计算机图形学.大作业

2018011324

尤艺霖

0.运行环境

系统：Windows 10 家庭版

g++：4.9.2

python：需要openCV来进行图像的转换，如果只需要复现渲染则不需要python

1.文件结构

没有采用小作业的框架，自己完成了一个新的框架。

1.1.src目录下：

这里存放了所有运行需要的代码和数据。除了放在外面的几个算法核心代码和场景文件解析器，还有四个子目录：

1.1.1.basic目录下：

存放框架需要的一些基础操作。包括了相机类、射线类、碰撞点类等。

1.1.2.objects目录下：

存放所有需要放置在场景里的物体，包括Bezier旋转面、三角形、复杂网格、球体等。所有物体继承于同一个基类，在代码中主要负责射线和物体的求交，并返回碰撞时间和法向量。

1.1.3.texture目录下：

存放所有需要使用的材质，包括折射、全反射、漫反射、贴图、渐变等。所有材质类继承于同一个基类。在代码中主要负责传入法向量和入射光，返回碰撞点的颜色和下一步的射线。

1.1.4.models目录下：

存放渲染所需要的模型文件、自己设计的场景文件、贴图的原始文件和经过转化的贴图文件，以及转化贴图文件的代码。其中：**bztest.txt**是对照组用的场景，**tritest.txt**是PT渲染用的场景，**sppmtest.txt**是SPPM渲染的场景

场景文件是自己的格式，只能适配自己的**parser.hpp**

1.2.result目录下：

存放了产生的三张结果图。

PT-2000.bmp：

尺寸为1200×800。使用PT渲染。每个像素用四个子像素进行超采样，每个子像素都采用2000的采样率。并加入了景深效果。最终结果很好，可作为最终提交的成果。

SPPM-100.bmp：

尺寸为1200×800。使用SPPM渲染的场景，加入了一个棱镜。每轮射出总像素数10倍的光子，总共进行100轮渲染。可以看出焦散效果，但是由于算法经过一定简化且渲染时间较短，整体上有一定瑕疵。

PT-对照组.bmp：

尺寸为600×400。采用与SPPM相同的场景但用PT渲染，用来验证SPPM所带来的焦散是否正确实现，因而采样率不高，但可以体现和SPPM的差距。

1.3.运行说明：

对于运行所需要的参数，如PT的采样数、是否抗锯齿、是否景深、以及输入的场景文件，又如SPPM的初始半径等参数，都需要在代码里进行修改。

src目录下有两个批处理文件，直接运行即可完成编译+运行的操作。输出的文件是**ppm**格式的，可以调用**imageturn.py**将图片转换为**bmp**格式。

2.实现的功能

2.1.Path Tracing

Path Tracing的实现参考了开源资料smallpt[1]，主体程序的实现是非常简单的，在**pathtracing.cpp**中。直接沿着光路递归，不断调用物体类的求交和材质类的求下一条线的函数即可。一些实现细节如下：

具体的三角形、球形的求交直接参考了PA1中的写法，并没有实现平面，因为我认为所有准备渲染出的平面都可以用极大的三角形或球体代替。为了方便包围盒的求交，我还实现了一个与坐标轴平行的长方体的类，命名为AAcube（axis-aligned cube），实现在**objects\aacube.hpp**中。

材质类则需要实现计算下一条出射射线的方向的算法。镜面反射是很好实现的，漫反射选择的是smallpt的实现方式，在半球上选择一个方向射出，同时可以通过调节垂直法向量的系数，来控制到在一个圆锥视角内射出。折射的选择也是按照smallpt的方式实现，但是由于这种实现方式无法进行光线分叉，就将所有的折射光线按照概率进行随机的折射或反射，并乘上对应的系数。其他的一些材质均是漫反射的变种，可以直接看**texture**目录下的对应代码。

递归判停的方法用了smallpt中的俄罗斯轮盘法，同时手动加了一个20000的递归上限防止爆栈。

2.2.Bezier旋转面求交

Bezier旋转面求交采取的方法是牛顿迭代法，实现在**objects\bezier.hpp**中，具体方法如下：

首先简化问题为绕着z轴进行旋转的曲线构成的旋转面，考虑Bezier曲线本身可以表示为$(X(t),Y(t))$，那么对应的旋转面的点集就是$(X(t)cos\theta,X(t)sin\theta,Y(t))$，那么我们固定$t$，就可以得到一个$xy$平面上的圆盘，那么思路就是找到这个圆盘上的交点。然后假设这个圆盘对应的t为未知数，那么再假设入射射线为$O+t'D$，那么这个射线的t'就可以表示为$\frac{Y(t)-O.z}{D.z}$，因而这个射线最终落在这个圆盘上的坐标为$(t'D.x+O.x,t'D.y+O.y,Y(t))$，同时为了让射线和旋转面有交，则需要有这个落点离旋转轴的距离为$X(t)$，接下来式子用X,Y代替$X(t)$和$Y(t)$。需要解的方程可以列为：

$$(\frac{Y-O.z}{D.z}D.x+O.x)^2+(\frac{Y-O.z}{D.z}D.y+O.y)^2=X^2$$

此时左右两侧均为关于t的多项式，那么这个式子就可以通过牛顿迭代来解。

在实现中，为了能够快速求导数，我将这个式子完全展开成了系数形式，损失的精度在点数不多的情况下可以忽略，而X和Y的系数表示则可以直接用二项式系数暴力展开$Bernstein$多项式来解决。当然，还需要用一个包围盒判断是否完全不可能相交。

同时还需要独立讨论$D.z$为零的情况，这个时候上面的式子是无意义的， 但是这种情况可以直接近似为和一个球体求交，只需要先用牛顿迭代求出$Y(t)=O.z$的t即可。计算法向量的时候首先算出$(X'(t),Y'(t))$，然后旋转到对应位置，就是一条切向量。另一条切向量选择水平方向的切向量，可以直接用落点的位置算出来，然后就可以叉乘两条切向量得到法向量。

在最终的结果中，采用的曲线有五个控制点，每次求交随机两百个初值，虽然有些低效，但是几乎总是正确的。而损失的效率由于有了包围盒，在整体渲染上也是可以接受的。

2.3.复杂网格求交

复杂网格求交采用了kd树进行优化，具体的实现在**objects\mesh.hpp**中，细节如下：

首先选择kd树切割的维度，是用所有三角形的所有点的坐标的方差来决定的，方差最大的一维优先切割。

然后在切割的过程中，选择切割点的判据使用了SAH方法。核心思路是让查询的总期望值最低。用包围盒的表面积来近似射线命中的概率，再用包围盒内的三角形数量来近似查询的代价。这个过程如果暴力实现是平方级别复杂度的，但是可以先按照需要切割的那一维在三角形内顶点的最大值来对三角形进行排序，再预处理一下前缀后缀的最大值最小值，就可以在线性时间内找到最合适的切割点。包围盒直接利用之前的AAcube即可。

在最终的结果中，选用了网络学堂上的恐龙骨架模型，四千个三角形，已经足以体现出kd树提供的优化了。

2.4.景深、抗锯齿、软阴影

景深的实现在**basic\camera.hpp**中**camera**类的**getRay**函数里，在确定有景深的要求的情况下，将射线生成时在焦平面上选择的点固定，然后将射线出射点按照规定的光圈半径进行随机扰动，即可实现景深的效果。

抗锯齿使用的是超采样抗锯齿，将每个像素分成四个子像素，然后对子像素分别进行采样，最后取出平均值。具体的实现在**pathtracing.cpp**的主循环中。

软阴影是Path Tracing算法自带的，不需要专门的实现。

2.5.贴图

贴图的图片全部来自于网络，均存储在**models**目录下，有对应名称的**python**代码负责将图片从图片的格式转化为像素点的颜色列表，便于主程序读入。

主要实现了两种贴图，一种是直接贴在一个矩形面上，作为背景贴图，实现在了**texture\background.hpp**中。这一贴图的实现非常简单，就是将每个坐标随便做一下线性变换映射到一个像素上即可。这个背景贴图仅仅是从美观角度出发的，因此在这一大作业中我直接把变换方式硬编码到了代码中。

另一种贴图是将矩形贴图贴到球体上，具体到作业中我选择了一个地球的贴图。实现在**texture\sphere.hpp**中。这一贴图的实现是将法向量使用反三角函数映射到球坐标系的角度上，然后再映射到输入的图片的像素上。不同于上一种贴图，这个贴图的实现是具有通用性的，使用其他的图片也可以贴到球上。

2.6.SPPM

阅读了相关的论文[2]，然后实现了一个简化的版本，在**sppmbasic.hpp**和**sppm.cpp**中，细节如下：

SPPM本身的操作流程是由若干次迭代构成，每次迭代首先包括一轮eye pass，从相机射出射线，然后找到第一次相交的漫反射面，然后记录下碰撞点、法向量、颜色累积等信息，作为一个HitPoint。然后针对这些HitPoint建立kd树，每个Hitpoint都赋予一个收光半径，然后开始这一轮迭代的photon pass，从光源射出若干个光子，每次和漫反射面相交的时候，就对所有收光范围包括碰撞点的Hitpoint贡献一次。持续做若干次迭代直到收敛。

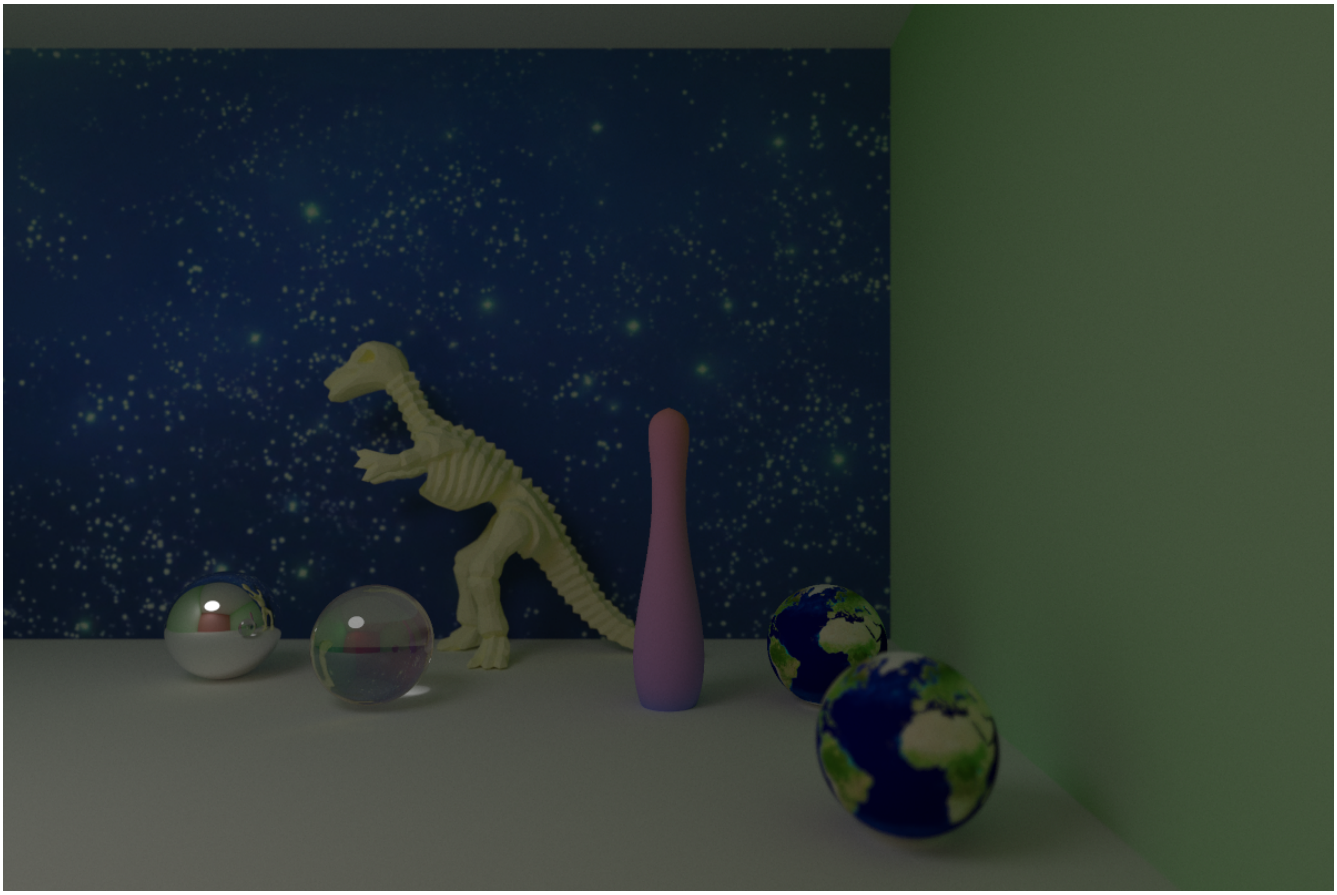
我做出的简化主要包括两个方面，首先是将论文中收光半径缩小的方式简化成所有HitPoint共享同一个半径，每一轮迭代之后让收光半径按固定的比例缩小。这样的近似方式，在合适的参数下应该是可以与论文中的算法达到相似水平的。另一个简化就是简化碰撞点的贡献为eye pass路径的颜色和photon pass路径颜色的乘积，从吸收率的角度看，这样的近似应该是十分合理的，然后通过全局参数调节光源亮度来实现一些系数的调整。在经过这样的简化后，可以直接基于Path Tracing的材质和物体类实现SPPM而不需要太多改动。

同时对光源也做了一些简化，固定成在天花板上的光源了，具体实现在**sppmbasic.hpp**中。

3.实验结果

两种算法都加入了OpenMP优化，实验效率提升明显。

3.1.Path Tracing

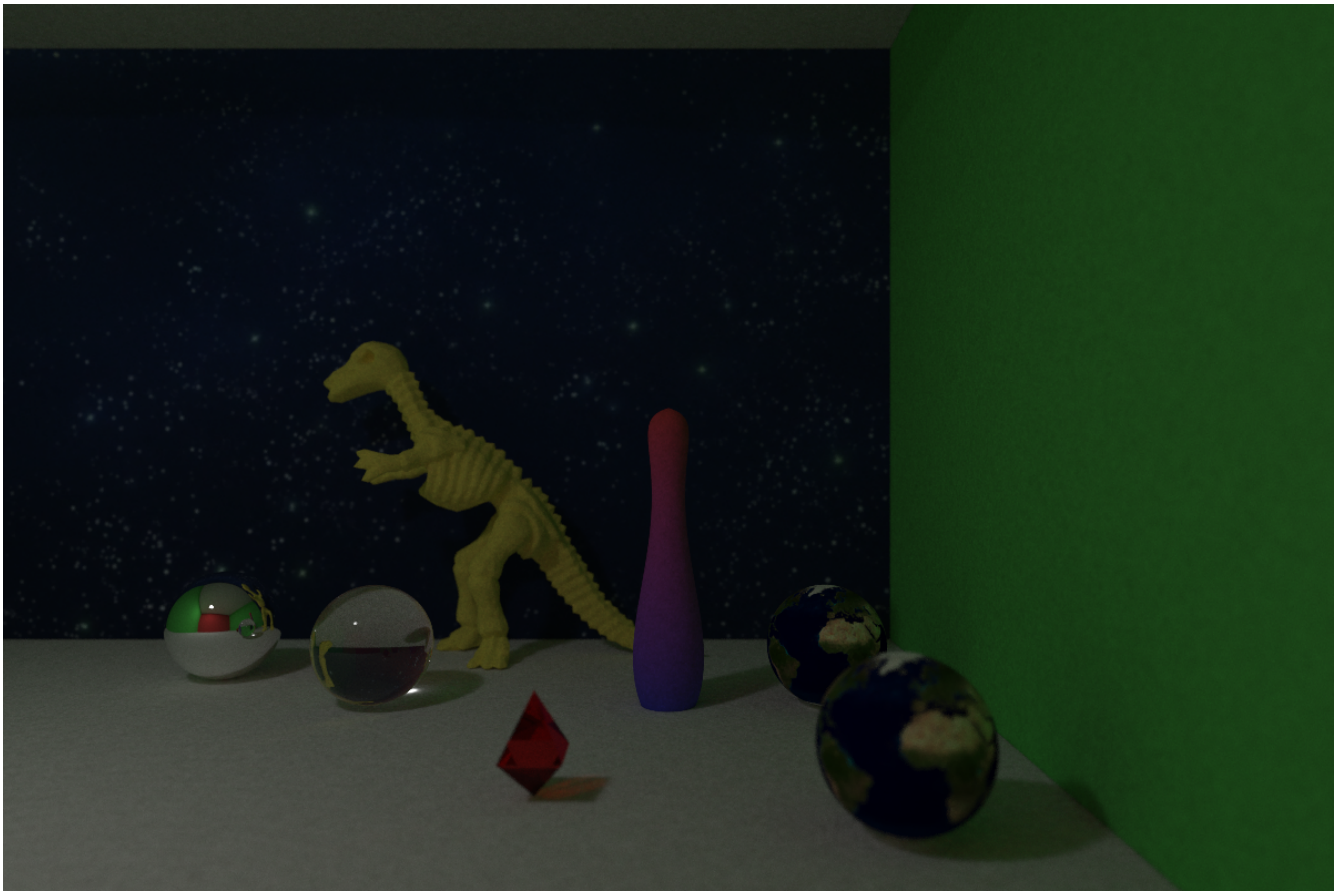


采样率2000，算上抗锯齿的4倍就是8000，最终渲染效果很好，可以作为最终实验结果。

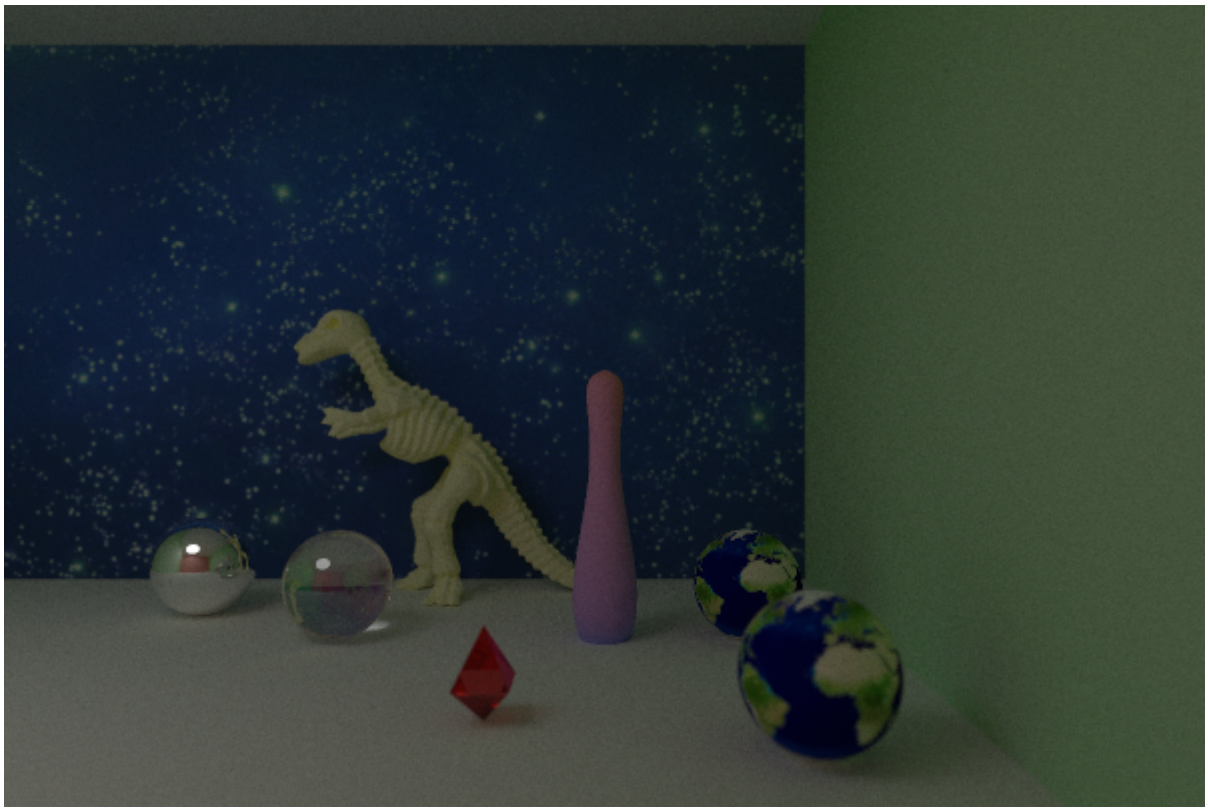
加入了景深效果，焦平面在旋转曲面的中轴线上，与背景墙平行。

可能看起来有点发白，但是从两个贴图的原图对比来看，这个算法实现和渲染结果都是没问题的。

3.2.SPPM



由于对算法做的简化，并且渲染时间不够长，导致最终效果不是很理想。但是从新加入的棱镜的影子来看，焦散效果应该是成功实现了的。可以对比以下是对照组渲染出的棱镜，整个影子都在泛红：



4.参考文献

[1] smallpt: Global Illumination in 99 lines of C++. http://www.kevinbeason.com/smallpt/

[2] Toshiya Hachisuka, Henrik Wann Jensen. Stochastic Progressive Photon Mapping. Vol. 28, No. 5. ACM Transactions on Graphics. December 2009.