Ray Tracing Report

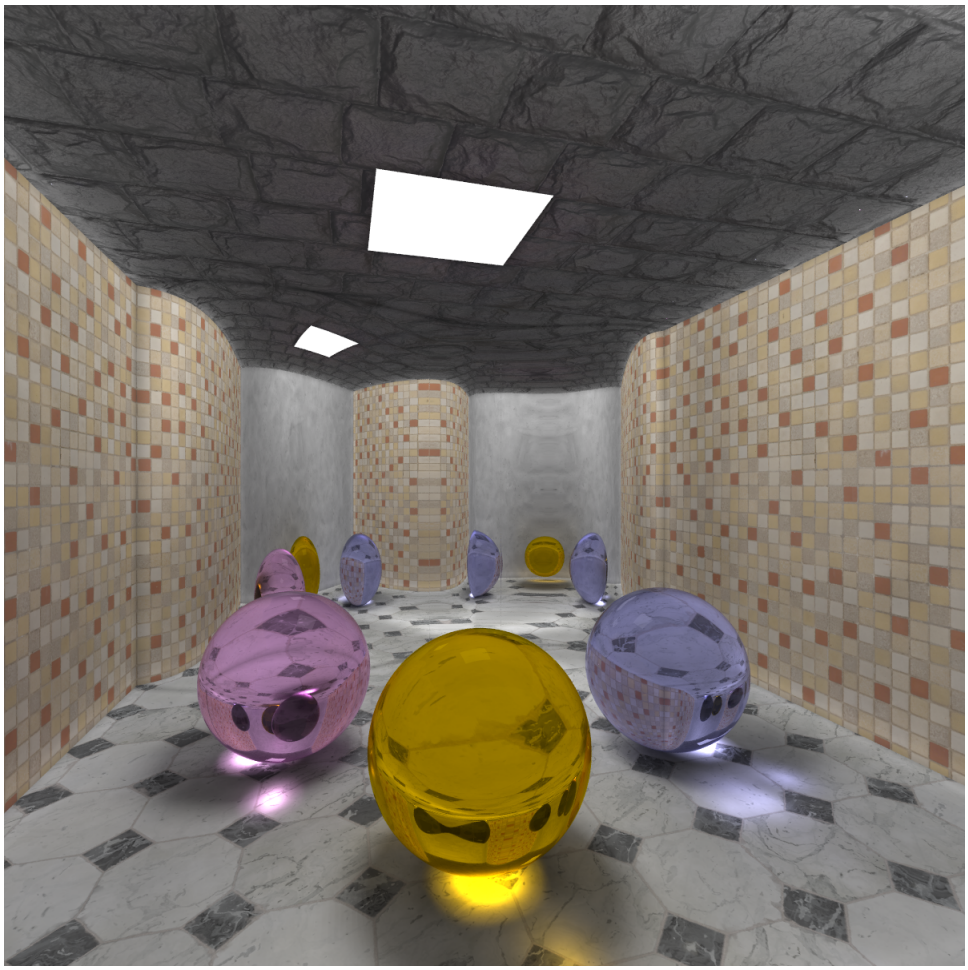
卢彦旻计64 2015010033

1 概述

本次光线跟踪大作业主要实现了以下效果：

- 光线跟踪算法：photon map(direct illumination + diffuse reflection + caustics)，自带软阴影效果
- 参数曲面求交
- 超采样抗锯齿（distribution ray tracing）
- 纹理贴图和凹凸贴图
- 加速：kdtree(photon map search)，包围盒（bezier 求交，其实是包围平面）

最终效果见下图（分辨率：1200×1200)



2 实现方法及主要代码

由于时间问题，来不及给代码写详细的注释了，造成的不便十分抱歉。各代码文件内容的简单说明：

- global.h/cpp: 要用到的一些头文件，定义了一些渲染的参数和要用到的常数，和一些使用较多的辅助函数。
- light.h/cpp: 光源的类
- material.h/cpp: 材质的类
- object.h/cpp: 碰撞物体的类，存储碰撞信息的结构体，光线结构体

- photon_map.h/cpp: 主要是 kdtree 和光子信息的结构体
- scene.h/cpp: 相机类，场景类，把基本上所有主要函数都放里面了
- main.cpp: 场景的搭建

2.1 photon map

第一步，从光源出发，随机地发射光子，光子与漫反射平面相遇时，将信息存入 global map，那些经过折射和反射的光子第一次遇到漫反射面时，将信息存入 caustics map。

第二步，进行光线跟踪。光线遇到折射和反射面时，递归求解。遇到漫发射面时，计算颜色。分为三部分：

1. 直接光照，由于是面光源，事先生成了一系列的采样点，将每点的贡献叠加（phong 模型计算）。
2. 漫发射，用 global map 的光子信息计算颜色值，用 cone filter 进行降噪。
3. 焦散效果，用 caustics map 的光子信息计算颜色值，用 cone filter 进行降噪。

相关代码太多贴不了了

2.2 参数曲面求交

根据想做的效果，我只写了三次 Bezier 曲线平移面的求交，最终的 Bezier 镜面是两条三次曲线的平移面拼接而成，拼接处 C^1 连续。这里假设平移方向垂直于 Bezier 曲线，这样可以在正交坐标系下求解，否则可能要用斜坐标系。求交时，将光线方向投影到 Bezier 曲线所在平面，求解平面求交问题得到 xy 坐标，立体交点，平面交点和光线起点事实上构成一个直角三角形，故 z 坐标容易求得。如果 Bezier 曲线所在平面不平行于坐标轴，则需要再进行坐标系变换。三次 Bezier 的一般方程为

$$f(u) = (1-3u+3u^2-u^3)\mathbf{p}_0+(3u-6u^2+3u^3)\mathbf{p}_1+(3u^2-3u^3)\mathbf{p}_2+u^3\mathbf{p}_3, 0 \leqslant u \leqslant 1$$

$$f(u) = u^3\mathbf{A} + u^2\mathbf{B} + u\mathbf{C} + \mathbf{D}$$

设光线方程 $\mathbf{e}+t\mathbf{d}=0$, 则要求解两元方程组：

$$\mathbf{e}+t\mathbf{d}=u^3\mathbf{A}+u^2\mathbf{B}+u\mathbf{C}+\mathbf{D}$$

设 $\mathbf{d}=[d_1\ d_2]^T$, 两边乘以矩阵 $[-d2\ d1]$ 可消去变量 t，然后可用一元三次方程求根公式得到 u 的值，回代得到 t 的值。

以下为解三次方程以及验证解合法性的代码，其中 equationCoeff 就是上面式子里的向量 A, B, C, D

```
void CubicBezierWall::solveCubicEquation(double a, double b, double c,
    double d, double* ans)
{
  double p, q;
  p = (3 * a*c - b*b) / (9 * a*a);
  q = (27 * a*a*d - 9 * a*b*c + 2 * b*b*b) / (54 * a*a*a);
  double delta = q*q + p*p*p;
  double sdelta = b / (3 * a);
  if (delta > 0)
  {
    delta = sqrt(delta);
    double x1 = powerWrap(-q + delta, 0.33333) + powerWrap(-q - delta,
    0.33333);

    x1 -= sdelta;
    ans[0] = x1;
    ans[1] = -1;
    ans[2] = -1;
  }
  else
  {
    complex<double> t1, t2, dd(delta);
    complex<double> o1(-0.5, sqrt(3) / 2.0),o2;
    o2._Val[0] = -0.5;
    o2._Val[1] = -o1.imag();
    dd = sqrt(dd);
    t1 = pow(-q + dd, 0.33333);
    t2 = pow(-q - dd, 0.33333);
    ans[0] = (t1 + t2).real() - sdelta;
    ans[1] = (o1*t1 + o2*t2).real() - sdelta;
    ans[2] = (o2*t1 + o1*t2).real() - sdelta;
  }
}

double CubicBezierWall::intersect(const Point2d& e, const Point2d& d,
    Point2d* intersection)
{
```

```
35    double ans[3];
36    double t = -1;
37    Vec2d dd(-d.val[1], d.val[0]);
38    solveCubicEquation(dd.ddot(equationCoeff[0]), dd.ddot(equationCoeff[1]),
        dd.ddot(equationCoeff[2]), dd.ddot(equationCoeff[3] - e), ans);
39    Point2d  tp;
40    double dist = INFINITY, td;
41    double c1, c2, c3;
42    for (int i = 0; i < 3; i++)
43    {
44      if (ans[i] > 0 && ans[i] < 1)
45      {
46        c3 = ans[i];
47        c2 = c3*c3;
48        c1 = c2*c3;
49        tp = equationCoeff[0] * c1 + equationCoeff[1] * c2 + equationCoeff[2]
      * c3 + equationCoeff[3];
50        int id = 0;
51        if (d.val[0] * d.val[0] < d.val[1] * d.val[1])
52        {
53          id = 1;
54        }
55        td = (tp.val[id] - e.val[id]) / d.val[id];
56        if (td < dist)
57        {
58          dist = td;
59          if(intersection)
60            *intersection = tp;
61          t = c3;
62        }
63      }
64    }
65    return t;
66  }
```

超采样抗锯齿

这一项比较简单，直接在原来每个像素内生成随机位置采样即可，将色彩值叠加再除以采样数即可。我的采样数是 20。

```
1   for (int i = 0; i < Height; i++)
2     {
3       Point3d renderPixel = canvasLeftUp - i * gridZ;
4       for (int j = 0; j < Width; j++, renderPixel+=gridX)
5       {
```

```
      color.val[0] = color.val[1] = color.val[2] = 0;
      for (int k = 0; k < RayTracingSampleNum; k++)
      {
        HitInfo hitInfo;
        sampleColor.val[0] = sampleColor.val[1] = sampleColor.val[2] = 0;
        samplePoint = renderPixel + Uniform1_1*halfgridX + Uniform1_1*
    halfgridZ;
        ray.start = cam.pos;
        ray.direction = normalize(samplePoint - cam.pos);
        depth = 0;

        findFirstObjectHit(ray, hitInfo);
        if (hitInfo.hitObj)
        {
          sampleColor = computeColor(ray.start + hitInfo.dist * ray.
    direction, ray.direction, hitInfo, 0);
        }
        color += sampleColor / RayTracingSampleNum;
      }
      img.at<Vec3b>(i, j) = Vec3b(gamma(color.val[2]), gamma(color.val[1]),
    gamma(color.val[0]));
    }
  }
```

2.3 纹理贴图和凹凸贴图

纹理贴图：代码中实现了平面和球的纹理映射，由于构图考虑，没有渲染纹理球。原理比较简单，将三维的点映射到二维坐标 (u,v)，然后根据 u,v 值取出相应位置像素点。

下面分别是球和平面映射到 uv，uv 映射到纹理的函数

```
void Sphere::computeTextureCoor(const Point3d& poi, double& u, double& v)
{
  Point3d tp = poi - center;
  u = 0.5 + atan2(tp.val[1], tp.val[0]) / (2 * PI);
  v = 1 - acos(tp.val[2] / radius) / PI;
}

void Plane::computeTextureCoor(const Point3d& p, double& u, double& v)
{
  Vec3d tv = p - point;
  u = tv.ddot(axesU)/TextureScale;
  v = tv.ddot(axesV)/TextureScale;
  u -= (int)u;
```

```
  u = u > 0 ? u : u + 1;
  v -= (int)v;
  v = v > 0 ? v : v + 1;
}

Vec3d Material::getColor(double u, double v)
{
  return texture->at<Vec3f>(u*size-0.5, v*size-0.5);
}
```

凹凸贴图：同样实现了平面和球的，通过对物体每点的法向量进行扰动实现凹凸效果。首先读入一张带有纹理高度信息的灰度图 bump map，将灰度值作为高度值，计算每一点的梯度值，从而得到该点的法向量的值，得到 normal map，使用方法类似于纹理映射。在计算物体每点法向量的时候，变为从 normal map 中查找，normal map 中存储的只是向量三个维度上的长度，对于平面而言每一点的含法向量的正交基是一样的，而对于球则是不一样的。对于球面一点的正交基的计算方法为：首先取径向方向为一个基底，交点与球的最高点连线向量和径向方向叉积得到第二个基底，前两个基底叉积得到第三个。

附上关键代码和上述纹理的高度图：

```
Material::Material(string imgfile, Vec3d _color) :type(Diffuse), color(
    _color)
{
  Mat bump = cv::imread(imgfile,0);
  size = std::min(bump.rows, bump.cols);
  normalMap = new Mat(size, size, CV_64FC3);
  Vec3d *ptr3 = &normalMap->at<Vec3d>(0, 0);
  for (int i = 0; i < size; i++)
  {
    for (int j = 0; j < size; j++)
    {
      if (j != 0 && j != size - 1)
      {
        ptr3->val[0] = -(bump.at<uchar>(i, j - 1) - bump.at<uchar>(i, j + 1)
    )/bumpScale1;
      }
      else if (j == 0)
      {
        ptr3->val[0] = -(bump.at<uchar>(i, j) - bump.at<uchar>(i, j + 1))/
    bumpScale2;
      }
      else
```

```
20        {
21          ptr3->val[0] = -(bump.at<uchar>(i, j - 1) - bump.at<uchar>(i, j)) /
      bumpScale2;
22        }
23        if (i != 0 && i != size - 1)
24        {
25          ptr3->val[1] = -(bump.at<uchar>(i - 1, j) - bump.at<uchar>(i + 1, j)
      )/bumpScale1;
26        }
27        else if (i == 0)
28        {
29          ptr3->val[1] = -(bump.at<uchar>(i, j) - bump.at<uchar>(i + 1, j)) /
      bumpScale2;
30        }
31        else
32        {
33          ptr3->val[1] = -(bump.at<uchar>(i - 1, j) - bump.at<uchar>(i, j))/
      bumpScale2;
34        }
35        ptr3->val[2] = 1;
36        normalize(*ptr3);
37        ptr3++;
38      }
39    }
40  }
41
42  Vec3d Material::getNormal(double u, double v)
43  {
44    return normalMap->at<Vec3d>(u*size -0.5, v*size -0.5);
45  }
46
47  Vec3d Sphere::computeBumpNormal(const Point3d& hitPoint)
48  {
49    Vec3d axesX, axesY, axesZ;
50    axesZ = normalize(hitPoint - center);
51    axesX = normalize(hitPoint - top).cross(axesZ);
52    axesY = axesZ.cross(axesY);
53    double u, v;
54    computeTextureCoor(hitPoint, u, v);
55    Vec3d coor = material.getNormal(u, v);
56    return coor.val[0] * axesX + coor.val[1] * axesY + coor.val[2] * axesZ;
57  }
```

2.4 加速

kdtree: 堆式建树，不用存指针节省内存，需要小心地计算好两颗子树的大小，好在不是很复杂。findNearestNeighbours 用堆来保存找到的光子，平衡插入和最小距离的更新。

```
void PhotonMap::construction(vector<Photon>::iterator beginIt, vector<Photon
    >::iterator endIt, int layer, int pos)
{
  int treeSize = endIt - beginIt + 1;
  int tsize = 1 << (log2floor(treeSize));
  tsize = treeSize >= tsize + (tsize >> 1) ? tsize : (tsize >> 1);
  int leftTreeSize = std::max(tsize - 1, treeSize - tsize - 1);
  if (endIt - beginIt == 1)
  {
    tree[pos] = *beginIt;
    tree[pos].layer = layer;
    return;
  }
  nth_element(beginIt, beginIt + leftTreeSize, endIt,compare[layer]);
  tree[pos] = *(beginIt + leftTreeSize);
  tree[pos].layer = layer;
  layer = (layer + 1) % 3;
  auto partitionIt = beginIt + leftTreeSize;
  construction(beginIt, partitionIt, layer, 2 * pos);
  if (endIt > partitionIt + 1)
  {
    construction(partitionIt + 1, endIt, layer, 2 * pos + 1);
  }
}

double PhotonMap::findNeighbours(Point3d p, double dist_2)
{
  neighbours.clear();
  poi = p;
  searchDist_2 = dist_2;
  rangeSearch(1);
  if (!neighbours.empty())
    return (*neighbours.begin())->distToPoi_2;
  return 1;
}


void PhotonMap::rangeSearch(int pos)
{
  Photon *p = &tree[pos];
```

```
  if (2 * pos + 1 <= size)
  {
    int layer = p->layer;
    double d = poi.val[layer] - p->pos.val[layer];
    if (d < 0)
    {
      rangeSearch(2 * pos);
      if (d*d < searchDist_2)
      {
        rangeSearch(2 * pos + 1);
      }
    }
    else
    {
      rangeSearch(2 * pos + 1);
      if (d*d < searchDist_2)
      {
        rangeSearch(2 * pos);
      }
    }
  }
  Vec3d t = poi - p->pos;
  p->distToPoi_2 = t.ddot(t);
  if (p->distToPoi_2 < searchDist_2)
  {
    neighbours.push_back(p);
    std::push_heap(neighbours.begin(), neighbours.end(), HeapCompare());
    if (neighbours.size() > estimateNum)
    {
      std::pop_heap(neighbours.begin(), neighbours.end(), HeapCompare());
      neighbours.pop_back();
      searchDist_2 = neighbours[0]->distToPoi_2;
    }
  }
}
```

bezier 求交加速：思路是找一个靠近相机一侧的，又离 bezier 曲面尽可能近的平面，先和这个平面求交。我的方法是：取 bezier 首尾两个控制顶点连线方向作为平面的方向，bezier 曲面处作为起始位置，将该平面向相机挪动一段较长的距离作为结束位置，用二分法找到平面和 bezier 曲面几乎相切的位置，再向相机挪动一个微小距离作为平面的最终位置。

```
void CubicBezierWall::findPlane()
{
```

```
  Point2d start = controlPoints[0];
  Vec2d d;
  Vec2d p = normalize(controlPoints[3] - controlPoints[0]);
  Vec3d td = Vec3d(p.val[0], p.val[1], 0).cross(Vec3d(0, 0, 1));
  d.val[0] = td.val[0];
  d.val[1] = td.val[1];
  Point2d end = start + 50 * d;
  Point2d e;
  for (int i = 0; i < 10; i++)
  {
    e = (start + end) / 2;
    double t = intersect(e, p, NULL);
    if (t > 0 && t < 1)
    {
      start = e;
    }
    else
    {
      end = e;
    }
  }
  e += 0.1*d;
  plane = new Plane(Point3d(e.val[0], e.val[1], 0), Vec3d(d.val[0],d.val
    [1],0),Material());
}
```