

清华大学计算机系  
计算机图形学基础

# 光线追踪及网格简化

## 作业文档

姓名 吴育昕  
学号 2011011271  
班级 计 14  
邮箱 ppwwyyxxc@gmail.com

## 目录

<b>1 简介</b>	<b>1</b>
1.1 依赖	2
1.2 编译	2
1.3 使用	2
<b>2 算法说明</b>	<b>2</b>
2.1 光线追踪及局部光照模型	2
2.2 全局光照模型	3
2.3 几何对象表示及计算	4
2.4 视图模型	4
2.5 KD 树	4
2.5.1 建树	5
2.5.2 求交	5
2.6 纹理	5
2.7 法向插值	5
2.8 抗锯齿	6
2.9 Phong 模型中的软阴影	6
2.10 景深	6
2.11 小量处理	6
2.12 网格简化	7
2.13 多线程	7
<b>3 项目设计</b>	<b>7</b>
3.1 目录结构	7
3.2 渲染物体相关类的设计	8
3.3 空间视图相关设计	9
<b>4 过程记录</b>	<b>10</b>
<b>5 References</b>	<b>23</b>

## 1 简介

本程序是一个 3D 渲染程序。选用 Phong 模型[1]作为局部光照模型或 path tracing[2]作为全局光照模型，渲染 3D 场景。支持平面、球、三角面片、三角网格(可从 obj 文件读入)几种几何对象，并可方便的扩展。渲染支持软阴影，抗锯齿，景深，自定义纹理等功能，并可对三角网格进行简化。三角网格，全局渲染，网格简化均采用数据结构(KD 树及堆)与多线程加速，效率很高。同时，将渲染功能嵌入了图形界面，可以支持 obj 文件的预览及简化。

## 1.1 依赖

1. 本程序用 C++11 编写,需要编译器支持 C++11 中的 ranged loop, initializer list, type inference 等语法, 且需标准库包含 `std::shared_ptr`, `std::future` 类. 建议使用 g++≥ 4.8 编译.
2. OpenCV2 <sup>1</sup>
3. ImageMagick <sup>2</sup>
4. Qt4 <sup>3</sup> (可选)

## 1.2 编译

在src目录中, 使用`make`命令和`make gui`命令分别编译命令行程序与图形界面程序.

## 1.3 使用

1. 命令行程序: 直接运行, 渲染一个演示场景. 可在`main()`函数中选择不同的场景.  
程序调用 opencv 进行图像显示, 显示时可通过键盘进行导航, 导航方法见下表:

	屏幕以视点到其连线为轴旋转,  顺时针 围绕固定中心旋转视点 视点及屏幕的平移 缩放 固定视点旋转视角 调节焦平面远近(景深模式下有用),  调远 保存当前图片至 output.png 输出当前视角信息 退出
--	--

表 1: 场景导航快捷键

2. 图形界面程序: 运行界面如图6所示, 运行后, 通过 open 按钮选择 obj 文件, trace 按钮渲染, smooth 控制是否开启法向插值. 其余按钮用于更改视角、渲染方式等参数, 改变视角后重新 trace 即可生效. simplify 按钮将 obj 模型按照给定的简化率进行简化, simplify rate 表示简化后保留的面片所占比例.

## 2 算法说明

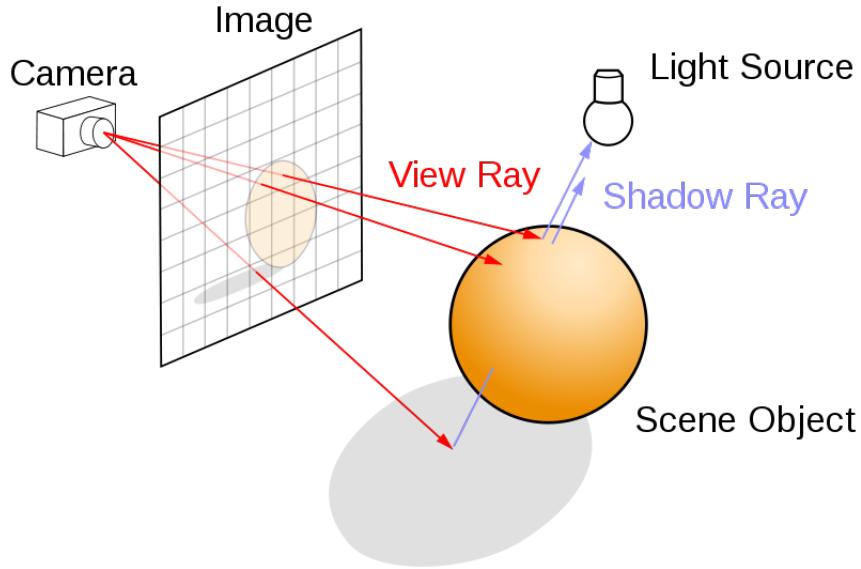
### 2.1 光线追踪及局部光照模型

光线追踪的基本原理如下图所示.

<sup>1</sup><http://opencv.org>

<sup>2</sup><http://wwwimagemagick.org/script/index.php>

<sup>3</sup><http://qt-project.org/>



选定视点位置及一观察屏,从视点到屏上各点发出光线与空间中物体求交. 在交点处根据局部光照模型计算颜色,再递归的计算反射、透射光颜色,混合后显示在屏上.

此程序使用的局部光照模型为 Phong 模型,其主要公式为[1]:

$$I_p = k_a i_a + \sum_{m \in lights} (k_d (\vec{L}_m \cdot \vec{N}) i_{m,d} + k_s (\vec{R}_m \cdot \vec{V})^\alpha i_{m,s})$$

其中  $k_s, k_d, k_a, \alpha$  分别为物体表面该点处的高光系数,漫反射系数,环境光系数,亮度.  $\vec{L}_m$  为该点指向光源的向量,  $\vec{N}$  为表面法向,  $\vec{R}_m$  为光源指向该点的光线经理想反射后的指向,  $\vec{V}$  为视点到表面交点处的向量. 实现见 `Space::trace()`

## 2.2 全局光照模型

程序还支持了基于 Path Tracing<sup>4</sup> 的全局光照模型,其渲染方程如下:

$$L(x \rightarrow v) = L_e(x \rightarrow v) + \int_{\Omega} L(\Phi \rightarrow x) F_s(x, \Phi \rightarrow v) \cos \theta_{\Phi} d\Omega$$

其中,  $L(a \rightarrow b)$  表示从  $b$  处观察  $a$  处所得结果,  $L_e$  为物体自身发光亮度,  $F_s(x, \Phi \rightarrow v)$  为物体表面 BRDF 函数.

采用 Monte Carlo Path Tracing[3]对此方程进行逼近,其基本方法是, 在光线与物体的交点处随机向其他方向发光线,递归求解. 随机时依据表面材质的不同选取不同的概率分布. 对于漫反射,遵循空间均匀分布,实现中在半球面中随机采样. 对于反射,分布近似一个尖峰函数,仅在反射方向上有概率. 对于透射,依照 Fresnel 方程[4]计算透射比与反射比,随机选择一者进行递归. 实现见 `Space::global_trace()`.

Monte Carlo Path Tracing 需要大量的采样才能够得到较好效果,因而其效率较低,一张质量较好的图需要二十分钟渲染. 但它得到的图像更真实,如图7, 图8, 图9.

<sup>4</sup>[http://en.wikipedia.org/wiki/Path\\_tracing](http://en.wikipedia.org/wiki/Path_tracing)

## 2.3 几何对象表示及计算

程序支持了平面、球、三角面片、包围盒四类基本几何物体, 物体都需要各自拥有与光线求交的方法.

**光线** 光线是一条射线, 包含一个起始点及方向. 见`include/geometry/ray.hh`

**无穷平面** 为了计算方便, 使用平面法向及平面到原点的距离作为确定平面的方式. 平面与光线求交时, 首先通过光线指向判断是否相交, 再通过光线在平面法向方向的投影长度计算交点. 见`include/geometry/infplane.hh`, `renderable/plane.cc`

**球** 球由球心及半径唯一确定. 球与光线求交时, 利用球心到它在光线所在直线上的投影的距离判断是否相交, 在根据投影位置及勾股定理计算交点. 求交时要考虑光线起始点在球内部的情形, 以供计算相对折射率. 见`include/geometry/sphere.hh`, `renderable/sphere.cc`

**三角面片** 三角面片用三个顶点坐标存储. 为了性能, 与光线的求交参考了[5, 6]的算法及实现, 其基本思想是求解满足方程

$$\overrightarrow{Orig} + t\overrightarrow{Dir} = x\vec{v_1} + y\vec{v_2} + (1 - x - y)\vec{v_3}, t > 0, x, y \in (0, 1), x + y \leq 1$$

的  $(t, x, y)$ , 在求交时同时能得到交点的重心坐标<sup>5</sup>, 便于之后进行法向插值. 见`include/renderable/face.hh`, `renderable/face.cc`

**轴平行包围盒** 轴平行包围盒用最小坐标与最大坐标两个向量存储. 为了效率, 包围盒与光线求交部分参考了[7]的算法. 其基本思想是对每个面逐一计算并更新交点. 见`geometry/aabb.hh`

## 2.4 视图模型

一个视图应包括视点及屏幕, 且应使视点在屏幕的中轴线上. 视图类`View`存储了视点坐标, 屏幕中心坐标, 屏幕尺寸, 屏幕边沿的空间指向, 并保证视点到屏幕中心的连线与屏幕边沿指向垂直. 这样可以方便的进行视图旋转, 视图平移, 缩放等导航操作. 见`include/view.hh`, `view.cc`

## 2.5 KD 树

KD 树是一种空间划分树, 原本用于在 K 维空间中快速查找点, 可利用在光线追踪中对物体及其包围盒进行索引. 基本方法是, 树中每个节点对应一个包围盒, 选取一个轴平行平面将包围盒一分为二作为两个子节点, 叶节点存储包围盒中的物体. 注意与切分面相交的物体在两子节点中都应维护.

---

<sup>5</sup>[https://en.wikipedia.org/wiki/Barycentric\\_coordinate\\_system](https://en.wikipedia.org/wiki/Barycentric_coordinate_system)

### 2.5.1 建树

传统的 KD 树中,按照使两边点的个数尽量接近的原则选取切分平面,这是由于假设了各个点被查询的概率均等. 在光线追踪中,一般采用面积启发式的平面选取方式[8],选取切平面使得两个子节点的包围盒表面积与包含物体个数的积之和尽量大,这样可以使 KD 树在查询时效率更高. 但启发式的建树需要枚举切分平面,计算包含物体个数,复杂度较高. 直接的枚举为  $O(n^2)$  复杂度,程序实现了[8]中提供的  $O(n \log^2 n)$  算法. 对于 20W 面片的龙模型<sup>6</sup>,采用不同方法单线程建树的用时及在几个固定视角渲染耗时如下(单位: 秒):

	建树	视角 1	视角 2	视角 3	视角 4	视角 5
二分建树(终止:100 层,15 个)	0.6	1.93	2.51	3.26	4.38	5.89
SAH 建树(终止:100 层,20 个)	5.41	0.29	0.37	0.45	0.59	0.78
SAH 建树(终止:100 层,15 个)	7.82	0.24	0.33	0.41	0.52	0.68

注: 1.建树时,以树深度及当前节点所管理的物体个数作为建树结束的判定条件.

2.此实验的视角 1 为 `main.cc` 中 `test_kdtree()` 提供的视角,其余视角由视角 1 zoom in 依次得到.

3.可在 `lib/kdtree.cc` 中通过注释 `KDTree::build()` 函数中相应代码切换两种建树算法.

由表可见 SAH 建树的查询效率有很大提高,但建树缓慢. 建树效率与渲染效率之间存在 trade-off,可以通过改变终止条件来调整.

另外,不使用 KD 树时,视角 1 渲染时间约为 800s.(不使用 KD 树时各像素所需时间大致相同,可由部分渲染时间估算总时间).

### 2.5.2 求交

求交的基本方法为,递归寻找两子树中最近物体的交点,取较近者为结果返回.

实现时,在每个节点处保存了当前节点的两个孩子的切分平面,这样可以预先判断出离光线较近的包围盒,若与其内物体相交则不用考虑另一包围盒. 使用此方法应注意,若光线与较近包围盒所管理的物体相交,应确认与最近物体的交点是否被较近包围盒包含. 因为若不包含,则光线首先打到的物体可能并不是此物体,而是第二个包围盒管理的物体.

## 2.6 纹理

一种纹理相当于一个二维坐标到表面属性的映射. 表面属性除 Phong 模型参数外,还包括了透明度,用于折射判定,以及发光强度,用于全局光照模型. 程序实现了均匀纹理、网格纹理、图片纹理三类纹理,并通过继承 `Texture` 类进行扩展.

对于一个物体,由其自己管理三维坐标到二维坐标的映射. 对于平面,采用平面上的二维欧式坐标. 球体采用其极坐标. 网格未支持二维纹理,仅可以使用均匀纹理.

### 2.7 法向插值

法向插值可以使三角网格表面更光滑,只需找到一个面片上的连续函数,就可以得到较好的效果. 此程序采用了线性插值.

在读入网格数据后,令各顶点法向为其相邻各面法向的平均. 在面片与光线求交后,根据交点的重心坐标及面片顶点的法向,即可插值出交点法向. 效果见图3, 图5.

<sup>6</sup>`models/fixed.perfect.dragon.100K.0.07.obj`

## 2.8 抗锯齿

1. 使用 Beer-Lambert 定律[9],使得光线亮度按传播距离指数衰减,可以有效消除远处纹理密集处的畸形. 见图1与图2的对比.
2. 使用全屏抗锯齿(FSAA),对图片整体应用卷积盒  $\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$ , 消除直线锯齿的同时使图片模糊,影响视觉效果.
3. 对每一像素,计算其与周围像素距离平方之和,若小于某一阈值则应用如上卷积盒,略有效果. 见CVRender::antialias()

## 2.9 Phong 模型中的软阴影

对场景中每一点光源,将其替换为点周围的多个密集点光源以模拟面光源的效果,即可实现软阴影.效果如图4所示,比起 Path Tracing 的软阴影的话还是十分不真实. 实现见Space::add\_light().

## 2.10 景深

程序中景深<sup>7</sup> 的实现方法为,以焦平面作为屏幕,取焦平面与视点之间某处建一感光器平面. 对视点到焦平面的每条光线,在其与感光器的交点周围随机采多个样本点,以这些样本点为观察点向屏幕同一位置发射光线并执行光线追踪,以各样本点颜色的平均值作为最终颜色. 这样就可以使得焦平面上物体清晰,而其余位置模糊.

main.cc中的dof\_ball\_scene()生成一个演示景深的场景,场景中可通过键盘控制焦平面位置,详见表1.demo 目录中有景深的演示视频.

随机取点会造成焦平面以外有无规律噪点,在生成视频后会比较明显,因而对输出图像统一做了高斯模糊,使噪点不太明显.

## 2.11 小量处理

在如下情形需要特别注意实数运算中的误差.

### 1. 反射折射交点

考虑光线斜射平面的情形,若交点由于误差落在了平面异侧,则反射光仍会打到平面,若落在了平面同侧,则透射光仍会打到平面.因而计算反射光线时,应将其起始点回退EPS,计算透射光线时,应将其起始点前进EPS.

### 2. 平行判定

判定直线与面片或平面是否平行时,利用直线与法线点乘的绝对值 < EPS 判定,否则可能导致交点坐标过大.

### 3. KD 树

---

<sup>7</sup>[http://en.wikipedia.org/wiki/Depth\\_of\\_field](http://en.wikipedia.org/wiki/Depth_of_field)

建树时,对于包围盒相交的判定应略微宽松,与包围盒距离  $< \text{EPS}$  的物体都应归入包围盒管理,查询时对光线与面片求交也可判的宽松一些,否则渲染的图片中容易出现黑点.对面片求包围盒时应注意最小值应减去 EPS,最大值应加上 EPS,否则可能出现 0 体积包围盒,影响算法实现.

## 2.12 网格简化

网格的坍缩简化算法参考[10]实现. 基本思路是,对每对顶点估算出简化代价,每次选取代价最小的顶点对执行简化操作,操作后更新相关的代价值.

由于算法具有优先队列结构,因此使用了 `std::priority_queue` 进行堆加速. 但由于需要执行堆元素修改操作,因而对堆结构进行了如下处理:

对每个顶点,维护它相邻顶点中最适合坍缩的一个顶点指针. 堆元素存储一顶点指针及它与相应邻点的坍缩代价,另有一时间戳. 堆元素按照坍缩代价保持堆性质,坍缩一对顶点后更新了附近顶点的最优代价,就将新的值打上新的时间戳压入堆中,而取堆顶元素时,若从时间戳发现其不是最新就抛弃.这样就可以替代修改操作.见 `include/mesh_simplifier.hh`, `mesh_simplifier.cc`

对于 20 万面片的龙模型<sup>8</sup>, 简化掉 80% 的面片需要 2.37s, 而不用堆加速时需要 55s. 效果见图5及 simplified 目录中的图片.

## 2.13 多线程

1. 对生成图片中每个像素,显然其计算相互独立,使用 openmp 及 C++11 的 `std::thread` 两种方式对其进行多线程优化,效果差不多. 见 `CVViewer::render_all()`.
2. 建树时,两子树的创建可以并行执行. 测试表明对根节点的两子节点并行建树有效,而对深层节点并行建树得不偿失. 另外,  $O(\log^2 n)$  的建树中有一步需对各物体每一维度上的坐标排序,这是效率的瓶颈所在. 不同维度的排序可以并行执行,经测试也只对根节点比较有效. 这两处并行都使用 C++11 的 `std::future` 实现,效率提升了约 30%.
3. 网格简化中,对各节点代价的更新可以并行完成,效率略有提高.

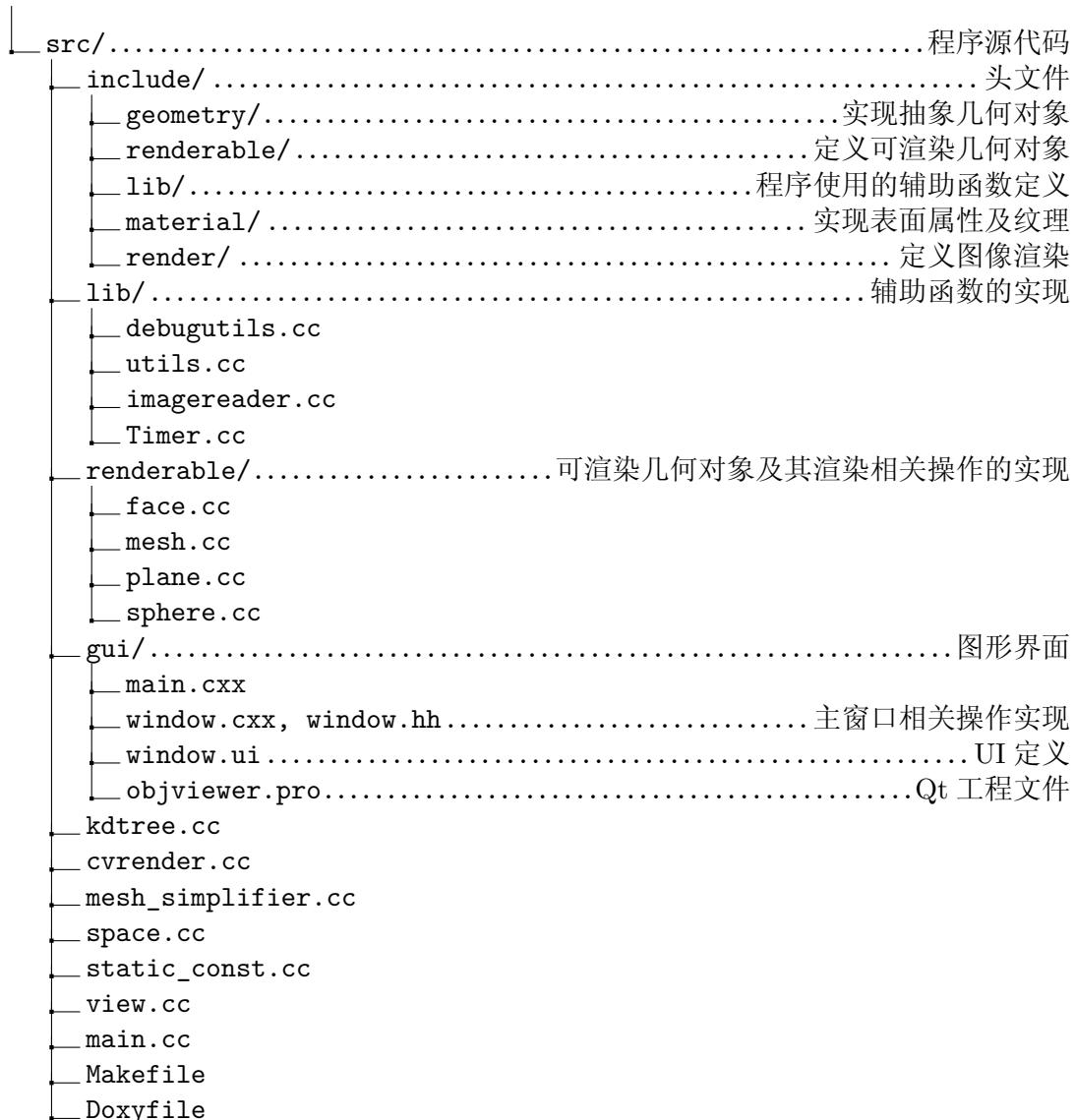
# 3 项目设计

## 3.1 目录结构

项目的目录结构大致如下:

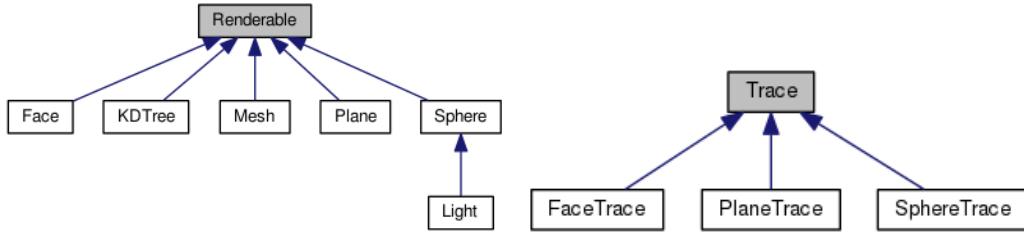


<sup>8</sup>models/fixed.perfect.dragon.100K.0.07.obj



### 3.2 渲染物体相关类的设计

所有可渲染物体,包括平面、球、面片、网格、KD 树,均继承自 `Renderable` 基类,当其需要与光线求交时,通过 `Renderable::get_trace()` 返回一个 `Trace` 类的子类对象指针,由 `Trace` 类完成求交相关的操作.一个 `Trace` 对象相当于一个物体与一条光线的组合.



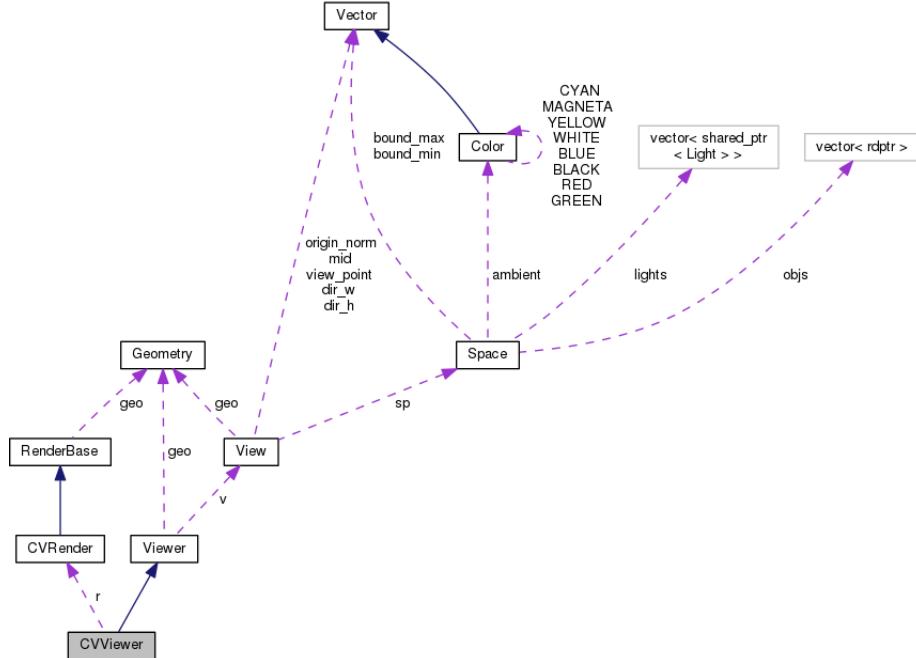
上图是**Renderable**与**Trace**的继承图,其中**Light**继承自**Renderable**是因为全局光照模型中需要对光源求交,程序中只支持球形光源(在局部光照模型中将球形光源看做点光源).  
**Trace**仅有三个子类是由于**Mesh**返回**FaceTrace**对象指针, **KDTree**返回它所管理物体对应的**Trace**对象指针.

**Renderable**类只有获取物体表面纹理及获取包围盒两种方法, **Trace**类包括了判断相交、求交点法向、交点表面属性、交点前方介质密度等方法.

这样做好处是,由**Trace**对象自己管理求交过程的中间结果,保留有用的结果以备其他方法使用,节省了计算资源. 如判断是否相交时一些中间结果可能在计算交点距离时使用,交点法向方向有可能被计算纹理映射坐标时使用,这些中间结果是每一对(物体,光线)特有的,又不该对外界暴露,因而用**Trace**类将其封装.

同时,这种设计使得同样的**KDTree**类只需接受一个**Renderable**对象的集合,就可以很好的管理物体,实现了**KD**树的数据结构在网格以及在整个空间中的复用,也能够支持**KD**树的嵌套.

### 3.3 空间视图相关设计



空间**Space**类为一系列物体及光源的封装.

视图**View**类为一个视点及屏幕的组合,负责生成光线并调用**Space::trace()**获取相应颜色.  
**Viewer**类提供用户操作的接口,根据用户操作调用**View**类的一系列改变视角的方法??节,同时调用**RenderBase**类进行显示.

## 4 过程记录

本项目的详细开发记录可以通过 git 查看. 项目托管在 github<sup>9</sup> 及计算机系 git<sup>10</sup>上,并于 6 月 24 日开源.以下是一些重要记录.

1. 首次出现正常的图片. 用了一个  $x - y$  平面上的无限大平面及一个点光源。此时仅仅参考[1]考虑了基本 Phong 模型中的漫反射与环境光,已经可以看到左下部分亮度较高,符合预期。

另外,远处的黑白纹理误差较大,不知是不可改进的浮点误差还是我的处理有误。

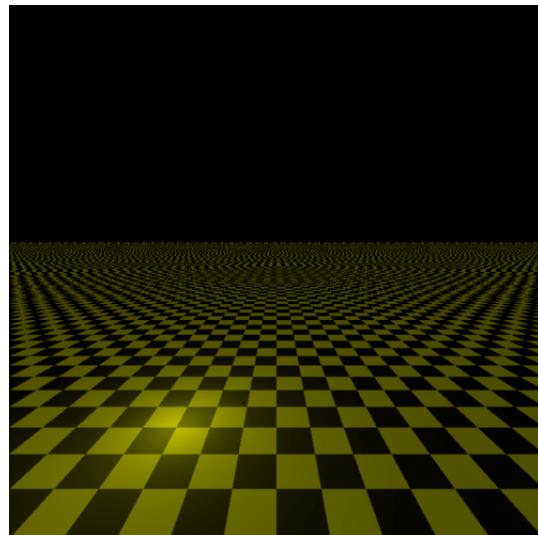


图 1

2. 依照 Beer-Lambert 定律[9]对光线能量进行了按距离的减弱:

$$E = E_0 e^{distance*density}$$

使得远处的纹理更自然.

---

<sup>9</sup><https://github.com/ppwwyyxx/Ray-Tracing-Engine>

<sup>10</sup><http://git.net9.org/ppwwyyxx/ray>

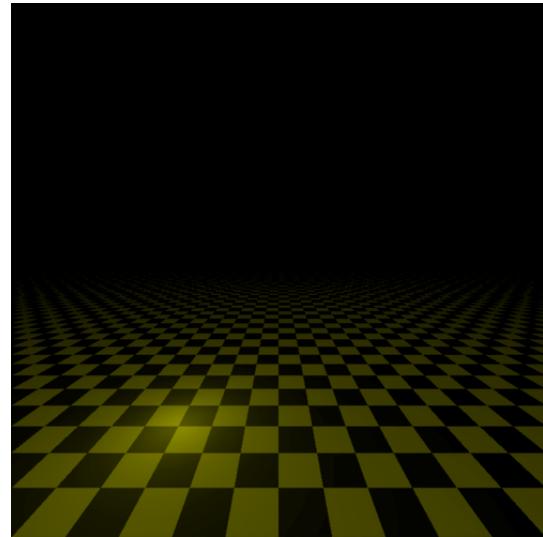
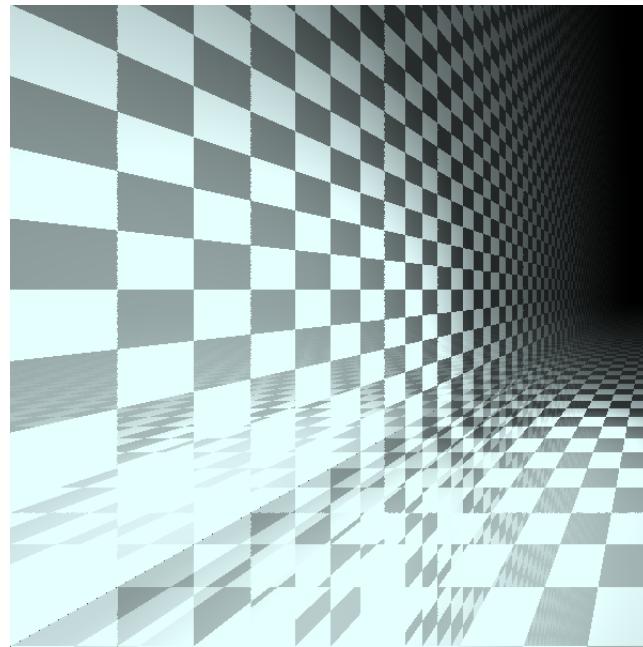


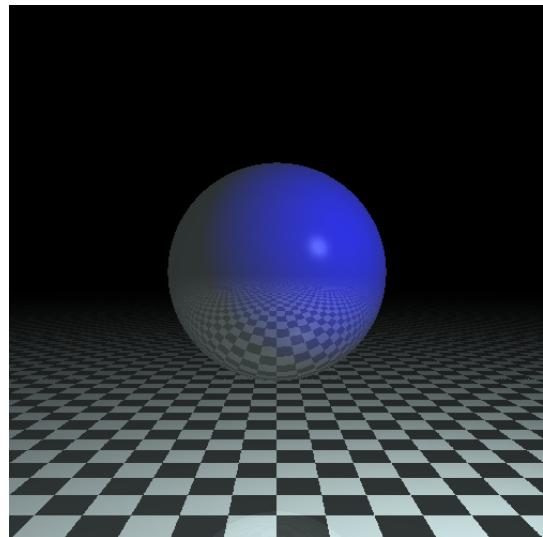
图 2: Beer-Lambert 定律效果图

3. 考虑了 Phong 模型中的高光,并对  $x - y$  平面和  $y - z$  平面加入了反射系数,有了互相反射的效果,并且左下部分能够看到高光.打开-03编译开关时,在我的机器上渲染一张  $600 \times 600$  的图片需要 0.39s.

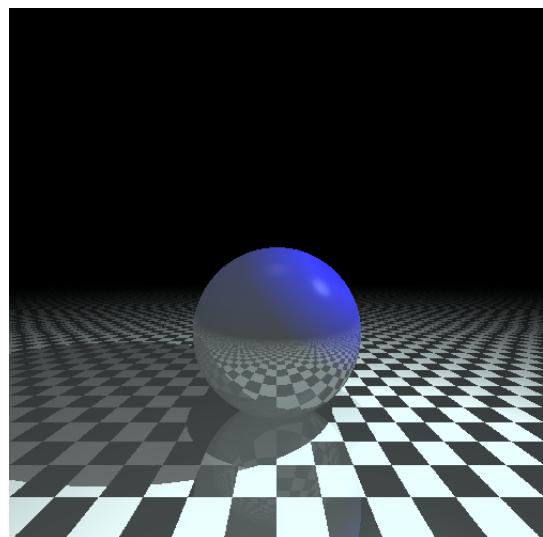


4. 使用球模型,可以看到明显的高光效果。同时由于球面specular参数高,使得球下部

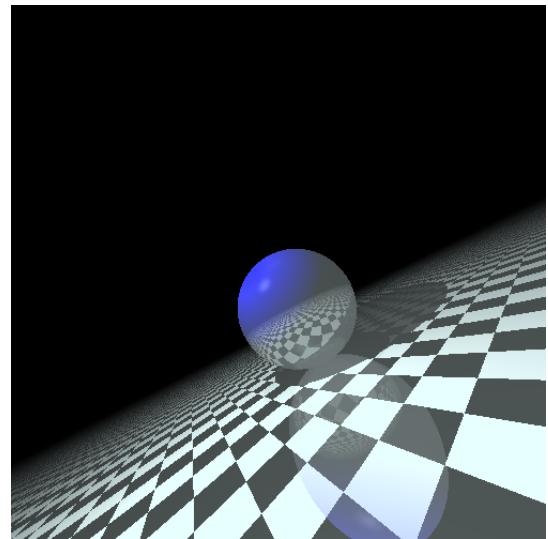
反射了平面。



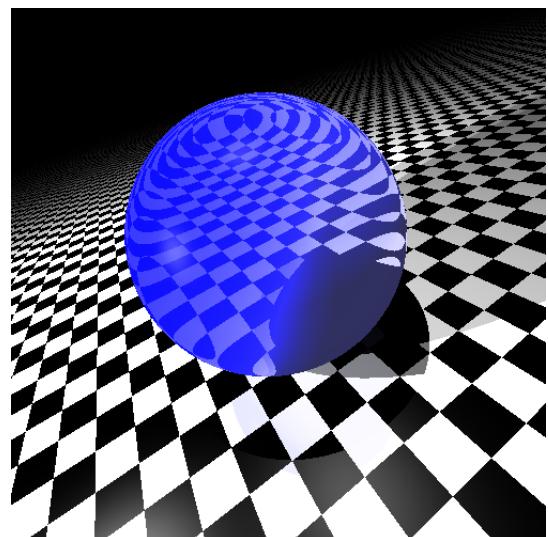
5. 对于找到的交点,判断它与光源之间是否被挡住,若被挡住就不计算漫反射和高光. 这样实现了阴影效果.



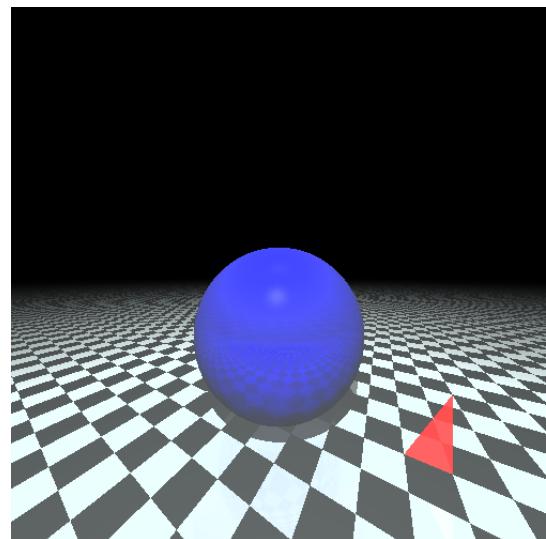
6. 对根据视点及对象生成视图(View)的方案进行修改,以支持视图的旋转,缩放.并利用 opencv 的 key event 实现了 gui 的旋转,缩放控制.



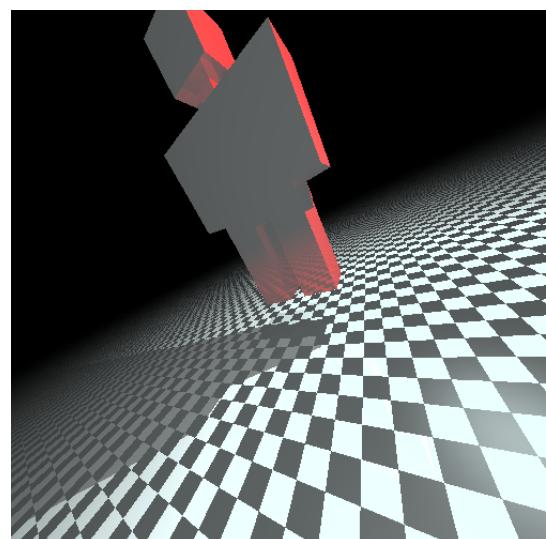
7. 加入了透射功能,依照预定义的介质密度及折射定律计算出射光方向.



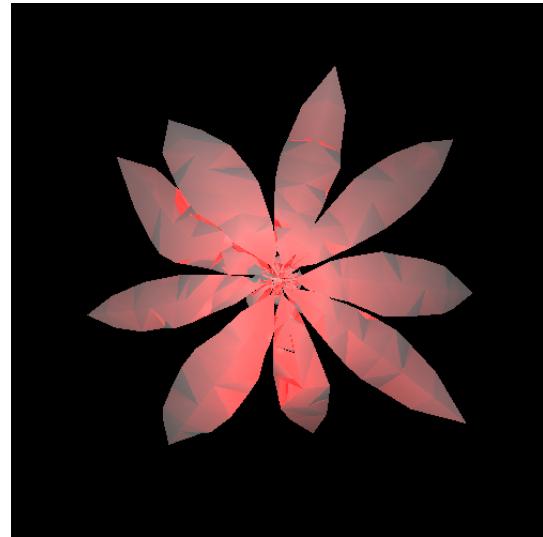
8. 实现了三角面片的渲染,在求交时计算齐次重心坐标,为网格中的法向插值做准备.



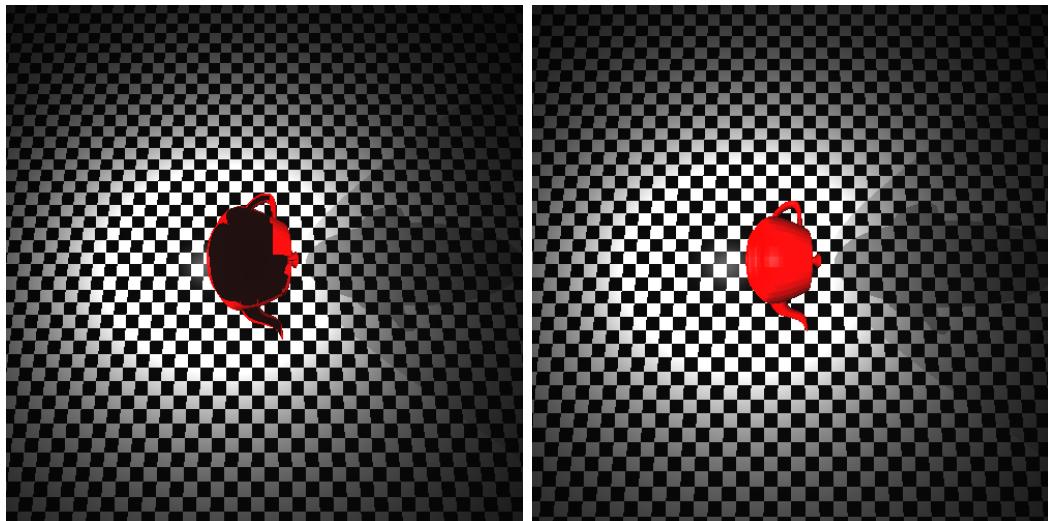
9. 实现了 obj 格式读取及基本的渲染,绘制出了一个红色小人.



10. 实现了 KDTree, 可以开始渲染更大的 obj.



11. 按照[8]优化了 KDTree 的实现之后发现如下左图所示 bug, 调试很久后发现两个原因. 一是建树时选取候选切割平面未偏移EPS, 二是包围盒求交存在小 bug, 少了一个绝对值运算. 其中第二个 bug 还严重影响了速度, 修复后的 KDTree 相比不用 KDTree 有了上千倍的速度提升.



12. 实现了法向量插值, 详见[2.7节](#). 插值前后的效果如下所示:

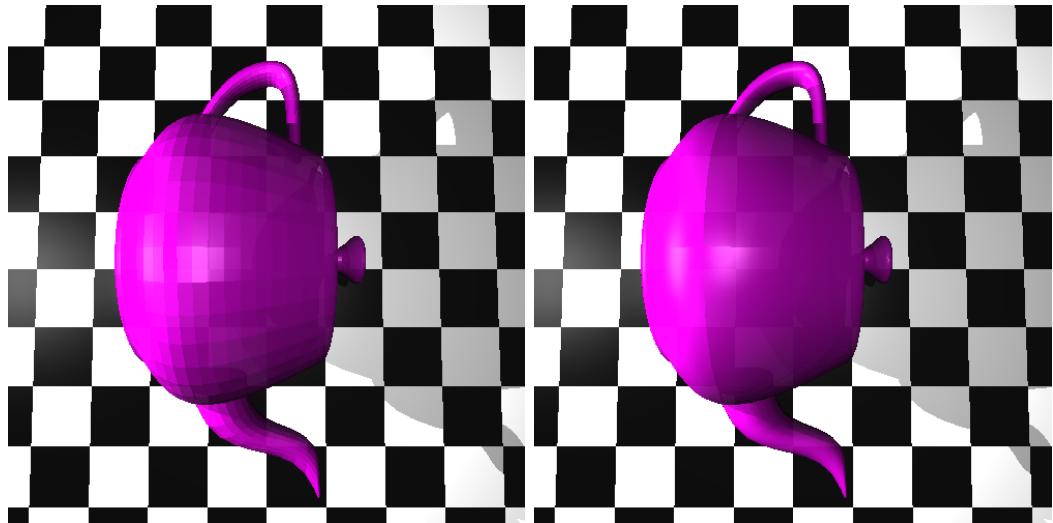
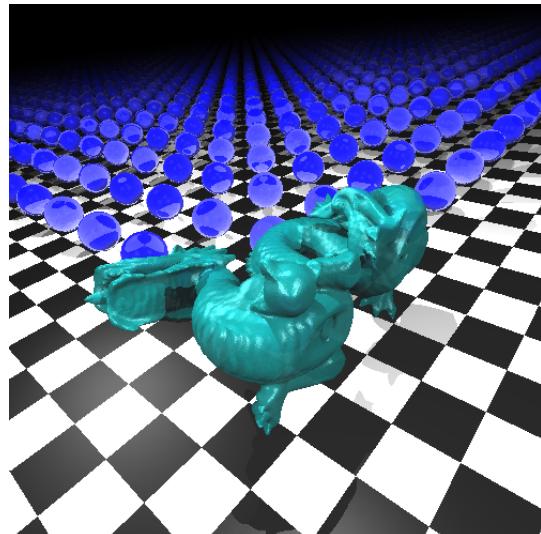
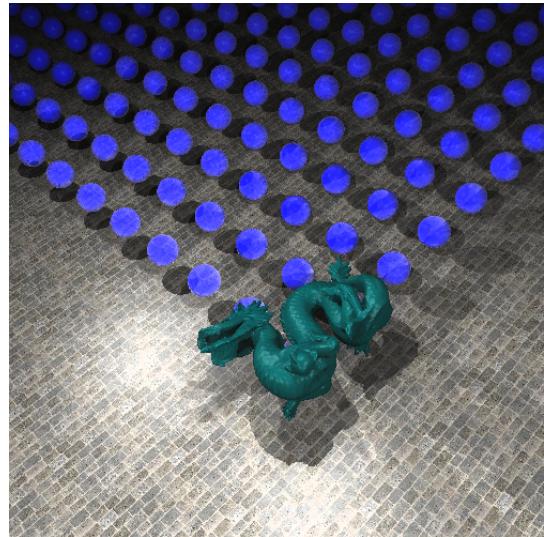


图 3: 插值的茶壶

13. 将 KDTree 扩展到了整个空间. 设计方法是将所有的有限大物体合成为一个“KDTree”物体(因为几何物体及 KDTree 都继承自**Renderable**基类),保留无限大物体,再在空间中进行渲染.下图是一个 20W 面片的龙加上 1 万个球组成的场景,渲染此图只需 1.4s.



14. 加入了图片纹理,设置在平面上.(图中亮斑为点光源所致)



15. 加入了软阴影效果,详见[2.9节](#).将每个点光源变为 20 个密集点光源可使左图场景渲染出右图效果. 左图渲染时间为 0.15s, 右图为 1.06s

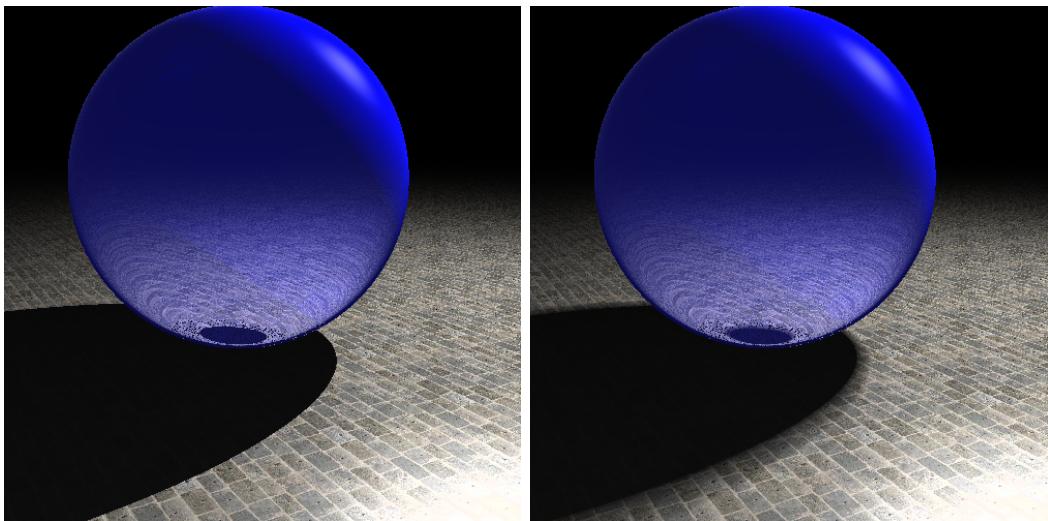
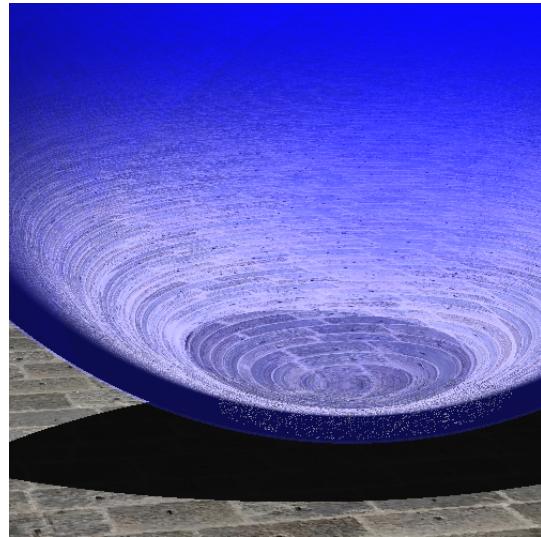
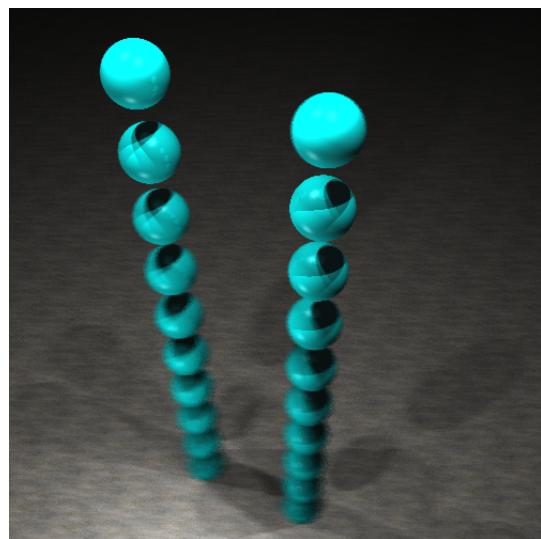


图 4: 软阴影效果图

16. 加入了抗锯齿效果,能看出变化但效果不明显. 为了图片清晰将此功能关闭.
17. 发现球与地面接触处有密集的小斑点,调试后发现是由于球底部与平面略微重合,求交时有时会出错.



18. 加入了景深效果,在感光器处随机取了 20 个伪视点,效率也相应降低了 20 倍,效果如图所示.



19. 由于`std::shared_ptr<T>`的默认构造函数会对引用计数和指针对象进行两次内存分配,效率不够高. 大量换用`std::make_shared<T>`后光线追踪时间缩短了 40%.
20. 利用 C++11 中的`std::future`实现了 KDTree 树构建的多线程,建树提速 30%.
21. 实现了网格简化,并利用堆加速. 对 20 万面片龙,效果如下.



图 5: 网格简化效果图

22. 发现对于某些模型,简化时会触发**assert**,调试后发现是由于网格坍缩时可能会导致一个面片中三点共线从而无法正确计算出法向. 将其修正为:若坍缩后面片三点共线,则保留原法向.
23. 发现在调用**Space::add\_obj()**后,若在调用栈中将**Mesh**析构,则会发生段错误. 调试很久后发现是由于**Face**类中存储了指向其宿主的指针**Mesh\* host**,用于获取插值及纹理信息用,此指针在**Mesh**复制时未被正确更新导致得不到其宿主.
24. 利用 Qt4 实现了一个简单的 GUI,支持单个 obj 的预览,视点移动,网格简化.

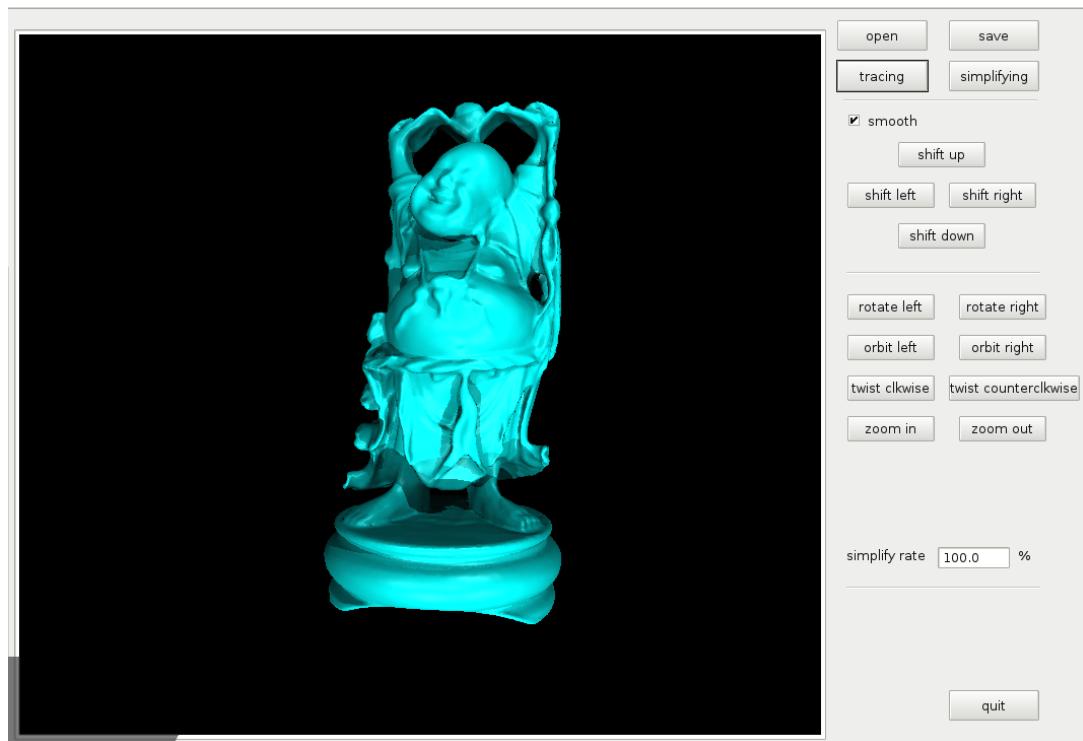


图 6: 图形界面展示

25. 实现了一个简易的全局光照模型,首先按照均匀分布支持了漫反射,渲染效果如下. 可以看到颜色变化及阴影都比右图的 Phong 模型更自然.

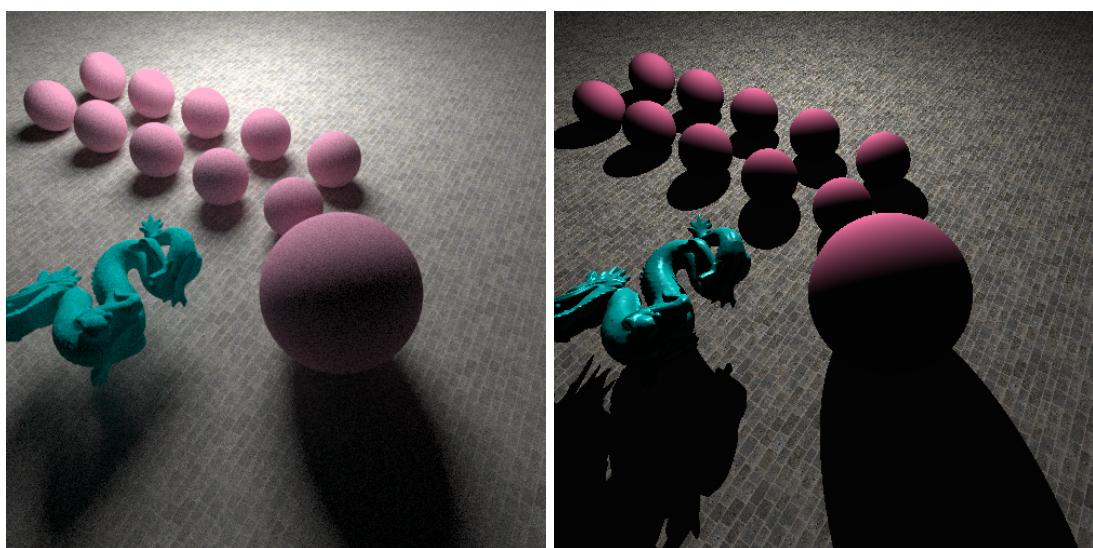


图 7: Path Tracing 漫反射效果图(左)

26. 为全局光照模型加入了镜面反射效果,若干个镜面的球上可以看到光源的亮斑及互相的反射,如下左图.

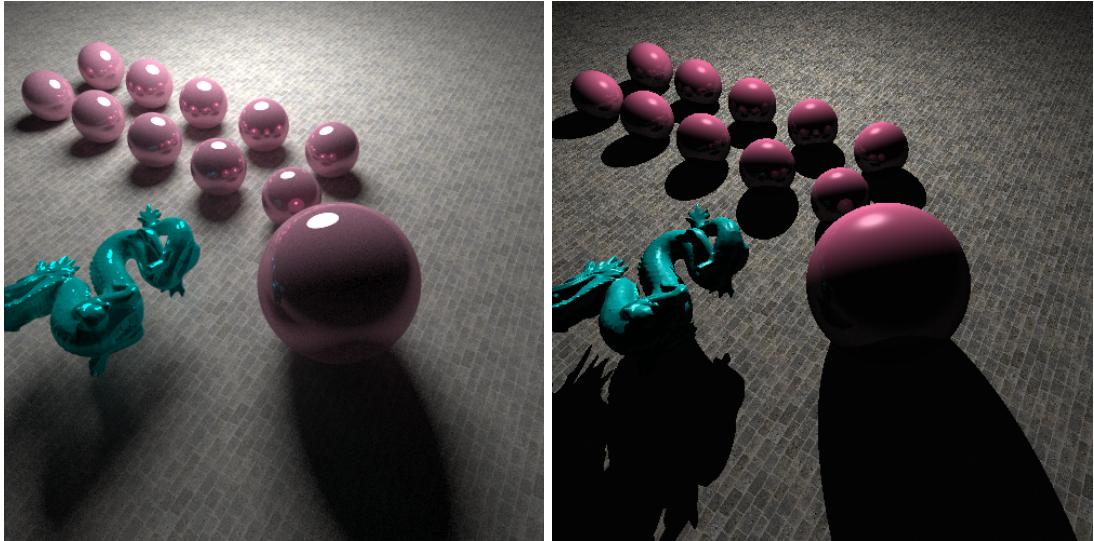


图 8: Path Tracing 镜面效果图(左)

27. 加入了全局光照模型的折射效果,结果很逼真,可以看到球下方的 caustic 效果,Phong 模型通常无法模拟此效果.

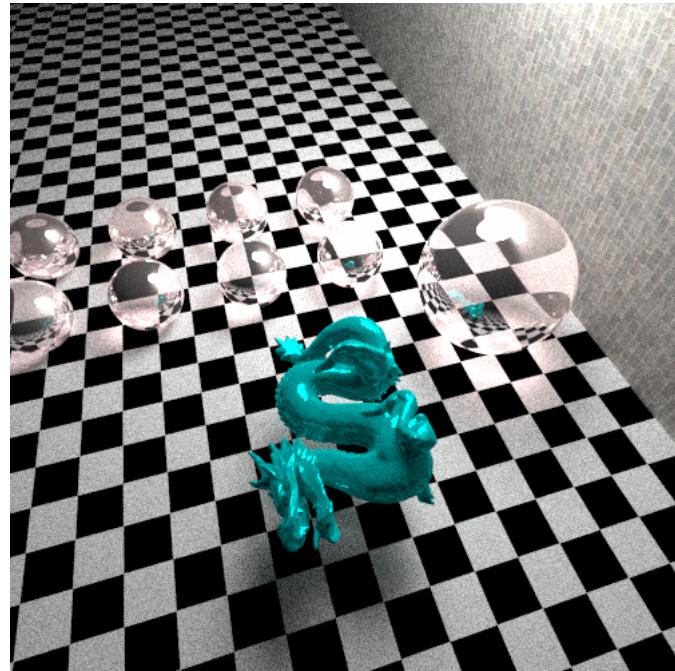
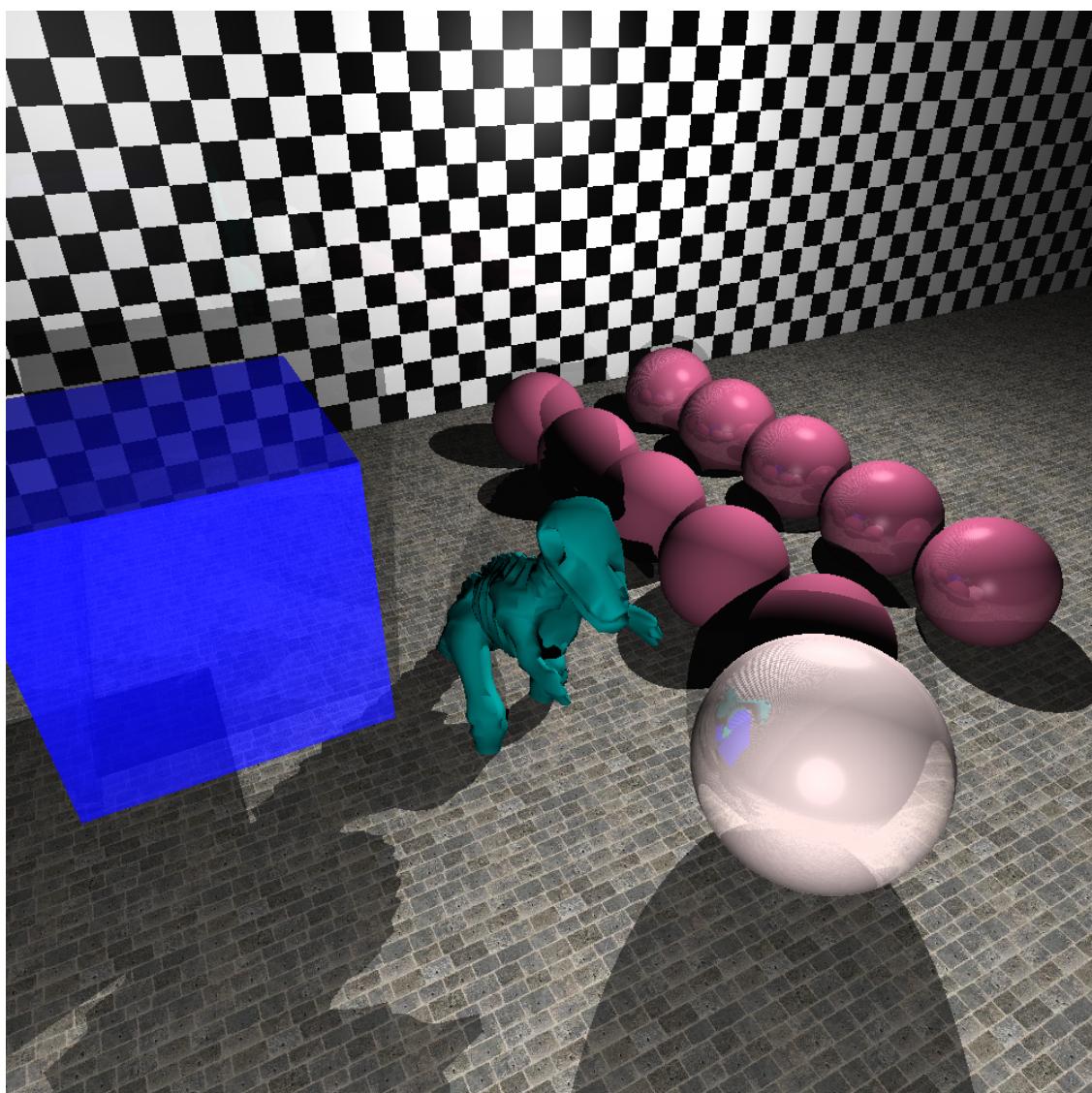
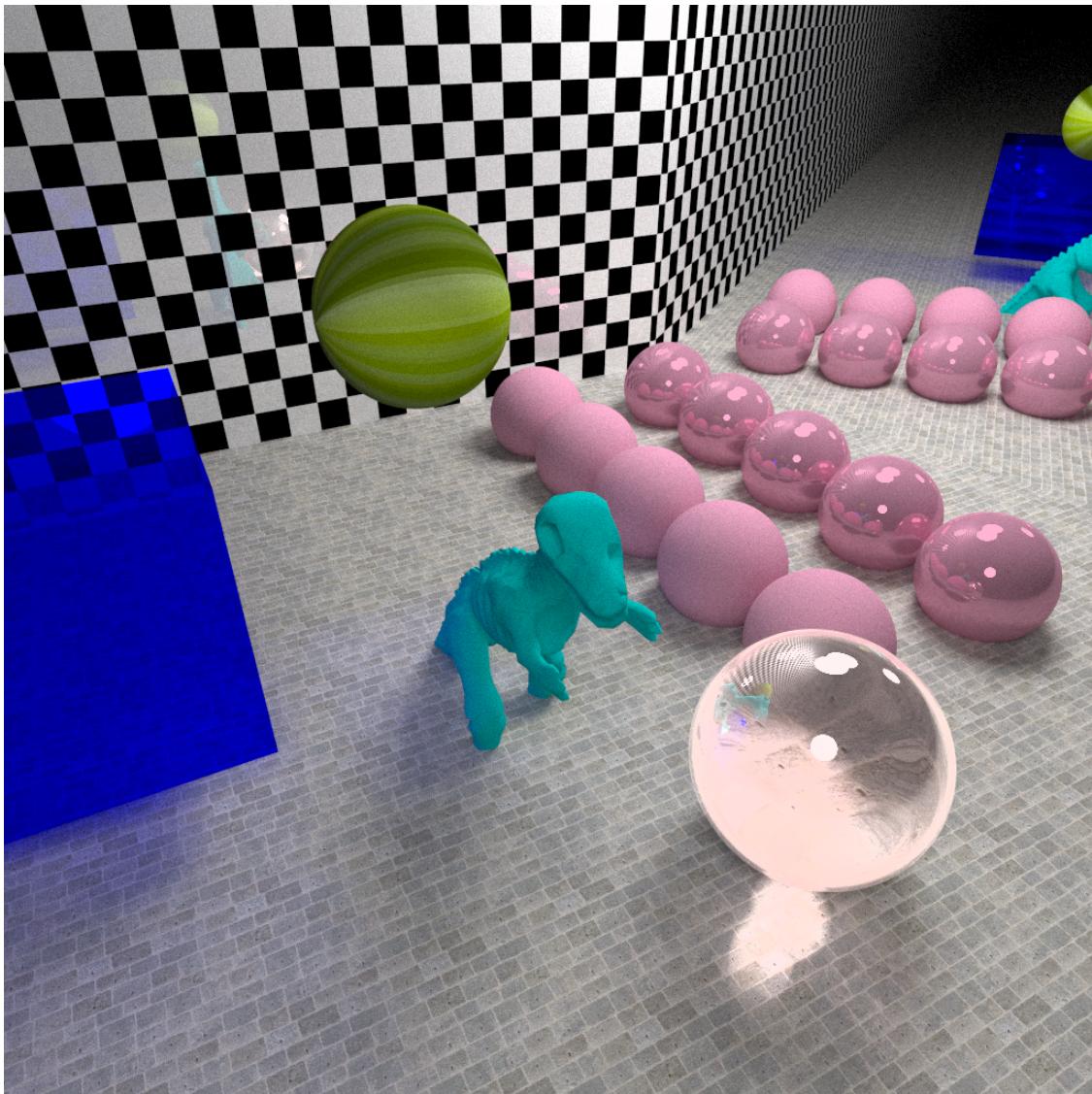


图 9: Path Tracing 透射效果图

## 两张效果图





## 5 References

- [1] Wikipedia page for Phong Reflection Model. URL: [http://en.wikipedia.org/wiki/Phong\\_reflection\\_model](http://en.wikipedia.org/wiki/Phong_reflection_model).
- [2] Wikipedia page for Path Tracing. URL: [http://en.wikipedia.org/wiki/Path\\_tracing](http://en.wikipedia.org/wiki/Path_tracing).
- [3] Jukka Koivisto. “Monte Carlo path tracing”. In: () .
- [4] Wikipedia page for Fresnel Equations. URL: [https://en.wikipedia.org/wiki/Fresnel\\_equations](https://en.wikipedia.org/wiki/Fresnel_equations).

- [5] Brian Curless. *Ray-triangle intersection*. URL: [http://www.cs.washington.edu/education/courses/cse457/09au/lectures/triangle\\_intersection.pdf](http://www.cs.washington.edu/education/courses/cse457/09au/lectures/triangle_intersection.pdf).
- [6] *Lighthouse3d Page for Ray-Triangle Intersection*. URL: <http://www.lighthouse3d.com/tutorials/math/ray-triangle-intersection/>.
- [7] Amy Williams et al. “An efficient and robust ray-box intersection algorithm”. In: *ACM SIGGRAPH 2005 Courses*. ACM. 2005, p. 9.
- [8] Ingo Wald and Vlastimil Havran. “On building fast kd-trees for ray tracing, and on doing that in  $O(N \log N)$ ”. In: *Interactive Ray Tracing 2006, IEEE Symposium on*. IEEE. 2006, pp. 61–69.
- [9] *Wikipedia page for Beer-Lambert Law*. URL: [http://en.wikipedia.org/wiki/Beer-Lambert\\_law](http://en.wikipedia.org/wiki/Beer-Lambert_law).
- [10] Stan Melax. “A simple, fast, and effective polygon reduction algorithm”. In: *Game Developer* 5.11 (1998), pp. 44–49.