基于 Bezier 曲线的三维造型与渲染作业报告

计 52 周京汉 2015011245

2017.06.25

一、作业要求

本作业的要求是基于 Bezier 曲线的三位造型与渲染。建立一个三维的场景，中间有一些物体和光源，通过光线追踪等方法计算出每个从相机出看去的像素的颜色，最终渲染出一副图片。

二、绘制算法的实现

1、写到的算法

在本次作业之中，我总共实现了一下这些算法：通过光子映射的方法来计算光在场景之中的反射，折射等情况，来计算场景中物体的颜色，一共发射了 3000000 个光子；通过 16 个控制点，控制一个 3 次的 bezier 曲面，并完成它在空间之中的求交，使用了 3 次牛顿迭代法的方法来完成它的求交。

在构图的场景中，我建立了三维的坐标系，来记录这些物品的坐标，光线方向及法向量。场景建立为一个康奈尔 box。光源为点光源，位于 box 之内。相机点为了使场景中的球看起来更圆，采取了远离场景的方法，因此相机点在 box 之外。

除去以上基本项目，我还做了以下附加项：抗锯齿，普通的纹理贴图和改变法向量的凹凸贴图。

2、程序的基本架构

程序的基本构架主要分为三部分：场景建模部分，物品部分和光线追踪（光子映射）部分。

第一部分主要就是场景中需要的坐标 vector3 的编写，场景的相机的设定，光线的设定和对于场景转回为像素的时候的画图的函数。主要包含头文件：Point.hpp,Camera.hpp,Ray.hpp 和 Drawing.hpp。

第二部分主要是用来存储将会出现在场景之中的物品和它们的参数和一些函数。场景中的物品主要为 3 类：球类，面类和 Bezier 曲面类。它们三类都是继承 Object 类，并重构了其中的虚函数。在三大类的函数中分别有它们的求交函数，其中 Bezier 曲面应用了 3 次牛顿迭代的方法来进行求交。代码如下：

```
bool Beizer_Surface::NewtonIteration(Ray input_ray, double& t, double& u, double& v)
{
    double delta_t = 0, delta_u = 0, delta_v = 0;
    for (int i = 0; i < 60; i++)
    {
        //vector3<double> beizer_point = get_point(u,v);

        vector3<double> surface_partial_du = getdpdu(u,v);
        vector3<double> surface_partial_dv = getdpdv(u,v);

        vector3<double> f_value = input_ray.start_point + input_ray.direction *
                    t - get_point(u, v);

        vector3<double> tmp = surface_partial_du.cross_product(surface_partial_dv);
        double D = input_ray.direction * tmp;

        if (abs(D) < limit_zero)
            return false;

        tmp = surface_partial_dv.cross_product(f_value);
        delta_t = surface_partial_du * tmp / D;

        tmp = surface_partial_dv.cross_product(f_value);
        delta_u = input_ray.direction * tmp / D;

        tmp = surface_partial_du.cross_product(f_value);
        delta_v = input_ray.direction * tmp / D;

        t -= delta_t*0.45;
        u -= delta_u*0.45;
        v += delta_v*0.45;

        if (u < 0 || u > 1 || v < 0 || v > 1)
            return false;

        //cout << t << " " << u << " " << v << endl;
```

```
    }
    //cout << "~~~~~~~~~~" << endl;
    if (abs(delta_t) > solve_precision)          //认为迭代时迭飞了，没有交点
    {
        return false;
    }
    else
    {
        if (u < limit_zero || u > 1 - limit_zero || v < limit_zero ||
                        v > 1 - limit_zero || t < limit_zero)
        {
            return false;
        }
        //cout << "uv: " << u << " " << v << endl;
        return true;
    }
}
```

在经过反复的调试之后我最终将迭代次数定为 60 次，每次的步长为原来的 0.45 倍。并且在每次求交的时候，对其中的参数 u 和 v 进行随机取值，并且进行 3 次计算来保证最终可以取到合适的初值来求解得出最终的答案。

第二部分中主要的头文件包括：Objects.hpp,Beizer_Surface.hpp,Plane.hpp 和 Sphere.hpp。

第三部分为程序中最主要的计算部分，为计算光线追踪和光子映射的部分。主要思路为向空间中随机播撒光子，根据求交结果计算光子在空间中弹的时候的可能发生的情况。使用轮盘赌的方法来决定是发生哪种的情况。然后再根据求交的情况得到的法向量等参数进行 Phong 模型的计算，得出最后每一个像素点的颜色的情况。最终包含的头文件包括：计算场景中的情况的 World.hpp, 计算 Phong 模型结果的 PhongModel.hpp，进行光子映射的 PhotonModel.hpp,kdTree.hpp 和 Photon_map.hpp。

三、附加项的描述

1、抗锯齿

抗锯齿部分我使用的方法是先渲染出一个相对较大的图，然后在输出图片的时候对大图进行采样，4 个像素一组，用取平均值的方法将其合成为一个像素。然后用新的像素组成一个新的图。这样可以使得像素之间颜色的变化变得缓和，而不再剧烈，来达到抗锯齿的效果。

程序中的代码:

```
bool Drawer::output_image()
{
    cv::imwrite("/Users/mac/Desktop/programme/program/4st_term/
␣␣␣␣␣␣␣␣Ray_Tracing/test_1.bmp", images);


    cv::Mat small_images(images.rows/2,images.cols/2,CV_8UC3);//缩小，抗锯齿
    for(int i = 0; i < small_images.rows; i++)
        for(int j = 0; j < small_images.cols; j++)
        {
            small_images.at<cv::Vec3b>(i,j)[0] = (images.at<cv::Vec3b>(i*2,j*2)
                [0]+images.at<cv::Vec3b>(i*2+1,j*2)[0]+images.at<cv::Vec3b>
                (i*2,j*2+1)[0]+images.at<cv::Vec3b>(i*2+1,j*2+1)[0])/4;
            small_images.at<cv::Vec3b>(i,j)[1] = (images.at<cv::Vec3b>(i*2,j*2)
                [1]+images.at<cv::Vec3b>(i*2+1,j*2)[1]+images.at<cv::Vec3b>
                (i*2,j*2+1)[1]+images.at<cv::Vec3b>(i*2+1,j*2+1)[1])/4;
            small_images.at<cv::Vec3b>(i,j)[2] = (images.at<cv::Vec3b>(i*2,j*2)
                [2]+images.at<cv::Vec3b>(i*2+1,j*2)[2]+images.at<cv::Vec3b>
                (i*2,j*2+1)[2]+images.at<cv::Vec3b>(i*2+1,j*2+1)[2])/4;
        }
    cv::imwrite("/Users/mac/Desktop/programme/program/4st_term/
␣␣␣␣␣␣␣␣Ray_Tracing/small_one.bmp", small_images);

    return true;
}
```

抗锯齿的效果较好，可以通过最终提交的两张同样大小的图对比看出效果。缺点就是耗费时间较长。

2、纹理贴图

纹理贴图部分的原理是，通过建立被贴的面和贴图的像素的一一对应，来讲贴图的颜色加到要贴的面上。以 Bezier 曲面的贴图为例，代码如下:

```
object_feature feature1;//开始普通贴图
feature1.absorb = 0.5;
feature1.diffuse_reflect = 0.5;

int yyy = last_u*200, zzz = last_v*200;
```

```
feature1.reflect_blue = (double)image2.at<cv::Vec3b>(yyy,zzz)[0] / (double)255;
feature1.reflect_green = (double)image2.at<cv::Vec3b>(yyy,zzz)[1] / (double)255;
feature1.reflect_red = (double)image2.at<cv::Vec3b>(yyy,zzz)[2] / (double)255;

return PhongModel::reflect_color(light, in, view_direction, feature1);
```

代码中的 image2 即为即将被贴上去的图片。当前像素点在曲面上的位置即为 u 和 v 两个参数表示。用 yyy 和 zzz 两个变量作为在图片中像素的坐标与 u 和 v 建立映射关系，然后 u 和 v 表示的位置的颜色就可以用其对应的图片当中的像素的颜色进行替换，最终达到贴图的效果。最终成果图中的贴图有：地面的地板贴图和曲面上的花纹贴图，贴图的效果完成的较好。

3、凹凸贴图

凹凸贴图的原理是根据输入的高度图，将输入的图中的代表三个方向的向量的 RGB 转化为当前自己的法向量。首先认为面的法向量为 (0,0,1) 向量，所以其 z 坐标要始终大于 0 才合理。因此，在计算 RGB 所代表的法向量的时候，要将 B 转化为 0-1 的数，而 RG 转化为-1-1 的数，计算出这个点应该有的法向量在现在设定的这个向量空间内的向量表达。然后，根据面所表示的 u 和 v 参数，计算出向量空间中的另外两个向量。

以天花板上面贴的凹凸纹理为例子，代码为：

```
if(B == 1 && target_pos.x > -10.0 && target_pos.x < 10.0 &&
    target_pos.z > -10.0 && target_pos.z < 10.0)//贴天花板
{
    double xx = (target_pos.x+10)/20, zz = (target_pos.z+10)/20;
    int xp = xx*500, zp = zz*500;

    vector3<double> qie_x, qie_y;
    qie_y = vector3<double>(1,0,0);
    qie_x = vector3<double>(0,0,1);
    double r_gradiant, g_gradiant, b_gradiant;
    r_gradiant = ((double)image6.at<cv::Vec3b>(zp,xp)[2]/(double)255 - 0.5)*2;
    g_gradiant = ((double)image6.at<cv::Vec3b>(zp,xp)[1]/(double)255 - 0.5)*2;
    b_gradiant = ((double)image6.at<cv::Vec3b>(zp,xp)[0]-128) / (double)128;

    in = vector3<double>(qie_x.x*r_gradiant+qie_y.x*g_gradiant+
            normal_vector.x*b_gradiant, qie_x.y*r_gradiant+qie_y.y*g_gradiant
            +normal_vector.y*b_gradiant, qie_x.z*r_gradiant+qie_y.z*g_gradiant
            +normal_vector.z*b_gradiant);
    in = in.normallize();
```

```
        return PhongModel::reflect_color(light, in, view_direction, feature);
    }
```

在计算的时候，将 RGB 转化出来的法向量方向与原来的三个方向的 x,y,z 分别进行乘法然后加法的运算，得出新的法向量的方向的三个值，最后在进行标准化，即为新的法向量，并输入到 Phong 模型当中进行计算。最终凹凸贴图完成的有天花板的纹理贴图和右边墙壁上面的砖纹理的贴图。贴图效果较好。

四、最终效果

最终的效果场景为:

左边的墙壁为带有反射的粉色墙壁，可以看见 Bezier 曲面的全部形状。右边墙壁和天花板为凹凸贴图，分别为砖墙和不规则的纹理。地板为普通贴图，贴了地板的图，正面的墙上贴图为一个滑稽。

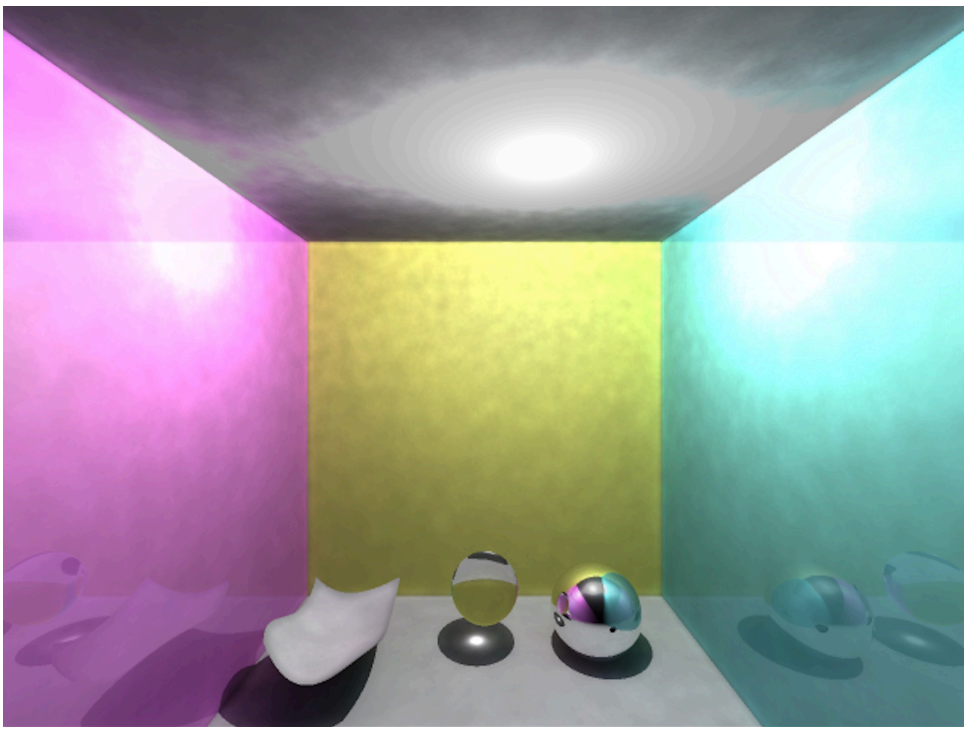
下面摆放着三个物体，分别为：带有花纹的三次 Bezier 曲面，只带有折射的下面有焦散效果的玻璃球和只带有反射的稍大一点的玻璃球。

应之前助教的要求，增强了自己的 Bezier 曲面，并增强了自己的求交算法。应用了 3 次牛顿迭代计算，只要一次计算为 True 即为求交。每次牛顿迭代要进行 60 次，每次迭代的变化量都要进行乘 0.45 的处理，来减小迭代的步长，这是为了防止在迭代的过程中，由于初值设置的不利，而直接出现中间的 t，u 或者 v 直接超出限制范围从而直接迭代失败的情况。在每一次迭代后，我都会进行判定，将迭代超出数据范围的情况直

接停止，并返回 False。

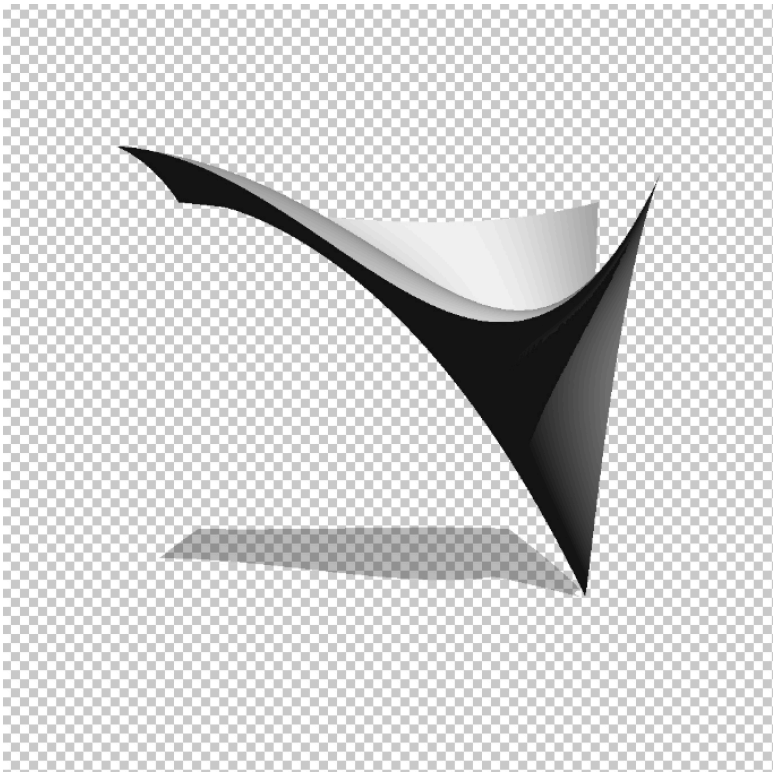
在最终的结果之中，按照与助教讨论时候助教的要求，我将之前的曲面进行了翻转，来提高其求交的难度。效果图未加抗锯齿效果。（图中左下角的曲面）

曲面翻转后的样子：



当然这个曲面的求交难度远不如现在的图中的曲面。

两个曲面的所分别对应的 obj 文件生成的网格的截图为：



在图像之中可能因为视角问题使曲面的图形被拉长了，但是总体来说是完整的表现出了曲面。

五、总结

总的来说这次的图形学大作业对于我来说是一个很大的挑战。在最初的时候我对于如何建立一个三维的模型是一无所知的，在经过数天的学习与摸索之后才逐渐入门。从中，我收获了从零开始完成一项大作业的宝贵经验。最终我完成这个作业所花的时间为 8 天，包括 5 天的光线追踪 + 光子映射，1 天完成了 Bezier 曲面的 obj 生成和求交公式的代码编写，以及最后 2 天的参数调整和普通贴图与凹凸贴图的书写。