

真实感图像渲染大作业 实验报告

杨力忱 2020011007

1 项目简介

1.1 已实现的效果

选用路径追踪 (Path Tracing) 算法，实现了一个简单的光线追踪渲染引擎。在光线的传播上，实现了漫反射、镜面反射、折射三种材质，而且物体表面的光线传播性质可以由三者混合。支持的几何体有球体、平面、旋转曲面、导入的三角网格模型，并可对它们进行线性变换改变大小/方向/位置。基于牛顿迭代法实现了 Bezier 和 B 样条旋转曲面的解析求交。

实现了景深效果，可以调节焦距和光圈大小；实现了软阴影，这是路径追踪算法的特性；通过在采样时增加随机扰动进行边缘抗锯齿；通过UV展开实现了球体和旋转曲面的贴图；对三角网格物体实现了法向插值平滑处理。

算法效率方面，对复杂网格和旋转曲面实现了轴对齐包围盒 (Axis-aligned bounding box) 加速，硬件上利用 OpenMP 从程序上充分利用 CPU 的多个线程进行加速。

1.2 项目框架

本次作业在 PA1 的代码框架的基础上进行实现，文件结构和它比较接近；编译使用 Makefile，在项目根目录下运行 make 即可编译，在 Windows 系统或 Linux 系统下都可以成功编译运行。实质上编译命令如下：

```
g++ src/image.cpp src/main.cpp src/mesh.cpp src/scene_parser.cpp src/texture.cpp  
src/vecmath.cpp -o main -O3 -Wall -fopenmp -I include/
```

代码框架介绍如下：

- include/: 所有的 .h 或 .hpp 文件，包含了各个物体和对象的类的定义，以及算法主逻辑。
- mesh/: .obj 格式的三角网格模型文件资源
- output/: 输出图片的存放目录
- src/: 所有的 .cpp 文件，包含了主函数代码以及图片存储、场景文件读取、三角网格模型文件和贴图文件的读取、PA1 提供的三维向量运算库的相关代码。
- testcases/: 生成结果需要的场景描述文件
- texture/: 存放贴图资源

程序使用方法：`./main <input scene file> <output bmp file> <method> <spp>`

其中 method 指程序使用的算法，可以是 rc (光线投射) 或者是 pt (路径追踪)。

用法示例：`./main testcases/test01.txt output/test01.bmp pt 15000`

2 算法与实现

2.1 路径追踪

路径追踪的算法思路大致为：从相机经由指定像素发射射线，如果未能击中物体，则将场景的背景色作为该像素的颜色。一般情况下，该射线在场景中不断被物体表面进行反射和折射。如果能够击中光源，则将光源发光强度乘上一路经过的物体的的反射颜色（衰减率），作为该光源对像素颜色的贡献；否则当超过预先设置的迭代深度后，则停止迭代并将之前所有积累的光照强度算作该像素的颜色。对每个像素进行多次采样（采样率），把每次计算得到的结果去平均值作为这一像素最终的颜色值。

路径追踪算法的代码主要位于render.hpp中，其主要逻辑部分如下：

```
void render() // 主函数在读取相关信息后调用
{
    PerspectiveCamera *camera = (PerspectiveCamera *)sceneParser.getCamera();
    int w = camera->getWidth(), h = camera->getHeight();
    Image renderedImage(w, h);
#pragma omp parallel for schedule(dynamic, 1) // OpenMP
    for (int x = 0; x < w; ++x) {
        unsigned short Xi[3] = {0, 0, (unsigned short)(x * x * x)};
        for (int y = 0; y < h; ++y) {
            Vector3f avg_color = Vector3f::ZERO;
            for (int s = 0; s < samps; ++s) { // samps: 采样率
                Ray camRay = camera->generateRay(Vector2f(x + erand48(Xi) - 0.5,
y + erand48(Xi) - 0.5)); // 生成射线
                avg_color += Color_func(camRay, sceneParser, Xi); // 计算对应的颜色
                值
            }
            renderedImage.SetPixel(x, y, avg_color / samps); // 取平均，确定该像素颜
            色值
        }
    }
    renderedImage.SaveImage(fout);
}
```

2.2 光线的传播

镜面反射光线方向 $R = I - 2(N \cdot I)N$ ，其中 I 是入射光线， N 是交点的法向。对于漫反射表面，反射光线在交点对应的半球面上随机选取。根据 Snell 定律和 Fresnel 效应，折射光线的计算方法如下，代码也位于render.hpp中：

```
float n = material->refr;
float R0 = ((1.0 - n) * (1.0 - n)) / ((1.0 + n) * (1.0 + n));
if (Vector3f::dot(N, ray.direction) > 0)
{ // inside the medium
    N.negate();
```

```

    n = 1 / n;
}

n = 1 / n;

float cost1 = -Vector3f::dot(N, ray.direction);
float cost2 = 1.0 - n * n * (1.0 - cost1 * cost1);
float Rprob = R0 + (1.0 - R0) * pow(1.0 - cost1, 5.0);
if (cost2 > 0 && erand48(Xi) > Rprob)
{ // refraction direction
    ray.direction = ((ray.direction * n) + (N * (n * cost1 -
sqrt(cost2)))).normalized();
}
else
{ // reflection direction
    ray.direction = (ray.direction + N * (cost1 * 2)).normalized();
}

```

可以看到，对于纯折射材质，在不同角度下也有不同比例的镜面反射。

一个物体的表面材质可以由漫反射、镜面反射、折射三者以不同比例混合，在计算时按比例随机选取传播方式，当采样率足够多的时候就可以达到期望的效果。

2.3 光线与物体的求交

射线与平面、球体、三角面片、三角网格的求交在PA1中已经实现，可以沿用。对于旋转曲面，我实现了牛顿法求 Bezier 曲面与射线交点的算法，代码位于 `revsurface.hpp` 中。

3 实现效果

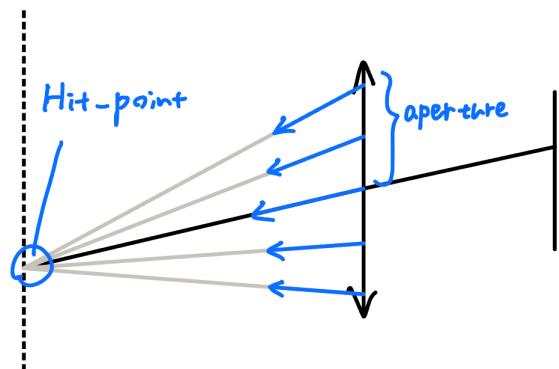
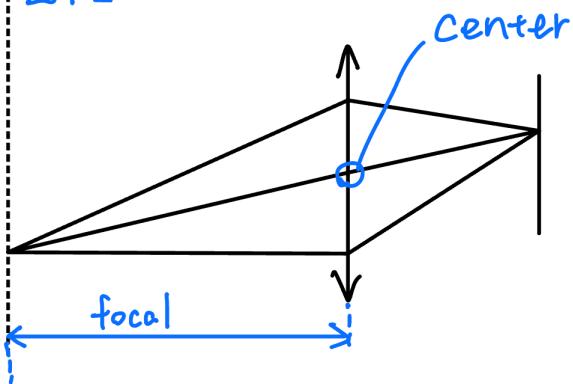
3.1 景深

现在的 `PerspectiveCamera` 的原理是小孔成像：从每一个像素点发出的射线是固定的，与场景中物体上的点一一对应，这样渲染出来的图像总是清晰的。

首先考虑一个有一定大小的光圈的、用凸透镜成像的相机的原理。对于这种相机，只有焦平面上的点发出的光才能够在感光面上汇聚到形成清晰的像，而过近过远的物体成的像都是模糊的（即景深效果）。焦平面与相机的凸透镜和感光平面都平行。

如下图左侧，`PerspectiveCamera` 的中心 `center` 就是凸透镜的光心。为了实现景深效果，我为之前的相机类加了两个变量：`focal`，相机中心到焦平面的距离；`aperture`，光圈半径。在路径追踪算法生成射线时，首先还是穿过 `center` 发射一条射线找到与焦平面的交点 `Hit_point`，随后将光线的出发点在半径为 `aperture` 的圆盘上随机选取，光线的方向为新的出发点与 `Hit_point` 的连线。原理示意图和代码如下，代码位于 `camera.hpp` 中。

焦平面



```
// 已经计算得到穿过光心的射线方向 Ray_direction. 相机朝向为 direction
float t = focal / (Vector3f::dot(Ray_direction, this->direction));
Vector3f Hit_point = center + Ray_direction * t;
Vector3f random_center = center + ((erand48() - 0.5) * aperture) * up +
((erand48() - 0.5) * aperture) * horizontal;
Vector3f random_dir = (Hit_point - random_center).normalized();
return Ray(random_center, random_dir);
```

下图是景深和贴图的效果图。焦平面位于地球的位置，只有地球是清晰的。



3.2 软阴影

Path Tracing 算法渲染出来的图自带软阴影，这是算法的特性。

3.3 抗锯齿

在对一个像素进行多次采样时，对它在 $[-0.5, 0.5] \times [-0.5, 0.5]$ 的范围内进行随机扰动。即原来每次采样从相机发出的射线都是穿过像素中心的同一条射线，现在射线会等概率地穿过像素所覆盖的范围中的点，得到的像素颜色为像素所覆盖区域内的平均颜色。这样物体的边缘会更加柔和，几乎消除了锯齿现象。

代码实现上，在 render.hpp 的射线生成部分加上随机扰动即可：

```

for (int s = 0; s < samps; ++s) {
    Ray camRay = camera->generateRay(Vector2f(x + erand48(Xi) - 0.5, y +
erand48(Xi) - 0.5));
    avg_color += Color_func(camRay, sceneParser, Xi);
}

```

3.4 贴图

贴图本质上在于找到一个物体表面的点与平面直角坐标系中的点的一一对应关系（UV展开），从而曲面上的点的颜色值就是载入的贴图图片的相应的坐标位置 (u, v) 的颜色值。球体和旋转曲面的UV展开分别位于 `sphere.hpp`、`remsurface.hpp` 中，载入的贴图图片存储在 `Material` 类中，得到对应的UV坐标后调用其中的 `Vector3f Material::getColor(float u, float v)` 函数来获取颜色值，这段代码位于 `material.hpp`。

3.5 复杂网格模型&法向插值

三角网格模型的读入和求交原先的代码框架中已经实现。我在此基础上实现了网格模型的法向插值，用较少的三角面片就可以获得比较柔和的曲面效果。

由于 `obj` 模型只提供了顶点坐标和面片信息，我需要计算每个顶点的法向。顶点的法向近似等于包含该顶点的（与之相邻的）各个三角面片的法向的均值，代码位于 `mesh.cpp` 中。在读入模型文件时进行计算，存储在 `Mesh` 类的对象中。

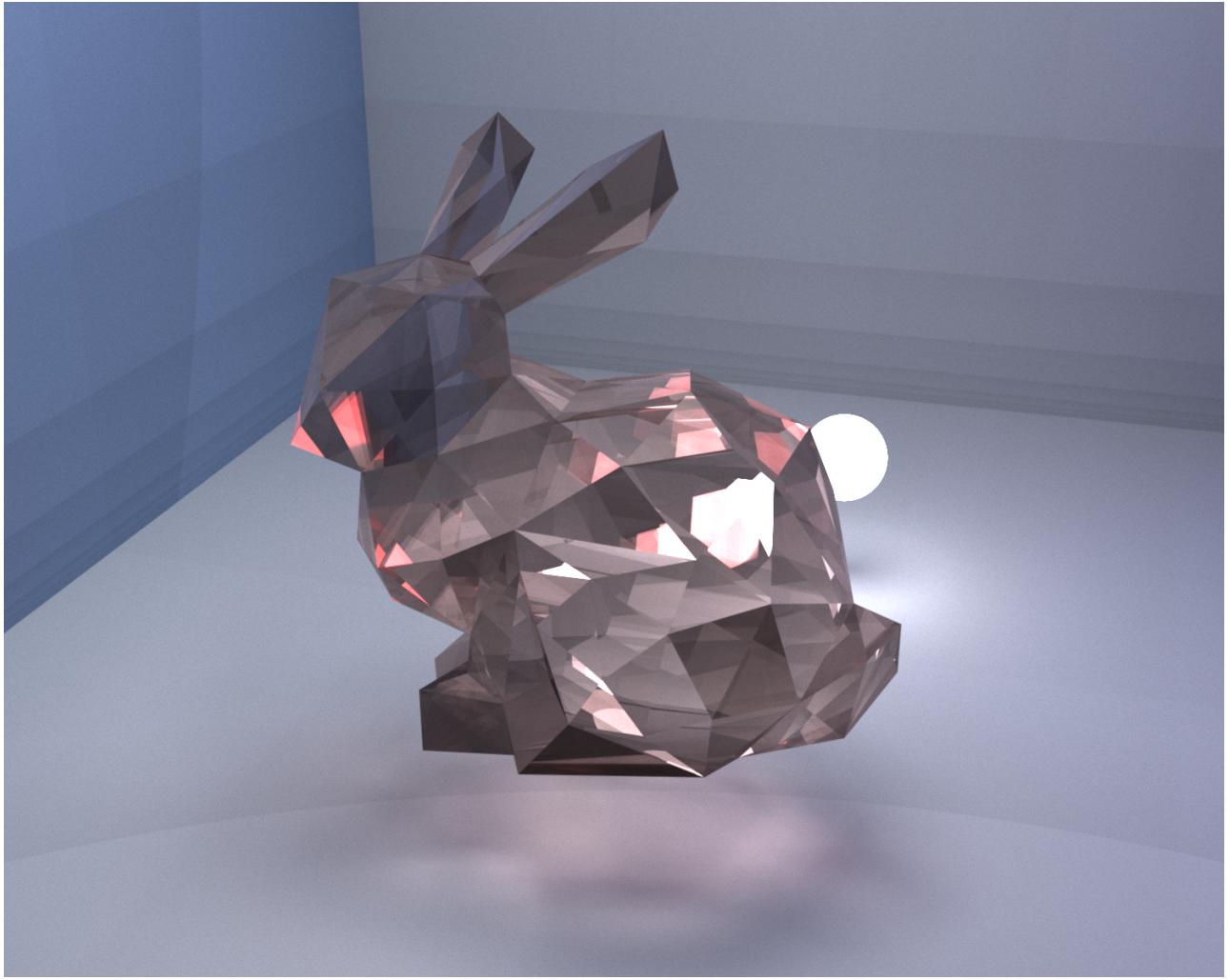
随后在进行射线与网格模型的求交计算时，对交点的法向量进行中心插值：根据交点在所在三角面片中的重心坐标，对三个顶点的法向进行加权平均得到交点的法向。代码实现如下（位于 `triangle.hpp` 中）：

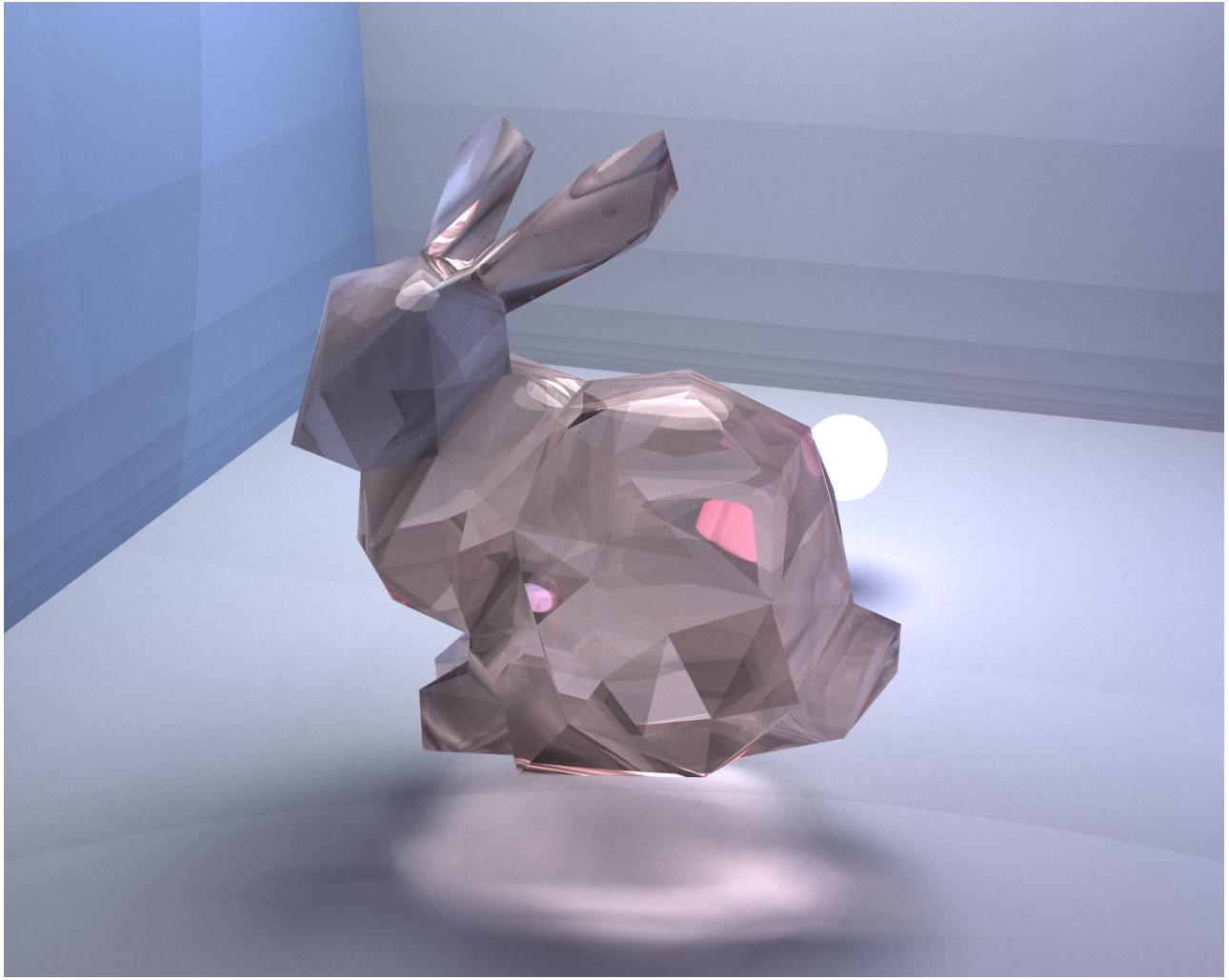
```

Vector3f getNorm(const Vector3f& p) {
    if (normal_set == false)
        return normal;
    else {
        Vector3f va = (vertices[0] - p), vb = (vertices[1] - p), vc =
(vertices[2] - p);
        float ra = Vector3f::cross(vb, vc).length(),
              rb = Vector3f::cross(vc, va).length(),
              rc = Vector3f::cross(va, vb).length();
        return (ra * norm_a + rb * norm_b + rc * norm_c).normalized();
    }
}

```

下面分别是没有进行法线插值和经过法线插值的效果对比。其中的网格模型使用的是含有 200 个三角面片的斯坦福兔子，考虑到渲染时间限制，使用的模型比较粗糙，平滑效果也无力回天，但还是可以看到法线插值带来的平滑效果的。由于整体算法简单、效率低下，即使用了包围盒和 OpenMP 加速，网格模型+折射的渲染还是很耗算力的，用 AMD R9 5950X 渲染了一整晚。





4 算法效率

4.1 算法加速

对于三角网格和旋转曲面这两种在光线求交过程中计算量较大的物体，我都实现了轴对齐包围盒加速。用一个与 x , y , z 三个坐标轴平齐的长方体盒子包围住目标物体，求交时先和这个长方体盒子求交，如果没有交点则一定不会与其中包围的物体相交。这样可以减少大量不必要的计算，加快含有三角网格和旋转曲面的场景的渲染。经测试，对于特定场景，包围盒加速可以把渲染时间缩短64%。相关代码位于 `mesh.cpp` 和 `revsurface.hpp` 中。

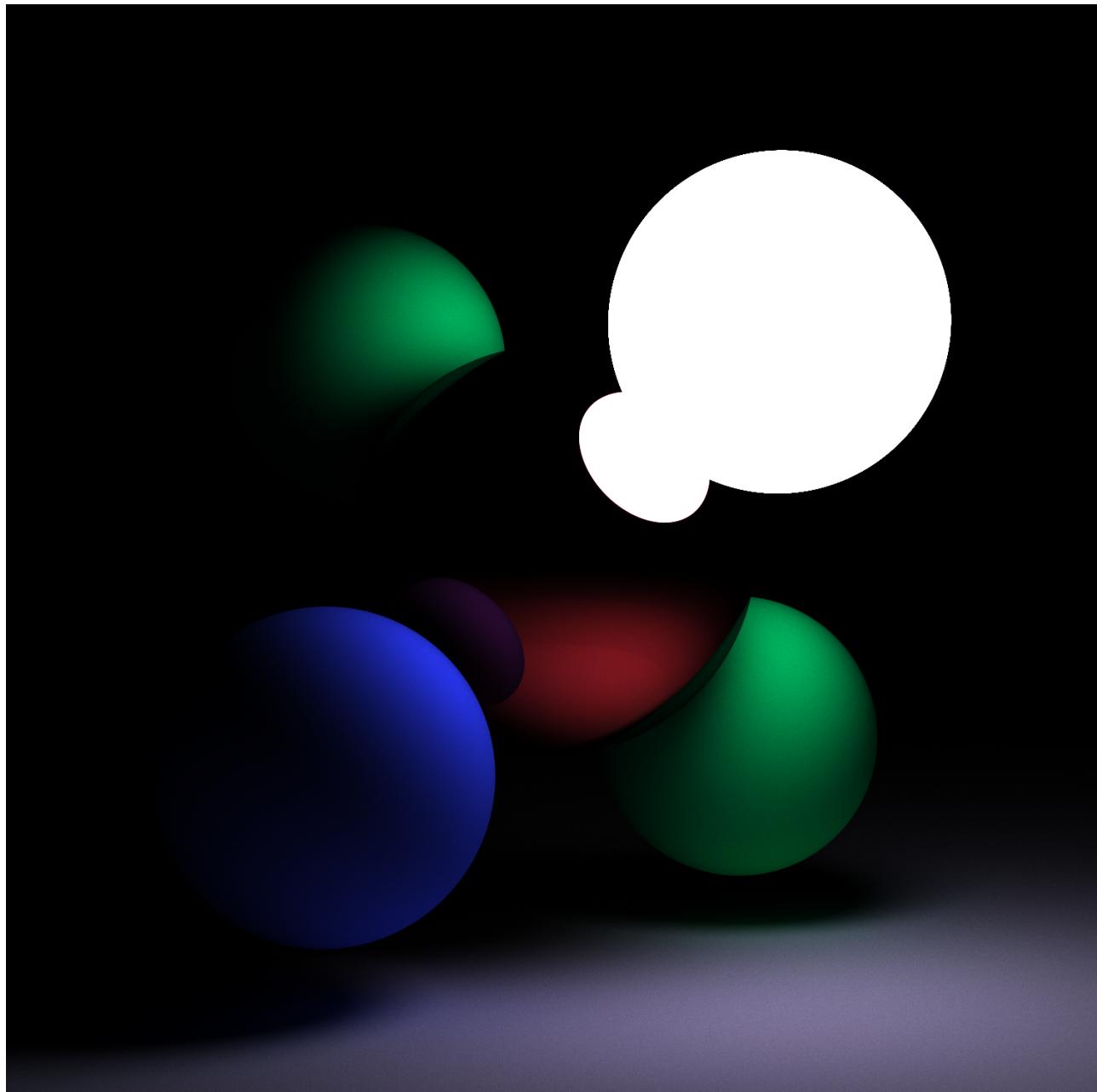
4.2 硬件加速

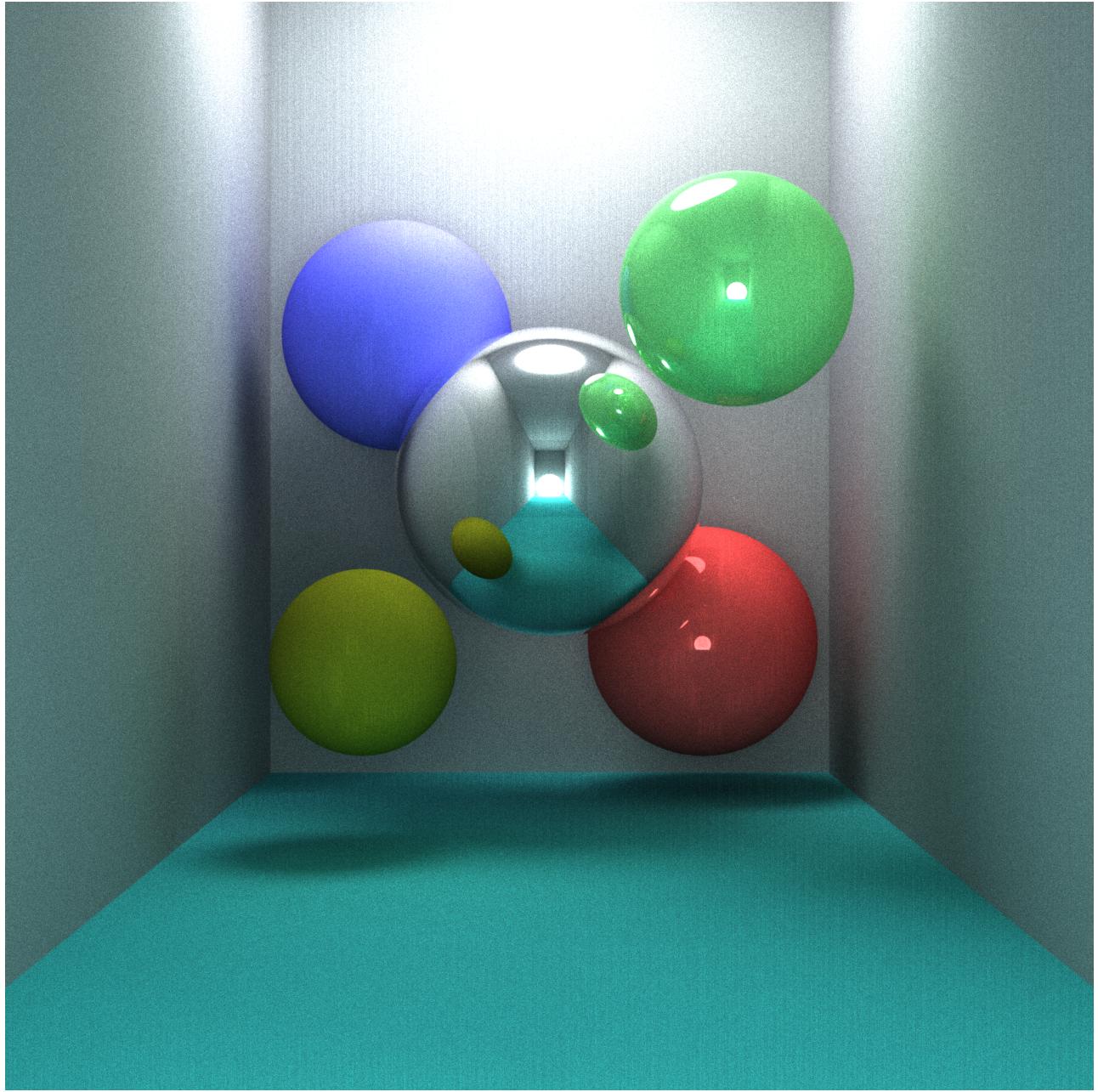
使用 C++ 中轻量级的多线程库 OpenMP 可以把程序并行化，对于图像渲染任务来说，由于每个像素的颜色值计算是相互独立的，可以把它们平均地分配给不同的线程运行。基本上计算机的CPU有多少个核心，性能就会翻多少倍，可以充分利用CPU的性能，加速效果很明显。

5 其他效果呈现

部分效果图已经作为一些实现效果的展示出现在上文中了，以下是另外一些效果图的展示。

5.1 路径追踪+基本反射、折射





5.2 Bezier旋转曲面+贴图

