光线跟踪实验报告

> 罗崚骁 计75
>
> 2017011364

github 仓库

目录结构

- `src`：所有源代码，用 rust 语言编写
- `results`：生成的全部图片
- `task`：场景输入文件，json 格式
- `asset`：贴图，三角网络等资源文件

测试方法

需要提前安装好 rust 工具链。

首先确定想要运行的测例名，比如 `example`，使用如下命令

```
cargo run --release example
```

然后程序会从 `task/example.json` 中读取配置文件，然后将结果输出到 `output/example.png` 中。

得分项总述

- 算法选型
  - 路径追踪
  - 渐进式光子映射（主要）
- 抗锯齿
- 纹理贴图
  - UV 纹理映射
- 软阴影
- 三角网格
  - 读写
  - 求交加速
    - 长方体包围盒
    - kd 树
  - 法向插值
- 旋转贝塞尔曲面
  - 解析法求交（阻尼牛顿迭代）
- 其它
  - rayon 多线程库加速

算法选型

本次实验使用的绘制算法有路径追踪（PT）和渐进式光子映射（PPM）算法，渲染图片时主要使用的 PPM 算法，渲染涉及到景深的图片时使用 PT 算法。

PT

PT 和 RT 相同的在于，与物体相交时需要根据物体的表面材质来计算光线下一步的传播方向，比如对反射和折射的处理基本上是一致的。

不同在于对漫反射的处理。对于 RT 中遇到漫反射表面时，直接遍历每个光源并计算局部光强；但 RT 中就没有这样的过程，而是光线会在漫反射时以一定的概率停止，有一定的概率会继续传播，如果继续传播的话，则按照某种分布随机选择一个方向。每次遇到一个物体，累加上对应的光亮度。

代码参见 `scene/rederer/pt.rs`。

PPM

PPM 是基于 PM 的改进。基本思路都是从光源发射出许多光子，用这些光子计算对每个点光亮度的贡献。不同在于：PM 首先进行光子发射的操作，建好光子图后再进行一轮光线跟踪渲染图片；而 PPM 变为首先进行一次光线追踪，找到所有的“视点”（可被每个像素发出的光线看见的位置），之后再不断地一轮一轮地发射光子，用这些光子不断地更新图片。PPM 的优势在于只用到了有限的存储，理论上只要时间足够可以生成任意精度的图片。

视点的数据结构如下：

```
```

```rust
struct ViewPoint {
    pub pos: Vector3f,
    /// 指向观察者的方向
    pub dir: Vector3f,
    // 该视点来自哪个像素
    pub pixel: (usize, usize),
    /// 光线从眼睛出发到此处（计算过 BRDF 后）的衰减系数
    pub weight: Color,
    /// 累计光子数
    pub n: usize,
    // 当前半径
    pub radius: FloatT,
    // 当前累计通量
    pub flux: Color,
}
```

光子的数据结构如下：

```rust
struct Photon {
    pub pos: Vector3f,
    // 当前光子携带的通量
    pub flux: Color,
    // 光子来的方向，用于计算 BRDF
    pub dir: Vector3f,
}
```

所有的视点直接使用线性列表存储；每一轮发射完光子后，要对所有的光子建立一个光子图，这里使用了 kd 树维护光子图，然后对于每个视点在一个球内寻找所有的光子并累计它们的贡献。根据相关结论，光子映射算法当光子密度趋于无穷时收敛，因此 PPM 希望随着迭代不断进行，每个视点搜索光子的半径逐渐减小，但这个球的范围内（已知的）光子数还能不断增加，每轮迭代后对于半径和通量有一个修正，按照文章给出的公式进行计算即可。

根据我实验过程中观察到的现象，PPM 算法当每轮光子数目较多时表现较好，因此我认为在内存允许的情况下，若想得到更高质量的图片，应当选择调大每轮发射的光子数而非总的迭代次数。

主要代码参见 `scene/rederer/ppm.rs`，ppm 使用的 KD 树位于 `utils/kdtree/mod.rs`（注意和后面求交加速使用的不同）。

抗锯齿

光线跟踪过程中，在像每个像素发射光线时，不像原来那样只是法向像素的中心，而是将每个像素更细致地划分，比如划分为 2 × 2 或 3 × 3 的子网格，不过我在实现中是在像素内随机若干点，让光线通过这些点发射。

代码参见 `scene/camera.rs` 中 `Camera` 的 `gen` 方法（用于在同一像素内产生若干条随机的射线，相机类的 `anti_alias` 参数指定了产生多少条），以及 PT、PPM 两种算法引用了它的地方。

纹理贴图

代码参见 `graphics/mod.rs` 中提供的 `TextureMap` trait，以及所有对它的实现。

平面

为平面选取一个原点，可设其为 $d\vec{n}$，然后在平面上建立一个直角坐标系，求出点在平面上的坐标，取整后取它们分别除以图片的长宽所得的余数作为映射坐标。这样的映射效果中图片会循环出现。

球

考虑球面上位置，用球坐标表示为 (r, θ, φ)，纹理映射时 r （因为球面上 r 值处处相等）可忽略，于是可按照 θ, φ 占各自合法区间的比例进行映射。为了使效果好看，实现时对半球进行映射。

长方形

类似平面，求出平面上的坐标即可，然后可以按照占长方形长宽的比例进行映射。

旋转贝塞尔曲线

对参数曲面求解后得到参数坐标，(t, θ)，t 为贝塞尔曲线的参数，θ 为旋转角度。然后按照各自占合法区间的比例进行映射。

景深

景深的实现方法是对透镜进行模拟。设当前考虑的像素在空间中的位置为 $\vec{p}$，透镜中心为 $\vec{O}$，焦距为 f，首先计算出聚点 $\vec{F}$ 的位置，即从透镜中心出发，沿从像素点来的方向前进 f 的距离：

$$\vec{F} = \vec{O} + f(\vec{O} - \vec{p})$$

然后在透镜上根据光圈大小参数 aperture 的值随机一个位置，作为此条光线的出发点，由于一定会通过聚点，因此也能计算出这条光线的方向。因为时间有限没能加入最终的图片中。

参见 `scene/camera.rs` 中 `Camera` 的 `gen` 方法中的处理。

软阴影

对于 PT 算法，由于其进行超采样，加上中途漫反射时的随机性，确实也能达到软阴影的效果；

对于 PPM 的算法，光子映射类的算法在从光源发射光子时便已经完成了光源采样的工作，自动支持软阴影。

复杂网络模型

读取

网络的读取参考了 PA1 的代码，并使用了 `wavefront_obj` 库。与 PA1 不同的是，不再对每个面存储一个法向量，而是对每个点存储一个法向量用于法向插值。

代码参见 `graphics/shape/mesh.rs` 的 `deserialize` 方法。

求交加速

包围盒

可以在一个点集上构建一个长方体包围盒，只需要对所有点的每个维度都求一个最大最小值即可。与包围盒求交的做法是，对每一维都求出直线的点落在当前维的区间中的 t 的范围，然后看三维的结果的交是否为空。另外，这个交集还可用于接下来的 kd 树求交加速。

另外注意到到三角形求交时其实会进行不少算数运算，在与三角形求教时，如果提前用计算量较小的包围盒判断一下，也能起到很好的优化作用。经过测试，确实能够缩短运行时间。

代码参见 `graphics/bounding.rs` 中 `Bounding` 类的 `build`、`intersect` 方法。

kd 树

既然多面体可以求包围盒，那么单个三角形也可求包围盒，可以尝试用 kd 树通过维护这些包围盒来维护所有的三角面片。

建树

对于一个三角面片的集合，仍然是通过某种方法首先选取一个划分维度，找到中间点。这里与一般的以点作为关键字的 kd 树有所不同，即有的三角形可能跨越中位数，这时需要将这些三角形保存在当前节点。将当前维度坐标完全小于中位数的三角形划入左子树并建树，右子树同理。若左子树、当前点、右子树中有两份的三角形的集合都为空，那么就没有必要划分了，直接将所有三角形保存在当前节点。

为了支持查询操作，需要对每个节点构建一个其子树中所有点的包围盒。

查询

查询需要返回所有三角面片与给定射线的交点中 t 参数最小的那一个。查询过程描述如下：

- 进入一个点时，首先让射线和当前点维护的三角形全部求交并尝试更新答案。这里需要指出的是，我们总是可以维护一个当前的得到的 t 的最后值，后面的过程中可以利用这个值进行剪枝。
- 接下来，尝试进入子树搜索，不妨假设子树都存在（这里只讨论这种最复杂的情况，其它情况类似且更简单）。
    - 首先让射线与两个子树的包围盒求交，可以得到对应子树可能求到的 t 值的区间。这里需要注意到一个比较重要事实：由于两个子树包围盒一定是不相交的，因此它们与射线相交的 t 值的区间也一定是不相交的。
    - 因此，一个启发式的想法，我们应该先进入这个区间更靠左的子树进行搜寻，如果找到了答案就不必再进入另一棵子树了
    - 如果没找到，在进入另外一棵子树前，若当前 t 的最优值位于区间的左侧，则也不必进入该子树寻找了

所有为加速三角面片求交使用的 kd 树代码位于 `utils/kdtree/triangle.rs`。

法向插值

在与三角形求交时，可以得到重心坐标 α, β, γ，利用 obj 文件中给出的三角形每个点对应的法向量与中心坐标加权求和，得到的结果作为当前交点处的法向量。

代码参见 `graphics/shape/triangle.rs` 第 43 行。

旋转贝塞尔曲线

项目中的参数曲面为二维平面上的贝塞尔曲线绕 $x = 0$ 得到，参数曲面可以表示为：

$$f(t, \theta) = \begin{pmatrix} \cos\theta & 0 & -\sin\theta \\ 0 & 1 & 0 \\ \sin\theta & 0 & \cos\theta \end{pmatrix} \begin{pmatrix} x(t) \\ y(t) \\ 0 \end{pmatrix} = \begin{pmatrix} x(t)\cos\theta \\ y(t) \\ x(t)\sin\theta \end{pmatrix}$$

贝塞尔函数求值

为了加速运算，考虑到使用到的控制节点数目较少，没有使用课上介绍的 de Casteljau 算法，而是直接求出了贝塞尔函数对应的多项式的各项系数，这样函数求值的时间就能降到最低。

求交

假设没有其它变换，要求解的方程其实为：

$$\vec{o} + k\vec{d} = \begin{pmatrix} x(t)\cos\theta \\ y(t) \\ x(t)\sin\theta \end{pmatrix}$$

首先注意到 y 这一行不含 θ，有

$$o_y + kd_y = y(t)$$

然后 x, z 两行平方相加后可以消去 θ:

$$(o_x + kd_x)^2 + (o_z + kd_z)^2 = x^2(t)$$

上面两式联立消去 k，得到关于 t 的非线性方程，使用数值分析课程上介绍的阻尼牛顿迭代解决。和一般的牛顿迭代不同在于，求解方程 $f(t) = 0$ 时，每轮迭代都保证 $f(t_n)$ 的绝对值在减小。还需要解决多解问题，办法是均匀地选出若干初始值进行迭代，取最优解，即 k 最小的解。

所有旋转贝塞尔函数相关的代码参见 `graphics/shape/bezier.rs`。

其它

光线跟踪中折射率的维护

对于简单的光线跟踪的实现，可以假定环境中只有一种材质，那么便算上空气的折射率便只有两种折射率。在交界处简单地判断当前光线是在物体内侧还是外侧即可获取接下来的折射率。但这样显然无法适应多种折射率。对此，我的想法是，首先不妨假设所有透明物体都不相交，即只会存在嵌套关系，那么可以在光线追踪的过程中使用一个栈来维护当前的折射率：进入介质时压栈，离开介质时弹栈，任意时刻栈顶就是当前介质的折射率。这样的设计能够渲染出诸如水中有一个透明玻璃球的图像。但无奈时间精力不足没能构造场景生成相关图片。

这个做法还存在的问题有：

- 只适用于透明物体表面闭合且互相嵌套的情况，如果只是相交而不嵌套便无法解决（不过这种情况好像不是很正常）；
- 光源或相机处初始折射率选取。这个问题仿佛等价于给定空间中任意一点求该处介质的折射率。大多数情况下可以设置为环境的折射率，但还要考虑到光源本来就处于某种介质中的情况。这时，我认为基于透明物体表面闭合且互相嵌套的假设，只需要找到第一个包含光源的物体，可以从光源处随便引一条射线，找到第一个交于内表面的物体。

可参见所有光线跟踪的相关代码中，递归里传入的 `n_stack` 参数。

多线程

使用了 rayon 多线程库，CPU 核多时提速显著。

结语

本次大实验学到了很多知识，跑出好看的图片也有很大的成就感，要特别感谢老师和助教一学期的辛苦付出。

如果可以，我希望可以提一点对这项大实验小小的建议：在完成作业的过程中，在查资料上花了不少时间。如果这是作业设置的目的，那便无可说者；但事实上，确实同学们会花费很多时间去网上甄别大量的资料，甚至有时会搜到错误的介绍。如果可以，希望以后老师和助教能提供一些可供同学们在完成大实验时学习的相关算法的参考资料，这样我想同学们能集中精力在完成作业上，也会有更多的同学尝试更多地用自己的理解去实现代码而非过多地参考往年代码。

最后再次感谢老师和助教用一学期的时间带我们走进了计算机图形学丰富多彩的世界。