INTERSECTIONS OF LINE SEGMENTS AND AXIS ALIGNED RECTANGLES, OVERLAY OF SUBDIVISIONS

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 19.11.2020
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
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
(5 / 96)
**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 (left) top-most event point is removed first
  - New event points may be created (usually as interaction of neighbors on the sweep line) and inserted into 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
  - 2\(n\) steps for end points,
  - \(I\) steps for intersections,
  - \(\log n\) search the status tree
- Ignore “degenerate 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 point $p$, there must be a placement of sweep line $\ell$ prior to $p$, such that segments $a, b$ are adjacent along $\ell$ (only adjacent will be tested for intersection).
  - segments $a, b$ are not adjacent when the alg. starts
  - segments $a, b$ are adjacent just before $p$

$=>$ there must be an event point when $a, b$ become adjacent and therefore are tested for intersection

$=>$ All intersections are found.
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 \( \ell \) or the leftmost right of \( e \))
  - Test, if the segment is already present in the queue
    (Locate and delete intersection event in the queue)
Problem with duplicities of intersections

Intersection may be detected many times

3x detected intersection
Data structures

Event queue data structure

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

b) 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-endpoint events store info about the segment

**Improved algorithm:**
Handles all in $p$ in a single step
**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 with nearest left and nearest right neighbor 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

\[ U(p) = \{s_2\} \]  \(\square\) start here
\[ C(p) = \{s_1, s_3\} \]  \(\square\) cross on \(\ell\)
\[ L(p) = \{s_4, s_5\} \]  \(\square\) end here

Felkel: Computational geometry
(16 / 96)
Handle Events  [modified Berg, page 25]

handleEventPoint(p)  // precisely: handle all events with point 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 \( L(p) \cup S(p) \) = segments whose Lower endpoint is \( p \)
   Let \( C(p) \cup S(p) \) = 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. if( \( U(p) \cup C(p) = \emptyset \) ) then findNewEvent(\( s_l, s_r, p \))  // left & right neighbors
7. else Insert the segments in \( U(p) \cup C(p) \) into \( T \)  // reverse order of \( C(p) \) in \( T \\
   \) (order as below \( \ell \), horizontal segment as the last)
8. \( s' = \) leftmost segm. of \( U(p) \cup C(p) \); \( s' \) findNewEvent(\( s_l, s', p \))
9. \( s'' = \) rightmost segm. of \( U(p) \cup C(p) \); \( s'' \) findNewEvent(\( s'', s_r, p \))
Detection of new intersections

\[ \text{findNewEvent}(s_l, s_r, p) \quad \text{// with handling of horizontal segments} \]

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

**Output:** updated event queue \( Q \) with new intersection \( \bullet \)

1. **if** \( (s_l \text{ and } s_r \text{ intersect below the sweep line } \ell) \) // intersection below \( \ell \)
   
   or \( (s_r \text{ intersect } s'' \text{ on } \ell \text{ and to the right of } p) \) ] // horizontal segment
   
   and( the intersection \( \bullet \) is not present in \( Q \) )

2. **then**
   
   insert intersection \( \bullet \) as a new event into \( Q \)

---

Non-overlapping

\( s_l \text{ and } s_r \text{ intersect below } \ell \)

\( s'_r \text{ intersect } s'' \text{ on } \ell \text{ and to the right of } p \)

Reported intersection - line 4

New intersection to \( Q \) - line 6,8,9
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

  $\Rightarrow O(I + n) \log n$ total

- The algorithm is by Bentley-Ottmann
  

  See also http://wapedia.mobi/en/Bentley%E2%80%93Ottmann_algorithm
Overlay of two subdivisions
(intersection of DCELs)
Overlay of two subdivisions

DCEL $S_1$

DCEL $S_2$

hole
Overlay is a new planar subdivision

DCEL $\mathcal{O}(S_1, S_2)$
Sweep line overlay algorithm

- Compute new planar subdivision
- Re-use not intersected half-edge records
- Compute intersections and new half-edge records
- Compute labels of new faces
The algorithm principle

Copy DCELs of both subdivisions to invalid DCEL $\mathcal{D}$

Transform the result into a valid DCEL for the subdivision overlay $\mathcal{O}(S_1, S_2)$

- Compute the intersection of edges (from different subdivisions $S_1 \cap S_2$)
- Link together appropriate parts of the two DCELs
  - Vertex and half-edge records
  - Face records
At an Event point

- Update queue $Q$ (pop, delete intersections of separated edges below) and sweep line status tree $T$ (add/remove/swap edges, compute intersections with neighbors) as in line segment intersection algorithm (cross pointers between edges in $T$ and $D$ to access part of $D$ when processing an intersection)

- For vertex from one subdivision
  - No additional work

- For Intersection of edges from different subdivisions
  - Link both DCELs
  - Handle all possible cases
Three types of intersections

New are intersections of different subdivisions

vertex – vertex: overlap of vertices

vertex – edge: edge passes through a vertex

edge – edge: edges intersect in their interior

Let’s discuss this case, the other two are similar.
vertex – edge update – the principle

Before:
The geometry

Before:
two half-edges

After:
four half-edges
(two shorter and two new)
1. Edge $e = (u, w)$ splits into two edges $e'$ and $e''$ at intersection $v$

$$e' = (w, v) \quad e'' = (v, u)$$

2. Shorten half-edge $(w, u)$ to $(w, v)$
   Shorten half-edge $(u, w)$ to $(u, v)$

3. Create their twin $(v, w)$ for $(w, v)$
   Create their twin $(v, u)$ for $(u, v)$

4. Set new twin’s next to former edge $e$ next
   $\text{next}(v, u) = \text{next}(w, u)$ now in $\text{next}(w, v)$
   $\text{next}(v, w) = \text{next}(u, w)$ now in $\text{next}(u, v)$

5. Set prev pointers to new twins
   $\text{prev}(\text{next}(v, u)) = (v, u)$
   $\text{prev}(\text{next}(v, w)) = (v, w)$
Pointers around intersection $v$

6. Find the next edge $x$ for $e'$ from half-edge $(w, v)$
   - $= \text{first CW half-edge from } e' \text{ with } v \text{ as origin}$
   - next$(w, v) = x$
   - prev$(x) = (w, v)$

7. Find the prev edge for $e'$ from half-edge $(v, w)$
   - $= \text{first CCW half-edge from } e' \text{ with } v \text{ as destination}$
   - next, prev similarly

8. Find the next edge for $e''$ from half-edge $(u, v)$
   - $= \text{first CW half-edge from } e'' \text{ with } v \text{ as origin}$
   - next, prev similarly

9. Find the prev edge for $e''$ from half-edge $(v, u)$
   - $= \text{first CCW half-edge from } e' \text{ with } v \text{ as destination}$
   - next, prev similarly
Time cost for updating half-edge records

- All operations with splitting of edges in intersections and reconnecting of prev, next pointers take $O(1)$ time.

- Locating of edge position in cyclic order
  - around single vertex $v$ takes $O(\text{deg}(v))$
  - which sums to $O(m) = \text{number of edges processed by the edge intersection algorithm} = O(n)$
  - The overall complexity is not increased

$$O(n \log n + k \log n)$$

$n = |S_1| + |S_2|$  
$k = \text{complexity of the overlay (≈ intersections)}$

Complexity of input subdivisions
Face records for the overlay subdivision

- Create face records for each face $f$ in $\mathcal{O}(S_1, S_2)$
  - Each face $f$ has its unique outer boundary (CCW) (except the background that has none)
  - Each face has its $OuterComponent(f)$ – store edge of it
  - Together faces = $\#outer$ boundaries + 1

- $InnerComponents(f)$ – list of edges of holes (cw)

- Label of $f$ in $S_1$
- Label of $f$ in $S_2$

Used for Boolean operations such as $S_1 \cap S_2$, $S_1 \cup S_2$, $S_1 \setminus S_2$

Polygon examples:
Extraction of faces

- Traverse cycles in DCEL (Tarjan alg. DFS) \( O(n) \)
- Decide, if the cycle is outer or inner boundary
  - Find leftmost vertex of the cycle (bottom leftmost)
  - Incident face lies to the left of edges
  - Angle \(< 180^\circ\) ⇒ outer
  - Angle \(> 180^\circ\) ⇒ inner (hole)
Which boundary cycles bound same face?

- Single outer boundary shares the face with its holes – inner boundaries

- Graph
  - Node for each cycle
    - $e_3$ inner
    - $e_2$ outer
    - $e_{\infty}$ unbounded
  - Arc if inner cycle has half-edge immediately to the left of the leftmost vertex
  - Each connected component – set of cycles of one face
Graph $\mathcal{G}$ of faces and their relations

- $C_3$ inner (cw)
- $C_2$ outer (ccw)
- $C_\infty$ unbounded

Connected component in $\mathcal{G}$
- represents a face $f$
- connects outer face with its holes

$\text{InnerComponents}(f)$
**Graph \( G \) construction**

Idea – during sweep line, we know the nearest left edge for every vertex \( v \) (and half-edge with origin \( v \))

1. Make node for every cycle (graph traversal)
2. During plane sweep,
   - store pointer to graph node for each edge
   - remember the leftmost vertex and its nearest left edge
3. Create arc between cycles of the leftmost vertex and its nearest left edge

Felkel: Computational geometry
(35 / 96)
Face label determination

For intersection \( v \) of two edges:
During the sweep-line
- In both new pieces, remember the face of half-edge being split into two
After
- Label the face by both labels

For face in other face:
Known half-edge label only from \( S_1 \)
Use graph \( G \) to locate outer boundary label for face from \( S_2 \)
(or store containing face \( f \) of other subdivision for each vertex)
Map overlay algorithm

MapOverlay($S_1, S_2$)

**Input:** Two planar subdivisions $S_1$ and $S_2$ stored in DCEL  \(\text{// complexity } n\)

**Output:** The overlay of $S_1$ and $S_2$ stored in DCEL $D$

1. Copy both DCELs for of $S_1$ and $S_2$ into DCEL $D$  \(\text{// } O(n)\)  \(\text{// } O(n \log n + k \log n)\)
2. Use plane sweep to compute intersections of edges from $S_1$ and $S_2$  \(\text{(intersection)}\)
   - Update vertex and edge records in $D$ when the event involves edges of both $S_1, S_2$
   - Store the half-edge to the left of the event point at the vertex in $D$
3. Traverse $D$ (depth-first search) to determine the boundary cycles  \(\text{// } O(n)\)
4. Construct the graph $G$ (boundary and hole cycles, immediately to the left of hole),
5. for each connected component in $G$ do
6. \(C\) $\leftarrow$ the unique outer boundary cycle
7. \(f\) $\leftarrow$ the face bounded by the cycle $C$.
8. Create a face record for $f$
9. \(OuterComponent(f)\) $\leftarrow$ some half-edge of $C$
10. \(InnerComponents(f)\) $\leftarrow$ list of pointers to one half-edge $e$ in each hole
11. \(IncidentFace(e)\) $\leftarrow$ $f$ for all half-edges bounding cycle $C$ and the holes
12. Label each face of $O(S_1, S_2)$ with the names of the faces of $S_1$ and $S_2$ containing it

Felkel: Computational geometry

(37 / 96)
Running time

The overlay of two planar subdivisions with total complexity $n$ can be constructed in

$$O(n \log n + k \log n)$$

where $k = \text{complexity of the overlay} \approx \text{intersections}$
Axis parallel rectangles intersection
Intersection of axis parallel rectangles

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

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

Alternate sides belong to two pencils of lines (trsy přímek)
(often used with points in infinity = axis parallel)
2D => 2 pencils

Felkel: Computational geometry (40 / 96)
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 at its left side or at its right side).

- **active rectangles** – a set
  - = rectangles currently intersecting the sweep line
  - **left side** event of a rectangle – start
    => the rectangle is added to the active set.
  - **right side** – end
    => the rectangle is deleted from the active set.

- The active set used to detect rectangle intersection
Example rectangles and sweep line

- Active rectangle
- Not active rectangle

sweep line

y

x
Interval tree as sweep line status structure

- Vertical sweep-line => only $y$-coordinates along it
- The status tree is drawn horizontal - turn $90^\circ$ right as if the sweep line ($y$-axis) is horizontal
Intersection test – between pair of intervals

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

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

  OR

  b) $y'_1 \leq y_1 \leq y'_2$

Intervals along the sweep line

Felkel: Computational geometry
(45 / 96)
Intersection test – between pair of intervals

Given two intervals \( I = [y_1, y_2] \) and \( I' = [y'_1, y'_2] \) the condition \( I \cap I' \) is equivalent to both of these conditions simultaneously:

1) \( y'_1 \leq y_2 \)

AND

2) \( y_1 \leq y'_2 \)

Intervals along the sweep line
Static interval tree – stores all end point $y_s$

- Let $v = y_{med}$ 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 $s_5$

$M_l = (s_4, s_6, s_1)$

$M_r = (s_1, s_4, s_6)$

Left ends – ascending

Right ends – descending

Interval tree on $s_2$ and $s_7$
Static interval tree [Edelsbrunner80]

- Stores intervals along y sweep line
- 3 kinds of information
  - end points
  - incident intervals
  - active nodes
Primary structure – static tree for endpoints

\[ v = \text{midpoint of all segment endpoints} \]
\[ H(v) = \text{value (y-coord) of } v \]

Static – known from beginning
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 currently intersected by the sweep line (nodes with intervals)
Entries in the event queue

(x_i, y_{il}, y_{ir}, t)

(x_1, 1, 3, left)
(x_2, 2, 4, left)
(x_3, 1, 3, right)
(x_4, 2, 4, right)

Static nodes in the SL status tree
1, 2, 3, 4
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 (x_i, y_{iL}, y_{iR}, t) from Q  // t ∈ {left | right }
4. if (t = left)  // left edge
   a) QueryInterval (y_{iL}, y_{iR}, root(T))  // report intersections
   b) InsertInterval (y_{iL}, y_{iR}, root(T))  // insert new interval
5. else  // right edge
6. c) DeleteInterval (y_{iL}, y_{iR}, root(T))
Preprocessing

Preproces(s S)

Input: Set S of rectangles
Output: Primary structure of the interval tree T and the event queue Q

1. T = 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 ~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, left), Q) // left edges of i-th rectangle
5. insert((x_iR, y_iL, y_iR, 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. $T = \text{BST}( S_y )$
3. return $T$

BST( $S_y$ )
1. if ( $|S_y| = 0$ ) return null
2. $y\text{Med} =$ median of $S_y$ // the smaller item for even $S_y$.size
3. $L =$ endpoints $p_y \leq y\text{Med}$
4. $R =$ endpoints $p_y > y\text{Med}$
5. $t =$ new IntervalTreeNode( $y\text{Med}$ )
6. $t.\text{left} =$ BST(L)
7. $t.\text{right} =$ 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) // jump to active
6. QueryInterval(b, e, T.RPTR) // node below
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)

DCCI

New interval being tested for intersection
Stored intervals of active rectangles
Crosses A, B
Crosses A, B, C
Crosses B
Crosses C
Crosses A, B, C
Crosses nothing
Interval tree - interval insertion

insertInterval \((b, e, T)\)

**Input:** Interval \([b,e]\) and interval tree \(T\)

**Output:** \(T\) after insertion of the interval

1. \(v = \text{root}(T)\)
2. while \((v \neq \text{null})\) // find the fork node
3. \hspace{1em} if \((H(v) < b < e)\)
4. \hspace{2em} \(v = v.\text{right}\) // continue right
5. \hspace{1em} else if \((b < e < H(v))\)
6. \hspace{2em} \(v = v.\text{left}\) // continue left
7. \hspace{1em} else // \(b \leq H(v) \leq e\) // insert interval
8. \hspace{2em} set \(v\) node to active
9. \hspace{2em} connect \(\text{LPTR}\) resp. \(\text{RPTR}\) to its parent (active node above)
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 \(T\)

\(\bigstar\) DCGI
Example 1
Example 1 – static tree on endpoints

H(ν) – value of node ν

A

B

H(v) – value of node v

Felkel: Computational geometry

(Drtina)
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

\[ 1 < 2 < 3 \]
Interval insertion \([1,3]\)  

b) Insert Interval 

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

\(? 1 \leq 2 \leq 3 ? \)
Interval insertion $[1,3]$   b) Insert Interval

$1 \leq 2 \leq 3$

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

fork => to lists

Active rectangle
Current node
Active node

Felkel: Computational geometry
(63 / 96)
Interval insertion [2,4]  a) Query Interval

Search MR(v) only:

MR(v)[1] = 3 ≥ 2?

=> intersection

H(v) ≤ b < e

② ≤ 2 < 4
Interval insertion $[2,4]$  

b) Insert Interval

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

\[ 2 \leq 2 \leq 4 \]

fork

=> to lists

Active rectangle

Current node

Active node

Felkel: Computational geometry

(65 / 96)
Interval delete \([1,3]\)
Interval delete $[1,3]$
Interval delete [2,4]
Interval delete [2,4]
Query = sweep and report intersections

\textbf{RectangleIntersections}( S )

\textbf{Input:} Set \( S \) of rectangles

\textbf{Output:} Intersected rectangle pairs

1. Preprocess( \( S \) ) \hspace{1cm} // create the interval tree \( T \) (for \( y \)-coords)
   \hspace{1cm} // and event queue \( Q \) (for \( x \)-coords)
2. \textbf{while} ( \( Q \neq \emptyset \) ) \textbf{do}
3. \hspace{1cm} Get next entry (\( x_i, y_{iL}, y_{iR}, t \)) from \( Q \) \hspace{1cm} // \( t \in \{ \text{left} | \text{right} \} \)
4. \hspace{1cm} \textbf{if} ( \( t = \text{left} \) ) \hspace{1cm} // left edge
5. \hspace{1.5cm} a) \textbf{QueryInterval} (\( y_{iL}, y_{iR}, \text{root}(T) \)) \hspace{1cm} // report intersections
6. \hspace{1cm} b) \textbf{InsertInterval} (\( y_{iL}, y_{iR}, \text{root}(T) \)) \hspace{1cm} // insert new interval
7. \hspace{1cm} \textbf{else} \hspace{1cm} // right edge
8. \hspace{1.5cm} c) \textbf{DeleteInterval} (\( y_{iL}, y_{iR}, \text{root}(T) \))

\textit{// this is a copy of the slide before}
\textit{// just to remember the algorithm}
Example 2 – tree created by PrimaryTree(S)
Example 2 – slightly unbalanced tree
Insert $[2,3] - $ empty $\Rightarrow$ b) Insert Interval

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

Insert the new interval to secondary lists

? $2 \leq 3 \leq 3$ ?

fork node $\Rightarrow$ active

$\Rightarrow$ to lists
**Insert** $[3,7]$  

a) Query Interval

For all $i$ in $\text{MR}(v)$ test $\text{MR}(v)[i] \geq 3$  

=> report intersection $c$

go right, nil, stop

$H(v) \leq b < e$

? $3 \leq 3 < 7$ ?
**Insert** $[3, 7]$  

b) **Insert Interval**

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

---

Insert the new interval to secondary lists

- **fork node** $\Rightarrow$ **active**
- $3 \leq 3 \leq 7$
- $\Rightarrow$ to lists

---

[Diagram showing active nodes, rectangles, and intervals]

- Active rectangle
- Current node
- Active node

Felkel: Computational geometry

(76 / 96)
Insert $[0,2]$ a) Query Interval

for (all in ML(v)) test $ML(v).[i] \leq 2$

$\Rightarrow$ report intersection c

go left, nil, stop

$b < e \leq H(v)$

? $0 < 2 \leq 3$?
Insert $[0, 2]$  

b) Insert Interval $1/2$  

$b < e < H(v)$  

? $0 < 2 < 3$?  

$\Rightarrow$ insert left
Insert $[0, 2]$ b) Insert Interval 2/2

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

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

- Fork node $\Rightarrow$ active
- $\Rightarrow$ to lists

Felkel: Computational geometry (79 / 96)
**Insert** \([1,5]\)  

a) **Query Interval** 1/2

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

\[\text{for (all in } MR(v)\text{)}\]

\[\Rightarrow \text{report intersection } c,d\]

- go left \(-\rightarrow 1\)
- go right \(-\rightarrow \text{nil}\)

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

**a) Query Interval 2/2**

for (all in MR(v)) test MR(v)[i] ≥ 1

=> report intersection a

go right, nil, stop

\[ H(v) \leq b < e \]
Insert $[1,5]$ b) Insert Interval

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

Insert the new interval to secondary lists

$1 \leq 3 \leq 5$?
Insert [7,8]  a) Query Interval

for (all in \(MR(v)\)) test \(MR(v).[i] \geq 7\) => report intersection d
go right, nil, stop

\(H(v) \leq b < e\)
Insert [7, 8] b) Insert Interval

Insert the new interval to secondary lists
link to parent

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

b ≤ H(v) ≤ e

Felkel: Computational geometry
(84/96)

Delete the interval $[3, 7]$ from secondary lists.
**Insert** $[4,6]$  

a) **Query Interval**

for (all in $\text{MR}(v)$) test $\text{MR}(v)[i] \geq 4$ => report intersection $b$

for (all in $\text{ML}(v)$) test $\text{ML}(v)[i] \leq 6$

=> no intersection

$H(v) \leq b < e$
**Insert** $[4, 6]$  

b) *Insert Interval*

$H(v) \leq b < e$

Insert the new interval to secondary lists

---

Felkel: Computational geometry  
(87 / 96)
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 $[0,2]$

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

$0 < 2 \leq 3$

Felkel: Computational geometry

(Drtina)
Delete $[0,2]$ Delete Interval 2/2

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

$0 \leq 1 \leq 2$?

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

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

$\begin{align*}
\text{Current node} & \quad \text{Active node} \\
0 & \quad 1 & \quad 2 & \quad 3 & \quad 4 & \quad 5 & \quad 6 & \quad 7 & \quad 8
\end{align*}$

$\begin{align*}
av & \quad b & \quad c & \quad d & \quad e & \quad f
\end{align*}$

$\begin{align*}
\text{Active rectangle} & \quad \text{Current node}
\end{align*}$

$\begin{align*}
b \leq H(v) \leq e
\end{align*}$

$\begin{align*}
?3 \leq 7 < 8?
?5 \leq 7 < 8?
?7 \leq 7 \leq 8?
\end{align*}$
Delete $[2,3]$ Delete Interval

Search for and delete node with interval $[2,3]$.
Delete $[4,6]$ Delete Interval

Search for and delete node with interval $[4,6]$
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


