

基于Bézier曲面的 三维造型与渲染

计算机图形学—实验报告

计54 周正平 2015011314

zhouzp15@mails.tsinghua.edu.cn

2017年6月25日

目录

1 实验内容	2
1.1 基本功能	2
1.2 附加功能	2
2 双三次Bezier曲面造型	3
2.1 Bezier曲面点坐标	3
2.2 Bezier曲面切向量、法向量	4
2.3 Bezier曲面求交	5
2.3.1 队列四分法+包围盒	5
2.3.2 牛顿迭代法	7
2.4 Bezier曲面纹理	8
3 场景渲染	10
3.1 光线追踪算法	10
3.1.1 景深算法	12
3.1.2 超采样抗锯齿算法	13
3.2 渐进式光子映射算法（PPM）	14
3.2.1 PASS1—光线追踪建立碰撞点图	15
3.2.2 PASS2—光子发射	16
4 最终结果	17
4.1 渲染效果图	17
4.2 Obj文件截图	19

1 实验内容

1.1 基本功能

本实验中，要求基于Bézier曲面进行三维造型与渲染。

在“曲面造型”环节，我实现了双三次Bézier曲面的求交（结合队列四分法及牛顿迭代法实现），并利用它构造了多种模型，包括叶片、画卷、发簪等；



图 1: 叶片



图 2: 发簪

在“场景渲染”环节，我实现了基于渐进式光子映射（PPM算法）的全局光照模型（参考论文Progressive Photon Mapping (Toshiya Hachisuka, et al.)实现），渲染效果不仅具有较高的精准度，更可以较为完美地体现Caustics等如梦似幻的效果。



图 3: 如梦似幻的Caustics



图 4: 纹理细节

1.2 附加功能

在基本功能之外，我还实现了AABB包围盒求交加速、纹理贴图（包括曲面纹理、平面纹理、

球面纹理)、景深效果、超采样抗锯齿等拓展功能，简述如下：

- **AABB包围盒：**在队列四分法+牛顿迭代法实现双三次曲面求交的过程中，需要对四分得到的每个小曲面作一包围盒求交判断，并把有交点的小曲面入队。这样一来，便筛掉了曲面的大部分无交点部分，大大降低了计算成本。由于曲面多较为规整，交点数目不多，故该算法复杂度仅为 $O(\log N)$ ，加速效果显著。
- **纹理贴图：**就实现难度而言，平面纹理与球面纹理相对较易，只需做一简单坐标变换即可；然而对于双三次Bezier曲面而言，由于我求交时采用了四分法，接下来的牛顿迭代法仅能给出小曲面的(u, v)值，而难以获取原曲面的(u, v)坐标。为解决这一瓶颈，我花费了较长时间进行思考，并最终原创了一种实现简明且附加计算成本极低的算法，将在后文详细介绍。
- **景深效果：**根据景深三要素光圈、焦距、物距，对光线的出发点在光圈上做一随机扰动，并确保光线仍能聚焦于定义的焦平面上。实验中，取随机采样次数为10次，可以较好地平衡渲染效果与渲染时间。
- **超采样抗锯齿：**该方法主要针对光线追踪，由于添加了景深效果之后，锯齿也便随之淡化，故而在光子映射生成的图片中未采用该算法。主要流程为：对每个物体，在构造时确定一个随机的id值；每个像素维护一个哈希值，记录本像素的颜色与哪些物体有关。反走样时，只需首先筛选出那些哈希值与上下左右不同的像素，构造需反走样的边界；而后对边界上的每个像素一分为4，对这4个子像素各发射一条光线进行超采样，并对获得的4个颜色值取平均值作为边界像素的最终颜色。

2 双三次Bezier曲面造型

2.1 Bezier曲面点坐标

双三次Bezier曲面由16个控制点构成，每个点的坐标值都是控制点在伯恩斯坦基函数作用下的加权和。

$$P(u, v) = \sum_{i=0}^3 \sum_{j=0}^3 B_i^3(u) B_j^3(v) P_{ij}$$

在程序中，根据(u, v)坐标确定Bezier曲面上的值是大量涉及的操作之一。在参考了网络资源（<https://www.scratchapixel.com/lessons/advanced-rendering/bezier-curve-rendering-utah-teapot>）中的实现之后，最终实现的坐标算法为：首先固定u，确定v方向上的4个控制点，确定一条沿着v方向延展的Bezier曲线；然后在这条v方向的Bezier曲线上，根据v值求出最终的坐标值。

在Bezier曲线上，求出参数为t的一点的坐标值，代码实现如图：

在Bezier曲面上，求出参数为(u, v)的一点的坐标值，代码实现如图：

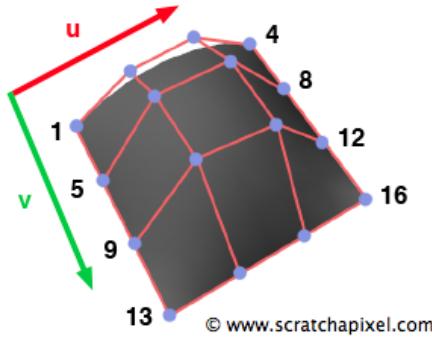


图 5: Bezier曲面坐标

```

18 // 在3次贝塞尔曲线上参数为t的点，P[]的大小必须为4
19 EVec3d Bezier::curvePoint(const EVec3d *P, double t)
20 {
21     double coeff0 = (1 - t) * (1 - t) * (1 - t);
22     double coeff1 = 3 * t * (1 - t) * (1 - t);
23     double coeff2 = 3 * t * t * (1 - t);
24     double coeff3 = t * t * t;
25     return P[0] * coeff0 + P[1] * coeff1 + P[2] * coeff2 + P[3] * coeff3;
26 }
```

图 6: Bezier曲线点坐标算法

```

38 // 在双3次贝塞尔曲面上参数(u, v)的点，P[]的大小必须为16
39 EVec3d Bezier::patchPoint(const EVec3d *P, double u, double v)
40 {
41     EVec3d curve[4];
42     for (int i = 0; i < 4; i++) curve[i] = curvePoint(P + 4 * i, u);
43     return curvePoint(curve, v);
44 }
```

图 7: Bezier曲面点坐标算法

2.2 Bezier曲面切向量、法向量

在微分几何中，曲面的切向量可以用曲面的偏导数来表示。特别地，对Bezier曲面坐标方程求导数时，同样首先固定一个维度，获取一条沿着U或者V方向延展的曲线；而后对这条曲线求导数。

Bezier曲线切向量的计算方法如下图：

Bezier曲面切向量的计算方法如下图（图中为V方向的偏导数，U方向与之类似）：

有了U和V方向的偏导数，便可以通过将偏导数进行叉乘，获得该点处的法向量： $N = U \times V$ ，这在光线追踪算法中起着重要的作用。

```

28 // 在3次贝塞尔曲线上参数为t处的切向量，P[]的大小必须为4
29 EVec3d Bezier::curveDeriv(const EVec3d *P, double t)
30 ▼ {
31     double coeff0 = -3 * (1 - t) * (1 - t);
32     double coeff1 = 3 * (1 - t) * (1 - t) - 6 * t * (1 - t);
33     double coeff2 = 6 * t * (1 - t) - 3 * t * t;
34     double coeff3 = 3 * t * t;
35     return P[0] * coeff0 + P[1] * coeff1 + P[2] * coeff2 + P[3] * coeff3;
36 }

```

图 8: Bezier曲线切向量算法

```

62 // 在双3次贝塞尔曲面上参数(u, v)处v方向的切向量，P[]的大小必须为16
63 EVec3d Bezier::patchDerivV(const EVec3d *P, double u, double v)
64 ▼ {
65     EVec3d curve[4];
66     for (int i = 0; i < 4; i++) curve[i] = curvePoint(P + 4 * i, u);
67     return curveDeriv(curve, v);
68 }

```

图 9: Bezier曲面切向量算法

2.3 Bezier曲面求交

Bezier曲面的求交算法，是本次程序编写中的一大难点。经过与同学的讨论、网上素材的收集工作之后，整理得到一个完整的高效且稳定的算法流程：

- 队列四分：建立一个队列，每次将队首曲面四分，将子曲面中与包围盒有交的那些加入队列，直到队列空或者剩余曲面足够小为止；
- 牛顿迭代：对队列中剩余的足够小的曲面运行牛顿迭代法，选择其中相交距离最近者，取其法向量、交点坐标、纹理颜色返回上层调用。

2.3.1 队列四分法+包围盒

利用Bezier曲面的凸包性，在将子曲面四分处理之后，只要光线与包围盒无交点，即可将该子曲面予以舍弃。

程序中实现了AABB包围盒，参考计算机图形学课件中的Slab快速求交算法及部分网络资源实现（节选）：

可以想象，由于一张Bezier曲面很大，如果直接在上面运行牛顿迭代法，则很容易导致低效、数值不稳定等结果。而四分法中，采取的是类似于八叉树等算法的加速思想，将牛顿迭代法的始点局限于较小的范围内，一方面，大大提高了效率（对数复杂度）；另一方面，使得牛顿迭代法有一个较为合理的初始值，是一种数值稳定的解决方案。

```

163 // 包围盒求交算法 (本段代码部分参考网络资源实现)
164 bool BoundingBox::intersect(const EVec3d &ori, const EVec3d &dir)
165 {
166     const double eps = 1e-6;
167     double ox = ori(0), oy = ori(1), oz = ori(2);
168     double dx = dir(0), dy = dir(1), dz = dir(2);
169     double x0 = Pmin(0), y0 = Pmin(1), z0 = Pmin(2);
170     double x1 = Pmax(0), y1 = Pmax(1), z1 = Pmax(2);
171     double tx_min = -RAND_MAX, ty_min = -RAND_MAX, tz_min = -RAND_MAX;
172     double tx_max = RAND_MAX, ty_max = RAND_MAX, tz_max = RAND_MAX;
173     //x0,y0,z0为包围体的最小顶点
174     //x1,y1,z1为包围体的最大顶点
175     if (fabs(dx) < eps) {
176         //若射线方向矢量的x轴分量为0且原点不在盒体内
177         if (ox < x0 || ox > x1) return false;
178     }
179     else {
180         if (dx >= 0) {
181             tx_min = (x0 - ox) / dx;
182             tx_max = (x1 - ox) / dx;
183         }
184         else {
185             tx_min = (x1 - ox) / dx;
186             tx_max = (x0 - ox) / dx;
187         }
188     }
189     if (fabs(dy) < eps) {
190         //若射线方向矢量的y轴分量为0且原点不在盒体内

```

图 10: AABB包围盒求交算法

实际操作中，要将曲面一分为4，不能简单地采取等分方法，否则新的曲面在分裂处将不能平滑衔接。一种较好的实现算法为课上曾经介绍过的De Casteljau算法，即对控制点进行线性插值，以保证接口平滑：设原曲面的控制点为 $P[]$ ，L和R分别是原曲面分裂出来的新曲面，则插值使用以下公式：

$$L_0 = P_0$$

$$L_1 = P_0/2 + P_1/2$$

$$L_2 = P_0/4 + P_1/2 + P_2/4$$

$$L_3 = P_0/8 + 3P_1/8 + 3P_2/8 + P_3/8$$

另一个子曲面R的公式与L完全对称。这个公式将作为此后笔者在实现纹理贴图时数学推导的基础，发挥重大的作用。代码实现如图：

```

79     // 第一次分裂：沿着一条水平线分裂，得到L，R上下2曲面
80     for (int iv = 0; iv < 4; iv++)
81     {
82         // 沿着V方向竖直的4个控制点
83         EVec3d P0 = P[iv];
84         EVec3d P1 = P[iv + 4];
85         EVec3d P2 = P[iv + 8];
86         EVec3d P3 = P[iv + 12];
87
88         // L(上半曲面)插值
89         L[iv] = P0;
90         L[iv + 4] = P0 / 2 + P1 / 2;
91         L[iv + 8] = P0 / 4 + P1 / 2 + P2 / 4;
92         L[iv + 12] = P0 / 8 + P1 * 3.0 / 8 + P2 * 3.0 / 8 + P3 / 8;
93
94         // R(下半曲面)插值
95         R[iv] = L[iv + 12];
96         R[iv + 4] = P1 / 4 + P2 / 2 + P3 / 4;
97         R[iv + 8] = P2 / 2 + P3 / 2;
98         R[iv + 12] = P3;
99     }
100

```

图 11: De Casteljau插值四分算法

2.3.2 牛顿迭代法

在通过四分法，获取了所有可能还有交点的小曲面之后，对曲面运行牛顿迭代法，利用数值解法求出交点，并在所有交点中，获取距离最小的一个。

牛顿迭代法满足迭代公式： $x_{n+1} = x_n - J^{-1}F(x_n)$ ，其中J为对应的Jacobi矩阵。实验中，取迭代次数为50次，对应代码实现如下：

```

353 // 牛顿迭代法
354 EVec3d Double3Bezier::newton(const EVec3d &ori, const EVec3d &dir, const Node &patch, int maxIter, int *nIter) const
355 {
356     EVec3d x(0, 0, 0), prevX(-1, -1, -1);
357     while ((x - prevX).lpNorm<Eigen::Infinity>() > EPSILON && *nIter < maxIter)
358     {
359         prevX = x;
360         EVec3d L = ori + dir * prevX(0);
361         EVec3d P_ = patchPoint(patch.P, prevX(1), prevX(2));
362         EVec3d derivU = patchDerivU(patch.P, prevX(1), prevX(2));
363         EVec3d derivV = patchDerivV(patch.P, prevX(1), prevX(2));
364
365         // Jacobi矩阵
366         Eigen::Matrix3d Jacobi;
367         Jacobi(0) = dir(0), Jacobi(1) = dir(1), Jacobi(2) = dir(2);
368         Jacobi(3) = -derivU(0), Jacobi(4) = -derivU(1), Jacobi(5) = -derivU(2);
369         Jacobi(6) = -derivV(0), Jacobi(7) = -derivV(1), Jacobi(8) = -derivV(2);
370         x = prevX - Jacobi.inverse() * (L - P_);
371         (*nIter)++;
372     }
373     return x;
374 }

```

图 12: 牛顿迭代法

2.4 Bezier曲面纹理

在以上基本算法已经实现之后，笔者遇到的最大的困难便是为Bezier曲面添加纹理。前面已经叙述，由于笔者在求交时，采用了队列四分法进行加速，这导致根据子曲面的(u, v)坐标倒推原曲面的(u, v)坐标变得异常艰难。然而，笔者在对上面的插值四分法进行深入的研究之后，构思出了一种附加成本极低的算法（当然，笔者推测，前面的插值四分算法便是基于这个规律提出的，这种算法一定曾被之前的学者创造出来，但笔者在实验中确实是基于独立的推导），简述其原理如下：

首先，将上面的插值四分法写成矩阵形式，可以得到：

$$\begin{bmatrix} L_0 \\ L_1 \\ L_2 \\ L_3 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & 0 & 0 \\ \frac{1}{4} & \frac{1}{2} & \frac{1}{4} & 0 \\ \frac{1}{8} & \frac{3}{8} & \frac{3}{8} & \frac{1}{8} \end{bmatrix} \begin{bmatrix} P_0 \\ P_1 \\ P_2 \\ P_3 \end{bmatrix}$$

我们不妨将上式记为 $L = AP$ 的形式。再结合Bezier曲线点坐标的计算公式：

$$f(t) = \begin{bmatrix} (1-t)^3 & 3t(1-t)^2 & 3t^2(1-t) & t^3 \end{bmatrix} \begin{bmatrix} P_0 \\ P_1 \\ P_2 \\ P_3 \end{bmatrix}$$

再将上式记为 $f(t) = BP$ 的形式。故而，结合以上2式，可以得出：

$$f(t) = (BA^{-1})L$$

求得A的逆矩阵为：

$$A^{-1} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ -1 & 2 & 0 & 0 \\ 1 & -4 & 4 & 0 \\ -1 & 6 & -12 & 8 \end{bmatrix}$$

代入 $f(t) = (BA^{-1})L$ 中计算，可以发现，适用于P的参数 t ，恰好是适用于子曲面L的参数 $2t$ 。

至此，笔者顿觉豁然开朗：一维情形如此，那么二维情形，不也该遵循同样的数学之美吗？子曲面的(u, v)，应当与原曲面同样地满足2倍的数量关系。特别地，可以做出如下猜测：（以下说明中， u_P, v_P 为原曲面的坐标， u_L, v_L 为子曲面的坐标）

$uP = uL / 2$	$uP = uL / 2 + 1 / 2$
$vP = uL / 2$	$vP = vL / 2$
$uP = uL / 2$	$uP = uL / 2 + 1 / 2$
$vP = vL / 2 + 1 / 2$	

至此，一个完整的基于曲面四分法的纹理算法呈现在笔者面前：在队列的每个节点Node类中，增加存储4个参数： kU, bU, kV, bV ，用于利用子节点的UV值倒推父节点的UV值。从父节点到子节点，根据上图的规律，不断更新这4个系数，即可得到原曲面中的UV坐标：

$$u_{origin} = kU * u_{final} + bU$$

$$v_{origin} = kV * v_{final} + bV$$

代码实现如图：

```

275      // 分裂出4块小曲面，并根据父节点的k, b计算得到子节点的k, b
276      patchSplit(parentPatch.P, Pchildren);
277      double kU = parentPatch.kU, bU = parentPatch.bU;
278      double kV = parentPatch.kV, bV = parentPatch.bV;
279
280      // 将包围盒有交点的子节点入队（子节点k, b的算法详见报告）
281      // LL : 左上； LR : 右上； RL : 左下； RR : 右下
282      Node LL(Pchildren, kU / 2, bU, kV / 2, bV);
283      if (LL.aabb.intersect(origin, direction))
284          q.push(LL);
285
286      Node LR(Pchildren + 16, kU / 2, bU + kU / 2, kV / 2, bV);
287      if (LR.aabb.intersect(origin, direction))
288          q.push(LR);
289
290      Node RL(Pchildren + 32, kU / 2, bU, kV / 2, bV + kV / 2);
291      if (RL.aabb.intersect(origin, direction))
292          q.push(RL);
293
294      Node RR(Pchildren + 48, kU / 2, bU + kU / 2, kV / 2, bV + kV / 2);
295      if (RR.aabb.intersect(origin, direction))
296          q.push(RR);
297  }
```

经过编程实现，笔者确认了这种算法的正确性，并获得了较为完美逼真的曲面纹理贴图效果：



图 13: 曲面纹理贴图效果

叶片和卷轴对应的Obj文件如下：

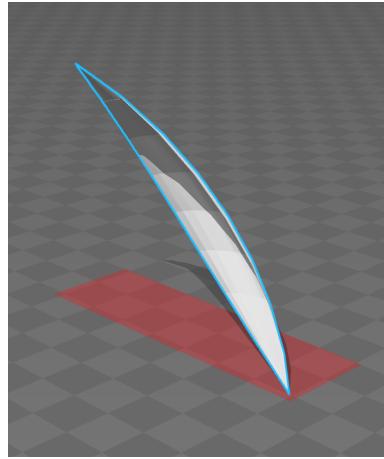


图 14: 叶片Obj

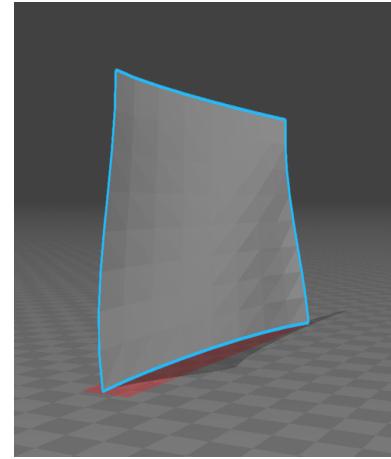


图 15: 卷轴Obj

3 场景渲染

3.1 光线追踪算法

光线追踪（Ray Tracing）是一种有效的渲染算法，其原理是基本的物理规律：想要求得屏幕上像素P的颜色，我们只需从视点发出一条经过P的光线，找到与场景中物体的第一个交点Q。交点Q的颜色也就是像素P的颜色；我们只需求出物体上Q点的颜色，便也就求得了像素P的颜色。

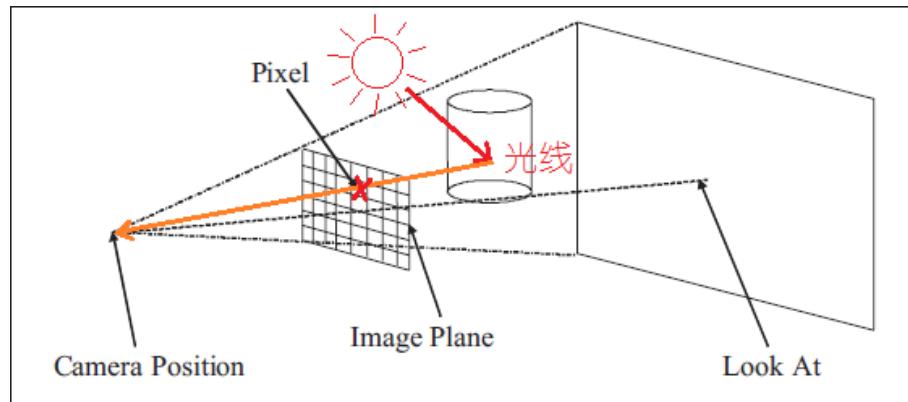


图 16: 光线追踪原理图

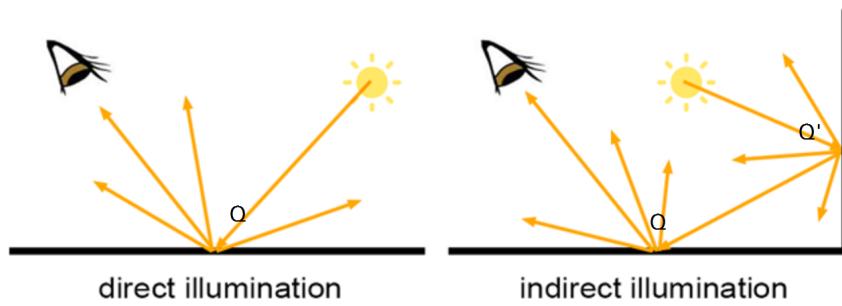


图 17: 直接光照与间接光照

光线追踪的伪代码如下：

```

IntersectColor( vBeginPoint, vDirection)
{
    for each light           // 对屏幕上每个光源
        Color = ambient color; // 初始化颜色值为环境颜色
        Determine IntersectPoint; // 当前光线与该光源求交点
        Color += local shading term; // 计入该光源对交点颜色的影响

        if (surface is reflective) // 如果该表面可以反射，递归追踪反射光线对交点的影响
            color += reflect Coefficient * IntersectColor(IntersecPoint, Reflect Ray);
        else if ( surface is refractive)// 如果该表面可以折射，递归追踪折射光线对交点的影响
            color += refract Coefficient * IntersectColor(IntersecPoint, Refract Ray);
    return color;
}

```

其中，漫反射使用Phong模型计算：

```

135     // 计算Phong模型
136     for (Light *light : m_world->lights)
137     {
138         // 判断阴影
139         bool visible = true;
140         Vec3 L = light->C - P;           // 通往光源的向量
141         double objectDist = L.length();   // 到nearestObject的距离
142         for (Object *object : m_world->objects)
143             if (object != nearestObject
144                 && object->intersect(P, L.normalized(), objectDist))
145                 { visible = false; break; }
146         if (!visible) continue;
147
148         // 计算Phong模型
149         L = L.normalized();
150         Vec3 V = (ori - P).normalized();
151         ret += light->phong(N, L, V, nearestObject->diff, nearestObject->spec, objectColor);
152     }
153 }
```

3.1.1 景深算法

在上面的步骤中，第一步需要从相机发出一条射向场景的光线。景深算法，则是通过在光圈上，对光线的起点进行随机采样，同时保证光线的焦点仍将汇聚于指定的焦平面。

核心代码实现如下：

```

31 // 获取射向屏幕中(h, w)像素的光线起点和方向，考虑景深
32 Camera::Ray Camera::rayAperture(double h, double w) const
33 {
34     h += shiftH; w += shiftW;
35     // 光线初始方向、焦平面上物体位置
36     Vec3 emitDir = F + H * (2 * h / height - 1) + W * (2 * w / width - 1);
37     Vec3 target = C + emitDir * focalDist / -emitDir.z;
38
39     // 在光圈上随机采样，获得随机偏移量detH, detW
40     double detH, detW;
41     do { detH = rand01() * 2.0 - 1.0; detW = rand01() * 2.0 - 1.0; } while (detH * detH + detW * detW >= 1.0);
42     Vec3 unitH = H.normalized(), unitW = W.normalized();
43
44     // 返回本次随机采样获得的光线：对光线起点加以随机扰动，但保证焦平面上物体清晰
45     Vec3 ori = C + (unitH * detH + unitW * detW) * aperture;
46     Vec3 dir = (target - ori).normalized();
47     return make_pair(ori, dir);
48 }
```

在实验中，笔者取随机采样次数为50，渲染出的效果如图：

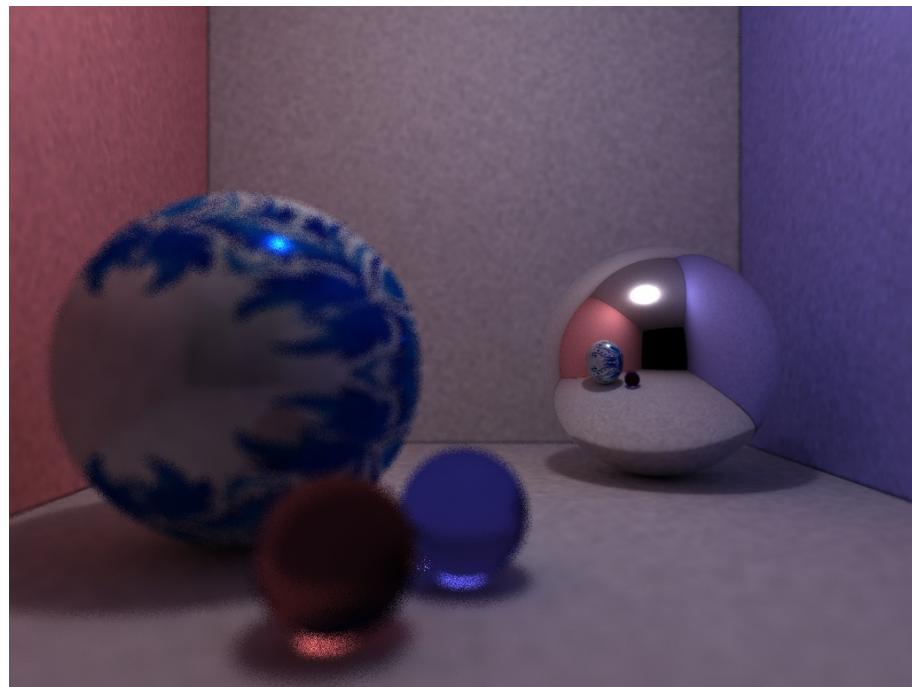


图 18: 景深效果

3.1.2 超采样抗锯齿算法

在光线追踪渲染出的图片中，一个很严重的问题便是锯齿现象严重。

一种行之有效且效率不错的算法，为超采样抗锯齿算法。具体实现方法为：对每个物体，在构造时确定一个随机的id值；每个像素维护一个哈希值，记录本像素的颜色与哪些物体有关。反走样时，只需首先筛选出那些哈希值与上下左右不同的像素，构造需反走样的边界；而后对边界上的每个像素一分为四，对这4个子像素各发射一条光线进行超采样，并对获得的4个颜色值取平均值作为边界像素的最终颜色。

笔者得到的效果如下：

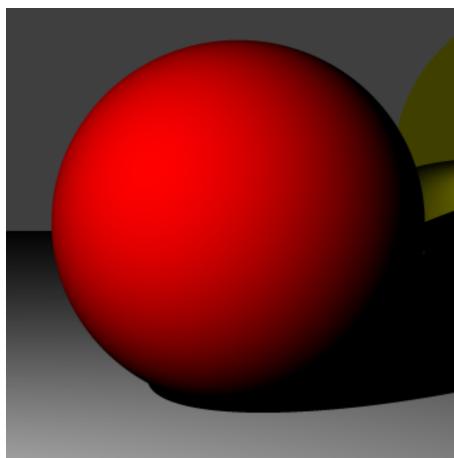


图 19: 经过反走样的物体和阴影边界

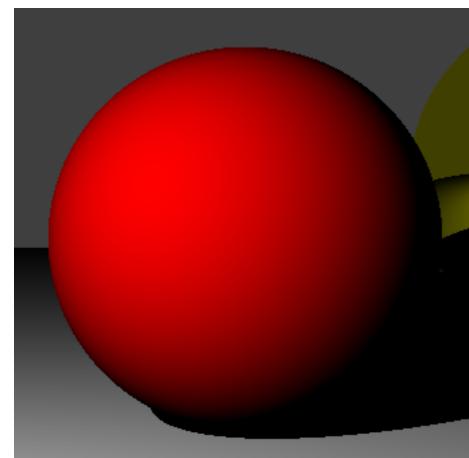


图 20: 未经反走样的物体和阴影边界

代码实现如下：（由于笔者为了实现PPM算法，对原有的光线追踪代码进行了重构，这个版本的代码并未与code一同提交。如果您需要检查这部分代码，请发邮件告知我，我的邮箱是zhouzp15@mails.tsinghua.edu.cn，我可以给您发送这个版本的代码）

```

30     _smoothing = true;
31     cout << "Smoothing..." << endl;
32     for ( int i = 0; i < height; i++ )
33         for ( int j = 0; j < width; j++ )
34             if ( ( i > 0 && _hashTable[i][j] != _hashTable[i - 1][j] )
35                 || ( j > 0 && _hashTable[i][j] != _hashTable[i][j - 1] )
36                 || ( i < height - 1 && _hashTable[i][j] != _hashTable[i + 1][j] )
37                 || ( j < width - 1 && _hashTable[i][j] != _hashTable[i][j + 1] ) ) {
38
39             Color color = 0;
40             for ( int h = -2; h < 2; h++ )
41                 for ( int w = -2; w < 2; w++ ) {
42                     Point O = camera.P;
43                     double idxI = i + double( h ) / 4;
44                     double idxJ = j + double( w ) / 4;
45                     Vector D = camera.ray( idxI, idxJ );
46                     color += rayTracing( O, D, 0 ) * ( 1.0 / 16.0 );
47                 }
48             camera( i, j ) = color;
49         }
50     }

```

3.2 渐进式光子映射算法（PPM）

笔者在认真研读了论文Progressive Photon Mapping (Toshiya Hachisuka, et al.)，并结合Realistic Image Synthesis using Photon Mapping (Henrik Wann Jenson)一书进行辅助理解之后，终于艰难地理解了PPM算法的基本原理。其基本流程如下：

- **光线追踪（PASS1）：**首先，根据上一节详述的光线追踪算法，对场景中的每个像素进行光线追踪，确定与每个像素P相关联的碰撞点 hp_1, hp_2, \dots （因为光线追踪只有在碰到漫反射表面才会停止递归，故而我们看到的碰撞点颜色其实都直接或者间接地来自于某一漫反射表面）。具体操作时，对每个漫反射表面存储其关联的碰撞点，每个碰撞点随着光线的传递，其权值weight相应地叠加沿途物体的颜色；
- **光子发射（PASS2）：**在通过PASS1建立了整个场景的碰撞点图之后，从每个光源发射一定数量的光子到场景中，每当光子打在漫反射表面，就在碰撞点图（KDTree）中询问：“这个光子在哪些碰撞点的半径内部？”将这个光子携带的光通量累计到查询返回的碰撞点中，并增加每个碰撞点的新增光子计数器；
- **更新碰撞点图信息：**分为2大部分：首先，每个碰撞点的累计光通量和半径需要更新；其次，碰撞点图中的每个点维护的最大半径需要更新。更新时，根据论文中的以下公式，进行实现

(N为原始累计光子数, M为本轮新增光子数, R^2 为本碰撞点半径, ϕ_P 为累计光通量, α 为控制衰减速率k的常数):

$$k = \frac{N + \alpha M}{N + M};$$

$$R^2* = k; \phi_P* = k;$$

$$N+ = M; M = 0;$$

- **估计辉度值:** 根据每个像素对应的碰撞点, 及各碰撞点的累计光子数, 可以得出最终屏幕上某一个像素(x, y)的颜色的估计值为 (其中, 图像的亮度需要通过参数 λ 进行调节, 实验中发现对于尺寸不同的场景、不同的光子数, 该参数需取不同的值)

$$\text{photo}[x][y] = \lambda \sum \frac{hp.\phi_P}{hp.R^2 \cdot N_{\text{emit}}}$$

以下, 就PASS1和PASS2两步进行详述。

3.2.1 PASS1—光线追踪建立碰撞点图

根据论文中的描述, 每个碰撞点存储如下信息:

```

71  /**
72  碰撞点类HitPoint, 用途有2 :
73  1. 在RayTracing中, 用于存储碰撞点信息, 并在各个子函数之间传递参数 ;
74  2. 获取碰撞点图(建立KD树), 用于PPM光子映射时计算此处的光通量。
75 */
76 struct HitPoint
77 {
78     Vec3 P, N;           // 碰撞点的位置、法向量
79     Object *object;      // 该碰撞点位于的物体
80     int row, col;        // 碰撞点对应的屏幕坐标(row, col)
81     Color weight;        // 计算此碰撞点处的色光权值, 乘以累计的光通量, 乘以一个系数后为最终颜色值
82     Color phi;          // 本碰撞点处的累计光通量
83     double radius2, maxRadius2; // 本碰撞点的最大半径、本子树下的最大半径
84     double nAccum, nNew;    // 对应于论文中的N、M:之前的累计光子数、本轮新增光子数

```

对场景中所有的漫反射物体, 存储碰撞点; 碰撞点的颜色是沿途经过的所有物体的颜色叠加。

代码实现如下:

```

125  ///////////////////////////////////////////////// 漫反射&高光
126  if (nearestObject->diff > EPSILON || nearestObject->spec > EPSILON)
127  {
128      // Non-specular表面: 存储HitPoint
129      HitPoint hpDiff = hp;
130      hpDiff.object = nearestObject; hpDiff.P = P; hpDiff.N = N;
131      hpDiff.weight *= objectColor * nearestObject->diff;
132      hpDiff.radius2 = INIT_RADIUS * INIT_RADIUS;
133      m_hitpoints.push_back(hpDiff);
134

```

碰撞点图采用KD树进行实现，可以较快地实现范围查询功能：

```

102     Node *another;
103     if (photon.P[split] < m_data[pos].P[split])
104     {
105         another = node->right;
106         insertPhoton(node->left, photon);
107     }
108     else
109     {
110         another = node->left;
111         insertPhoton(node->right, photon);
112     }
113     if ((another) &&
114         (m_data[pos].P[split] - photon.P[split]) * (m_data[pos].P[split] - photon.P[split]) < m_data[another->value].maxRadius2)
115         insertPhoton(another, photon);
116 }
```

3.2.2 PASS2—光子发射

这里，每个光源，依据其能量分布，向场景中随机地发射光子（笔者采用了OpenMP优化性能，也发现了cstdlib中的随机数rand()函数线程不安全，需要每次重新 srand），代码实现如下：

```

65     // 各个光源发射光子
66     for (Light *light : m_world->lights)
67     {
68         int nPhoton = light->color.power() / photonPower;    // 本光源发射的光子数
69         Vec3 photonColor = light->color / nPhoton; // 本光源的光子能量
70
71         // OpenMP多线程加速
72         omp_set_dynamic(0);
73         omp_set_num_threads(8);
74 #pragma omp parallel
75     {
76         srand(int(time(NULL)) ^ omp_get_thread_num()); // rand()线程不安全，需重新设定随机种子
77 #pragma omp for
78         for (int j = 0; j < nPhoton; j++)
79         {
80             if (j % 10000 == 0) cout << "j = " << j << endl;
81             // 在光源上随机选择光线始点、方向
82             Vec3 ori = light->randomPoint();
83             Vec3 dir = Vec3::random();
84             Photon photon(ori, dir, photonColor);
85             tracePhoton(photon, 0);
86         }
87     }
88 }
89 cout << "Elapsed time: " << (clock() - startTime) / CLOCKS_PER_SEC << "s." << endl;
90 }
```

这一步中，可以发现，参数 α 的取值越大，半径收敛也越快，但结果也越不精细；参数初始半径INIT_RADIUS的取值越大，场景中的高频噪声越不明显，但随之造成的问题是，此时对于一些室内场景（如康奈尔盒）墙角的阴影会比较严重：

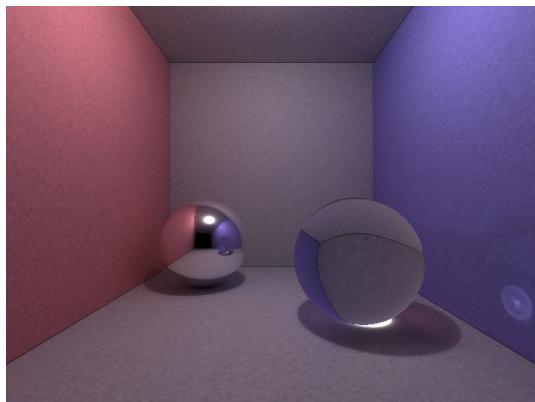


图 21: 较大的初始半径, 墙角有较严重的阴影

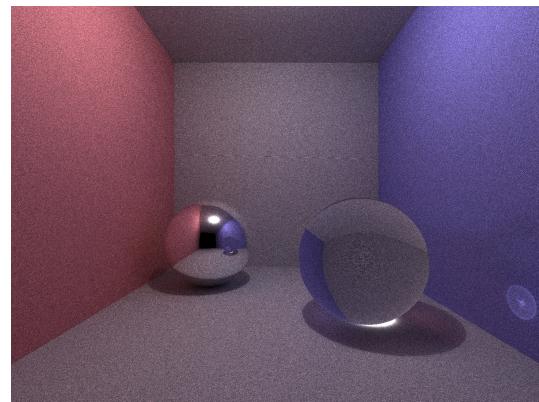


图 22: 较小的初始半径, 高频噪声严重

4 最终结果

4.1 渲染效果图



图 23: 锦瑟流年 (光子映射)

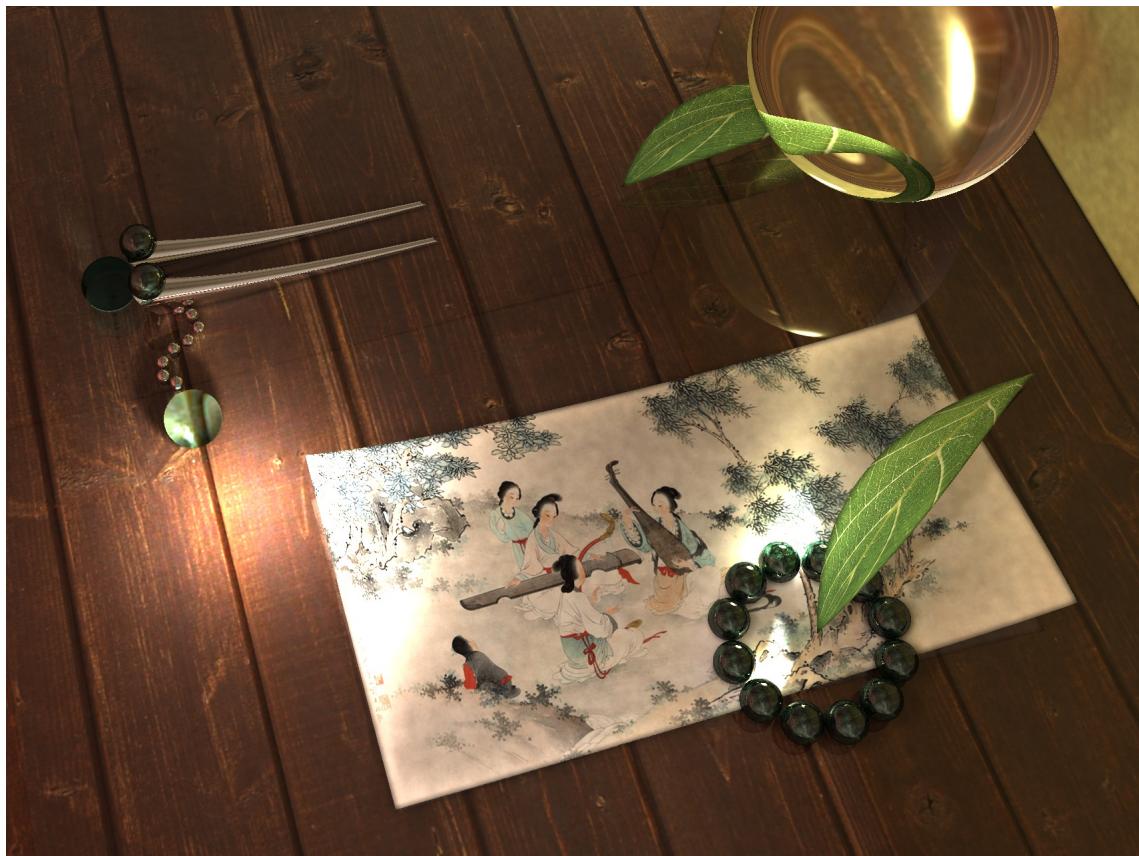


图 24: 锦瑟流年2 (光子映射)

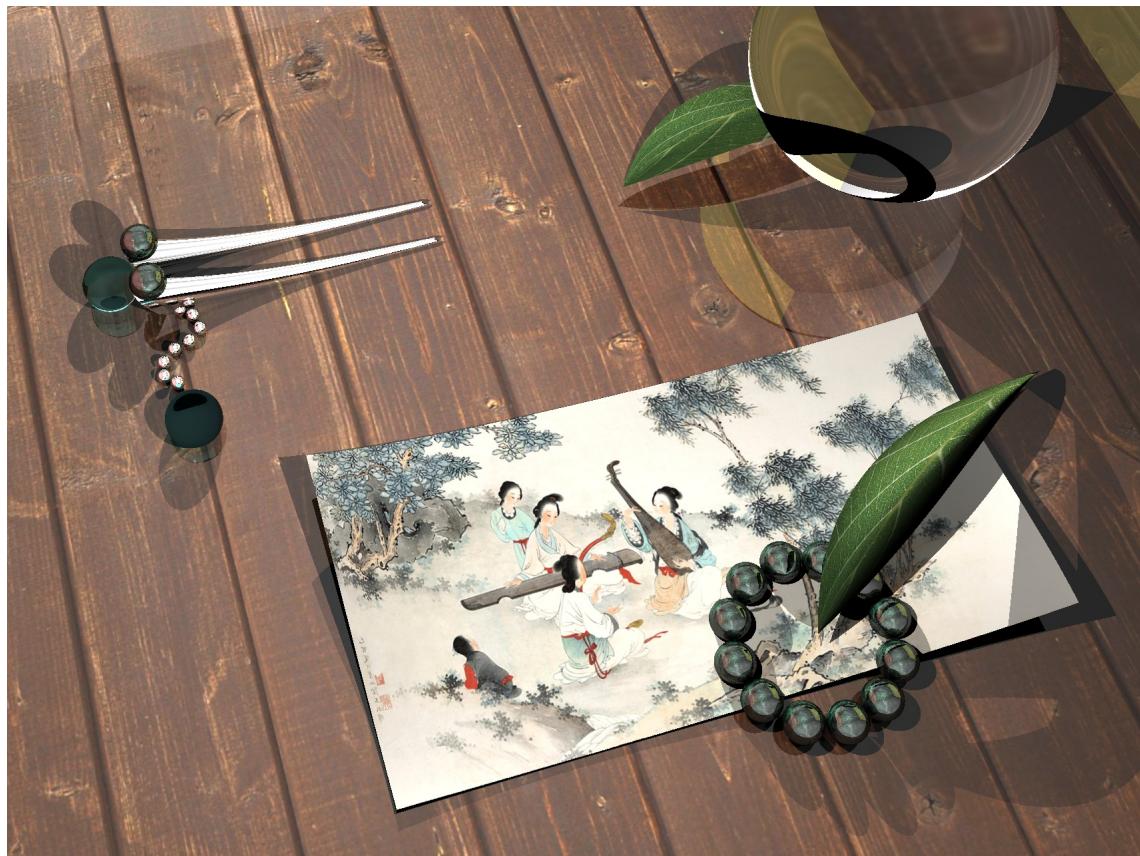


图 25: 锦瑟流年 (光线追踪)

4.2 Obj文件截图

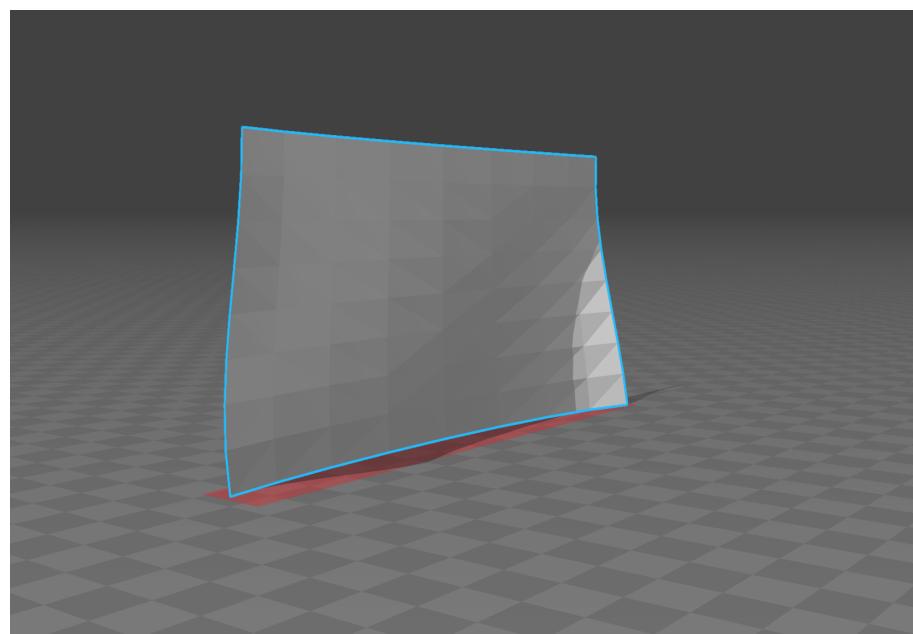


图 26: 卷轴

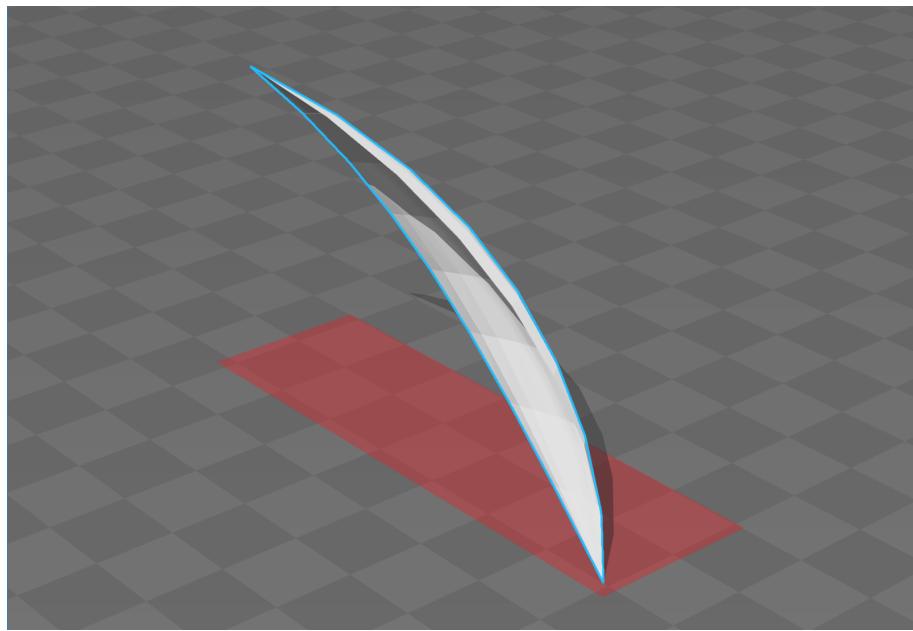


图 27: 叶片

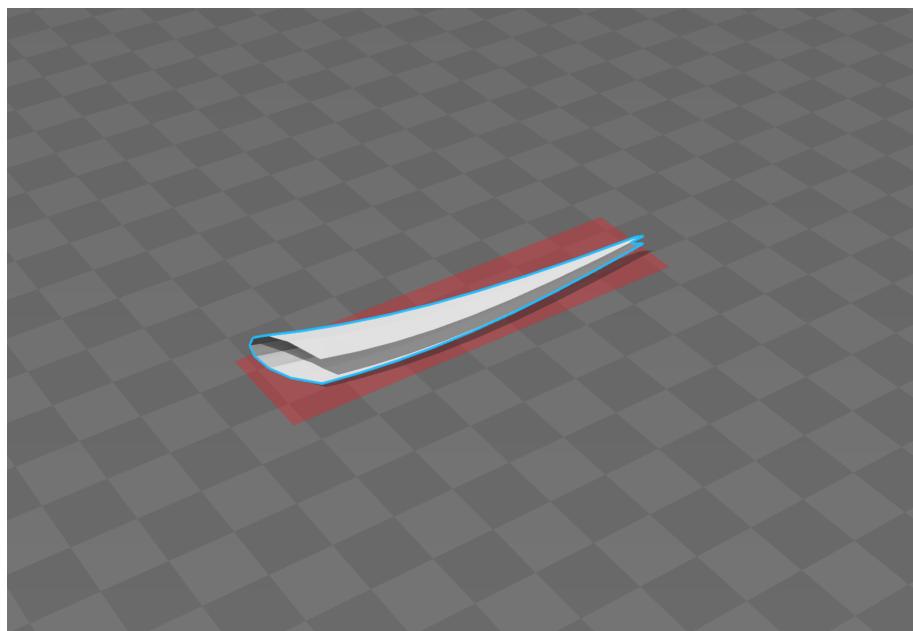


图 28: 发簪