Intersections of a Set of Segments

by Dan Sunday

Sometimes an application needs to find the set of intersection points for a collection of many line segments. Often these applications involve polygons which are just an ordered set of connected segments. Specific problems that might need an algorithmic solution are:

1. Compute the intersection (or union, or difference) of two simple polygons or planar graphs. To do this, one must determine all intersection points, and use them as new vertices to construct the intersection (or union, or difference).

2. Test if two polygons or planar graphs intersect. One has to determine the intersections of one object's edges with those of the other. As soon as any valid intersection is found, the test can stop, and it doesn't have to determine the complete set of intersections.

3. Test if a polyline or polygon is simple. That is, determine if any two nonsequential edges of a polyline intersect. This is an important property since many algorithms only work for simple polylines or polygons. Again, his test can stop as soon as any intersection is found.

4. Decompose a polygon into simple pieces. To do this, one needs to know the complete set of intersection points between the edges, and use each of them as a cut-point in the decomposition.

Algorithms solving these problems are used in many application areas such as computer graphics, CAD, circuit design, hidden line elimination, computer vision, and so on.

A Short Survey of Intersection Algorithms

In general, for a set of n line segments, there can be up to $O(n^2)$ intersection points, since if every segment intersected every other segment, there would be $n(n-1)/2 = O(n^2)$ intersection points. In the worst case, to compute them all would require a $O(n^2)$ algorithm. The "brute force" algorithm would simply consider all $O(n^2)$ pairs of line segments, test each pair for intersection, and record the ones it finds. This is a lot of computing. However, when there are only a few intersection points, or only one such point needs to be detected (or not), there are faster algorithms.

In fact, these problems can be solved by "output-sensitive" algorithms whose efficiency depends on both the input and the output sizes. Here the input is a set Ω of n segments, and the output is the set Λ of k computed intersections, where $k = n^2$ in the worst case, but is usually much smaller. An early algorithm [Shamos & Hoey, 1976] showed how to detect if at least one intersection exists in $O(n \log n)$ time and $O(n)$ space by "sweeping" over a linear ordering of Ω. Extending their idea, [Bentley & Ottmann, 1979] gave an algorithm to compute all k intersections in $O((n+k)\log n)$ time and $O(n+k)$ space. After more than 30 years, the well-known "***Bentley-Ottmann Algorithm***" is still the most popular one to implement in practice ([Bartuschka, Mehlhorn & Naher, 1997], [de Berg et al, 2000], [Hobby, 1999], [O'Rourke, 1998], [Preparata & Shamos, 1985]) since it is relatively easy to both understand and implement. However, their algorithm did not achieve the theoretical lower bound; and thus, was only the first of many output-sensitive algorithms for solving the segment intersection problem.

A decade later, [Chazelle & Edelsbrunner, 1988 and 1992] discovered an optimal $O(n \log n + k)$ time algorithm. But, their algorithm still needs $O(n+k)$ storage space, and it is difficult to implement. Subsequent work made further improvements, and [Balaban, 1995] found an $O(n \log n + k)$ time and $O(n)$ space deterministic algorithm. There have also been a number of "randomized" algorithms with *expected* $O(n \log n + k)$ running time. The earliest of these by [Myers, 1985] uses $O(n+k)$ space. However, the later one by [Clarkson & Shor, 1989] uses only $O(n)$ space.

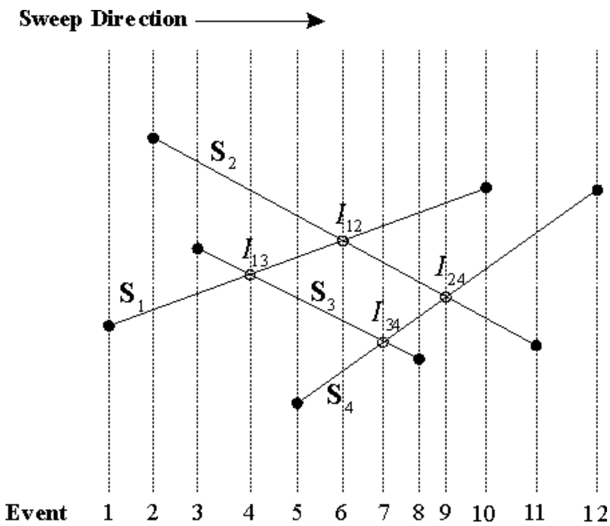
Additionally, improved algorithms have been found for the more restrictive "red-blue intersection" problem. Here there are two separate sets of segments, the "red" set Ω_1 and the "blue" set Ω_2. One wants to find intersections between the sets, but not within the same set; that is, red-blue intersections, but not red-red or blue-blue ones. A simple deterministic $O(n \log n + k)$ time and $O(n)$ space "trapezoid sweep" algorithm was developed by [Chan, 1994] based on earlier work of [Mairson & Stolfi, 1988]. These algorithms can be used to perform boolean set operations, like intersections or unions, between two different simple polygons or planar subdivision graphs.

Nevertheless, the Shamos-Hoey and Bentley-Ottmann algorithms remain the landmarks of the field. Note, however, that when *k* is large of order $O(n^2)$, the Bentley-Ottmann algorithm takes $O(n^2 \log n)$ time which is worse than the $O(n^2)$ brute-force algorithm! Also, the more complicated optimal algorithms are $O(n^2)$ which is the same as the simple brute-force one. So, when *k* is expected to be much larger than $O(n)$, one might as well use the easy-to-implement brute-force algorithm. But, when *k* is expected to be less than or equal to $O(n)$, Bentley-Ottmann is the simplest expected $O(n \log n)$ time and $O(n)$ space algorithm.

The Bentley-Ottmann Algorithm

The input for the Bentley-Ottmann algorithm is a collection $\Omega = \{\mathbf{L}_i\}$ of line segments $\mathbf{L}_i$, and its output will be a set $\Lambda = \{I_j\}$ of intersection points. This algorithm is referred to as a "***sweep line algorithm***" because it's operation can be visualized as having another line, a "sweep line" **SL,** sweeping over the collection Ω and collecting information as it passes over the individual segments $\mathbf{L}_i$. The information collected for each position of **SL** is basically an ordered list of all segments in Ω that are currently being intersected by **SL**. The data structure maintaining this information is often also called the "sweep line". This class structure also detects and outputs intersections as it discovers them. The process by which it discovers intersections is the heart of the algorithm and its efficiency.

To implement the sweep logic, we must first linearly order the segments of Ω to determine the sequence in which **SL** encounters them. That is, we need to order the endpoints $\{E_{i0}, E_{i1}\}_{i=1,n}$ of all the segments $\mathbf{L}_i$ so we can detect when **SL** starts and stops intersecting each segment of Ω. Traditionally, the endpoints are ordered by increasing *x* first and then increasing *y*-coordinate values, but any linear order will do (some authors prefer decreasing *y* first and then increasing *x*). With the traditional ordering, the sweep line is vertical and moves from left to right as it encounters each segment, as shown in the diagram:



At any point in the algorithm, the sweep line **SL** intersects only those segments with one endpoint to the left of (or on) it and the other endpoint to the right of it. The **SL** data structure keeps a dynamic list of these segments by: (1) adding a segment when its leftmost endpoint is encountered, and (2) deleting a segment when its rightmost endpoint is encountered. Further, the **SL** orders the list of segments with an "above-below" relation. So, to add or delete a segment, its position in the list must be determined, which can be done by a worst-case $O(\log n)$ binary search of the current segments in the list. In addition, besides adding or deleting segments, there is another event that changes the list structure; namely, whenever two segments intersect, then their positions in the ordered list must be swapped. Given the two segments, which must be neighbors in the list, this swap is an $O(\log n)$ operation.

To organize all this, the algorithm maintains an ordered "***event queue***" **EQ** whose elements cause a change in the **SL** segment list. Initially, **EQ** is set to the sweep-ordered list of all segment endpoints. But as intersections between segments are found, then they are also added to **EQ** in the same sweep-order as used for the endpoints One must test, though, to avoid inserting duplicate intersections onto the event queue. The example in the above diagram shows how this can happen. At event 2, segments $\mathbf{S}_1$ and $\mathbf{S}_2$ cause intersection I_{12} to be computed and put on the queue. Then, at event 3, segment $\mathbf{S}_3$ comes between and separates $\mathbf{S}_1$ and $\mathbf{S}_2$. Next, at event 4, $\mathbf{S}_1$ and $\mathbf{S}_3$ swap places on the sweep line, and $\mathbf{S}_1$ is brought next to $\mathbf{S}_2$ again causing I_{12} to be computed again. But, there can only be one event for each intersection, and I_{12} cannot be put on the queue twice. So, when an intersection is being put on the queue, we must find its potential x-sorted location in the queue, and check that it is not already there. Since there is at most one intersect point for any two segments, labeling an intersection with identifiers for the segments is sufficient to uniquely identify it. As a result of all this, the maximum size of the event queue = $2n+k \leq 2n+n^2$, and any insertion or deletion can be done with a $O(\log(2n+n^2)) = O(\log n)$ binary search.

But, what does all this have to do with efficiently finding the complete set of segment intersections? Well, as segments are sequentially added to the **SL** segment list, their possible intersections with other eligible segments are determined. When a valid intersection is found, then it is inserted into the event queue. Further, when an intersection-event on **EQ** is processed during the sweep, then it causes a re-ordering of the **SL** list, and the intersection is also added to the output list Λ. In the end, when all events have been processed, Λ will contain the complete ordered set of all intersections.

However, there is one critical detail, the heart of the algorithm, that we still need to describe; namely, how does one compute a valid intersection? Clearly, two segments can only intersect if they occur simultaneously on the sweep-line at some time. But this by itself is not enough to make the algorithm efficient. The important observation is that two intersecting segments must be immediate above-below neighbors on the sweep-line. Thus, there are only a few restricted cases for which possible intersections need to be computed:

1. When a segment is added to the **SL** list, determine if it intersects with its above and below neighbors.
2. When a segment is deleted from the **SL** list, its previous above and below neighbors are brought together as new neighbors. So, their possible intersection needs to be determined.
3. At an intersection event, two segments switch positions in the **SL** list, and their intersection with their new neighbors (one for each) must be determined.

This means that for the processing of any one event (endpoint or intersection) of **EQ**, there are at most two intersection determinations that need to be made.

One detail remains, namely the time needed to add, find, swap, and remove segments from the **SL** structure. To do this, the **SL** can be implemented as a balanced binary tree (such as an AVL, a 2-3, or a red-black tree) which guarantees that these operations will take at most $O(\log n)$ time since n is the maximum size of the **SL** list. Thus, each of the ($2n$+k) events has at worst $O(\log n)$ processing to do. Adding up the initial sort and the event processing, the efficiency of the algorithm is: $O(n \log n)+O((2n+k)\log n)=O((n+k)\log n)$.

Pseudo-Code: Bentley-Ottmann Algorithm

Putting all of this together, the top-level logic for an implementation of the Bentley-Ottmann algorithm is given by the following pseudo-code:

```
Initialize event queue EQ = all segment endpoints;
Sort EQ by increasing x and y;
Initialize sweep line SL to be empty;
Initialize output intersection list IL to be empty;

While (EQ is nonempty) {
    Let E = the next event from EQ;
    If (E is a left endpoint) {
        Let segE = E's segment;
        Add segE to SL;
        Let segA = the segment Above segE in SL;
        Let segB = the segment Below segE in SL;
        If (I = Intersect( segE with segA) exists)
            Insert I into EQ;
        If (I = Intersect( segE with segB) exists)
            Insert I into EQ;
    }
    Else If (E is a right endpoint) {
        Let segE = E's segment;
        Let segA = the segment Above segE in SL;
        Let segB = the segment Below segE in SL;
        Delete segE from SL;
        If (I = Intersect( segA with segB) exists)
            If (I is not in EQ already)
                Insert I into EQ;
    }
    Else {  // E is an intersection event
        Add E’s intersect point to the output list IL;
        Let segE1 above segE2 be E's intersecting segments in SL;
        Swap their positions so that segE2 is now above segE1;
        Let segA = the segment above segE2 in SL;
        Let segB = the segment below segE1 in SL;
        If (I = Intersect(segE2 with segA) exists)
            If (I is not in EQ already)
                Insert I into EQ;
        If (I = Intersect(segE1 with segB) exists)
```

```
                If (I is not in EQ already)
                    Insert I into EQ;
            }
            remove E from EQ;
        }
        return IL;
    }
```

This routine outputs the complete ordered list of all intersection points.

The Shamos-Hoey Algorithm

If one only wants to know if an intersection exists, then as soon as any intersection is detected, the routine can terminate immediately. This results in a greatly simplified algorithm. Intersections don't ever have to be put on the event queue, and so its size is only $2n$ for the endpoints of all the segments. And, code for processing this non-existent event can be removed. Further, the event (priority) queue can be implemented as a simple ordered array since it never changes. Additionally, no output list needs to be built since the algorithm terminates as soon as any intersection is found. Consequently, this algorithm needs only $O(n)$ space and runs in $O(n \log n)$ time. This is the original algorithm of [Shamos & Hoey, 1976].

Pseudo-Code: Shamos-Hoey Algorithm

The simplified pseudo-code is:

```
    Initialize event queue EQ = all segment endpoints;
    Sort EQ by increasing x and y;
    Initialize sweep line SL to be empty;

    While (EQ is nonempty) {
        Let E = the next event from EQ;
        If (E is a left endpoint) {
            Let segE = E's segment;
            Add segE to SL;
            Let segA = the segment Above segE in SL;
            Let segB = the segment Below segE in SL;
            If (I = Intersect( segE with segA) exists)
                return TRUE;   // an Intersect Exists
            If (I = Intersect( segE with segB) exists)
                return TRUE;   // an Intersect Exists
        }
        Else {  // E is a right endpoint
            Let segE = E's segment;
            Let segA = the segment above segE in SL;
            Let segB = the segment below segE in SL;
            Delete segE from SL;
            If (I = Intersect( segA with segB) exists)
                return TRUE;   // an Intersect Exists
        }
        remove E from EQ;
    }
    return FALSE;      // No  Intersections
}
```

Applications

Simple Polygons

(A) **Test if Simple**. The Shamos-Hoey algorithm can be used to ***test if a polygon is simple or not***. We give a C++ implementation simple_Polygon() for this algorithm below. Note that the shared endpoint between sequential edges does not count as a non-simple intersection point, and the intersection test routine must check for that.

Also, we do not provide the code for a balanced binary tree, which is needed for the sweepline data structure. We did this to concentrate only on the geometric aspect of the algorithm, and not recommend a specific type of balanced tree, such as AVL, or red-black, or 2-3. These can be found in standard libraries. Nevertheless, there have often been requests to include a complete standalone algorithm.

Recently, a complete simple_Polygon() algorithm using an AVL tree has been developed by [Glenn Burkhardt, 2014], and provided to us for publication here. He used AVL tree code previously developed by [Brad Appleton, 1997], who gave his permission to republish his code here (including his License agreement). In addition to integrating these two software packages, Glenn also made modifications for greater efficiency, and to make the code compatible with the current C++ standards.

(B) **Decompose into Simple Pieces**. The Bentley-Ottmann algorithm can be used to ***decompose a non-simple polygon into simple pieces***. To do this, all intersection points are needed. One approach for a simple decomposition algorithm is to perform "surgery" at each intersection point. This proceedure can be incorporated into the sweepline algorithm as it discovers new intersection points, resulting in an $O(n \log n + k)$ decomposition algorithm. To do this, one must also output the edges that persist in a linked list, whose connected sublists will be the new simple polygons.

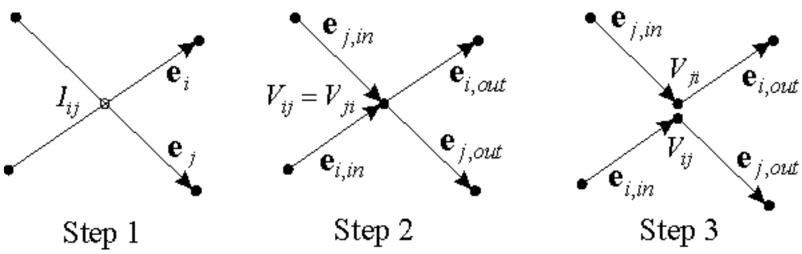
Let the original polygon be given by the set of segment endpoints $\mathbf{\Omega} = \{E_i\}_{i=0,n}$, with $E_n = E_0$. Its directed edges are given by the vectors $\mathbf{e}_i = E_{i+1} - E_i$.

1. Compute all the intersection points of the edge segments using the Bentley-Ottmann algorithm. The following steps may be incorporated into this algorithm whenever the sweepline finds a new intersection.

2. For an intersection point I_{ij} between $\mathbf{e}_i$ and $\mathbf{e}_j$, ($j > i+1 > 0$), add 2 new vertices V_{ij} and V_{ji} (one on each edge $\mathbf{e}_i$ and $\mathbf{e}_j$). Split each edge $\mathbf{e}_k$ into two new edges $\mathbf{e}_{k,in}$ and $\mathbf{e}_{k,out}$ joined at the new vertex on $\mathbf{e}_k$.

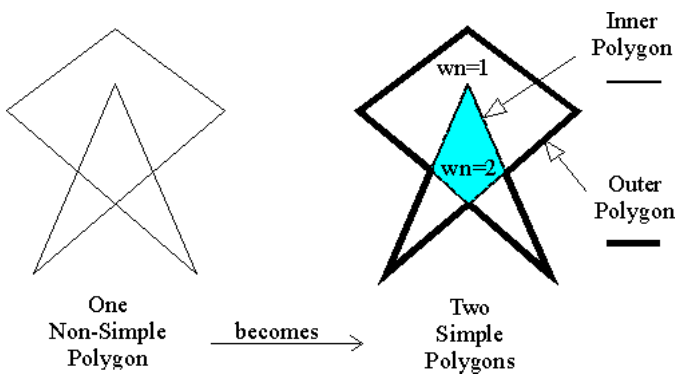
   Add these new edges to the sweepline list, and also record them as new polygon edges. Also, reassign any other intersections that $\mathbf{e}_i$ and $\mathbf{e}_j$ may have had (with other edges) to the new *in* & *out* edges. When implemented within the sweepline algorithm, this reassignment is made to the rightmost new edge, and the leftmost new edge gets deleted from the sweepline list.

3. Next, do surgery at these new vertices to remove the crossover. This is done by
   a) attaching $\mathbf{e}_{i,in}$ to $\mathbf{e}_{j,out}$ at V_{ij}, and
   b) attaching $\mathbf{e}_{j,in}$ to $\mathbf{e}_{i,out}$ at V_{ji},
   as shown in the following diagram.



4. After doing this at all intersections, then the remaining connected edge sets are the simple polygons decomposing the original non-simple one.

Note that the resulting simple polygons may not be disjoint since one could be contained inside another. In fact, the decomposition inclusion hierarchy is based on the inclusion winding number of each simple polygon in the original non-simple one (see Algorithm 3 about Winding Number Inclusion). For example:



Polygon Set Operations

The Bentley-Ottmann algorithm can be used to speed up computing the intersection, union, or difference of two general non-convex simple polygons. Of course, before using any complicated algorithm to perform these operations, one should first test the bounding boxes or spheres of the polygons for overlap (see Algorithm 8 on Bounding Containers). If the bounding containers are disjoint, then so are the two polygons, and the set operations become trivial.

However, when two polygons overlap, the sweep line strategy of the Bentley-Ottmann algorithm can be adapted to perform a set operation on any two simple polygons. For further details see [O'Rourke, 1998, 266-269]. If the two polygons are known to be simple, then one just needs intersections for segments from different polygons, which is a red-blue intersection problem.

Planar Subdivisions

The Bentley-Ottmann algorithm can be used to efficiently compute the overlay of two planar subdivisions. For details, see [de Berg et al, 2000, 33-39]. A planar subdivision is a planar graph with straight line segments for edges, and it divides the plane into a finite number of regions. For example, boundary lines divide a country into states. When two such planar graphs are overlaid (or superimposed), then their combined graph defines a subdivision refinement of each one. To compute this refinement, one needs to calculate all intersections between the line segments in both graphs. For a segment in one graph, we only need the intersections with segments in the other graph, and so this is another red-blue intersection problem.

Implementations

Here are some sample "C++" implementations of these algorithms.

```
// Copyright 2001 softSurfer, 2012 Dan Sunday
// This code may be freely used and modified for any purpose
// providing that this copyright notice is included with it.
// SoftSurfer makes no warranty for this code, and cannot be held
// liable for any real or imagined damage resulting from its use.
// Users of this code must verify correctness for their application.


// Assume that classes are already given for the objects:
//    Point with 2D coordinates {float x, y;}
//    Polygon with n vertices {int n; Point *V;} with V[n]=V[0]
//    Tnode is a node element structure for a BBT
//    BBT is a class for a Balanced Binary Tree
//        such as an AVL, a 2-3, or a  red-black tree
//        with methods given by the  placeholder code:

typedef struct _BBTnode Tnode;
struct _BBTnode {
        void* val;
        // plus node mgmt info ...
};

class BBT {
        Tnode   *root;
public:
                 BBT() {root = (Tnode*)0;}   // constructor
                 ~BBT() {freetree();}        // destructor

        Tnode*  insert( void* ){};       // insert data into the tree
        Tnode*  find( void* ){};         // find data from the tree
        Tnode*  next( Tnode* ){};        // get next tree node
        Tnode*  prev( Tnode* ){};        // get previous tree node
        void    remove( Tnode*  ){};     // remove node from the tree
        void    freetree(){};            // free all tree data structs
};
// NOTE:
// Code for these methods must be provided for the algorithm to work.
// We have not provided it since binary tree algorithms are well-known
// and code is widely available. Further, we want to reduce the clutter
// accompanying the essential sweep line algorithm.

/**
Recently, a complete simple_Polygon() algorithm using an AVL tree has been developed by
[Glenn Burkhardt, 2014], and provided to us for publication here. He integrated our code
with AVL tree code previously developed by [Brad Appleton, 1997], and modified both to
improve the algorithm’s efficiency.
 **/
//===================================================================


#define FALSE   0
#define TRUE    1
#define LEFT    0
#define RIGHT   1

extern void
qsort(void*, unsigned, unsigned, int(*)(const void*,const void*));

// xyorder(): determines the xy lexicographical order of two points
//      returns: (+1) if p1 > p2; (-1) if p1 < p2; and  0 if equal
int xyorder( Point* p1, Point* p2 )
{
    // test the x-coord first
    if (p1->x > p2->x) return 1;
    if (p1->x < p2->x) return (-1);
    // and test the y-coord second
    if (p1->y > p2->y) return 1;
    if (p1->y < p2->y) return (-1);
    // when you exclude all other possibilities, what remains  is...
    return 0;  // they are the same point
}

// isLeft(): tests if point P2 is Left|On|Right of the line P0 to P1.
//      returns: >0 for left, 0 for on, and <0 for  right of the line.
```

```
//      (see Algorithm 1 on Area of Triangles)
inline float
isLeft( Point P0, Point P1, Point P2 )
{
    return (P1.x - P0.x)*(P2.y - P0.y) - (P2.x - P0.x)*(P1.y -  P0.y);
}
//===================================================================



// EventQueue Class

// Event element data struct
typedef struct _event Event;
struct _event {
    int      edge;          // polygon edge i is V[i] to V[i+1]
    int      type;          // event type: LEFT or RIGHT vertex
    Point*   eV;            // event vertex
};

int E_compare( const void* v1, const void* v2 ) // qsort compare two events
{
    Event**    pe1 = (Event**)v1;
    Event**    pe2 = (Event**)v2;

    return xyorder( (*pe1)->eV, (*pe2)->eV );
}

// the EventQueue is a presorted array (no insertions needed)
class EventQueue {
    int      ne;                // total number of events in array
    int      ix;                // index of next event on queue
    Event*   Edata;             // array of all events
    Event**  Eq;                // sorted list of event pointers
public:
              EventQueue(Polygon P);     // constructor
             ~EventQueue(void)           // destructor
                 { delete[] Eq; delete[] Edata;}

    Event*   next();                     // next event on queue
};

// EventQueue Routines
EventQueue::EventQueue( Polygon P )
{
    ix = 0;
    ne = 2 * P.n;           // 2 vertex events for each edge
    Edata = (Event*)new Event[ne];
    Eq = (Event**)new (Event*)[ne];
    for (int i=0; i < ne; i++)           // init Eq array pointers
        Eq[i] = &Edata[i];

    // Initialize event queue with edge segment endpoints
    for (int i=0; i < P.n; i++) {        // init data for edge i
        Eq[2*i]->edge = i;
        Eq[2*i+1]->edge = i;
        Eq[2*i]->eV   = &(P.V[i]);
        Eq[2*i+1]->eV = &(P.V[i+1]);
        if (xyorder( &P.V[i], &P.V[i+1]) < 0)  { // determine type
            Eq[2*i]->type    = LEFT;
             Eq[2*i+1]->type = RIGHT;
        }
        else {
            Eq[2*i]->type    = RIGHT;
             Eq[2*i+1]->type = LEFT;
        }
    }
    // Sort Eq[] by increasing x and y
    qsort( Eq, ne, sizeof(Event*), E_compare );
}

Event* EventQueue::next()
{
    if (ix >= ne)
        return (Event*)0;
    else
        return Eq[ix++];
}
//===================================================================



// SweepLine Class

// SweepLine segment data struct
typedef struct _SL_segment SLseg;
struct _SL_segment {
    int      edge;          // polygon edge i is V[i] to V[i+1]
    Point    lP;            // leftmost vertex point
    Point    rP;            // rightmost vertex point
    SLseg*   above;         // segment above this one
    SLseg*   below;         // segment below this one
};

// the Sweep Line itself
class SweepLine {
    int      nv;            // number of vertices in polygon
    Polygon* Pn;            // initial Polygon
    BBT      Tree;          // balanced binary tree
public:
              SweepLine(Polygon P)            // constructor
                  { nv = P.n; Pn = &P; }
             ~SweepLine(void)                 // destructor
```

```
                  { Tree.freetree();}

    SLseg*   add( Event* );
    SLseg*   find( Event* );
    int      intersect( SLseg*, SLseg*  );
    void     remove( SLseg* );
};

SLseg* SweepLine::add( Event* E )
{
    // fill in SLseg element data
    SLseg* s = new SLseg;
    s->edge  = E->edge;

    // if it is being added, then it must be a LEFT edge event
    // but need to determine which endpoint is the left one
    Point* v1 = &(Pn->V[s->edge]);
    Point* v2 = &(Pn->V[s->edge+1]);
    if (xyorder( v1, v2) < 0) { // determine which is leftmost
        s->lP = *v1;
        s->rP = *v2;
    }
    else {
        s->rP = *v1;
        s->lP = *v2;
    }
    s->above = (SLseg*)0;
    s->below = (SLseg*)0;

    // add a node to the balanced binary tree
    Tnode* nd = Tree.insert(s);
    Tnode* nx = Tree.next(nd);
    Tnode* np = Tree.prev(nd);
    if (nx != (Tnode*)0) {
        s->above = (SLseg*)nx->val;
        s->above->below = s;
    }
    if (np != (Tnode*)0) {
        s->below = (SLseg*)np->val;
        s->below->above = s;
    }
    return s;
}

SLseg* SweepLine::find( Event* E )
{
    // need a segment to find it in the tree
    SLseg* s = new SLseg;
    s->edge  = E->edge;
    s->above = (SLseg*)0;
    s->below = (SLseg*)0;

    Tnode* nd = Tree.find(s);
    delete s;
    if (nd == (Tnode*)0)
        return (SLseg*)0;

    return (SLseg*)nd->val;
}

void SweepLine::remove( SLseg* s )
{
    // remove the node from the balanced binary tree
    Tnode* nd = Tree.find(s);
    if (nd == (Tnode*)0)
        return;       // not there

    // get the above and below segments pointing to each other
    Tnode* nx = Tree.next(nd);
    if (nx != (Tnode*)0) {
        SLseg* sx = (SLseg*)(nx->val);
        sx->below = s->below;
    }
    Tnode* np = Tree.prev(nd);
    if (np != (Tnode*)0) {
        SLseg* sp = (SLseg*)(np->val);
        sp->above = s->above;
    }
    Tree.remove(nd);       // now  can safely remove it
    delete s;
}

// test intersect of 2 segments and return: 0=none, 1=intersect
int SweepLine::intersect( SLseg* s1, SLseg* s2)
{
    if (s1 == (SLseg*)0 || s2 == (SLseg*)0)
        return FALSE;       // no intersect if either segment doesn't exist

    // check for consecutive edges in polygon
    int e1 = s1->edge;
    int e2 = s2->edge;
    if (((e1+1)%nv == e2) || (e1 == (e2+1)%nv))
        return FALSE;       // no non-simple intersect since consecutive

    // test for existence of an intersect point
    float lsign, rsign;
    lsign = isLeft(s1->lP, s1->rP, s2->lP);    //  s2 left point sign
    rsign = isLeft(s1->lP, s1->rP, s2->rP);    //  s2 right point sign
    if (lsign * rsign > 0) // s2 endpoints have same sign  relative to s1
        return FALSE;       // => on same side => no intersect is possible
    lsign = isLeft(s2->lP, s2->rP, s1->lP);    //  s1 left point sign
    rsign = isLeft(s2->lP, s2->rP, s1->rP);    //  s1 right point sign
    if (lsign * rsign > 0) // s1 endpoints have same sign  relative to s2
        return FALSE;       // => on same side => no intersect is possible
```

```
    // the segments s1 and s2 straddle each other
    return TRUE;            // => an intersect exists
}
//===================================================================



// simple_Polygon(): test if a Polygon is simple or not
//     Input:  Pn = a polygon with n vertices V[]
//     Return: FALSE(0) = is NOT simple
//             TRUE(1)  = IS simple
int
simple_Polygon( Polygon Pn )
{
    EventQueue  Eq(Pn);
    SweepLine   SL(Pn);
    Event*      e;                  // the current event
    SLseg*      s;                  // the current SL segment

    // This loop processes all events in the sorted queue
    // Events are only left or right vertices since
    // No new events will be added (an intersect => Done)
    while (e = Eq.next()) {         // while there are events
        if (e->type == LEFT) {      // process a left vertex
            s = SL.add(e);          // add it to the sweep line
            if (SL.intersect(  s, s->above))
                 return FALSE;      // Pn is NOT simple
            if (SL.intersect(  s, s->below))
                 return FALSE;      // Pn is NOT simple
        }
        else {                      // processs a right vertex
            s = SL.find(e);
            if (SL.intersect(  s->above, s->below))
                 return FALSE;      // Pn is NOT simple
            SL.remove(s);           // remove it from the sweep line
        }
    }
    return TRUE;      // Pn IS simple
}
//===================================================================
```

References

Brad Appleton, C++ code for an AVL balanced Tree, see: www.bradapp.com, oopweb.com/Algorithms/Documents/AvlTrees/VolumeFrames.html, www.bradapp.com/ftp/src/libs/C++/AvlTrees.html (1997)

I.J. Balaban, "An Optimal Algorithm for Finding Segment Intersections", Proc. 11-th Ann. ACM Sympos. Comp. Geom., 211-219 (1995)

Ulrike Bartuschka, Kurt Mehlhorn & Stefan Naher, "A Robust and Efficient Implementation of a Sweep Line Algorithm for the Straight Line Segment Intersection Problem", Proc. Workshop on Algor. Engineering, Venice, Italy, 124-135 (1997)

Jon Bentley & Thomas Ottmann, "Algorithms for Reporting and Counting Geometric Intersections", IEEE Trans. Computers C-28, 643-647 (1979)

Mark de Berg et al, Computational Geometry : Algorithms and Applications (2nd Ed), Chapter 2 "Line Segment Intersection" (2010)

Glenn Burkhardt, personal communication (2014).

Tim Chan, "A Simple Trapezoid Sweep Algorithm for Reporting Red/Blue Segment Intersections", Proc. 6-th Can. Conf. Comp. Geom., Saskatoon, Saskatchewan, Canada, 263-268 (1994)

Bernard Chazelle & Herbert Edelsbrunner, "An Optimal Algorithm for Intersecting Line Segments in the Plane", Proc. 29-th Ann. IEEE Sympos. Found. Comp. Sci., 590-600 (1988)

Bernard Chazelle & Herbert Edelsbrunner, "An Optimal Algorithm for Intersecting Line Segments in the Plane", J. ACM 39, 1-54 (1992)

K.L. Clarkson & P.W. Shor, "Applications of Random Sampling in Computational Geometry, II", Discrete Comp. Geom. 4, 387-421 (1989)

John Hobby, "Practical Segment Intersection with Finite Precision Output", Comp. Geom. : Theory & Applics. 13(4), (1999) [Note: the original Bell Labs paper appeared in 1993]

H.G. Mairson & J. Stolfi, "Reporting and Counting Intersections between Two Sets of Line Segments", in: Theoretic Found. of Comp. Graphics and CAD, NATO ASI Series Vol. F40, 307-326 (1988)

E. Myers, "An O(E log E + I) Expected Time Algorithm for the Planar Segment Intersection Problem", SIAM J. Comput., 625-636 (1985)

Joseph O'Rourke, Computational Geometry in C (2nd Edition), Section 7.7 "Intersection of Segments" (1998)

Franco Preparata & Michael Shamos, Computational Geometry: An Introduction, Chapter 7 "Intersections" (1985)

Michael Shamos & Dan Hoey, "Geometric Intersection Problems", Proc. 17-th Ann. Conf. Found. Comp. Sci., 208-215 (1976)

Home | Math | Algorithms | Code | Book Store | WebSites