数据结构 地理围栏

GEO-FENCING

实验报告

专业：计算机科学与技术

学号：17307130178

姓名：宁晨然

目录：

一、解决方案

A）存储数据

核心亮点：网格预处理、一一对应、分类封装

1.point 存储：输入点集合--vector<pair<double,double>> pointbase

设置 pointbase 存储点集合和多边形集合，根据输入顺序顺序存放。

2.polygon 存储：输入多边形集合--vector<GridSet*> G

不存储多边形集合，先对多边形进行网格化预处理，保存网格信息，设置 G 存储多边形网格集合，根据输入顺序存放。亮点在于**预处理**，后面会提到。

3.id 存储：输入点、多边形 id 集合--vector<int> ID, ID2

设置 ID, ID2 存储点集合对应的 id 和多边形对应的 id，根据输入顺序存放，与 pointbase 和 G **一一对应**。

4.disable 存储：删除点、多边形信息集合--vector<bool> dispoint, dispoly

设置 dispoint，一开始均设置为 false，当需要删除某个 id 时，dis[k] = true，dispoly 存储被删去的点和多边形，与 pointbase,G 存储一一对应。亮点在于**一一对应，更高效的索引**。

5.运作方式：vector::emplace_back( )

输入的数据流均使用 emplace_back 提高效率存储，是用 vector 封装库类型而不是自己定义类更加安全（也没必要定义类），点直接添加、多边形顺序存储后添加、id 直接添加、disable 直接添加。亮点在于**分类封装**。

B）预处理

1.polygon 网格化：GridSet(polygon, Grid_Resolution, &gridset)

对于每个输入的多边形，不直接是用多边形，先进行预处理。对于每个多边形都通过 O（n）的方式网格化，可以大大节省时间复杂度。不过这个的空间要求可能更大。几个需要阐明的地方：

①**网格化：**根据 Grid_Resolution 确定多边形划分的网格个数，对多边形每条边进行遍历，计算经过的网格格子，将多边形处理转换成可以判断点的网格格子信息。

②**网格判断：（亮点）**网格将多边形分解为三个板块，完全属于多边形内部的格子、多边形边经过的格子、多边形外部的格子。判断点时更加迅速。

③**格子个数：**Grid_Resolution 越大格子越多，判断一定程度能够加快，因为划分更加细致，但同时预处理时间更多，并且空间耗费更多。对于存储的多边形我是用 20，对于询问的多边形我是用 85。

2.网格判断法：快速排除大多数情况

对每个输入的多边形进行预处理，处理成网格信息，不存储多边形集合，直接存储网格；每个网格可以将点在多边形问题划分为三种情况，完全在多边形网格内部、完全在多边形网格外部、落在多边形边所经过网格，前两种情况可以通过 O(1)直接排除，最后一种情况通过 O（n）的算法计算出点是否在多边形内部。

3.预编译：开设-O2

优化的编译-O2 能够提高 main 主体实际运行速度，-O3 并没有更多的提升，-O2 即可。亮点在于**开设-O2**。

C)查找删除索引

1.给定点查找多边形：顺序遍历 G

给定点查找多边形时，直接遍历一遍多边形网格集合，对该点进行一一判断，由于网格

是多边形已经预处理过的信息，所以复杂度非常低（取决于数据）。

2.给定多边形查找点：顺序遍历 pointbase

给定多边形查找点时，更加方便，先对该多边形进行预处理网格化，然后遍历一遍点集合，一一判断，由于网格可以迅速排除多边形外和多边形内接网格内，所以非常迅速。

3.删除某个 id 的多边形或点：顺序遍历 disable

顺序遍历 ID 或 ID2 访问到所求 id 值，disable[k] = true。在判断前先判断是否已经删除，若已经删除则无需判断该点、多边形。

4.索引 id 方式：一一对应的 ID 列表 O(1)

问题要求输出点或多边形的 id，每次通过 O(N)的方法遍历 pointbase 或 G，对每个点或多边形直接进行判断，如果符合条件，根据 ID[i]可以直接索引得到此时的 id。每个 id 只需要 O(1)就可以访问到。

二、细节问题

A）复杂度

操作	时间复杂度	空间复杂度	特点
多边形网格化 GridSet	O(n)	取决于 GridResolution 值	一次预处理即可加快整体判断速度
点在多边形 GridTest	O(1) ~ O(n)（根据数据特点、非常接近 O(1)）		特别适合判断多个点在一个多边形内部，一次预处理永久收获
存储操作 emplace_back	比 push_back 快		封装库函数，比 push_back 要快
删除操作 remove	O(n) 顺序查找		顺序存储 id 的劣势就是查找必须顺序遍历

B）存储细节

Index	0	1	2	……	n-1
Int counter	1	2	3	……	n
Vector<int> ID	Id1	Id2	Id3	……	Idn
Vector<int> ID2	Id1	Id2	Id3	……	Idn
Vector<pdd> pointbase	Point1	Point2	Point3	……	Point n
Vector<vpdd> G (polygonbase)	PolyGrid1	PolyGrid2	PolyGrid3	……	PolyGrid n
Vector<bool> dispoint	false	false	false	……	false
Vector<bool> dispoly	false	false	false	……	false

C）代码结构

用 common.h 申明所有的全局变量，然后在 common.cpp 中定义全局变量，所有的 submit.h 均包含 common.h。Grid 相关包含在 common.h 中。

亮点：所有的 submit 只需要包含 common.h，实现方便，封装安全。

三、思考过程

1.总体理解

本 pj 主要包括以下几个深刻的问题值得思考：

如何在空间中存储多边形和点？

如何给定一个多边形、点，使其找到该范围内附近的多边形和点？

如何判断点在多边形内部？

如何索引 ID？

2.问题分析

核心问题：PointInPolygon in large database

A）存储：

首先要建立一个合理的存储模式，我思考后有三种存储方式：一，最直接的方法就是顺序存储（我采用的方法），不需要任何操作和类型构造，直接是用原类型可以 emplace_back 进 vector 中，用 vector 的封装性可以不用多余考虑添加效率和空间分配。二，构造 polygon 和 point 类，point 中封装一个 double[2]数组，polygon 类中封装 n 个 point，使用 R 树的话需要顺序遍历求出 polygon 对应的 max 点和 min 点，需要对 polygon 和 point 进行初始化构造处理，可能更加繁琐，但是封装性特别好，id 可以存在类中，也可以单独存放在 ID 中。三，存放在 R 树（或其他索引结构）中，根据 R 树（或其他索引结构）的存放方式，将 point 和 polygon 与 id 打包，直接存在索引中，这样 id 查找、删除、判断等操作都比较好实现，但是构造、传参等比较耗费时间和空间。

不同的存储方式都有不同的优点。三种方式我都尝试过，第一种是网格判断法使用的，后两种是配合 R 树索引结构使用的。第一种方式封装性较差，但是传参操作少、最简明易懂、索引极其方便。第二种方式封装性较好，便于理解，但是传参操作多、需要构造函数、索引方式繁琐。第三种方式封装性也好、使用 R 树内部的构造方式，同样也是传参、构造、索引方式繁琐，并且本身 R 树中的 Remove 操作也很繁琐。

一开始尝试的第三种（存在 R 树里），后来发现没必要，存放在了封装好的外部结构中，将索引和存储分离，后来发现还是繁琐，然后改用了 Grid 判断法后发现不需要索引结构了，就直接使用第一种方法简单明了的阐明了所有的结构。

B）判断：

我在做 pj 前从百度和 Google 查询了很多索引结构和判断方法，从射线法开始找，最后发现了一个比较全面的点在多边形(pointinpolygon)的方法总结【1】，里面总结了各种 pip 的策略：射线法、角度法、三角法、凸多边形法、Y 轴分层法、网格法等，并且做了详细的评测。我查看了其中的几种算法，结合统计数据，一开始使用了 CrossingMultiple 的射线法，配合 R 树索引结构做出了初稿。后来发现射线法对于每个 pip 都是 O(n)的判断复杂度，很难再优化。我单独测试了加判断和不加判断的时间，得出了结论：

判断算法在本 pj 中运行时间占很大部分，选择好的判断算法极其关键。

PIP 问题中三种策略，单独点与单独多边形、凸多边形、预处理多边形与单独点。第一种最快的方法是 Crossing(by MacMartin)，复杂度是 O(n)，对于临界情况也处理较好，但缺点是每个 PIP 问题都需要一定的时间。第二种方法中比较优秀的是一个 O(nlogn)的角度算

法，可惜由于 pj 样例不是凸多边形不能在 pj 中使用，我也思考过先把多边形预处理成 convex-polygon，复杂度很高、代价过大后放弃。第三种方法我使用了网格（应该比 bins 更快），使用 O(n)预处理多边形，转换为网格信息，在“多个点与一个多边形”的问题中表现极其出色，一次预处理永久收益。

各种算法时间比较如下【1】，最终我选择了网格判断法，并构造了对应的存储结构与询问方式。

General Algorithms, Regular Polygons:

	number of edges per polygon				
	3	4	10	100	1000
MacMartin	2.7	2.8	4.0	23.7	225
Crossings	2.8	3.1	5.3	42.3	444
Triangle Fan+edge sort	1.3	1.9	5.2	53.1	546
Triangle Fan	1.3	2.2	7.5	86.7	894
Barycentric	2.1	3.9	13.0	143.5	1482
Angle Summation	52.9	68.1	158.8	1489.3	15762
Grid (100x100)	1.5	1.5	1.5	1.5	1.5
Grid (20x20)	1.6	1.6	1.6	1.7	2.5
Bins (100)	2.1	2.2	2.6	4.6	3.8
Bins (20)	2.4	2.5	3.4	9.3	55.0

PIP Question Tests

C）索引：

由于数据量较大，需要一个合理的方式索引得到需要进行判断的点或多边形。在查阅了 KD 树、四分树、RangeTree、R 树等之后选择了 R 树进行尝试。R 树是对矩形进行分类，所以需要对多边形遍历一遍寻找外包矩形，然后存入 R 树中。并且可以用 R 树直接进行插入、索引、删除，对于数据的管理极其方便，所以在第一次尝试的时候使用了 R 树进行实验。

R 树模板的确很实用，对所有的多边形、点进行一边范围缩小，再进行判断 PIP 问题，所以会缩短很多时间，并且 R 树作为一个标准库，可以提高封装性，内存分配也不需要担心。但是 R 树在这个 PJ 中也有劣势，PJ 并不是只是对数据进行管理和划分、不是纯粹的存储问题，还有判断、预处理等过程，另一半的时间还需要分配在判断上，这一点上 R 树只做到了数据存储管理、没有办法（或许是我没想到好的方法）再更近一步优化。

地理围栏重要的不仅是数据存储与索引，还有 PIP 问题的判断。

我采取的是直接使用网格法，和顺序存储。索引只需要 O（1）。

D）删除与插入：

如果使用 R 树也有几种选择，可以选择直接使用库函数中的 Remove 函数删除数据、插入数据，也可以选择外部存储、外部“删除”。我选择了后者，即外部存储 ID 和多边形、点数据，然后保存删除信息 disable，实行“伪删除”操作。插入时，R 树存储“转录 ID”，即 counter，ID 容器中顺序存储真实 id，这样方便管理和索引，也不需要在 R 树中存放数据。

使用“伪删除”和“伪插入”的方法，可以绕过大批量数据的操作。

使用 counter 一一对应法有点类似指针的操作，disable 操作有点类似 visited 数组操作，均是课堂内容教会给我的思路。

E）最终方案

综上所述，设置一一对应的存储很关键，ID、disable、base 三类容器一一对应，可以实现很快的存储、插入、删除、索引操作。

Index	0	1	2	……	n-1
Int counter	1	2	3	……	n
Vector<int> ID	Id1	Id2	Id3	……	Idn
Vector<int> ID2	Id1	Id2	Id3	……	Idn
Vector<pdd> pointbase	Point1	Point2	Point3	……	Point n

Vector<vpdd> G (polygonbase)	PolyGrid1	PolyGrid2	PolyGrid3	……	PolyGrid n
Vector<bool> dispoint	false	false	false	……	false
Vector<bool> dispoly	false	false	false	……	false

四、实验过程

1.出发点：问题模型

A）Blender 开源代码

拿到 PJ 问题“地理围栏”着实很懵逼，用自己的语言描述了一下就是：需要找到一个高级的索引方式存储大批量数据，判断点在多边形的问题。看起来已经分成了两个问题，我花了很多时间着手第一个问题，合理数据结构管理。

我曾今上过一门“虚拟空间设计”的课程，用的是 Maya 三维建模软件、blender 制作模型渲染等，其中就接触了很多刚体碰撞的物理模拟过程。这让我想到了，三维模型的碰撞问题中，肯定有一种数据结构判断两个刚体的位置，是否发生碰撞；也可能有一种数据结构判断刚体的一个棱角是否落在另一个刚体的其中一个面中，或许正是点在多边形的空间变体。心里一阵窃喜，就在网上乱搜一通。

我发现了 Blender 的软件代码是完全开源的，并且使用 C++写的各种数据结构。我立刻下载了 blender2.79b 的开源代码开始查看其中的结构，发现了刚体碰撞的确有两种刚体碰撞判断问题分别为 broadphasecollision 和 narrowphasecollision，讲述了刚体在空间中先经过预处理（broadphasecollision），判断空间最有可能发生碰撞的两个刚体，有点类似平面中最有可能发生点在多边形的预处理一样；再利用 narrowphasecollision 判断刚体碰撞的具体问题，有点类似 PIP 判断问题。

先排除大部分数据，再对小部分数据进行精细判断。

从这里虽然没有获得有用的代码借鉴，但是我在查阅资料的时候，慢慢理解了这个大问题如何化解为小问题，并且这几个小问题的思路应该向什么方向思考。

B）KD 树和 Range 树

在 blender 的开源代码中，我只发现了一个能看懂的数据结构，并且猜测可能与 PJ 的存储索引方式有关，就是 RangeTree。不过在此之前我从助教给的几个方向出发，查阅了四分树和 KD 树。

国内实在是资料少…所以 Google 里面找了很多 pdf 看，四分树和 KD 树都不难理解（此时还没有看 R 树），后来发现这两种数据结构求解这次的问题都并不算特别好。后来我把方向放在了 Range 树中。Range 树是 blender 的一个模板类型树，目前我也不太清楚 blender 使用 range 树做什么，我打印下来完整的看完了一遍，代码量与 R 树差不多。

Range 树是利用左倾红黑树和顺序链表，将空间划分成一段一段的区间，然后能够更快的查找到一个区间。查资料期间看完了红黑树，当然看完是很有成就感的，不过后来发现 Range 树很难运用在这次作业中，首先 blender 的 range 树是一维空间的区间划分，二维空间的区间划分 Range 树也有，不过 google 的资料不仅少我已经看不懂了，超出我目前的能力范围了。至此 blender 的研究结束。

虽然没有找的合适的数据结构，但是从中收获了一些启示。

对于不同的问题，要根据问题本身出发寻找合适的数据结构。

C）R 树

R 树是大家都在尝试的方法，所以我一开始并不是特别想用这个结构，我猜测如果大家都用同样的方法做出来差距不会太大。所以我之前一直在广度优先的搜索资料，最后还是没能逃脱要看 R 树。

R 树实在是太过于复杂，我看了一半的函数就实在无法硬刚剩下的了，不过看了关键的几个函数。所以初次尝试时，使用 R 树开始开工。

D）PIP 问题

一开始我并没有把点在多边形的问题看得很重要，只是象征性的搜索了一下射线法，了解了其中的原理，然后搜索到了 Refence[1]，其中介绍了很多种判断方法，也让我开了眼界，最终初次尝试时使用了射线法。

2.初次尝试：R 树硬核存储

所有问题均使用 R 树，将数据直接存放在 R 树中。

第一个让我纠结的就是全局变量的使用。extern 虽然学过，但是还是还给了老师，在项目管理中，应该把所有的全局变量申明在.h 文件中，然后在 cpp 中初始化。不过一开始我是每个 cpp 里面都来一个全局变量，也把自己弄得够呛。后来发现只需要在 common 文件中定义全局变量，每次调用变量，在 environmentsettings 中清空（类似初始化）变量，就可以实现全局变量 6 个 submit 文件共享。

重复使用的变量，可以定义在 common 中。

然后是实现存放 polygon 和 point。首先定义 polygon 和 point 的类型，并且构造函数时，就构造好他们的外包矩阵，（点的外包矩阵就是点本身）多边形的外包矩阵就是点的最小坐标和最大坐标确定的矩阵。然后 R 树内部直接存储多边形和点。运用 R 树的 Insert 函数插入多边形或点，运用 R 树的 Remove 操作删除多边形和点，运用 Search 操作查找 R 树中的数据。对于多边形和点，使用不同的 R 树存储。当然初始化就是把树清空，全局变量中定义 R 树。这样的做法非常的直白简单，只是用库函数的调用，写几个构造函数即可。

当然这样的代价就是，出现很多差错。第一次跑大数据，case_1 就用了 **10 分钟**，正确率 1，我以为是电脑的问题，后来发现室友的 R 树暴力法做出来的也只需要 30s。我意识到有些问题，后来第一个发现的错误是粗心的错：vector 传参时用的全部是复制，忘记添加&符号。不过加上&之后还是需要 **4 分半**。

后来发现是 clion 的编译器选择问题。Clion 中我选择的编译器是 VS，我换了编译器 MinGw 后，跑只需要 **44s**。还是有哪些地方有问题。我意识到了，直接把数据存在 R 树中，是不是会使整个树结构庞大、并且封装性、索引方式并不是我想象中那么好。我才发现，不需要把整个数据库存放在 R 树中，存在 R 树外面也是一样的，R 树中只需要存一个矩形、和一个 counter 即可。Remove 此时还是暴力 Remove。

这是第一次开始使用一一对应的方法。

一一对应的好处很显然，就是不需要大批量的移动数据，R 树的结构也更加简单，内存管理会更加方便。我改好了这个过后，时间虽然没有怎么进步，还是停留在 **42s** 左右，不过我自我感觉这样做会比之前好很多，数据管理方面。不过肯定还有好多问题没有解决。

3.再次优化：R 树硬核优化

R 树中 Search 函数有一个 overlap 的判断函数，即判断矩形是否与另一个矩形重合，我思考这个是不是可以优化。因为多边形与点的问题，必须是完全在内部，才需要判断。于是我在 R 树模板中重新写了一个 overlap 函数和 Search 函数，专门用来搜索点、搜索多边形。我发现这样做并没有什么特别大的优化，还不如直接使用模板，所以又返回 R 树原模板。

不过尝试是好的，因为可以探索新思路。

我实在没有很多好的方法优化目前的结构，询问了其他同学。在陈疏桐同学的提示下，我明白了编译的 cmakelist 还需要优化，在网上查阅资料后，学会开启了-O2 的预编译。这

样 case_1 成功跑到了 **26s**，还是不够，还差的很远。

然后尝试了在 linux 里面测试了样例，case_1 跑了 **11s**。

至此，对于 R 树暴力的优化，我并没有发现更好的方法了。

使用 R 树就必须要找到多边形的外包矩形，就需要先对多边形进行 O（n）的遍历一遍，我认为是特别耗费时间的。然后在对每个点判断时，又需要用射线法进行 O（n）遍历一遍，又是非常耗时的方法。我开始思考是不是大部分时间都在判断点是否在多边形，粗测了一下去掉判断，直接 emplace_back，可以发现 case_1 在 linux 中只需要跑 3s 左右，说明判断真的花费了超级多的时间，所以我又把重心放回了判断的方法中。

R 树的结构并不是主题，判断方法反而占用了大部分时间。

4.改革换新：网格法初次尝试

在 Reference[1]中找到了很多方法，正如在“思考过程-判断”中写的一样，不与赘述。我寻找到了网格法，并且发现了它的优势。

网格法将索引与判断融合在了一起。

重点分析：之前使用 R 树的原因是，为了减少判断的次数，用一种数据结构存储大数据库，使用区间范围搜索的方法，降低查找范围，找到多边形、点所在的最小可能范围，在进行精细的射线法判断 PIP 问题。正如 blender 的 bullet 碰撞问题的分解，也是分为大空间排除、小空间判断的思路。

网格法使这个问题能够合并。

网格法对多边形进行预处理，对于每种多边形，预处理将空间划分为三个板块，其中两个板块的划分可以迅速排除 PIP 问题，在对于边网格的点问题进行下一步处理。这使得数据结构的改革有了新的方向，即直接使用网格，放弃存储 polygon-database。

还有一个优点是，网格法极其适用多个点在一个多边形的问题判断。

那么就只需要对每个多边形预处理，然后不保存多边形信息，直接存放网格即可。具体操作前面已经讲过，不予赘述。那么我还尝试了网格的优化，就是网格法中有一个关键的参数就是 Grid_Resolution，即每个多边形划分成几个格子。初始设定的是 20，对于大部分多边形都很有效。然后经过一系列尝试，由于在 case_1/case_4 中的多边形存储较多，网格法耗损内存较多，在效率和空间节省的平衡下，我发现 20 是一个很好的选择。然后在 case_2 和 case_5 这种，给定一个新的多边形，判断点集，可以分配更多的空间，则就有了选择：网格划分越细，判断越快，建网格越慢；网格划分越少，判断越慢，建网格越快。在一系列尝试后，我选择了 Grid_Resolution = 85，这是一个较为平衡的点。测试内容如下。

对于 case_4 – windows 系统的试验：

Grid_Resolution test

20:5.589

40:5.372

50:4.97

60:5.093

70:4.925

80:4.836

85:4.827

90:4.903

100:4.912

5.网格法优化：删除操作的尝试

在动态问题中重要的就是删除操作。之前 R 树中我使用了 Remove 函数硬核删除数据，后来经过助教的提醒，发现可以用标记法进行“伪删除”操作，看似删除实则未删除。

当然网格法也可以这样实现。首先我存储数据使用的是一一对应 id 的方法，那么删除操作也是一一对应，那就设置一个 disable 容器存储“是否已被删除”的信息。前面已经提到过，就不予赘述。

此时我遇到了一个严重的问题，查找 disable 中的特定 id。

由于之前 disable 容器使用了顺序存储，方便是方便，因为判断时只需要 O（1）的复杂度得知是否被删除，但是如果要删除某个特定的 id，就要在 disable 容器中遍历一遍，查找这个 id，这个**平均复杂度是 O（n）**。我发现这个问题后，也思考了很久，找到一种解决问题的方法，就是另外在建一个索引表，专门用于查找删除的元素，这次反过来存，存放 id-counter，并且使用 hashtable 存放，尽量降低为 O（1）复杂度查找需要删除的 id。

这个思想就是一直秉持的思想，空间换时间。

我在创立 disable 容器时，就创立好 id-counter 的 hash 表，用拉链法解决冲突问题，用除数取余法构建 hash 函数，这样按理来说会快很多，但是！实际告诉我并没有……我对使用 hash 查找和不使用 hash 进行顺序遍历查找，所用的时间基本一样，hash 查找甚至还需要更多的时间。这让我的确摸不着头脑。并且只建 hash 表、用顺序遍历也是一样的时间，说明建表并不是时间花费所在，而是在这个案例中，顺序查找和 hash 查找的时间复杂度随不同，但是实际上时间花费差不多。所以最终还是采用的原来的删除方式，顺序遍历找到 id 值，disable 之。

果然，实践是检验真理的唯一标准。

6.六个问题分析：一个思想、六种运用

讲述到这里，我认为我的思路基本讲述完整了。对于六个小问题，我采用的都是一个思想，网格法。需要存储的多边形，先预处理，再存储；需要存储的点，直接存储；需要查找的多边形，直接顺序遍历存储的点集，直接判断；需要查找的点，直接顺序遍历存储的多边形网格集，直接判断；需要删除的点，顺序遍历 ID，disable 之；需要删除的多边形，顺序遍历 ID，disable 之。

总而言之，就是运用网格法存储、空间换时间，索引与判断融合、缩小问题规模，一一对应存储、顺序暴力求解。

7.项目管理：Local History

注册了 Git 仓库不太会用，发现了 Clion 里面本身自带了一个叫做 Local History 的东西，可以直接恢复到之前的任意版本，特别方便，在做项目是也多次使用这个恢复。

五、结语与感想

网格法，看似简单，实则蕴含了很多我的思考的成分与取舍，为什么要选取网格法、为什么要使用一一对应封装存储数据、为什么不采用 R 树存储数据、为什么不选用射线法，这些等等问题都已经我的脑海里思考过，并且内化过。数据结构重要的是思考的过程，如何找到符合问题的解的最佳方法，才是符合实际问题的最优解。

从最开始的三维建模软件到最后的网格法的思考，实际上是一个探索的功能，利用这个世界上相同的原理、相同的方法运作的各种机能体系，去探索一个一个实际应用问题背后的道路。从一个思维点，发散到多个思维面，我想这才是这个小小的“地理围栏”想要告诉我们的。多年以后的程序员可能会忘记 R 树，可能会忘记网格法，可能会忘记点在多边形的问题，但是其中蕴含的大化小、小化无的思路能深深的印刻脑海中，陪伴我以后的编程之旅。

希望以后的我也能记得。

六、Reference

[1] http://erich.realtimerendering.com/ptinpoly/ 网格法