计算机图形学真实感图形渲染作业报告

计算机科学与技术系　顾煜贤　2017011421

June 30, 2019

Contents

1 编译运行 **3**

2 代码整体框架 **3**

3 图形建模 **3**

3.1 球与部分球 . . . 3

3.1.1 求交算法 . . . 3

3.2 Bezier 曲面建模 . . . 4

3.2.1 求交算法 . . . 4

3.3 网格建模 . . . 5

3.3.1 求交算法 . . . 5

3.4 光源 . . . 6

3.4.1 求交算法 . . . 6

4 场景渲染 **7**

4.1 渲染算法 . . . 7

4.1.1 Ray Tracing . . . 7

4.1.2 Progressive Photon Mapping . . . 7

4.2 几何光学性质 . . . 9

4.2.1 反射 . . . 9

4.2.2 折射 . . . 10

4.3 渲染加分项 . . . 10

4.3.1 面光源采样 . . . 10

4.3.2 纹理贴图 . . . 10

4.3.3 景深 . . . 12

4.3.4 超采样抗锯齿 . . . 12

4.4 渲染加速 . . . 12

4.4.1 包围盒 . . . 12

4.4.2 kd树 . . . 12

4.4.3 OpenMP . . . 13

5 配置文件 **13**

6 最终效果图 **14**

6.1 Marvel Memory . . . 14

6.2 Museum . . . 15

6.3 Van Gogh Room . . . 16

1 编译运行

```
cd hw2/src
make clean
make
./main config/pic1.json
```

2 代码整体框架

本次作业代码由渲染器、场景、物体和配置四个主要部分组成。通过修改配置文件，改变场景中物体的摆放、纹理贴图、渲染器设置等，由渲染器渲染。图片的输入以及纹理图片的导入借助OpenCV的imread与imwrite函数，配置文件的输入输出使用Jsoncpp库，其他部分均为自己实现，合计3631 行代码。各文件功能如下：

- main.cpp: 主程序入口
- render.h render.cpp: 渲染器
- scene.h scene.cpp: 场景类，可在其中定义各种Object
- bezier.h bezier.cpp b_obj.hpp: 旋转Bezier曲线类
- mesh_obj.hpp: 网格化物体类
- object.h object.cpp plane.hpp ball.hpp sub_ball.hpp: 物体类
- light.hpp: 光线类
- vec3.hpp: 三维向量类，定义了多种向量运算
- utils.h utils.cpp: 定义了一些常量，一些全局的函数

物体的定义采用继承的机制，场景通过物体基类的指针（Object*）引用所有物体，调用成员函数时通过虚函数实现多态。图1为继承关系图。

3 图形建模

3.1 球与部分球

定义球体的参数为：半径r（double），球心位置c（vec3）。部分球在此基础上增加了球面中心方向dir（vec3），以及部分球面张开的最大角度θ (double)。

3.1.1 求交算法

光线与球的求交参考[4]。与部分球求交时需要计算交点与球心的连线和部分球面中心方向的夹角，当此夹角小于$theta$时，认为光线与部分球面相交，否则认为不相交。

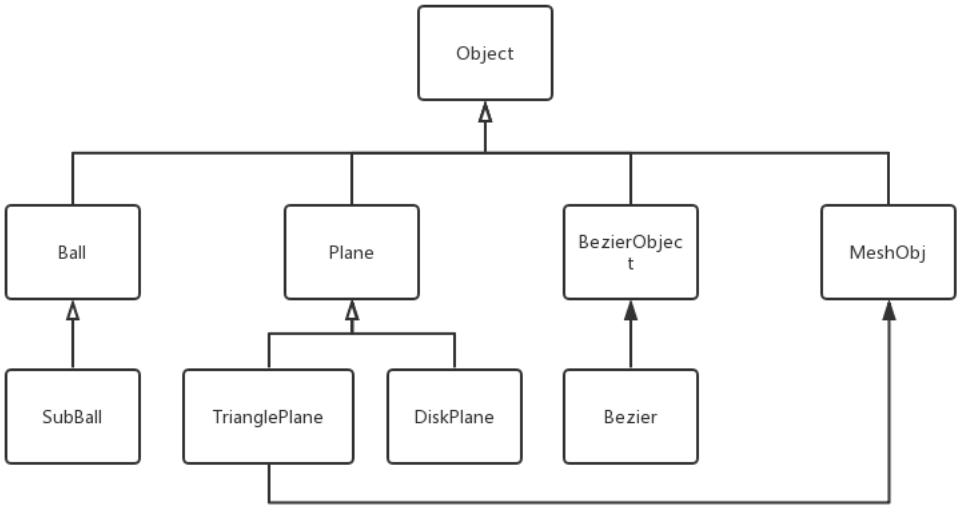

Figure 1: 物体的UML图

3.2 Bezier 曲面建模

高次贝塞尔曲面由n个控制点组成，通过下列参数方程定义：

$$f(t) = \sum_{i=0}^{n} P_i C_n^i t^i (1-t)^{n-i} \tag{1}$$

其中，$P_i = (x_i, y_i)$，为第i个控制点的位置。

但是直接计算求和式的成本很高，因此可以使用de Casteljau算法[1]计算贝塞尔曲线每点的值：

$$P_i^k = \begin{cases} P_i & k = 0, \\ (1-t)P_i^{k-1} + tP_{i+1}^{k-1} & k = 1, 2, \ldots, n, i = 0, 1, \ldots, n-k. \end{cases} \tag{2}$$

该算法复杂度为$O(n^2)$

旋转Bezier曲面需要以下参数确定：旋转曲线的控制点$\{P_i\}$（vec3），旋转面底部中心位置p(vec3)。

3.2.1 求交算法

本次作业采用的光线与参数曲面求交的算法参考了6字班一位学长[3]的方法：将一个三维的求教问题转化为一个一维的方程求解问题。

设直线上的点$(x_0 + t \cdot n_x, y_0 + t \cdot n_y, z_0 + t \cdot n_z)$，旋转Bezier曲面上的点$(x, y, z)$，对应Bezier曲面上的点$x_B(t), y_B(t)$（旋转轴垂直于$x-y$平面放置，旋转面底部中心位置$(p_x, p_y, p_z)$）。该直线与Bezier曲面相交的充要条件是存在某个$z = z_0'$ 平面，使得直线与此平面交点到旋转轴的距离d_l等于旋转Bezier 曲线与平面交点到旋转

轴的距离d_r。即：

$$\sqrt{(x_0+\frac{y_B(t)-z_0}{n_z}n_x-p_x)^2+(y_0+\frac{y_B(t)-z_0}{n_z}n_y-p_y)^2}=x_B(t) \tag{3}$$

此式经过化简可得：

$$f(t)=\sqrt{ay_B(t)^2+by_B(t)+c}-x_B(t)=0 \tag{4}$$

只需解出式中的t，便可得到交点：

$$P=(x_0+\frac{y_B(t)-z_0}{n_z}n_x, y_0+\frac{y_B(t)-z_0}{n_z}n_y, y_B(t)) \tag{5}$$

至此，原问题转化为了一个一元函数求零点的问题。

采用牛顿迭代法求解此函数的零点，选定初值t_0，按下式进行迭代更新：

$$t=t-\frac{f(t)}{f'(t)} \tag{6}$$

当$|f(t)|<\epsilon$时，迭代停止。由于这样可能会产生多个解，因此取离光线起始点最近的解。

经过实验，取$\epsilon=10^{-4}$效果较好。为保证精度以及解出所有解，上述方程会被多次求解，每次求解的初始值不同，每次初始值t_0选择方法：将$[0,1]$平均分成10个区间，对每个区间选择此区间的三等分点作为t的初始值进行上述迭代。对于所解得不同的t，取其对应的距离光线起点最近的交点作为最终的解。

最终效果如图2所示：

Figure 2: 通过上述求交算法实现的旋转Beizer曲面

3.3 网格建模

本次作业支持.ply格式的网格文件，并支持纹理贴图。

3.3.1 求交算法

网格化求交本质上是光线与三角面片的求交。具体实现中，先计算光线与三角面对应平面的交点，然后判断交

点是否在三角面片内部。判断方法：设三角面片三个顶点的坐标分别为$u_{xi}, u_{yi}, uzi, i = 1, 2, 3$。则交点$P(x, y, z)$应满足方程：

$$\begin{cases} u_{x1}t_1 + u_{x2}t_2 + u_{x3}(1 - t_1 - t_2) = x \\ u_{y1}t_1 + u_{y2}t_2 + u_{y3}(1 - t_1 - t_2) = y \\ u_{z1}t_1 + u_{z2}t_2 + u_{z3}(1 - t_1 - t_2) = z \end{cases} \tag{7}$$

该方程有一个冗余方程，且三个方程的两两组合中必有一个的系数行列式不为0，用改组合即可解得t_1, t_2。交点在三角面片内部当且仅当:

$$\begin{cases} 0 < t1 < 1 \\ 0 < t2 < 1 \\ 0 < 1 - (t1 + t2) < 1 \end{cases} \tag{8}$$

通过此方法可以实现三角面片与光线的求交(图3)。同时，此处解得的t_1, t_2在之后的复杂网格纹理映射中还会使用。

Figure 3: 通过上述求交算法实现的网格模型

3.4 光源

场景中的光源可以为任何物体，只需在物体的配置中加上光源属性即可。实现中，可以重载generateLight()函数实现不同物体的发光特性。

3.4.1 求交算法

与光源的基础物体求交算法基本无异，不同之处在于如果一个物体被定义为了光源属性，颜色返回时将会乘以一个较大的系数。效果如图4：

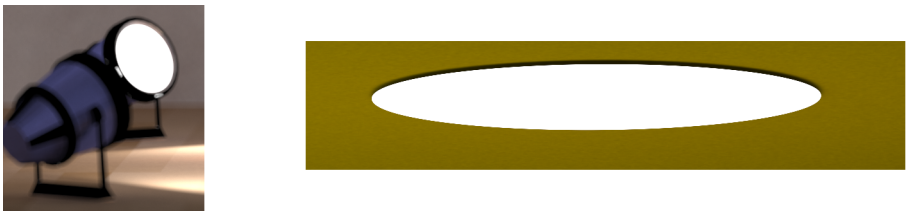

Figure 4: 不同的光源

4 场景渲染

4.1 渲染算法

本次作业一共尝试过两种渲染算法：经典的Ray Tracing 以及Progressive Photon Mapping.

4.1.1 Ray Tracing

经典的Ray Tracing算法从观察平面的像素点处开始向场景透射光线，计算光线与物体的交点，达到漫反射面或规定深度后终止。然后使用Phong模型计算局部光照。 Ray Tracing在光线追踪的过程中没有加入任何随机的因

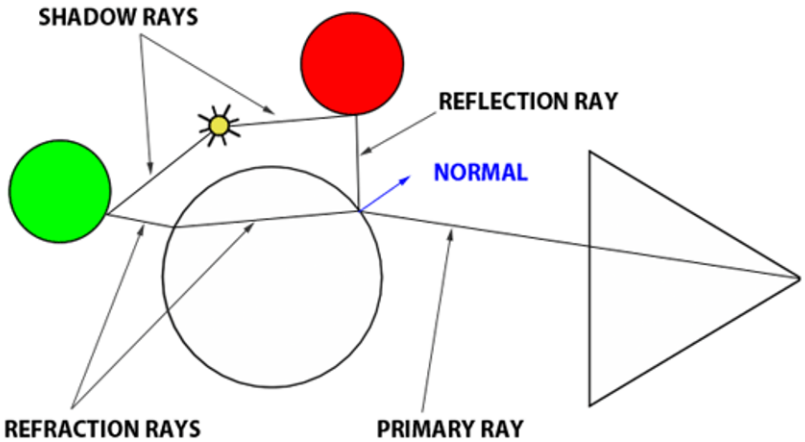

Figure 5: Ray Tracing 原理图

素，遇到漫反射面也不反射，并且无法做出光在透明物体下产生的汇聚效果，渲染时间虽短，但是效果一般，不太真实（图6）。

4.1.2 Progressive Photon Mapping

渐进式光子映射(ppm)[2]流程如下：将上述光线追踪过程中遇到漫反射面停止后的视点(View Point)组织成一棵kd树，每个视点都有一个自己的搜索半径，称为收光半径。然后从光源按方向均匀发射光子，每个光子携带一定光源对应的能量。光子在场景中反射、折射，也进行漫反射，这个过程中有能量的损失，同时也加入了碰到物体的颜色信息。对于落入视点收光半径内的光子，计算其对视点光亮度的贡献，将其与原来视点的光亮度相加。视点

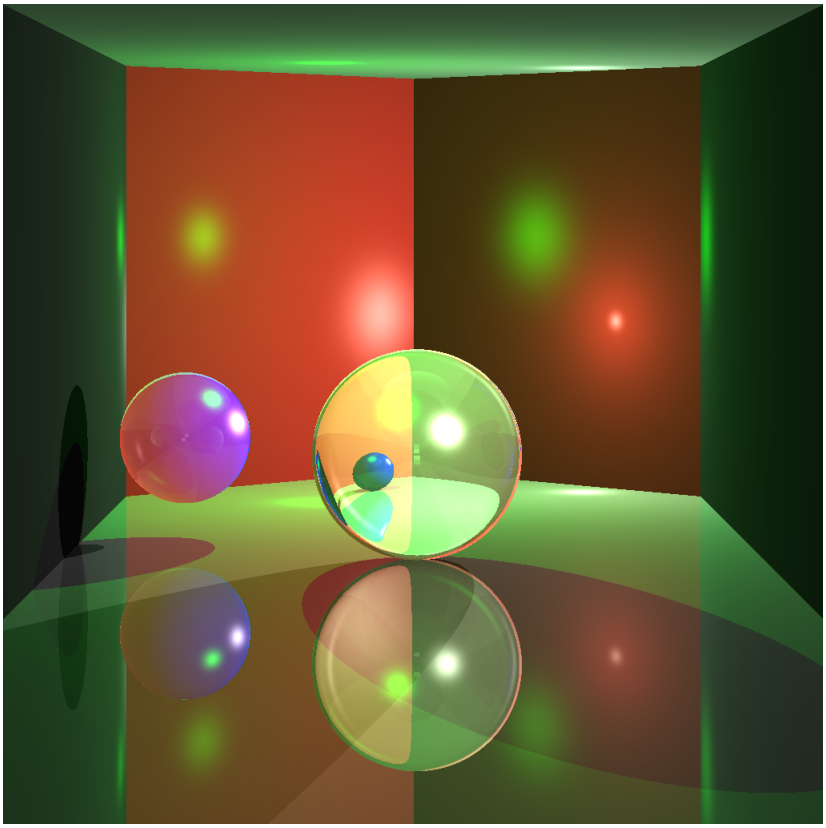

Figure 6: Ray Tracing 结果

每接收到一个光子，其收光半径缩小一次。根据论文结果，半径收缩方式：

$$\hat{R(x)} = R(x)\sqrt{\frac{N(x)+\alpha M(x)}{N(x)+M(x)}} \tag{9}$$

其中，$R(x)$为受到光子之前的半径，$\hat{R(x)}$为收到光子之后的半径。$N(x)$为当前视点收到的光子数。可见，随着渲染的推进，$N(x)$逐渐增大，$R(x)$减小的速率逐渐变小。

最终每个像素点对应一个视点，对应一个光亮度。使用如下公式计算最终像素点的颜色：

$$L(x) = \frac{\tau_n(x)}{\pi R_n(x)^2 N_{emitted}} \cdot C \tag{10}$$

其中，τ_n为此像素点对应视点接受n个光子后的总亮度，$R_n(x)$为此时视点的收光半径，$N_{emitted}$为发射的光子总数。与原论文不同的是，此公式还额外乘了一个常数C，用来调节图片的整体亮度。可以证明，当$N_{emitted} \to 0$时，$L(x)$收敛。

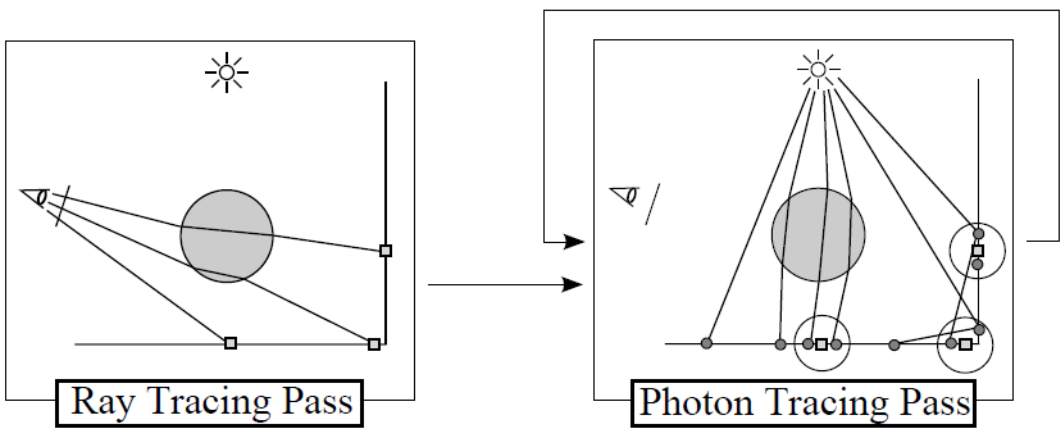

Figure 7: Progressive Photon Mapping 结果

图8是使用了ppm算法的效果。可见，球体真实感提升了很多。

4.2 几何光学性质

4.2.1 反射

光在表面上的反射遵循反射定律：

$$\cos\theta_i = \cos\theta_r$$

计算反射方向的公式为：

$$\vec{R} = \vec{I} - 2(\vec{I}\cdot\vec{N})\vec{N} \tag{11}$$

其中，$\vec{I}$为入射光方向，$\vec{N}$为法线方向，$\vec{R}$为反射光方向。

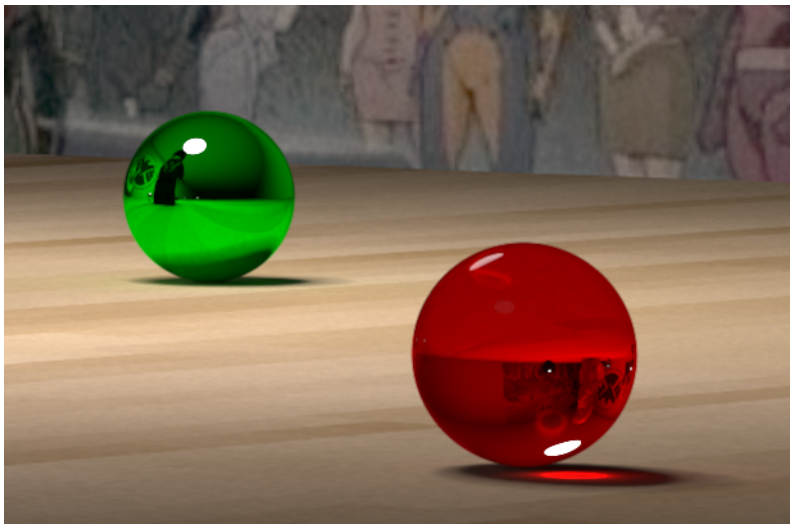

Figure 8: Progressive Photon Mapping 结果

4.2.2 折射

物理学中斯涅尔定律：

$$n = \frac{\sin\theta_i}{\sin\theta_t}$$

可得折射光线：

$$\vec{T} = n \cdot \vec{I} + (n(-\vec{I} \cdot \vec{N}) - \cos\theta_t) \cdot \vec{N} \tag{12}$$

其中，$\theta_t = \sqrt{1 - n^2(1 - \cos^2\theta_i)}$，$n$为两表面的相对折射率。

光在透射面反射折射、反射的时候光强对于反射、折射的光强分配满足菲涅尔公式：

$$k_r = \frac{1}{2}(r_p^2 + r_s^2), \quad k_t = 1 - k_r \tag{13}$$

$$r_p = \frac{n\cos\theta_i - \cos\theta_t}{n\cos\theta_i + \cos\theta_t} \quad r_s = \frac{\cos\theta_i - n\cos\theta_t}{\cos\theta_i + n\cos\theta_t} \tag{14}$$

每次光射到一个透射面，反射、透射光强分别是原来光强的k_r与k_t倍。

4.3 渲染加分项

4.3.1 面光源采样

通过面光源采样可以实现软阴影效果，只需在每次选择光源发出的光子时，光子发出的位置在面光源内随机选择即可。两种效果的对比如图9。

4.3.2 纹理贴图

本次作业实现了复杂网格的纹理贴图、旋转Bezier曲面的纹理贴图，平面贴图以及部分球的纹理贴图。复杂网格的纹理贴图根据输入文件中三角面片三个点对应纹理图中的位置，反算出场景中三角面片内部的任一点对应的纹

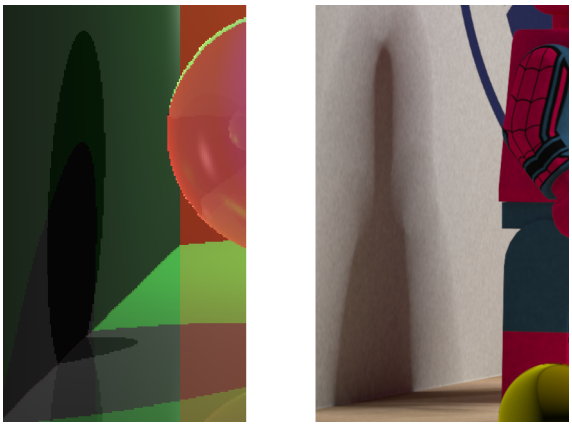

Figure 9: 两种阴影对比，左边为单点光源，右边为面光源采样

理坐标。设一个三角面片在场景坐标系下三个点为(P_1, P_2, p_3)，在纹理图坐标系下坐标为(P_1', P_2', P_3')。根据Section 3.3.1的结论，对三角面片中的任意一点P，有：

$$P = t_1 P_1 + t_2 P_2 + (1 - t_1 - t_2) P_3$$

因此，在纹理图坐标系下，同样有：

$$P' = t_1 P_1' + t_2 P_2' + (1 - t_1 - t_2) P_3'$$

用场景坐标系下坐标解出t_1与t_2后，便可得到一点对应的纹理坐标，从而得到其颜色。图10为网格贴图示例，可见各种细节还是十分细致的。 部分球的纹理贴图只需根据球面上一点P到部分球中心轴方向的角距离以及部分球中

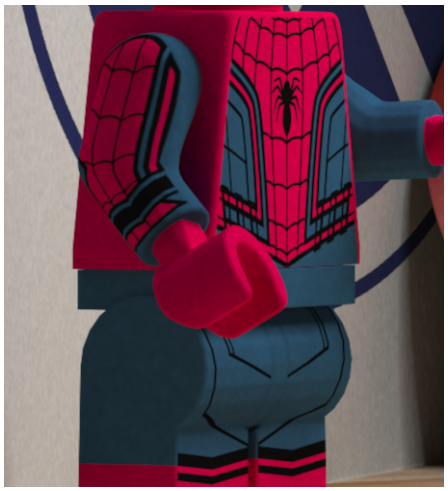

Figure 10: 复杂网格贴图

心在纹理图上的位置确定出P点在纹理图上对应的位置以及颜色即可。图11为部分球贴图示例。

Figure 11: 部分球贴图

4.3.3 景深

光线追踪时，对每个像素点发出的射线，保持焦平面上的点不动，对射线发出处的端点做微小扰动，端点扰动范围的半径即为光圈。这样原来打在视平面上的光点会变成一个光斑，物体越远，光斑越大，使较远物体产生模糊的效果(图12)。

4.3.4 超采样抗锯齿

原始算法的锯齿感较明显，因此采用超采样抗锯齿的方法，即每个像素点实际对应四条光线，四个视点，除了从其原来位置发出的光线之外，还包括旁边三个点发出的光线。从图9中可以看出，左边（未做抗锯齿）物体的分界面有明显的锯齿，右边（做了抗锯齿）基本没有锯齿感。

4.4 渲染加速

4.4.1 包围盒

计算所有物体的长方体包围盒（AABB），只有当光线与包围盒相交时才计算光线与内部物体的交点。

4.4.2 kd树

kd树主要有两个用途：ppm中视点收光的查询以及求交时的加速。ppm中以视点分割空间，求交时以包围盒中心分割空间。理想随机情况下，单次查询与单次求交的复杂度为$\log(n)$。但由于网格化三角面片的分布实际上很不随机，因此实际求交的复杂度应高于$\log(n)$，但是仍然有加速效果。

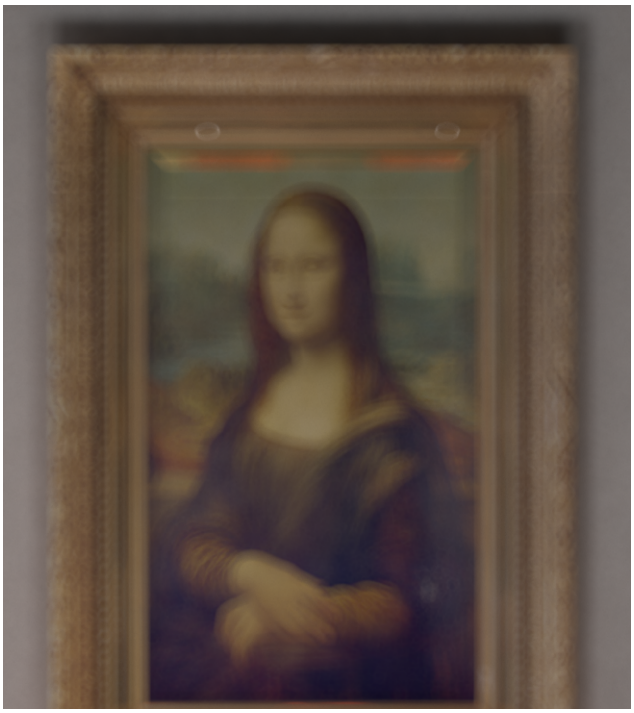

Figure 12: 景深效果使背景虚化

4.4.3 OpenMP

使用OpenMP可实现多线程并行加速。对于PPM而言，可以将光源发射的光子数均分给若干个线程，每个线程出来一张缓存图片，等到所有线程都结束之后，再将其合并至一张图中，这样能够在不用加锁的情况下实现快速渲染。

5 配置文件

本次作业实现了Json格式配置文件的读写（使用Jsoncpp库），以方便图像的调试。Json配置文件存储于config目录下，pic1.json，pic2.json，pic3.json分别为三张图的配置文件。这种实现为后期繁琐的调图工作提供了很大的便捷，以至于调图可以成为考试周茶余饭后的一种消遣了。（由于文件过长，在此不放示例了）

6 最终效果图

所有最终结果均为2560x1920的图片，存放在final目录下。

6.1 Marvel Memory

两个Lego人偶为复杂网格，同时展示了复杂网格纹理贴图的效果。后边的盾牌为球的一部分，也实现了贴图。地上有六种颜色的球（分别对应六种无限宝石的颜色），其中有两个球为透射，其他四个球为反射，可见透射、反射效果还是十分清晰的。

Figure 13: Marvel Memory

6.2 Museum

一张类似博物馆的图，四个探照灯都是复杂网格模型，灯的发光面为圆形片状光源。花瓶为旋转贝塞尔曲线模型，并实现了纹理贴图，可以看到探照灯的灯光经过花瓶反射打在地上的一片光影。这张图同时还实现了景深效果，后面蒙娜丽莎为网格模型，可以看到有明显的虚化。

Figure 14: Museum

6.3 Van Gogh Room

整张图就是一张复杂网格模型，大概有20w+的面片，做了上述各种优化后，2560x1920的图像，3000轮，每轮发射30w个光子可以在20小时之内跑完。窗户外面通过光源的配合做出自然光的效果。

Figure 15: Van Gogh Room

References

[1] Hachisuka, Toshiya, Shinji Ogaki, and Henrik Wann Jensen. "Progressive photon mapping." ACM Transactions on Graphics (TOG). Vol. 27. No. 5. ACM, 2008.

[2] De casteljau's algorithm. `https://en.wikipedia.org/wiki/De_Casteljau\%27s_algorithm.`

[3] `https://github.com/Trinkle23897/Computational-Graphics-THU-2018/tree/master/hw2`

[4] 孙家广, 胡事民. 计算机图形学基础教程. 清华大学出版社, 2005