

# 目录

<b>1 SPPM</b>	3
1.1 储存可视点并建树	3
1.2 光子追踪并渲染	3
1.3 kdtree 的实现	5
1.4 递归的实现	6
<b>2 参数曲面求交</b>	7
<b>3 包围盒求交加速</b>	11
3.1 Sphere 包围盒	11
3.2 Triangle 包围盒	12
3.3 Mesh 包围盒	12
3.4 Transform 包围盒	13
3.5 参数曲线曲面	13
3.6 平面	13
<b>4 景深</b>	15
<b>5 抗锯齿</b>	16
<b>6 贴图</b>	17
6.1 球面	18
6.2 平面	18
6.3 参数曲面	19
<b>7 运动模糊</b>	19
<b>8 OpenMP</b>	20
<b>9 高光</b>	21

# 1 SPPM

为节省内存，我将可视点储存在了 kdTree 里，而不储存光子图。同时在每轮 iteration 的过程中，我也通过实验确定了一个较好的半径衰减系数。

## 1.1 储存可视点并建树

在生成可视点的时候，我对每一个像素进行多次采样进行射线的发射，然后进行 RayTracing

```
1 for (int i = start_row; i < x; i++)
2 {
3     for (int j = start_col; j < y; j++)
4     {
5         for (int ww = 0; ww < 8; ww++)
6         {
7             double _i = i + (rand() * 1.0 / RAND_MAX - 0.5) * 1;
8             double _j = j + (rand() * 1.0 / RAND_MAX - 0.5) * 1;
9             // Ray camRay = camera->generateRay(Vector2f(i, j));
10            Ray camRay = camera->generateRay(Vector2f(_i, _j));
11            RayTracing(camRay, Vector3f(1, 1, 1), i, j, 0, current_e, 0);
12            samples_count[i * y + j]++;
13        }
14    }
15 }
```

在进行 RayTracing 的时候，如果当前射线遇到了反射面或折射面，则将目前的能量乘上对应材料的折射系数或反射系数，然后递归进行查找。如果遇到漫反射面，则进行可视点的储存。储存可视点的位置、法向、颜色、能量以及对应的像素位置。

## 1.2 光子追踪并渲染

对图片的每一个像素进行可视点的储存之后，进行 PhotonTracing。

对于图片中的每一个光源，根据光源的 brightness 值把总光子数进行均分，分别进行跟踪。

对于不同的光源形状，使用 PBRT 的方法发射随机方向的光子。在处理点光源的时候，我犯了一个错误。我之前是使用随机产生经纬度的方法进行光子的发射。但是在查阅 mathworld 的 Sphere Point Picking 的时候我发现这种方法发射光子并不是均匀的，因为立体角的计算如下：

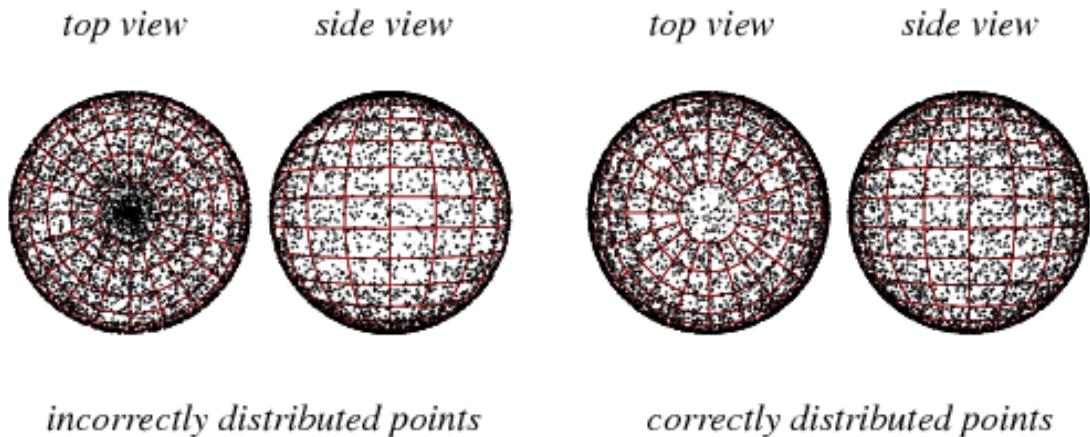
$$d\Omega = \sin\phi d\theta d\phi \quad (1)$$

所以  $\phi$  并不是均匀的，相反， $\cos\phi$  才是均匀的，所以我采取如下方法进行经纬度的生成：

$$\theta = 2\pi u \quad (2)$$

$$\phi = \cos^{-1}(2v - 1) \quad (3)$$

其中， $u$  和  $v$  为 0 和 1 之间的随机数，即可进行均匀的生成。具体二者的差别如下：



对于每一个随机的光子，进行追踪。追踪的过程中，我设置了一个最大追踪深度为 5。光子遇到反射面，进行能量的衰减，继续递归。遇到折射面，进行能量衰减，继续递归。遇到漫反射面，使用 kdtree 的 query 操作，寻找距离这个交点范围  $R$  内的全部可视点，然后对于每一个可视点储存的像素坐标，在图片对应的像素中进行颜色的渲染。计算像素的过程中，光子的能量随着递归次数的增

加，不断减小，可视点的坐标离光子击打点越远，能量越小，同时也要考虑到光源的颜色与可视点的颜色和可视点的能量，最终综合考虑几者因素进行颜色的渲染。

然后，随机生成一个漫反射的方向进行递归。递归的同时，光源的颜色随着材料的颜色进行变化。核心代码如下：

```
1 for (int li = 0; li < parser->getNumLights(); li++)
2 {
3     Light *light = parser->getLight(li);
4     brightness += light->brightness;
5 }
6 for (int li = 0; li < parser->getNumLights(); li++)
7 {
8     Light *light = parser->getLight(li);
9     int light_emit_count = photon_num * light->brightness / brightness;
10 #pragma omp parallel for num_threads(12)
11     for (int i = 0; i < light_emit_count; i++)
12     {
13         fprintf(stderr, "\rRendering %d of %d iteration %.2f% \%", iter +
14             1, total_round, 100 * double(i) / double(light_emit_count));
15         Ray ray = light->randomlyEmit();
16         photonTracing(ray, light->getColor(), 0, current_r,
17                         total_brightness / photon_num, 0, 0);
18     }
19 }
```

### 1.3 kd-tree 的实现

使用常规的 kd-tree，在三个维度进行顺序分割，kd-tree 节点储存可视点和包围盒的最大坐标与最小坐标。

在询问的过程，如果询问点距离整个包围盒过远，直接剪枝，否则递归查找左孩子和右孩子。并将所有查找到的可视点 push 到对应的 vector 中返回给上层。

## 1.4 递归的实现

与基本的光线追踪类似，对于反射平面，直接获得反射光线。对于折射平面，从角度可以判断进行全反射还是折射。通过发射光线与法向的点积符号可以判断光线是入射物体还是射出物体，从而进行折射角的计算。对于漫反射平面，在球面上生成一个随机向量，并与法向叠加，就可以生成一个射向半球面的随机射线，进一步递归即可。

反射：

```
1 Vector3f I = ray.getDirection().normalized();
2 Vector3f N = hit.getNormal().normalized();
3 Vector3f reflect_direction = (I - 2.0 * (Vector3f::dot(I, N)) * N);
4 Vector3f origin = ray.pointAtParameter(hit.getT());
5 origin += 0.001 * N;
6 Ray reflect_ray(origin, reflect_direction.normalized());
7
```

折射：

```
1 Vector3f I = ray.getDirection().normalized();
2 Vector3f N = hit.getNormal().normalized();
3 Vector3f reflect_direction = (I - 2.0 * (Vector3f::dot(I, N)) * N);
4 Vector3f origin = ray.pointAtParameter(hit.getT());
5 origin += 0.001 * N;
6 Ray reflect_ray(origin, reflect_direction);
7 Vector3f nl = (Vector3f::dot(N, I) < 0) ? N : N * -1;
8 bool into = Vector3f::dot(nl, N) > 0;
9 float nc = 1, nt = 1.5, nnt = (into) ? (nc / nt) : (nt / nc), ddn =
    Vector3f::dot(I, nl), cos2t;
10 if ((cos2t = 1 - nnt * nnt * (1 - ddn * ddn)) < 0)
11 {
12     RayTracing(reflect_ray, rayC, xx, yy, depth + 1, lambda * hit.
        getMaterial()->refr, dist + hit.getT());
13 }
14 else
15 {
16     Vector3f t;
17     t = nnt * I + (nnt * Vector3f::dot(-1 * I, nl) - sqrt(cos2t)) * nl;
18     t = t.normalized();
```

```

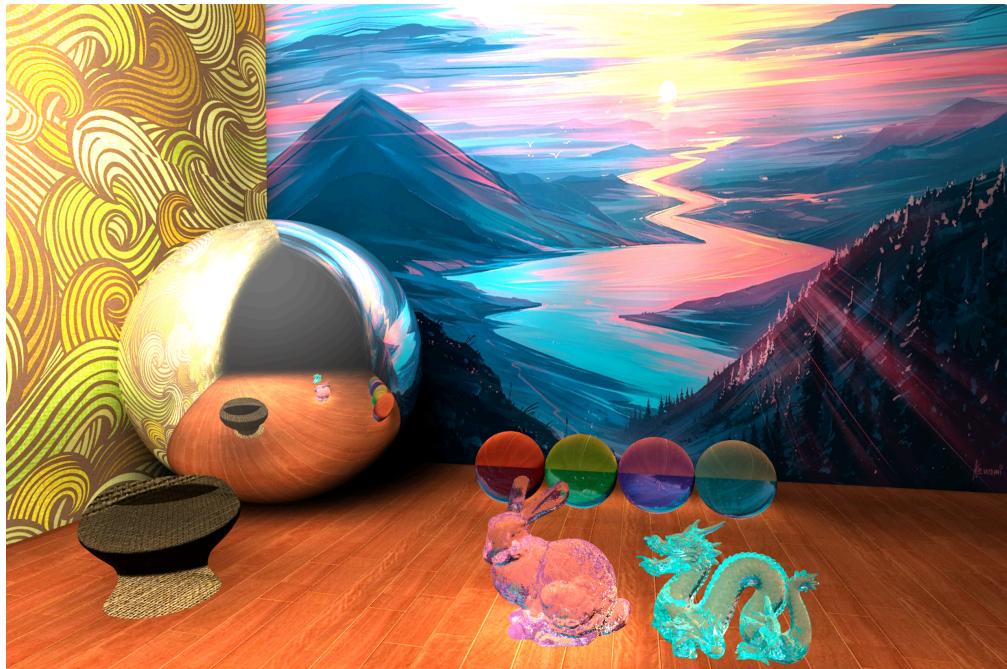
19     origin = ray.pointAtParameter(hit.getT());
20     origin += 0.01 * t;
21     Ray refract_ray(origin, t);
22     RayTracing(refract_ray, rayC, xx, yy, depth + 1, lambda * hit.
23         getMaterial()->refr, dist + hit.getT());
24 }
```

漫反射:

```

1 Vector3f DD = (hit.getNormal() * Vector3f::randomVectorOnSphere()).
2     normalized();
3 double phi = acos(rand() * 1.0 / RAND_MAX);
4 p = ray.pointAtParameter(hit.getT()) + hit.getNormal() * 0.0001;
5 Ray diffuse_ray(p, DD * sin(phi) + hit.getNormal() * cos(phi));
```

SPPM 的渲染图如下:



## 2 参数曲面求交

采用牛顿迭代法进行参数曲面的求交，具体的推导过程如下：

射线的方程为：

$$Ray(t) = Origin + t * Direction \quad (4)$$

曲面的方程为：

$$S(u, \theta) = Position + \begin{pmatrix} \cos\theta P_x(u) \\ P_y(u) \\ \sin\theta P_x(u) \end{pmatrix} \quad (5)$$

想要解的方程为：

$$F(t, u, \theta) = Ray(t) - S(u, \theta) = 0 \quad (6)$$

记这个解为  $x$ , 则有：

$$x = \begin{pmatrix} t \\ u \\ \theta \end{pmatrix} \quad (7)$$

根据牛顿迭代公式, 可以得到求解  $x$  时需要进行的迭代公式：

$$x_{n+1} = x_n - [F'(x_n)]^{-1} F(x_n) \quad (8)$$

其中,  $F'(x_n)$  为 Jacobi 矩阵:

$$F'(x_n) = \begin{pmatrix} \frac{\partial F_1}{\partial t} & \frac{\partial F_1}{\partial u} & \frac{\partial F_1}{\partial \theta} \\ \frac{\partial F_2}{\partial t} & \frac{\partial F_2}{\partial u} & \frac{\partial F_2}{\partial \theta} \\ \frac{\partial F_3}{\partial t} & \frac{\partial F_3}{\partial u} & \frac{\partial F_3}{\partial \theta} \end{pmatrix} \quad (9)$$

其中,  $P(u)$  为样条曲线的函数:

$$P(u) = \sum P_i B_{i,k}(u) \quad (10)$$

代入上式可以得到：

$$F'(x_n) = \begin{pmatrix} D_x & -\cos\theta P'_x(u) & \sin\theta P_x(u) \\ D_y & -P'_y(u) & 0 \\ D_z & -\sin\theta P'_x(u) & -\cos\theta P_x(u) \end{pmatrix} \quad (11)$$

其中， $B_{i,k}(u)$  为基函数。

对于参数曲面法向的算法，我对 S 进行偏导，并将得到的两个向量进行叉积，即可得到曲面法向向量：

$$\frac{\partial S}{\partial u} = \begin{pmatrix} \cos\theta P'_x(u) \\ P'_y(u) \\ \sin\theta P'_x(u) \end{pmatrix} \quad (12)$$

$$\frac{\partial S}{\partial \theta} = \begin{pmatrix} -\sin\theta P_x(u) \\ 0 \\ \cos\theta P_x(u) \end{pmatrix} \quad (13)$$

$$Normal = \frac{\partial S}{\partial \theta} \times \frac{\partial S}{\partial u} \quad (14)$$

注意，以上公式 x,y,z 坐标的顺序与参数曲面生成的旋转轴有关。具体的代码如下：

```

1 Vector3f x(t, u, theta);
2 double lr = 0.7;
3 // 牛顿迭代
4 for (int kk = 0; kk < 20; kk++)
5 {
6     t = x.x();
7     u = x.y();
8     theta = x.z();
9     Vector3f f = F(r, t, u, theta);
10    Matrix3f df = dF(r, t, u, theta);

```

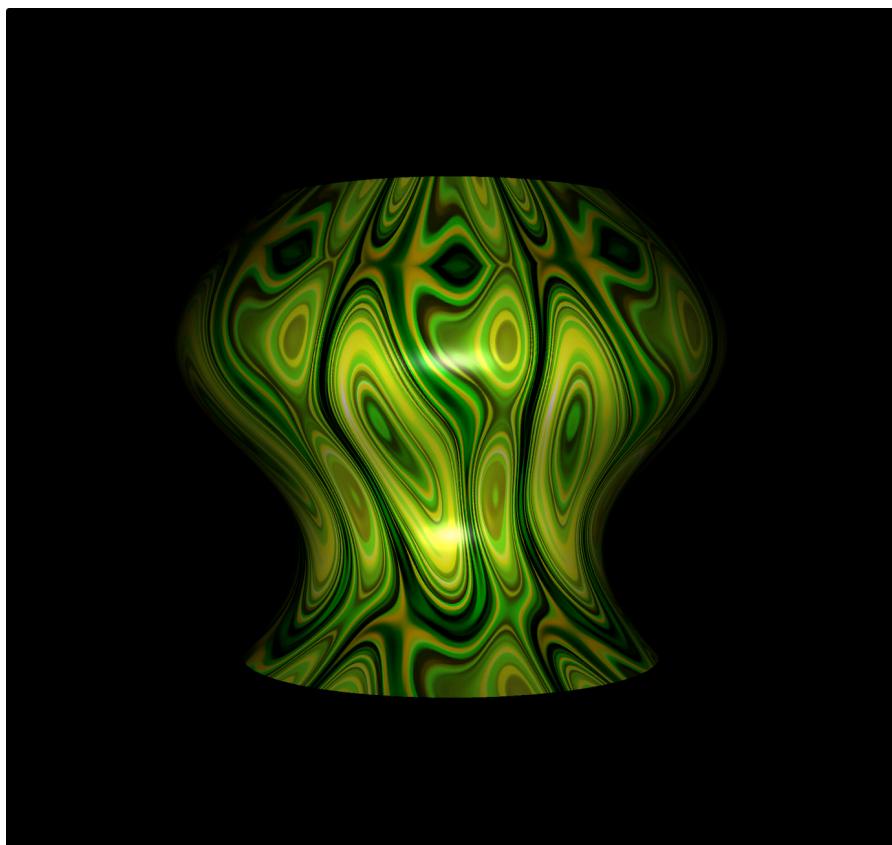
```

11     x = x - (df.inverse() * f);
12 }
13 // 更新 hit 的 t 和 normal, 以及贴图使用到的 u 和 v
14 h.u = x.y();
15 h.v = x.z();
16 h.t = x.x();
17 Vector3f v1(cos(x.z()) * pCurve->dpX(x.y()), pCurve->dpY(x.y()), sin(
    x.z()) * pCurve->dpX(x.y())));
18 Vector3f v2(-sin(x.z()) * pCurve->pX(x.y()), 0, cos(x.z()) * pCurve->
    pX(x.y())));
19 h.normal = Vector3f::cross(v1, v2).normalized();
20 if (Vector3f::dot(r.getDirection(), h.normal) > 0)
21     h.normal = -h.normal;
22

```

我进行了 20 次迭代，发现效果已经非常不错。

渲染的结果如下：



### 3 包围盒求交加速

我使用 bvh 进行包围盒的设计。

首先，我定义了一个包围盒类，里面储存了本盒子最大的坐标和最小的坐标。对于盒子的 intersect 函数，则采用老师上课讲的长方体求交方法。

然后，我定义了 bvh\_node 类，即 bvh 的一个节点，里面设置了递归建树的功能，求交的功能。递归建树随机划分维度，通过对比不同包围盒的最小节点的 x 坐标大小进行排序，并将中位数作为当前节点的保存值。求交时，直接递归左孩子右孩子求交即可。在一棵 bvh 树中，我将所有面片和物体储存在了叶结点里。

对于 Group 类和 Mesh 类，我在初始化 Object 的时候都进行了递归建树操作。对于无限大的平面，由于不存在包围盒，所以我没有对其进行包围。

我进行了一个测试，在 sppm 使用 1000 个光子和 2 次迭代的条件下，渲染 1536\*1024 的图片，里面放置一个材料为折射面的 100k 龙，不对 mesh 进行递归建树渲染时间达到了两分钟，而对 mesh 递归建树渲染的时间为前者的 20 分之一，可见对小面片进行包围也是非常重要的。

而 kd-tree 的逻辑和 bvh 基本完全相同。

#### 3.1 Sphere 包围盒

使用中心坐标减去半径和加上半径作为包围盒的最小坐标和最大坐标。

```
1 virtual bool bounding_box(double time0, double time1, aabb&
2     output_box) const {
3     aabb box0(center(time0) - Vector3f(radius, radius, radius), center(
4         time0)+Vector3f(radius, radius, radius));
5     aabb box1(center(time1) - Vector3f(radius, radius, radius), center(
6         time1)+Vector3f(radius, radius, radius));
7     output_box = surrounding_box(box0, box1);
8     return true;
9 }
```

## 3.2 Triangle 包围盒

直接将最大坐标和最小坐标作为包围盒的最大坐标和最小坐标。

```
1 virtual bool bounding_box(double time0, double time1, aabb&
2     output_box) const {
3     float minx = min(vertices[0].x(), min(vertices[1].x(), vertices[2].
4         x()));
5     float miny = min(vertices[0].y(), min(vertices[1].y(), vertices[2].
6         y()));
7     float minz = min(vertices[0].z(), min(vertices[1].z(), vertices[2].
8         z()));
9     float maxx = max(vertices[0].x(), max(vertices[1].x(), vertices[2].
10        x()));
11    float maxy = max(vertices[0].y(), max(vertices[1].y(), vertices[2].
12        y()));
13    float maxz = max(vertices[0].z(), max(vertices[1].z(), vertices[2].
14        z()));
15    output_box = aabb(Vector3f(minx, miny, minz), Vector3f(maxx, maxy,
16        maxz));
17    return true;
18 }
```

## 3.3 Mesh 包围盒

获得所有三角形的最小坐标的最小值和所有三角形的最大坐标的最大值作为包围盒的最大坐标和最小坐标。

```
1 virtual bool bounding_box(double time0, double time1, aabb &
2     output_box) const
3 {
4     float minx = 1e7, miny = 1e7, minz = 1e7, maxx = -10000, maxy =
5         -10000, maxz = -10000;
6     for (auto p : v)
7     {
8         minx = min(minx, p.x());
9         miny = min(miny, p.y());
10        minz = min(minz, p.z());
```

```

9     maxx = max(maxx, p.x());
10    maxy = max(maxy, p.y());
11    maxz = max(maxz, p.z());
12 }
13 output_box = aabb(Vector3f(minx, miny, minz), Vector3f(maxx, maxy,
14                         maxz));
14 return true;
15 }
```

### 3.4 Transform 包围盒

这个比较特殊。我先获得 Transform 的原始 Object 的 bounding box，然后根据 Transform 的平移坐标 translate 和缩放系数 scale 将对应包围盒的点进行三维坐标点的转换，得到的转换结果就是真正的包围盒了。

```

1 virtual bool bounding_box(double time0, double time1, aabb &
2   output_box) const
3 {
4   aabb temp_box;
5   o->bounding_box(time0, time1, temp_box);
6   // 输出信息
7   cout << "transform " << (translate+temp_box.min)*scale << " " << (
8     translate+temp_box.max)*scale << endl;
9   output_box = aabb((translate+temp_box.min)*scale, (translate+
10    temp_box.max)*scale);
11  return true;
12 }
```

### 3.5 参数曲线曲面

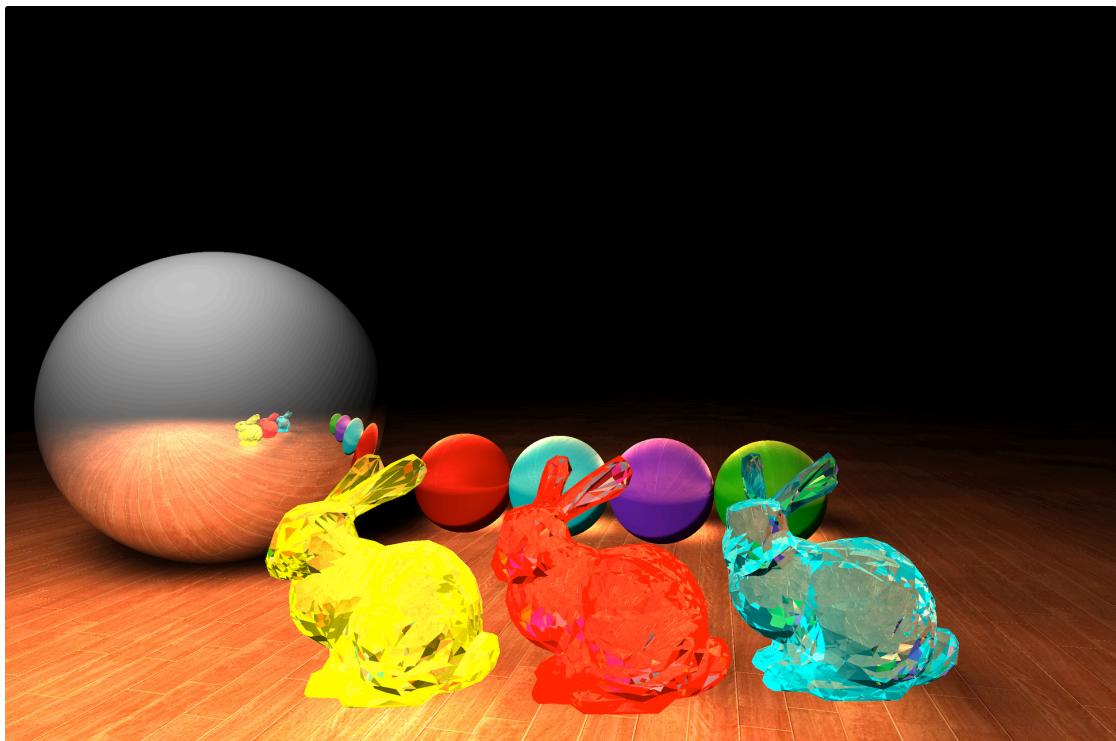
理想曲线，永远无法相交，直接返回无包围盒。参数曲面，返回所有三角面片的最小坐标和最大坐标，并进行稍微扩大，进行估计。

### 3.6 平面

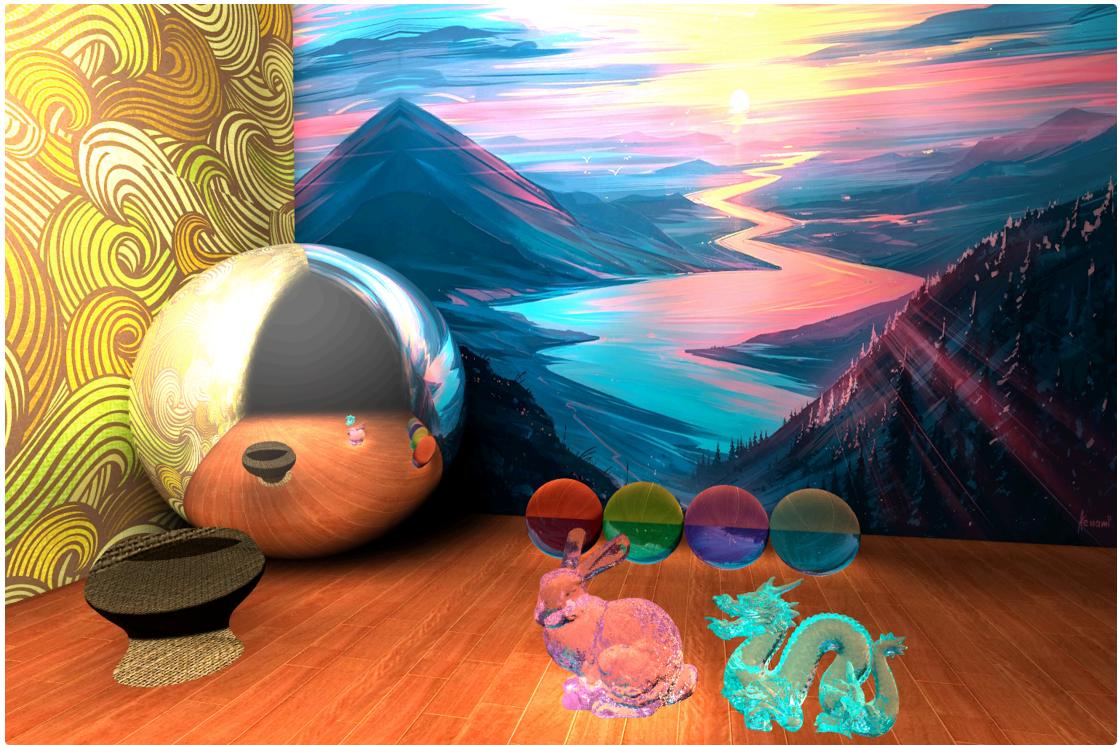
无限大平面，不存在包围盒。

在处理相交的过程中，对于 Group 内的每一个 Object，如果存在包围盒，进入到包围盒的求交方法。如果不存在包围盒，直接使用原始的暴力方法求交（在我的渲染物体中，只有无限大物体没有包围盒，所以渲染效率几乎无影响）。

使用包围盒渲染的兔子如下：



使用包围盒渲染的龙和兔子如下：



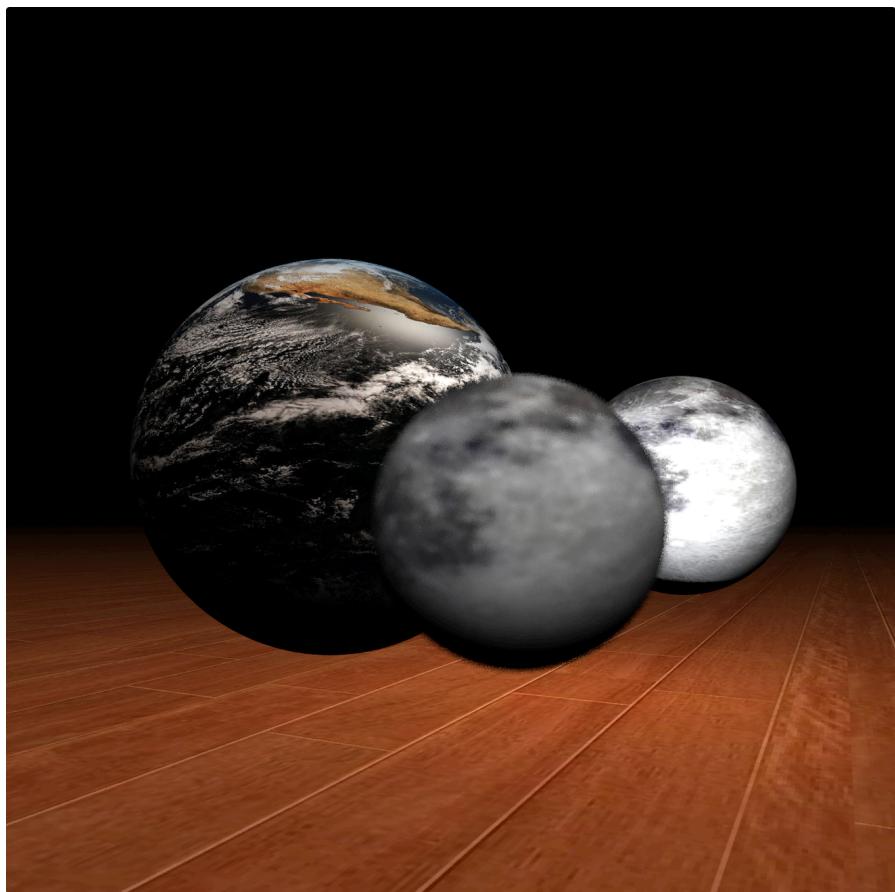
## 4 景深

在 parse 相机的时候初始化一个焦平面的距离。根据景深的原理，场景中在焦平面上的点是清楚的，不在焦平面上的点是模糊的。所以只需要在相机生成射线的时候进行轻微的扰动，并保持光线一定射在焦平面上即可。

我的操作方法是在生成射线的时候生成一个较小的随机向量，加上相机的位置作为相机的新位置，然后将这个新位置指向焦平面对应位置的方向作为这条射线的方向向量，进行发射，即可实现景深效果。

```
1 Ray addDepth(const Ray &ray) override
2 {
3     Vector3f fpoint = ray.getOrigin() + fdepth * ray.getDirection();
4     return Ray(ray.getOrigin() + Vector3f::randomVectorOnSphere() *
5                fdepth, (fpoint - ray.getOrigin()).normalized());
6 }
```

渲染图如下：

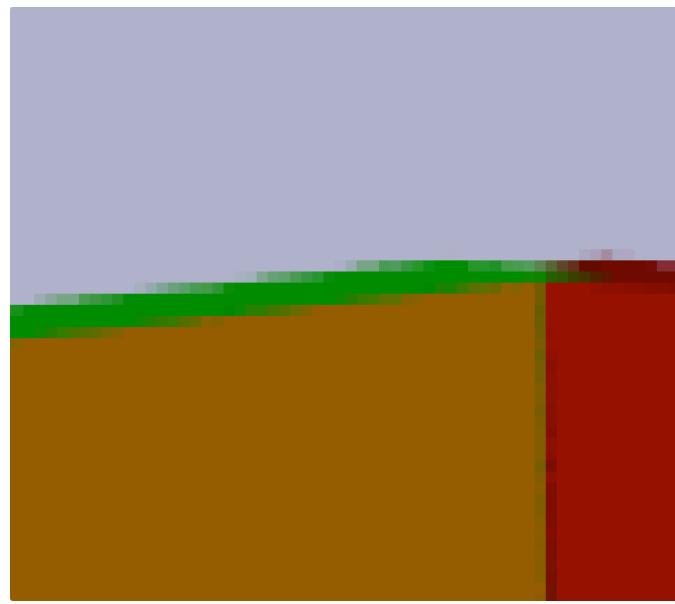


## 5 抗锯齿

抗锯齿实现的原理可以认为是在处理一个像素点的颜色时，对其和周边的位置进行多次采样，加入随机扰动，从而将这个像素的颜色稍微平均化。

由于 sppm 在实现的过程中，是对每一个像素进行多次采样，所以自然实现了抗锯齿效果。

Phone 模型抗锯齿效果：



sppm 抗锯齿效果：（两个平面均为贴图，中间分界线抗锯齿）



## 6 贴图

定义了一个 texture 类，并使用了 stb\_image 库进行图片的读取。在光线和物体求交的时候，在 Hit 中，同时也要存入这个击中位置在物体上对应的坐标。

考虑到图片的大小，我对于较大的贴图平面进行了延展复制操作，即在过大的 Object 上不同位置重复渲染一个纹理，达到较好的效果。

```

1 virtual Vector3f value(double u, double v, const Vector3f &p) const
2     override
3 {
4     //如果没有颜色，返回青色
5     if (data == nullptr)
6         return Vector3f(0., 1., 1.);
7
8     if (u<0) u=-u;
9     if (v<0) v=-v;
10
11    while (u>1.0) u-=1.0;
12    while (v>1.0) v-=1.0;
13    v = 1.0-v;
14
15    auto i = static_cast<int>(u * width);
16    auto j = static_cast<int>(v * height);
17    if (i >= width)
18        i = width - 1;
19    if (j >= height)
20        j = height - 1;
21
22    const auto color_scale = 1.0 / 255.0;
23    auto pixel = data + j * bytes_per_scanline + i * bytes_per_pixel;
24    return Vector3f(color_scale * pixel[0], color_scale * pixel[1],
25                    color_scale * pixel[2]);
26 }
```

## 6.1 球面

使用  $\phi$  和  $\theta$  作为坐标。同时，我也设置了能够在球面上旋转贴图的功能。

## 6.2 平面

使用相交点和原点的相对距离作为坐标，同时在 parse 平面的过程中，也加入了图片的放缩参数，实现不同大小贴图的适应。

### 6.3 参数曲面

如上文所示，参数曲面中的两个参数  $u$  和  $\theta$  可以作为图片内对应的坐标。  
具体实例如下：

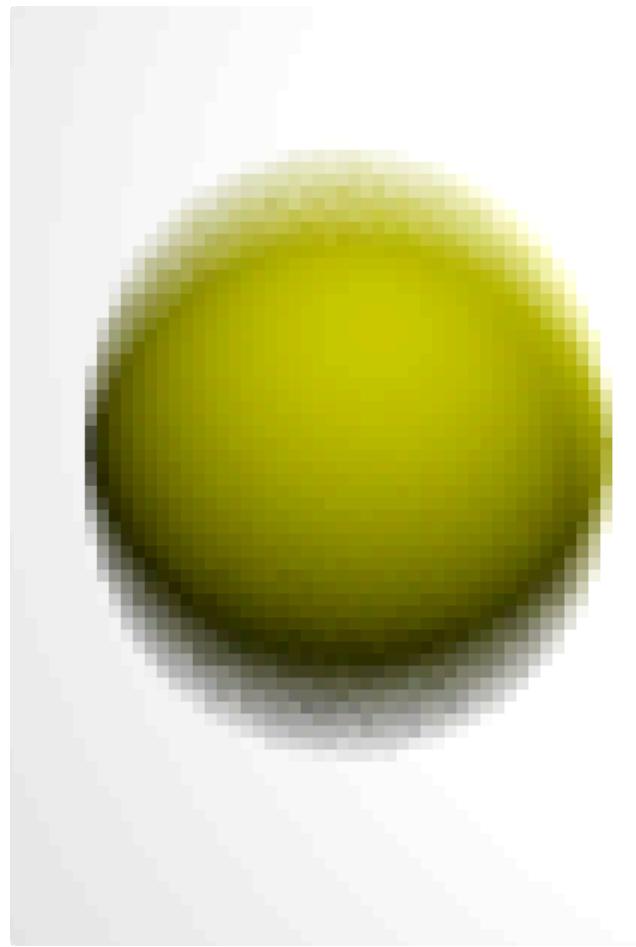


## 7 运动模糊

对于球的运动模糊，可以理解为一个球拥有两个中点坐标，球在一定的时间范围内在两个坐标间移动。其中这个时间可以在 parse 时进行人为指定。这样在相机的光线射向球时，对于边缘会存在模糊效果，而中间的位置则基本不存在模糊效果。其中球心坐标的获得方法如下：

```
1 Vector3f center(double time) const
2 {
3     return center0 + ((time - time0) / (time1 - time0)) * (center1 -
4         center0);
5 }
```

对于包围盒，只需要将球移动的全部空间放置在包围盒里即可。



## 8 OpenMP

由于 N 卡没给我的系统提供对应接口，我没有进行 GPU 加速，就只使用了 OpenMP 进行多线程加速。我在递归寻找可视点和光子映射的 for 循环使用了并行的方法。

在并行的过程中，我发现向 vector push 元素的时候会存在内存操作错误。我估计是因为有多个线程同时访问 vector 进行扩容的时候，如果操作系统不在原位置进行扩容，而是将现有的元素搬到内存的其他地方，对相同的内存块进行了多次释放导致的。将 vector 操作强制单独执行便可以防止这种结果。

## 9 高光

为了达到和 Phone 模型类似的高光效果，我尝试在 sppm 的基础上在对应的像素上叠加一个 Phone 模型，也实现了较好的高光。