计算机图形学第二次作业

实验报告

王　昭　软件32班　2013013330

目录

1 实验环境

- CPU：Intel i5-6500 @ 3.20GHz * 4
- Memory：16GB
- OS：Windows 10 教育版
- Visual Studio：2013

2 操作说明

2.1 程序运行方法

在控制台运行渲染程序，需要输入三个参数：用于渲染的场景文件（".scene"），漫游/只渲染模式（-travel/-render），在渲染模式下图像的存储位置。其中场景文件的定义将会在之后详细讲述。比如：

RayTracer.exe cornell_box.scene -travel

即可让渲染器执行对康奈尔盒场景进行渲染，并可以在其中进行漫游的操作。漫游模式只建议在小场景中使用，对于较大或者需要进行超采样的场景，建议使用渲染模式：

RayTracer.exe dragon.scene -render dragon.png

即可让渲染器执行对dragon.scene场景进行渲染并将图像存储为dragon.png的操作。渲染过程中会在控制台中显示进度条及剩余时间，并在渲染成功后显示渲染时间。

2.2 漫游方法

在以漫游方式进入程序后，可以通过表1所示方法来进行漫游。

键盘按键	效果
W、S、A、D键	前进、后退、左移、右移一小步
上、下、左、右箭头键	向上、向下、向左、向右旋转一点角度
T键	切换正投影/透视投影模式
Q键	退出程序

表 1: 漫游按键表

2.3 场景文件的定义

场景文件中存储了照相机以及各个景物的信息。以#开头的行被视为注释，每块信

息的首行为即将定义的目标，随后几行为该目标的属性。建议在定义场景时，属性的顺序严格按照如下的顺序进行书写（带*的属性为非必需属性）。

照相机 Viewer

图像大小 geo int(width) int(height)
视口大小（正投影参数）* view int(width) int(height)
照相机位置 center vec3
目标位置 target vec3
照相机北方 up vec3
视角（上下） fovy double
景深* dop double(光圈大小) double(焦距) int(超采样数)
抗锯齿* anti int(超采样数)

点光源 Light

面光源设置* area int(采样数的平方根) double(模拟球光源的半径)
位置 origin vec3
颜色 color string(颜色名)
光强 intensity double

无限平面 Plane

纹理 texture ...
法向 normal vec3
偏移量 offset double

球 Sphere

纹理 texture ...
中心 center vec3
半径 radius double
内部折射率* refraction_index string(折射率种类)

三角网格面 Face

纹理 texture ...
点**A** a vec3
点**B** b vec3
点**C** c vec3

网格模型 Mesh

纹理 texture ...
模型文件名 file string
中心位置 center vec3
包围球半径 radius double
光滑着色* smooth bool(true/false)

接下来对几个需要输入特殊值的属性做详细说明：

颜色名 共有8种颜色：WHITE,BLACK,RED,GREEN,BLUE,YELLOW,CYAN,MAGENTA
材料名 共有5种材料：FLOOR,MIRROR（全镜面反射）,A_BIT_MIRROR（一点镜面反射）,TRANSPARENT_MATERIAL（全透明）,PLASTIC
折射率种类 共有3种折射率：VACUUM_REFRACTION_INDEX（真空）,WATER_REFRACTION_INDEX（水）,GLASS_REFRACTION_INDEX （玻璃）
纹理 均以texture开头，需要注意的是，图片纹理只支持正方形图片作为输入。

纯色纹理 PureTexture string(材料名) string(颜色名)
黑白格纹理 GridTexture string(材料名) int(格密度)
图片纹理 ImageTexture string(材料名) string(纹理图片名) int(纹理大小)*

具体样例可以见scene目录下的所有场景文件。其中cornell_box.scene是建议用来进行漫游的场景，其余场景可以根据情况进行各种程度上的渲染。

3 实现内容

本次实验实现的内容涉及较多，接下来将会一一说明：

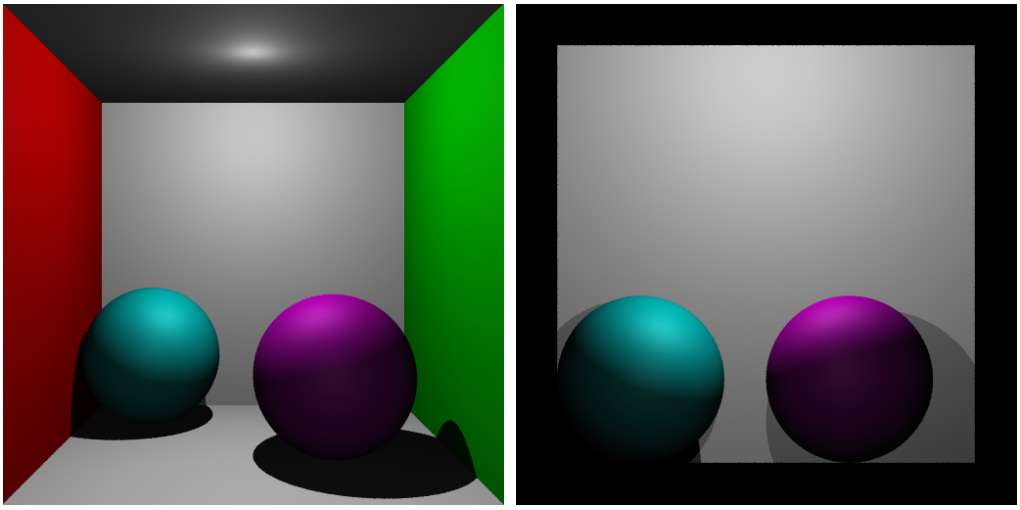
3.1 光线追踪

首先规定好照相机的各个参数，之后对每个像素计算出需要追踪的光线。对于每条追踪光线，求出它与场景内物体的交点并计算出局部光照，随后根据情况递归计算该点的反射及折射光强，直至反射/折射次数达到上限（本次实验中为5）。

3.2 正投影/透视投影

3.2.1 透视投影

由视点位置、方向及照相机的北方可以计算出视点前方虚拟屏幕的左上角及屏幕的单位向量。随后即可计算出虚拟屏幕中各个像素点的位置，再与视点位置相减即可得到



(a) 透视投影 (b) 正投影

图 1: 不同投影效果对比

追踪光线的方向。在实际使用过程中，发现存在着透视畸变的现象，曾考虑将其改为等分视角地跟踪光线，但发现效果较差。随后将视角从60度减小为45度后，畸变现象明显减小。

3.2.2 正投影

通过输入视口大小，以视点为中心，可以计算出视口屏幕的参数信息，从而可以生成一系列平行光线用于追踪。

正投影与透视投影的对比图如图1，可以看出在康奈尔盒场景中，正投影只能看到后面那堵墙，从而体现了该场景的正视图；而透视投影则像肉眼观察场景一样，显示了较为真实的效果。

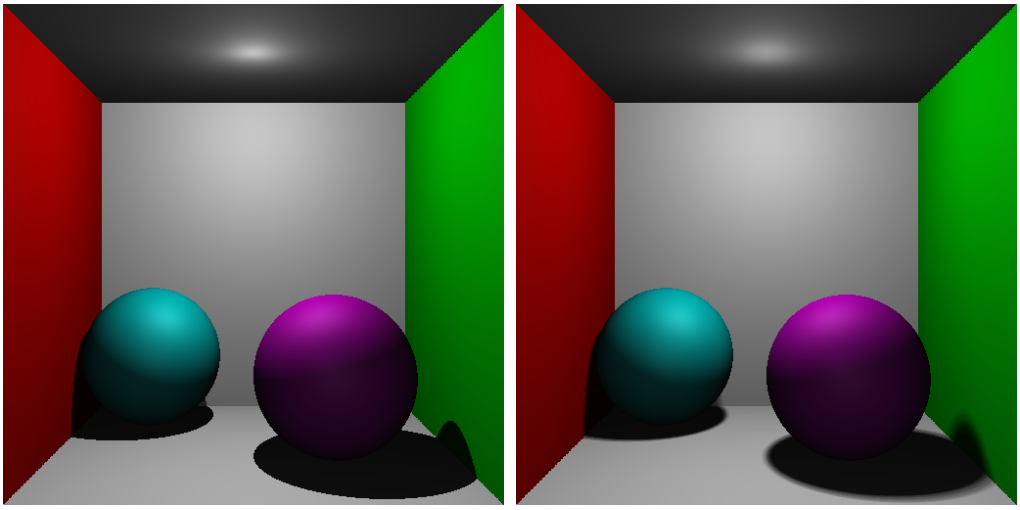
3.3 局部光照明

在本次实验中使用Phong光照模型[1]对物体进行着色。对于物体上每个点局部光照的计算公式如下：

$$I_p = k_a i_a + \sum_{m \in lights} (k_d(\hat{L_m} \cdot \hat{N}) i_{m,d} + k_s(\hat{R_m} \cdot \hat{V})^{\alpha} i_{m,s})$$

其中k_a, k_d, k_s, α为环境光反射系数、漫反射系数、高光系数、反光系数，$\hat{L_m}, \hat{N}, \hat{R_m}, \hat{V}$分

[1] https://en.wikipedia.org/wiki/Phong_reflection_model



(a) 没有软阴影 (b) 半径0.5，采样数10*10的软阴影效果

图 2: 软阴影效果对比

别为交点指向光源、交点处法向、入射光反射方向以及交点指向视点向量的归一化向量。

3.4 阴影

对于每个光线与物体的交点，需要从该点向所有光源发射阴影测试线，来判断该光源是否能照射到这一点。如果不能，则不计算该点的局部光照。

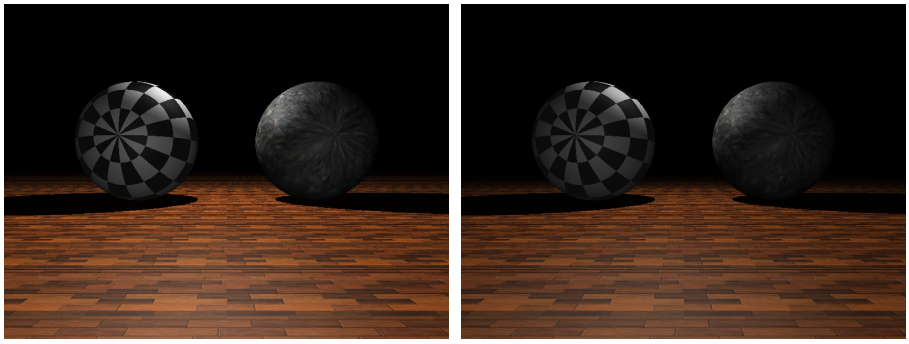
3.4.1 软阴影

为增加生成图像的真实感，考虑引入软阴影。通过在球面上均匀放置多个光强较小的点光源，从而达到模拟球体光源的效果。对比图见图2。

3.5 物体的表示及求交

无限平面 每个无限平面由一对法向及偏移量唯一构成。在判断交点位置时，首先判断光线是否与平面相交，之后通过计算光线起点在平面法线上的投影来得到结果。

球 每个球可以通过中心和半径来定义。可以通过上课讲过的几何法来判断光线与球的交点位置。



(a) 没有衰减 (b) 存在衰减

图 3: 光强衰减对比

三角形 用三个顶点来定义三角形。采用了Moller-Trumbore方法[2] 来实现求交，该方法在得到交点的同时，也获取了交点同三个顶点的系数关系，为之后法向插值做准备。

包围盒 这里采用与坐标轴平行的包围盒。采用了Andrew Kensler方法[3] 进行求交，该方法使用了较少的计算和比较，主要是无需对NaN 情况进行特判，减少了一部分的计算量。

3.6 光强衰减模型

使用Lambert定律[4]实现了对光强衰减的模拟计算，公式如下:

$$I_t = I_0 \cdot e^{-\tau} = I_0 \cdot e^{-\mu l}$$

其中，μ为衰减系数（实验中取为0.02），l为光线所走的距离。图3体现了有无光强衰减模型的区别。

3.7 读入网格模型

在本次实验中，读入网格模型（.obj文件）模块由自己完成，没有使用开源库。目前支持输入为三角网格的模型，且不支持纹理映射。在根目录下的"obj/"目录中存储的模型文件均可以被读取。

[2] http://pkuwwt.github.io/scholarship/2014-04-03-ray-triangle-intersection-tests-for-dummies/
[3] http://psgraphics.blogspot.com/2016/02/new-simple-ray-box-test-from-andrew.html
[4] https://en.wikipedia.org/wiki/Beer\T1\textendash\CJK@ignorespacesLambert_law

Time/s	build	render
100层 5个	1.56	2.90
100层 10个	1.51	2.96
100层 20个	1.31	3.21

表 2

3.8 纹理

本次实验中实现了三种纹理：纯色纹理、黑白格纹理及图像纹理。对于纯色纹理，不存在纹理映射的问题；对其余两种纹理，需要对物体的纹理映射进行定义。

对无限平面来说，首先计算出该平面的一组相互垂直的单位向量。那么对平面上任何一点，可以算出它相对这对单位向量的二维坐标，从而实现从三维点到二维坐标的映射。

对球上任意一点，可以将其转换为球面坐标。由于球面参数方程仅具有两个参数，所以对该方程反求出这两个参数，就可以得到纹理映射。

由于网格模型结构比较复杂，故没有实现在网格模型上的黑白格及图像纹理，仅支持纯色纹理。

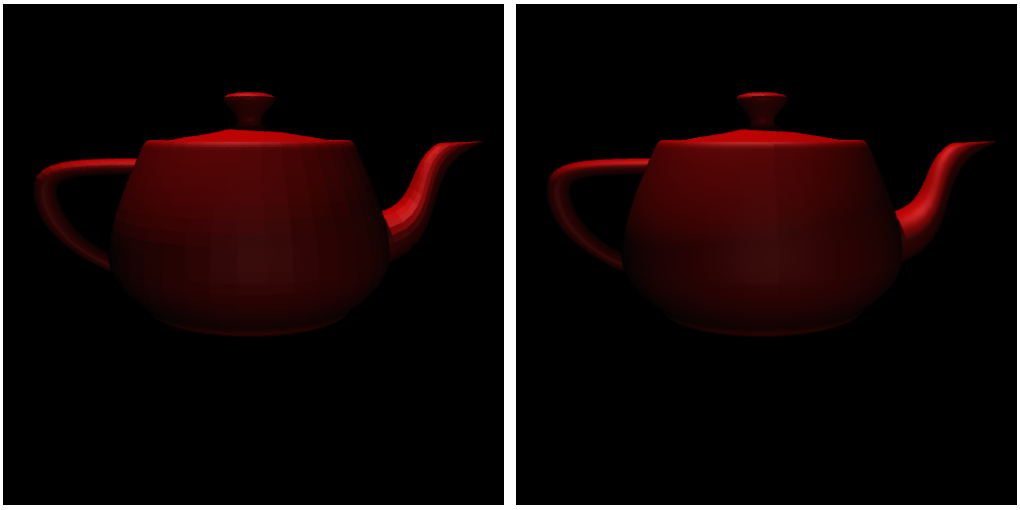
3.9 KD树

本实验中采用了基于SAH的KD树构建，构建算法参考了[1]的快速构建SAH-KD树算法，将建树算法的时间复杂度降低至$O(NlogN)$。基于表面积启发式的KD 树在查询操作上与基于中值切割的KD树相比，效率提高了许多，这主要是由于基于SAH的建树方法倾向于选择切分后遍历代价较小的切分平面，而不只是简单地选取包围盒中心的中值进行切分。

构建这种KD树时需要考虑两个参数：叶结点内最多管理物体的个数及KD树的深度。针对只有龙模型的场景进行建树并且渲染的测试，建树与渲染均为单线程，测试数据如表2，可以看出建树时间与渲染时间存在一个权衡。

3.9.1 遍历方法

对于KD树中每个内部结点，存放着一个分割平面P，结点的左、右儿子分别管理与左、右空间有交集（与P重合的物体作为特殊情况在[1]中有说明）的那一部分物体。光线与结点求交的过程可以分为如下几步：首先与分割出的左、右空间进行求交，如果仅



(a) 不光滑着色 (b) 光滑着色

图 4: 光滑着色对比

与其中一个空间有交，则只遍历那个空间；否则先遍历光线首先遍历到的空间，后遍历另外那半个空间。这样可以保证不漏掉任何物体，且不遍历明显没有交点的情况。

在本次实验中采用递归的方法对遍历进行实现。我尝试了将其改为非递归方法，但发现经过编译器优化后，表现反而不如递归方法，故仍采用原方法。

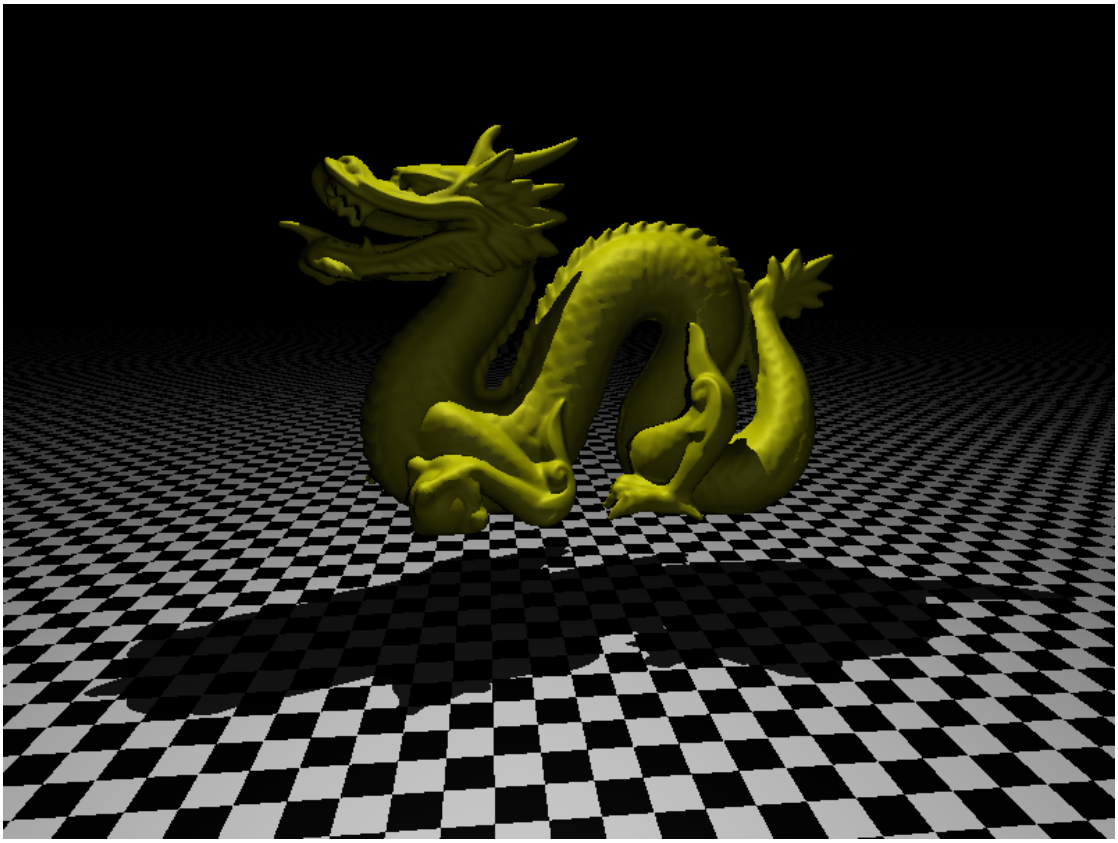
3.10 光滑着色

双线性法向插值方法进行光滑着色。首先对模型中顶点计算其相邻网格面的法向平均值，作为其法向，随后对网格面上任意一点，可以通过该面上三个顶点的法向插值得到该点的法向，在该点作为光线与物体的交点时，就可以使用该法向值进行局部光照的计算。对于犹他茶壶的对比图见图4，可以看出壶身和壶嘴部分的差距比较明显。

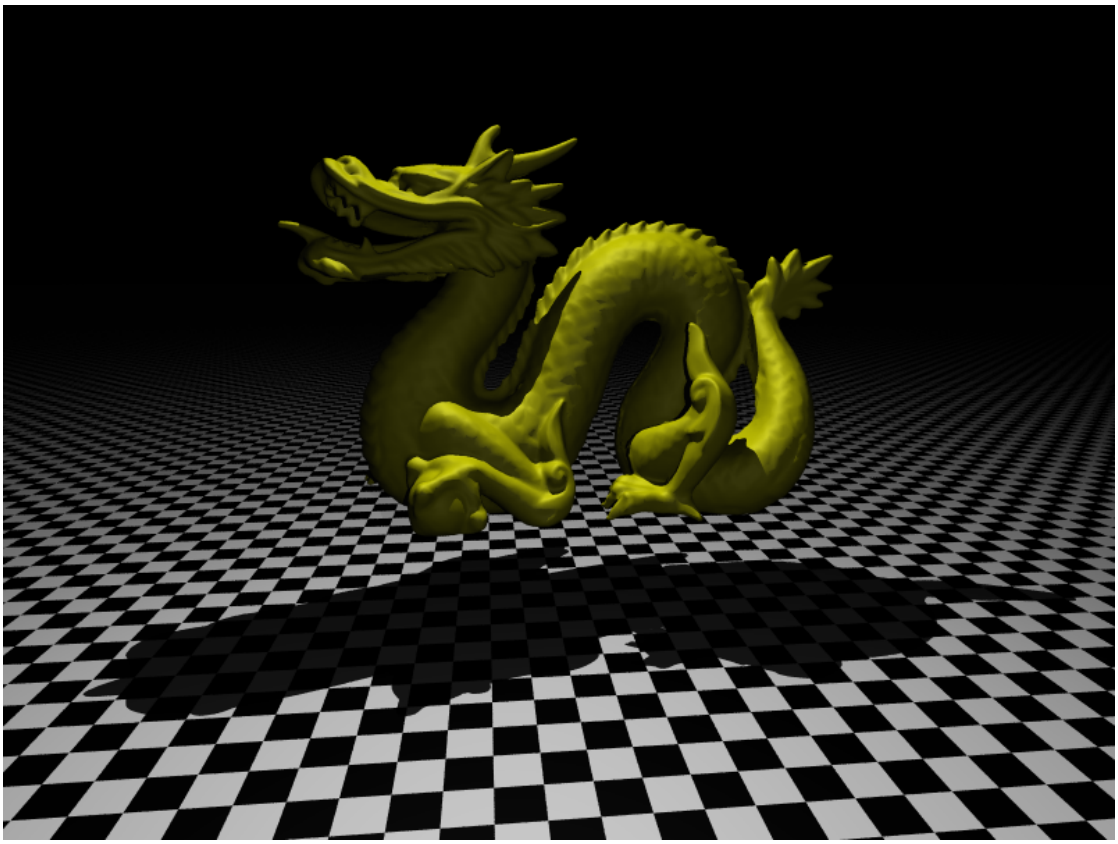
3.11 抗锯齿

采用抖动采样法进行抗锯齿处理。之前只是将虚拟屏幕中每个像素对应的小方格的中央点看做追踪光线所指的方向，现在需要多次在小方格中随机采样，分别追踪之，得到所有颜色值后取平均，也就近似获得了该像素的正确颜色值。

根据图5可以从地板纹理及龙模型的细节看出有无抗锯齿的差别。在没有抗锯齿的情况下，地面的黑白格纹理在远方变形得十分严重，这个问题在经过抗锯齿操作后得到了解决。

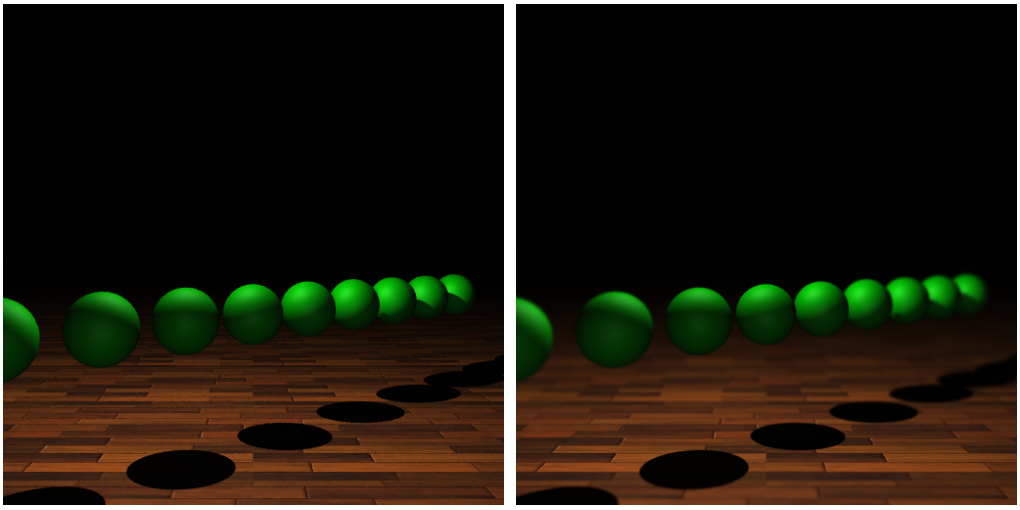


(a) 没有抗锯齿



(b) 抗锯齿

图 5: 抗锯齿对比



(a) 无景深 (b) 采样数30的景深效果

图 6: 景深效果对比

3.12 景深

根据照相机的参数，可以模拟景深效果，也即在焦平面附近的景物可以显示得较为清楚，过近或过远都会产生一定模糊的现象。对于虚拟屏幕中的某个像素，位置为P，视点中心位置为O，连接OP并延长，求出其与焦平面的交点T。之后在光圈内随机采样一些点A_i，需要追踪的光线即为$\overrightarrow{A_iT}$。根据图6可以看出加入景深效果后的渲染效果相较前者要真实许多。

3.13 并行加速

本实验中在建KD树及追踪光线这两处进行了CPU多线程加速，均使用VS2013所支持的openmp2.0实现。

3.13.1 建KD树

由于建树是一个递归过程，对每个结点会递归对它的左、右孩子递归进行建树操作。在实现时，首先获取执行计算机CPU的核数，随后每次进行递归建树操作时，如果分配给当前函数的核数大于1，则将核数平分给左、右两个建树过程；否则不采用多线程加速。

针对只有一个网格模型的场景进行KD树的建树测试，测试数据如表3，可以发现并行后的效率提高了约六成。

Build time/ms	teapot	bunny	dragon
Before parallelization	78	108	1507
After parallelization	48	69	948

表 3

Render time/s	teapot	bunny	dragon
Before parallelization	1.28	1.52	3.04
After parallelization	0.37	0.46	0.94

表 4

3.13.2 追踪光线

由于每个计算每个像素的颜色是独立的，可以使用openmp对for循环进行自动并行处理。这里如果直接使用openmp并行化，由于不同像素计算复杂度不同，可能会出现不同线程完成速度不同，从而导致资源的浪费。我采用的办法是对纵、横两个维度（$0 \rightarrow width, 0 \rightarrow height$）分别进行随机排列，再进行并行计算，这样就会在一定程度上减少资源的浪费情况，而且随机排列过程也不会占用太多时间。

针对只有一个网格模型的场景进行渲染测试，测试数据如表4，可以发现并行后的效率约是单线程的3倍。

4 文件结构

/
- exe ..... 可执行程序及依赖库
  - scene ..... 用于渲染的场景文件
  - objs ..... 网格模型文件
  - texture ..... 纹理图片文件
- src ..... 工程源代码
- diary ..... 开发过程及日志
- gallery ..... 质量较高的渲染图片
- report ..... 报告相关tex及资源文件
- report.pdf ..... 实验报告

5 实验感想

在完成本次作业的过程中，从框架的构建到具体实现这一过程锻炼了我的C++编程能力。在实现算法的过程中翻阅了许多与光线追踪算法相关的材料，不仅了解了关于光线追踪的知识，并且也复习到了许多光学的知识。本次实验有许多地方还没有做到完善，比如并行就只用到了CPU并行，没有实现在GPU上；没有实现全局光照等等。这些不足之处我会考虑在之后有时间会进行相应的尝试。

6 代码管理

代码全程托管在github上，在其中可以看到该程序从无到有这一步步的过程：
https://www.github.com/nero19960329/RayTracer

参考文献

[1] Ingo Wald and Vlastimil Havran. On building fast kd-trees for ray tracing, and on doing that in O (N log N). In *2006 IEEE Symposium on Interactive Ray Tracing*, pages 61–69. IEEE, 2006.