光子映射实验报告

本次作业我参考了诸多资料，采用了光线跟踪加**光子映射**的形式实现。把场景的光照分为三个部分：直接光照(漫反射表面)，间接光照，镜面和折射光照，直接光照和镜面和折射光照由光线跟踪来计算，间接光照由光子图来计算，将光子图可视化。

1. 首先我定义了光线、颜色等基础类，包括基础的加减乘除，定义颜色的光强为 rgb 分量的平均值，定义了光线的模长、单位向量、垂直向量、随机向量、反射、折射、旋转、散射等函数。
2. 接着定义输出输入图片的辅助类，使用 opencv 读入输出 jpg 格式的图片。
3. 接着我定义了物品类，由 object 基类派生出 Sphere、plain、bezier 三个子类。Object 中有一个 Material 类储存材质的信息，包括颜色（漫反射、反射、高光），吸收的颜色（光线通过折射的物体的内部），折射率、反射率等，以及**纹理贴图**的图片。在 object 类中还储存了哈希值，用以判断一束光线是否有相同的路径（即击中同一个序列的物品）。还有最近一次碰撞的信息，包括碰撞的位置、法向量、光从发射到碰撞所经过的距离，以及碰撞的方向（前后）。

```
int code;//哈希值，用于判断某个光子或光是否经过同一序列的物体和光，若不同则
  有可能出现锯齿
Material* material;
Object* next;
ic:
Point crash_point;//碰撞点
Ray crash_normal_vector;//碰撞点法向量
double crash_distance;//碰撞点离出发点的距离，用来判断两个物体的远近
bool front;//在内/外或前/后碰撞
```

在球体计算碰撞时，使用判别式解二次方程，得到碰撞点。

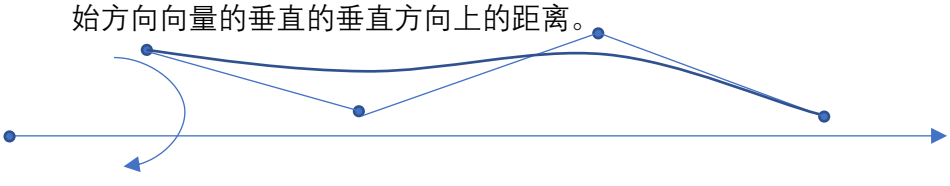
```
Ray ray_v = direction.unit();
Ray ray_p = origin - center;
double b = -ray_p.dot(ray_v);
double discriminant = b * b - ray_p.M2() + r * r;//判别式 b2-4ac
if(discriminant > ZERO) {
    discriminant = sqrt(discriminant);
    double t1 = b - discriminant, t2 = b + discriminant;
    if(t2 < ZERO)
        return false;
    else if(t1 > ZERO) {
        crash_distance = t1;
        front = true;
    }
    else {
        crash_distance = t2;
        front = false;
    }
}
else
    return false;
crash_point = origin + ray_v * crash_distance;
crash_normal_vector = (crash_point - center).unit();
if(!front)
    crash_normal_vector = -crash_normal_vector;
return true;
```

计算平面碰撞则计算光线方向和平面法向量的夹角，判断是否平行、光源是否在面内，以及碰撞方向。

```
double theta = normal_vector.dot(ray_v);
if(fabs(theta) < ZERO)//平行
    return false;
double dist = (normal_vector * distance - origin).dot(normal_vector) /
  theta;
if(dist < ZERO)//点在面内
    return false;
crash_distance = dist;
front = (theta < 0);
crash_point = origin + ray_v * crash_distance;
crash_normal_vector = (front) ? normal_vector : -normal_vector;
```

计算纹理时，则把碰撞点的球面坐标映射到平面上，得到其颜色并返回。

对于 **Bezier** 图形，通过输入起始点，起始方向得到图形的位置与构造方向。接着输入 n 组二维点作为控制点，其中第一维为起始方向的距离，第二维为与起始方向向量的垂直的垂直方向上的距离。



接着将这条曲线沿起始方向旋转，得到一个三维图形。使用一个**包围球**，球心为起始方向最大最小点的中点，半径为中心与离中心最远的点的距离。计算碰撞时，先判断是否与包围球碰撞，是的话取碰撞点离垂直方向的角度为初始角度，在（0-1）中取四个等距的点为 u 的初值，碰撞点的距离为 t 的初值，进行**牛顿迭代**。一开始我使用三维曲面来表示 Bezier 曲面，可是在求交上需要迭代的次数较多，效率较低。采用曲线旋转可以较好地确定包围球、以及迭代初值。

```
bool result = false;
if(!result) {
    for(double i = 0.0; i <= 4.0; i++) {
        if(newton(origin, direction, 0 + i / 3, theta + PI / 4, box-
          >crash_distance))
            result = true;
        if(newton(origin, direction, 0 + i / 3, theta - PI / 4, box-
          >crash_distance))
            result = true;
        if(result)
            continue;
        if(newton(origin, direction, 0 + i / 3, theta + 3 * PI / 4, box-
          >crash_distance))
            result = true;
        if(newton(origin, direction, 0 + i / 3, theta - 3 * PI / 4, box-
          >crash_distance))
            result = true;
    }
}
```

迭代方程采用书上的方程，最多进行五轮。对 u 求导即为对曲线求导再旋转，对 v 求导为对旋转过程中的三角函数求导，对 t 求导即为光的方向。

```
Point P = getP(u, v);
Point Pu = dPu(u, v);
Point Pv = dPv(u, v);
Point df = P - (origin + direction * t);
for(int k = 0; k < 5; k++){
    if(df.M() < ZERO)
        break;
    double D = direction.dot(Pu*Pv);
    t = t + Pu.dot(Pv*(df)) / D;
    u = u + direction.dot(Pv*(df)) / D;
    v = v - direction.dot(Pu*(df)) / D;
    P = getP(u, v);
    Pu = dPu(u, v);
    Pv = dPv(u, v);
    df = P - (origin + direction * t);
}
```

4. 光源类，由基类派生出平面光源和点光源。由光源计算从该光打在某个物体上

的直接光照，包括漫反射和高光。

```
if (hash != NULL)
    *hash = (*hash + code) % HASH_MOD;
if (object->getmaterial()->diffuse_rate > ZERO) {
    result += color * object->getmaterial()->diffuse_rate * dot / dist2;
}
if (object->getmaterial()->specular_rate > ZERO) {
    result += color * object->getmaterial()->specular_rate * pow(dot,
      SPEC_POWER) / dist2;
}
```

计算实际光照时，对点光源判断光源与物体之间是否有遮挡，有则返回空颜色（0,0,0），否则返回光照结果。对面光源使用随机多次采样，将面分为 16 个小方格，每个小方格内随机取几点发出光照，得到结果平均后即为该点直接光照。这个方法可以计算**软阴影**。

```
bool shade;
for(int i = -2; i < 2; i++)
    for(int j = -2; j < 2; j++)
        for(int k = 0; k < shade_quality; k++) {//多次采样
            Ray ray_v = center + vector_x * ((ran() + i) / 2) + vector_y *
              ((ran() + j) / 2) - object->crash_point;//在方格内随机取点
            shade=false;
            double dist = ray_v.M();
            Object* head = object_head;
            while(head) {
                if(head->collide(object->crash_point, ray_v) && head-
                >crash_distance < dist) {//挡着
                    shade=true;
                    break;
                }
                head = head->getnext();
            }
            if(!shade)
                result += irradiance(object, ray_v, NULL);
        }
```

5. 构造场景类和相机类，其中场景类负责读入 txt 文件中的初始化数据，初始化一个场景，包括物体和光源。相机类负责储存图像的参数，包括大小、光子数、图片阴影质量。从相机中心发出光线，方向为图片的像素的相对于中心的角度。

```
return normal_vector + vector_y * (2 * i / h - 1) + vector_x * (2 * j / w -
  1);
```

6. 构造光子和光子图，光子包含颜色、位置、方向、以及在 kd 树中的分界面。在光子图中，使用数组储存 kd 树，递归的构建树。

Algorithm BUILDKDTREE(P, *depth*)

1. **if** P contains only one point
2. **then return** a leaf storing this point
3. **else if** *depth* is even
4. **then** Split P with a vertical line ℓ through the median x-coordinate into P_1 (left of or on ℓ) and P_2 (right of ℓ)
5. **else** Split P with a horizontal line ℓ through the median y-coordinate into P_1 (below or on ℓ) and P_2 (above ℓ)
6. $v_{\text{left}} \leftarrow$ BUILDKDTREE(P_1, *depth* + 1)
7. $v_{\text{right}} \leftarrow$ BUILDKDTREE(P_2, *depth* + 1)
8. Create a node v storing ℓ, make v_{left} the left child of v, and make v_{right} the right child of v.
9. **return** v

查询时则构造一个最近光子的结构体，存储位置以及得到的光子。采用算法，具体见文件。

输入：以构造的kd树，目标点x；
输出：x 的最近邻
算法步骤如下：

1. 在kd树种找出包含目标点x的叶结点：从根结点出发，递归地向下搜索kd树。若目标点x当前维的坐标小于切分点的坐标，则移动到左子结点，否则移动到右子结点，直到子结点为叶结点为止。
2. 以此叶结点为"当前最近点"。
3. 递归的向上回溯，在每个结点进行以下操作：
   （a）如果该结点保存的实例点比当前最近点距离目标点更近，则更新"当前最近点"，也就是说以该实例点为"当前最近点"。
   （b）当前最近点一定存在于该结点一个子结点对应的区域，检查子结点的父结点的另一子结点对应的区域是否有更近的点。具体做法是，检查另一子结点对应的区域是否以目标点位球心，以目标点与"当前最近点"间的距离为半径的圆或超球体相交：
   如果相交，可能在另一个子结点对应的区域内存在距目标点更近的点，移动到另一个子结点，接着，继续递归地进行最近邻搜索；
   如果不相交，向上回溯。
4. 当**回退到根结点时，搜索结束**，最后的"当前最近点"即为x 的最近邻点。

得到光子后，取其平均能量为该点的间接光照。

7. 光子映射算法类，由光源发出光子，对其进行追踪，如果在路径上有物体，使用轮盘赌判断它是漫反射、反射还是折射。漫反射使其沿法向量散射，折射则判断其在物体内部行走的距离，计算吸收光的能量。具体见 pt.cpp 文件，函数名即为其功能。在光子碰撞时，对反映间接光照的光子（即多次碰撞后、非镜面和折射面）进行储存。最多递归 8 层。

```
if(dep > MAX_PHOTONTRACING_DEP)
    return;
Object* nearest_object = scene->nearestobject(photon.position ,
  photon.direction);
if (nearest_object != NULL) {
    Object* object = nearest_object->copy();
    photon.position = object->crash_point;
    if (object->getmaterial()->diffuse_rate > ZERO && dep > 1)
        tree->insert(photon);//间接光照
    double prob = ran();//光子要不漫反射，要不反射，要不折射
    if (!diffusion(object , photon , dep , prob))
        if (!reflection(object , photon , dep , prob))
            refraction(object , photon , dep , prob);
    delete object;
}
```

```
bool Photontrace::diffusion(Object* object , Photon photon , int dep , double&
  prob) {
    double probability = (object->getmaterial()->diffuse_rate + object-
      >getmaterial()->specular_rate) * object->getmaterial()->color.intensity();
    if (probability <= prob) {//轮盘赌
        prob -= probability;
        return false;
    }
    photon.intensity *= object->getmaterial()->color / object->getmaterial()-
      >color.intensity();//受材料颜色影响，保持光强不变
    photon.direction = object->crash_normal_vector.diffuse();//随机发散
    trace(photon , dep + 1);
    return true;
```

```
bool Photontrace::reflection(Object* object , Photon photon , int dep , double&
  prob) {
    double probability = object->getmaterial()->reflect_rate * object-
      >getmaterial()->color.intensity();
    if(probability <= prob) {
        prob -= probability;
        return false;
    }
    photon.intensity *= object->getmaterial()->color / object->getmaterial()-
      >color.intensity();
    photon.direction = photon.direction.reflect(object->crash_normal_vector);
    trace(photon , dep + 1);
    return true;
```

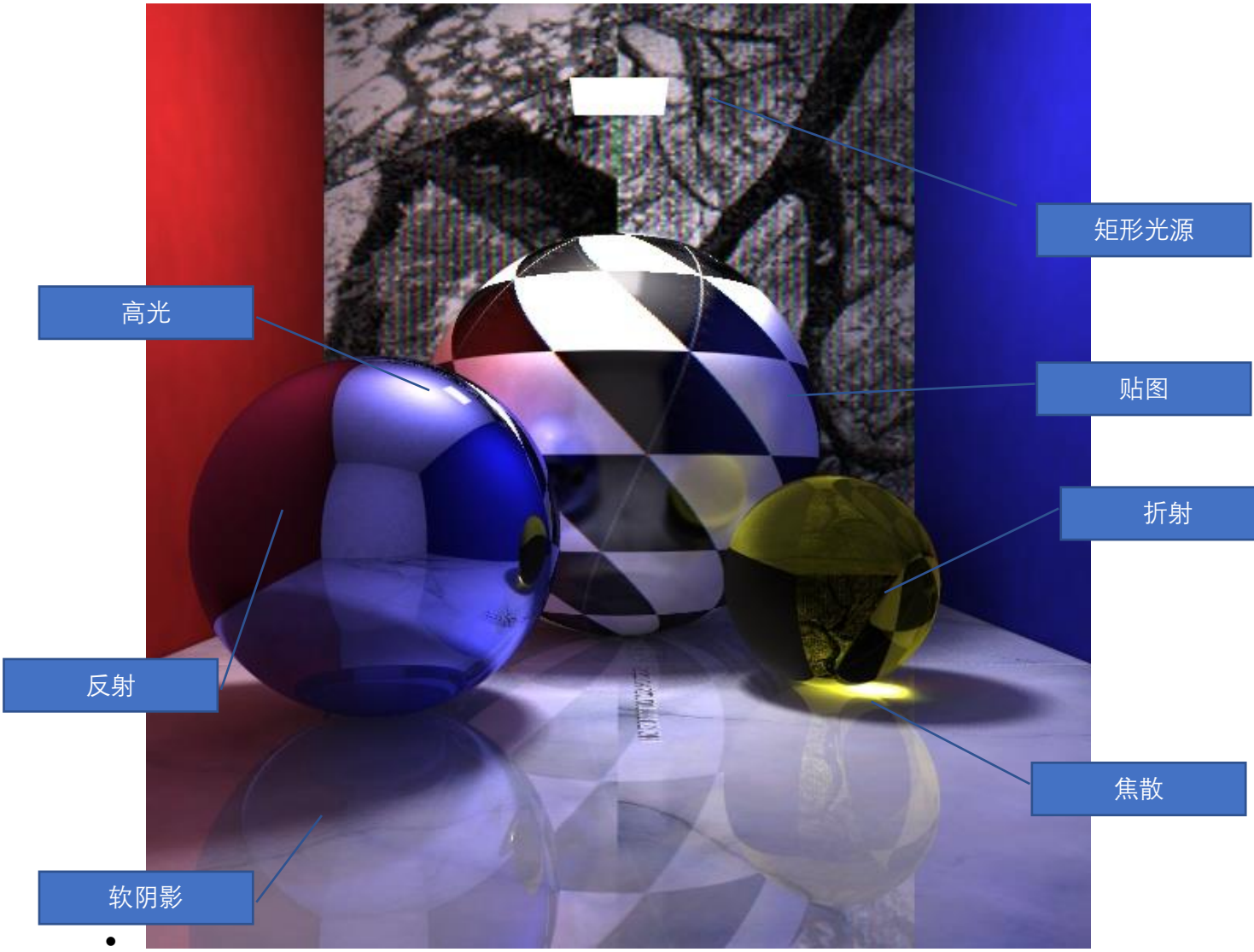
```
bool Photontrace::refraction(Object* object , Photon photon , int dep , double&
  prob) {
    double probability = object->getmaterial()->refract_rate;
    if (!object->front) {
        Color absor = -(object->getmaterial()->absor * object->crash_distance);
        absor = absor.exp();
        probability *= absor.intensity();
        photon.intensity *= absor / absor.intensity();
    }
    if(probability <= prob) {
        prob -= probability;
        return false;
    }
    double n = object->getmaterial()->n;
    if (object->front)
        n = 1 / n;
    photon.direction = photon.direction.refract(object->crash_normal_vector ,
      n);
    trace(photon , dep + 1);
    return true;
}
```

8. 光线跟踪算法，先由场景类构建场景，再从视点发出光线，对每一条光线进行跟踪。光线碰到物体后，根据物体的反射、折射、漫反射特性，分析其直接光照，间接光照和反射折射光照，追踪深度最大为 8 层。之后对结果进行**重采样**，如果一个像素点与周围像素点的 hash 值不同，则说明其有不同的碰撞路径，有可能出现锯齿。将该像素点分为 9 份，重新发出方向不同的光线进行追踪，得到结果平均后作为其最终结果。

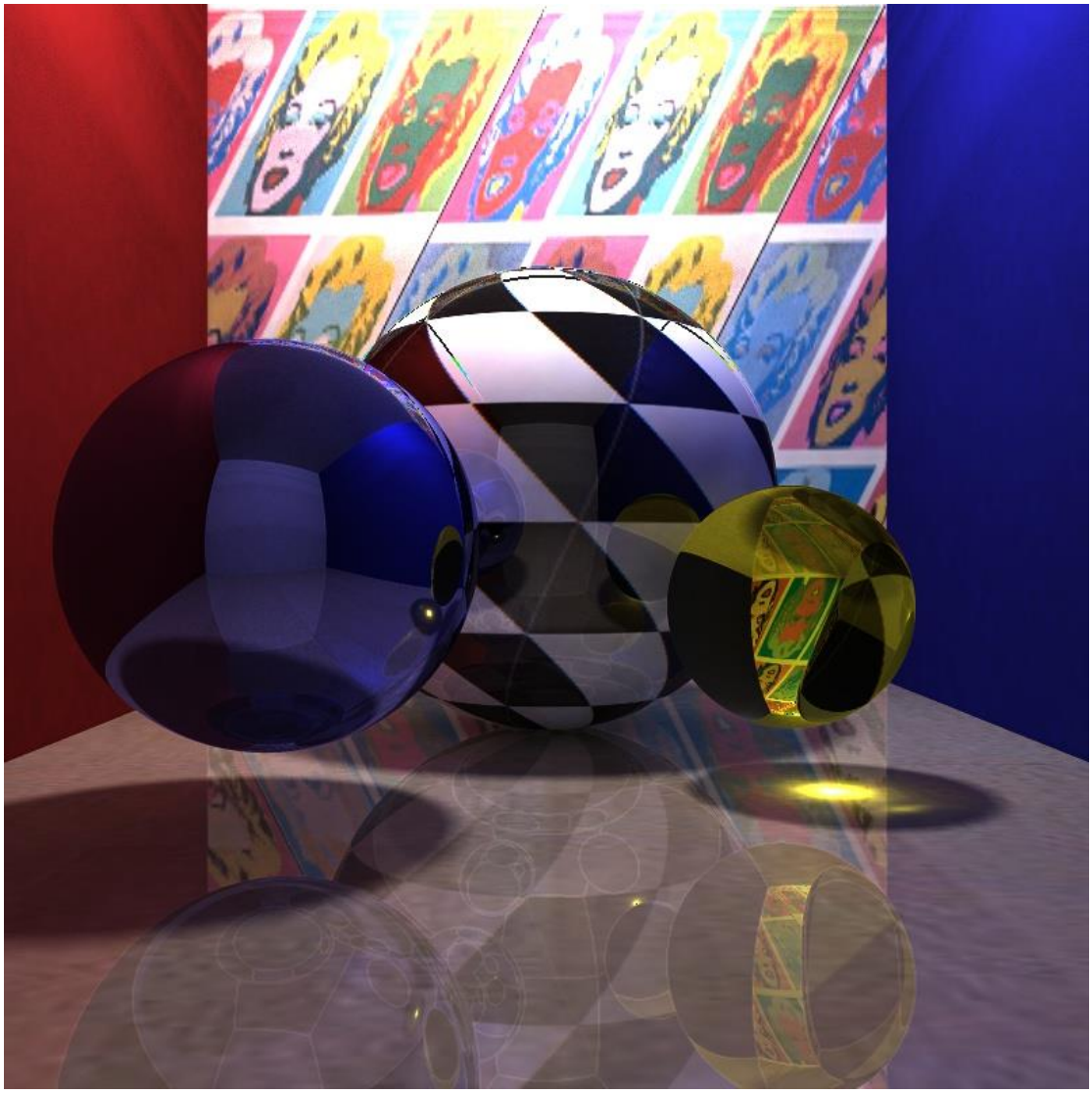
```
Color color = camera->getcolor(i, j) / 9;
for(int r = -1; r <= 1; r++){
    for(int c = -1; c <= 1; c++) {
        if(r == 0 && c == 0)
            continue;
        Ray ray_V = camera->getview(i + (double)r / 3, j + (double)
        c / 3);
        color += raytracing(ray_O, ray_V, 1, NULL) / 9;
    }
}
camera->setcolor(i, j, color);
```

具体见 rt.cpp 文件，函数名即为其功能。

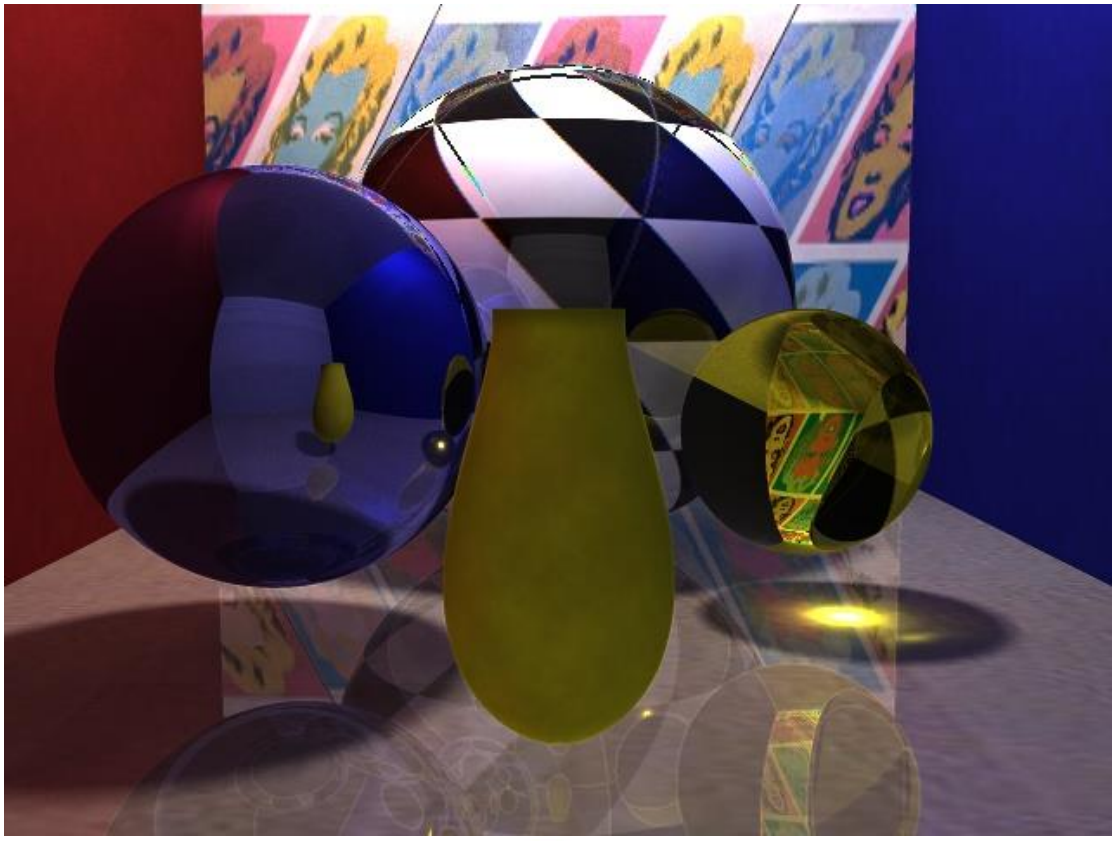
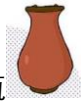
结果展示



-
- 第一次结果，由于光源较低，可以直接从图中看到，高光、焦散效果。但是后墙贴图选错了源图片，显示出错。光子量较低，可以在两侧看出明显的不均匀。



- 第二次结果，上移了光源位置，更换了背景贴图，同时增加了光子数，使间接光照较为均匀。

- 
- 第三次结果，为了展示一个希腊式花瓶  Bezier 曲面的结构，上移了视点。可以从正面（背光）和蓝球反射（面光）中看出在光照下的结果。

作业总结

这次大作业我由于前期时间分配不充分，没有按时完成。在补做过程中，我先采用光线跟踪算法完成，再在此基础上完成光子映射。初次实验使用直接将光子图可视化的算法，发现结果较差，有不少噪点。于是我参考了网上的教程，采用直接光照(漫反射表面)，间接光照，镜面和折射光照这一模型构建光子映射算法模型。在 Bezier 曲线的计算过程中花了很多时间（3-4 天），  ，可以发现只有上半部分完成了求交，而且光子似乎被拘禁在图形内多次反射，因此亮度很高。始终没有发现三维曲面算法的错误在哪，最后没办法采用了曲线旋转的算法，没法构造更复杂的图形，但算是实现了 bezier 曲面。