;
                depth += kdimage.getPixel(neighbor_x, neighbor_y) * weight;
        }
    }
    image.setPixel(x, y, color / 16.0f);
    nimage.setPixel(x, y, (normal /16.0f).normalized());
    dimage.setPixel(x, y, depth / 16.0f);
    }
}
```

2.5: 结果生成:

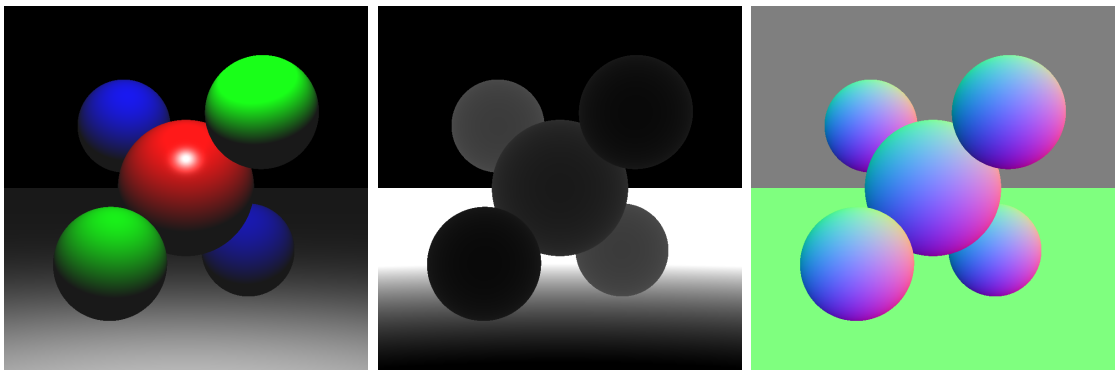
2.5.1: 生成指令:

对于前 5 个场景,指令如下:"build/a2 -size 800 800 -input data/scene01_plane.txt -output test/a01.png -normals test/a01n.png -depth 8 18 test/a01d.png"。(除了场景五的深度图用的"0.8 1.0" 的参数)。

对于场景 6 和 7,指令如下:"build/a2 -size 800 800 -input data/scene06_bunny_1k.txt -bounces 4 -output test/a06.png -normals test/a06n.png -depth 8 18 test/a06d.png"。

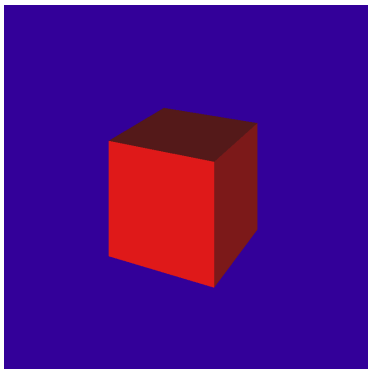
2.5.2: 生成图片:

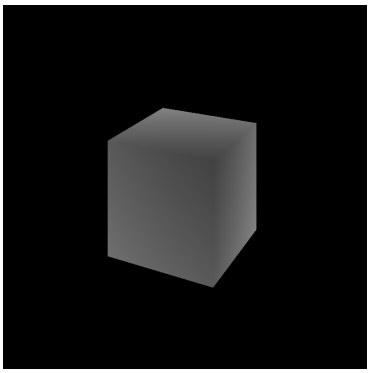
(1)scene01_plane:

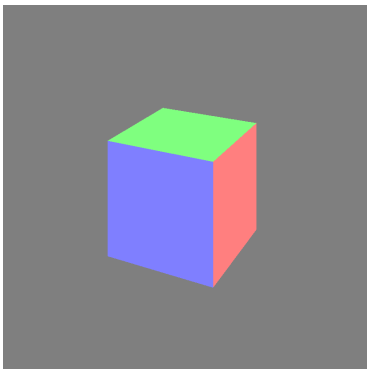


scene01-原图　　scence01-深度图　　scene01-法线图

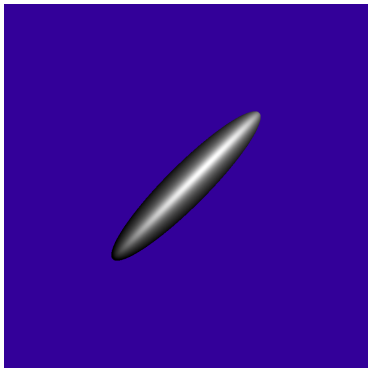
(2)scene02_cube:

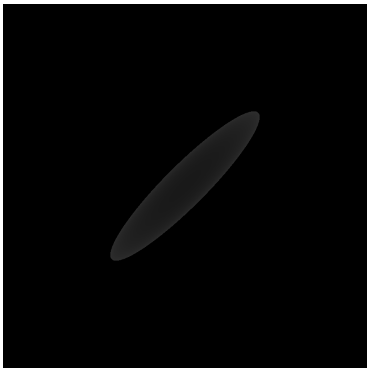

scene02-原图

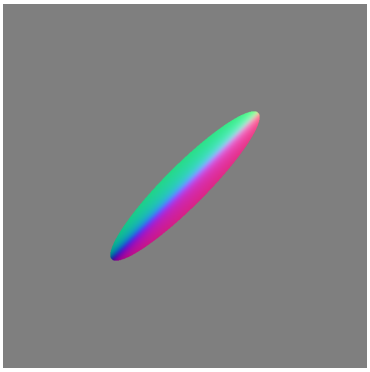

scence02-深度图


scene02-法线图

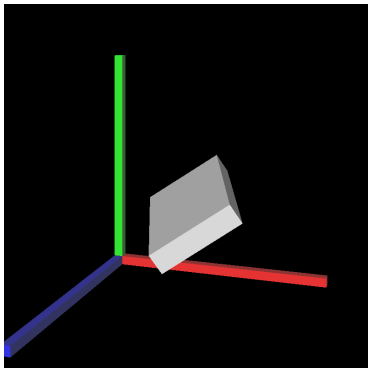
(3)scene03_sphere:


scene03-原图

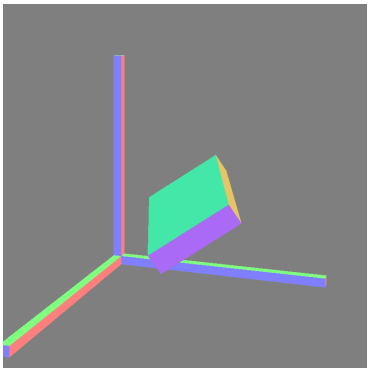

scence03-深度图


scene03-法线图

(4)scene04_axes:


scene04-原图


scence04-深度图

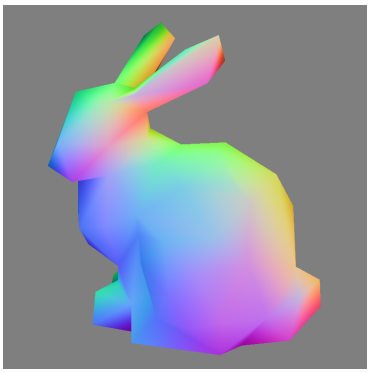

scene04-法线图

位置不够了，下一页接着放。

(5)scene05_bunny_200:


scene05-原图

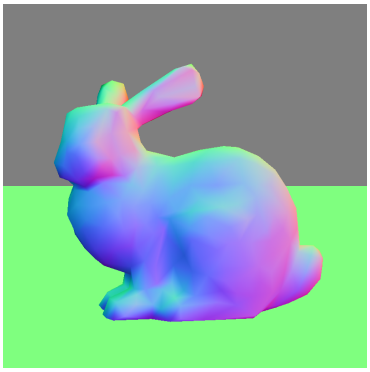

scence05-深度图


scene05-法线图

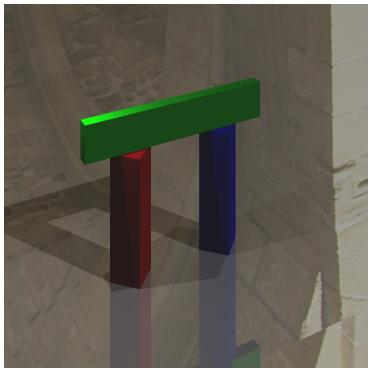
(6)scene06_bunny_1k:

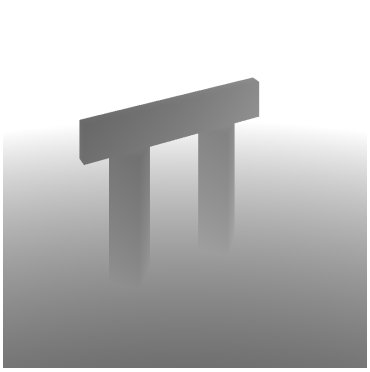

scene06-原图

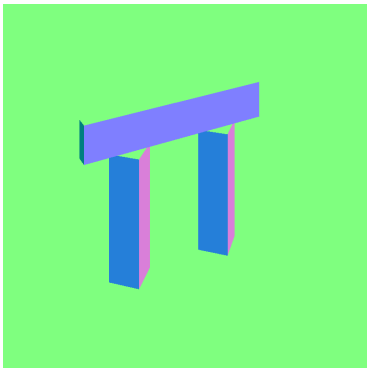

scence06-深度图


scene06-法线图

(7)scene07_arch:


scene07-原图


scence07-深度图


scene07-法线图

下一页还有抗锯齿图像。

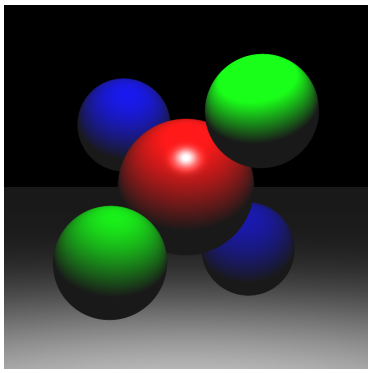
2.5.3：抗锯齿及生成样例：

(1) 指令：

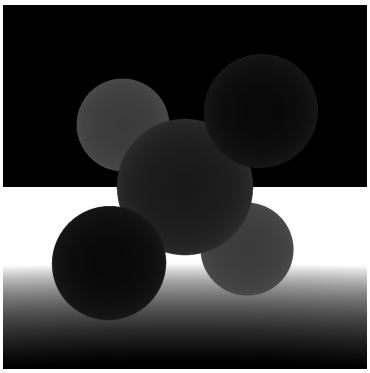
关注抗锯齿前后的 scene01，指令为：”build/a2 -size 800 800 -input data/scene01_plane.txt -output test/c001.png -normals test/c001n.png -depth 8 18 test/c001d.png -jitter -filter”。（话说这里用 800 800 的分辨率，运行时间真的好久、）。

(2) 生成图像及对比：

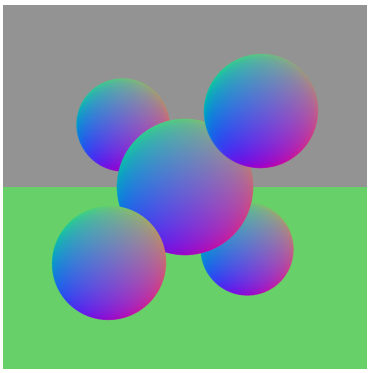
(i) 总体看：



scene01-抗锯齿-原图

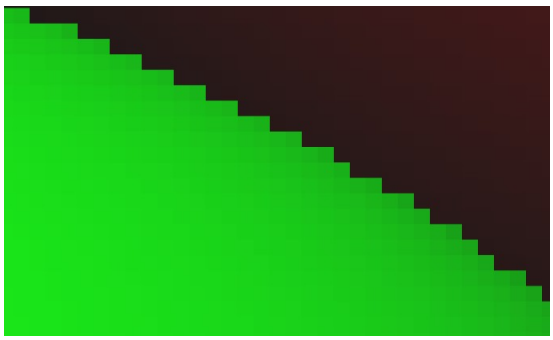


scence01-抗锯齿-深度图

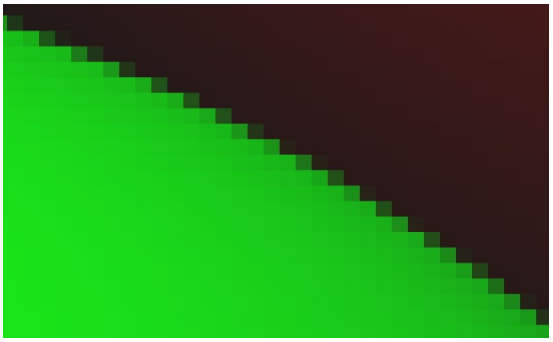


scene01-抗锯齿-法线图

(ii) 细节对比：能看到使三角形的锯齿不那么明显了



scene01-原图-细节



scence01-抗锯齿-原图-细节