Parallel KD-tree Construction

for Ray Tracing

项目报告

杨子岳 1300012741

问题介绍

光线追踪是近年来被广泛应用的渲染技术。光线追踪通过模拟光线在场景中的反射、折射行为，能够为渲染场景提供照片等级的真实感。

光线追踪的计算代价主要来自于光线和场景中所有物体的求交计算：假设一个场景中有 N 个物体，共发射 M 条光线，最大追踪深度为 D，则光线追踪的总代价是 O(NMD)。一般而言，场景中的物体由物体元（Primitives）表示，典型的物体元是一个三角形，一个物体由许多三角形拼接而成。由于光线和物体的求交实质上是和物体元求交，即使对于很小的场景，N 也会非常大，使得光线追踪的代价很高。

一种降低 N 因子代价的方法是利用空间划分加速，例如对整个场景的空间建立 K-D 树。当光线在与场景求交时，可以利用 K-D 树对空间的划分对空间进行剪枝，最后只需要做光线和一小部分空间中的物体元的求交运算。这样，光线追踪的代价下降至 O(MD * logN)。

构建 K-D 树的代价经常是不可忽略的。在很多要求实时性较好的光线追踪的运用场景，K-D 树的构建代价可能成为瓶颈。例如，对于同一个场景，可以并行地发射多根光线进行与场景的求交计算，但是在之前需要做的 K-D 树构建过程却可能不是并行化的，不能充分利用计算资源。

本项目参考论文 *Highly Parallel Fast KD-tree Construction for Interactive Ray Tracing of Dynamic Scenes* 中的说明，基于 pbrt-v2 光线追踪框架实现了并行化的 K-D 树构建。测试结果表明，K-D 树构建的并行化能够显著提升其构建速度。

算法说明

串行化的 K-D 树构建算法

给定一个场景及场景中的物体元，首先需要对所有物体元计算其包围盒，在之后的空间划分中使用包围盒代表物体所占的空间位置。

之后可以开始构建 K-D 树了，串行的 K-D 树构建算法可以划分为下面几个过程：

1. 对每个划分轴（光线追踪在三维下进行，即 X, Y 和 Z 轴），通过某种方法找到最佳的划分点；

2. 将场景中的物体根据划分点划分到两个子空间内，其中与划分点有重合的物体同时出现在两个子空间内。将两个子空间作为两个子场景；
3. 递归地将子场景传递给 K-D 树构建算法

并行化的 K-D 树构建算法

串行化的 K-D 树构建算法可以通过如下两方面进行并行化：

1. 对所有物体元计算包围盒的工作是各自无关的，可以完全并行化；
2. K-D 树构建的并行化

对于整个场景的 K-D 树的任意一个非子节点，以其为根的子树代表场景空间的一部分。由于 K-D 树的划分是基于空间划分的，它满足同层子树代表的子空间不相交的特性。可以利用这个特性，将对子空间的 K-D 树构建过程并行化。

一个简单的思路（Figure4）是设置一个串行深度 D，当串行化算法来到深度为 D 的节点时，将该子空间的构建任务作为一个 Task 发射给线程池。在串行化算法结束后，等待所有线程结束后，将所有子树与串行化算法生成的一部分 K-D 树进行合并。这个合并过程一般需要遍历所有的 K-D 树节点一次，是主要的并行化代价。

但是，这个简单的思路不能实现很好的负载均衡：对于某些物体元分布不是很均匀的场景，K-D 树的不同子树的深度可能很不一致。为了克服这个问题，将并行化条件改为，当场景物体元数量下降到一定程度时，将其子树的构建任务作为一个 Task 发射给线程池。（Figure5）

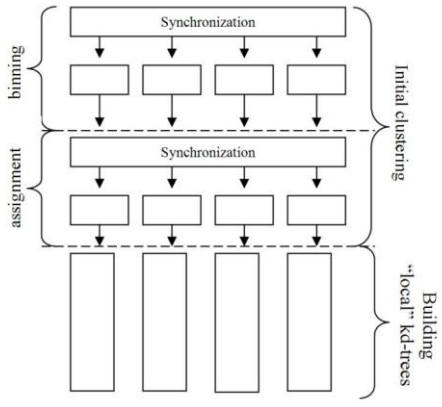


Figure 4: *Hybrid parallelization scheme which does parallel initial decomposition (clustering) of data to create jobs for independent processing.*

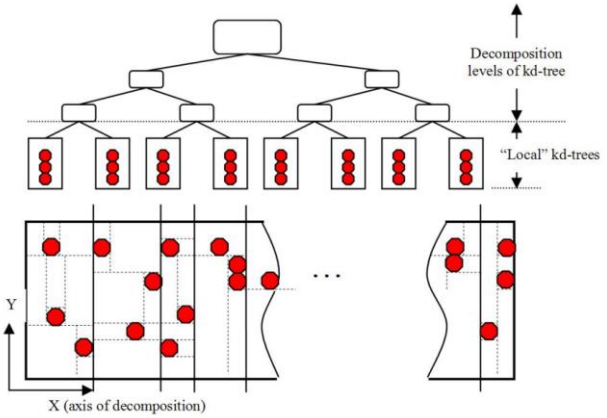


Figure 5: *Balanced decomposition by initial clustering*

代码实现

代码实现选用了 pbrt-v2 框架。这是一个配套 *Physically Base Rendering: From Theory to Implementation* 一书的光线追踪框架，其中 src/accelerator/kdtreeaccel*中实现了一个串行化的 K-D 树构建算法。在 http://www.pbrt.org 有一系列实验场景可供下载。之后的实现和测试全部基于该框架进行。

代码也开源在 https://github.com/yzygitzh/pbrt_v2_parallel_kdtree_construction，以对 pbrt-v2 进行 fork 的形式。修改历史可查看 git log 中 yzygitzh(yzylivezh@hotmail.com)的提交。

框架结构

这个框架读取 pbrt 格式的场景描述文件，进行语法解析后生成对应的参数、场景和物体。在 pbrt 文件开头（WorldBegin 之前）使用语句

```
Accelerator "kdtree"
```

可声明这个场景将使用 K-D 树进行渲染加速。实验中除了默认是 K-D 树加速的场景外，某些场景是采用这种方式开启 K-D 树渲染加速的。

src/core/api.cpp 中的 MakeAccelerator 函数是生成加速数据结构（包括 K-D 树）的入口。如果在对 pbrt 文件的解析中发现了使用 K-D 树加速的声明，这个函数将调用 src/accelerator/kdtreeaccel.cpp 中的 CreateKdTreeAccelerator 函数。该函数对传入的场景生成一个 K-D 树数据结构返回。

并行化的主要工作在 kdtreeaccel.cpp 及其头文件 kdtreeaccel.h 中进行。

框架代码特性

K-D 树相关代码

框架中的 K-D 树表示比较特别，主要体现在以下几个方面：

1. K-D 树表示：框架中含有 BVH、K-D 树等种类的加速数据结构。这些结构全部使用 C++的类进行表示，构造过程在相应的构造函数中完成。K-D 树的类是 KdTreeAccel。
2. K-D 树节点存储：在该框架中，K-D 树的节点存放在一个数组内。为了节省空间，

对于任意一个 K-D 树节点，其左儿子一定紧邻该节点后存放，节点内保存右儿子的数组内索引。这给 K-D 树合并带来了一定的困难。

3. 真正执行计算 K-D 树的是 kdtreeaccel.cpp 中的 buildTree 函数，这个过程在 KdTreeAccel 的构造函数中被调用。

并行化相关代码

该框架是跨平台的，代码中针对不同平台（Win32API 和 POSIX）打包了一个基本的并行接口。这个并行接口使用线程池机制，可以创建自定义线程池，向线程池中添加任务（继承 Task 类并实现了 Run 函数的类），通过 EnqueueTasks 和 WaitAllTasksDone 调用进行并发调度。其中也包装了基本的锁机制，不过在项目中没有用上。

实现思路和步骤

对应并行算法描述中的内容，代码并行化主要分为两部分：

物体元展开、包围盒计算的并行化

这部分可以直接并行化。对物体元数组（primitives）进行拆分，将计算代码实现在 KdTreeComputeBoundTask 和 KdTreePrimitiveRefineTask 两个类中，使用线程池并发调度即可。需要进行数组在线程之间的分发和回收。

K-D 树构建的并行化

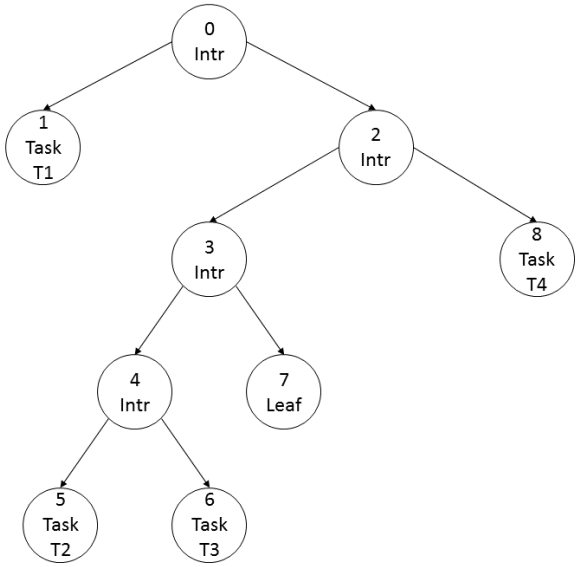
通过创建新 KdTreeAccel 的方式进行子树计算而不是开启新的 buildTree 实例，因为后者会产生太多的状态变量。

在初始 KdTreeAccel 调用的 buildTree 中按照并行化算法进行任务构建和发射，需要保存一些状态变量。在该 buildTree 结束后，初始 KdTreeAccel 将等待所有子树（子 KdTreeAccel 实例）构建完毕。

子树构建完毕后，初始 KdTreeAccel 对每个子 KdTreeAccel 收集其子树信息，完成 K-D 树合并工作，并返回。需要遍历 K-D 树所有节点。

K-D 树合并

由于框架中 K-D 树存储的特性，合并 K-D 树需要进行一些特殊的计算。下面结合例子进行说明。



如图是一棵在 KdTreeAccel 的 buildTree 完成后的 K-D 树，称为原初 K-D 树。每个节点最上方的数字代表其在节点数组中的存放索引。一共有三种节点，Intr（初始 KdTreeAccel 计算好的非叶子节点），Leaf（初始 KdTreeAccel 计算好的叶子节点），Task（交给线程池计算的待填充子树节点）。K-D 树的合并即 Task 节点的拼接。图中共有四个 Task 节点，这些节点分别对应线程池中的一个计算好的子树，子树们将被拼接到对应节点上，生成最终的 K-D 树，称为完全 K-D 树。Tn 代表某一棵子树的节点个数。

合并是一个串行化扫描的过程。首先统计原初 K-D 树的节点数和线程池中各子树的节点数，得到合并后 K-D 树的节点数并开辟存储空间。为这段空间初始化一个下标 x，扫描整个原初 K-D 树，对 Intr 和 Leaf 节点直接复制到 x 指向的新空间并令 x 自增 1，对 Task 节点需要将其所有节点复制到 x 指向的新空间，并令 x 自增子树节点数。根据 K-D 树的存储规律，这个步骤结束之后，新空间中各节点的位置均与完全 K-D 树完全一致（子树是自持的），还需要处理每个节点的右儿子索引。

Leaf 没有右儿子，不需要计算；Task 的右儿子索引需要整体增加一个数（子树内部的索引是自持的），这个数是复制该子树前的 x 值；Intr 除了需要考虑 x 值外，还需要考虑其左子树扩充后的影响。因此，在进行合并之前，需要对原初 K-D 树进行一次遍历，以计算所有 Intr 节点的左子树的实际节点个数。这也是并行化代价的一部分，但是考虑到原初 K-D 树一般规模有限（深入不了几层就发射线程了），且遍历代价是线性的，是可以接受的。

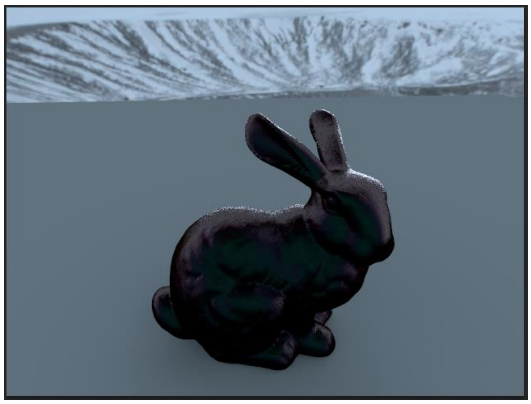
性能测试和分析

运行环境：

代码可以在 Windows 和 Linux 上运行（已经测试）。考虑到开发的便捷性，开发和测试选择了 Windows 平台。具体环境为 i7 5700HQ (4 cores 8 threads)，16GB RAM，VS2013。

正确性评估：

在调试过程中，将并行化算法构建的 K-D 树与原本串行化算法构建的 K-D 树通过打印输出进行了逐节点比对，完全一致。使用 K-D 树构建场景的测试也完全一致。下面是一些通过并行化算法生成的 K-D 树渲染出的实例场景：



Bunny

room-igi

teapot-area-light

buddha

性能测试：

主要评估了对 K-D 树构建耗时的优化效果，使用 VS 的 Release 编译优化。具体数据如下：

	teapot-area-light	room-igi	bunny	buddha
Serial(ms)	18	56	503	7716
Parallel 1 thread (ms)	20	60	550	8163
Efficiency 1 thread	0.9	0.933333333	0.914545455	0.94524072
Speed up 1 thread	0.9	0.933333333	0.914545455	0.94524072
Parallel 2 threads(ms)	13	36	380	5825
Efficiency 2 threads	0.692307692	0.777777778	0.661842105	0.662317597
Speed up 2 threads	1.384615385	1.555555556	1.323684211	1.324635193
Parallel 4 threads(ms)	9	28	313	4773
Efficiency 4 threads	0.5	0.5	0.401757188	0.404148334
Speed up 4 threads	2	2	1.607028754	1.616593338
Parallel 8 threads(ms)	9	24	273	4478
Efficiency 8 threads	0.25	0.291666667	0.230311355	0.215386333
Speed up 4 threads	2	2.333333333	1.842490842	1.723090665

性能测试结果评估：

1. 从并行化后的算法在单核上的运行时间来看，效率还不错，证明并行化代价较低；
2. 2 核心取得的加速比增长最为明显；8 核心最多能在 room-igi 样例上取得 2.3 左右的加速比。
3. 8 核心取得的加速比增长已经开始变得优先。如果不考虑 CPU 实际只有 4 个物理核，从表格中可以看出，串行化算法运行时间的一半左右是该并行算法当前实现的瓶颈，距离参考的论文中实验结果还有一定差距。推测存在 K-D 树存储方式的不同产生的影响。

不足和改进方向

1. 内存管理做得比较粗略，可能存在内存泄漏问题。
2. 论文中提到了一些对内存访问的优化，通过优化访问模式来降低总线拥塞的影响。在本项目中没有实现。
3. 受框架的 C++组织结构限制，类的包装层数可能较多，访问模式复杂，造成潜在的冗余耗时。

参考文献

[1] Shevtsov, Maxim, Alexei Soupikov, and Alexander Kapustin. "Highly Parallel Fast KD‐tree Construction for Interactive Ray Tracing of Dynamic Scenes."*Computer Graphics Forum*. Vol. 26. No. 3. Blackwell Publishing Ltd, 2007.

[2] Pharr, Matt, and Greg Humphreys. *Physically based rendering: From theory to implementation*. Morgan Kaufmann, 2004.

[3] https://github.com/mmp/pbrt-v2