《计算几何》课程实验报告

——动态数据结构（KDS）下的二维最近邻点问题

杨　磊(2004310482)　阮晓宇(2004210962)　余士良(2004211059)

Mails:{yanglei04,ruanxiaoyu00,yusl04}@mails.tsinghua.edu.cn

1　实验内容

在二维平面上,有任意有限个运动的点，问题是如何实时的报告出所有点中距离最近的那个点对。二维平面可以是有界，也可以是无界。点的运动可以是匀速运动，也可以是任意方程为多项式函数的运动，在点的运动过程中，其运动计划（flight plan）可以有有限次数的变化。比如在有界平面中，点在碰到平面的边界是会发生弹性反弹，这即会引起点的运动方程的改变。

2　算法原理

这一问题直观的 bruteforce 解法是：把时间离散为一个个时间点，求出每一个时间点各个点的位置，然后在使用相关的方法求出最邻近的两个的点，比如求出任意两个点之间的距离，然后进行比较得出距离最小的两个点。这种解法的优点是直观、易懂，缺点是：

1) 时间复杂度高。在每一时刻点为求出最邻近的点对，都要开销 $O(n^2)$ 的时间。
2) 把时间离散为一个个时间点的做法，并不能满足实时的要求。离散的精度影响实验的结果。如果相邻的两个时间点的距离比较大，则在相邻两个时间点之间一个点对由非最邻近变为最邻近再变为非最邻近的可能性很大，而这样的点对是无法被报告出来。相反，如果相邻的两个时间点的距离比较小，则漏报的可能性就比较小，但系统的处理开销会成倍的增加。

上面提到的直观算法缺点的根源就在于它没有注意到点的位置变化是连续的这一特性。由于点的位置变化是联系，所以本来距离很远的点对不可能一下子变成最邻近。对于这样的点对，在计算最邻近点对时可以暂时不用考虑。为了利用这一特性，Julien Bash、Leonidas J.Guibas 和 John Hershberger 提出了一种基于动态数据结构（Kinetic Data Structure）算法，有效的解决了这一问题。下面将详细介绍这一算法。

在以每一点为中心把平面分成六个顶角为 $60°$ 的楔子。如果有 N 个点，就有 6N 个楔子。在每一个楔子内部都分布着如果个点（$M \geq 1$），如果 $M \geq 2$,则在每一个楔子中能够找出一个最短距离点对。通过观察可知：N 个点的最短距离点对

批注 [Jordan1]: Wedge

是其中某些楔子中的最短距离点对。

上面的观察结果可以进一步的简化为：以每一点为中心把做其$-90°\sim-30°$、$-30°\sim30°$、$30°\sim90°$三个楔子，总共有 3N 个楔子。N 个点中的最短距离点对是其中某一个楔子中的最短距离点对。

$Dom(p)$表示 p 的一个楔子所覆盖的领域。

现在来观察楔子的性质。假设 a、b 是所有点中距离最短的点对。不失一般性，假设$a \prec b$（ax<bx 或 ax=bx 但 ay<by），这样 b 总会落在 a 的某一个楔子中，不妨假设落在$-30°\sim30°$的楔子中。如图 1 所示。

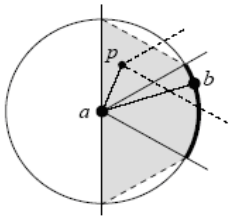


图 1　引理 1 证明

引理 1：b 不可能落在其他点 p 的$-30°\sim30°$的楔子中，其中$a \prec p$。

证明：如果存在点 p，满足$a \prec p$，且 b 落在 p 的$-30°\sim30°$的楔子中，则 p 必定落在图一中的阴影部分。如果 p 落在图一中的阴影部分，则必然有 pb<ab, 于假设的前提相互矛盾。

引理 2：b 是 a 的楔子中最左边的点。

证明：如果如图二，假设在 Dom（p）中存在另一个点$b^{'}$，且$b' \prec b$，则b'必然落在图 2 的扇形区域中或阴影部分。很显然，$bb' < ab$，这与假设前提矛盾。

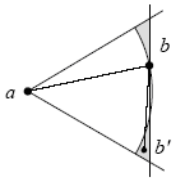


图 2 引理 2 证明

$\bigcup_{\substack{q\in S \\ p\prec q}} Dom(p)$

Maxima(p)表示的是构成　　边界的点集。

Cands(p)表示构成 $Maxima(p) \cap Dom(p)$ 点集。

lcand(p)表示 Cands(p)中最左边的点。

Parent(p)表示　$\{x \mid x$所对应的楔子有一条边被p所对应的楔子的边截断$\}$

Parent^a(p)表示的是 Parent(p)中位于 p 点之上的那些点。

Parent^b(p)表示的是 Parent(p)中位于 p 点之下的那些点。

具体表示如图 3：$p \in Parent_b(a)$　　$p \in Parent_a(a)$

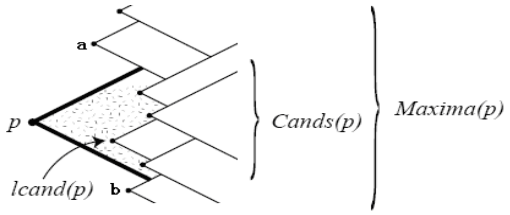


图 3　parent、candidate 等图示

上述的观察结果可以表述为：N 个点中距离最短的点对必然示所有的（p,lcand(p)）中的一个。因此只要找出所有的(p,lcand(p))点对，并求出他们的距离，再从中找出距离最短的那个点对就所有点中距离最小的点对。

对于运动的点集，只要维护好（p,lcand(p)）这一数据结构，就可以实时动态地求出任意时刻的最近点对。

引理 3：对于点集的两个不同分布（绝对位置不一样）或者叫配置 configuration，如果他们在$-60°$、$0°$、$60°$三个方向上的排序是一样，那么这两个配置的 Maxima、Cands 和 lcand 是相同的。

这里我们称任一方向上的一对邻近点交换一次顺序为一个事件。也就是说每一事件都会引起配置的改变，相应地引起 Maxima、Cands 和 land 状态的改变。

点集从配置 a 转化为配置 b，这个转换有多种不同的实现方案。其中每一个方案由若干事件组成，每一事件对应一种配置，这样每一方案对对应一种配置转换序列，也对应一种状态转换序列。根据引理三，尽管状态转换序列各不相同，但最终的配置 b 对应的状态是相同。这样，假设如果有多个事件同时发生，可以以任意的顺序对他们分别进行处理，其顺序并不影响最终的状态。

下面介绍如何构建状态，以及在事件发生时如何修改状态。

在构建状态时采用的是扫描线算法。这里只介绍$0°$方向上状态的建立，其他两个方向上与此类似。

首先先根据各点在 X 轴上的投影对各点进行排序，然后根据投影的从右到左

依次把各点加进来。同时生成的有各点对应的 Cands、lcand、Parent^a 和 Parent^b 等结构。为了生成这些结构，Maxima(p)起到了辅助的作用。

如图 3，p 点引入之前的时刻 Maxima 的状态就是图中除 p 点外的所有点。在引入 p 点的时刻，从图中可以看出原来的 Maxima 被分成三部分，一部分位于 Dom(p)上面，一部分位于 Dom(p)下面，一部分位于 Dom(p)内部。其中位于 Dom(p)内部的构成了 Cands(p),位于 Dom(p)上面、位于 Dom(p)下面和 p 共同构成了新的 Maxima。同时要把 p 点加入 Parent(a)和 Parent(b)。在 Cands(p)中找出最靠左边的点为 lcand(p)。

由于点的运动是相互联系的，所以在考虑点投影顺序交换事件的时候可以只考虑在轴上投影相邻两个点交换投影顺序的事件。

图 4

假设 X 轴上有两个点交换位置，那么可能导致 X 轴方向上的状态发生改变的情况不外乎下面两种：

第一种情况：类似图 5 的情况。

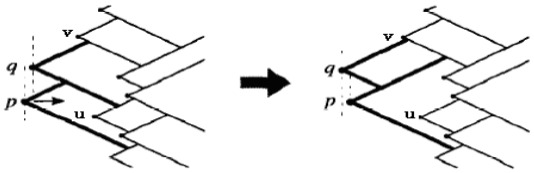


图 5

关于这种情况的处理如下：

- 把 Cands(q)$\cap$Dom(p)从 Cands(q)中去除，并加入 Cands(p)。
- 把 q 从 Parent^a(u)中删除，并且把 q 加入 Parent^a(p)。其中 u 满足 q$\in$ Parent^a (u)把 p 从 ParentL(q)删除，并且把 p 加入 Parent^b(v)

第二种情况：类似图 6 的情况（从 a 到 b 或者从 b 到 a）。

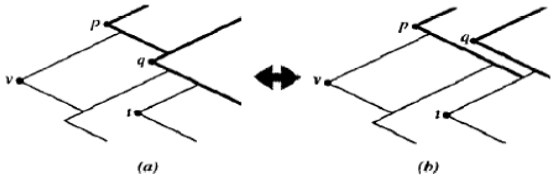


图 6

先讨论由 a 到 b 的情况：

- 把 q 从 Cands(v)中删除，然后加入 Cands(p)
- 把 p 从 $Parent^a(q)$中删除，然后加入 $Parent^a(t)$.其中 t 满足如下的条件：

t 是 $Parent^a(q)$中位于 p 左边的那些点中最左边的点；如果这样的点不存在，则 p 是满足 $q \in Parent^a(t)$的点。

- 把 $Parent^b(q)$中位于 t 左侧的那些点从 $Parent^b(q)$中删除，并加入 $Parent^b(p)$的底部。

由 b 到 a 的过程与由 a 到 b 的过程相反。

算法复杂度分析：

每处理一个交换顺序事件的时间开销是 O(logn)。这类事件的处理开销主要在更新 lcand、 parent 、cand 和更新竞争树上。更新 lcand、parent 和 cand 方法如上面所介绍的，处理的开销是 O(logn)。更新竞争树的开销也是 O(logn)。

此外，系统的时间复杂度还取决于处理事件的频率。处理事件的频率与点的个数有关，还与点的运动速度和点的运动区域的大小有关。一般来说，点的个数越多，运动的速度越大，运动的区域越小，事件发生的频率就越大。要试图通过数学分析的方法得到具体的关系表达式不是很容易,我们通过实验的过程中观察具体的事件数，详见报告第 5 部分。

3　小组分工

杨磊：负责系统结构设计、kds、kcp、kwall、cert 等算法和结构实现

阮晓宇：负责用户界面设计实现、仿真结果图形显示的 openGL 实现

余士良：负责算法分析、数学函数库编写和红黑树结构实现

4　程序结构与算法实现

4.1　kds 公共体系结构

kds 公共体系结构包含了整个仿真系统中处在高层的，和具体 kds 无关的抽象结构的集合，同时也包含了仿真控制和基本数据结构（比如点集、事件队列）。在设计这些结构时，我们参考了参考文献[3]中 CGAL 库 KDS 结构和原作者提供的部分 KDS 算法结构。

批注 [J2]: 利用已有成果

kds 仿真部分的体系结构如图 7 所示：

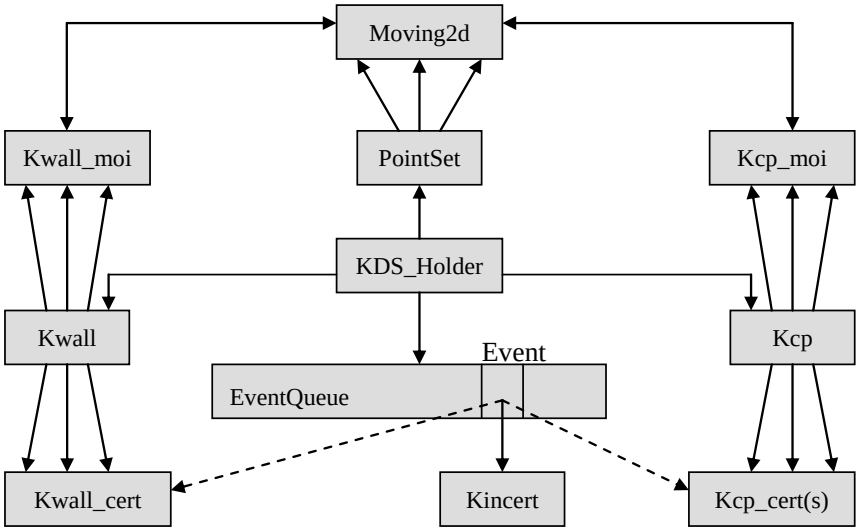


图 7　kds 仿真系统体系结构简图

为了完成点运动仿真以及动态数据结构的计算，程序包含以下几个关键结构（类）：

- pointset：运动点集，包含平面上所有的运动点对象 moving2d，每个对象包含自身运动方程和当前的位置坐标，为所有运行中的 kds 所共享
- kds: Kinetic Data Structure 一种动态数据结构的整体的描述。kds 是所有动态数据结构的基类，其派生类有 kcp (Kinetic Closest Pair)、kwall (Kinetic Wall) 等，分别维护最近邻点、墙壁的动态数据结构。kds 中存放着与之相关的运动点控制对象如 kcp_moi、kwall_moi 等，也保存着它所用到的所有 certificates 如 kcp_cert、kwall_cert 等。
- event queue: 事件队列，保存着所有 kds 的 certificate 所对应的事件。事件队列按照时间排序，采用平衡二叉树结构存储，以保证事件插入删除的效率。程序运行的主要流程就是不断地从事件队列中弹出时间最早的事件并处理：

```
WHILE （事件队列不为空）
    取得队首的事件 e
      IF （e.time < 本帧模拟的结束时间）
        弹出事件 e，把当前时间推进到 e.time，
        执行事件处理 e.process_death()
      ELSE
        BREAK
```

- kincert: Kinetic Certification. 算法中各类 certificate 的基类。certificates 和相应的 kds 相关联，在相关事件发生的时刻由事件队列弹出并执行

process_death 虚函数。该函数的执行即代表了相应事件的处理，包括维护数据结构，产生新事件等。

- moving2d: 与特定的 kds 无关的运动点对象。
- movingobject_item: 与特定 kds 相关的运动点对象，如 kcp_moi，kwall_moi 等的基类。一个 moving2d 根据特定的 kds 关联多个相关的 movingobject_item 的派生类。当 moving2d 改变运动计划（flight plan）的时候，可以通知各个与之相关的派生类，以使其 certificates 重新计算。
- kds_holder: 整个动态数据结构模拟的主控类。包含移动点集 pointset，事件队列 event queue，和所有在运行中的 kds 列表。系统通过此对各 kds 和仿真运行过程进行访问控制。

上面所描述的几种继承关系实际上是不同的 kds 所带来的平行的继承关系。比如 kcp 相关的派生类有 kcp_moi，kcp_cert 等。它们的关系可用图示表示如下：

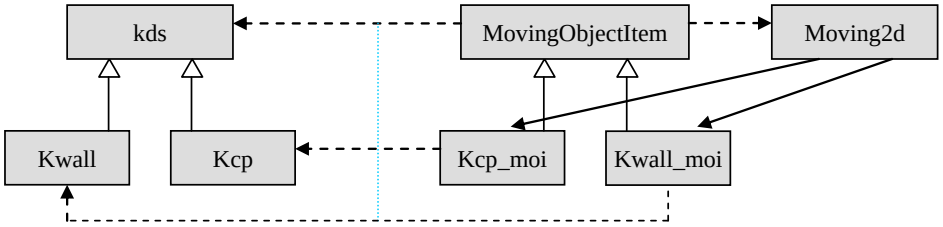


图 8　kcp_moi、kwall_moi 与 kcp、kwall 等的平行继承关系

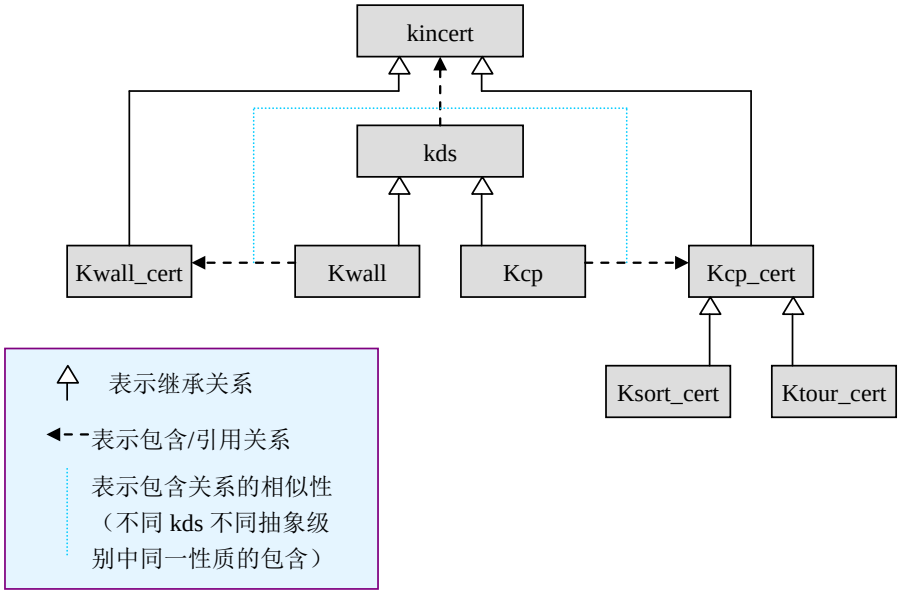


图 9　kcp_cert、kwall_cert 等与 kcp、kwall 等的平行继承关系

4.2 kcp 最近邻点动态数据结构

各类之间的继承关系在以上几幅图中已经有所介绍。本节主要介绍最近邻点动态数据结构派生类的结构和功能：

- kcp_moi: kcp_moi 是公共数据结构中二维移动点在 kcp 最近邻点结构中的一个映像。它主要包含了在 kcp 算法中所维持的关键结构，具体列出如下：
  - proj[3]，在每个轴（三个相互 60 度的轴）上的投影点 ksort_moi
  - lcand[3]，在每个轴方向上的最近邻点（自身在左作为主点）ktour_moi
  - lcanded[3]，在每个轴方向上的最近邻点（自身在右作为从点）ktour_moi
  - cand[3]，dom 范围内所有点所组成的红黑树 candidate tree
  - parentU[3]，所有从上方射向本点 dom 区域的点组成的红黑树
  - parentL[3]，所有从下方射向本点 dom 区域的点组成的红黑树
  - relNode[3]，本结点在另一个结点的 candidate tree 中的位置指针
  - cuNode[3]，本结点在另一个结点的 parentU tree 中的位置指针
  - clNode[3]，本结点在另一个结点的 parentL tree 中的位置指针
  - lcando[3]，在三个轴方向上最近邻点

  可见 kcp_moi 几乎保存了所有的算法相关的关键数据结构。这些数据结构的更新和维护计算量相当的大，有很多是由相关对象自己完成的。kcp_moi 主要完成这些数据结构的综合，以及 kcp 算法演示结构的绘制。这些结构与 kcp_moi 之间的部分关系可由下图所示（限于空间，途中只绘制了部分关系，事实上很多关系都是双向的）

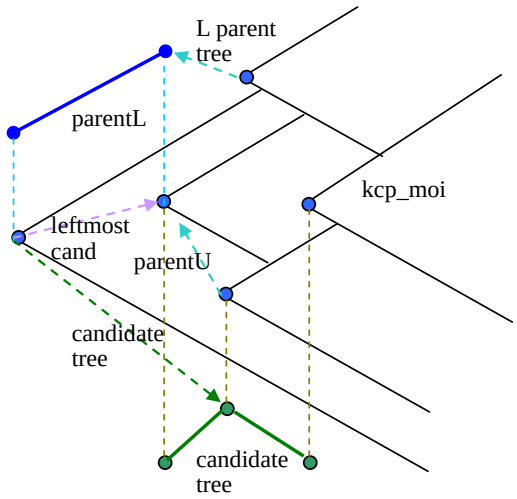


图 10　kcp_moi 主要控制的数据结构关系简图

- ksort_cert 和 ksort_moi

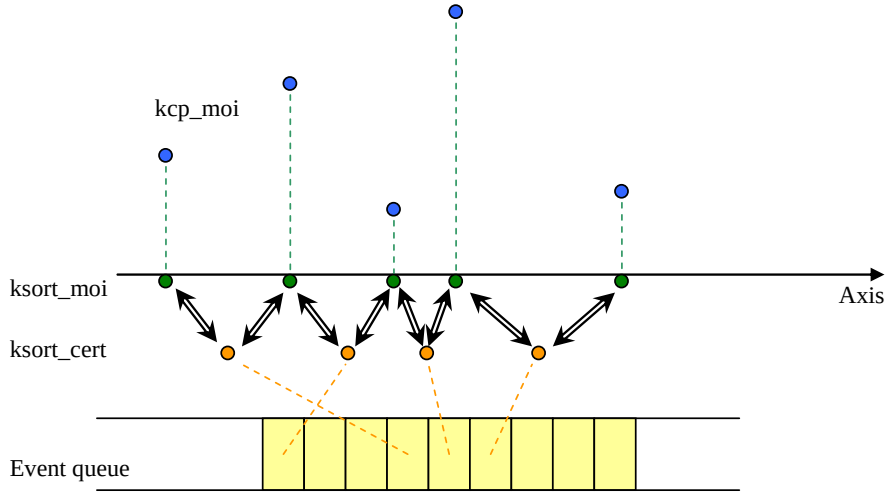
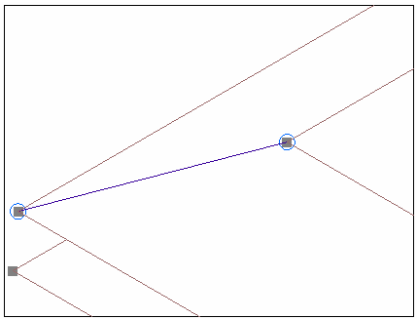
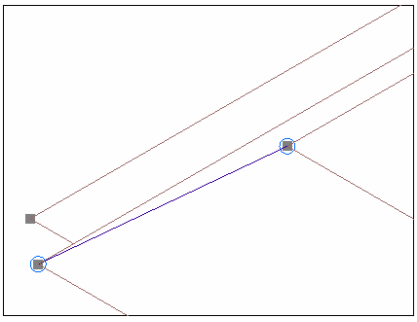
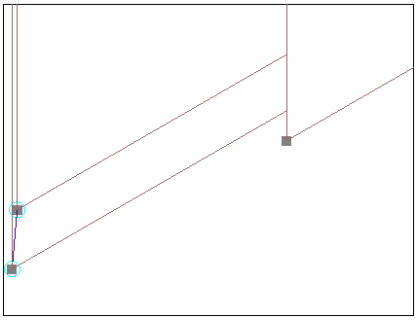
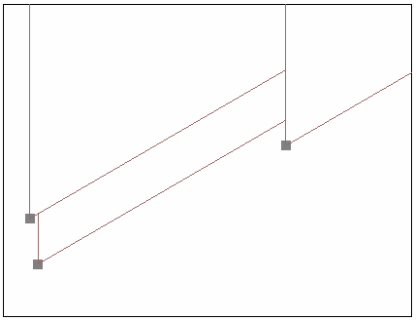
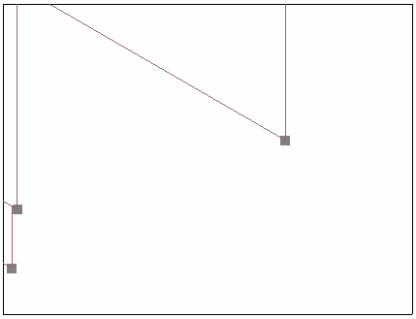
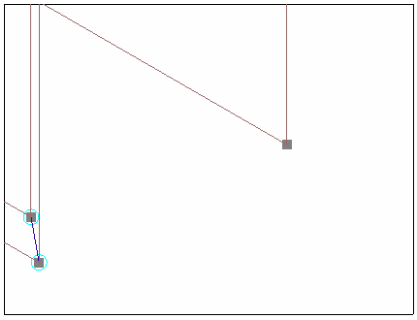


图 11　ksort_cert 与 ksort_moi 关系简图

ksort_moi 和 ksort_cert 负责处理运动点在三个轴线方向投影排序变化的情况。每一个运动点 kcp_moi 在每一轴线方向都有一个对应的投影对象 ksort_moi，而这些点的排序关系由相应的 certificate 对象来保证。如图所示，每两个相邻的投影 ksort_moi 之间都有一个关联的 ksort_cert，它负责维持 ksort_moi 的前后关系。一旦有两个相邻点的前后位置发生交换，则这两个相邻点之间的 ksort_cert 所对应的事件队列里的事件就会弹出并得到处理。排序位置交换事件的处理流程如下：

- 按照排序顺序的交换规律更新相邻的 ksort_moi 和 ksort_cert 之间的指针连接关系，使得两个点交换顺序后 ksort_cert 仍然能够维持点之间正确的排序关系。
- 重新规划本 ksort_cert（发生交换的两个点之间的 cert）和与其相邻的左右两个 ksort_cert 发生时间，调整它们在事件队列中的位置。
- 调整算法结构中各点的关联关系：

  0 度轴方向上排序顺序的变化不但会影响 0 度轴 parent、candidate 等结构的变化，也会影响其他两轴上这些结构的变化。下表列出在 0 度轴排序顺序变化时对各种数据结构的调整例子，其他两轴上排序变化时的调整方式类似。图中示意的仅仅是最简单的情况（cand/parent 树上都仅有一个结点，直接交换、增删就可以，实际算法要更复杂一些，因为牵扯到树结构的维护操作）。

<table>
<tr><td colspan="3">左边两点在 0 度轴上交换顺序的情况：</td></tr>
<tr><td>投影轴</td><td>交换前</td><td>交换后</td></tr>
<tr><td rowspan="2">0°轴</td><td>  </td><td>  </td></tr>
<tr><td colspan="2">● 交换两点的 parernt 关系相互发生变化：新的左点下射向新的右点
● 交换两点的 candidate tree 中相互重叠的部分由旧的右点传给新的右点</td></tr>
<tr><td rowspan="2">60°轴</td><td>  </td><td>  </td></tr>
<tr><td colspan="2">● 新的左点从新的右点 candidate 树中除去（因为已经移出其 dom 域）</td></tr>
<tr><td rowspan="2">150°轴</td><td>  </td><td>  </td></tr>
<tr><td colspan="2">● 新的左点增加到新的右点 candidate 树中（因为已经移入其 dom 域）</td></tr>
</table>

■ 更新交换顺序两点的 dominator 的 left candidate

由于两点左右顺序交换，如果该两点共同的 dominator（以该两点为 candidate 的点）的 lcand 为原左点的话，则该 lcand 需要更新为新左点，并将此改变重新传递到竞争树，重新计算一些关键变化发生的时间。如图 12 所示：

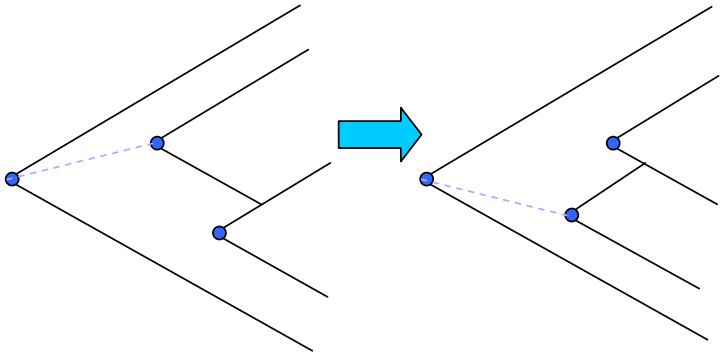


图 12　left candidate 交换示意图

另外一个问题是如何计算一个 ksort_cert 对应事件发生的时间，也就是两个点在某轴上交换位置的时间。由于我们有每个点的运动方程，因此将相邻两点的运动方程相减，在该轴上投影，并计算投影值为 0 的时间，也就得到了这两个点相遇的时间。将这个时间与该 cert 关联起来放入事件队列，就能保证仿真程序能在正确的时间处理该事件。如果有相关点的 flight plan 发生变化，则这些点的 moving2d 对象会通知关联的 kcp_moi，kcp_moi 也即会将该变化报告到 ksort_moi，进行新的交换时间的变化。

- ktour_cert 和 ktour_moi

  ksort 系列对象用于维护图中楔形结构以及 candidate 对象等。而 ktour 系列对象则用于维护和计算最近邻点的结果。根据算法，最近邻点的结果必然出现在三个方向图的 cand 对中，因此，我们将所有的 cand 对组成一个动态竞争树的叶子结点，竞争树的树顶即该图的最近邻点对。三个最近邻点中的最小值即是最终的最近邻点结果。

  ktour_moi 代表点和其对应的 lcand 点对，每一个 kcp_moi 都对应一个 ktour_moi，这样就构成了竞争树中的叶子结点。竞争树的内部结点由 ktour_cert 构成，每个 ktour_cert 和其他的 certification 一样，都和事件队列中的一个时间关联。这些事件当且仅当结点的 winner 最近邻点对发生变化，或者其左右孩子的最近邻点大小关系发生变化的时侯才会发生。每一个 ktour_cert 都会计算其左右结点 winner 最近邻点对发生变化时间和左右孩子的最近邻点大小关系发生变化事件的最小值，将这个最近的时间与自身关联放在时间队列中。

  当 ktour_cert 所对应的事件发生时，进行如下的处理操作：

  1. 如果原来的 winner 是左孩子的，则换为右孩子的，反之亦然。
  2. 沿着竞争树上行“过滤”，将所有 winner 为本结点 oldwinner 的都更新为本结点的 newwinner。
  3. 如果“过滤”到达树根，则更新最近邻点的结果。

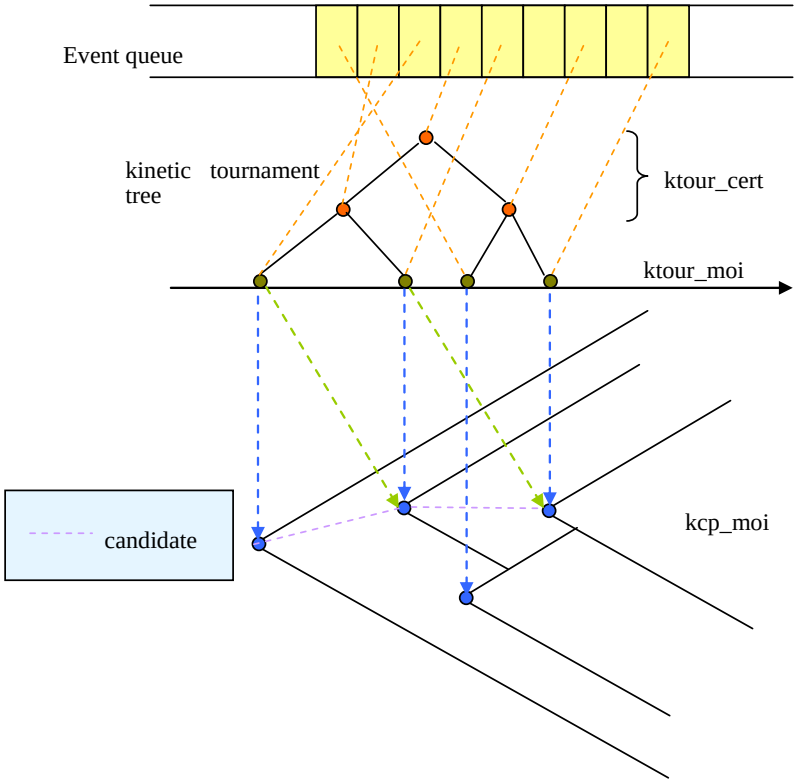


图 13　tournament 结构与 kcp_moi 关系与竞争树结构

4.3 kwall 围墙动态数据结构

围墙控制主要是用于处理运动点和四周墙壁的碰撞事件。kwall 本来不需要作为一个 kds，但考虑到整个系统的一致性和可移植性，将 kwall 作为一个动态数据结构 kds 的实例来实现。与其它 kds 不同，kwall 可以直接控制点的运动。当点碰到墙壁时，kwall 按照弹性碰撞的规律重新设定点的运动速度。按照 kds 的运行原理，每一个点与墙壁碰撞的事件作为一个 certificate 来设计。点与 kwall_cert 的关系如图 14 所示。

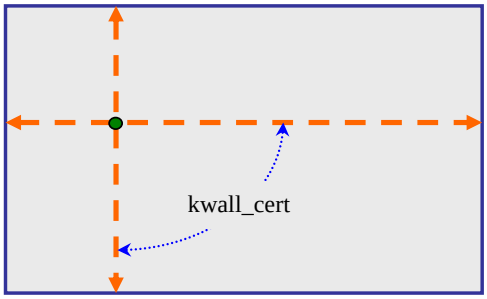


图 14　kwall_cert 示意图

由于各点的运动方程已知，计算各 kwall_cert 事件发生的时间非常容易。kwall_cert 的处理（process_death）也即根据撞到的墙壁而将点的 x 方向速度或 y 方向速度取相反值。

4.4 关键数据结构的实现

- 事件队列：采用 C++ STL map 容器实现。map 容器内部采用红黑树结构，可以保证插入、删除、查找事件的时间为 O(logn)。红黑树为有序结构，因此提取队列首事件的时间为 O(logn)。
- lcand、parentU、parentL 等楔形结构，用红黑树存储。算法里面由于需要在两个点寻找其共同 dominator 时沿着红黑树上行，因此在普通的红黑树结点中加入父结点指针。这些指针仍然可以在保持时间复杂度的前提下进行维护更新。此外，每个内部结点还保存着具有最小权值（在相应轴上从左到右排序序值）的子结点指针，这些都为相应操作减小了复杂度。
- 竞争树：竞争树的结点为 ktour_cert，如上所述每一个结点都是一个 certification。竞争树的构建采用一个辅助的 ktour_cert 队列，初始时队列中为每个 ksort_moi 相关联的 cert，每次从队首弹出两个 ktour_cert，建立一个以这两个 cert 为子结点的 ksort_cert 加入队尾，如此进行下去直到队列为空，即生成一棵竞争树。

5 算法测试结果

硬件平台：

CPU:Intel® Pentium(R) 4 CPU 1.70GHZ

内存：SDRAM 512M

图形加速卡：NVidia TNT2 32M

操作系统：Windows XP® Professional SP1

编译平台：Visual C++ 6.0 enterprise edition sp6

我们使用不同的点数、速度、加速度对程序进行测试，测试的指标主要包括程序运行的流畅性，这反映了程序的负载能力。测试结果表明程序的负载能力是比较令人满意的：系统的负载能力既不取决于点数，也不取决于速度或加速度，而是取决于事件发生的频率。在上述的系统中，如果事件发生的频率达到 12000 件每秒钟，在程序运行中就会出现停顿的现象。当点数再增多时，每秒钟能够处理的事件数达到了极限值（约 15000）。

批注 [Jordan3]:
来自实验统计
是否可以从理论上做一分析？

通过测试，我们还分析了在同一速度同一加速度同一运动区域的条件下，事件队列长度、事件发生的频率和运动点数之间的关系，得出如下的结论：在同一模拟速度和点数随机速度分布相同的情况下，事件队列的长度和运动点数是线性的关系，而事件发生的频率与运动点数之间是平方量

级的关系。图 15、16 分别显示了正常模拟速度和 0.2 倍模拟速度下事件队列长度 / 处理速度与点数的关系。

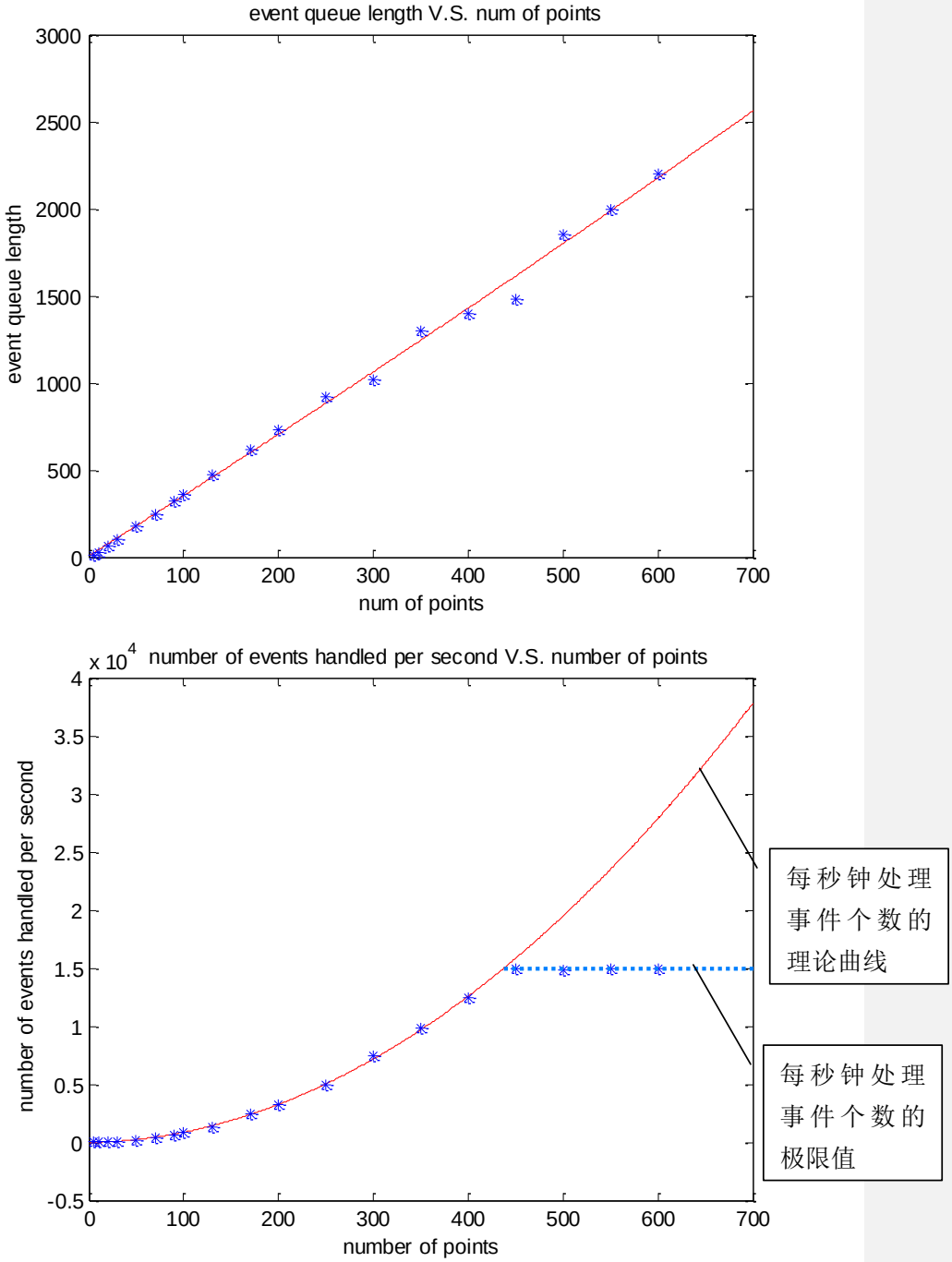
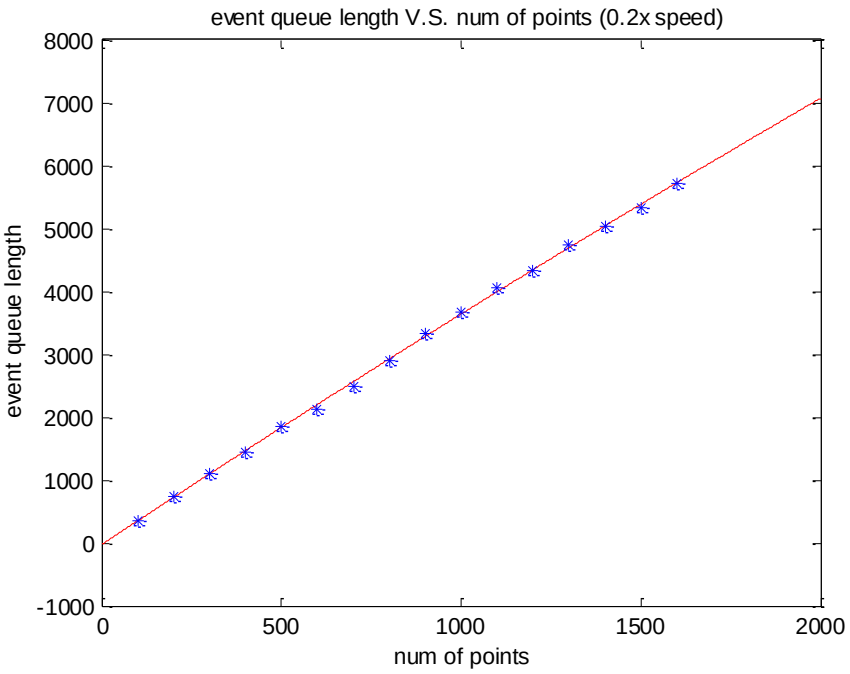


图 15　正常模拟速度下事件队列长度和处理速度与点数关系



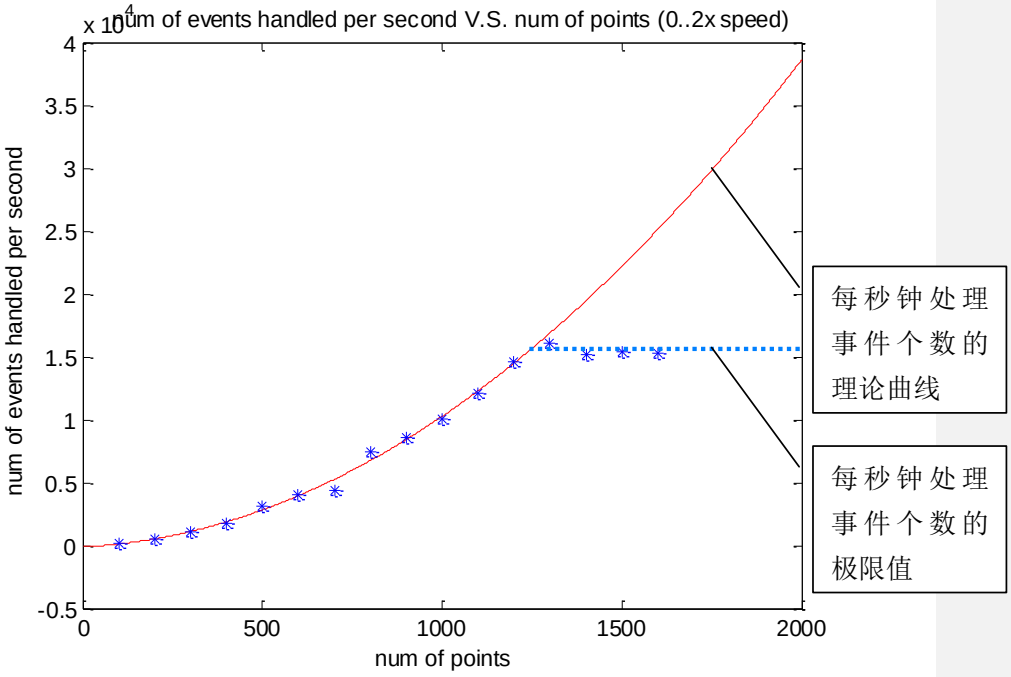


图 16　0.2 倍模拟速度下事件队列长度和处理速度与点数关系

事件队列的长度和点的个数成正比这个结论从算法本身比较容易分析，

因为总的 certificate 个数和点的个数是成一个线性的关系的。至于每秒钟处理的事件个数和点的个数成平方量级的关系，这个从概率统计的角度也能够理解，在相同矩形面积、相同速度分布的情况下，点与点之间交换次序的可能性确实是和点的个数成平方的关系。容易看出，在点的数量比较多（比如 1000 以上）时，如果不能降低各点的速度或者减少点的稠密程度，那么事件的数目将会非常大，从而使实时处理成为不可能。这是该算法的一个缺陷，也是所有以点之间的相互关系为基础的算法共同的问题。

另外由于受到计算机只能表示有限精度的数据的限制，程序无法精确的表示两个相邻事件的前后关系，在点数非常多、运动速度比较快而区域大小有限的时候可能导致两个事件的处理顺序颠倒，从而造成程序无法正确的维护动态数据结构，也就导致的程序运行结果的错误。这种情况在运动点比较稀疏的情况发生的可能性很小（我们测试的过程中从未碰到过），但在点的密度较大的情况下，错误发生的概率较大，而且随着运行时间的增长，发生的可能性就越大。一旦发生一次这样的错误，算法所必需的楔形结构就遭到破坏，进而引起更多错误发生的连锁反应。这个问题在参考文献 1 中已经提到，但是原作者并没有给出实际有效的处理方案。

6　进一步工作方向

由于受到计算精度的限制，出现两个事件的处理顺序颠倒等情况是不可避免的。但由于程序中的动态数据结构是都是基于它的前一时刻的状态。如果某一时刻开始，状态发生错误，这种错误就会引起连锁反应。我们认为算法应该引进一种程序自我检测机制，能够及时的发现错误并纠正过来，避免程序的错误扩散而导致程序的崩溃。

另一个问题是关于算法的实时性。由于上面所提到的原因，在事件数目很多的时候不太可能保证实时的处理所有的事件。一种可能的解决方案，也是作者在[1]中提到的，是否有可能只维持部分正确的数据结构，或者能够一次（批）处理多个相关的事件。这样的做法也有可能解决上面计算错误的问题。

另外，算法所依靠的楔形结构目前的内部表示方法（红黑树）不够稳健也是算法容易出现数值计算错误的一个主要原因。我们觉得邓老师提出的利用消隐来绘制演示图像的方法实际上是有可能用于改进数据结构存储的，如果能够找到有效的、只依靠楔形的次序来决定结构的表示方法，那么上面两个改进的目标就有可能实现，从而使算法的鲁棒性大幅提高。（由于经测试绘制相对于事件处理而言开销是可以忽略的，加之时间有限，我们并没有采用消隐绘制的方式。关于老师所提出的界面操作不够方便等问

批注 [Jordan4]:
实验过吗？

题我们都进行了相应的改进。）

7　参考文献

[1] J. Basch, L. Guibas, and J. Hershberger. Data structures for mobile data. In Proc. 8th Annual ACM-SIAM Symposium on Discrete Algorithms, pages 747–756, 1997.

[2] L. J. Guibas. Kinetic data structures: A state of the art report. Proc. 3rd Workshop on Algorithmic Foundations of Robotics, P. K. Agarwal, L. Kavraki, and M. Mason, editors, A. K. Peters, 1998.

[3] Leonidas Guibas, Menelaos Karavelas, Daniel Russel, A computational framework for handling motion. 6th Workshop on Algorithm Engineering and Experiments, 2004

8　用户手册

为了能够清楚的反映系统的原理，如前所示这里采用分视图显示的方式。其中包含一个主视图和三个分视图，主视图主要显示当前所有点的最近点对；其余三个分视图则主要显示的是三个 60 度角范围内的所有点的最近点情况以及所有点间在相应 60 度范围的边界相交情况，具体的原理参见算法分析部分。程序演示示意图如下所示：

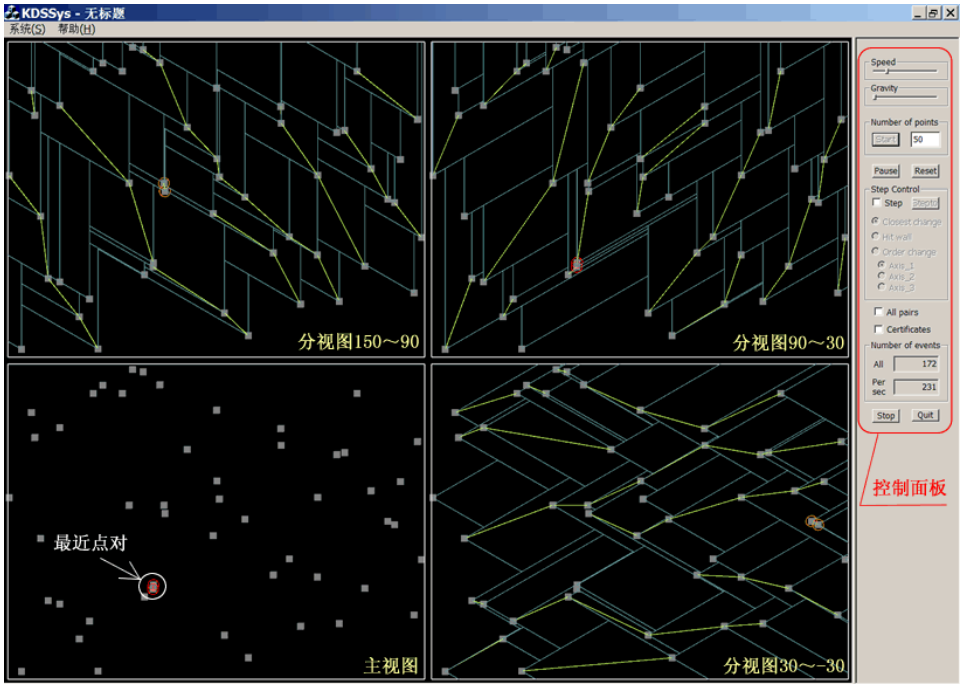


图 17 程序演示示意图

上图中左下视图为系统的主视图，其中红圈包围的两个点为当前时刻的最近点对；在各分视图中分别也有被圈住的最近点对，不过它们都是在各视图对应方向范围内的最近点对，再通过比较这三个最近点对中哪对的距离值最小，就将对应视图的最近点对用红圈包围，否则用桔黄色圈包围。示意图右侧用红圈包围的区域是系统的控制面板，其中包含了对系统仿真速率的控制，可以对每个点的运动添加重力加速度（加速度方向为竖直向下）；另外程序还提供了 Step 单步的功能，其中有 3 中模式：最近点改变、点撞击墙、在某个方向点的顺序发生变化。这其实是系统中出现的事件的类型，这里就是分事件来单步运行可以是系统的流程更加清晰。另外此系统还可以更改仿真中的运动点数，同时实时显示当前事件队列中的事件数和每分钟处理的事件数。具体操作流程如下所示：

设置系统仿真的点数；

点击 Start 按钮，系统进入仿真状态；

此时可以设置系统仿真的速率（如控制面板上的 Speed 滑动条），以及设置点的运动重力加速度（gravity 滑动条）；

Pause 按钮用于控制系统的暂停和恢复；

Reset 按钮控制系统重新开始运行；

Step Checkbox 按钮用于控制是否打开 Step 调试模式：

Step 模式一共分为 3 种模式：系统最近点对更新模式，点撞墙模式和点序交换模式。

Stepto 按钮控制 Step 模式继续运行直到下一个断点；

Closest Change Radio 按钮用于控制系统采用“最近点对更新”断点模式，它表示当系统的最近点对发生变化时，系统暂停；

Hit Wall Radio 按钮用于控制系统采用“点撞墙”断点模式，它表示当前时刻有点撞到了墙上，系统暂停；

Order Change Radio 按钮用于控制系统采用“点序交换”断点模式，与此功能对应的响应还有用于标识当前与点序对应的 3 个轴的 Radio 按钮（Axis0、Axis1 和 Axis2）。当所有点在所指示的某个方向上位置发生交换时，系统暂停；

Axis1 Radio 按钮用于检测系统的点序更新事件是否发生在 0 度方向上，此时与视图 150~90 和视图 90～30 中每个点的竖直射线重合事件对应；

Axis2 Radio 按钮用于检测系统的点序更新事件是否发生在 60 度方向上，此时与视图 150~90 和视图 30～-30 中每个点的 150 度和-30 度射线重合事件对应；

Axis3 Radio 按钮用于检测系统的点序更新事件是否发生在 120 度方向上，此时与视图 90～30 和视图 30～-30 中每个点的 30 度射线重合事件对应；

All Pairs Checkbox 按钮用于控制是否打开主视图的所有点对连线的绘制模

式；

Certificates Checkbox 按钮用于控制是否打开主视图的所有点与 4 面墙的交线（即 kwall_cert 的示意）；

Number of All Events (Edit Box)中显示的是当前时刻在事件队列中存在的所有事件数（即时间队列的长度）；

Number of Events Per Second (Edit Box)中显示的是当前每秒处理的事件数；

Stop 按钮用于控制系统停止运行；

Quit 按钮用于控制退出系统；

注：在我们的测试环境下，默认模拟速度（1x 速）在 400 个点左右时每秒钟处理的事件数接近极限。如果点数更多则会出现严重停顿（由于算法是按实时的要求编写的，每一个计时器事件会要求算法更新到当前系统时间）。若需要测试 400 点以上的话需要降低模拟速度，调整 speed 滚动条，并尽量减少重力加速度，使得每秒处理的事件数降到 12000 以下才可达到平滑显示。另外在由于前面所分析的原因，在 300 点以上时数值计算误差发生的可能性会变得很大。当运行一段时间图像则可能出现杂乱线条。如果需要重新运行可单击 reset 按钮。

9 致谢

经过一学期的课程实践，我们在对动态数据结构算法理解和编程、测试过程中，学到了很多有用的知识，也得到了很多有益的启示。在此特别感谢邓老师在选题时的帮助和中期检查时提出的中肯建议。课堂上讲解的一些算法特别是扫描线算法等对我们编程时的设计也很有启发。另外非常感谢原作者 Julien Basch 和 Li Zhang 对程序结构的建议以及无私的提供一些 kds 样例。希望我们的实现能够对希望了解 kds 思想和动态最近邻点算法的人有所帮助。