Algorithm for windowing the axis-parallel rectangles

Sergei Bezrukov

1 Problem statement

A problem arising in the 3-dim visualization of geographic maps is to select those map cells that intersect the observer's frustum. In a trivial approach one just checks all the cells form a database for the intersection with the given frustum, which causes noticeable hang-ups of the visualization. We present a faster algorithm here.

We assume that the map cells are given by their axis-parallel bounding rectangles. This way we come to the following problem for a given set S of n axis-parallel rectangles

$$S = \{R_i\}_{i=1}^n = \{[l_i, r_i] \times [t_i, b_i],\ i = 1, \ldots, n\}$$

represented by their left/right horizontal coordinates (l_i, r_i) and their top/bottom vertical coordinates (t_i, b_i). We are also given a query axis-parallel rectangle $Q = [l, r] \times [t, b]$ representing the observer's frustum. The objective is to report those rectangles from S that have a non-empty intersection with Q.

2 Computing rectangles overlapping the query

To solve this problem efficiently we first preprocess the set S of rectangles and put them is a new data structure $\mathcal{S}$ that we call Rectangle Tree. The preprocessing is as follows:

1. Compute the median (average value) of all the x-coordinates of the corner points

$$x_{\text{mid}} = \frac{1}{2n} \sum_{i=1}^{n} (l_i + r_i).$$

2. Partition the set S into 3 groups:

   S_{mid} : rectangles that intersect the vertical line x_{mid}

   S_{left} : rectangles that are to the left from the vertical line

   S_{right} : rectangles that are to the right from the vertical line

   There groups will form the nodes of the Rectangle Tree $\mathcal{S}$. The node S_{mid} is the root of $\mathcal{S}$. The nodes S_{left} and S_{right} are its left and right childs, respectively, to be processed recursively.

Each node v of the Rectangles Tree $\mathcal{S}$ contains the following information:

- the vertical midpoint x-value $M(v)$
- pointers $Mptr(v)$ to the secondary Rectangle Tree $\mathcal{S}(v)$
- pointers $Lptr(v)$ and $Rptr(v)$ to the child nodes corresponding to the sets S_{left} and S_{right}.

Let us see how the Rectangles Tree $\mathcal{S}$ can be used for our search. First note, that a rectangle $R_i = [l_i, r_i] \times [t_i, b_i]$ intersects $Q = [l, r] \times [t, b]$ if and only if the segment $[l_i, r_i]$ intersects $[l, r]$ AND the segment $[t_i, b_i]$ intersects $[t, b]$. We distinguish the following complimentary cases.

Case 1. Suppose the vertical line x_{mid} intersects the segment $[l, r]$ of the query rectangle Q, that is $l \leq x_{\text{mid}} \leq r$.

In this case every rectangle $R_i \in S_{\text{mid}}$ intersects the horizontal range of Q and we must report those that intersect the vertical range of Q. For this we will do a search in the secondary structure. In this search there is no limitation of the horizontal range of the rectangles (so we pass to the search procedure the value $[-\infty, \infty]$ for that range), but their vertical range must intersect $[t, b]$. Also, we must process the rectangles from the sets L and R recursively.

Case 2. Suppose the query rectangle Q is completely to the left from the line x_{mid}, that is $r < x_{\text{mid}}$.

In this case we have to report those rectangles $R_i \in S_{\text{mid}}$ that intersect $[l, r]$ and $[t, b]$. Since the right edge of these rectangles is to the right of x_{mid}, we have to check that $l_i \leq r$. Since no rectangle from S_{right} intersects Q, we only need to process the set S_{left} recursively.

Case 3. Suppose the query rectangle Q is completely to the right from the line x_{mid}, that is $x_{\text{mid}} < l$.

In this case we have to report those rectangles $R_i \in S_{\text{mid}}$ that intersect $[l, r]$ and $[t, b]$. Since the left edge of these rectangles is to the left of x_{mid}, we have to check that $l \leq r_i$. Since no rectangle from S_{left} intersects Q, we only need to process the set S_{right} recursively.

This leads to the following recursive search procedure. We invoke it from the main program as `QueryRectTree(`Q`, root(`$\mathcal{S}$`))`.

`QueryRectTree`(Q, v)
Input: query rectangle Q and a node v in the Rectangle Tree
Output: rectangles from S that overlap with Q

if $v =$ nil **then return** /* termination condition for the recursion */
if $(l \leq M(v) \leq r)$ **then** /* Case 1 */
    `QuerySecondaryTree`$(Mptr(v), [-\infty, \infty], [t, b])$
    `QueryRectTree`$(Q, Lptr(v))$

```
    QueryRectTree(Q, Rptr(v))
else if (r < M(v)) then                          /* Case 2 */
    QuerySecondaryTree(Mptr(v), [−∞, r], [t, b])
    QueryRectTree(Q, Lptr(v))
else                                             /* Case 3 */
    QuerySecondaryTree(Mptr(v), [l, ∞], [t, b])
    QueryRectTree(Q, Rptr(v))
```

Note that the `QuerySecondaryTree` procedure is called for infinite horizontal ranges only ($l = -\infty$ or $r = \infty$ or both). This is a big deal and allows to decrease the storage requirements from $O(n \log n)$ down to $O(\log n)$.

The secondary Rectangle Tree $\mathcal{S}(v)$ is similar and stores the pointers to the rectangles from the set S_{mid}. We process them as follows:

1. Compute the median y_{mid} of all the y-coordinates of the corner points.

2. Partition the set S_{mid} into 3 groups

   S_{cent} : rectangles that intersect the horizontal line y_{mid}

   S_{bot} : rectangles that are below the horizontal line

   S_{top} : rectangles that are above the horizontal line

   These three groups will form the nodes of the secondary Rectangle Tree $\mathcal{S}(v)$. The node S_{cent} is the root vertex, while the nodes S_{bot} and S_{top} are its childs to be processed recursively.

3. The rectangles from S_{cent} are organized into four 2-dim dynamic arrays LT, LB, RT, and RB (for left/right and top/bottom). The array LB lists the rectangles from S_{cent} ordered first in increasing order of their l-values (as rows) and for each l-value the corresponding rectangles are ordered according to the increasing b-values (as columns). This provides a kind of lexicographic ordering of the rectangle corner points.

   The array LT lists the same rectangles (the ones belonging to the set S_{cent} ordered first in increasing order of their l-values (as rows) and for each l-value the corresponding rectangles are ordered according to the decreasing t-values (as columns).

   Similarly, the arrays RT and RB store the rectangles from S_{cent}, where the first ordering is done according to the decreasing r-values of the corresponding rectangles.

Therefore, each rectangle from the original set S is represented 4 times in our data structure, so its size is about $4n$. Note that each rectangle is actually stored in memory just once, we only use the pointers to the rectangle objects in the data structures. A twice larger storage for the rectangles compared to the known algorithm from [1] is the price for a faster search. However, in our approach there is no need to build additional data structures for line segments and points. We use just arrays to report the passing rectangles and this additionally speed up the search. Each node w in the secondary Rectangle Tree $\mathcal{S}(v)$ stores the following information:

- the horizonal midpoint y-coordinate $M(w)$
- pointers $LB(w)$, $LT(w)$, $RB(w)$, and $RT(w)$ to the lists LT, LB, RB, and RT.
- pointers $Bptr(w)$ and $Tptr(w)$ to the nodes corresponding to the sets S_{bot} and S_{top}, respectively.

The call `QuerySecondaryTree`$(w, [p, q], [t, b])$ can now be processed similarly to the one for the Rectangle Tree $\mathcal{S}$. We only consider the case of the left-open first range here for brevity, although the pseudocode below is free of this assumption. We split the analysis into the same 3 complimentary cases as for the primary tree

Case a. Suppose the line y_{mid} intersects the segment $[t, b]$ of the query rectangle Q, that is $t \leq y_{\text{mid}} \leq b$.

In this case every rectangle $R_i \in S_{\text{cent}}$ intersects the vertical range of Q and we must report them all in the order of increasing l-values to check all the time for the condition $l_i \leq l$. Also, we must process the rectangles from the sets B and T recursively.

Case b. Suppose the query rectangle Q is completely below the line y_{mid}, that is $t < y_{\text{mid}}$.

In this case we have to report those rectangles $R_i \in S_{\text{cent}}$ that satisfy the conditions $l_i \leq l$ and $b_i \leq t$. But since the rectangles from S_{cent} are double-ordered in B by increasing l-value and increasing b-values, we can just loop through them in order and stop when the above conditions are not satisfied. Since no rectangle from S_{top} intersects Q, we only need to process the set S_{bot} recursively.

Case c. Suppose the query rectangle Q is completely above the line y_{mid}, that is $y_{\text{mid}} < b$.

In this case we have to report those rectangles $R_i \in S_{\text{cent}}$ that satisfy the conditions $l_i \leq l$ and $b \leq t_i$. But since the rectangles from S_{cent} are double-ordered in T by increasing l-value and decreasing t-values, we can just loop through them in order and stop when the above conditions are not satisfied. Since no rectangle from S_{bot} intersects Q, we only need to process the set S_{top} recursively.

This leads to the following pseudocode, which we involve from the calling program as `QuerySecondaryTree`$(\text{root}(\mathcal{S}(v)), [p, q], [t, b])$.

`QuerySecondaryTree`$(w, [p, q], [t, b])$
Input: a node w in the secondary Rectangle Tree, (semi)-infinite horizontal range $[p, q]$, and a vertical range $[b, t]$
Output: set of rectangles with the l- (resp. r-values) in $[p, q]$ overlapping the range $[b, t]$

if $w = \text{nil}$ **then return** /* termination condition for the recursion */
if $(b \leq M(w) \leq t)$ **then** /* Case a */
    if $(p == -\infty)$ **then** /* first range is $[-\infty, q]$ */
        `ReportRectangles`$(LB(w), [p, q], [-\infty, \infty])$
    else if $(q == \infty)$ **then** /* first range is $[p, \infty]$ */
        `ReportRectangles`$(RB(w), [p, q], [-\infty, \infty])$
    `QuerySecondaryTree`$(Bptr(w), [p, q], [t, b])$

```
        QuerySecondaryTree(Tptr(w), [p, q], [t, b])
else if (t < M(w)) then                          /* Case b */
        if (p == −∞) then                        /* first range is [−∞, q] */
                ReportRectangles(LB(w), [p, q], [−∞, t])
        else if (q == ∞) then                    /* first range is [p, ∞] */
                ReportRectangles(RB(w), [p, q], [−∞, t])
        QuerySecondaryTree(Bptr(w), [p, q], [t, b])
else                                             /* Case c */
        if (p == −∞) then                        /* first range is [−∞, q] */
                ReportRectangles(LT(w), [p, q], [b, ∞])
        else if (q == ∞) then                    /* first range is [p, ∞] */
                ReportRectangles(RT(w), [p, q], [b, ∞])
        QuerySecondaryTree(Tptr(w), [p, q], [t, b])
```

Note that the procedure `ReportRectangles` is called only for (semi)-infinite ranges. This is one of the key points to speed up the entire search, since the rectangles are organized in a pre-sorted dynamic 2-dim arrays and we just need to loop through these arrays to report the rectangles. For example, the rectangles in the set $LB(w)$ are ordered in the 2-dim structure as follows. The first row in the structure stores all the rectangles with the minimum x-coordinate in the order of increasing y-coordinate. The next row stores all the rectangles with the next smallest value of the x-coordinate in the order of increasing y-coordinate. Thus, all the rectangles in one row have the same y-coordinate, and the rows are ordered in the increasing order of the y-coordinates.

`ReportRectangles`$(M, [p, q], [c, d])$
Input: set of ordered rectangles M and (semi)-infinite ranges $[p, q]$ and $[c, d]$
Output: rectangles from M with the l-values (resp. r-values) in $[p, q]$
and the b-values (resp. t-values) in $[c, d]$

```
if (p == −∞) then                                /* the first range is [−∞, q] */
        if (c == −∞) then                        /* the second range is [−∞, d] */
                for (i = 1; i < length(M) && M[i][0].l ≤ q; i++)
                        for (j = 1; j < length(M[i]) && M[i][j].b ≤ d; j++)
                                report rectangle M[i][j]
        else if (d == ∞) then                    /* the second range is [c, ∞] */
                for (i = 1; i < length(M) && M[i][0].l ≤ q; i++)
                        for (j = 1; j < length(M[i]) && M[i][j].t ≥ c; j++)
                                report rectangle M[i][j]
else if (q == ∞) then                            /* the first range is [p, ∞] */
        if (c == −∞) then                        /* the second range is [−∞, d] */
                for (i = 1; i < length(M) && M[i][0].r ≥ p; i++)
                        for (j = 1; j < length(M[i]) && M[i][j].b ≤ d; j++)
                                report rectangle M[i][j]
        else if (d == ∞) then                    /* the second range is [c, ∞] */
                for (i = 1; i < length(M) && M[i][0].r ≥ p; i++)
                        for (j = 1; j < length(M[i]) && M[i][j].t ≥ c; j++)
                                report rectangle M[i][j]
```

The involved Rectangle Tree can be constructed by the following algorithm:

`ConstructRectTree`(S)
Input: set of 2-dim rectangles S
Output: the root of the Rectangle Tree for S
1. **if** ($S = \emptyset$) **then return** null
2. Create a node v
3. Compute and store x_{mid} with v
4. Compute S_{mid}, S_{left}, S_{right}
5. $Mptr(v)$ = `ConstructSecondaryTree`(S_{mid})
6. $Lptr(v)$ = `ConstructRectTree`(S_{left})
7. $Rptr(v)$ = `ConstructRectTree`(S_{right})

The construction of the secondary tree follows the same pattern. Note that only the secondary tree actually stores pointers to the rectangles.

`ConstructSecondaryTree`(S)
Input: set of 2-dim rectangles S intersecting a straight line
Output: the root of the secondary Rectangle Tree for S
1. **if** ($S = \emptyset$) **then return** null
2. Create a node w
3. Compute and store y_{mid} with w
4. Compute S_{cent}, S_{bot}, S_{top}
5. Create four 2-dim arrays LT, LB, RT, and RB as follows:
   LB: order the rectangles of S_{cent} according to the increasing x and increasing y-coordinates of their bottom left corner points in the lex. order
   RB: order the rectangles of S_{cent} according to the decreasing x and increasing y-coordinates of their bottom right corner points in the lex. order
   LT: order the rectangles of S_{cent} according to the increasing x and decreasing y-coordinates of their top left corner points in the lex. order
   RT: order the rectangles of S_{cent} according to the decreasing x and decreasing y-coordinates of their top right corner points in the lex. order
   Store the pointers to these arrays in w.
6. $Bptr(v)$ = `ConstructSecondaryTree`(S_{bot})
7. $Tptr(v)$ = `ConstructSecondaryTree`(S_{top})

3 The algorithmic complexity of the method

Proposition 1 *The Rectangle Tree of n rectangles can be built in $O(n \log n)$ time.*

Proof.
Finding the median of a set of points takes a linear time. To sort the rectangles from S_{cent} takes $O(|S_{\text{cent}}| \cdot \log(|S_{\text{cent}}|))$ time. We need to do 4 different sorting of this set, but this all will take the same time up to the multiplicative constant. Since the sets S_{cent} are

disjoint, the entire processing time is $\sum_{S_{\text{cent}}} O(|S_{\text{cent}}| \cdot \log |S_{\text{cent}}|) = \sum_{S_{\text{cent}}} O(|S_{\text{cent}}| \log n) = O(\sum_{S_{\text{cent}}} |S_{\text{cent}}|) \log n = O(n \log n)$. □

Proposition 2 *The Rectangle Tree of n rectangles uses $O(n)$ storage.*

Proof.
Due to the splitting of the rectangles in the primary and the secondary trees, the usage of medians ensures that the number of rectangles in each of the left and right subtrees does not exceed n. Indeed, there are at most $p/2$ values that are below the median of p numbers. The same argument is also valid for the values exceeding the median. Applying this observation to $p = 2n$ (each rectangle constitutes 2 points in the computation of the mid-values), we come to the above conclusion.

Therefore, the depth of the primary search tree is $O(\log n)$. Hence, the number of nodes in this tree is $O(n)$, because each node stores just a constant (up to 5) values. The same holds for the secondary tree with n_i nodes: its space complexity of $O(n_i)$. Note that the secondary trees store disjoint sets of rectangles, so $\sum_i n_i = n$. Finally, a pointer to each rectangle is stored 4 times, in the 2-dim dynamic arrays, so the total space complexity is $\sum_i (O(n_i)) = O(\sum_i n_i) = O(n)$. □

Proposition 3 *Algorithm* `QueryRectTree` *reports the rectangles overlapping a given query in $O(\log^2 n + k)$ time, where k is the number of reported rectangles.*

Proof.
A rather complicated analysis presented in [1] shows that a search in the standard Interval Tree structure takes $O(\log n)$ time. The same argument can also be applied to our Rectangle Tree. So, the overall search time in the primary tree takes $O(\log n)$ time. For each node in the primary tree the save amount of time is taken for the search in every secondary subtree on the search path. Finally, since each rectangle is reporting just once, the total report time is $O(k)$. Taking this all into account, the total running time of the algorithm is $\sum_i O(\log n(O(\log n + k_i)) = O(\log^2 n + k)$, where k_i is the number of rectangles reported at the i-th call of the `ReportRectangles` procedure.

4 Implementation

4.1 Preliminaries

The algorithm is implemented in Java language, version 5. The implementation consists of 4 source files: `Wind.java`, `WindTest.java`, `SBrect.java`, `RectTree.java`, and `SecondaryTree.java`. Additionally, a manifest file `Wind.mf` is needed for creating the Java Archive JAR file for the compactness. This file is also needed to run the application over the network by using the JNLP (Java Network Launch Protocol) technology.

The first two files `Wind.java` and `WindTest.java` contain the Java API calls for creating the application graphics window and user interface. There are well commented and are not described here, because their irrelevance to the algorithm implementation.

The file `SBrect.java` described a class for representing the rectangles. Its implementation is straightforward:

```
public class SBrect
{
  public int l;    // left coordinate
  public int r;    // right coordinate
  public int b;    // bottom coordinate
  public int t;    // top coordinate

  public SBrect(int l, int r, int b, int t)
  {
    this.l = l;    // constructor of the SBrect object
    this.r = r;
    this.b = b;
    this.t = t;
  }

  public SBrect(SBrect r)   // copy constructor
  {
    this(r.l, r.r, r.b, r.t);
  }
}
```

4.2 Constructing and querying the Rectangle Tree

The file `RectTree.java` describes a node structure in the Rectangle Tree. Each node of this tree is made a leaf by the node constructor and contains the following information, according to the algorithm:

```
public class RectTree
{
  private double median;       // mid value of x-coordinates
  private SecondaryTree mPtr;  // pointer to the secondary tree
  private RectTree lPtr;       // pointers to the left and right
  private RectTree rPtr;       //   primary subtrees

  public void RectTree()       // default constructor
  {                            // information stored in each node
    median = 0;                // mid value of all x-coordinates
    mPtr = null;               // pointer to the secondary tree
    lPtr = rPtr = null;        // pointers to the left/right subtrees
  }
```

The method constructing the tree is pretty much the same as the pseudocode described above. It is assumed that the set of rectangles is given as a linked list of the `SBrect`

classes. The method is declared as `static` because it does not use the underlying class members. First, we compute the mid value of the x-coordinates of the rectangle vertical sides:

```
RectTree pT = new RectTree();           // create a new tree node
for (ListIterator<SBrect> lI=lList.listIterator(0); lI.hasNext(); )
{
  SBrect rect = lI.next();
  pT.median += rect.l + rect.r;
}
pT.median /= 2*lList.size();            // ... and put the median there
```

After that the set of all rectangles is partitioned in 3 disjoint classes (see the algorithm) represented in the linked lists:

```
LinkedList<SBrect> sMid = new LinkedList<SBrect>();
LinkedList<SBrect> sLeft = new LinkedList<SBrect>();
LinkedList<SBrect> sRight = new LinkedList<SBrect>();
for (ListIterator<SBrect> lI=lList.listIterator(0); lI.hasNext(); )
{
  SBrect rect = lI.next();
  if (rect.r < pT.median) sLeft.add(rect);
  else if (rect.l > pT.median) sRight.add(rect);
  else sMid.add(rect);
}
```

Finally, we apply the recursive calls to the constructed sets of rectangles:

```
// create the secondary tree for that node
pT.mPtr = SecondaryTree.constructSecondaryTree(sMid);
pT.lPtr = constructRectTree(sLeft);     // proceed recursively with
pT.rPtr = constructRectTree(sRight);    // the left and right subtrees
return(pT);
```

The Rectangle Tree querying algorithm is also practically identical to its pseudocode. It also does not use the underlying class members, hence is declared as `static` for the efficiency. To query the secondary tree, we send there the vertical and horizontal ranges combined for the convenience in one `SBrect` structure. The coordinates of all the involved points in our application are non-negative integers, so the predefined Java values `Integer.MIN_VALUE` and `Integer.MAX_VALUE` serve as $-\infty$ and ∞, respectively.

```
public static void queryRectTree(SBrect query, RectTree v)
{
  if (v == null) return;             // terminating condition
```

```
    SBrect range = new SBrect(query);  // search range for the secondary tree
    if (query.l <= v.median && v.median <= query.r)  // Case 1
    {
      range.l = Integer.MIN_VALUE;
      range.r = Integer.MAX_VALUE;
      SecondaryTree.querySecondaryTree(v.mPtr, range);
      queryRectTree(query, v.lPtr);
      queryRectTree(query, v.rPtr);
    }
    else if (query.r < v.median)       // Case 2
    {
      range.l = Integer.MIN_VALUE;
      SecondaryTree.querySecondaryTree(v.mPtr, range);
      queryRectTree(query, v.lPtr);
    }
    else                               // Case 3
    {
      range.r = Integer.MAX_VALUE;
      SecondaryTree.querySecondaryTree(v.mPtr, range);
      queryRectTree(query, v.rPtr);
    }
  }
```

4.3 Constructing and querying the secondary tree

Each node of the secondary tree stores the information shown below and is constructed by the following constructor:

```
public class SecondaryTree
{
  private double median;          // mid value of y-coordinates
  private SBrect[][] lb, lt, rb, rt; // pointers the rect. lists
  private SecondaryTree bPtr;     // pointers to the bottom and top
  private SecondaryTree tPtr;     //   secondary subtrees

  public void SecondaryTree()                // constructor
  {
    median = 0.0;                            // mid value of the y-coords.
    lb = lt = rb = rt = null;                // pointers to 2-dim arrays
    bPtr = tPtr = null;                      // pointers to subtrees
  }
```

This class also contains four comparators to be used for sorting. We apply the standard Java API sorting algorithm for arrays that implements the merge-sort algorithm. The comparators are needed for the sorting method. We show only one of them here, they all

are similar and appear as inner Java classes implementing the Comparator interface of Java API:

```
private static Comparator<SBrect> ltComparator = new Comparator<SBrect>()
{
  public int compare(SBrect r1, SBrect r2)  // sort the rectangles
  {                                         // in increasing order of their
    if (r1.l < r2.l) return(-1);            // l-values and decreasing order
    else if (r1.l > r2.l) return(1);        // of the t-values
    else return(r2.t - r1.t);
  }
};
```

The constructor of the secondary tree starts similarly as for the Rectangle Tree:

```
public static SecondaryTree constructSecondaryTree(LinkedList<SBrect> lList)
{
  if (lList == null || lList.size() == 0)   // terminating condition
    return(null);  // empty list

  SecondaryTree sT = new SecondaryTree();   // create a new node
  for (ListIterator<SBrect> lI=lList.listIterator(0); lI.hasNext(); )
  {
    SBrect rect = lI.next();
    sT.median += rect.b + rect.t;
  }
  sT.median /= 2*lList.size();             //.. and put the median there

  // split the rectangles in 3 disjoint sets
  LinkedList<SBrect> sCent = new LinkedList<SBrect>();
  LinkedList<SBrect> sBot = new LinkedList<SBrect>();
  LinkedList<SBrect> sTop = new LinkedList<SBrect>();
  for (ListIterator<SBrect> lI=lList.listIterator(0); lI.hasNext(); )
  {
    SBrect rect = lI.next();
    if (rect.t < sT.median) sBot.add(rect);
    else if (rect.b > sT.median) sTop.add(rect);
    else sCent.add(rect);
  }
```

In the next code segment we create the dynamic 2-dim arrays (see the algorithm description). For this we need to sort the Java linked list. It turns out that the implementation of the Java sort method first converts the linked list into an array, sorts it and then converts it back to the linked list. To speed up the processing, we first convert the linked list into the array just once and use the Java sorting methods available for the arrays. We will later need to convert the sorted structures into a 2-dim arrays, so it does not matter how to represent them:

```
  if (sCent.size() > 0)                    // create 2-dim structures
  {                                        // and attach then to the new node
    SBrect[] sCentArray = sCent.toArray(new SBrect[sCent.size()]);

    Arrays.sort(sCentArray, lbComparator);
    sT.lb = create2DimArrayL(sCentArray);
    Arrays.sort(sCentArray, ltComparator);
    sT.lt = create2DimArrayL(sCentArray);
    Arrays.sort(sCentArray, rbComparator);
    sT.rb = create2DimArrayR(sCentArray);
    Arrays.sort(sCentArray, rtComparator);
    sT.rt = create2DimArrayR(sCentArray);
  }
```

We use two procedures for creating the 2-dim dynamic arrays created from the corresponding presorted 1-dim arrays. The procedures are practically identical and differ just in three lines. One of them works with the left coordinates of the rectangles, and the second one with the right ones. Here is one of them:

```
  private static SBrect[][] create2DimArrayL(SBrect[] arr)
  {
    LinkedList<SBrect[]> rows = new LinkedList<SBrect[]>();
    LinkedList<SBrect> row = null;
    int x = Integer.MIN_VALUE;     // here is the difference
    for (int i=0; i<arr.length; i++)
    {
      SBrect rect = arr[i];
      if (x != arr[i].l)           // ... here
      {
        x = rect.l;                // ... and here
        if (row != null)
          rows.add(row.toArray(new SBrect[row.size()]));
        row = new LinkedList<SBrect>();
      }
      row.add(rect);
    }
    if (row != null && row.size() > 0)      // add the last row
      rows.add(row.toArray(new SBrect[row.size()]));
    return(rows.toArray(new SBrect[rows.size()][]));
  }
```

The rows in the created 2-dim array store all the rectangles that have the same x-coordinate (left or right, respectively). We use the initial value for $x = \pm\infty$ to construct the first row of the array.

Finally, the secondary tree creating procedure end up with two recursive calls for creating the left and right subtrees:

```
sT.bPtr = constructSecondaryTree(sBot); // proceed recursively with the
sT.tPtr = constructSecondaryTree(sTop); // left and right subtrees
return(sT);
```

The involved methods for querying the secondary structure and reporting the rectangles are practically identical to their pseudocodes in the algorithm description and are omitted here. The only minor difference is that we also represent two ranges in the `reportRectangles` procedure as a single `SBrect` structure. We also use the Java values `Integer.MIN_VALUE` and `Integer.MAX_VALUE` for $-\infty$ and ∞, respectively.

The `reportRectangles` procedure creates a linked list of rectangles overlapping the query rectangle. This linked list `overlap` is declared in class `WindTest` and has a package access there.

5 Profiling the algorithm

Recall, that by the trivial algorithm we mean a linear-time algorithm that checks each rectangle in sequence for the intersection with the query rectangle. In order to compute the actual running time of our algorithm and compare it with the trivial algorithm, we performed a series of 1000 tests. The number 1000 can be changed by the user.

During these tests we generated a random placement of n rectangles, $1,000 \leq n \leq 1,000,000$ which was fixed in all the tests. Then we place randomly the query rectangle and compute the total running time. The total elapsed time was normalized by dividing it by the number of experiments (1000). We used the following code:

```
int n = getNumber(tf1);                  // # of rectanges to generate
int m = getNumber(tf2);                  // # of tests to perform
if (n<0 || m<0) return;
drawPanel.generateRects(n);
algoTime.setText("");
trivTime.setText("");
overlapCnt.setText("");

StopWatch sWatch = new StopWatch();      // profiling the algorithm
drawPanel.randGen.setSeed(1000);
sWatch.start();
double cntr = 0;
for (int i=0; i<m; i++)
{
  drawPanel.moveQuery();                 // new query position
  drawPanel.computeOverlap();            // run the algorithm
  cntr += WindTest.overlap.size();
}
sWatch.stop();
algoTime.setText("Algo: " + 1.0*sWatch.getElapsedTime()/m + " ms");
```

A similar code was used to profile the trivial algorithm. To make both algorithms to work with the same set of queries, we set up the seed for the random numbers generator, responsible for placement the random query rectangles. This way the sets of queries for both methods are the same.

```
sWatch.reset();
drawPanel.randGen.setSeed(1000);
sWatch.start();
for (int i=0; i<m; i++)                    // profiling the trivial alg.
{
  drawPanel.moveQuery();
  drawPanel.computeTrivialOverlap();
}
sWatch.stop();
trivTime.setText("Triv: " + 1.0*sWatch.getElapsedTime()/m + " ms");
overlapCnt.setText("Overlap: " + (long)(cntr/m));
```

The results of the running time (in msec) are shown in the following table:

# of rect.	1000	5000	10,000	50,000	100,000	500,000	1,000,000
Our alg:	0.01	0.04	0.080	0.370	0.701	3.686	5.959
Trivial:	0.05	0.27	1.322	6.209	10.436	35.993	73.719
Overlap:	18	91	182	908	1815	9090	17666

Table 1: The running time (msec) and number of overlapping rectangles

It follows from this table, that our algorithm is about 10-20 times faster than the trivial one. The difference becomes more noticeable as n grows.

To perform the test for $n = 10^6$, we ran the Java application form the command line:
`java -Xmx512M -jar Wind.jar`
The reason for this is that it needs more memory than the default heap size provided by the settings of JVM of the JNLP technology.

Note that our algorithm is very stable in terms of running time. Repeating the series of tests returns almost the same average running time. The difference is minor and usually affects the second or the third sign after the decimal comma. In contrast to this, the running time of the trivial algorithm can differ up to a factor of 2. This is explained by the way how the trivial algorithm computes the intersection of rectangles. It checks up to 4 conditions per rectangle. So, sometimes 1 or checks are done to reject a rectangle. The computations in our algorithm are more uniform and are less affected by the actual set of rectangles.

References

[1] M. de Berg, M. van Krefeld, M. Overmars, O. Schwarzkopf *Computational Geometry: Algorithms and Applications*, Springer-Verlag 1997.