INTERSECTIONS OF LINE SEGMENTS AND POLYGONS

PETR FELKEL
FEL CTU PRAGUE
felkel@fel.cvut.cz
https://cw.felk.cvut.cz/doku.php/courses/a4m39vg/start

Based on [Berg], [Mount], [Kukral], and [Drtina]

Version from 13.11.2014
Talk overview

- Intersections of line segments (Bentley-Ottmann)
  - Motivation
  - Sweep line algorithm recapitulation
  - Sweep line intersections of line segments
- Intersection of polygons or planar subdivisions
  - See assignment [21] or [Berg, Section 2.3]
- Intersection of axis parallel rectangles
  - See assignment [26]
Geometric intersections – what are they for?

One of the most basic problems in computational geometry

- **Solid modeling**
  - Intersection of object boundaries in CSG

- **Overlay of subdivisions, e.g. layers in GIS**
  - Bridges on intersections of roads and rivers
  - Maintenance responsibilities (road network X county boundaries)

- **Robotics**
  - Collision detection and collision avoidance

- **Computer graphics**
  - Rendering via ray shooting (intersection of the ray with objects)
Line segment intersection

- Intersection of complex shapes is often reduced to simpler and simpler intersection problems
- Line segment intersection is the most basic intersection algorithm
- Problem statement: Given $n$ line segments in the plane, report all points where a pair of line segments intersect.
- Problem complexity
  - Worst case – $I = O(n^2)$ intersections
  - Practical case – only some intersections
  - Use an output sensitive algorithm
    - $O(n \log n + I)$ optimal randomized algorithm
    - $O(n \log n + I \log n)$ sweep line algorithm

Felkel: Computational geometry
(4 / 71)
Plane sweep line algorithm recapitulation

- Horizontal line (sweep line, scan line) $\ell$ moves top-down (or vertical line: left to right) over the set of objects
- The move is not continuous, but $\ell$ jumps from one event point to another
  - Event points are in priority queue or sorted list ($\sim y$)
  - The top-most event point is removed first
  - New event points may be created (usually as interaction of neighbors on the sweep line) and inserted in the queue
- Scan-line status
  - Stores information about the objects intersected by $\ell$
  - It is updated while stopping on event point
Line segment intersection - Sweep line alg.

- Avoid testing of pairs of segments far apart
- Compute intersections of neighbors on the sweep line only
- $O(n \log n + I \log n)$ time in $O(n)$ memory
  - $2n$ steps for end points,
  - $I$ steps for intersections,
  - $\log n$ search the status tree
- Ignore “nasty cases” (most of them will be solved later on)
  - No segment is parallel to the sweep line
  - Segments intersect in one point and do not overlap
  - No three segments meet in a common point
Line segment intersections

- **Status** = ordered sequence of segments intersecting the sweep line \( \ell \)

- **Events** (waiting in the priority queue)
  = points, where the algorithm actually does something
  - Segment *end-points*
    • known at algorithm start
  - Segment *intersections* between neighboring segments along SL
    • Discovered as the sweep executes
Detecting intersections

- Intersection events must be detected and inserted to the event queue before they occur.
- Given two segments $a$, $b$ intersecting in a point $p$, there must be a placement of sweep line $l$ prior to $p$, such that segments $a$, $b$ are adjacent along $l$ (only adjacent will be tested for intersection).
  - segments $a$, $b$ are not adjacent when the alg. starts
  - segments $a$, $b$ are adjacent just before $p$

$\Rightarrow$ there must be an event point when $a,b$ become adjacent and therefore are tested for intersection.
Data structures

Sweep line $\ell$ status = order of segments along $\ell$

- Balanced binary search tree of segments
- Coords of intersections with $\ell$ vary as $\ell$ moves
  => store pointers to line segments in tree nodes
    - Position of $\ell$ is plugged in the $y=mx+b$ to get the x-key
Data structures

Event queue (postupový plán, časový plán)

- Define: **Order** $<$ (top-down, lexicographic)

  $p < q \iff p_y > q_y \text{ or } p_y = q_y \text{ and } p_x < q_x$

  top-down, left-right approach

  (points on $\ell$ treated left to right)

- Operations
  - Insertion of computed intersection points
  - Fetching the **next event** (highest $y$ below $\ell$)
  - Test, if the segment is already **present in the queue**
    (Delete intersection event in the queue)
Problem with duplicities of intersections

3x detected intersection
Data structures

Event queue data structure

- **Heap**
  - Problem: can not check *duplicated intersection events* (reinvented more than once)
  - Intersections processed twice or even more
  - Memory complexity up to $O(n^2)$

- **Ordered dictionary (balanced binary tree)**
  - Can check duplicated events (adds just constant factor)
  - Nothing inserted twice
  - If non-neighbor intersections are deleted, i.e., if only intersections of neighbors along $l$ are stored then memory complexity just $O(n)$
Line segment intersection algorithm

FindIntersections(S)

Input: A set $S$ of line segments in the plane
Output: The set of intersection points + pointers to segments in each

1. init an empty event queue $Q$ and insert the segment endpoints
2. init an empty status structure $T$
3. while $Q$ in not empty
4. remove next event $p$ from $Q$
5. handleEventPoint($p$)

- Upper endpoint
- Intersection
- Lower endpoint

Note: Upper-end-point events store info about the segment
handleEventPoint principle

- **Upper endpoint** $U(p)$
  - insert $p$ (on $s_j$) to status $T$
  - add intersections with left and right neighbors to $Q$

- **Intersection** $C(p)$
  - switch order of segments in $T$
  - add intersections of left and right neighbors to $Q$

- **Lower endpoint** $L(p)$
  - remove $p$ (on $s_l$) from $T$
  - add intersections of left and right neighbors to $Q$
More than two segments incident

$U(p) = \{s_2\}$ — start here

$C(p) = \{s_1, s_3\}$ — cross on $\ell$

$L(p) = \{s_4, s_5\}$ — end here
**Handle Events** [Berg, page 25]

**handleEventPoint(p)**

1. Let $U(p)$ = set of segments whose Upper endpoint is $p$. These segments are stored with the event point $p$ (will be added to $T$)

2. Search $T$ for all segments $S(p)$ that contain $p$ (are adjacent in $T$):
   - Let $L(p) \subset S(p)$ = segments whose Lower endpoint is $p$
   - Let $C(p) \subset S(p)$ = segments that Contain $p$ in interior

3. if( $L(p) \cup U(p) \cup C(p)$ contains more than one segment )

4. report $p$ as intersection together with $L(p)$, $U(p)$, $C(p)$

5. Delete the segments in $L(p) \cup C(p)$ from $T$

6. Insert the segments in $U(p) \cup C(p)$ into $T$

   Reverse order of $C(p)$ in $T$

   (order as below $l$, horizontal segment as the last)

7. if( $U(p) \cup C(p) = \emptyset$ ) then findNewEvent($s_l$, $s_r$, $p$) // left & right neighbors

8. else $s'$ = leftmost segment of $U(p) \cup C(p)$; findNewEvent($s_l$, $s'$, $p$)
   $s''$ = rightmost segment of $U(p) \cup C(p)$; findNewEvent($s''$, $s_r$, $p$)
Detection of new intersections

\textbf{findNewEvent}(s_l, s_r, p)  // with handling of horizontal segments

\textit{Input:} two segments (left & right from \( p \) in \( T \)) and a current event point \( p \)

\textit{Output:} updated event queue \( Q \) with new intersection

1. if \([ (s_l \text{ and } s_r \text{ intersect below the sweep line } \ell) \text{ or } (s_l \text{ and } s_r \text{ intersect on } \ell \text{ and to the right of } p ) ] \text{ and } \) \( (\text{the intersection is not present in } Q) \)

2. then

   insert \( p \) as a new event into \( Q \)

\( s'_r = \text{leftmost segment of } U(p) \cup C(p); \)

\( s_l \text{ and } s_r = s'_r \text{ intersect on } \ell \)

\( s_l \text{ and } s_r = s' \text{ intersect below and to the right of } p \)
Line segment intersections

- Memory $O(I) = O(n^2)$ with duplicities in Q
  or $O(n)$ with duplicities in Q deleted

- Operational complexity
  - $n + I$ stops
  - $\log n$ each
  => $O(I + n) \log n$ total

- The algorithm is by Bentley-Ottmann


  See also [http://wapedia.mobi/en/Bentley%E2%80%93Ottmann_algorithm](http://wapedia.mobi/en/Bentley%E2%80%93Ottmann_algorithm)
Intersection of axis parallel rectangles

Given the collection of \( n \) isothetic rectangles, report all intersecting parts

Answer: \((r_1, r_2) \quad (r_1, r_3) \quad (r_1, r_8) \quad (r_3, r_4) \quad (r_3, r_5) \quad (r_3, r_9) \quad (r_4, r_5) \quad (r_7, r_8)\)

Alternate sides belong to two pencils of lines

(often used with points in infinity = axis parallel)
Brute force intersection

Brute force algorithm

Input: set $S$ of axis parallel rectangles
Output: pairs of intersected rectangles

1. For every pair $(r_i, r_j)$ of rectangles $\in S, i \neq j$
2. if $(r_i \cap r_j \neq \emptyset)$ then
3. report $(r_i, r_j)$

Analysis
Preprocessing: None.
Query: $O(N^2)$
\[
\binom{N}{2} = \frac{N(N-1)}{2} \in O(N^2).
\]
Storage: $O(N)$
Plane sweep intersection algorithm

- Vertical sweep line moves from left to right
- Stops at every x-coordinate of a rectangle (either its left side or its right side).
- **active rectangles** – a set
  - = rectangles currently intersecting the sweep line
  - left side event of a rectangle
    => the rectangle is added to the active set.
  - right side
    => the rectangle is deleted from the active set.

- The active set used to detect rectangle intersection
Example rectangles and sweep line
Interval tree as sweep line status structure

- Vertical sweep-line => Only y-coordinates along it
- Turn our view in slides 90° right
- Sweep line (y-axis) will be drawn as horizontal

Diagram showing active and not active rectangles along the sweep line.
Intersection test – between pair of intervals

- Given two intervals $R = [y_1, y_2]$ and $R' = [y'_1, y'_2]$ the condition $R \cap R'$ is equivalent to one of these mutually exclusive conditions:

  a) $y_1 \leq y'_1 \leq y_2$

  b) $y'_1 \leq y_1 \leq y'_2$
Static interval tree – stores all end points

- Let \( v = \frac{y_{med}}{} \) be the median of end-points of segments
- \( S_l \) : segments of S that are completely to the left of \( y_{med} \)
- \( S_{med} \) : segments of S that contain \( y_{med} \)
- \( S_r \) : segments of S that are completely to the right of \( y_{med} \)
Static interval tree – Example

Interval tree on $s_3$ and $s_5$

Interval tree on $s_2$ and $s_7$

$M_l = (s_4, s_6, s_1)$

$M_r = (s_1, s_4, s_6)$

Left ends – ascending

Right ends – descending
Static interval tree [Edelsbrunner80]

- Stores intervals along y sweep line
- 3 kinds of information
  - end points
  - incident intervals
  - active nodes

[Kukral]
$v = \text{midpoint of all segment endpoints}$

$H(v) = \text{value (y-coord) of } v$
Secondary lists of incident interval end-pts.

ML(v) – left endpoints of interval containing v (sorted ascending)

MR(v) – right endpoints (descending)
Active nodes – intersected by the sweep line

Subset of all nodes at present intersected by sweep line (nodes with intervals)
Query = sweep and report intersections

RectangleIntersections( S )

Input: Set S of rectangles
Output: Intersected rectangle pairs

1. Preprocess( S ) ◆ create the interval tree T (for y-coords)
   ◆ and event queue Q (for x-coords)

2. while ( Q ≠ ∅ ) do
3.   Get next entry (xᵢ, yᵢ₀, yᵢᵢ, t) from Q ◆ t ∈ { left | right }
4.   if ( t = left ) ◆ left edge
5.     a) QueryInterval ( yᵢ₀, yᵢᵢ, root(T) ) ◆ report intersections
6.     b) InsertInterval ( yᵢ₀, yᵢᵢ, root(T) ) ◆ insert new interval
7.   else ◆ right edge
8.     c) DeleteInterval ( yᵢ₀, yᵢᵢ, root(T) )
Preprocessing

Preprocess( S )

**Input:** Set S of rectangles

**Output:** Primary structure of the interval tree T and the event queue Q

1. \( T = \text{PrimaryTree}(S) \)    // Construct the static primary structure
   // of the interval tree -> sweep line STATUS T

2. // Init event queue Q with vertical rectangle edges in ascending order \( \sim x \)
   // Put the left edges with the same \( x \) ahead of right ones
3. for \( i = 1 \) to \( n \)
4.    insert( ( \( x_{il}, y_{il}, y_{ir}, \text{left} \), Q)  // left edges of \( i-th \) rectangle
5.    insert( ( \( x_{ir}, y_{il}, y_{ir}, \text{right} \), Q)  // right edges
Interval tree – primary structure construction

**PrimaryTree(S)**  // only the y-tree structure, without intervals

*Input:* Set S of rectangles  
*Output:* Primary structure of an interval tree T

1. \( S_y = \) Sort endpoints of all segments in \( S \) according to \( y \)-coordinate
2. \( T = \) BST( \( S_y \) )
3. return \( T \)

**BST( \( S_y \) )**

1. if \( |S_y| = 0 \) return null
2. \( yMed = \) median of \( S_y \)
3. \( L = \) endpoints \( p_y \leq yMed \)
4. \( R = \) endpoints \( p_y > yMed \)
5. \( t = \) new IntervalTreeNode( \( yMed \) )
6. \( t.left = \) BST(\( L \))
7. \( t.right = \) BST(\( R \))
8. return \( t \)
Interval tree – search the intersections

**QueryInterval** (b, e, T)

**Input:** Interval of the edge and current tree T

**Output:** Report the rectangles that intersect [b, e]

1. if (T = null) return
2. i=0; if (b < H(v) < e) // forks at this node
3. while (MR(v).[i] >= b) && (i < Count(v)) .. Report all intervals in M
4. ReportIntersection; i++
5. QueryInterval(b,e,T.LPTR)
6. QueryInterval(b,e,T.RPTR)
7. else if (H(v) ≤ b < e) // search RIGHT (←)
8. while (MR(v).[i] >= b) && (i < Count(v))
9. ReportIntersection; i++
10. QueryInterval(b,e,T.RPTR)
11. else // b < e ≤ H(v) //search LEFT(→)
12. while (ML(v).[i] <= e)
13. ReportIntersection; i++
14. QueryInterval(b,e,T.LPTR)

New interval being tested for intersection

Stored intervals of active rectangles

A
B
C
Crosses A, B
Crosses A, B, C
Crosses nothing
Cross.B

DCGI

Felkel: Computational geometry

Interval of the edge and current tree T

Report the rectangles that intersect [b, e]
Interval tree - interval insertion

InsertInterval ( b, e, T )

Input: Interval [b,e] and interval tree T
Output: T after insertion of the interval

1. v = root(T)
2. while ( v != null ) // find the fork node
3.  if (H(v) < b < e)
4.    v = v.right // continue right
5.  else if (b < e < H(v))
6.    v = v.left // continue left
7.  else // b ≤ H(v) ≤ e // insert interval
8.    set v node to active
9.    connect LPTR resp. RPTR to its parent
10.   insert [b,e] into list ML(v) – sorted in ascending order of b’s
11.   insert [b,e] into list MR(v) – sorted in descending order of e’s
12.   break
13. endwhile
14. return T
Example 1
Example 1 – static tree on endpoints

H(v) – value of node v

Felkel: Computational geometry

(37 / 71)
Interval insertion [1,3]  a) Query Interval

Search MR(v) or ML(v): \[ b < H(v) < e \]

MR(v) is empty
No active sons, stop

1 \(<\circ\) < 3
Interval insertion $[1,3]$  b) Insert Interval

b $\leq H(v) \leq e$

$? \ 1 \leq 2 \leq 3 ?$

Active rectangle
Current node
Active node

Felkel: Computational geometry
(39 / 71)
**Interval insertion [1,3]**

b) Insert Interval

\[ b \leq H(v) \leq e \]

\[ 1 \leq 2 \leq 3 \]

fork

\[ \Rightarrow \text{to lists} \]
Interval insertion $[2,4]$  

a) Query Interval

Search $\text{MR}(v)$ only:

$H(v) \leq b < e$

$\text{MR}(v)[1] = 3 \geq 2$?

$2 \leq 2 < 4$

$\Rightarrow$ intersection

Active rectangle

Current node

Active node

Felkel: Computational geometry

(Drtina)
Interval insertion $[2,4]$  

b) Insert Interval

$2 \leq (2) \leq 4$

fork => to lists
Interval delete [1,3]
Interval delete [1,3]
Interval delete \([2,4]\)
Interval delete $[2, 4]$
Example 2

RectangleIntersections( S )  // this is a copy of the slide before
\textit{Input:} Set S of rectangles  // just to remember the algorithm
\textit{Output:} Intersected rectangle pairs

1. Preprocess( S )  // create the interval tree T and event queue Q

2. while ( Q \neq \emptyset ) do

3. Get next entry (x_{il}, y_{il}, y_{ir}, t) from Q  // t \in \{ \text{left} | \text{right} \}

4. if ( t = \text{left} )  // left edge

5. a) QueryInterval ( y_{il}, y_{ir}, \text{root}(T) )  // report intersections

6. b) InsertInterval ( y_{il}, y_{ir}, \text{root}(T) )  // insert new interval

7. else  // right edge

8. c) DeleteInterval ( y_{il}, y_{ir}, \text{root}(T) )
Example 2
**Insert** $[2,3] – empty => b) Insert Interval

$b \leq H(v) \leq e$

Insert the new interval to secondary lists

fork node => active
=> to lists

[Drtina] (49 / 71)
for (all in \text{MR}(v)) \text{test MR}(v)[i] \geq 3
\Rightarrow \text{report intersection}

\text{H}(v) \leq b < e

\text{Insert } [3, 7] \quad \text{a) Query Interval}

\text{Current node}

\text{Active node}

\text{Active rectangle}

Felkel: Computational geometry

[Drtina]
**Insert** $[3,7]$ b) **Insert Interval**

Insert the new interval to secondary lists

$b \leq H(v) \leq e$

$3 \leq 3 \leq 7$

fork node $\Rightarrow$ active

$\Rightarrow$ to lists
**Insert $[0,2]$**

a) **Query Interval**

- for (all in ML(v)) test $ML(v).[i] \leq 2$
- $\Rightarrow$ report intersection
- go left, nil, stop

b) $b < e \leq H(v)$

? $0 < 2 \leq 3$?

Drgina (52/71)

Felkel: Computational geometry

(52/71)
Insert $[0,2]$ b) Insert Interval 1/2

$b < e < H(v)$

? $0 < 2 < 3$?

=>$ \text{insert left}$

[Diagram of a data structure with intervals and nodes]
**Insert** $[0, 2]$ b) Insert Interval 2/2

$b \leq H(v) \leq e$

? $0 \leq 1 \leq 2$ ?

Insert the new interval to secondary lists of the left son link to parent

fork node => active

=> to lists
**Insert** [1, 5]  

a) Query Interval 1/2

for (all in MR(v))

=> report intersection c, d

go left \( \rightarrow 1 \)
go right - nil

\[ b < H(v) < e \]

? 1 \( < \) 3 \( < \) 5 ?
**Insert [1,5]**  

a) **Query Interval 2/2**

\[ H(v) \leq b < e \]

for (all in \( MR(v) \)) test \( MR(v)[i] \geq 1 \)  
=> report intersection a  
go right, nil, stop
**Insert** $[1, 5]$ b) Insert Interval

$3 \leq H(v) \leq e$

? $1 \leq 3 \leq 5$ ?

Insert the new interval to secondary lists

Insert [1,5] b) Insert Interval

$3 \leq H(v) \leq e$

? $1 \leq 3 \leq 5$ ?

Insert the new interval to secondary lists
**Insert** $[7,8]$  

**a) Query Interval**

$H(v) \leq b < e$

for (all in $MR(v)$) test $MR(v)\[i\] \geq 7$

$\Rightarrow$ report intersection $d$

go right, nil, stop
**Insert [7, 8]**

b) **Insert Interval**

Insert the new interval to secondary lists
link to parent

<table>
<thead>
<tr>
<th>X</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Y</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
</tbody>
</table>

```
right <= ? 3 ≤ 7 < 8 ?
right <= ? 5 ≤ 7 < 8 ?
7 ≤ 7 ≤ 8
```

Insert [7, 8]

Current node
Active node
Active rectangle

2) Insert Interval

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

b ≤ H(v) ≤ e
Delete [3, 7] Delete Interval

Delete the interval [3, 7] from secondary lists

b ≤ H(v) ≤ e

? 3 ≤ 7 ≤ 8 ?

Active rectangle
Current node
Active node

Felkel: Computational geometry

[Drtina] (60 / 71)
Insert $[4,6]$ a) Query Interval

for (all in MR(v)) test MR(v).[i] $\geq 4$ => report intersection b
for (all in ML(v)) test ML(v).[i] $\leq 6$

$\implies$ no intersection
**Insert** \([4, 6]\) b) **Insert Interval**

Insert the new interval to secondary lists

\[ H(v) \leq b < e \]

? \(3 \leq 4 < 6\)?

\[ \text{Current node} \]

\[ \text{Active rectangle} \]

\[ \text{Active node} \]

\[ \text{DCGI} \]
Delete $[1,5]$ Delete Interval

Delete the interval $[1,5]$ from secondary lists

$b \leq H(v) \leq e$

? $1 \leq 3 \leq 5$?
Delete $[0,2]$ Delete Interval $1/2$

Search for node with interval $[0,2]$

$0 < 2 \leq 3$?

$b < e \leq H(v)$
Delete $[0, 2]$  
Delete Interval 2/2

Delete the interval $[0, 2]$ from secondary lists of node 1

$b \leq H(v) \leq e$
Delete $[7,8]$ Delete Interval

Search for and delete node with interval $[7,8]$

$\begin{align*}
\text{Active rectangle} & : \text{Active rectangle} \\
\text{Current node} & : \text{Current node} \\
\text{Active node} & : \text{Active node}
\end{align*}$
Delete \([2,3]\) Delete Interval

Search for and delete node with interval \([2,3]\)

\[b \leq H(v) \leq e\]

? \(2 \leq 3 \leq 3\) ?
Delete \([4, 6]\) Delete Interval  

Find and delete node with interval \([4, 6]\) 

\[b \leq H(v) \leq e\]

? \(4 \leq 5 \leq 6\) ?
Empty tree

Search for and delete node with interval [4,6]
Complexities of rectangle intersections

- $n$ rectangles, $s$ intersected pairs found
- $O(n \log n)$ preprocessing time to separately sort
  - x-coordinates of the rectangles for the plane sweep
  - the y-coordinates for initializing the interval tree.
- The plane sweep itself takes $O(n \log n + s)$ time, so the overall time is $O(n \log n + s)$
- $O(n)$ space
- This time is optimal for a decision-tree algorithm (i.e., one that only makes comparisons between rectangle coordinates).
References


http://maven.smith.edu/~orourke/books/compgeom.html


[Vigneron] Segment trees and interval trees, presentation, INRA, France,
http://w3.jouy.inra.fr/unites/miaj/public/vigneron/cs4235/slides.html