

# Line Segments

Alexander Gran

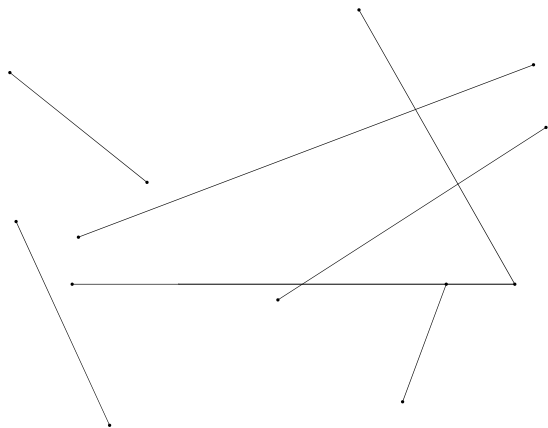
Datenstrukturen und Algorithmen

SS 2005

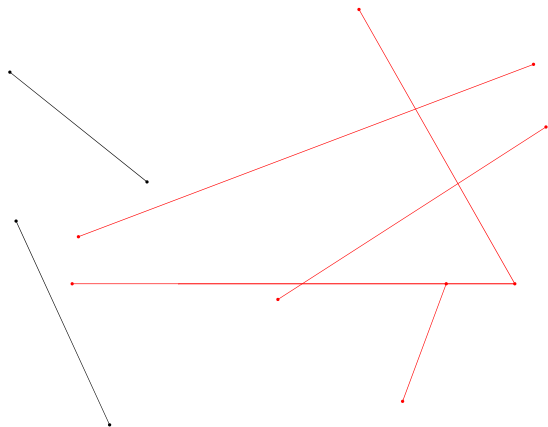
Problemstellung:

Gegeben: Eine Menge Linien, gesucht die sich schneidenden Linien.

# Problem



# Lösung



## Formalismus

Die Punkte liegen für gewöhnlich als Koordinatentupel  $(x,y)$  vor.

## Formalismus

Die Punkte liegen für gewöhnlich als Koordinatentupel  $(x,y)$  vor.  
Linien typischerweise als Anfangs- und Endpunkt.

## Formalismus

Die Punkte liegen für gewöhnlich als Koordinatentupel  $(x,y)$  vor.

Linien typischerweise als Anfangs- und Endpunkt.

Alternativ auch als Vektoren, Gerade etc. Die Definition der Linie über eine convexe Hülle der beiden Endpunkte findet sich auch.

## Formalismus

Die Punkte liegen für gewöhnlich als Koordinatentupel  $(x,y)$  vor.

Linien typischerweise als Anfangs- und Endpunkt.

Alternativ auch als Vektoren, Gerade etc. Die Definition der Linie über eine convexe Hülle der beiden Endpunkte findet sich auch.

Mit Abstand ist hier der euklidische Abstand gemeint

$$(\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}).$$



Wann schneiden sich zwei Linien?

Wann schneiden sich zwei Linien?

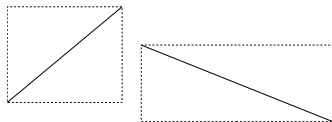
- ▶ Umgebende Rechtecke müssen überlappen

Wann schneiden sich zwei Linien?

- ▶ Umgebende Rechtecke müssen überlappen
- ▶ Start- und Endpunkt der einen Linie müssen auf verschiedenen Seiten der andern liegen

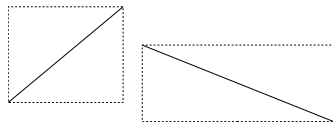
## Umgebende Rechtecke:

Schnitt unmöglich

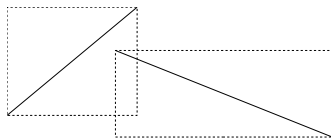


## Umgebende Rechtecke:

Schnitt unmöglich

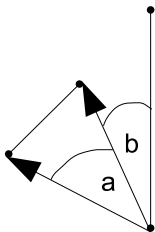


Schnitt möglich



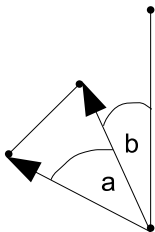
## Seitentest

Kein Schnitt:  $a < 0 \wedge b < 0$

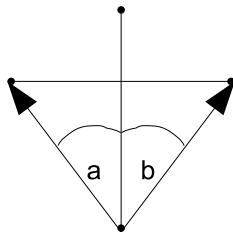


## Seitentest

Kein Schnitt:  $a < 0 \wedge b < 0$

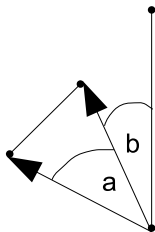


Schnitt:  $a < 0 \wedge b > 0$

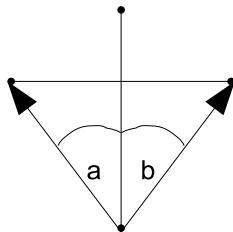


## Seitentest

Kein Schnitt:  $a < 0 \wedge b < 0$



Schnitt:  $a < 0 \wedge b > 0$



Generell: Verschiedene Vorzeichen von a und b



## Seitentest Implementierung

Implementierung von Sun, Java 5 Tiger

```
1 public static int relativeCCW(double X1, double Y1,
2     double X2, double Y2,
3     double PX, double PY) {
4     X2 -= X1;
5     Y2 -= Y1;
6     PX -= X1;
7     PY -= Y1;
8     double ccw = PX * Y2 - PY * X2;
9     if (ccw == 0.0) {
10        ccw = PX * X2 + PY * Y2;
11        if (ccw > 0.0) {
12            PX -= X2;
13            PY -= Y2;
14            ccw = PX * X2 + PY * Y2;
15            if (ccw < 0.0) {
16                ccw = 0.0;
17            }
18        }
19    }
20    return (ccw < 0.0) ? -1 : ((ccw > 0.0) ? 1 : 0);
}
```

## Laufzeitanalyse Schnittpunkttest

Rechtecküberlappung:  $O(1)$

## Laufzeitanalyse Schnittpunkttest

Rechtecküberlappung:  $O(1)$

Seitentest:  $O(1)$

## Laufzeitanalyse Schnittpunkttest

Rechtecküberlappung:  $O(1)$

Seitentest:  $O(1)$

Gesamt:  $O(1)$

Alle Linien miteinander vergleichen:

Alle Linien miteinander vergleichen:

```
1 TreeSet<TwoLines> result = new TreeSet<TwoLines>(new
    TwoLineComparatorX());
2 for (int i=0; i!= lines.size(); i++) {
3     for (int j=0; j!= lines.size(); j++) {
4         if (i != j) {
5             if (lines.get(i).intersects(lines.get(j))) {
6                 result.add(new
                    TwoLines(lines.get(i), lines.get(j)));
7             }
8         }
9     }
10 }
```

## Problem

Bei  $n$  Linen  $n^2$  Vergleiche.

## Problem

Bei  $n$  Linien  $n^2$  Vergleiche.

Übliche Datenmenge:  $10^6$  Linien  $\rightarrow$  10.000.000.000.000 Vergleiche.



## Sweeping Idee

Je näher Linien bei einander liegen, desto eher schneiden sie sich.

## Sweeping Idee

Je näher Linien bei einander liegen, desto eher schneiden sie sich.  
Ursprung: Rechtecküberlappung

## Sweeping Idee

Je näher Linien bei einander liegen, desto eher schneiden sie sich.

Ursprung: Rechtecküberlappung

Definition der Nähe:

Totale Ordnung in X Richtung, dann in Y Richtung.

## Vorgehen

Eine gedachte Sweepline wandert - swept - in X Richtung - in aller Regel also von links nach rechts - über die Linien.

## Vorgehen

Eine gedachte Sweepline wandert - swept - in X Richtung - in aller Regel also von links nach rechts - über die Linien. Dabei werden möglichst wenig Linien untereinander auf Schnitt getestet.

## Implementierung

Implementierung mittels Comparator:

```
1 public class PointComparatorXY implements Comparator<Point>{
2     public int compare(Point a, Point b) {
3         double dif = a.getX()-b.getX();
4         double dif2 = a.getY()-b.getY();
5         if (dif > 0) {
6             return 1;
7         } else if (dif < 0) {
8             return -1;
9         } else {
10            if (dif2 > 0) {
11                return 1;
12            } else if (dif2 < 0) {
13                return -1;
14            } else {
15                return 0;
16            }
17        }
18    }
19 }
```

## Laufzeitanalyse

Ein Vergleich:  $O(1)$

## Laufzeitanalyse

Ein Vergleich:  $O(1)$

Damit sortierung von  $n$  Punkt wie üblich in  $O(n \log(n))$   
mittels autobalancierenden Baum oder QuickSort.



## Laufzeitanalyse

Ein Vergleich:  $O(1)$

Damit sortierung von  $n$  Punkt wie üblich in  $O(n \log(n))$   
mittels autobalancierenden Baum oder QuickSort.

Hier wird ein Red-Black Baum aus der API verwendet  
(`java.util.TreeMap` bzw `java.util.TreeSet`)

## Ereignis-Punkte

Die Sweepline "hält" nur an interessanten Punkten an.

## Ereignis-Punkte

Die Sweepline "hält" nur an interessanten Punkten an.  
Interessant sind:

- ▶ Startpunkte von Linien

## Ereignis-Punkte

Die Sweepline "hält" nur an interessanten Punkten an.  
Interessant sind:

- ▶ Startpunkte von Linien
- ▶ Endpunkte von Linien

## Ereignis-Punkte

Die Sweepline "hält" nur an interessanten Punkten an.  
Interessant sind:

- ▶ Startpunkte von Linien
- ▶ Endpunkte von Linien
- ▶ Schnittpunkte von Linien

## Ereignis-Punkte

Die Sweepline "hält" nur an interessanten Punkten an.  
Interessant sind:

- ▶ Startpunkte von Linien
- ▶ Endpunkte von Linien
- ▶ Schnittpunkte von Linien

Schnittpunkte müssen berechnet werden!

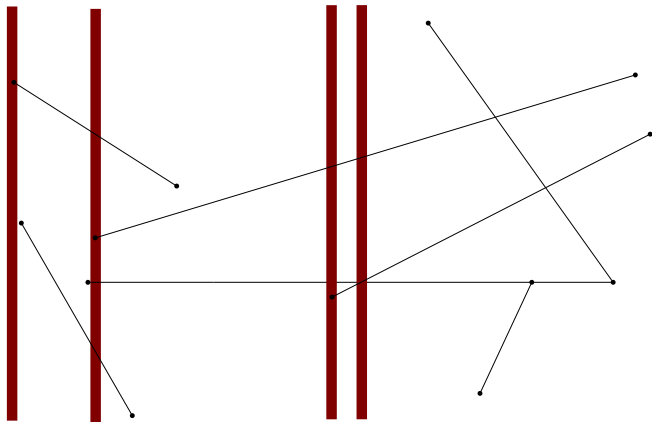
## Ereignis-Punkte

Die Sweepline "hält" nur an interessanten Punkten an.  
Interessant sind:

- ▶ Startpunkte von Linien
- ▶ Endpunkte von Linien
- ▶ Schnittpunkte von Linien

Schnittpunkte müssen berechnet werden!  
Berechnung muss on the fly erfolgen.

## Ereignis-Punkte





## Datenstrukturen

Es werden benötigt:

- ▶ *schedule* Eine sortierte Map, die die noch zu verarbeitenden Punkte enthält, und auf ihre Linie(n) abbildet.

## Datenstrukturen

Es werden benötigt:

- ▶ *schedule* Eine sortierte Map, die die noch zu verarbeitenden Punkte enthält, und auf ihre Linie(n) abbildet. Darin müssen die Schnittpunkte während der Berechnung eingefügt werden.

## Datenstrukturen

Es werden benötigt:

- ▶ *schedule* Eine sortierte Map, die die noch zu verarbeitenden Punkte enthält, und auf ihre Linie(n) abbildet.  
Darin müssen die Schnittpunkte während der Berechnung eingefügt werden.  
Hier: `java.util.TreeMap`

## Datenstrukturen

Es werden benötigt:

- ▶ *schedule* Eine sortierte Map, die die noch zu verarbeitenden Punkte enthält, und auf ihre Linie(n) abbildet.  
Darin müssen die Schnittpunkte während der Berechnung eingefügt werden.  
Hier: `java.util.TreeMap`
- ▶ *status* Eine sortierte Menge der Linien, die gerade unter der Sweepline liegen

## Datenstrukturen

Es werden benötigt:

- ▶ *schedule* Eine sortierte Map, die die noch zu verarbeitenden Punkte enthält, und auf ihre Linie(n) abbildet.  
Darin müssen die Schnittpunkte während der Berechnung eingefügt werden.  
Hier: `java.util.TreeMap`
- ▶ *status* Eine sortierte Menge der Linien, die gerade unter der Sweepline liegen  
Hier `java.util.TreeSet`

## Datenstrukturen

Es werden benötigt:

- ▶ *schedule* Eine sortierte Map, die die noch zu verarbeitenden Punkte enthält, und auf ihre Linie(n) abbildet.  
Darin müssen die Schnittpunkte während der Berechnung eingefügt werden.  
Hier: `java.util.TreeMap`
- ▶ *status* Eine sortierte Menge der Linien, die gerade unter der Sweepline liegen  
Hier `java.util.TreeSet`
- ▶ *result* Eine Menge der sich schneidenden Linien

## Datenstrukturen

Es werden benötigt:

- ▶ *schedule* Eine sortierte Map, die die noch zu verarbeitenden Punkte enthält, und auf ihre Linie(n) abbildet.  
Darin müssen die Schnittpunkte während der Berechnung eingefügt werden.  
Hier: `java.util.TreeMap`
- ▶ *status* Eine sortierte Menge der Linien, die gerade unter der Sweepline liegen  
Hier `java.util.TreeSet`
- ▶ *result* Eine Menge der sich schneidenden Linien  
Hier `java.util.HashSet`

## Datenstrukturen

Es werden benötigt:

- ▶ *schedule* Eine sortierte Map, die die noch zu verarbeitenden Punkte enthält, und auf ihre Linie(n) abbildet.  
Darin müssen die Schnittpunkte während der Berechnung eingefügt werden.  
Hier: `java.util.TreeMap`
- ▶ *status* Eine sortierte Menge der Linien, die gerade unter der Sweepline liegen  
Hier `java.util.TreeSet`
- ▶ *result* Eine Menge der sich schneidenden Linien  
Hier `java.util.HashSet`

Alle diese unterstützen die nötigen Operationen (insert, delete, getNext etc.) in  $O(\log(n))$



## Der eigentliche Algorithmus

- ▶ Startpunkt

## Der eigentliche Algorithmus

- ▶ Startpunkt  
Linie des Punktes in *status* einfügen.

## Der eigentliche Algorithmus

- ▶ Startpunkt  
Linie des Punktes in *status* einfügen.  
Schnitttest mit Vorgänger und Nachfolger.

## Der eigentliche Algorithmus

- ▶ Startpunkt  
Linie des Punktes in *status* einfügen.  
Schnitttest mit Vorgänger und Nachfolger.
- ▶ Endpunkt

## Der eigentliche Algorithmus

- ▶ Startpunkt  
Linie des Punktes in *status* einfügen.  
Schnitttest mit Vorgänger und Nachfolger.
- ▶ Endpunkt  
Linie des Punktes aus *status* entfernen.

## Der eigentliche Algorithmus

- ▶ Startpunkt  
Linie des Punktes in *status* einfügen.  
Schnitttest mit Vorgänger und Nachfolger.
- ▶ Endpunkt  
Linie des Punktes aus *status* entfernen.  
Es entstehen zwei neue Nachbarn, Vorgänger und Nachfolger der Linie. Schnitttest dieser Beiden.

## Der eigentliche Algorithmus

- ▶ Startpunkt  
Linie des Punktes in *status* einfügen.  
Schnitttest mit Vorgänger und Nachfolger.
- ▶ Endpunkt  
Linie des Punktes aus *status* entfernen.  
Es entstehen zwei neue Nachbarn, Vorgänger und Nachfolger der Linie. Schnitttest dieser Beiden.

Bei jedem positivem Schnitttest werden die sich schneidenden in *result* gespeichert;

## Der eigentliche Algorithmus

- ▶ Startpunkt  
Linie des Punktes in *status* einfügen.  
Schnitttest mit Vorgänger und Nachfolger.
- ▶ Endpunkt  
Linie des Punktes aus *status* entfernen.  
Es entstehen zwei neue Nachbarn, Vorgänger und Nachfolger der Linie. Schnitttest dieser Beiden.

Bei jedem positivem Schnitttest werden die sich schneidenden in *result* gespeichert; Der Schnittpunkt kommt als neuer Ereignis-Punkt in *schedule*. Er verweist auf die schneidenden Linien.



## Der eigentliche Algorithmus

- ▶ Schnittpunkt

## Der eigentliche Algorithmus

- ▶ Schnittpunkt  
Hier tauschen die Schneidenden ihre Position in *status*. Daher bekommen Beide einen neuen Nachbarn → zwei Schnittpunkte.

# Implementierung

## Initialisierung

```
1 private TreeSet<Line> status;  
2  
3 private SortedMap<Point, Object> schedule;  
4  
5 private HashSet<TwoLines> result;  
6  
7 private LineComparatorY comp;  
8  
9 public TwoLines[] getLineSegments( ArrayList<Line> lines) {  
10     schedule = getSortedEndPoints( lines);  
11     comp = new LineComparatorY();  
12     status = new TreeSet<Line>(comp);  
13     result = new HashSet<TwoLines>(lines.size()/10);
```

## Implementierung

### Hauptschleife

```
14     Point p = schedule.firstKey();
15     while(p != null) {
16         Object o = schedule.get(p);
17         if(o instanceof Line) { //endpoint
27     } else { //intersection
47         p = nextEPoint();
48         comp.x = p.getX();
```

## Implementierung

### Endpunkt

```
16         Object o = schedule.get(p);
17         if(o instanceof Line) { //endpoint
18             Line l = (Line)o;
19             if(l.getLeft() == p) { //left endpoint
20                 status.add(l);
21                 testIntersect(getPrev(l), l);
22                 testIntersect(getNext(l), l);
23             } else { //right endpoint
24                 testIntersect(getPrev(l), getNext(l));
25                 status.remove(l);
26             }
27         }
```

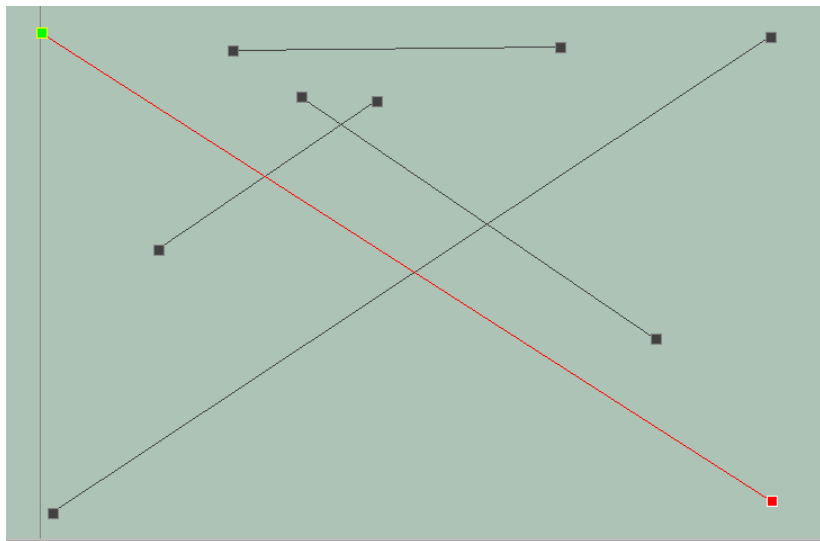
## Implementierung

### Schnittpunkt

```
28         TwoLines l = (TwoLines)o;
29         comp.x -= DELTA; //ensuring order
30         Line beforeUpper;
31         Line beforeLower;
32         if (comp.compare(l.getA(), l.getB()) < 0) {
33             beforeUpper = l.getA();
34             beforeLower = l.getB();
35         } else {
36             beforeUpper = l.getB();
37             beforeLower = l.getA();
38         }
39         Line prev = getPrev(beforeUpper);
40         Line next = getNext(beforeLower);
41         status.remove(beforeUpper);
42         comp.x += 3*DELTA; //ensuring order
43         status.add(beforeUpper);
44         testIntersect(prev, beforeLower);
45         testIntersect(next, beforeUpper);
```

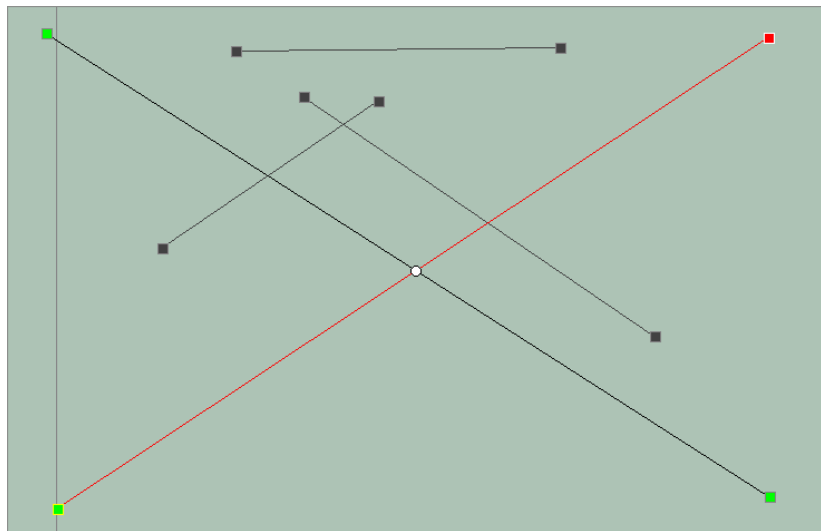
# Line Segments

## └ Beispieldurchlauf

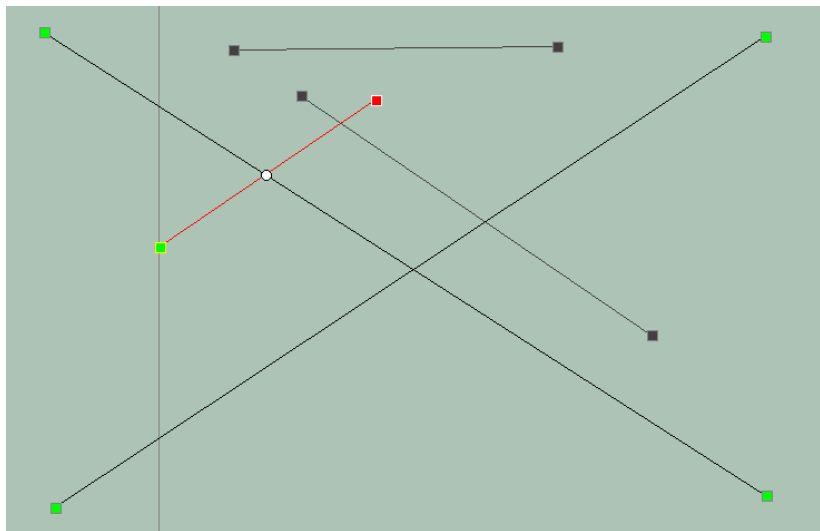


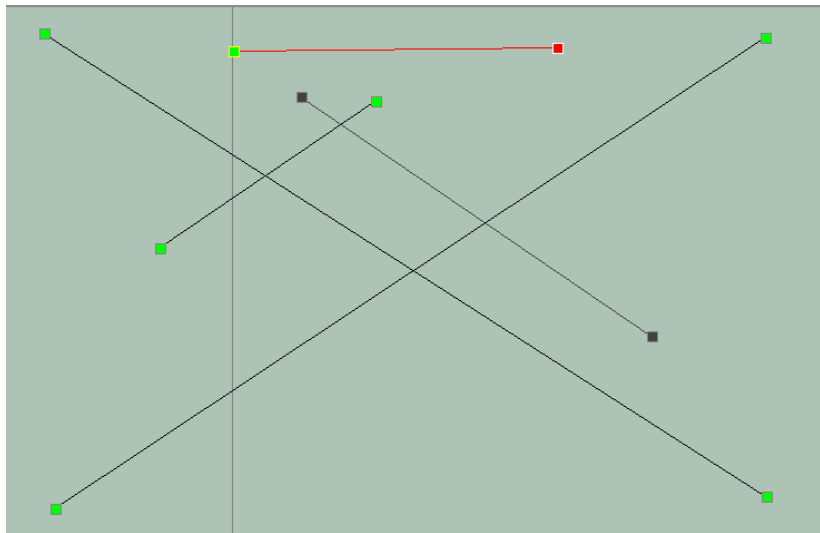
# Line Segments

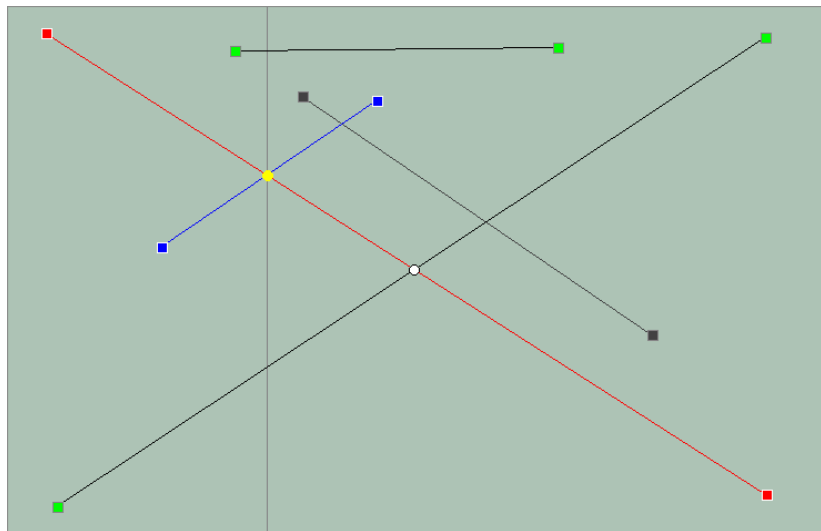
└ Beispieldurchlauf





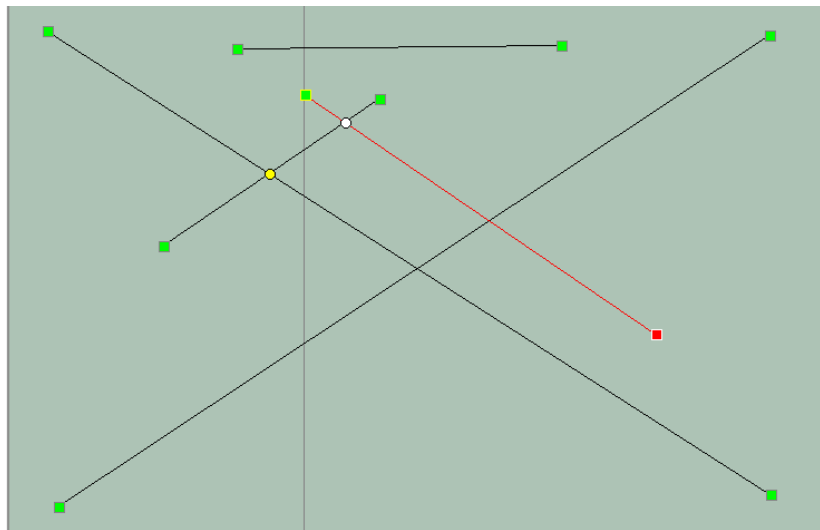


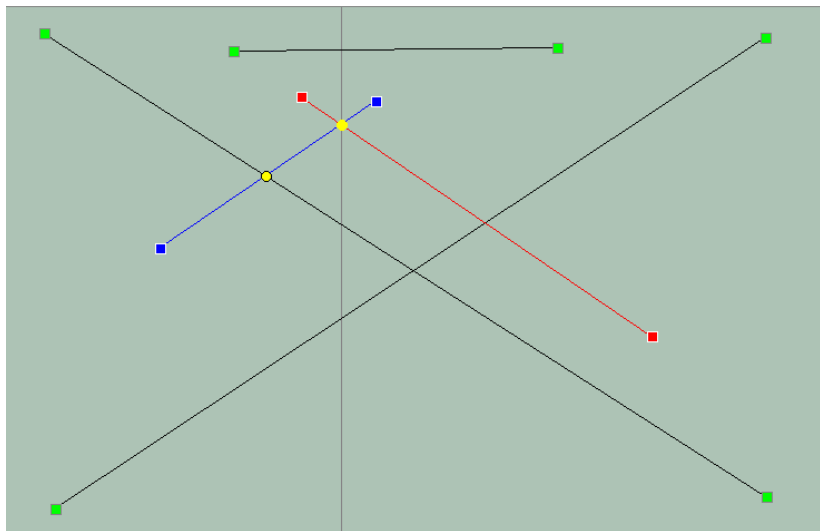


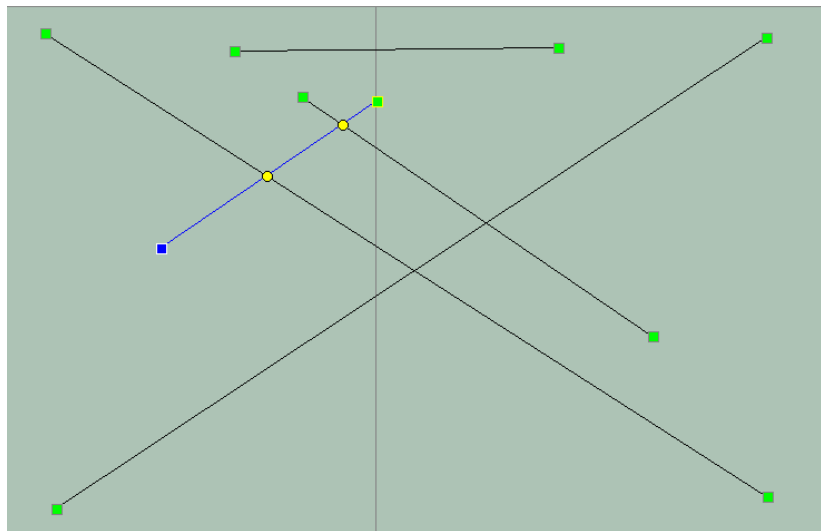


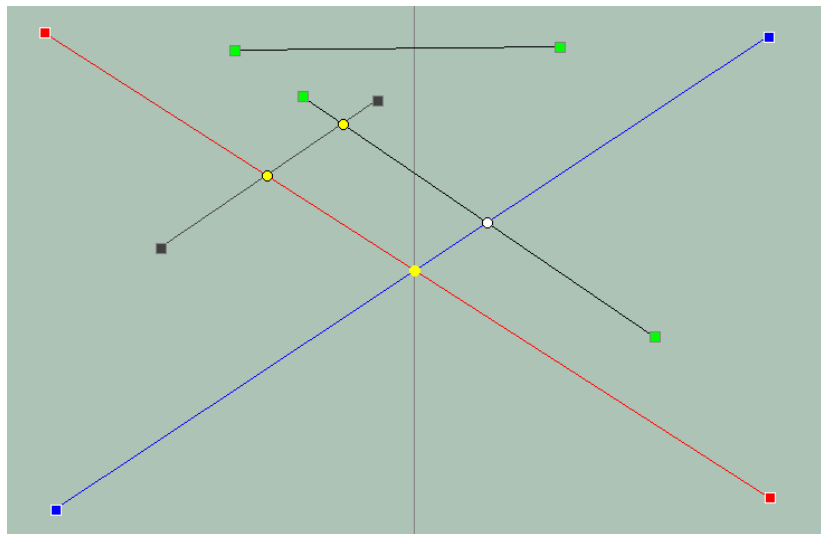
# Line Segments

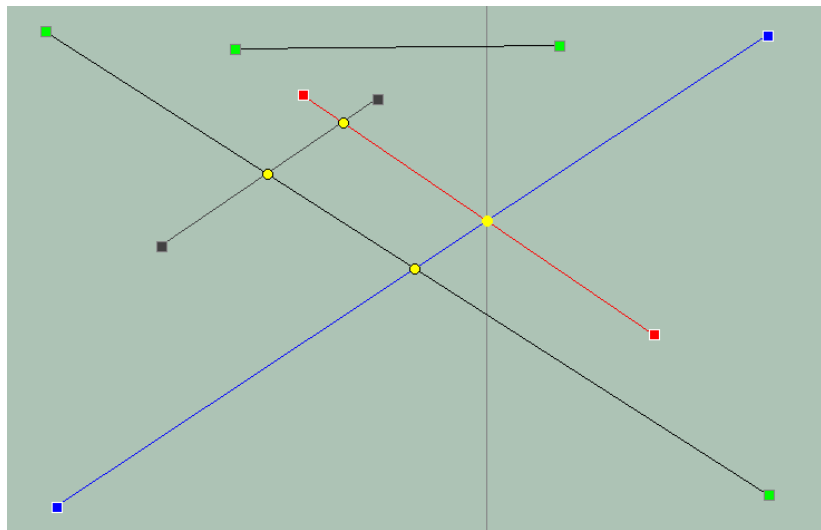
└ Beispieldurchlauf













## Laufzeitanalyse

Seien  $n$  die Anzahl der Linien,  $k$  die Anzahl der sich Schneidenden.

► Vorsortieren

```
54 private TreeMap<Point, Object>
    getSortedEndPoints( ArrayList<Line> lines) {
55     TreeMap<Point, Object> a = new
        TreeMap<Point, Object>(new PointComparatorXY());
56     for (Line l : lines) {
57         a.put(l.getStart(), l);
58         a.put(l.getEnd(), l);
59     }
60     return a;
61 }
```

## Laufzeitanalyse

Seien  $n$  die Anzahl der Linien,  $k$  die Anzahl der sich Schneidenden.

► Vorsortieren

```
54 private TreeMap<Point, Object>
    getSortedEndpoints( ArrayList<Line> lines ) {
55     TreeMap<Point, Object> a = new
        TreeMap<Point, Object>(new PointComparatorXY());
56     for (Line l : lines) {
57         a.put(l.getStart(), l);
58         a.put(l.getEnd(), l);
59     }
60     return a;
61 }
```

Einfügen in *schedule*. Jedes Einfügen ist in  $O(\log(n))$ , es werden  $n$  Linien eingefügt

## Laufzeitanalyse

Seien  $n$  die Anzahl der Linien,  $k$  die Anzahl der sich Schneidenden.

► Vorsortieren

```
54 private TreeMap<Point, Object>
    getSortedEndPoints( ArrayList<Line> lines ) {
55     TreeMap<Point, Object> a = new
        TreeMap<Point, Object>(new PointComparatorXY());
56     for (Line l : lines) {
57         a.put(l.getStart(), l);
58         a.put(l.getEnd(), l);
59     }
60     return a;
61 }
```

Einfügen in *schedule*. Jedes Einfügen ist in  $O(\log(n))$ , es werden  $n$  Linien eingefügt  
Gesamt  $O(n \log(n))$

# Laufzeitanalyse

- ▶ Hauptschleife

## Laufzeitanalyse

► Hauptschleife

Insgesamt  $2 * n + k$  Durchläufe. In jedem wird ein neuer Punkt aus *schedule* geholt.

```
63 private Point nextEPoint() {
64     //java api can be ugly
65     Iterator<Point> pit = schedule.keySet().iterator();
66     if (pit.hasNext()) pit.next();
67     if (pit.hasNext()) {
68         schedule = schedule.tailMap(pit.next());
69         return schedule.firstKey();
70     } else {
71         return null;
72     }
73 }
```

## Laufzeitanalyse

► Hauptschleife

Insgesamt  $2 * n + k$  Durchläufe. In jedem wird ein neuer Punkt aus *schedule* geholt.

```
63 private Point nextEPoint() {
64     //java api can be ugly
65     Iterator<Point> pit = schedule.keySet().iterator();
66     if (pit.hasNext()) pit.next();
67     if (pit.hasNext()) {
68         schedule = schedule.tailMap(pit.next());
69         return schedule.firstKey();
70     } else {
71         return null;
72     }
73 }
```

Holen ist in  $O(\log(n))$ , also auch wieder  $O(n \log(n))$

## Laufzeitanalyse Hauptschleife

### ► Endpunkt

```
19         if (l.getLeft() == p) { // left endpoint
20             status.add(l);
21             testIntersect(getPrev(l), l);
22             testIntersect(getNext(l), l);
23         } else { // right endpoint
24             testIntersect(getPrev(l), getNext(l));
25             status.remove(l);
26         }
```

## Laufzeitanalyse Hauptschleife

### ► Endpunkt

```
19         if (l.getLeft() == p) { //left endpoint
20             status.add(l);
21             testIntersect(getPrev(l), l);
22             testIntersect(getNext(l), l);
23         } else { //right endpoint
24             testIntersect(getPrev(l), getNext(l));
25             status.remove(l);
26         }
```

Einfügen, Löschen und Herrausholen aus *status* ist in  $O(\log(n))$ .  
Schnitttest ist in  $O(1)$ .



## Laufzeitanalyse Hauptschleife

### ► Schnittpunkt

```
39         Line prev = getNext(beforeUpper);  
40         Line next = getNext(beforeLower);  
41         status.remove(beforeUpper);  
42         comp.x += 3*DELTA; //ensuring order  
43         status.add(beforeUpper);  
44         testIntersect(prev, beforeLower);  
45         testIntersect(next, beforeUpper);
```

## Laufzeitanalyse Hauptschleife

### ► Schnittpunkt

```
39         Line prev = getNext(beforeUpper);
40         Line next = getNext(beforeLower);
41         status.remove(beforeUpper);
42         comp.x += 3*DELTA; //ensuring order
43         status.add(beforeUpper);
44         testIntersect(prev, beforeLower);
45         testIntersect(next, beforeUpper);
```

Einfügen, Löschen und Herausheben aus *status* ist in  $O(\log(n))$ .

Schnitttest ist in  $O(1)$ .

## Laufzeitanalyse Abschluss

Pro Durchlauf nie schlechter als  $O(\log(n))$ .

## Laufzeitanalyse Abschluss

Pro Durchlauf nie schlechter als  $O(\log(n))$ .

Bei  $2 * n + k$  Durchläufen ist das  $O((2 * n + k) * \log(n))$ .

## Laufzeitanalyse Abschluss

Pro Durchlauf nie schlechter als  $O(\log(n))$ .

Bei  $2 * n + k$  Durchläufen ist das  $O((2 * n + k) * \log(n))$ .

$k$  ist max  $n^2$ , in der Regel ist aber  $n \gg k$

## Ergebnis und Ausblick

Der Algorithmus ist optimal bezüglich des Problems.

## Ergebnis und Ausblick

Der Algorithmus ist optimal bezüglich des Problems.

Das Ausarten zu  $O(n^2)$  ist nicht zu vermeiden wenn man alle Schnittpunkte ausgeben will.

## Ergebnis und Ausblick

Der Algorithmus ist optimal bezüglich des Problems.

Das Ausarten zu  $O(n^2)$  ist nicht zu vermeiden wenn man alle Schnittpunkte ausgeben will.

Trivialerweise tritt das bei der Prüfung nicht auf.



## Weitere Varianten

In höherdimensionalen Räume ist das selbe Verfahren anwendbar, im 3 dimensionalen wird beispielsweise eine Sweep Ebene verwendet.



Introduction to Algorithms. MIT Press 2001.



Computational Geometry: An Introduction. Springer-Verlag, 1985.



<http://www.lupinho.de/gishur/html/Sweeps.html>

Den Sourcecode der Algorithmen gibt es online unter  
<http://zodiac.dnsalias.org/misc/ProSemDastru.tar.gz>