

# 高等计算机图形学 基于 Bézier 曲线的三维造形与渲染

计科 60 洪文浩 学号：2016011266

## 1 实验要求

使用参数方法设计三维形状，并利用光线跟踪算法渲染出一个有趣的三维场景。

## 2 实验结果

### 2.1 效果图

取 1750spp，以  $s = 15, n = 10$  在伺服器上跑出如下的图。



劇院休息室的幻影們

### 2.2 功能一览

以下是我实现的功能的一览表，具体细节会在相应章节说明。

**渲染引擎** 我实现了 Monte Carlo Path Tracing:

- 镜面反射
- Lambertian 漫反射
- 软阴影
- 折射 (比尔-朗伯特定律) (含焦散)

## 物件

- 球体
- 无限平面, 长方形
- 坐标轴包围盒 (长方体)
- 圆盘、圆环
- Bézier 旋转体曲面

## 额外效果

- 多线程加速
- 包围盒加速
- 抗锯齿 (超采样)
- 纹理贴图

## 3 实验内容

### 3.1 代码框架

src 文件夹到包含了这次实验用到的代码，分别列举如下。

- Bezier.h/.cpp Bézier 曲线和旋转体曲面类。
- Canvas.h/.cpp 画布类，支持 png 格式读入和输出，亦用作纹理。
- CMakeLists.txt CMake。
- Gauss.h/.cpp 线性方程求解器。
- LimitedObject.h/.cpp 二维矩形及圆盘类等。
- Object.h/.cpp 基类 Object 及球体。
- Plane.h/.cpp 平面类及 AABB 类。
- Stage.h/.cpp 舞台类。(因为舞台 Stage 比场景 Scene 更炫酷所以用了这个命名。)
- utils.h 基本类 Vec 及一些帮助函数，部分常数。

## 3.2 场景描述

场景的输入使用 `input.txt` 达成，物件的格式大致如下 (修改自 [3]):

```
1 # Disk (\)
2 HoledDisk
3   plane
4     dist 40
5     N 0.686229  0.537151  -0.490467
6     color .6 .99 .6
7     spec 1
8   end
9   axis 0 1 0
10  theta 0.5
11  pos 48 34 99.9975
12  rad_in 13
13  rad_out 16
14 end
```

程序会自动忽略物件之间以 `#` 起始的行，每个物件由物件的识别符起始，约于个属性名字和对应的值组成，并以 `end` 结束。

注意：程序只实现了基本的文本解析，因此如果物件不慎输入错误，有可能不会报错而发生未预期的错误。

## 3.3 渲染引擎

光线跟踪的算法我选择了 Monte Carlo Path Tracing，这个算法主要是通过 Monte Carlo 方法取样 (逆) 光路，估计积分，并得到像素的颜色。

我实现时主要参考了 [1] 给出的 99 行 C 实现。引擎基本功能分别支持了如下的特性，

**镜面反射** 直接使用反射定律： $\theta_{\text{in}} = \theta_{\text{out}}$  计算反射方向。

**Lambertian 漫反射** 在法线的上半球按  $\cos \theta$  的比例取样即可达到效果。

**软阴影** 软阴影由面光源和漫反射自动达成。

**折射** 折射按照折射定律，折射角与入射角的正弦及界面两侧的折射率成正比，并使用了俄罗斯轮盘来决定进行折射或反射。

**界质的光吸收** 出射时使用比尔-朗伯特定律模拟色光的指数衰减。

以上的效果实际代码如下：

```
1 Vec Stage::radiance(const Ray &ray, int depth, unsigned short *Xi) {
2   Intersection intersection = intersect(ray);
3
4   if (intersection.type == MISS)
5     return COLOR_BLACK;
6
7   const Object *hit = intersection.hit;
8
9   if (++depth > REFLECT_CNT) return hit->emit;
10
11 // point of contact
12 Vec poc = intersection.poc;
13
14 Vec normal_orig, normal = intersection.normal;
15 bool into = intersection.type == INTO;
```

```

17 if (!into)
18     normal_orig = normal * -1;
19 else
20     normal_orig = normal;

21 Vec color, hit_color = hit->get_color(poc);
22 color = hit->emit;

23 /* compute the shadow rays */
24
25 // refraction
26 if (hit->refr > EPS && hit->n > 0) {
27     Ray reflect = Ray(poc, (ray.dir - normal * 2 * normal.dot(ray.dir)).unit());
28     double nc = 1, nt = hit->n, nnt = into ? nc / nt : nt / nc, ddn = ray.dir.
29         dot(normal), cos2t;
30     if ((cos2t = 1 - nnt * nnt * (1 - ddn * ddn)) < 0)      // Total internal
31         reflection
32         return hit->emit + hit->color * (radiance(reflect, depth, Xi));
33     Vec tdir = (ray.dir * nnt - normal * (ddn * nnt + sqrt(cos2t))).unit();
34     Ray trans = Ray(poc, tdir);
35     double a = nt - nc, b = nt + nc, R0 = a * a / (b * b), c = 1 - (into ? -ddn
36 : tdir.dot(normal_orig));
37     double Re = R0 + (1 - R0) * c * c * c * c * c, Tr = 1 - Re, P = .25 + .5 *
38 Re, RP = Re / P, TP = Tr / (1 - P);
39     Vec to_add = hit_color;
40     Vec alpha = to_add;
41     if (!into)
42         alpha = hit->absorb.exp(intersection.t);
43     if (depth < 2)
44         to_add = (radiance(reflect, depth, Xi) * Re * to_add + radiance(trans,
45 depth, Xi) * Tr * alpha);
46     else if (erand48(Xi) < P)
47         to_add = to_add * radiance(reflect, depth, Xi) * RP;
48     else
49         to_add = radiance(trans, depth, Xi) * TP * alpha;
50     color = color + to_add;
51 }
52
53 // specular
54 if (hit->spec > EPS) {
55     Ray reflect = Ray(poc, (ray.dir - normal * 2 * normal.dot(ray.dir)).unit());
56     color = color + hit_color * radiance(reflect, depth, Xi) * hit->spec;
57 }
58
59 // diffuse
60 if (hit->diff > EPS) {
61     Ray shadow = Ray(poc, random_hemi_ray_cos(normal, Xi));
62     color = color + hit_color * radiance(shadow, depth, Xi) * hit->diff;
63 }

64

```

## 3.4 各种简单物件的求交

**球体** 球体求交只需要解一个一元二次方程，求得交点之后计算法线也很直接，此处不再赘述。

**平面求交** 平面求交可以通过计算光线方向在平面法线上的有效距离得到。

**坐标轴包围盒** 坐标轴包围盒的实现比一般包围盒简单，因此我在实验里只实现了坐标轴方向的长方体求交。实际做法是先计算光线与六个面的交点，然后取最接近的交点，并判断交点是否在包围盒上。

```
1 Intersection AABBBox::intersect(const Ray &ray, bool with_BB) const {
2     Intersection rst;
3     HitType type = contains(ray.src) ? OUTO : INTO;
4     for (int i = 0; i < 6; i++) {
5         Intersection tmp = faces[i].intersect(ray, false);
6         if (tmp.type != MISS && tmp.t > EPS && tmp.t < rst.t) {
7             Vec poc = tmp.poc;
8             if (contains(poc)) {
9                 rst = tmp;
10                rst.hit = this;
11                rst.type = type;
12            }
13        }
14    }
15    return rst;
}
```

**长方形** 长方形是平面的特殊情况，求交只需判断光线与平面的交点是否在长方形内便可。

**圆盘形** 圆盘形和长方形的实现思路一样，不同的是最后判断的是交点是否与圆心小于一定距离。(此形状最后并没有应用在场景中。)

**圆环形** 因为圆环很棒所以我实现了这个物件。圆环和圆盘一样，只是最后判断的时候还要判断不在内圆中。实现如下：

```
1 Intersection HoledDisk::intersect(const Ray &ray, bool with_BB) const {
2     Intersection tmp = plane.intersect(ray, false);
3     if (tmp.type == MISS || !contains(tmp.poc)) {
4         tmp.type = MISS;
5         return tmp;
6     }
7     tmp.hit = this;
8     return tmp;
}
10
11 bool HoledDisk::contains(const Vec &pt) const {
12     double dist = (pt - pos).norm();
13     return dist <= rad_out && dist >= rad_in;
}
14 }
```

## 3.5 参数曲面

曲面求交的设计主要是参考了 [3]。

为了实验方便起见，我采用了先计算 2 维的曲线以后绕轴旋转的方式生成曲面。

### 3.5.1 二维曲线

**曲面的计算** Bézier 曲线是由一个控制点集  $\{P_1, P_2, \dots, P_n\}$  定义的曲线,  $P(t)$  可以用 Bernstein 多项式显式定义。但显式计算的效率低且没有数值稳定性保证, 我使用了课上讲过的 de Casteljau 算法递归计算。

```
1 Vec BezierCurve::eval(const BezierCurve &curve, double t) {
2     // bezier curve evaluation using de Casteljau's algo
3     if (curve.c_points.size() == 1)
4         return curve.c_points[0];
5     int size = curve.c_points.size();
6     vector<Vec> new_points(size - 1);
7     for (int i = 0; i < size - 1; i++)
8         new_points[i] = curve.c_points[i] * (1 - t) + curve.c_points[i + 1] * t;
9     BezierCurve reduced = BezierCurve(new_points);
10    return eval(reduced, t);
11 }
```

此外, 我使用了静态函数尾递归的形式, 编译优化以后可以节省递归的开支。

**求导** 计算导数的公式和助教在幻灯片上给出的一致, 代码实现如下, 同样写成了静态函数使用尾递归。

```
1 Vec BezierCurve::deri(const BezierCurve &curve, double t) {
2     int size = curve.c_points.size();
3     vector<Vec> new_points(size - 1);
4     for (int i = 0; i < size - 1; i++)
5         new_points[i] = (curve.c_points[i + 1] - curve.c_points[i]) * (size - 1);
6     BezierCurve reduced = BezierCurve(new_points);
7     return eval(reduced, t);
8 }
```

### 3.5.2 旋转体求交

旋转体求交是比较昂贵的求交开支, 实验里我使用的方法是先取恰当的初值 (利用包围盒和启发式初值), 然后进行牛顿迭代。

曲线由  $P(u, v)$  代示, 其中  $u$  是二维曲线的参数  $t$ ,  $v = \frac{\theta}{2\pi}$  是该点对于对称轴右手方向的旋转度数 (逆时针)。

```
1 Vec BezierRotational::eval(double u, double v) const {
2     Vec point = BezierCurve::eval(curve, u);
3     point = point.rotate(axis, 2 * PI * v) + pos;
4     return point;
5 }
```

牛顿法需要用到偏导数如下: 使用助教幻灯片上的链式法则可以得到如下的实现方式。

```
1 Vec BezierRotational::du(double u, double v) const {
2     Vec deriv = BezierCurve::deri(curve, u);
3     deriv = deriv.rotate(axis, 2 * PI * v);
4     return deriv;
5 }
6 Vec BezierRotational::dv(Vec pt) const {
7     pt = pt - pos;
8     Vec d = axis.cross(pt).unit();
9     d = d * sqrt(pt.norm() * pt.norm() - (pt.dot(axis)) * pt.dot(axis)) * 2 * PI;
10 }
```

```

12 }      return d;
}

```

使用牛顿法得到交点后，法向量可以由以上的两个偏导数向量取叉积而得

**包围盒加速** 由于曲线控制点包围了曲线，计算旋转体的包围盒可以转化成计算控制点的凸包，这个过程比较麻烦，此处不再阐述。

实际实现上，我采取了用户手动输入包围盒的方式，通过调试找到比较合适的包围盒。另外，手动输入还可以起到裁剪曲线部分边缘的效果，使曲面视觉上产生一部分直截面。

计算包围盒以后，每次求交舞台上的物件时先求光线与包围盒的交点，如果求得的交点是包围盒的交点才进行对应物件的牛顿迭代法，否则返回之前求得最接近的交点，这样可以使程序效率提升不少。

以上的策略实现如下：(注意本实现要求曲面在 object list 最后)

```

1 Intersection Stage::intersect(const Ray &ray) {
2     // the following speed up require Bezier to be last
3     Intersection rst, last;
4     for (int i = 0; i < objects.size(); i++) {
5         Intersection tmp = objects[i]->intersect(ray, true);
6         if (tmp.type != MISS && tmp.t < rst.t) {
7             last = rst;
8             rst = tmp;
9         }
10    }
11    // try to speed up
12    if (rst.hit->get_type() == Object::BOX) {
13        auto box = (AABBox *)rst.hit;
14        if (box->enclosed != nullptr) {
15            rst = (box->enclosed)->intersect(ray, false);
16            if (rst.type == MISS)
17                return last;
18        }
19    }
20    sreturn rst;
21 }

```

**非线性方程求解** 我采取的牛顿法是直接对  $(t, u, v)$  迭代，其中  $t$  是光线起点与交点的距离， $u, v$  的定义同上。

算法进行如下的  $s$  次采样，取求得的合法解到最小的正值：首先选取猜测向量  $X_0$ ，然后对于每一个猜测向量  $X$ ，进行最多  $i$  次  $X \leftarrow X - J^{-1}\Delta f$  的迭代，其中  $J$  是剩余函数  $F(t, u, v) = P(u, v) - src - t \times dir$  的雅可比矩阵。实现时可以避免矩阵求逆，使用解方程  $-J(X' - X) = \Delta f$  来处理。

经过测试，取  $s = 15, i = 10$  配合有效的初值选取可以得到不错的结果。

进行以上的迭代时，我使用了如下的两点优化：

1. 当  $t$  为较大的负数或  $u, v$  偏离  $[0, 1]$  太多时， $y$  进行剪枝，返回无解。这是参考了一位学长的做法 [2]
2. 由于我渲染的曲面较为简单，如果找到的解个数大于 3，就为已经找到了最近的交点，不再进行采样。

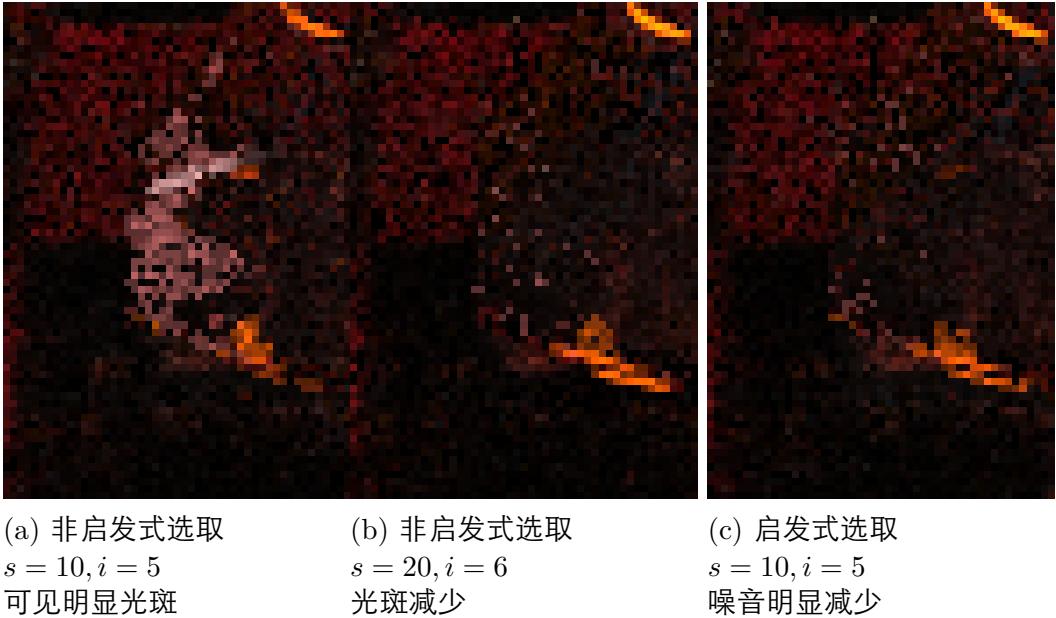
另外我还尝试了保留上次迭代  $F$  的值，如果  $F$  的值递增超过 3 次则直接返回无解的做法，但后没有采用。

由于最后调试的时间有限，我并没有来得及验证以上的做法是否能有效提速而不影响质量，但最终效果图还算可以。

**初值选取** 初值选取是牛顿法一个最重要的地方，因为牛顿法只有局部收敛性，对初值极其敏感。

首先可选  $t_0$  为光线与包围盒的距离。

对于另外两个面参数，我一开始选择的方式是  $u, v$  分别在  $[0, 1)$  内随机选初值。但实际效果一般，后来选取  $v$  为包围盒交点相对于对称轴右侧向量的角度附近为初值， $u$  均匀随机，提升了求交的效果，如下图。



实现代码如下

```

Intersection BezierRotational::intersect(const Ray &ray, bool with_BB) const {
    2     Intersection rst;

    4     Intersection with_box = b_box.intersect(ray, true);

    6     if (with_BB) return with_box;

    8     if (with_box.type == MISS) return rst;

    10    double ans_t = INF_D;
    11    Vec ans_du, ans_dv;

    12    // only if with_box is not miss will we use Newton's method
    13    Vec X;

    16    // t, u, v respectively
    17    Vec src = ray.src, dir = ray.dir;
    18    for (int q = 0; q < NEWTON_ATTEMPT; q++) {
    19        Vec du, dv, f;

    20        // first guess
    21        double v0 = compute_angle(with_box.poc) + (drand48() - .5) * .2;
    22        //double v0 = drand48();
    23        if (v0 > 1) v0 = v0 - 1;
    24        if (v0 < EPS) v0 = v0 + 1;
    25        X = Vec(with_box.t, drand48(), v0);
    26        int found_cnt = 0; // number of increasing distance
    27        double last_dist = INF_D;

    29        for (int i = 0; i < NEWTON_ITER; i++) {
    30            // one iteration of newton's method
    31            Vec p = eval(X[1], X[2]);
    32
```

```

f = src + dir * X[0] - p;
du = this->du(X[1], X[2]);
dv = this->dv(p);
// By -J(X - X0) = f
Vec d_x = Gauss::solve(dir * (-1), du, dv, f);
X = X + d_x;
double dist = f.inf_norm();
double t = X[0];
if (t < -.3 || X[1] > 1.3 || X[1] < -.3 || X[2] > 1.3 || X[1] < -.3)
    break;
if (t > EPS && X[1] > EPS && X[2] > EPS && X[1] < 1 - EPS && X[2] <
1 - EPS
    && ((last_dist = dist) < NEWTON_DELTA)) {
    found_cnt++;
    if (t < ans_t) {
        Vec p = eval(X[1], X[2]);
        du = this->du(X[1], X[2]);
        dv = this->dv(p);
        ans_t = t;
        ans_du = du;
        ans_dv = dv;
    }
    break;
}
if (found_cnt >= 3)
    break;
}
if (ans_t < INF_D) {
    rst.t = ans_t;
    rst.normal = ans_du.cross(ans_dv).unit();
    rst.type = dir.dot(rst.normal) > EPS ? INTO : OUTO;
    rst.normal = rst.normal * (rst.type == OUTO ? 1 : -1);
    rst.poc = src + dir * rst.t;
    rst.hit = this;
    return rst;
}
return rst;
}

```

**有趣的 bug** 曲面求交是极度容易出错的环节，我途中有一次求交有 Bug 但是效果很特别，请见附录A。

## 4 额外效果

**多线程加速** 因为 pt 的色彩计算像素之间是完全独立的，所以 CPU 并行化非常简单。我使用了 openmp 加速，加一段 macro 就可以。

**包围盒加速** 包围盒的使用已在3.5.2中阐述。

**抗锯齿** 我实现了简单的全局超采样，采样每个像素附近 4 个点的颜色取平均作为输出，取点的重要度采样按照 smallpt 用的分布函数。

**纹理贴图** 贴图的思路常简单直接，只要读入一个图像附在物件上，让有纹理的物件实现相应的坐标映射函数 `obj -> get_color(pt)` 就可以，因为矩形贴图最方便，我实现了矩形贴图。

```
1 Vec Rectangle::get_color(const Vec &pt) const {
2 // only axis-alien textures are supported currently
3 if (texture != nullptr) {
4     for (int d = 0; d < 3; d++) {
5         if (abs(abs(plane.normalized[d]) - 1) < EPS) {
6             if (scaled == 0) {
7                 double x, y;
8                 x = get_offset(pt, (d + 1) % 3);
9                 y = get_offset(pt, (d + 2) % 3);
10                return texture->get_color(y, x);
11            } else {
12                int x, y;
13                x = get_offset_unscaled(pt, (d + 1) % 3) * scaled;
14                y = get_offset_unscaled(pt, (d + 2) % 3) * scaled;
15                return texture->get_color(y, x);
16            }
17        } else {
18            cerr << "only x-y,y-z,z-x planes are currently supported" << endl;
19            exit(5);
20        }
21    } else
22        return color;
23 }
24
25 double Rectangle::get_offset(const Vec &pt, int d) const {
26     return (pt[d] - low[d]) / width[d];
27 }
28
29 double Rectangle::get_offset_unscaled(const Vec &pt, int d) const {
30     return (pt[d] - low[d]);
31 }
```

## 5 实验总结

注意以下大部分都是废话，但可说是一点心得体会，如果你也将要做这个大作业，也许会对你有所帮助。

当初选图形学就是为了这个作业而来的，因为我的室友刚好在贵系上过，看到他们的图有些特别好看的，觉得渲染完肯定非常有成就感，当然实际上做完这个大作业还是非常开心的。

### 5.1 实现的艰辛

实现光线跟踪真的是非常的耗时，尤其是经常会不小心用了 debug build 和忘记开多线程，而且经常会忘了可以只渲染一个区域来 debug，所以每次 debug 一地方都要花上好久的时间。看见图上的 bug 的时候一般来说都觉得很绝望，因为除了奶瓶 Bug(附录A) 以外基本上能看到的 bug 都奇丑无比甚至令自己怀疑人生。

### 5.2 实现的内容

我觉得因为自己这学期选课选太多，把这项作业推得太晚了，虽然 pt 在 11 月的时候已经写完，但是别的内容都是第十八周才加入，以致我觉得最后的图还有很多改善空间。

**改善算法** 现在的图虽然跑很久很久以后效果也许会比 sppm 好 (其实我在写这里的时候最终图还在跑，可以说是很极限了，希望能跑完 + 没有 bug, #UPDATE 现在跑完一半了好像没有明显的 bug, 太开心了)，但是我还是很想试试看写 sppm，不过我想以后机会也不多了。

**加入更 fancy 的特效？** 比如说旋转的动态模糊试着加在中间的环上？三角面片就算了，舞台太满了

**算法加速** 空间的数据结据最后没写上，因为舞台的物件不多也没三角面。本来是想写 ppm 的时候加上，但是最后没来得及写。

### 5.3 关于成品

最后的构图也是一个 Bug 引起的，当我发现奶瓶效果原来是 Bug 以后发现左边的区面变得没什么意义，而且为了效率我决定把它换成球体的镜子，但是一时之间我不小心把半径设得太大了，跑完之后我发现结果非常震撼，所以就采用了这个构图。

### 5.4 感想

总而言之我觉得通过这个作业我的代码能力肯定是有很大的提升的，不过感觉自己本来应该要投入更多的时间去写就是了。

## References

- [1] Kevin Beason. smallpt. homepage : <http://www.kevinbeason.com/smallpt/>.
- [2] 秦岳. 真实感图像渲染报告, 2017.
- [3] 谭咏霖. 基于 bézier 曲线的三维造形与渲染, 2017.

## A 贝塞尔牌奶瓶

在3.5.2我提到了实现曲面的时候发生了一个 bug，具体来说是我求交的时候做了一个判断，使得如果求到了曲面上距离  $t > 0$  的交点以后，下一次取样时可以接受  $t < 0$  的解。这样会使得透明曲面在光线出射时有一定概率按原路反射。

具体的原理我也不知道，但这样的效果应该是使得整个透明的曲面（的特定部位）变成了一个能将一个入射光向四面八方折射的透镜。这个原理我命名为贝塞尔牌奶瓶。

应用这个奇妙的 Bug 以后，我们可以构造出中间光亮两边较暗的模糊光源，以及各种漂亮的焦散形状。



Figure 3: 牛奶瓶模拟路灯 (真正光源在右侧被镜挡住)

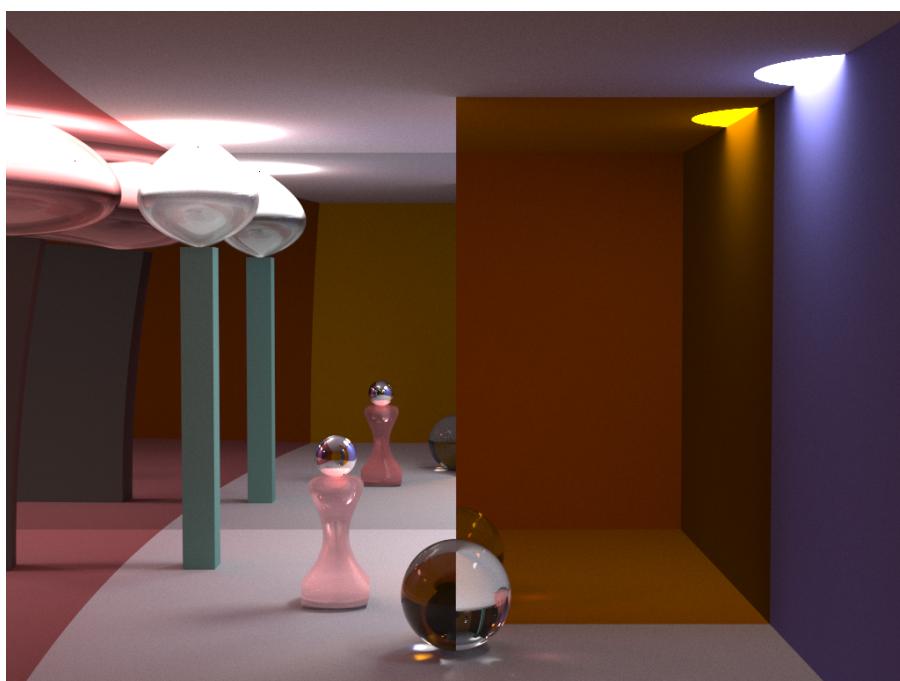


Figure 4: 地上的奶瓶很通透 (不过本图有黑点)

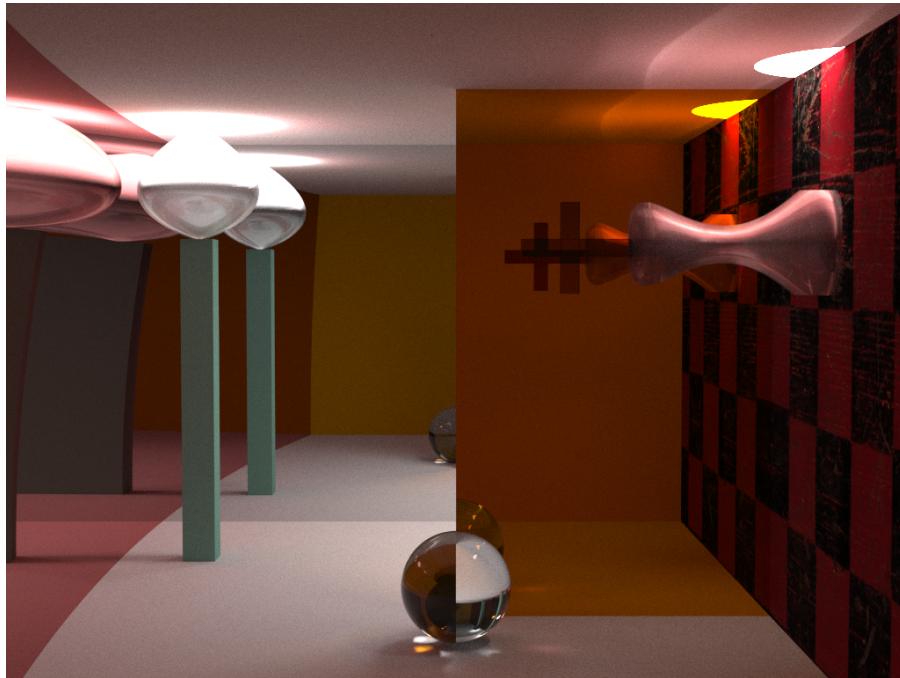


Figure 5: 奶瓶的焦散 (注意右上角的墙)

## B 休息室里的幽靈

以下是我选择了输出  $682 \times 512$  的图，在本地用 Macbook 四线程跑出来的图。  
渲染参数选取为 1000spp。牛顿法参数  $s = 15$ ,  $n = 10$ 。



劇院休息室的幽靈們

注意到这里的曲面样子不太一样(因此叫做幽靈『)。同样的程序得到不同结果我认为是平台实现的问题，服务器的图整体效果更好所以我决定交服务器的图。