参数曲线、曲面的三维造形与渲染

计 62　胡致远
2016011260

2018 年 6 月 9 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和球形的求交操作
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

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。

```
Photonmap* Photontracer::Start()
{
    int maxphotons = 0;
    float maxpower = 0.0;
    Light* tmp = scene->GetLightHead();
    while(tmp)
    {
        maxphotons += tmp->GetMaxPhotons();
        maxpower += tmp->GetColor().Power();
        tmp = tmp->GetNext();
```

```
    }
    Photonmap* photonmap = new Photonmap(maxphotons * MAX_PHOTON_DEP, scene);
    float photonpower = float(maxpower) / maxphotons;
    tmp = scene->GetLightHead();
    while(tmp)
    {
#pragma parallel omp for
        for(int i = 0; i < tmp->GetMaxPhotons(); i++)
        {
            Photon photon = tmp->EmitPhoton();
            photon.color *= photonpower;
            PhotonTrace(photon, 1, photonmap, false);
        }
        tmp = tmp->GetNext();
    }
    photonmap->Balance();
    return photonmap;
}
```

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。

```
void Photontracer::PhotonTrace(Photon photon, int dep, Photonmap* photonmap, bool
     refracted)
{
    if(photon.color.IsBlack())return;
    if(dep > MAX_PHOTON_DEP)return;
    Intsct* intsct = scene->GetNearstObj(photon.pos, photon.dir);
    if(!intsct)return;
    photon.pos = intsct->P;
    Material* mat = intsct->GetObj()->GetMaterial();
    if(mat->cdiff > EPS)
        photonmap->Store(photon);

    double boardsize = mat->cabso + mat->crefl + mat->crefc;
    double end = ran() * boardsize;
         if(end < mat->crefl)              Reflect(intsct, photon, dep, photonmap
              , refracted);
    else if(end < mat->crefl + mat->crefc) Refract(intsct, photon, dep, photonmap
         , refracted);

    if(intsct)  delete intsct;
}
```

光子投射算法，负责跟踪光子并在必要时将其存储。

```
void Photontracer::Reflect(Intsct* intsct, Photon photon, int dep, Photonmap*
     photonmap, bool refracted)
```

```
{
    Object* obj = intsct->GetObj();
    photon.dir = ReflDir(intsct->I, intsct->N);
    Color basecolor = obj->GetColor(intsct->P) * obj->GetMaterial()->crefl;
    float power = photon.color.Power();
    photon.color = photon.color * basecolor;
    PhotonTrace(photon, dep + 1, photonmap, refracted);
}

void Photontracer::Refract(Intsct* intsct, Photon photon, int dep, Photonmap*
     photonmap, bool refracted)
{
    Object* obj = intsct->GetObj();
    double n = obj->GetMaterial()->N;
    bool valid;
    n = refracted ? 1.0 / n : n;
    photon.dir = RefrDir(intsct->I, intsct->N, n, valid);
    if(!valid)return;
    Color basecolor = obj->GetMaterial()->incolor * obj->GetMaterial()->crefc;
    if(refracted)
        basecolor = basecolor * (obj->GetMaterial()->absorb * -intsct->dep).Exp()
             ;
    float power = photon.color.Power();
    photon.color = photon.color * basecolor;
    PhotonTrace(photon, dep + 1, photonmap, refracted);
}
```

处理光子的反射和折射。

```
void Photonmap::Balance()
{
    std::cout << "Stored␣Photons␣=␣" << photons.size() << std::endl;
    Photon** p = new Photon*[photons.size()];
    for(int i = 0; i < photons.size(); i++)
            p[i] = &photons[i];
    head = SegBalance(p, 0, photons.size());
}
KDT* Photonmap::SegBalance(Photon** p, int l, int r)
{
    if(l >= r)return NULL;
    int mid = (l + r) / 2;
    KDT* node = new KDT();
    int axis = 1;
    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;
    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;
    Axis = axis;
    std::nth_element(p + l, p + mid, p + r, cmp);

    node->p = p[mid];
    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
                    ->N, -intsct->I);
20              ret += tmp->color * BRDF;
21          }
22          q.pop();
23      }
24      if(maxdis > 0)ret /= maxdis;
25      ret = ret * intsct->GetObj()->GetColor(intsct->P);
26      return ret;
27  }
28  void Photonmap::Detect(std::priority_queue<KDT>& q, KDT* node, const Vector3& P)
29  {
30      if(node->lc == NULL && node->rc == NULL)
31      {
32          if(q.size() < K)
33              q.push(*node);
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

```
Intsct* Rectangle::Intersect(const Ray& r0, const Ray& rt)
{
    Intsct* intsct = NULL;
    Vector3 Rt = rt.Unit();

    double d = N.Dot(Rt);
    if (fabs(d) < EPS) return intsct;

    double l = (N * R - r0).Dot(N) / d;
    if (l < EPS) return intsct;

    Vector3 Pos = r0 + Rt * l;
    double u = (Pos - O).Dot(Dx) / Dx.Norm2();
    double v = (Pos - O).Dot(Dy) / Dy.Norm2();
    if(!(0 <= u && u <= 1 && 0 <= v && v <= 1)) return intsct;

    intsct = new Intsct();

    intsct->dep = l;
    intsct->P = Pos;
    intsct->N = (d < 0) ? N : -N;
    intsct->I = Rt;
    intsct->SetObj(this);
    return intsct;
}
Intsct* Sphere::Intersect(const Ray& r0, const Ray& rt)
{
    Intsct* intsct = NULL;
    Vector3 l = O - r0;
    Vector3 Rt = rt.Unit();
    double tp = l.Dot(Rt);
    double l2 = l.Norm2();
    double R2 = R * R;

    bool outflag = (l2 - R2 > EPS);
    if((outflag || fabs(l2 - R2) < EPS) && tp < EPS)return intsct;
    double d2 = l2 - tp * tp;
    if(d2 - R2 > EPS)return intsct;
    double t = sqrt(R2 - d2);
    intsct = new Intsct();
    intsct->dep = outflag ? tp - t : tp + t;
    intsct->P = r0 + Rt * intsct->dep;
    intsct->N = (intsct->P - O).Unit();
    if(!outflag)    intsct->N = - intsct->N;
    intsct->I = Rt;
    intsct->SetObj(this);
    return intsct;
}
```

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。

```
Bmp* Camera::Shot()
{
        Bmp* ans = new Bmp(w, h);
#pragma omp parallel for
        for(int i = 0; i < w; i++)
                for(int j = 0; j < h; j++)
                {
                        Ray rt = Emit(i,j);
                        ans->SetColor(i,j, raytracer->Raytrace(O, rt, 1, false));
                }
        return ans;
}
```

4. 针对面积光源的软阴影效果

```
void AreaLight::Init()
{
        N = N.Unit();
        R = O.Dot(N);
        int maxsamp  = int(sqrt((Dx.GetMax() * Dy.GetMax()) / SampleArea));
        for(int i = 0; i < maxsamp; i++)
        for(int j = 0; j < maxsamp; j++)
            v.push_back(O + Dx * ((double(i) + ran()) / double(maxsamp)) + Dy *
                 ((double(j) + ran()) / double(maxsamp)));
    if(maxsamp <= 0)
        v.push_back(O + Dx / 2 + Dy / 2);
}
Color Scene::GivePhongColor(Intsct* intsct)
{
        Light* tmp = lighthead;
        Intsct* tmpint;
        Color color;

        while(tmp)
        {
                Color lcolor;
                for(int i = 0; i < tmp->v.size(); i++)
                {
                        double dist = (tmp->v[i] - intsct->P).Norm();
                        tmpint = GetNearstObj(intsct->P, tmp->v[i] - intsct->P);
                        if(tmpint && tmpint->dep < dist - EPS)delete tmpint;
                        else
                        {
                                if(tmpint)      delete tmpint;
                                Object* obj = intsct->GetObj();
                                lcolor += tmp->GetColor() *  obj->GetMaterial()->
                                     BRDF(intsct->P - tmp->v[i], intsct->N, -
                                     intsct->I) * obj->GetColor(intsct->P);
                        }
```

```
                }
                lcolor /= tmp->GetColor().Power();
                if(tmp->v.size() > 0)
                        color += lcolor / (tmp->v.size() );
                tmp = tmp->GetNext();
        }
        return color;
}
```

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

```
Color Rectangle::GetColor(const Vector3& P)
{
    if(!mat->texture)return mat->color;
    double u = (P - O).Dot(Dx) / Dx.Norm2();
    double v = (P - O).Dot(Dy) / Dy.Norm2();
    return mat->texture->GetColor(u,v);
}
Color Sphere::GetColor(const Vector3& P)
{
    if(!mat->texture)   return mat->color;

    Vector3 dir = (P - O).Unit();
    double u = acos(dir.Dot(Vector3(0,0,1))) / PI;
    double v = acos(dir.Dot(Vector3(1,0,0))) / (2 * PI);
    if(dir.y < -EPS)v = 1 - v;
    return mat->texture->GetColor(u,v);
}
Color Texture::GetColor(double u, double v)
{
    double U = u * m;
    double V = v * n;
    int U1 = int(floor(U - 0.5 - EPS)), U2 = U1 + 1;
    int V1 = int(floor(V - 0.5 - EPS)), V2 = V1 + 1;
    double cU = 0.5 - U + U2;
    double cV = 0.5 - V + V2;
    if(cU < EPS)cU = 0.0;
    if(cV < EPS)cV = 0.0;
    if(U1 < 0)  U1 = m - 1;
    if(U2 == m) U2 = 0;
    if(V1 < 0)  V1 = m - 1;
    if(V2 == n) V2 = 0;
    Color ret;
    ret = ret + texture->GetColor(U1,V1) * cU * cV;
        ret = ret + texture->GetColor(U1,V2) * cU * (1 - cV);
        ret = ret + texture->GetColor(U2,V1) * (1 - cU) * cV;
        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。