

参数曲线、曲面的三维造形与渲染

计 62 胡致远

2016011260

2018 年 7 月 1 日

1 模型简介

本模型使用基于光子映射的真实感图形绘制方法，对于由矩形、球形、点光源、面积光源构成的场景进行真实感图形绘制，主要有以下几个方面的特点：

1. 实现了 PM 算法，能够表现出 caustics 的效果
2. 实现了针对矩形面片，球形及任意次 Bezier 曲线旋转体的求交操作
3. 使用了 OpenMP 进行并行运算，对渲染过程进行加速
4. 实现了针对面积光源的软阴影效果
5. 使用了纹理贴图，美化设计

与 Bezier 曲线相关的理论推导将在第三部分给出。

2 对应代码段

1. PM 算法，包括光子投射，光子图的组织，平衡以及颜色查询。

```
1 Photonmap* Photontracer::Start()
2 {
3     int maxphotons = 0;
4     float maxpower = 0.0;
5     Light* tmp = scene->GetLightHead();
6     while(tmp)
7     {
8         maxphotons += tmp->GetMaxPhotons();
```

```

9         maxpower += tmp->GetColor().Power();
10        tmp = tmp->GetNext();
11    }
12    Photonmap* photonmap = new Photonmap(maxphotons * MAX_PHOTON_DEP, scene);
13    float photonpower = float(maxpower) / maxphotons;
14    tmp = scene->GetLightHead();
15    while(tmp)
16    {
17    #pragma parallel omp for
18        for(int i = 0; i < tmp->GetMaxPhotons(); i++)
19        {
20            Photon photon = tmp->EmitPhoton();
21            photon.color *= photonpower;
22            PhotonTrace(photon, 1, photonmap, false);
23        }
24        tmp = tmp->GetNext();
25    }
26    photonmap->Balance();
27    return photonmap;
28 }

```

这是光子投射的主函数，负责将各个光源中的光子依次投出，并生成一幅光子图。对于每一个光源，在投射光子时采用并行投射的方法以提高效率。

```

1 void Photontracer::PhotonTrace(Photon photon, int dep, Photonmap* photonmap, bool
    refracted)
2 {
3     if(photon.color.IsBlack())return;
4     if(dep > MAX_PHOTON_DEP)return;
5     Intsct* intsct = scene->GetNearstObj(photon.pos, photon.dir);
6     if(!intsct)return;
7     photon.pos = intsct->P;
8     Material* mat = intsct->GetObj()->GetMaterial();
9     if(mat->cdiff > EPS)
10        photonmap->Store(photon);
11
12    double boardsize = mat->cabso + mat->crefl + mat->crefc;
13    double end = ran() * boardsize;
14    if(end < mat->crefl) Reflect(intsct, photon, dep, photonmap,
        , refracted);
15    else if(end < mat->crefl + mat->crefc) Refract(intsct, photon, dep, photonmap,
        , refracted);
16
17    if(intsct) delete intsct;
18 }

```

光子投射算法，负责跟踪光子并在必要时将其存储。

```

1 void Photontracer::Reflect(Intsct* intsct, Photon photon, int dep, Photonmap*
  photonmap, bool refracted)
2 {
3     Object* obj = intsct->GetObj();
4     photon.dir = ReflDir(intsct->I, intsct->N);
5     Color basecolor = obj->GetColor(intsct->P) * obj->GetMaterial()->crefl;
6     float power = photon.color.Power();
7     photon.color = photon.color * basecolor;
8     PhotonTrace(photon, dep + 1, photonmap, refracted);
9 }
10
11 void Photontracer::Refract(Intsct* intsct, Photon photon, int dep, Photonmap*
  photonmap, bool refracted)
12 {
13     Object* obj = intsct->GetObj();
14     double n = obj->GetMaterial()->N;
15     bool valid;
16     n = refracted ? 1.0 / n : n;
17     photon.dir = RefrDir(intsct->I, intsct->N, n, valid);
18     if(!valid) return;
19     Color basecolor = obj->GetMaterial()->incolor * obj->GetMaterial()->crefc;
20     if(refracted)
21         basecolor = basecolor * (obj->GetMaterial()->absorb * -intsct->dep).Exp()
22         ;
23     float power = photon.color.Power();
24     photon.color = photon.color * basecolor;
25     PhotonTrace(photon, dep + 1, photonmap, refracted);
26 }

```

处理光子的反射和折射。

```

1 void Photonmap::Balance()
2 {
3     std::cout << "Stored Photons=" << photons.size() << std::endl;
4     Photon** p = new Photon*[photons.size()];
5     for(int i = 0; i < photons.size(); i++)
6         p[i] = &photons[i];
7     head = SegBalance(p, 0, photons.size());
8 }
9 KDT* Photonmap::SegBalance(Photon** p, int l, int r)
10 {
11     if(l >= r) return NULL;
12     int mid = (l + r) / 2;
13     KDT* node = new KDT();
14     int axis = 1;
15     if(Box_max.y - Box_min.y > Box_max.x - Box_min.x && Box_max.y - Box_min.y >
      Box_max.z - Box_min.z) axis = 2;
16     if(Box_max.z - Box_min.z > Box_max.x - Box_min.x && Box_max.z - Box_min.z >
      Box_max.y - Box_min.y) axis = 3;
17     Axis = axis;
18     std::nth_element(p + l, p + mid, p + r, cmp);
19 }

```

```

20     node->p = p[mid];
21     node->axis = axis;
22
23     double tmp = Box_max.Project(axis);
24     Box_max.Project(axis) = p[mid]->pos.Project(axis);
25     node->lc = SegBalance(p, l, mid);
26     Box_max.Project(axis) = tmp;
27
28     tmp = Box_min.Project(axis);
29     Box_min.Project(axis) = p[mid]->pos.Project(axis);
30     node->rc = SegBalance(p, mid + 1, r);
31     Box_min.Project(axis) = tmp;
32
33     return node;
34 }

```

将光子图组织为一棵 KDtree, 其中 KDT 为 KDtree 的节点类。

```

1  Color Photonmap::GetColor(Intsct* intsct)
2  {
3      std::priority_queue<KDT> q;
4      head->c = intsct->P;
5
6      Detect(q, head, intsct->P);
7
8      Color ret;
9
10     double maxdis = -1;
11     double coef = scene->GetCamera()->GetCOEF();
12     for(int i = 0; i < K; i++)
13     {
14         Photon* tmp = q.top().p;
15         if(tmp->dir.Dot(intsct->N) < -EPS)
16         {
17             if(maxdis < 0)
18                 maxdis = intsct->P.Dist2(tmp->pos) * coef;
19             double BRDF = intsct->GetObj()->GetMaterial()->BRDF(tmp->dir, intsct
20                 ->N, -intsct->I);
21             ret += tmp->color * BRDF;
22         }
23         q.pop();
24     }
25     if(maxdis > 0) ret /= maxdis;
26     ret = ret * intsct->GetObj()->GetColor(intsct->P);
27     return ret;
28 }
29 void Photonmap::Detect(std::priority_queue<KDT>& q, KDT* node, const Vector3& P)
30 {
31     if(node->lc == NULL && node->rc == NULL)
32     {
33         if(q.size() < K)
34             q.push(*node);
35     }
36 }

```

```

34         else if (node->Norm2() < q.top().Norm2())
35         {
36             q.pop();
37             q.push(*node);
38         }
39         return;
40     }
41     double dist = P.Project(node->axis) - node->p->pos.Project(node->axis);
42     if (dist < 0)
43     {
44         if (node->lc)
45         {
46             node->lc->c = P;
47             Detect(q, node->lc, P);
48         }
49         if (q.size() < K)
50             q.push(*node);
51         else if (node->Norm2() < q.top().Norm2())
52         {
53             q.pop();
54             q.push(*node);
55         }
56         double dist2 = dist * dist;
57         if (node->rc && dist2 < q.top().Norm2())
58         {
59             node->rc->c = P;
60             Detect(q, node->rc, P);
61         }
62     }
63     else
64     {
65         if (node->rc)
66         {
67             node->rc->c = P;
68             Detect(q, node->rc, P);
69         }
70         if (q.size() < K)
71             q.push(*node);
72         else if (node->Norm2() < q.top().Norm2())
73         {
74             q.pop();
75             q.push(*node);
76         }
77         double dist2 = dist * dist;
78         if (node->lc && dist2 < q.top().Norm2())
79         {
80             node->lc->c = P;
81             Detect(q, node->lc, P);
82         }
83     }
84 }

```

颜色查询算法，首先通过 Detect 函数查找最近的 K 个点，将其组织

到一个优先级队列中，然后按照 BRDF 计算光照效果。

2. 矩形面片和球的求交处理

```
1  Intsct* Rectangle::Intersect(const Ray& r0, const Ray& rt)
2  {
3      Intsct* intsct = NULL;
4      Vector3 Rt = rt.Unit();
5
6      double d = N.Dot(Rt);
7      if (fabs(d) < EPS) return intsct;
8
9      double l = (N * R - r0).Dot(N) / d;
10     if (l < EPS) return intsct;
11
12     Vector3 Pos = r0 + Rt * l;
13     double u = (Pos - O).Dot(Dx) / Dx.Norm2();
14     double v = (Pos - O).Dot(Dy) / Dy.Norm2();
15     if (!(0 <= u && u <= 1 && 0 <= v && v <= 1)) return intsct;
16
17     intsct = new Intsct();
18
19     intsct->dep = l;
20     intsct->P = Pos;
21     intsct->N = (d < 0) ? N : -N;
22     intsct->I = Rt;
23     intsct->SetObj(this);
24     return intsct;
25 }
26 Intsct* Sphere::Intersect(const Ray& r0, const Ray& rt)
27 {
28     Intsct* intsct = NULL;
29     Vector3 l = O - r0;
30     Vector3 Rt = rt.Unit();
31     double tp = l.Dot(Rt);
32     double l2 = l.Norm2();
33     double R2 = R * R;
34
35     bool outflag = (l2 - R2 > EPS);
36     if ((outflag || fabs(l2 - R2) < EPS) && tp < EPS) return intsct;
37     double d2 = l2 - tp * tp;
38     if (d2 - R2 > EPS) return intsct;
39     double t = sqrt(R2 - d2);
40     intsct = new Intsct();
41     intsct->dep = outflag ? tp - t : tp + t;
42     intsct->P = r0 + Rt * intsct->dep;
43     intsct->N = (intsct->P - O).Unit();
44     if (!outflag) intsct->N = -intsct->N;
45     intsct->I = Rt;
46     intsct->SetObj(this);
47     return intsct;
48 }
```

3. OpenMP 并行运算

除了在光子投射时采用了并行运算以外，在光线投射时也采用了并行操作。

```
1 Bmp* Camera::Shot()
2 {
3     Bmp* ans = new Bmp(w, h);
4     #pragma omp parallel for
5     for(int i = 0; i < w; i++)
6         for(int j = 0; j < h; j++)
7         {
8             Ray rt = Emit(i,j);
9             ans->SetColor(i,j, raytracer->Raytrace(0, rt, 1, false));
10        }
11    return ans;
12 }
```

4. 针对面积光源的软阴影效果

```
1 void AreaLight::Init()
2 {
3     N = N.Unit();
4     R = O.Dot(N);
5     int maxsamp = int(sqrt((Dx.GetMax() * Dy.GetMax()) / SampleArea));
6     for(int i = 0; i < maxsamp; i++)
7     for(int j = 0; j < maxsamp; j++)
8         v.push_back(0 + Dx * ((double(i) + ran()) / double(maxsamp)) + Dy *
9             ((double(j) + ran()) / double(maxsamp)));
10    if(maxsamp <= 0)
11        v.push_back(0 + Dx / 2 + Dy / 2);
12 }
13 Color Scene::GivePhongColor(Intsct* intsct)
14 {
15     Light* tmp = lighthead;
16     Intsct* tmpint;
17     Color color;
18     while(tmp)
19     {
20         Color lcolor;
21         for(int i = 0; i < tmp->v.size(); i++)
22         {
23             double dist = (tmp->v[i] - intsct->P).Norm();
24             tmpint = GetNearstObj(intsct->P, tmp->v[i] - intsct->P);
25             if(tmpint && tmpint->dep < dist - EPS) delete tmpint;
26             else
27             {
28                 if(tmpint) delete tmpint;
29                 Object* obj = intsct->GetObj();
30                 lcolor += tmp->GetColor() * obj->GetMaterial()->
31                     BRDF(intsct->P - tmp->v[i], intsct->N, -
32                         intsct->I) * obj->GetColor(intsct->P);
33             }
34         }
35     }
36 }
```

```

32         }
33         lcolor /= tmp->GetColor().Power();
34         if(tmp->v.size() > 0)
35             color += lcolor / (tmp->v.size() );
36         tmp = tmp->GetNext();
37     }
38     return color;
39 }

```

在面积光源初始化时采用一定的采样率进行采样，并将采样点存储下来，然后在需要计算局部光照时，依次遍历所有的采样点进行效果的加总。

5. 纹理贴图

```

1  Color Rectangle::GetColor(const Vector3& P)
2  {
3      if(!mat->texture)return mat->color;
4      double u = (P - O).Dot(Dx) / Dx.Norm2();
5      double v = (P - O).Dot(Dy) / Dy.Norm2();
6      return mat->texture->GetColor(u,v);
7  }
8  Color Sphere::GetColor(const Vector3& P)
9  {
10     if(!mat->texture) return mat->color;
11
12     Vector3 dir = (P - O).Unit();
13     double u = acos(dir.Dot(Vector3(0,0,1))) / PI;
14     double v = acos(dir.Dot(Vector3(1,0,0))) / (2 * PI);
15     if(dir.y < -EPS)v = 1 - v;
16     return mat->texture->GetColor(u,v);
17 }
18 Color Texture::GetColor(double u, double v)
19 {
20     double U = u * m;
21     double V = v * n;
22     int U1 = int(floor(U - 0.5 - EPS)), U2 = U1 + 1;
23     int V1 = int(floor(V - 0.5 - EPS)), V2 = V1 + 1;
24     double cU = 0.5 - U + U2;
25     double cV = 0.5 - V + V2;
26     if(cU < EPS)cU = 0.0;
27     if(cV < EPS)cV = 0.0;
28     if(U1 < 0) U1 = m - 1;
29     if(U2 == m) U2 = 0;
30     if(V1 < 0) V1 = n - 1;
31     if(V2 == n) V2 = 0;
32     Color ret;
33     ret = ret + texture->GetColor(U1,V1) * cU * cV;
34     ret = ret + texture->GetColor(U1,V2) * cU * (1 - cV);
35     ret = ret + texture->GetColor(U2,V1) * (1 - cU) * cV;
36     ret = ret + texture->GetColor(U2,V2) * (1 - cU) * (1 - cV);

```



```

37         return ret;
38     }

```

对于矩形和球，分别采用一定的方法计算参数的坐标 u , v ，然后在纹理文件 texture 对应的局部进行采样加总。

Bezier 旋转体

```

1  Intsct* Bezier::Intersect(const Ray& r0, const Ray& rt)
2  {
3      double ut[2] = {0.0, 1.0};
4      bool v = Gend0(r0, rt);
5      if(!v) return NULL;
6      int k[2], K;
7      double e[2], eps, ult[2], lt;
8      for(int i = 0; i < 2; i++)
9      {
10         ult[i] = INF;
11         t = ut[i];
12         for(k[i] = 0; k[i] < 20; k[i]++)
13         {
14             GenB(t);
15             e[i] = iter();
16             if(e[i] < EPS) break;
17         }
18         ut[i] = t;
19         if(e[i] == -1) continue;
20         if(k[i] == 20 && e[i] >= EPS) continue;
21         Vector3 ud = d0 + pn * Pos.x;
22         Vector3 udp = 0 - r0;
23         ult[i] = (udp + Dx * Pos.x + ud).Norm();
24     }
25     if(ult[0] < ult[1])
26     {
27         t = ut[0];
28         GenB(t);
29         eps = e[0];
30         K = k[0];
31         lt = ult[0];
32     }
33     else
34     {
35         eps = e[1];
36         K = k[1];
37         lt = ult[1];
38     }
39     if(eps == -1) return NULL;
40     if(K == 20 && eps >= EPS) return NULL;
41     Vector3 d = d0 + pn * Pos.x;
42     Vector3 dp = 0 - r0;
43
44     Vector3 v1 = Dx.Cross(d);
45     Vector3 v2 = GenDv(d);

```

```

46     Vector3 N = v1.Cross(v2).Unit();
47     Vector3 P = O + d + Dx * Pos.x;
48     if((P - r0).Norm2() < EPS)
49     {
50         double pos = (r0 - O - pn * Pos.x).Norm2() - d.Norm2();
51         if(pos > 0 && rt.Dot(N) > EPS) return NULL;
52         if(pos < 0 && rt.Dot(-N) > EPS) return NULL;
53     }
54
55     if(lt < BEPS) return NULL;
56
57     double delta = N.Dot(rt);
58     if(delta > EPS)
59         N = -N;
60     P += N * EPS * 10;
61
62     Intsct* intsct = new Intsct();
63     intsct->dep = lt;
64     intsct->P = P;
65     intsct->N = N;
66     intsct->I = rt.Unit();
67     intsct->SetObj(this);
68     return intsct;
69 }

```

首先在平面上画出一条 Bezier 曲线，然后绕着给定的旋转轴进行旋转，获得相应的曲面，然后使用 de Casteljau 算法递归计算给定参数值处的坐标，并应用牛顿法迭代得出交点处的参数值，具体算法请参见第三部分。

3 Bezier 曲面相关公式

首先我们约定，这里的 Bezier 曲面指的是，在一个平面上给定一些参数点，将做出的 Bezier 曲面绕给定旋转轴旋转得到的旋转面。

1. 曲面的表示

给定 Bezier 曲面所在平面（下称 Bezier 平面）的原点、x 轴法向、y 轴法向在真实空间中的表示 O , Dx , Dy ，并约定 Bezier 曲线绕 x 轴旋转，然后给出控制点 P_i 在 Bezier 平面中的坐标，即可表示 Bezier 旋转面。

2. 给定参数 t 时计算点的坐标

给定参数 t 时，使用 de Casteljau 算法，可以递归的算出对应的点在 Bezier 平面中的坐标，由于我们在迭代的过程中并不需要这个点在真实空间中的坐标，所以不需要进一步将其转化到真实空间中来。同时

在使用 de Casteljau 算法求出了 B_{ij} 之后还可以进一步求出这一点处 x, y 对 t 的导数, 方便迭代计算。

3. 给定直线, 判断是否相交并求出交点处的参数值

这里我们使用迭代法对交点处的参数 t 进行求解, 迭代法的核心是要找到迭代函数 $f(t)$ 使得相交时函数值刚好为 0, 约定 r_0 表示入射光线原点, rt 为入射光线方向, 并假定 rt 已经单位化, 构造迭代函数如下:

$$\begin{aligned}\vec{n} &= \frac{\vec{rt}}{\vec{rt} \cdot \vec{Dx}} - \vec{Dx} \\ \vec{d}_0 &= \frac{\vec{Dx} \cdot (\vec{O} - \vec{r}_0)}{\vec{Dx} \cdot \vec{rt}} \vec{rt} + \vec{r}_0 - \vec{O} \\ \vec{d} &= \vec{d}_0 + X(t)\vec{n} \\ f(t) &= d^2 - Y(t)^2 = d_0^2 + n^2 X(t)^2 + 2\vec{d}_0 \cdot \vec{n} X(t) - Y(t)^2 \\ &= c_1 + c_2 X(t)^2 + c_3 X(t) - Y(t)^2 \\ f'(t) &= 2c_2 X(t) \frac{dX}{dt} + c_3 \frac{dX}{dr} - 2Y(t) \frac{dY}{dt} \\ t_{n+1} &= t_n - \frac{f(t_n)}{f'(t_n)}\end{aligned}$$

设交点处对应参数为 t , 相应地, 交点在 Bezier 平面上的坐标为 $(X(t), Y(t))$, 这样的—个 Bezier 平面上的点对应于真实空间中以 $O + X(t)\vec{Dx}$ 为圆心, $Y(t)$ 为半径的圆, 且该圆所在平面与 \vec{Dx} 垂直。 \vec{d} 表示直线与参数 t 对应圆平面交点到垂足 (即该圆平面的原点) 的位置向量, 当直线与曲面相交时, 该交点应当正好在圆上, 即有 $\|d\| = Y(t)$, 因此我们可以将 $\|d\|^2 - Y(t)^2$ 作为迭代的指标函数进行迭代。

另外—方面我们发现 $f(t)$ 中的系数是与 t 无关的, 因此可以在直线给出时算—遍参数值存下来即可, 不必重复计算, 引入 $\vec{d}_0 \vec{n}$ 也是这个缘故, 他们分别代表光线与 O 所在圆平面交点对应位置向量即该位置向量随 $X(t)$ 变化的方向向量。

4. 给出交点处参数值, 求解改点处的法向量

首先, 将 \vec{Dx} 和 \vec{d} 叉乘, 可以计算出圆平面中的切向量 \vec{v}_1
其次, 根据 \vec{Dy} 和 \vec{d} 的方向, 可以计算出当前位置相对于基准平面旋转的角度 θ , 然后使用旋转坐标变换, 将控制点 P_i 旋转角度 θ 到对

应交点平面，然后利用切向量计算公式计算出沿 Bezier 曲线的切向量 \vec{v}_2 ，最后有 $N = v1 \times v2$ ，归一化之后即可得到最终的法向量。