光子映射中的 KD Tree

Dezeming Family

2023 年 3 月 19 日

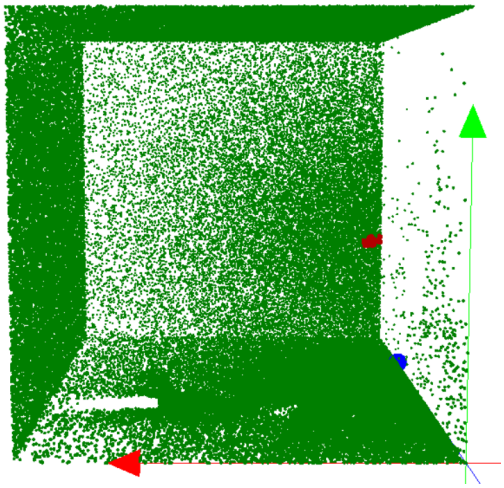
目录

一　基本介绍

最近的项目中有一些构建空间加速结构的需求，大概会基于现有的 KD Tree 结构来改进，因此打算再重拾一下基本的 KD Tree 构建方案，以及并行加速构建 KD Tree 的技术。本文描述的是论文 [1] 中的 KD-Tree 构建，我们会提供代码和光子图的可视化结果。

KD Tree（K 维二进搜索树）是一种三维数据结构，用于将三维空间中的光子点在空间位置上进行划分。除了叶节点，KD Tree 中的每一个节点都会包含一个三维空间中的点（在光子映射中就是一个光子）以及一个垂直于轴的分隔面，将该节点下的其他点分隔到两个子空间中。如果构建的树是平衡的，那么搜索时间复杂度会降低到 $O(\log n)$（n 表示光子数）。

本文不介绍光子映射技术,但是产生光子的代码来自于《零基础实现一个最简单的光子映射器》,我们将产生的光子的三维坐标放入到 PhotonPos.txt 文件中,本文使用时直接读取（通过 PhotonMap::readPhotonFromTxtFile 函数读取）然后构建光子树即可。本文的随书源码见我们的图形学零散代码的合集目录 [2]。使用 CMake 可以直接编译为 Visual Studio 工程，注意要编译成 x64 位的。运行以后会出现一个 OpenGL 窗口：



该窗口内的深绿色点表示产生的光子，深红色表示随机选定的某个点周边最近的 N 个点。PhotonMap::readPhotonFromTxtFile 读取我们提前生成好的光子点，然后构建光子 KdTree。每当鼠标点击一下窗口，则会重新选择一个随机点，然后搜索其最邻近的 N 个光子，并显示为深红色。

```
1 // 随机选择一个点，然后搜寻其周边点
2 searchIndex = getRandom() * m_PhotonMap.PhotonNum;
3 m_PhotonMap.getNearestPhotons(m_Nearestphotons, m_PhotonMap.mPhoton[
    searchIndex].Pos, 0.6, 100);
```

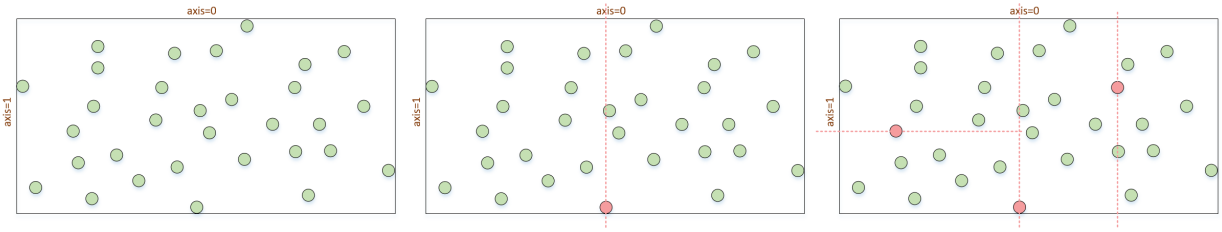
二　光子的表示与构建平衡 KdTree

[1] 中的光子结构为了减少存储占用会使用 char 来存储光照和角度，由于现代硬件容量较大，不必再过于在意存储，我们就用比较简单的存储方法，都用 float 类型：

```
1 struct Photon {
2         vec3 Pos; //位置
3         vec3 Dir; //入射方向
4         vec3 power; //能量，通常用颜色值表示
5         int axis; //划分轴
6 };
```

axis 表示划分轴，对于平衡 KdTree 的划分规则如下，我们以二维为例。第一步先构建全部光子位置的包围盒，对长度最长的轴，也就是这里的轴 0 来划分，把所有点根据轴 0 坐标值排序，找到中间点作为

划分位置，将全部区域划分为两个子区域。然后再对两个子区域分别再进行上述操作，直到划分到子区域中只有较少的光子数量时停止（比如只有 2 个光子时）。



这样构建出的 KdTree 是平衡的，构建平衡 KdTree 的算法复杂度是 $O(n \log n)$（n 是光子数，我们会再详细描述其复杂度）。伪代码描述为：

```
kdtree *balance( points ) {
  Find the cube surrounding the points
  Select dimension dim in which the cube is largest
  Find median of the points in dim
  s1 = all points below median
  s2 = all points above median
  node = median
  node.left = balance( s1 )
  node.right = balance( s2 )
  return node
}
```

2 1　构建平衡 KdTree

PhotonMap::balance() 函数中先按照原来的光子顺序生成一个备份 tempPhoton，在调用完 balanceSegment() 函数以后会删除该备份。

我们存储的光子在 mPhoton 数组中的索引是 1 到 PhotonNum，而不是 0 到 PhotonNum-1，这点需要明确注意。经过 PhotonMap::balance() 函数以后，mPhoton 数组便是一个平衡 KdTree 的铺平表示。

PhotonMap::balanceSegment() 函数的过程分为如下几步：

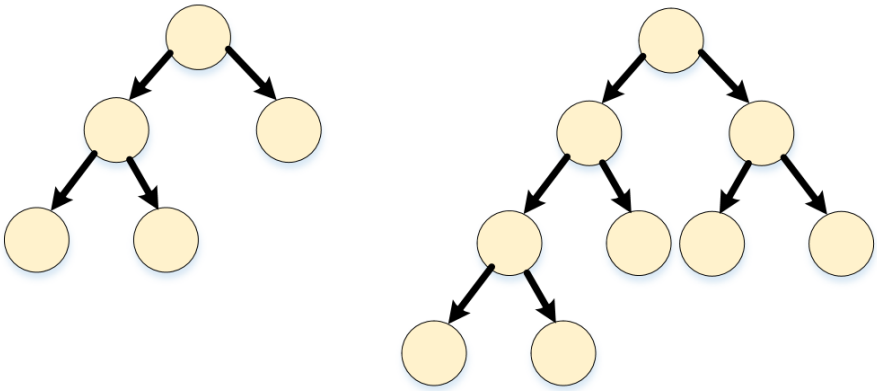
```
1 计算坐标中位数（暂且理解为中位数）
2 计算包围盒最大边界
3 根据坐标中位数将tempPhoton数组划分为两个区间
4 储存划分轴的光子
5 构建左子树
6 构建右子树
```

单独讲一些关于左子树的构建。box_max 会根据场景的划分而在子区域重新计算。因此，子区域的包围盒范围会先更新，然后再调用 balanceSegment 后再恢复，这个过程是递归的：

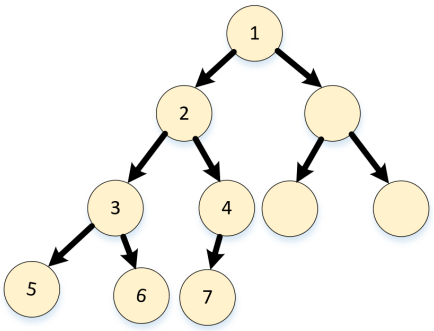
```
1 // 构建左子树
2 if (start < med) {
3     double tmp = box_max[axis];
4     box_max[axis] = mPhoton[index].Pos[axis];
5     balanceSegment(tempPhoton, index * 2, start, med - 1);
6     box_max[axis] = tmp;
7 }
```

2 2　寻找坐标中位数以及根据中位数划分

为了使得构建的 KdTree 是平衡的，要尽量保证得到的二叉树是一个完全二叉树（虽然不完全的二叉树可能也是平衡的，但在构建时，无法做到保证构建出的不完全二叉树一定是平衡的），即比如下图：

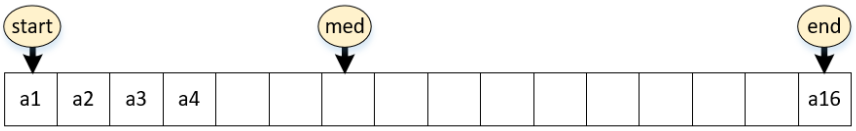


如果一个节点下总共 4 个光子，那么左子树应该总共有 3 个光子，右子树应该有 1 个；如果一个节点下总共 8 个光子，那么左子树应该总共有 5 个光子，右边应该有 3 个。计算左边光子的坐标索引的函数就是 calMed，调用 calMed(1,10) 得到 7（7 就是中位数坐标，下图中的序号只是为了计数，并不是索引号）：

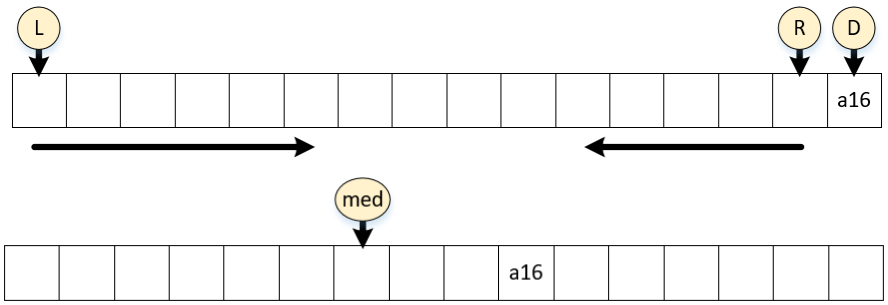


现在问题是，如何把全部的数据根据所设定的划分轴来将轴小的数据划分到左边，将轴大的数据划分到右边，这个过程就是 MedianSplit() 函数。这个函数的过程实在不是很容易描述清楚，我只能尽量多写一点文字来叙述。顺便提一句，我们也没有必要一定要构建完全二叉树，只是构建完全二叉树可以很容易地用“表”数据结构来存储。而且完全二叉树一定是平衡的二叉树。

我们的任务目标是找出数组中的如果从小到大排序后位列第 m 的数（在下图中，$m = [0, 15]$），索引 med=start+m，然后将数组的 start 到 med 索引下的元素保证都不大于 med；med 到 end 的元素都不小于 med。假设一开始数组是这样的：



我们的步骤如下。设表里最后一个数据是 a16，我们从两侧扫描一遍以后，保证得到当 a16 是里面从小到大第 k 个数据时（k=[0,15]），其坐标是 start+k，而且 a16 左边的数据都要小于 a16，右边数据都要大于 a16：

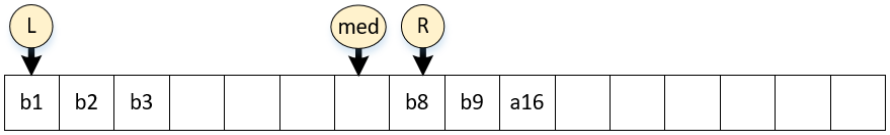


代码实现是：

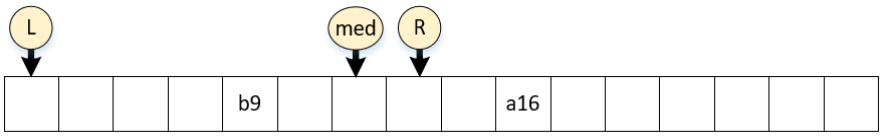
```
1 double key = tempPhoton[r].Pos[axis];
```

```
int i = l - 1, j = r;
for (; ; ) {
    while (tempPhoton[++i].Pos[axis] < key);
    while (tempPhoton[--j].Pos[axis] > key && j > l);
    if (i >= j) break;
    std::swap(tempPhoton[i], tempPhoton[j]);
}
std::swap(tempPhoton[i], tempPhoton[r]);
```

之后，如果 a16 的坐标不是 med，比如下图中 a16 的坐标大于 med，但我们知道 a16 左边的数据都要小于 a16，因此只需要再在下面的子区间循环进行上述过程即可（由于数据进行了重新排序，我们用 b 来表示序号）：



循环一次以后，使得 b9 左边的数据都小于 b9，b9 右边的数据都大于 b9：



如果 b9 的坐标不是 med，则还需要继续循环前面的过程。

2 3 算法复杂度

我们先不直接分析上述代码的复杂度，而是思考一下这个问题的最优复杂度应该是多少。

我们假设划分轴并不是每次都要根据最大长度的轴来选择，而是提前设定好一个固定的划分轴。由于构建的平衡 KdTree 左子树的值都小于节点，右子树的值都大于节点，而且对于子树也是一样，所以该过程一定不低于排序的计算时间复杂度，即 $O(n \log n)$。

而且假设我们只按一个轴进行划分，那么其实完全可以先给全部数据按照该轴的坐标进行排序，然后将排好序的数据直接以复杂度 N 来遍历生成平滑 KdTree，所以最优复杂度也不会高于排序的复杂度。因此，其算法复杂度为 $O(n \log n)$。

在我们的构建过程中，假设每次的中位数都恰好是一半，那么相当于递归的过程每次都是对半分，因此一共要执行 $o(\log n)$ 量级的 MedianSplit 计算。而在每个 MedianSplit 中，比如一开始将 n 个光子点进行 MedianSplit，它的计算复杂度假设是 $o(n)$，而划分为两个子树，这两个子树分别做 MedianSplit 的总共的时间复杂度应该也是 $o(n)$。也就是说，无论递归到多少级，该级所在的全部子树的总的 MedianSplit 的时间复杂度都是 $o(n)$，所以构建平衡 KdTree 的全部时间复杂度就是 $o(n \log n)$。但可惜，我们的 MedianSplit 复杂度并不能一定保证是 $o(n)$，在一些最坏的情况下，比如一开始在 start 到 end 之间的数据是从小到大排列的，那么此时的算法复杂度甚至会达到 $o(n^2)$。

有什么解决思路吗？其实很简单，对于同样的 MedianSplit 的任务目标，我们可以先以时间复杂度 $o(N)$（这里的 N 是 start 到 end 的长度）找到数组段中第 m 大的数（即索引 med=start+m）。然后把该数所在位置与 end 索引下的数位置互换，然后再执行同 MedianSplit 相同的步骤，这样，MedianSplit 中的 while 只需要循环一轮就能完成任务，总时间复杂度是 $o(N)$，而不再是可能存在的 $o(n^2)$ 了。我们并没有在程序 [1] 中实现寻找数据中第 m 大的数并换位的这个过程，但是在《寻找数组中第 k 小的数/寻找中位数》中有详细的介绍，读者可以自行补充当做练习。

三　搜索最近的 N 个光子

搜索最近的 N 个光子的过程也是范围搜索的过程，即在有限的半径内搜索，所以搜索到的光子数可能会小于 N。在判断时我们使用平方距离即可，而不必求真实距离。

在搜索过程中，将在搜索范围内搜索到的光子排序，然后当搜索到 N 个光子以后，当找到的新光子离着搜索点的距离比这 N 个光子中离着搜索点最远的距离更近时，就替换掉离着最远的光子。用最大堆的方式可以很容易地及时删除最远的光子，而不必每次都进行排序。

调用 getNearestPhotons 的参数是：

```
getNearestPhotons(&np, 1);
```

大致步骤如下：

```
void PhotonMap::getNearestPhotons(Nearestphotons* np, int index) {
    //超出索引范围，就返回
    if (index > PhotonNum) return;
    Photon *photon = &mPhoton[index];
    // 子树存在，就向下搜索子树
    if (index * 2 <= PhotonNum) {
        double dist = np->Pos[photon->axis] - photon->Pos[photon->axis];
        // 搜索点在划分面左侧
        if (dist < 0) {
            // 搜索左子树
            getNearestPhotons(np, index * 2);
            // 如果距离划分面小于搜索半径，就也搜索一下右子树
            if (dist * dist < np->dist2[0]) getNearestPhotons(np, index * 2
               + 1);
        }
        // 搜索点在划分面右侧
        else {
            // 搜索右子树
            getNearestPhotons(np, index * 2 + 1);
            // 如果距离划分面小于搜索半径，就也搜索一下左子树
            if (dist * dist < np->dist2[0]) getNearestPhotons(np, index * 2)
               ;
        }
    }
    // 计算搜索点距离当前节点的光子的距离
    float dist2 = (photon->Pos-np->Pos).squaredLength();
    if (dist2 > np->dist2[0]) return;
    // 如果距离小于搜索半径，就考虑是否将其添加到现有的堆中
    if (np->found < np->max_photons) {
        // 如果已经搜索到的光子量还不到最大搜索数
        ......
    }
    else {
        // 已经搜索到的光子量达到了最大搜索数
        ......
    }
```

```
35 }
```

np->dist2[0] 中存储的是搜索半径的平方，搜索的最近的光子数量达到了最大搜索数 max_photons 后该值会变为已经搜索到的最近的 N 个光子的距离搜索点的最远的距离。

当搜索到的光子距离搜索点的距离小于搜索半径，就考虑是否将其添加到堆中。如果当前堆中的光子量还不够，就直接添加：

```
1 np->found++;
2 np->dist2[np->found] = dist2;
3 np->photons[np->found] = photon;
```

否则，如果此时光子量已经达到了最大上限，但才刚刚达到，还没有建立最大堆，则先构建最大堆：

```
1 if (np->got_heap == false) {
2     // 构建最大堆
3     ......
4     np->got_heap = true;
5 }
```

如果已经构建好了最大堆，就在最大堆中操作。操作完以后，可以更新 np->dist2[0]，使得远于 np->dist2[0] 的光子不会再被搜索到。

```
1     np->dist2[0] = np->dist2[1];
```

我们先描述已经在构建好最大堆以后的操作：

```
1     int par = 1;
2     // 如果存在子树，就向下遍历
3     while ((par << 1) <= np->found) {
4         int j = par << 1;
5         // 如果右子树节点光子与搜索点之间的距离大于左子树节点光子与搜索点之
              间的距离，就去更新右子树
6         if (j + 1 <= np->found && np->dist2[j] < np->dist2[j + 1]) j++;
7         if (dist2 > np->dist2[j]) break;
8
9         np->photons[par] = np->photons[j];
10        np->dist2[par] = np->dist2[j];
11        // 赋新值给par
12        par = j;
13    }
14    np->photons[par] = photon;
15    np->dist2[par] = dist2;
```

如果比较熟悉数据结构，就能够看懂构建最大堆的过程，我们不再细讲。

```
1 for (int i = np->found >> 1; i >= 1; i--) {
2     int par = i;
3     Photon* tmp_photon = np->photons[i];
4     float tmp_dist2 = np->dist2[i];
5     while ((par << 1) <= np->found) {
6         int j = par << 1;
7         if (j + 1 <= np->found && np->dist2[j] < np->dist2[j + 1]) j++;
```

```
8           if (tmp_dist2 >= np->dist2[j]) break;
9
10          np->photons[par] = np->photons[j];
11          np->dist2[par] = np->dist2[j];
12          par = j;
13      }
14      np->photons[par] = tmp_photon;
15      np->dist2[par] = tmp_dist2;
16  }
```

参考文献

[1] Jensen, H. W. (2001). Realistic image synthesis using photon mapping (Vol. 364). Natick: Ak Peters.

[2] https://github.com/feimos32/Computer-Graphics-Code