



计算机图形学期末报告

袁小迪 2019012382

January 2022

1 得分项总述

- 算法选型：路径追踪（PT）
- 求交加速：层次包围盒（BVH）
- 参数曲面解析法求交
- 抗锯齿（超采样）
- 景深
- 软阴影
- 运动模糊

- 纹理贴图 (UV 纹理贴图、凹凸贴图)
- 复杂网格模型/场景 (网格读写、求交加速、法向插值)
- 体积光 (简单实现)

代码链接: <https://github.com/Rabbit-Hu/RayTracer>

效果图见 output 文件夹。

2 核心算法

本项目使用的光线追踪算法是路径追踪 (Path Tracing), 核心代码主要参考了smallpt。光线的颜色由两部分组成, 分别为镜面反射/折射颜色和漫反射颜色。

2.1 镜面反射/折射

使用 Fresnel 方程的 Schlick 近似¹计算反射光强和折射光强。在 Schlick 近似中, 反射系数为

$$R(\theta) = R_0 + (1 - R_0) * (1 - \cos \theta)^5, \quad (1)$$

其中

$$R_0 = \left(\frac{n_1 - n_2}{n_1 + n_2} \right)^2, \quad (2)$$

n_1, n_2 为两介质的折射率。

在递归求颜色的过程中, 当从摄像机出发的光线第一次接触物体时, 分别递归计算反射光和折射光的颜色; 第二次及此后的接触, 按照折射和反射光强的比例, 随机其中一种递归求解。

相关代码 (Scene.h):

```

1 Ray reflect = Ray(its.poc, (ray.d - normal * 2 * dot(normal,
2   ↵ ray.d)).normalized());
3 double eta = its.type == INT0 ? 1 / Ni : Ni; // sin(r)/sin(i)
4 double cos_i = dot(normal, ray.d); // Actually it's -cos(i)
5 double cos_r_square = 1 - eta * eta * (1 - cos_i * cos_i);
6 if (cos_r_square < EPS) {
7   return emit_color + get_radiance_with_coeff(reflect, depth + 1, Xi,
8     ↵ diffuse_color);
9 }
10 Ray refract = Ray(its.poc, (ray.d * eta - normal * (cos_i * eta +
11   ↵ sqrt(cos_r_square))).normalized());
12 double a = Ni - 1, b = Ni + 1;
13 double R0 = (a * a) / (b * b), c = 1 - (its.type == INT0 ? -cos_i : dot(refract.d,
14   ↵ its.normal));
15 double Re = R0 + (1 - R0) * c * c * c * c * c, Tr = 1 - Re, P = 0.25 + 0.5 * Re,
16   ↵ RP = Re / P, TP = Tr / (1 - P);

```

¹https://en.wikipedia.org/wiki/Schlick%27s_approximation

```

12 Vector3f to_add = specular_color, alpha = specular_color;
13 if (its.type != INT0) alpha = pow(absorb_color, its.t);
14 // if (depth < 1) to_add = radiance(reflect, depth + 1, Xi) * Re * to_add +
15 //   radiance(refract, depth + 1, Xi) * Tr * alpha;
16 if (depth < 1) {
17     to_add = get_radiance_with_coeff(reflect, depth + 1, Xi, Re * to_add)
18     + get_radiance_with_coeff(refract, depth + 1, Xi, Tr * alpha);
19 }
20 else if (erand48(Xi) < P) to_add = get_radiance_with_coeff(reflect, depth + 1, Xi,
21   to_add * RP);
22 else to_add = get_radiance_with_coeff(refract, depth + 1, Xi, TP * alpha);
23 color = color + to_add;

```

2.2 漫反射

漫反射方向通过在半球面上做余弦采样²得到。在半球面上，漫反射方向的概率密度与该方向与法线的夹角的余弦成正比，即 $p(\theta, \phi) \propto \cos \theta \sin \theta$ ($\sin \theta$ 来自球面坐标变换)，则

$$p(\theta, \phi) = \frac{\cos \theta \sin \theta}{\int_{\theta=0}^{\pi/2} \int_{\phi=0}^{2\pi} \cos \theta \sin \theta d\phi d\theta} = \frac{1}{\pi} \cos \theta \sin \theta, \quad (3)$$

关于 θ 的累计概率函数为

$$F(\theta) = \int_{t=0}^{\theta} \int_{\phi=0}^{2\pi} p(t, \phi) = \sin^2 \theta + C, \quad (4)$$

代入 $F(0) = 0$ 得 $F(\theta) = \sin^2 \theta$ 。则从 $[0, 1]$ 中均匀采样 ξ ，令 $\theta = \arcsin(\sqrt{\xi})$ 即可采样 θ 。

假设法线为 $(0, 0, 1)^\top$ (其余情形可以通过简单旋转变换转化为这一情形)，则可在 $[0, 1)$ 中均匀采样 ξ_1, ξ_2 ，令

$$\phi = 2\pi\xi_2, \theta = \arcsin(\sqrt{\xi_1}), \quad (5)$$

$$x = \sin \theta \cos \phi = \cos(2\pi\xi_2)\sqrt{\xi_1}, y = \sin \theta \sin \phi = \sin(2\pi\xi_2)\sqrt{\xi_1}, z = \cos \theta = \sqrt{1 - \xi_1}, \quad (6)$$

则 $(x, y, z)^\top$ 为漫反射方向的采样结果。

相关代码 (Scene.h):

```

1 Vector3f random_hemi_ray_cos(const Vector3f &normal, unsigned short *Xi) {
2     // generate random vector obeying cosine distribution
3     Vector3f xx;
4     if (fabs(normal.x) > EPS) xx = Vector3f(normal.y, -normal.x, 0).normalized();
5     else xx = Vector3f(0, -normal.z, normal.y).normalized();
6     Vector3f yy = cross(normal, xx);
7     double theta = 2 * PI * erand48(Xi);
8     double r = erand48(Xi);
9     double sr = sqrt(r);

```

²https://cg.informatik.uni-freiburg.de/course_notes/graphics2_08_renderingEquation.pdf

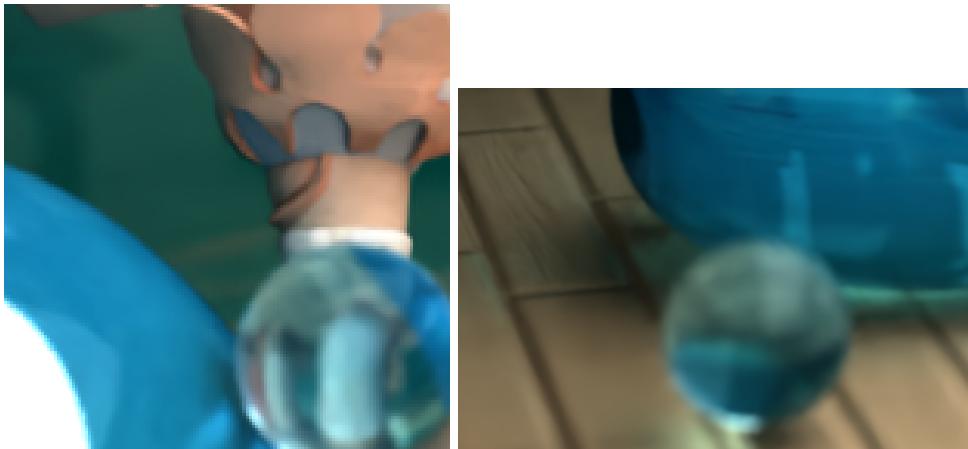


图 1: 路径追踪算法可以实现较为真实的渗色和软阴影效果。左图中，人物模型腿部的白色部分反射出来自左下方蓝色物体的蓝色；右图中，蓝色物体下方有软阴影。

```

10     return (xx * sin(theta) * sr + yy * cos(theta) * sr + normal * sqrt(1 -
    ↵   r)).normalized();
11 }
```

3 物体求交

实现的物体有：球体（Sphere）、平面（Plane）、矩形（Rectangle）、圆柱（Cylinder）、贝塞尔曲线的旋转体（Bezier）、三角形（Triangle）、网格（Mesh）。

所有物体继承 Object 基类，并重载 intersect() 函数。

球体、平面、矩形、三角形都是在前面的小作业中出现过的物体，在大作业中重新实现了相应的算法。另外，为了方便拓展功能，我重新规定了输入输出文件的格式，并实现了相应的输入输出函数。下文着重介绍贝塞尔曲线参数求交以及网格求交的加速。

3.1 贝塞尔曲线的旋转体（Bezier.h）

设曲线绕旋转轴的旋转角为 $2\pi u$ ，贝塞尔曲线的参数是 v ，则点对 $(u, v)(u \in [0, 1], v \in [0, 1])$ 能唯一确定旋转曲面上的一点。把射线与曲面求交转化为求 (t, u, v) 使得 $ray(t) = bezier(u, v)$ ，其中 $ray(t) \in R^3$ 表示射线上距离射线原点 t 的位置， $bezier(u, v)$ 表示贝塞尔曲线上参数为 (u, v) 的位置。使用牛顿迭代求解该方程。

因为方程往往有多个解，所以需要设定合理的初值、并且重复多次，从求出的解中选取首先求出射线与该旋转曲面的“包围圆柱”的交点（因为是旋转，所以用包围圆柱而不是包围盒，更为紧密且实现方便），求出对应的旋转角作为 u 的初值，到射线原点的距离作为 t 的初值，在 $[0, 1]$ 中均匀采样作为 v 的初值。

贝塞尔曲线代码（“getAabb()”函数实际为求包围圆柱的函数）：

```

1 class BezierCurve2D {
2 public:
```



图 2: 两个“史莱姆”均为贝塞尔曲线旋转体。(右图红色史莱姆有运动模糊效果。)

```
3     std::vector<Vector2f> control;
4
5     double y_min = INF_D, y_max = -INF_D, x_max = 0; // 2D AABB
6     BezierCurve2D(){}
7
8     BezierCurve2D(int size): control(size){}
9     BezierCurve2D(const std::vector<Vector2f> &_control): control(_control){}
10
11    int size() const {
12        return control.size();
13    }
14
15    BezierCurve2D operator*(double b) const {
16        BezierCurve2D ret(control);
17        for(int i = 0; i < ret.control.size(); i++)
18            ret.control[i] = ret.control[i] * b;
19        ret.y_min = y_min * b, ret.y_max = y_max * b, ret.x_max = x_max * b;
20        return ret;
21    }
22
23    BezierCurve2D operator+(const BezierCurve2D &b) const {
24        BezierCurve2D ret(control);
```

```

23     assert(control.size() == b.control.size());
24
25     for(int i = 0; i < ret.control.size(); i++)
26         ret.control[i] = ret.control[i] + b.control[i];
27
28     ret.y_min = y_min + b.y_min, ret.y_max = y_max + b.y_max, ret.x_max =
29         ↳ x_max + b.x_max;
30
31     return ret;
32 }
33
34 void getAabb() {
35     int size = control.size();
36
37     y_min = fmin(control[0].y, control[size-1].y);
38     y_max = fmax(control[0].y, control[size-1].y);
39
40     x_max = fmax(control[0].x, control[size-1].x);
41
42     if (size <= 2) return;
43
44     // find all the zero points of derivative curve
45     BezierCurve2D reduced(size - 1);
46
47     for (int i = 0; i < size - 1; i++)
48         reduced.control[i] = (control[i + 1] - control[i]) * (size - 1);
49
50     // x_max
51
52     for (int q = 0; q < NEWTON_ATTEMPT; q++) {
53         double t = drand48();
54
55         for (int i = 0; i < NEWTON_ITER && t >= 0 && t <= 1; i++) {
56             double f = reduced.eval(t).x;
57
58             if (fabs(f) < NEWTON_DELTA) {
59                 x_max = fmax(x_max, this->eval(t).x);
60
61                 break;
62             }
63
64             double d = reduced.deriv(t).x;
65
66             t -= f / d;
67         }
68     }
69
70     // y_min and y_max
71
72     for (int q = 0; q < NEWTON_ATTEMPT; q++) {
73         double t = drand48();
74
75         for (int i = 0; i < NEWTON_ITER && t >= 0 && t <= 1; i++) {
76             double f = reduced.eval(t).y;
77
78             if (fabs(f) < NEWTON_DELTA) {
79                 double y_new = this->eval(t).y;
80
81             }
82         }
83     }
84
85     return;
86 }

```

```

62             y_max = fmax(y_max, y_new);
63             y_min = fmin(y_min, y_new);
64             break;
65         }
66         double d = reduced.deriv(t).y;
67         t -= f / d;
68     }
69 }
70 }
71
72 Vector2f eval(double t) const {
73     if (control.size() == 1)
74         return control[0];
75     int size = control.size();
76     BezierCurve2D reduced(size - 1);
77     for (int i = 0; i < size - 1; i++)
78         reduced.control[i] = control[i] * (1 - t) + control[i + 1] * t;
79     return reduced.eval(t);
80 }
81
82 Vector2f deriv(double t) const {
83     int size = control.size();
84     BezierCurve2D reduced(size - 1);
85     for (int i = 0; i < size - 1; i++)
86         reduced.control[i] = (control[i + 1] - control[i]) * (size - 1);
87     return reduced.eval(t);
88 }
89 };

```

贝塞尔曲线的旋转曲面求交函数代码:

```

1 Intersection intersect_at_time(const Ray &ray, double t, unsigned short *Xi) const
2     {
3         BezierCurve2D curve = start_curve.size() ? end_curve * (1 - t) + start_curve *
4             t : end_curve;
5
6         // test intersection with the bounding cylinder, radius = curve.x_max;
7         Cylinder cyl(pos, up, right, curve.x_max * scale, curve.y_min * scale,
8             curve.y_max * scale);
9         Intersection cyl_its = cyl.intersect(ray, Xi);
10        if (cyl_its.type == MISS) return Intersection();
11    }

```

```

8     Vector3f flat = cyl_its.poc - pos;
9     flat = flat - dot(flat, up) * up;
10    double cyl_its_angle = atan2(dot(cross(right, up), flat), dot(right, flat));
11
12    Intersection ret;
13    for (int q = 0; q < NEWTON_ATTEMPT; q++) {
14        double t = cyl_its.t * (1 + (erand48(Xi) - 0.5) * 0.2), u = cyl_its_angle
15        ↪ / (2*PI) + (erand48(Xi) - 0.5) * 0.2, v = erand48(Xi);
16        for (int i = 0; i < NEWTON_ITER; i++) {
17            v = fmod(v, 2);
18            if (v < 0) v += 2;
19            if (v > 1) v = 2 - v, u += 0.5;
20            u = fmod(u, 1); // 0 is equivalent to 2*PI
21            if (u < 0) u += 1;
22            Vector3f p = eval(u, v, curve), f = ray.o + ray.d * t - p, du =
23            ↪ deri_u(u, v, curve), dv = deri_v(u, v, curve);
24            if (t > EPS && f.norm() < NEWTON_DELTA) {
25                if (t < ret.t && curve.eval(v).x > 0) { // crossing the rotation
26                    ↪ axis is not allowed!
27                    ret.t = t, ret.u = u, ret.v = v, ret.poc = p;
28                    ret.normal = cross(du, dv).normalized();
29                    ret.type = dot(ray.d, ret.normal) > 0 ? OUTFROM : INTO;
30                }
31                break;
32            }
33        }
34        ret.material = material;
35    }
36}

```

3.1.1 运动模糊

设贝塞尔曲线的控制点随时间线性变化，在求交时随机采样“时间”，求该时间点光线与曲面的交点，即可实现动态模糊效果。

```

1  Intersection intersect(const Ray &ray, unsigned short *Xi) const override {
2      double t = exp(-1.0 * erand48(Xi) * 14 / blur); // log(1e-6) \approx 14

```

```
3     return intersect_at_time(ray, t, Xi);
4 }
```

3.2 网格 (Mesh.h)

Mesh.h 中实现了三角网格的读取、求交。

3.2.1 层次包围盒

测试用的“派蒙”有 13,798 个三角形，暴力求交比较缓慢；为了加速求交，使用了层次包围盒 (BVH) 维护三角形，用于加速。

对所有三角形建立一棵二叉树。对于每个节点，按照每个三角形的重心的位置递归划分左右子树：在 X、Y、Z 三个坐标轴中，选择所有三角形重心“跨度”最大的轴，重心的该轴坐标小于等于中位数的三角形划分到左子树，其余划分到右子树。当前节点维护的三角形个数小于等于 5 时，作为叶子节点，不再继续划分。

层次包围盒代码 (Mesh.h)：

```
1 class BVH {
2 public:
3     class BVHNode {
4     public:
5         int id = -1, n_triangles = 0;
6         Aabb aabb;
7         Triangle* triangles = nullptr;
8         BVHNode *child[2];
9
10        BVHNode(){
11            child[0] = child[1] = nullptr;
12        }
13        ~BVHNode(){
14            if (triangles != nullptr) delete[] triangles;
15        }
16
17        Intersection intersect(const Ray &ray, unsigned short *Xi) const {
18            if (aabb.intersect(ray, Xi).type == MISS) {
19                // std::cout << "MISS" << std::endl;
20                return Intersection();
21            }
22            Intersection ret = Intersection();
23            if (n_triangles) {
24                for (int i = 0; i < n_triangles; i++) {
25                    Intersection tmp = triangles[i].intersect(ray, Xi);
```

```

26         if (tmp.type != MISS && tmp.t < ret.t) ret = tmp;
27     }
28
29     return ret;
30 }
31
32 for (int i = 0; i < 2; i++) {
33     if (child[i] != nullptr) {
34         Intersection tmp = child[i]->intersect(ray, Xi);
35         if (tmp.type != MISS && tmp.t < ret.t) ret = tmp;
36     }
37
38     return ret;
39 }
40
41 BVHNode *root;
42
43 BVH() {}
44 BVH(Mesh *mesh) {
45     build(mesh);
46 }
47 ~BVH() {
48     if (root != nullptr) delete_treenode(root);
49 }
50
51 class center_less {
52 public:
53     int axis;
54     Vector3f *centers;
55     center_less(int _axis, Vector3f *_centers): axis(_axis), centers(_centers)
56     {}
57     bool operator()(int a, int b) const {
58         if (axis == 0) return centers[a].x < centers[b].x;
59         else if (axis == 1) return centers[a].y < centers[b].y;
60         else return centers[a].z < centers[b].z;
61     }
62 };
63
64 void build_node(BVHNode *u, Mesh *mesh, Vector3f *centers, int *ids, int l,
65 ← int r) {
66     std::vector<Mesh::TriangleIndex> &f = mesh->f;
67     std::vector<Vector3f> &v = mesh->v;

```

```

64     Aabb &aabb = u->aabb;
65     // compute bounding box
66     for (int i = l; i < r; i++) {
67         int id = ids[i]; // triangle id
68         for (int j = 0; j < 3; j++) {
69             Vector3f p = v[f[id][j][0]];
70             aabb.p0.x = std::min(aabb.p0.x, p.x), aabb.p0.y =
71                 ↳ std::min(aabb.p0.y, p.y), aabb.p0.z = std::min(aabb.p0.z,
72                 ↳ p.z);
73             aabb.p1.x = std::max(aabb.p1.x, p.x), aabb.p1.y =
74                 ↳ std::max(aabb.p1.y, p.y), aabb.p1.z = std::max(aabb.p1.z,
75                 ↳ p.z);
76         }
77     }
78     if (r - l <= 5) { // leaf if there are <= 5 triangles
79         u->n_triangles = r - l;
80         // num_tree_triangles += u->n_triangles;
81         u->triangles = new Triangle[u->n_triangles];
82         for (int i = 0; i < u->n_triangles; i++) {
83             TriangleIndex idx = f[ids[l + i]];
84             u->triangles[i] = Triangle(v[idx[0][0]], v[idx[1][0]],
85                 ↳ v[idx[2][0]],
86                     mesh->vt[idx[0][1]], mesh->vt[idx[1][1]],
87                     ↳ mesh->vt[idx[2][1]],
88                     mesh->vn[idx[0][2]], mesh->vn[idx[1][2]],
89                     ↳ mesh->vn[idx[2][2]], mesh->f_materials[ids[l +
90                     ↳ i]]);
91         }
92     }
93     else {
94         // find the axis with largest length
95         int axis;
96         double l0 = aabb.p1.x - aabb.p0.x, l1 = aabb.p1.y - aabb.p0.y, l2 =
97             ↳ aabb.p1.z - aabb.p0.z;
98         if (l1 > l0) {
99             if (l2 > l1) axis = 2;
100            else axis = 1;
101        }
102        else {
103            if (l2 > l0) axis = 2;

```

```

95         else axis = 0;
96     }
97     int mid = (l + r) / 2; // index in array "ids"
98     std::nth_element(ids + l, ids + mid, ids + r, center_less(axis,
99                      centers));
100    if (l < mid) {
101        u->child[0] = new BVHNode();
102        u->child[0]->id = 2 * u->id;
103        build_node(u->child[0], mesh, centers, ids, l, mid);
104    }
105    if (mid + 1 < r) {
106        u->child[1] = new BVHNode();
107        u->child[1]->id = 2 * u->id + 1;
108        build_node(u->child[1], mesh, centers, ids, mid, r);
109    }
110    // std::cout << "Node " << u->id << ":" [ " << l << ", " << r << ") AABB = (" 
111    // << aabb.p0 << ", " << aabb.p1 << ") axis = " << axis << std::endl;
112 }
113 void delete_treenode(BVHNode *u) {
114     for (int i = 0; i < 2; i++)
115         if (u->child[i] != nullptr)
116             delete_treenode(u->child[i]);
117     delete u;
118 }
119
120 void build(Mesh *mesh) {
121     std::vector<Mesh::TriangleIndex> &f = mesh->f;
122     std::vector<Vector3f> &v = mesh->v;
123     int num_faces = f.size();
124     Vector3f *centers = new Vector3f[num_faces];
125     int *ids = new int[num_faces];
126     for(int i = 0; i < num_faces; i++) {
127         TriangleIndex idx = f[i];
128         centers[i] = (v[idx[0][0]] + v[idx[1][0]] + v[idx[2][0]]) * (1.0/3);
129         ids[i] = i;
130     }
131     root = new BVHNode();
132     root->id = 1;

```

```

133     build_node(root, mesh, centers, ids, 0, num_faces);
134     delete[] ids;
135     delete[] centers;
136 }
137
138     Intersection intersect(const Ray &ray, unsigned short *Xi) const {
139         return root->intersect(ray, Xi);
140     }
141 };

```

3.2.2 法向插值

在.obj 文件中，可以存储每个顶点的法线方向，光线与三角形求交时，按照重心坐标插值即可获得平滑的视觉效果。

```

1 ret.normal = (alpha * vn[0] + beta * vn[1] + gamma * vn[2]).normalized(); //  
↪ smooth shading

```

4 贴图

4.1 纹理贴图

Object.h 文件中实现了 Material 类，可以使用 Canvas 类型的成员 map_Kd 读取 png 图片文件作为贴图。物体求交时，一并求出交点的 UV 坐标，用于纹理贴图。

相关代码 (Object.h):

```

1 Vector3f get_diffuse(double u = 0, double v = 0) const {
2     // if have texture map, Kd.x serves as coefficient
3     if (map_Kd != nullptr) return Kd * map_Kd->get_color_uv(u * repeat.x, v *
4         repeat.y);
5     return Kd;
}

```

4.2 凹凸贴图

项目中采用了两种凹凸贴图方式：

- 法向贴图：直接在贴图中存储表面的法向信息。
- 凹凸贴图：读取表面高度信息，差分以近似地求出法向。

相关代码 (Object.h):



图 3: 网格贴图



图 4: 贝塞尔曲线旋转面贴图 (“眼睛” 和下方的火焰纹样)

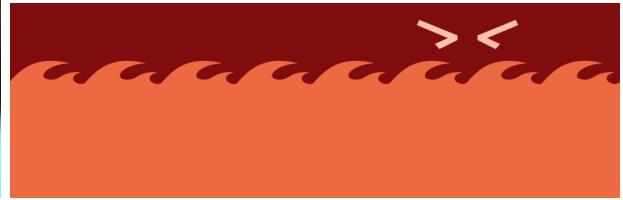


图 5: 平面贴图文件

图 6: 一些纹理贴图效果示例。

```
1 Vector3f get_normal(double u = 0, double v = 0) const {
2     if (map_normal != nullptr) return map_normal->get_color_uv(u * repeat.x, v *
3         repeat.y);
4     return Vector3f(0, 0, 1);
5 }
```

5 其他功能

5.1 景深

获取从照相机出发的光线时，在一个圆盘中均匀随机采样一个点作为出发点，模拟光圈；连接出发点和成像平面上所求像素的坐标，作为光线的方向。

相关代码 (Camera.h):

```
1 Vector2f random_in_unit_disk(unsigned short *Xi) {
2     Vector2f p;
3     do {
4         p = 2 * Vector2f(erand48(Xi), erand48(Xi)) - Vector2f(1, 1);
5     } while (dot(p, p) > 1);
6     return p;
7 }
8 Ray get_ray(double x, double y, unsigned short *Xi) { // x, y are pixel indices;
9     double dx = (2.0 * (x + 0.5) / w - 1), dy = -(2.0 * (y + 0.5) / h - 1); //
```

```

10     Vector2f offset = random_in_unit_disk(Xi) * aperture;
11     Vector3f ray_o = o + right * offset.x + up * offset.y;
12     // std::cout << "ray_o = " << ray_o << std::endl;
13     Vector3f ray_d = (d + right * (dx * w_scale - offset.x / focus_distance) + up
14     ↵ * (dy * h_scale - offset.y / focus_distance)).normalized();
15     return Ray(ray_o, ray_d);
}

```

5.2 抗锯齿

获取从照相机出发的光线时，将每个像素划分为 2×2 个“子像素”，将每个“子像素”的中心加上随机扰动的位置作为成像平面上光线经过的位置，将四个子像素颜色的平均值作为像素的颜色，减少锯齿。

相关代码 (Scene.h):

```

1 Vector3f color;
2 for (int s = 0; s < samp; s++) {
3     for (int sx = 0; sx < 2; sx++)
4         for (int sy = 0; sy < 2; sy++) {
5             double r1 = 2 * erand48(Xi), dx = r1 < 1 ? sqrt(r1) - 1 : 1 - sqrt(2 -
6             ↵ r1);
7             double r2 = 2 * erand48(Xi), dy = r2 < 1 ? sqrt(r2) - 1 : 1 - sqrt(2 -
8             ↵ r2);
9             Ray ray = camera.get_ray((sx + 0.5 + dx)/2 + x, (sy + 0.5 + dy)/2 + y,
10             ↵ Xi);
11             color = color + radiance(ray, 0, Xi) * (1. / samp) * .25;
12         }
13     (*canvas)[y - h1][x - w1] = color;
}

```

5.3 体积光

实现了比较简单的体积光效果，在全局加入了均匀的雾，让光在空间中每个位置有一定概率发生散射，散射方向均匀随机。雾的效果见图7，因为效果不够美观，没有整合到封面的最终效果图中。

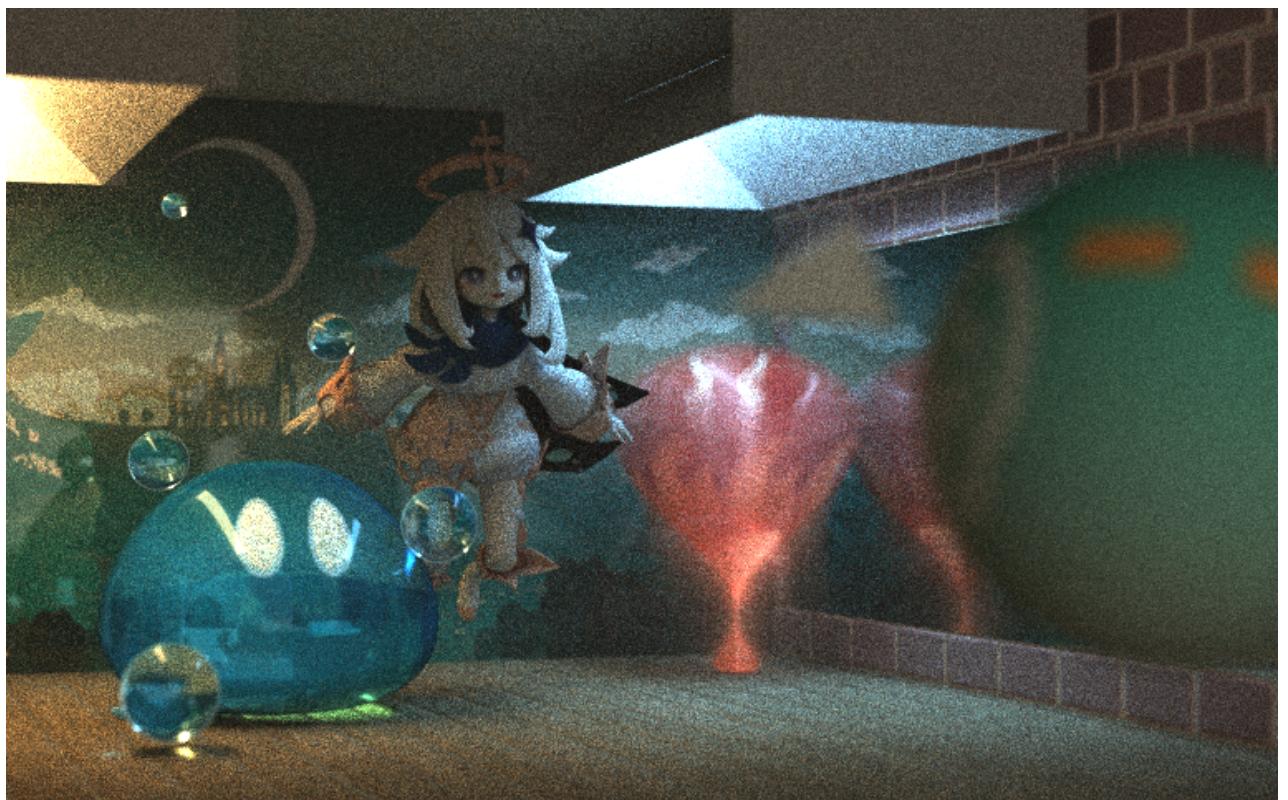


图 7: 全局雾效果。可以看到, 天花板上的“灯罩”中有明显的“雾”。