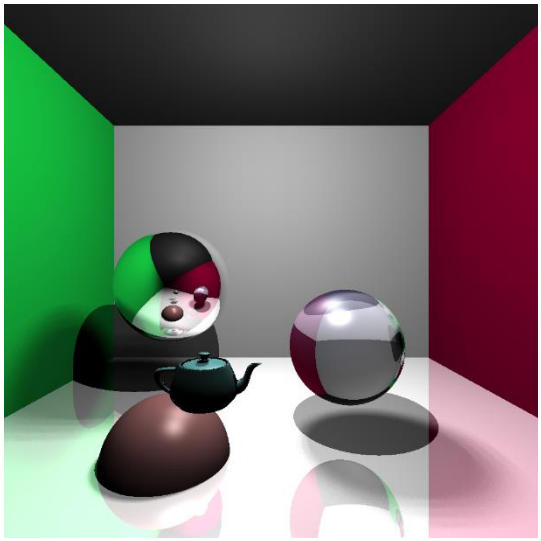
光线追踪大作业

计 63 黄冰鉴 2016011296

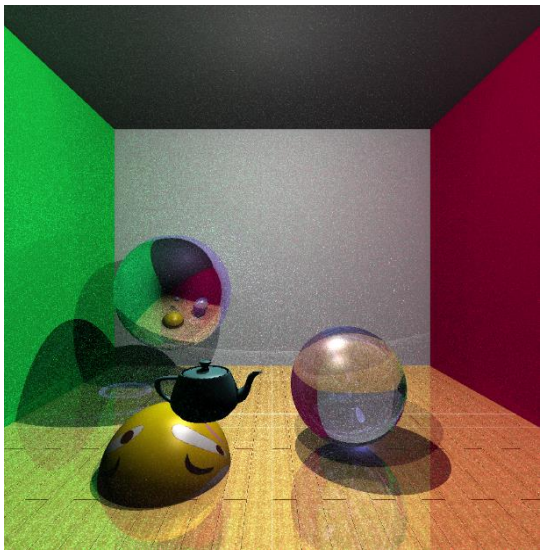
2018 年 6 月

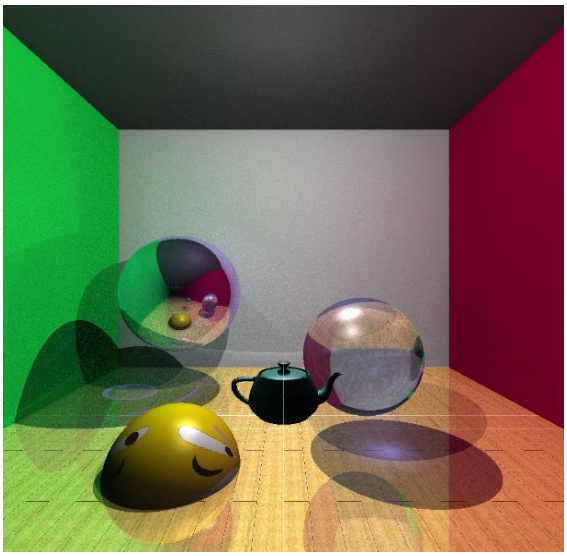
结果图

Ray Tracing



Progressive Photon Mapping





实现内容

基本效果：反射，折射，阴影。

```
//反射
if (thing[p]->material.reflection > 0) {
  Line newline;
  newline.dir = line.dir.Reflect(vecN);
  newline.point = crash_bag.crash_point + newline.dir * EPS;
  colorlight += Intersect(newline, time + 1, i, j, weight * thing[p]->m
aterial.reflection) * thing[p]->material.Getcolor(crash_bag.u, crash_ba
g.v) * thing[p]->material.reflection;//calculate reflect color
}
//折射
if (thing[p]->material.refraction > 0) {
  double n = thing[p]->material.refraction_index;
  Line newline;
  n = 1 / n;
  newline.dir = line.dir.Refract(vecN, n);
  if (!newline.dir.IsZeroVector()) {
    newline.point = crash_bag.crash_point + newline.dir * EPS;
    double absorb = exp((-0.01f) * thing[p]->LengthInside(vecN, newline.
dir));
    colorlight += Intersect(newline, time + 1, i, j, weight * thing[p]->
material.refraction * absorb) * (thing[p]->material.refraction * absor
b);
  }
}
//阴影
for (int i = 0; i < thingnum; ++i) {
  if (i == p) continue;
  other_bag = thing[i]->Crash(crash_point, l);
  if (!other_bag.crash_point.IsNullVector()) {
    if ((other_bag.crash_point - crash_point).Module2() < dist) {
      return Color(0, 0, 0);
    }
  }
}
//phong 模型漫反射和高光
if (crash_thing->material.diffusion > 0) {
  double dot = l.Dot(vecN);
  if (dot > 0) {
    double diff = dot * crash_thing->material.diffusion;
    c += crash_thing->material.Getcolor(bag.u, bag.v) * crash_light->ma
terial.color * diff;
  }
}
if (crash_thing->material.specular > 0) {
  double dot = r.Dot(view_direction);
  if (dot > 0) {
```

```
    c += crash_light->material.color * crash_thing->material.specular *
 pow(dot, 20);
  }
}
```

参数曲面求交

```
CrashBag Bezier::Crash(Vec3 source, Vec3 dir)
{
    Vec3 crash_point = Crash_cover(source, dir);//包围盒
    if (crash_point.IsNullVector()) return CrashBag();//和包围盒不相交，
退出。
    //牛顿迭代
    double u, v, t;
    double t0 = (crash_point - source).Module();
    Vec3 temp, tempu, tempv;
    Eigen::Matrix3d dF, dFinverse;
    Eigen::Vector3d p, pnext, F;
    dF(0, 2) = -dir.x;
    dF(1, 2) = -dir.y;
    dF(2, 2) = -dir.z;
    for (int q = 0; q < 10; ++q) {//10 次不同的初始值尝试
        u = (double)rand() / RAND_MAX;
        v = (double)rand() / RAND_MAX;
        t = t0;
        crash_point = source + dir*t;
        for (int i = 0; i < 10; ++i) {//每个初始值迭代 10 轮
            p << u, v, t;
            temp = GetPoint(u, v) - crash_point;
            F << temp.x, temp.y, temp.z;
            tempu = GetDiffU(u, v);
            tempv = GetDiffV(u, v);
            dF(0, 0) = tempu.x;
            dF(1, 0) = tempu.y;
            dF(2, 0) = tempu.z;
            dF(0, 1) = tempv.x;
            dF(1, 1) = tempv.y;
            dF(2, 1) = tempv.z;
            dFinverse = dF.inverse();dFinverse(2, 1));
            pnext = p - dFinverse * F;
            u = pnext(0);
            v = pnext(1);
            t = pnext(2);
            crash_point = source + dir*t;
            if (temp.IsZeroVector() && u >= 0 && u <= 1 && v >= 0 && v
<= 1) {//迭代误差达到要求并且 u，v 在[0,1]之间。
                CrashBag bag;
                bag.crash_point = crash_point;
                Vec3 cross = (tempu*tempv).GetUnitVector();
                if (cross.Dot(dir) < 0) bag.vecN = cross;
```

```
                else bag.vecN = cross*(-1);
                bag.u = u;
                bag.v = v;
                return bag;
            }
        }
    }
    return CrashBag();
}
```

PPM

```
for (int round = 0; round < 500; ++round) {
  count = 0;
  tracer->ShootPhoton();
  tracer->hplist->CalculatePhotonImage();
}

void Raytracer::ShootPhoton()
{
    for (int i = 0; i < lightnum; ++i) {
        #pragma omp parallel num_threads(8) schedule(dynamic, 1)
        for (int j = 0; j < 1000000; ++j) {
            Line photon = light[i]->GeneratePhoton();
            PhotonIntersect(photon, light[i]->material.color, 0, 0.9);
        }
    }
    printf("photon tracing finished\n");
}
void HitPointList::CalculatePhotonImage()
{
    int x, y;
    Color contribute;
    for (int i = 0; i < hashvalue; ++i) {
        HitPoint *p = list[i].next;
        while (p != NULL) {
            if (p->count == 0) {
                p = p->next;
                continue;
            }
            contribute = p->colorflux / p->count;
            x = p->imagex;
            y = p->imagey;
            photonimage[x][y] += contribute * p->weight;
            p->radius *= 0.99;
            p = p->next;
        }
    }
}
```

加速：bezier 曲面求交使用了包围盒，光子映射使用了 hash 的方法组织光子，openmp 并行加速。

```
//六个面的包围盒
Vec3 Bezier::Crash_cover(Vec3 source, Vec3 dir)
{
    Vec3 crash_point;
    crash_point = cover[0].Crash(source, dir).crash_point;
    if (!crash_point.IsNullVector()) {
        if (crash_point.y >= mincover.y && crash_point.y <= maxcover.y
&& crash_point.z >= mincover.z && crash_point.z <= maxcover.z)
            return crash_point;
    }
    crash_point = cover[1].Crash(source, dir).crash_point;
    if (!crash_point.IsNullVector()) {
        if (crash_point.y >= mincover.y && crash_point.y <= maxcover.y
&& crash_point.z >= mincover.z && crash_point.z <= maxcover.z)
            return crash_point;
    }
    crash_point = cover[2].Crash(source, dir).crash_point;
    if (!crash_point.IsNullVector()) {
        if (crash_point.x >= mincover.x && crash_point.x <= maxcover.x
&& crash_point.z >= mincover.z && crash_point.z <= maxcover.z)
            return crash_point;
    }
    crash_point = cover[3].Crash(source, dir).crash_point;
    if (!crash_point.IsNullVector()) {
        if (crash_point.x >= mincover.x && crash_point.x <= maxcover.x
&& crash_point.z >= mincover.z && crash_point.z <= maxcover.z)
            return crash_point;
    }
    crash_point = cover[4].Crash(source, dir).crash_point;
    if (!crash_point.IsNullVector()) {
        if (crash_point.x >= mincover.x && crash_point.x <= maxcover.x
&& crash_point.y >= mincover.y && crash_point.y <= maxcover.y)
            return crash_point;
    }
    crash_point = cover[5].Crash(source, dir).crash_point;
    if (!crash_point.IsNullVector()) {
        if (crash_point.x >= mincover.x && crash_point.x <= maxcover.x
&& crash_point.y >= mincover.y && crash_point.y <= maxcover.y)
            return crash_point;
    }
    return Vec3(10000,10000,10000);
}

//hash 函数
int HitPointList::Hash(Vec3 point)
{
    int x = point.x * 10;
```

```
    int y = point.y * 10;
    int z = point.z * 10;
    return (unsigned int)((x * 73856093) ^ (y * 19349663) ^ (z * 834927
91)) % hashvalue;
}
//把光子加入到周围碰撞点的光通量。
void HitPointList::AddPhoton(Vec3 point, Vec3 dir, Color color)
{
    int hash = Hash(point);
    HitPoint *p = &list[hash];
    while (p != NULL) {
        p->Update(point, dir, color);
        p = p->next;
    }
}

//openmp
#pragma omp parallel for num_threads(8)
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < m; ++j) {
            Line line = camera->viewlines[i][j];
            Color c = Intersect(line, 0, i, j, 1);
            image[i][j] = c;
        }
    }
```

其它：软阴影，UV 贴图。

```
//面光源带位置扰动
Vec3 Plainlight::Getpos()
{
    double w = (double)rand() / RAND_MAX - 0.5;//(-0.5,0.5)
    double l = (double)rand() / RAND_MAX - 0.5;
    Vec3 ret(center);
    ret.x += w*width;
    ret.z += l*length;
    return ret;
}

//UV 贴图
Color Material::Getcolor(double u, double v)
{
    if (haveveins && u >= 0 && u <= 1 && v >= 0 && v <= 1) {//有贴图
        int x = veins.rows * u;
        int y = veins.cols * v;
        cv::Vec3b v = veins.at<cv::Vec3b>(x, y);
        Color c(v[2], v[1], v[0]);
        c /= 255;
        return c;
    }
    else {//无贴图
```

```
        return color;
    }
}
```

实现过程

- 基本功能在此就不介绍了。

- 参数曲面求交使用的是牛顿迭代的方法。
  - 给每一个 bezier 曲面构造一个沿坐标轴方向的包围盒，光线先和包围盒求交，如果有交点，再牛顿迭代。
  - 利用包围盒求交时的参数作为牛顿迭代的初始值，每个初始值迭代 10 轮，一共 10 组初始值。如果最后误差达到要求范围，则返回交点，法向和 uv 等信息。

- PPM 的实现参考了原作者的论文。
  - 先进行一轮正常的 raytracing，计算每条光线的碰撞点（hitpoint），用 hash 函数组织好。
  - 接下来，每发射一轮光子，
    - 先将所有光子根据碰撞点归到每个 hitpoint 的光通量中。
    - 再根据每个 hitpoint 的光通量，光子数和半径计算出 hitpoint 的颜色。
    - 将 hitpoint 加权求和作为视线中视点的颜色。
- 软阴影的实现是把一个面光源表示为 16 个带位置扰动的点，每次算阴影的时候和这些点求阴影即可。
- UV 贴图。任何参数模型上的点都可以定义一个 uv(范围是(0,1))，把 uv 和纹理图片的坐标对应即可。
  - bezier 曲面：uv 是已经定义好的，直接使用就可以。
  - 球面：求出球面上的点在球坐标下的 φ 和 θ，用 φ/2π，θ/2π 作为 u，v。
  - 平面：取法向中较小的两维（比如（0,1,0），则取 x 轴和 z 轴），平面上的任意一点 P，取 Px 和 Pz 的小数部分作为 u 和 v。
- 另外，在求交的时候，我把交点，法向和 uv 打包传输，发现这是一个好方法，避免了很多不必要的多余计算和函数调用。

总结

历时两个月的超级大作业终于写完了！！！

从最开始根本不知道如何入手，到写出第一个版本，却失望地看到渲染出来的是看不懂的图案，花了好几天时间才看到正常图案。从对 bezier 参数曲面求交一头雾水，到和同学交流之后一步一步自己写出了求交，满眼欣喜地看着出现在画面中的茶壶。从满足于简单的 ray_tracing，到看到同学都做了 ppm，逼迫自己看论文，理解思想，最后写出了基本能看的 ppm。

项目本身的代码量不是太大，但是每一行都凝聚了很多思考的精华。代码本身的架构也是一改再改，不断适应新功能的要求。逐渐体验到了 OOP 对扩展的帮助有多大。

虽然最后还是没有能够写出真实感足够的场景，但是已经很满足了。