Homework 2

计 52 宋世虹 2015011267

2017 年 6 月

1 代码简介

我的代码有以下头文件：

- base.h: 定义了一些全局需要用的函数和变量，如 eps, 点到点的距离，点到线的距离，叉乘等。
- Camera.h: 定义了摄像机类，摄像机存储了图像的 map，并且摄像机可以发出光线来 Query 在某点的 RGB 值。
- Color.h: 定义了一个点的 RGB，并且重载了一些运算符。
- kdtree.h: 定义了一颗 KD 树，用来加速 K 近邻查询。
- Light.h: 定义了一根光线的特征。并且实现了光线的折射和反射。
- LightSource.h: 定义了光源，其中有点光源和面光源，两种光源都有发射光子的功能。
- Object.h: 定义了物体，包括面、盒子、球和 Bezier 曲面。对于每种物体都实现了求交函数，求法向量函数，和加入光子函数。
- Photon.h: 定义了一个光子。
- PhotonMap.h: 定义了每个物体存储光子的 map，这个 map 可以用来计算一个点的亮度。
- Scene.h: 定义了光子映射的过程，包括撒光子和渲染的过程，还有最后的保存过程。
- Description.txt: 这个文件里是对于场景的描述。

2 实验算法

2.1 光子映射

具体实现：光子映射的过程是先撒光子，然后渲染场景。

2.2 撒光子

对于场景中的每一个光源，如果是点光源，就向随机方向发射光子，如果是面光源，就随机一个面上的点，把它当做点光源发射光子。发射光子的过程是，根据这个光子的方向和开始点确定一条光线，然后在 Scene 中使用这条光线和每个物体求交，然后这些物体和它交点离它最近的那个物体，并且在这个物体上记下这个光子，同时修改这个光子的信息，如本身的 RGB，hit 到物体上时光子的方向等等，用于之后渲染计算光强用。如果遇到折射和反射的情况，首先调用 Light.h 中的函数来计算光子下一步的方向，然后继续求交，注意这个时候会带上原物体的颜色。撒光子直到光子全部撒完为止。

2.3 渲染

渲染前先使用 kdtree 对撒了的光子建图，建图后就可以快速查找 k 近邻。对于 Camera 上的每个 pixel 渲染，都是使用和撒光子类似的方法，从相机发出一条光线，然后求交，返回交点的颜色。交点颜色的计算利用的是周围 k 个光子的光亮度乘以他们的颜色除以包围他们最小的球半径的平方乘以 π。由于这里直接使用这样的光亮度，会使得很多东西都暗的看不清楚或者白茫茫的一片，所以我对所有的光亮度先 normalize 了一遍，然后乘了一个很小的系数开方，这会使图片的效果好很多。

2.4 扩展

2.4.1 Bezier 高次曲面求交

这里我使用的是牛顿迭代法。首先先用包围盒判断是否不相交，这个会加速很多。然后对于每条可能相交的光线，将 t,u,v 初始化为任意 0,1 之间的数，进行迭代 20 轮，如果不收敛重新初始化，这里有 5 次重新初始化机会，如果还是不能相交则为不相交，否则即为相交。

实验中实现了 25 个控制点的 Bezier 曲面求交。

代码段 :(在 Object.cpp 里)

```
bool Bezier::isInbox(Light& light)
{
  // printf("%f %f %f\n",BoundingLB[0],BoundingLB[1],BoundingLB[2] );
  // printf("%f %f %f\n",BoundingRT[0],BoundingRT[1],BoundingRT[2] );
  for (int i = 0;i < 3;++i)
  {
    Surface* temp = new Surface(i == 0,i == 1,i == 2,-BoundingLB[i],cv::Vec3f
        (0,0,0));
    Point inter = temp->intersect(light);
    // printf("inter %f %f %f \n",inter[0],inter[1],inter[2]);
    bool thisinter = true;
    for (int j = 0;j < 3;++j)
      if (i != j)
        if (inter[j] >= BoundingLB[j] || inter[j] <= BoundingRT[j])
          thisinter = false;
    if (thisinter) return true;
  }
  for (int i = 0;i < 3;++i)
  {
    Surface* temp = new Surface(i == 0,i == 1,i == 2,-BoundingRT[i],cv::Vec3f
        (0,0,0));
    Point inter = temp->intersect(light);
    bool thisinter = true;
    for (int j = 0;j < 3;++j)
      if (i != j)
        if (inter[j] >= BoundingLB[j] || inter[j] <= BoundingRT[j])
          thisinter = false;
    if (thisinter) return true;
  }
  return false;
}

Point Bezier::intersect(Light& light)
{
  // printf("%f %f %f\n", light.direction.x,light.direction.y,light.direction.z);
  // fflush(stdout);
  bool WillInter = isInbox(light);
  // printf("%d\n", int(WillInter));
  if (!WillInter) return BackgroundPoint;
  // printf("gg\n");
  VectorXd origin(3);
  origin[0] = origin[1] = origin[2] = 0;
  VectorXd nowPoint(3);
```

```
42    bool getAns = false;
43    int tot = 0;
44    while (!getAns && tot < 20)
45    {
46      nowPoint[0] = std::rand() * 1.0/RAND_MAX;
47      nowPoint[1] = std::rand() * 1.0/RAND_MAX;
48      nowPoint[2] = std::rand() * 1.0/RAND_MAX;
49      int count = 0;
50      while (1)
51      {
52        origin = nowPoint;
53        Matrix3d derive;
54        for (int i = 0;i < 3;++i)
55          for (int j = 0;j < 3;++j)
56            derive(i,j) = getDerivedF(i,j,origin,light);
57        // std::cout << derive << std::endl;
58        Matrix3d inverse;
59        bool invertible;
60        double determinant;
61        derive.computeInverseAndDetWithCheck(inverse,determinant,invertible);
62        // std::cout << inverse << std::endl;
63        if(invertible)
64        {
65          VectorXd value(3);
66          value[0] = light.beginPoint.x + light.direction.x * nowPoint[0];
67          for (int i = 0;i <= n;++i)
68            for (int j = 0;j <= m;++j)
69              value[0] -= P[i][j].x * getBezier(i,n,nowPoint[1]) * getBezier(j,m,
                    nowPoint[2]);
70          value[1] = light.beginPoint.y + light.direction.y * nowPoint[0];
71          for (int i = 0;i <= n;++i)
72            for (int j = 0;j <= m;++j)
73              value[1] -= P[i][j].y * getBezier(i,n,nowPoint[1]) * getBezier(j,m,
                    nowPoint[2]);
74          value[2] = light.beginPoint.z + light.direction.z * nowPoint[0];
75          for (int i = 0;i <= n;++i)
76            for (int j = 0;j <= m;++j)
77              value[2] -= P[i][j].z * getBezier(i,n,nowPoint[1]) * getBezier(j,m,
                    nowPoint[2]);
78          // std::cout << value << std::endl;
79          // printf("%f %f %f\n", nowPoint[0],nowPoint[1],nowPoint[2]);
80          if ((pow(value[0],2) + pow(value[1],2) + pow(value[2],2)) <= eps)
81          {
82            getAns = true;
83            break;
84          }
85          nowPoint = nowPoint - inverse * value;
86          count++;
87          if (count > 20) break;
88          // printf("%f %f %f\n", nowPoint[0],nowPoint[1],nowPoint[2]);
89        }
90        else
91        {
92          // printf("invertible\n");
93          break;
94        }
95      }
96      if (nowPoint[0] > eps && nowPoint[1] > eps && nowPoint[1] < 1 && nowPoint[2] >
            eps && nowPoint[2] < 1 && getAns)
97        break;
98      else getAns = false;
99      tot++;
100   }
```

```
101   if (nowPoint[0] > eps && nowPoint[1] > eps && nowPoint[1] < 1 && nowPoint[2] >
          eps && nowPoint[2] < 1 && getAns)
102   {
103     // printf("%f %f %f\n", nowPoint[0],nowPoint[1],nowPoint[2]);
104     return Point(light.beginPoint.x + nowPoint[0] * light.direction.x,light.
          beginPoint.y + nowPoint[0] * light.direction.y,light.beginPoint.z + nowPoint
          [0] * light.direction.z,nowPoint[1],nowPoint[2]);
105   }
106   else return BackgroundPoint;
107 }
108
109 double Bezier::getBezier(int i,int m,double u)
110 {
111   if (i < 0) return 0;
112   if (u < eps || u > 1-eps) return 0;
113   return C[m][i] * pow(u,i) * pow(1-u,m-i);
114 }
115
116 double Bezier::getBezierDerive(int i,int m,double u)
117 {
118   return m * (getBezier(i-1,m-1,u) - getBezier(i,m-1,u));
119 }
120
121 double Bezier::getDerivedF(int i,int j,VectorXd& origin,Light& light)
122 {
123   if (j == 0)
124   {
125     return light.direction[i];
126   }
127   else if (j == 1)
128   {
129     double temp = 0;
130     for (int p = 0;p <= n;++p)
131       for (int q = 0;q <= m;++q)
132         temp += P[p][q][i] * getBezierDerive(p,n,origin[1]) * getBezier(q,m,origin
              [2]);
133     return -temp;
134   }
135   else
136   {
137     double temp = 0;
138     for (int p = 0;p <= n;++p)
139       for (int q = 0;q <= m;++q)
140         temp += P[p][q][i] * getBezier(p,n,origin[1]) * getBezierDerive(q,m,origin
              [2]);
141     return -temp;
142   }
143 }
```

2.4.2 贴图/景深

实现了给 Bezier 曲面贴图和整体的景深。

贴图：对于 Bezier 曲面，使用它的 u,v 贴图，即对于每一个交点，返回它在这个交点的某个图上的 RGB。具体的做法是建立 u,v 到这个图的映射。

景深：对于相机发出的光线进行重采样，如果已知焦平面的位置，那么焦平面上的物体是一定能被看清的。这样我们就可以每次要想知道一个相机像素上一点的值，我们只需要从相机先发出一条光线和焦平面相交，然后对于这个交点，再从相机周围一个圆上随机采样一些点来计算新的 RGB 值，对这些算出的 RGB 重新加权得到真实的 RGB。

贴图代码片段：(在 Object.cpp 中)

```
Color Bezier::colorAt(Point& point)
{
  int col = wangzai.cols * (1-point.u);
  int row = wangzai.rows * (1-point.v);
  if (col == wangzai.cols || col == wangzai.cols-1) col = wangzai.cols - 2;
  if (row == wangzai.rows || row == wangzai.rows-1) row = wangzai.rows - 2;
  return Color(wangzai.at<cv::Vec3b>(row,col)[0] * 1.0/256,wangzai.at<cv::Vec3b>(
      row,col)[1] * 1.0/256,wangzai.at<cv::Vec3b>(row,col)[2] * 1.0/256) ;
}
```

景深代码片段 :(在 Camera.cpp 和 Scene.cpp 中)

```
std::vector<Light> Camera::getLight(int x,int y)
{
  bool jingshen = false;
  double fl = 2.5;
  std::vector<Light> result;
  Point direction(tan( (x - position.x -(width/2)) * fov_w / width) , tan( ((height
      /2)-position.y-y) * fov_h / height ),1);
  direction = direction.normz();
  direction = fl * direction;
  // printf("%f %f %f\n", direction.x, direction.y, direction.z );
  Point beginPoint = position;
  result.push_back(Light(beginPoint,direction));
  if (jingshen)
  {
    double sanfenzhipi = pi/3;
    for (int i = 0;i < 6;++i)
    {
      double x1 = 0.1 * sin(sanfenzhipi * i), x2 = 0.1 * cos(sanfenzhipi * i);
      beginPoint = Point(x1,x2,0);
      Point nowDir = direction - beginPoint;
      result.push_back(Light(beginPoint,nowDir));
    }
  }
  return result;
}
for (int i = 0;i < camera->rows;++i)
    for (int j = 0;j < camera->cols;++j)
    {
      printf("%d\r",i);
      fflush(stdout);
      std::vector<Light> lights = camera->getLight(i,j);
      double right = 1.0 / lights.size();
      camera->image[i][j] = Color(0,0,0);
      for (int k = 0;k < lights.size(); ++k)
        camera->image[i][j] += right * getPointColor(lights[k],1);
    }
```

2.4.3 渲染加速

主要由求包围盒和 kd 树加速。

包围盒针对 Bezier 曲线使用，即取 Bezier 控制点的 x,y,z 边界建立包围盒，对于每一条光线，先判断是否和这个包围盒相交，如果不相交则一定不会和 Bezier 曲面相交，否则有可能和 Bezier 曲面相交，则继续用牛顿迭代法寻找交点。

kd 树加速，即对于光子建立 kd 树，每次查找 k 近邻的时候可以做到约根号光子数的复杂度。代码见 kdtree.h。

包围盒代码 :(见 Object.cpp)

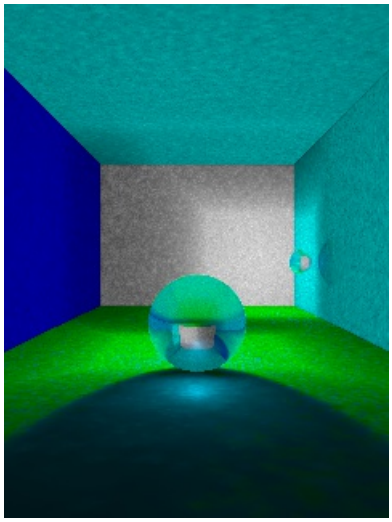
```
BoundingLB.x = -INT_MAX;
```

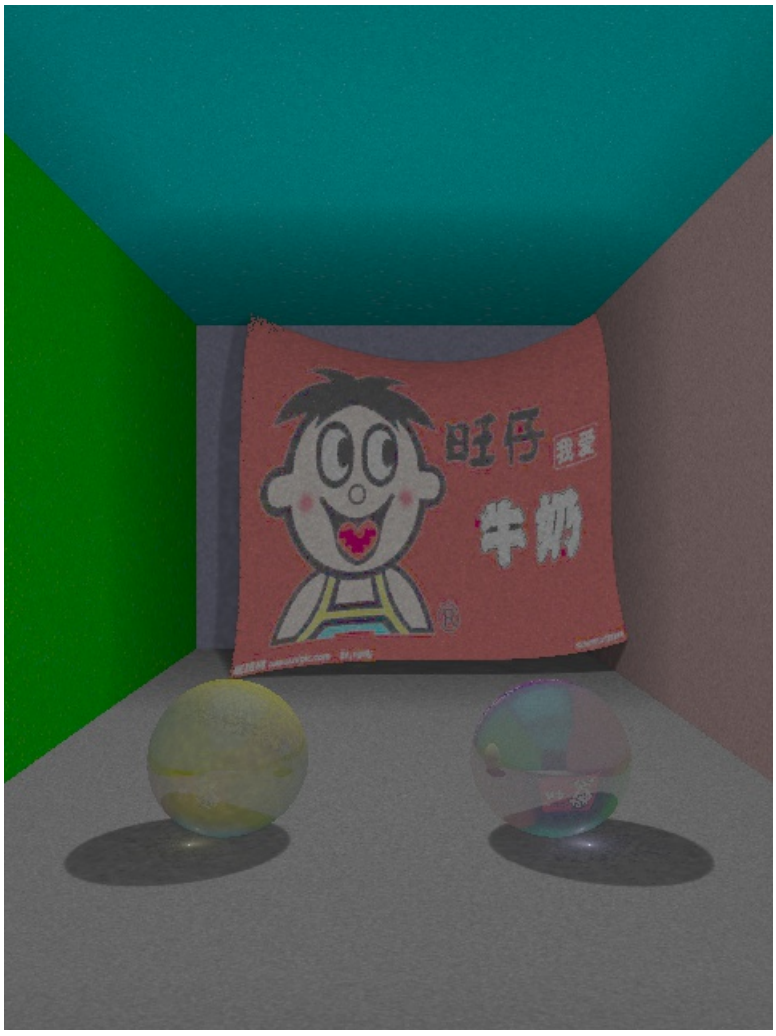
```
  BoundingLB.y = -INT_MAX;
  BoundingLB.z = -INT_MAX;
  BoundingRT.x = INT_MAX;
  BoundingRT.y = INT_MAX;
  BoundingRT.z = INT_MAX;
  for (int i = 0;i <= n;++i)
  {
    std::vector<Point > tempvec;
    P.push_back(tempvec);
    for (int j = 0;j <= m;++j)
    {
      double xx,yy,zz;
      fin >> temp;
      xx = atof(temp.c_str());
      fin >> temp;
      yy = atof(temp.c_str());
      fin >> temp;
      zz = atof(temp.c_str());
      this->P[i].push_back(Point(xx,yy,zz));
      if (xx < BoundingRT.x)
        BoundingRT.x = xx;
      if (yy < BoundingRT.y)
        BoundingRT.y = yy;
      if (zz < BoundingRT.z)
        BoundingRT.z = zz;
      if (xx > BoundingLB.x)
        BoundingLB.x = xx;
      if (yy > BoundingLB.y)
        BoundingLB.y = yy;
      if (zz > BoundingLB.z)
        BoundingLB.z = zz;
    }
  }
bool Bezier::isInbox(Light& light)
{
  // printf("%f %f %f\n",BoundingLB[0],BoundingLB[1],BoundingLB[2] );
  // printf("%f %f %f\n",BoundingRT[0],BoundingRT[1],BoundingRT[2] );
  for (int i = 0;i < 3;++i)
  {
    Surface* temp = new Surface(i == 0,i == 1,i == 2,-BoundingLB[i],cv::Vec3f
        (0,0,0));
    Point inter = temp->intersect(light);
    // printf("inter %f %f %f \n",inter[0],inter[1],inter[2]);
    bool thisinter = true;
    for (int j = 0;j < 3;++j)
      if (i != j)
        if (inter[j] >= BoundingLB[j] || inter[j] <= BoundingRT[j])
          thisinter = false;
    if (thisinter) return true;
  }
  for (int i = 0;i < 3;++i)
  {
    Surface* temp = new Surface(i == 0,i == 1,i == 2,-BoundingRT[i],cv::Vec3f
        (0,0,0));
    Point inter = temp->intersect(light);
    bool thisinter = true;
    for (int j = 0;j < 3;++j)
      if (i != j)
        if (inter[j] >= BoundingLB[j] || inter[j] <= BoundingRT[j])
          thisinter = false;
    if (thisinter) return true;
  }
  return false;
```

```
63 }
```

3 实验效果

3.1 焦散和次表面折射

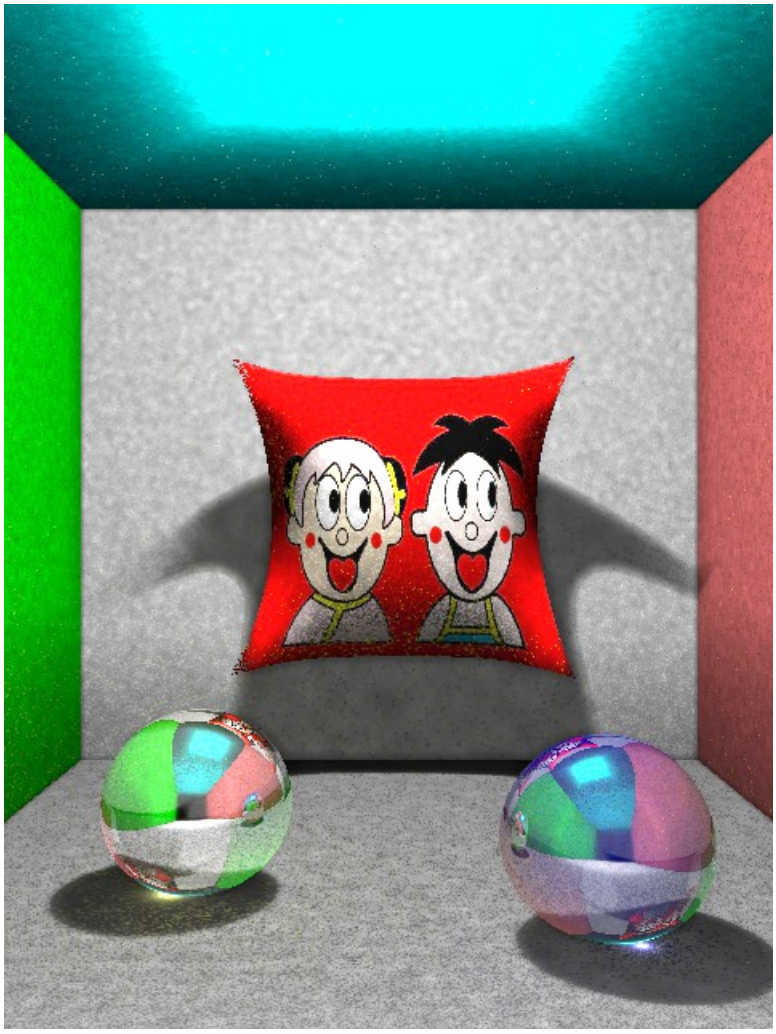




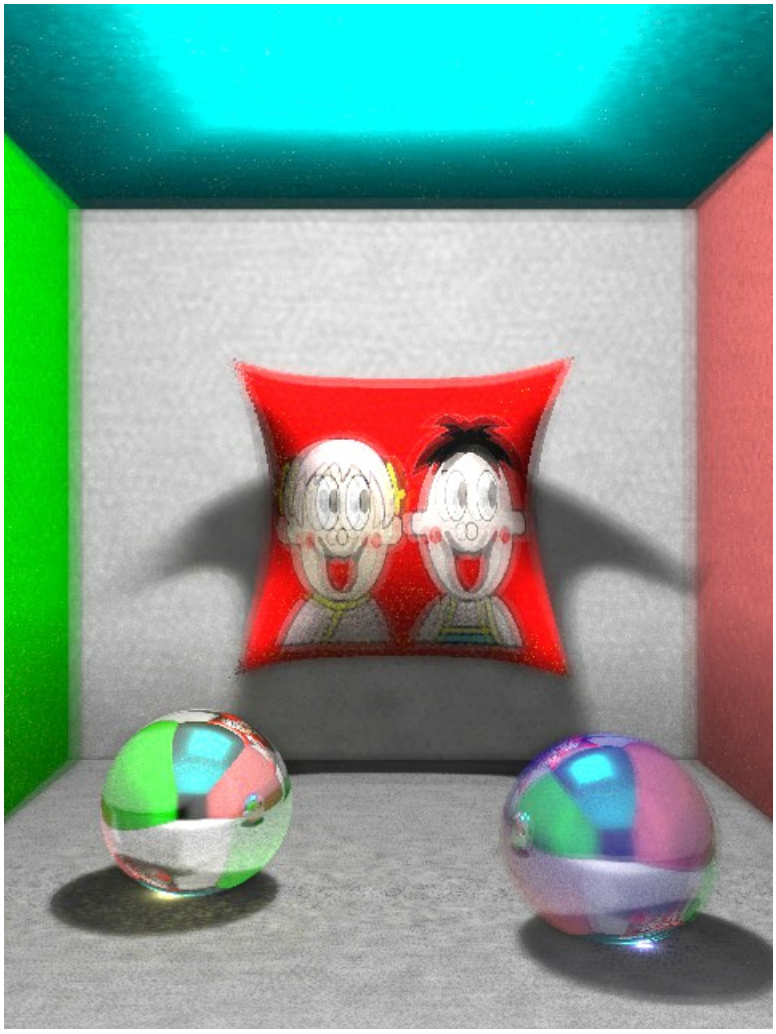
3.2 贴图和景深

我尝试了集中景深的半径取值，后来发现取 0.1 好像效果比较好。贴图如图。

半径取值 0.01，几乎看不出景深：



半径取值 0.1，效果显著：



3.3 另一构景