

# Datenstrukturen und Algorithmen

## Vorlesung 21: Algorithmische Geometrie

Joost-Pieter Katoen

Lehrstuhl für Informatik 2  
Software Modeling and Verification Group

<http://www-i2.informatik.rwth-aachen.de/i2/dsal12/>

6. Juli 2012



## Übersicht

### 1 Algorithmische Geometrie

- Winkelbestimmung
- Schnitt zweier Strecken

### 2 Schnitt eines beliebigen Streckenpaares

- Ordnen von Strecken
- Sweepline

### 3 Konvexe Hülle

## Übersicht

### 1 Algorithmische Geometrie

- Winkelbestimmung
- Schnitt zweier Strecken

### 2 Schnitt eines beliebigen Streckenpaares

- Ordnen von Strecken
- Sweepline

### 3 Konvexe Hülle

## Einführung

- ▶ Allgemein: Geometrische Probleme im  $n$ -dimensionalen Raum  $\mathbb{R}^n$ .
- ▶ Z. B. Schneiden sich zwei Geraden? etc.
- ▶ Wir betrachten hier Probleme im **zweidimensionalen** Raum, also  $n = 2$ .
- ▶ Dazu nutzen wir Konzepte aus der **Linearen Algebra**.
- ▶ Anwendungen: Computergraphik, CAD, Robotertechnik, usw.

# Mathematische Hilfsmittel

## Vektor, Skalarprodukt, Betrag, Determinante

- **Vektor** (im  $\mathbb{R}^n$ , insbesondere  $n = 2$ ):  $\vec{x} = \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$ .
- **Skalarprodukt** (Dot Product) von  $\vec{x}$  und  $\vec{y}$ :  

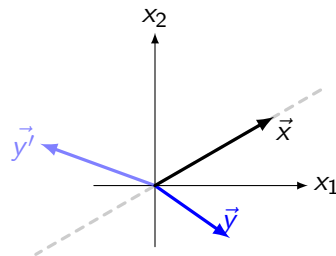
$$\vec{x} \cdot \vec{y} = \sum_{i=1}^n x_i \cdot y_i = x_1 y_1 + x_2 y_2.$$
- **Betrag** (Länge) von  $\vec{x}$ :  $|\vec{x}| = \sqrt{\vec{x} \cdot \vec{x}} = \sqrt{x_1^2 + x_2^2}$ .
- **Determinante** für  $[\vec{x}, \vec{y}] = \begin{bmatrix} x_1 & y_1 \\ x_2 & y_2 \end{bmatrix} =: A$ :  

$$\det A = \det(\vec{x}, \vec{y}) = x_1 y_2 - x_2 y_1.$$

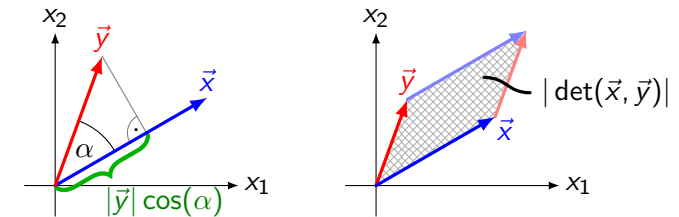
# Winkelbestimmung (I)

## Problem

Liegt ein Vektor  $\vec{y}$  *links* oder *rechts* von einem gegebenen Vektor  $\vec{x}$ ?



# Geometrische Interpretation

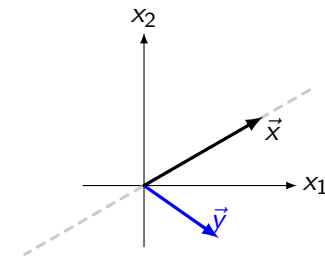


- Es gilt:  $\vec{x} \cdot \vec{y} = |\vec{x}| |\vec{y}| \cos(\alpha)$ . („Länge der **Projektion**“).
- Die **Fläche** (allgemein: Volumen) des durch  $\vec{x}$  und  $\vec{y}$  aufgespannten Parallelogramms ist gerade der Absolutwert der Determinanten.  
 Oder: Die Determinante liefert eine **vorzeichenbehaftete** Fläche.

# Winkelbestimmung (I)

## Problem

Liegt ein Vektor  $\vec{y}$  *links* oder *rechts* von einem gegebenen Vektor  $\vec{x}$ ?

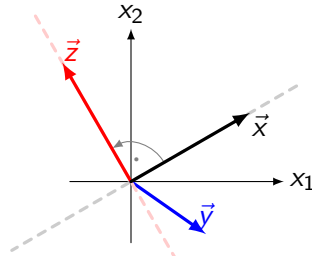


- Wir betrachten zunächst  $\vec{y}$ .

## Winkelbestimmung (I)

### Problem

Liegt ein Vektor  $\vec{y}$  *links* oder *rechts* von einem gegebenen Vektor  $\vec{x}$ ?

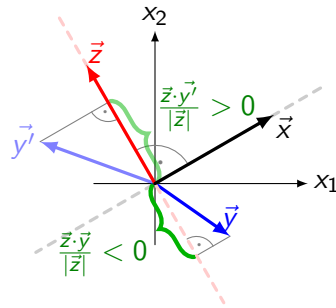


- Wir betrachten zunächst  $\vec{y}$ .
- Konstruiere  $\vec{z}$ , den zu  $\vec{x}$  im mathematisch positiven Sinn (Gegenuhrzeigersinn) um 90 Grad gedrehten Vektor.

## Winkelbestimmung (I)

### Problem

Liegt ein Vektor  $\vec{y}$  *links* oder *rechts* von einem gegebenen Vektor  $\vec{x}$ ?

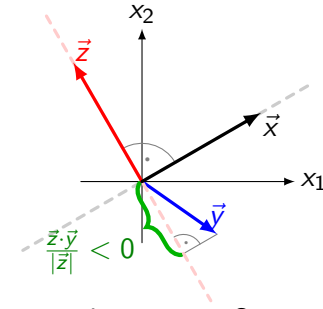


- Wir betrachten zunächst  $\vec{y}$ .
- Konstruiere  $\vec{z}$ , den zu  $\vec{x}$  im mathematisch positiven Sinn (Gegenuhrzeigersinn) um 90 Grad gedrehten Vektor.
- Projiziere  $\vec{y}$  auf  $\vec{z}$ . Da  $\vec{y}$  *rechts* von  $\vec{x}$  liegt und damit von  $\vec{z}$  wegzeigt, ist  $\vec{z} \cdot \vec{y}$  *negativ*.
- $\vec{y}$  dagegen liegt *links* von  $\vec{x}$ , daher ist  $\vec{z} \cdot \vec{y}$  *positiv*.

## Winkelbestimmung (I)

### Problem

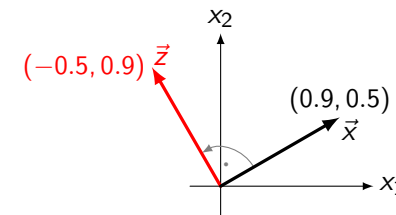
Liegt ein Vektor  $\vec{y}$  *links* oder *rechts* von einem gegebenen Vektor  $\vec{x}$ ?



- Wir betrachten zunächst  $\vec{y}$ .
- Konstruiere  $\vec{z}$ , den zu  $\vec{x}$  im mathematisch positiven Sinn (Gegenuhrzeigersinn) um 90 Grad gedrehten Vektor.
- Projiziere  $\vec{y}$  auf  $\vec{z}$ . Da  $\vec{y}$  *rechts* von  $\vec{x}$  liegt und damit von  $\vec{z}$  wegzeigt, ist  $\vec{z} \cdot \vec{y}$  *negativ*.

## Winkelbestimmung (II)

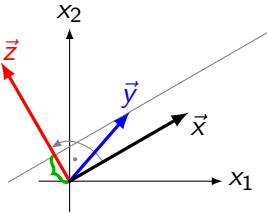
Wie berechnet sich aber  $\vec{z}$  aus  $\vec{x}$ ?



- Für  $\vec{x} = (x_1, x_2)$  ist  $\vec{z}$  gerade  $(-x_2, x_1)$ .

## Winkelbestimmung (II)

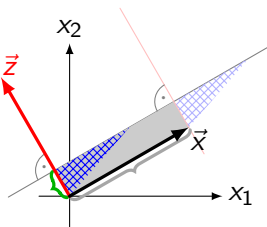
Wie berechnet sich aber  $\vec{z}$  aus  $\vec{x}$ ?



- Für  $\vec{x} = (x_1, x_2)$  ist  $\vec{z}$  gerade  $(-x_2, x_1)$ .
- Insgesamt ist damit  $\vec{z} \cdot \vec{y} = z_1 y_1 + z_2 y_2 = -x_2 y_1 + x_1 y_2$

## Winkelbestimmung (II)

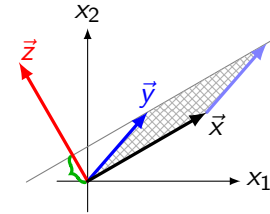
Wie berechnet sich aber  $\vec{z}$  aus  $\vec{x}$ ?



- Für  $\vec{x} = (x_1, x_2)$  ist  $\vec{z}$  gerade  $(-x_2, x_1)$ .
- Insgesamt ist damit  $\vec{z} \cdot \vec{y} = z_1 y_1 + z_2 y_2 = -x_2 y_1 + x_1 y_2 = \det(\vec{x}, \vec{y})$ .

## Winkelbestimmung (II)

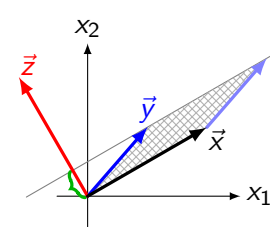
Wie berechnet sich aber  $\vec{z}$  aus  $\vec{x}$ ?



- Für  $\vec{x} = (x_1, x_2)$  ist  $\vec{z}$  gerade  $(-x_2, x_1)$ .
- Insgesamt ist damit  $\vec{z} \cdot \vec{y} = z_1 y_1 + z_2 y_2 = -x_2 y_1 + x_1 y_2 = \det(\vec{x}, \vec{y})$ .

## Winkelbestimmung (II)

Wie berechnet sich aber  $\vec{z}$  aus  $\vec{x}$ ?



- Für  $\vec{x} = (x_1, x_2)$  ist  $\vec{z}$  gerade  $(-x_2, x_1)$ .
- Insgesamt ist damit  $\vec{z} \cdot \vec{y} = z_1 y_1 + z_2 y_2 = -x_2 y_1 + x_1 y_2 = \det(\vec{x}, \vec{y})$ .
- Ist  $\det(\vec{x}, \vec{y}) = 0$ , dann sind  $\vec{x}$  und  $\vec{y}$  parallel (bzw. antiparallel).

## Strecken

### Punkt, Strecke, Polygon

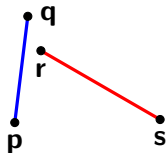
- **Punkte** aus dem  $\mathbb{R}^2$ :  $\mathbf{p} = \begin{bmatrix} p_1 \\ p_2 \end{bmatrix} = (p_1, p_2)$ .
- Der Punkt  $(0,0)$  heißt **Ursprung**.
- Mit dem Vektor  $\vec{d}_{pq} = \mathbf{q} - \mathbf{p}$  kommt man dann von  $\mathbf{p}$  nach  $\mathbf{q}$ .
- Die (ungerichtete) **Strecke**  $\overline{pq}$  ist die Menge aller Punkte zwischen den beiden **Endpunkten**  $\mathbf{p}$  und  $\mathbf{q}$  (Konvexkombination):  

$$\overline{pq} = \{ (1 - \alpha) \cdot \mathbf{p} + \alpha \cdot \mathbf{q} \mid 0 \leq \alpha \leq 1 \} = \{ \mathbf{p} + \alpha \cdot \vec{d} \mid 0 \leq \alpha \leq 1 \}.$$
- Fasst man  $\vec{pq}$  als gerichtete Strecke auf, so ist  $\vec{d}_{pq}$  die **Richtung**.
- Eine **Streckenzug** ist eine Folge von Punkten  $(\mathbf{p}_1, \dots, \mathbf{p}_n)$ , die durch Strecken miteinander verbunden sind:  $\overline{p_1 p_2}, \overline{p_2 p_3}, \dots, \overline{p_{n-1} p_n}$ .
- Ein **Polygon** mit den **Ecken**  $\mathbf{p}_1, \dots, \mathbf{p}_n$  hat als **Rand** gerade den **geschlossenen Streckenzug**  $(\mathbf{p}_1, \dots, \mathbf{p}_n, \mathbf{p}_1)$ .

## Schnitt zweier Strecken

### Problem

Gegeben zwei Strecken  $\overline{pq}$  und  $\overline{rs}$ . Schneiden sich diese?

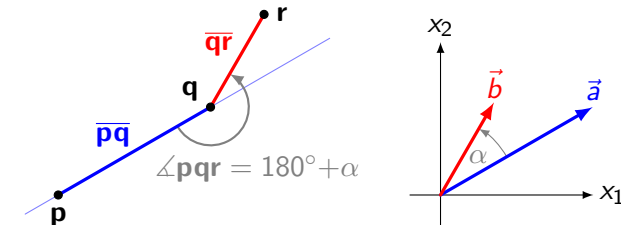


- Wir sind nicht an der Position des Schnittpunktes interessiert.
- Idee: Wir testen, ob die Endpunkte  $\mathbf{r}$  und  $\mathbf{s}$  auf verschiedenen Seiten von  $\overline{pq}$  liegen. Ebenso für  $\mathbf{p}$  und  $\mathbf{q}$  bezüglich  $\overline{rs}$ .

## Winkelbestimmung (III)

### Problem

Gegeben der Streckenzug  $(\mathbf{p}, \mathbf{q}, \mathbf{r})$ . Wird bei  $\mathbf{q}$  nach **links** oder **rechts** abgelenkt? Oder: Ist der Winkel  $\angle pqr > 180^\circ$  oder  $< 180^\circ$ ?

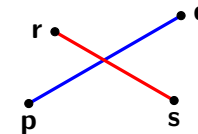


- Wir verwenden wieder die Determinante.
- Dazu berechnen wir  $\vec{a} = \vec{d}_{pq} = \mathbf{q} - \mathbf{p}$  und  $\vec{b} = \vec{d}_{qr} = \mathbf{r} - \mathbf{q}$ .
- $\det(\vec{a}, \vec{b}) > 0$ , falls der Knick nach **links** geht ( $\angle pqr > 180^\circ$ ,  $\alpha > 0$ ).
- Wenn  $\mathbf{r}$  auf (der Verlängerung von)  $\overline{pq}$  liegt, dann ist  $\det(\vec{a}, \vec{b}) = 0$  ( $\angle pqr = 0^\circ$  oder  $= 180^\circ$ ).

## Schnitt zweier Strecken

### Problem

Gegeben zwei Strecken  $\overline{pq}$  und  $\overline{rs}$ . Schneiden sich diese?

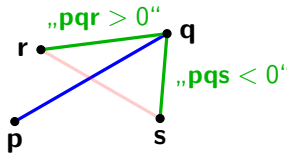


- Wir sind nicht an der Position des Schnittpunktes interessiert.
- Idee: Wir testen, ob die Endpunkte  $\mathbf{r}$  und  $\mathbf{s}$  auf verschiedenen Seiten von  $\overline{pq}$  liegen. Ebenso für  $\mathbf{p}$  und  $\mathbf{q}$  bezüglich  $\overline{rs}$ .

## Schnitt zweier Strecken

### Problem

Gegeben zwei Strecken  $\overline{pq}$  und  $\overline{rs}$ . Schneiden sich diese?

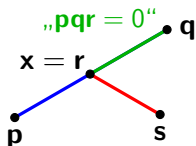


- Wir sind nicht an der Position des Schnittpunktes interessiert.
- Idee: Wir testen, ob die Endpunkte  $r$  und  $s$  auf verschiedenen Seiten von  $\overline{pq}$  liegen. Ebenso für  $p$  und  $q$  bezüglich  $\overline{rs}$ .

## Schnitt zweier Strecken

### Problem

Gegeben zwei Strecken  $\overline{pq}$  und  $\overline{rs}$ . Schneiden sich diese?

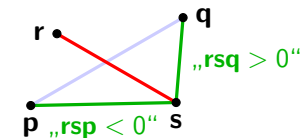


- Wir sind nicht an der Position des Schnittpunktes interessiert.
- Idee: Wir testen, ob die Endpunkte  $r$  und  $s$  auf verschiedenen Seiten von  $\overline{pq}$  liegen. Ebenso für  $p$  und  $q$  bezüglich  $\overline{rs}$ .
- Sonderfall:  $\det = 0$ . Der Endpunkt, etwa  $x$ , liegt also auf der Verlängerung von  $\overline{pq}$  (bzw.  $\overline{rs}$ ). Es bleibt zu prüfen, ob  $x$  auch zwischen  $p$  und  $q$  liegt.

## Schnitt zweier Strecken

### Problem

Gegeben zwei Strecken  $\overline{pq}$  und  $\overline{rs}$ . Schneiden sich diese?



- Wir sind nicht an der Position des Schnittpunktes interessiert.
- Idee: Wir testen, ob die Endpunkte  $r$  und  $s$  auf verschiedenen Seiten von  $\overline{pq}$  liegen. Ebenso für  $p$  und  $q$  bezüglich  $\overline{rs}$ .

## Schnitt zweier Strecken – Algorithmus (I)

```

1 float det(float a[2], float b[2]) {
2   return a[0]*b[1] - a[1]*b[0];
3 }

4
5 // Richtung des Knicks zwischen pq und qr?
6 float direction(float p[2], float q[2], float r[2]) {
7   float a[2] = {q[0]-p[0], q[1]-p[1]}; // q-p
8   float b[2] = {r[0]-q[0], r[1]-q[1]}; // r-q
9   return det(a,b);
10 }

11
12 // Vorbedingung: x liegt auf (der Verlängerung von) pq.
13 // Teste, ob x auch zwischen p und q liegt.
14 bool onSegment(float p[2], float q[2], float x[2]) {
15   float topright[2] = {max(p[0],q[0]), max(p[1],q[1])};
16   float botleft[2] = {min(p[0],q[0]), min(p[1],q[1])};
17   // return (botleft[0] <= x[0] <= topright[0] &&
18   //        botleft[1] <= x[1] <= topright[1]);
19   return (x[0] <= topright[0] && x[1] <= topright[1] &&
20          (botleft[0] <= x[0] && botleft[1] <= x[1]));
21 }

```

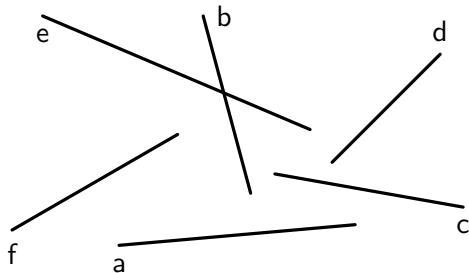
## Schnitt zweier Strecken – Algorithmus (II)

```

1 // Testet, ob pq und rs sich schneiden
2 bool segIntersect(float p[2], float q[2],
3                 float r[2], float s[2]) {
4     float d1 = direction(p,q,r), d2 = direction(p,q,s);
5     // liegt r bzw. s auf pq?
6     if (d1 == 0 && onSegment(p,q,r)) return true;
7     if (d2 == 0 && onSegment(p,q,s)) return true;
8     // r und s auf der selben Seite von pq?
9     if ((d1 > 0 && d2 > 0) || (d1 < 0 && d2 < 0)) return false;
10
11    float d3 = direction(r,s,p), d4 = direction(r,s,q);
12    // liegt p bzw. q auf rs?
13    if (d3 == 0 && onSegment(r,s,p)) return true;
14    if (d4 == 0 && onSegment(r,s,q)) return true;
15    // p und q auf der selben Seite von rs?
16    if ((d3 > 0 && d4 > 0) || (d3 < 0 && d4 < 0)) return false;
17    return true;
18 }

```

## Schnitt eines beliebigen Streckenpaares



### Problem

Gegeben seien  $n$  Strecken. Gibt es einen Schnitt zwischen zwei dieser Strecken? Lässt sich die Frage schneller als  $O(n^2)$  beantworten?

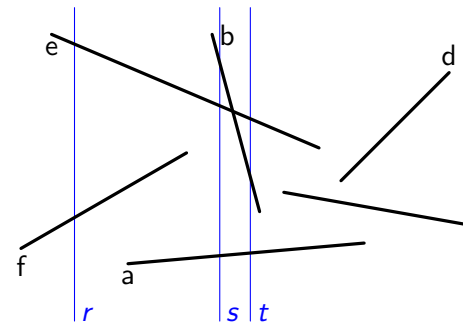
### Annahmen

- Wir lassen keine vertikalen Strecken zu.
- Es schneiden sich nicht mehr als zwei Strecken im selben Punkt.

## Übersicht

- 1 Algorithmische Geometrie
  - Winkelbestimmung
  - Schnitt zweier Strecken
- 2 Schnitt eines beliebigen Streckenpaares
  - Ordnen von Strecken
  - Sweepeline
- 3 Konvexe Hülle

## Ordnen von Strecken



### Beispiel

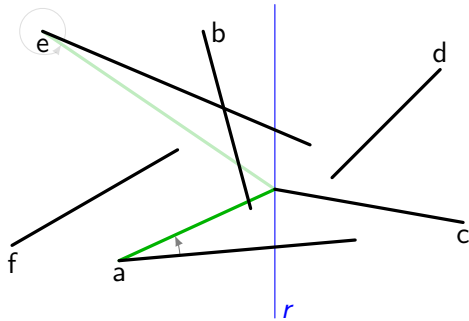
- $e >_r f$ ; sowie  $f$  mit  $a$  nicht vergleichbar bei  $r$  (usw.).
- $b >_s a$ ,  $e >_s a$ ,  $b >_s e$ .
- $b >_t a$ ,  $e >_t a$ ,  $e >_t b$ .  
(Der Schnitt vertauscht die Reihenfolge von  $e$  und  $b$ ).

### Vergleichbarkeit

Zwei Strecken  $s_1$  und  $s_2$  heißen **vergleichbar an der Stelle  $r$** , wenn beide die vertikale Linie mit  $x_1$ -Koordinate  $= r$  schneiden.

- Wenn  $s_1$  an der Stelle  $r$  **über**  $s_2$  liegt schreiben wir  $s_1 >_r s_2$ , sonst  $s_2 >_r s_1$ , bzw.  $s_1 =_r s_2$ .

## Wie ordnet man Strecken? (I)



- ▶ Für beliebige  $r$ : Gar nicht .... Allerdings:
- ▶ Beobachtung 1: Die Ordnung kann sich nur ändern, wenn eine Strecke **hinzukommt** (vergleichbar wird) bzw. **herausfällt**, oder wenn sich zwei Strecken **schneiden**.
- ▶ Beobachtung 2: Für den linken Endpunkt einer hinzukommenden Strecke lässt sich mit der Determinante bestimmen, ob er **über** oder **unter** einer an dieser Stelle vergleichbaren Strecke liegt.

## Sweepline-Algorithmen

Ein **Sweepline-Algorithmus** verwaltet gewöhnlich zwei Datenmengen:

### Sweepline-Status:

Gibt die Beziehung zwischen den von der Sweepline geschnittenen Objekten an.

### Ereignisliste:

Eine Liste, in der die **Ereignispunkte** sortiert aufgelistet sind.

Nur an diesen hält die Sweepline an, da sich der Status nur an solchen ändern kann.

- ▶ Je nach Anwendung kann die Ereignisliste schon im voraus bestimmt (und sortiert) werden (**statische Ereignisliste**), oder aber sie entsteht erst beim Durchlauf (**dynamische Ereignisliste**).
- ▶ Dynamische Ereignisliste können z. B. mit binären Suchbäumen effizient implementiert werden.

## Wie ordnet man Strecken? (II)

- ▶ Insbesondere lässt sich so ein Endpunkt, und damit die Strecke, in eine gegebenen Ordnung von Strecken einsortieren.
- ▶ Hält man die Ordnung in einem balanciertem Binärbaum vor (Details später), dann benötigt man bei  $n$  Strecken  $\Theta(n \log n)$  Operationen.

Das führt zur Idee der **Sweepline**:

- ▶ Wir wandern von links nach rechts über die Ebene und fügen die Strecken unserer Ordnung hinzu, bzw. entfernen sie **beim Passieren** der jeweiligen Endpunkte.
- ▶ So können wir für jede Position die Ordnung angeben, solange wir Schnitte erkennen.
- ▶ Da sich schneidende Linien immer zunächst in der Ordnung **benachbart** sind, brauchen wir nun, für eine hinzukommende oder verlassende Strecke, nur die beiden benachbarten Strecken direkt ober- und unterhalb unseres Endpunktes auf etwaige Schnitte testen.

## Schnitt eines beliebigen Streckenpaares – Sweepline

Für den Schnitt eines beliebigen Streckenpaares heißt das:

### Sweepline-Status:

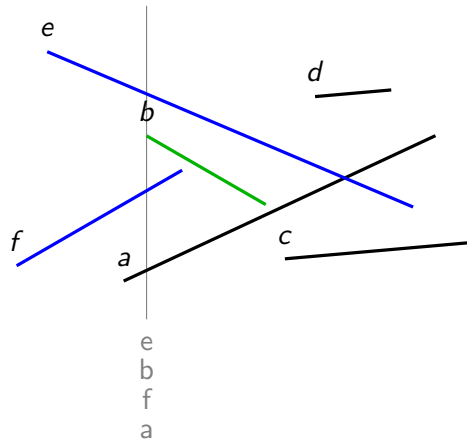
- ⇒ Die Ordnung der Strecken an der aktuellen Position der Sweepline.
  - ▶ In die Ordnung müssen hinzukommende Strecken **eingefügt** werden, verlassende Strecken **gelöscht** werden.
  - ▶ Außerdem benötigen wir Operationen, um die direkt **über** und **unter** einer Strecke liegende Strecke zu erhalten, da (nur) zwischen solchen auf Schnitt geprüft wird.
  - ▶ Etwa **RBTs** implementieren diesen dynamischen, sortierten ADT.
  - ▶ „Unter“ und „Über“ entspricht dem Nachfolger (**bstSucc**) und dem Vorgänger (analog: **bstPred**).

### Ereignisliste:

- ▶ Ereignispunkte sind alle Endpunkte der Strecken. Diese sind bereits im Vorfeld bekannt.
- ▶ **Dazu** kommen – *beim Durchlauf* – ggf. gefundene Schnittpunkte, da sich die Ordnung der beteiligten Linien vertauscht.

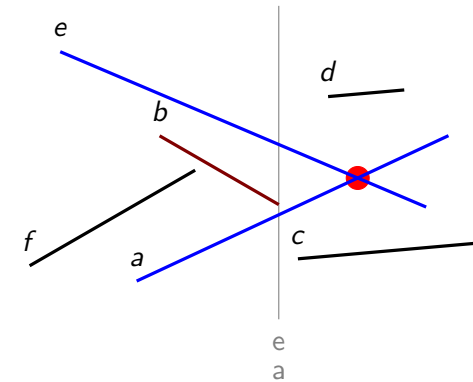


## Schnitt eines beliebigen Streckenpaares – Beispiel



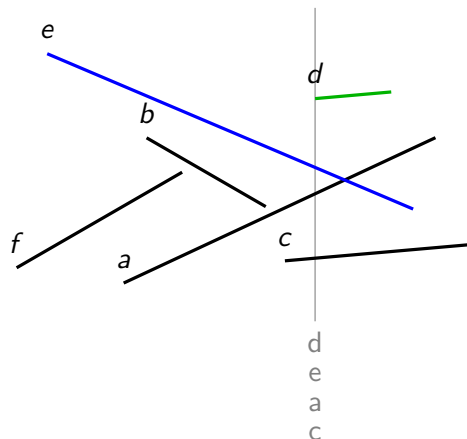
- ▶ Im Algorithmus brechen wir ab, sobald der erste Schnitt **gefunden** ist.
- ▶ Hier muss jedoch die Ordnung der beiden Linien **am Schnittpunkt** getauscht werden (mit einer speziellen ADT-Operation).

## Schnitt eines beliebigen Streckenpaares – Beispiel



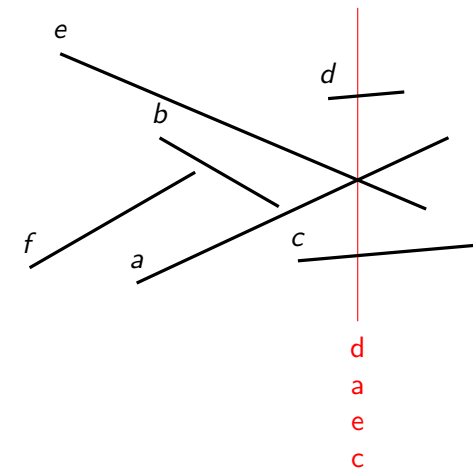
- ▶ Im Algorithmus brechen wir ab, sobald der erste Schnitt **gefunden** ist.
- ▶ Hier muss jedoch die Ordnung der beiden Linien **am Schnittpunkt** getauscht werden (mit einer speziellen ADT-Operation).

## Schnitt eines beliebigen Streckenpaares – Beispiel



- ▶ Im Algorithmus brechen wir ab, sobald der erste Schnitt **gefunden** ist.
- ▶ Hier muss jedoch die Ordnung der beiden Linien **am Schnittpunkt** getauscht werden (mit einer speziellen ADT-Operation).

## Schnitt eines beliebigen Streckenpaares – Beispiel



- ▶ Im Algorithmus brechen wir ab, sobald der erste Schnitt **gefunden** ist.
- ▶ Hier muss jedoch die Ordnung der beiden Linien **am Schnittpunkt** getauscht werden (mit einer speziellen ADT-Operation).

## Schnitt, beliebiges Streckenpaar – Algorithmus (I)

```

1 typedef float[2] Point; // wir schreiben Point für float[2]

3 // Wir übergeben die n Strecken in einem Array von Punkten:
4 // Point linept[2*n]; wobei linept[0]-linept[1],
5 // linept[2]-linept[3], ..., linept[2*(n-1)]-linept[2*(n-1)+1]

7 // 1. Tausche ggf. [2*i] und [2*i+1], so dass [2*i] links ist.
8 // 2. Sortiere die Punkte von links nach rechts; vergleiche  $x_2$ 
9 // bei gleichem  $x_1$ . Tausche nun aber nicht die Punkte, sondern
10 // bestimme die Permutation (tausche auf einem Index-Array).
11 int[2*n] sortLeftRight(Point &points[2*n]) { ... selbst ... }

13 // Schnitt lines[i]-lines[i+1] und lines[j]-lines[j+1]?
14 bool Intersect(Point lines[], int i, int j) {
15     if (i == -1 || j == -1) // Strecke i oder j nicht gefunden
16         return false;
17     return segIntersect(lines[i], lines[i+1], lines[j], lines[j+1]);
18 }

```

## Schnitt eines beliebigen Streckenpaares – ADT

Wie kann man aber binäre Suchbäume für Strecken mit zwei Endpunkten verwenden? – Erinnerung:

```

1 void bstIns(Tree t, Node node) // Füge node in den Baum t ein
2 // Suche freien Platz [...]
3     if (node.key < root.key) {
4         root = root.left;
5     } else {
6         root = root.right;
7     }
8 // [...] Einfügen [...]

```

- ▶ Wir müssen einen geeigneten Vergleich verwenden.
- ▶ Im Algorithmus haben wir Strecken über ihren *linken Endpunkt* eindeutig identifiziert.
- ▶ Wir speichern also als Schlüssel nur den Index des linken Endpunktes.

```

1 void Tree::Insert(Point lnp[], int i) // [...]
2     if (direction(lnp[root.key+1], lnp[root.key], lnp[i]) < 0)
3         // [...]

```

## Schnitt, beliebiges Streckenpaar – Algorithmus (II)

```

1 bool anyIntersect(Point linept[2*n], int n) {
2     int sortMap[2*n] = sortLeftRight(linept);
3
4     Tree order; // speichere linken Endpunkt als Repräsentant
5     for (int i = 0; i < 2*n; i++) { // Sweepline
6         int line = sortMap[i]; // Originalindex des i-ten Punktes
7         if (line % 2 == 0) { // linker Endpunkt
8             order.Insert(linept, line);
9             int above = order.Pred(line), below = order.Succ(line);
10            if (Intersect(linept, line, above)) return true;
11            if (Intersect(linept, line, below)) return true;
12        } else { // rechter Endpunkt
13            line--; // Repräsentant ist aber der linke Punkt
14            int above = order.Pred(line), below = order.Succ(line);
15            if (Intersect(linept, above, below)) return true;
16            order.Delete(linept, line);
17        }
18    }
19    return false;
20 }

```

## Schnitt eines beliebigen Streckenpaares – Komplexität

### Zeitkomplexität

Die Worst-Case Zeitkomplexität, zu bestimmen ob sich zwei beliebige Strecken aus einer Menge von  $n$  Strecken schneiden, ist  $O(n \log n)$ .

### Beweis.

- ▶ Der Test, ob sich zwei Strecken schneiden geht in  $O(1)$ .
- ▶ Zum Sortieren der Ereignispunkte können wir auf bekannte Sortierverfahren mit  $O(n \log n)$  zurückgreifen.
- ▶ Wir iterieren über die  $2 \cdot n$  Endpunkte, wobei wir  $O(\log n)$ -Operationen der RBTs verwenden. Somit:  $O(n \log n)$ .

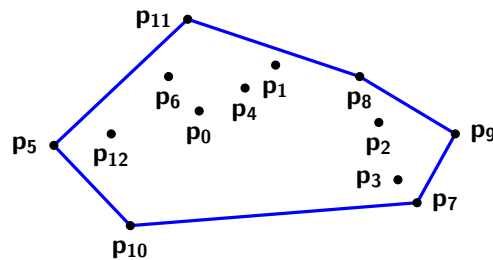
⇒ Im Worst-Case benötigt anyIntersect  $O(n \log n)$  Zeit.



# Übersicht

- 1 Algorithmische Geometrie
  - Winkelbestimmung
  - Schnitt zweier Strecken
- 2 Schnitt eines beliebigen Streckenpaares
  - Ordnen von Strecken
  - Sweepeline
- 3 Konvexe Hülle

## Konvexe Hülle

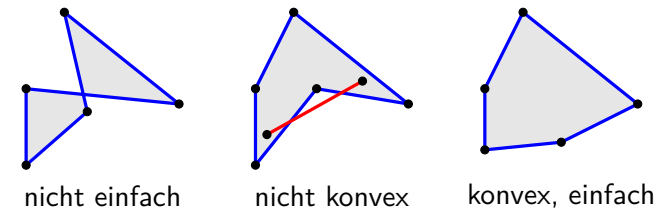


### Konvexe Hülle

Die **konvexe Hülle** einer Menge  $Q$  von Punkten ist das kleinste konvexe Polygon  $P$ , für das sich jeder Punkt in  $Q$  entweder auf dem Rand von  $P$  oder in seinem Inneren befindet.

- ▶ Betrachte jeden Punkt als Nagel, der aus einem Brett herausragt.
- ▶ Die konvexe Hülle hat dann die Form, die durch ein straffes Gummiband gebildet wird, das alle Nägel umschließt.

## Polygone



### einfach

Ein Polygon heißt **einfach**, wenn es sich nicht selbst schneidet.

### konvex

Ein Polygon heißt **konvex**, wenn jede Verbindung (Konvexkombination) zweier Punkte des Polygons nie außerhalb des Polygons liegt.

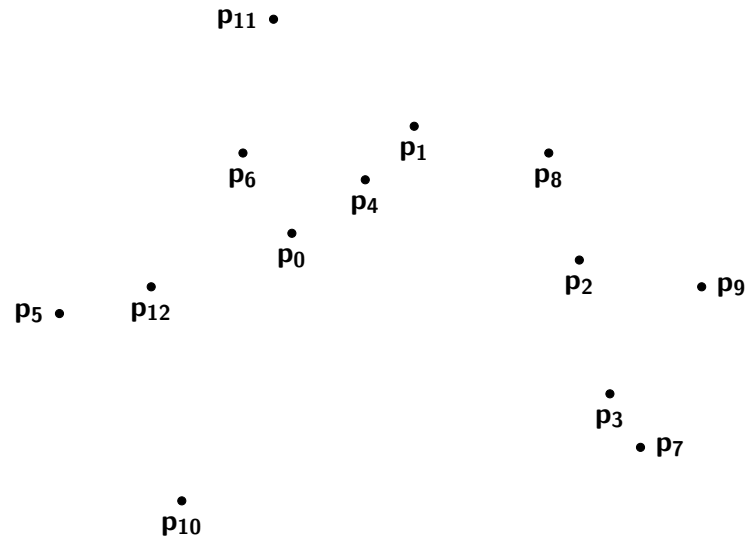
## Konvexe Hülle – Graham-Scan – Idee

- ▶ Wir ziehen ein Gummiband Punkt für Punkt weiter.
- ▶ Ausgehend von einem ausgezeichnetem Punkt, der **auf der Hülle** liegt:
  - ▶ Der Punkt mit der geringsten  $x_2$ -Koordinate, bei Mehrdeutigkeiten außerdem geringsten  $x_1$ -Koordinate, ist geeignet.
- ▶ Von diesem ausgehend sortiere die Punkte, diesmal nach **zunehmendem Winkel** (mittels Determinante).
- ▶ Das entspricht einer **rotierenden Sweepeline**.

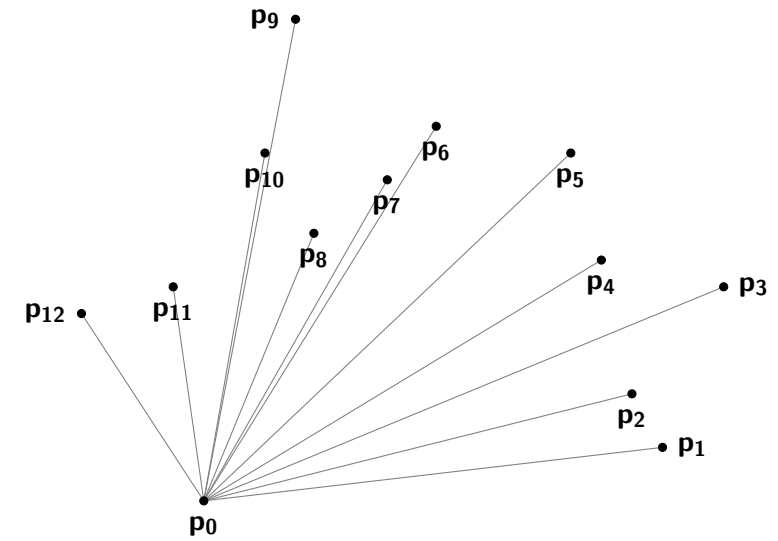
Dann gilt:

- ▶ Entweder das Gummiband liegt weiterhin an, oder der neue Punkt hebt das Gummiband vom vorigen Punkt weg. Dann ist der vorige Punkt sicherlich nicht Teil der konvexen Hülle. Überprüfe in dem Fall nun den „neuen“ vorigen Punkt (usw.).
- ▶ Bemerke, dass auch (neben dem Startpunkt) der Punkt mit dem geringsten Polarwinkel sicher **auf der Hülle** liegt.
  - ▶ Gleiches gilt für den Punkt mit größtem Polarwinkel.

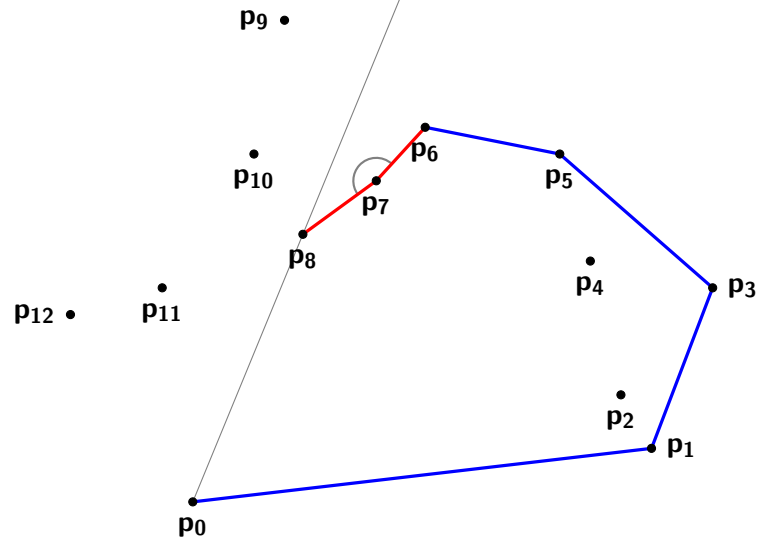
## Konvexe Hülle – Graham-Scan – Beispiel



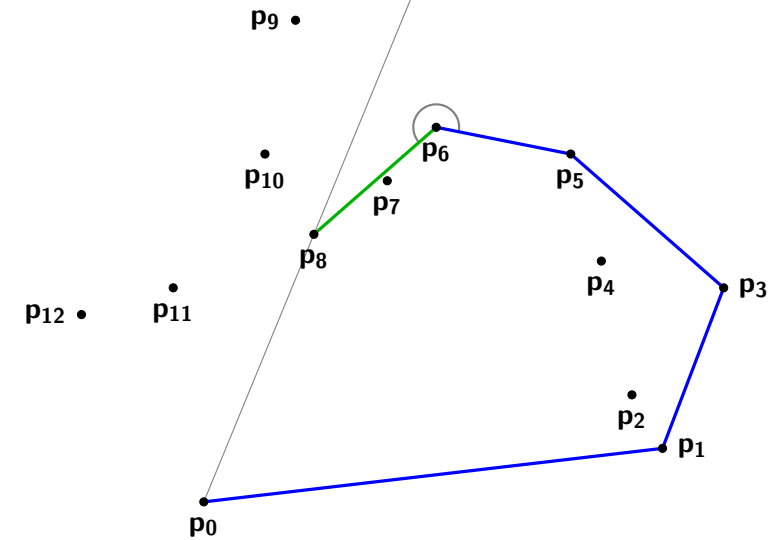
## Konvexe Hülle – Graham-Scan – Beispiel



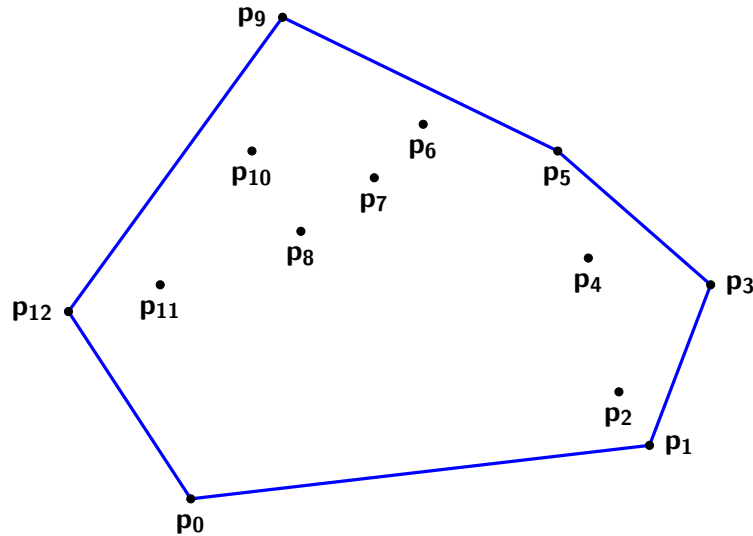
## Konvexe Hülle – Graham-Scan – Beispiel



## Konvexe Hülle – Graham-Scan – Beispiel



## Konvexe Hülle – Graham-Scan – Beispiel



## Graham-Scan – Korrektheit

### Korrektheit

Wenn `grahamScan` auf einer Punktmenge  $Q$  läuft, dann gilt bei Terminierung, dass der Stapel `hull` von unten nach oben die Eckpunkte der konvexen Hüllen von  $Q$  enthält in der dem Uhrzeigersinn entgegengesetzten Reihenfolge.

### Beweis.

(Skizze). Die Schleifeninvariante ist: zu Beginn der  $i$ -ten Iteration besteht der Stapel `hull` von unten nach oben genau aus den Eckpunkten von der konvexen Hülle der Punktmenge  $\{p_0, \dots, p_{i-1}\}$ .

Wir verzichten hier auf weitere Details.  $\square$

## Konvexe Hülle – Graham-Scan – Algorithmus

```

1 // gibt index der Punkte auf der Hülle zurück
2 int[] grahamScan(Point poly[n], int n) {
3     // Finde Index des Punktes mit minimaler  $x_1$  Koordinate
4     int refPnt = findRef(poly);
5     // Sortiere Punkte nach aufsteigendem Polarwinkel bezüglich
6     // refPnt, setze dabei refPnt an Position 0. Lösche alle
7     // bis auf den äußersten, bei mehreren mit gleichem Winkel.
8     inverseMap = polarSort(poly, refPnt);
9
10    Stack hull; // Punkte, die bis jetzt auf der Hülle sind.
11    for (int i = 0; i < n; i++) {
12        if (i > 2)
13            while (direction(poly[hull[-2]], poly[hull[-1]], poly[i])
14                  <= 0) // die oberen beiden Punkte vom Stack
15                hull.pop();
16        hull.push(i);
17    }
18    return inverseMap(hull); // macht die Umsortierung rückgängig
19 }
20 
```

## Graham-Scan – Komplexität

### Zeitkomplexität

Die Worst-Case Zeitkomplexität von `grahamScan` für eine Menge mit  $n$  Punkten ist  $\Theta(n \log n)$ .

### Beweis.

- ▶ `direction`  $\in \Theta(1)$ .
- ▶ `findRef` benötigt einen Durchlauf über die Punkte, also  $\Theta(n)$ .
- ▶ Sortieren: `polarSort`  $\in \Theta(n \log n)$ .
- ▶  $n$  Schleifendurchläufe mit, von der `while`-Schleife abgesehen,  $\Theta(1)$ .
- ▶ Da im `while` nur Punkte von Stack genommen werden, die vorher durch die `for`-Schleife hinzugefügt wurden, können alle Iterationen der `while`-Schleife in der Analyse jeweils zum jeweiligen `push` gerechnet werden.  
Somit bleibt es bei insgesamt  $\Theta(n)$  für alle Schleifen.  $\square$