Universität Trier

Fachbereich IV, Informatik, Universität Trier

Polygone im 2D-Raum

„Punkt-im-Polygon“ und „Schnitt von konvexen Polygonen“

Praktikum „Algorithmische Geometrie / Graphalgorithmen“
Sommersemester 2007

Professur:

Prof. Dr. Stefan Näher
Fachbereich IV, Informatik:
Datenstrukturen und Effiziente Algorithmen

von:

Eric Deutsch
E-Mail: eric@eric-deutsch.de
MatrikelNr.: 700976

Peter Müller
E-Mail: muel4701@uni-trier.de
MatrikelNr.: 685720

Inhaltsverzeichnis

(1) Aufgabenstellung

Unser Praktikum beschäftigt sich mit verschiedenen Polygonen im zweidimensionalen Raum.

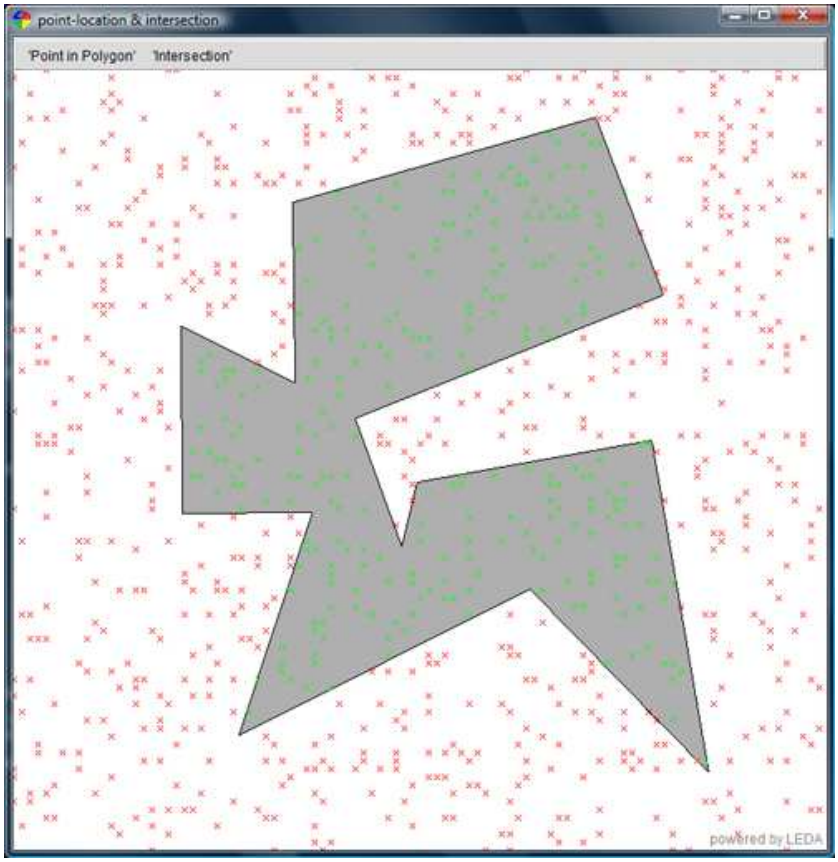
Der erste Teil der Aufgabe umfasst den Test, ob ein zufälliger Punkt innerhalb oder außerhalb eines beliebigen (nicht) konvexen Polygons liegt. Dazu implementieren wir die beiden Algorithmen „Ray Crossings“ und „Winding Number“. Abschließend soll ein Vergleich der Geschwindigkeit dieser beiden Methoden stattfinden.

Der zweite Aufgabenteil beschäftigt sich mit der Berechnung des Schnitts von zwei konvexen Polygonen. Dazu existieren mehrere mögliche Algorithmen, wir nutzen denjenigen von O’Rourke, Chien, Olson und Naddor aus dem Jahr 1982.

Als Programmiersprache dient C++ unter Zuhilfenahme von LEDA (Bibliothek effizienter Datenstrukturen und Algorithmen).

Die Algorithmen sollen nicht nur eine theoretische Berechnung sein, sondern alles wird graphisch aufbereitet und die Lösung auch am Bildschirm dargestellt. Dem Nutzer soll die Möglichkeit gegeben werden, für den Punkt-im-Polygon-Test und die Schnittberechnung individuelle Polygone einzugeben.

(2) „Punkt-im-Polygon“ – Tests



(2.1) Ray Crossings

Gegeben sei ein Punkt *q* und ein Polygon *P*. Dabei kann es sich um ein konvexes oder ein nicht-konvexes Polygon handeln. Der Punkt kann innerhalb oder außerhalb des Polygons liegen.
Die Position des Punktes gilt es nun mit dem „Ray Crossings“ – Algorithmus herauszufinden.

Die Grundidee ist die, dass man von *q* aus einen Strahl *r* in eine beliebige Richtung zeichnet. Wir nutzen in unserem Fall einen, vom Punkt aus gesehen, horizontalen Strahl nach rechts.

Nun gilt es die Anzahl der Schnitte des Strahls mit Segmenten des Polygons herauszufinden. Ungerade Anzahl bedeutet, dass das Polygon unseren Testpunkt beinhaltet, gerade bedeutet, dass er außerhalb liegt.
Hat man beispielsweise drei Schnittpunkte berechnet und wandert am Strahl entlang rückwärts in Richtung des Punktes q, dann führt der erste Schnittpunkt innerhalb des Polygons, der zweite Schnitt führt den Strahl wieder außerhalb und beim dritten Schnitt befindet man sich entsprechend wieder innerhalb. Daraus folgert sich, dass auch der Testpunkt innerhalb des Polygons liegt, da hier der Strahl geendet hat, also $q \in P$.

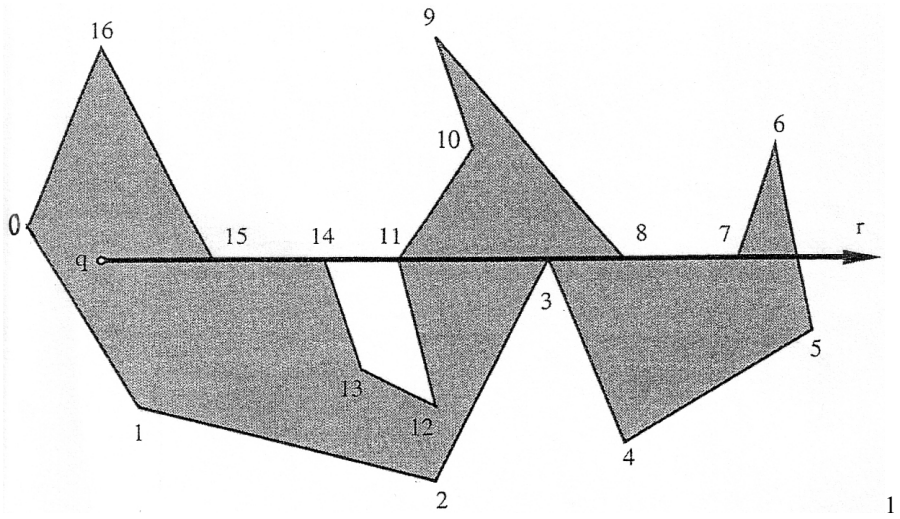


Fig. 1. Beispiel eines nicht-konvexen Polygons

Bei diesem Algorithmus liegt die Schwierigkeit hauptsächlich in der Beachtung etlicher möglicher Sonderfälle bezüglich des Schnitts von r und P:

- Kollinearität von Strahl r und einem Segment s des Polygons P
- Verlauf des Strahls r durch einen Eckpunkt von P
- Testpunkt q liegt direkt auf P (wird als innerer Punkt gewertet, dh. $q \in P$)

Damit ein Segment s als ein Schnitt mit r gewertet wird, muss gelten, dass ein Endpunkt von s eindeutig über r liegen muss und der andere Endpunkt darauf oder darunter.

[1] Quelle: Buch “Computational Geometry in C” von Joseph O’Rourke

Daraus folgt, dass Segment (10, 11) in Fig. 1 Strahl *r* schneidet und (11, 12) nicht. Mit genannter Konvention wird vermieden, dass auf dem Pfad (10, 11, 12) zwei Schnitte gezählt werden, obwohl es sich effektiv nur um einen handelt.
Eckpunkt (3) erhöht die Anzahl der Schnitte nicht, da weder Segment (2, 3), noch Segment (3, 4) einen gemeinsamen Schnitt mit Strahl *r* besitzen.

Um den Test durchzuführen, benötigen wir als Erstes ein Testpolygon.
Hierzu erlaubt unser Programm dem Benutzer, selbst ein Polygon per Hand einzugeben. Desweiteren stehen verschiedene N-Gone und komplexe Polygone, die mit Hilfe der rekursiven Hilbert-Funktion erstellt werden, zur Verfügung. Beide Polygon-Typen können mit Hilfe von LEDA erzeugt werden.
Die Testpunkte werden zufällig am Bildschirm im und um das Polygon verteilt. Die Anzahl derer ist für den Benutzer variabel einstellbar.
Das Programm arbeitet dann jeden Punkt nacheinander ab und entscheidet, ob dieser innerhalb oder außerhalb des Polygons liegt. Innerhalb wird der Punkt farblich anders dargestellt als außerhalb.

Der Quellcode beinhaltet eine eigene Klasse Ray_crossings, mit Hilfe derer der Test durchgeführt werden kann.
Der Funktion

```
bool Ray_crossings::ray_crossings(const polygon& P,

                                  const point q) { … }
```

wird in der Hauptsache ein Polygon *P* und ein Testpunkt *q* übergeben. Wir arbeiten mit einem „*segment ray*“ und einem „*segment ray_reverse*“, benutzen also nicht nur einen Strahl, sondern zwei. Beide sind horizontal, der eine verläuft in Richtung +x, der andere in Richtung –x. Die maximale Länge dieser beiden wird durch die Variable *max* vom Typ Integer festgelegt. Der Typ *segment* ist ein LEDA-Objekttyp.
Nun werden in einer Schleife nacheinander alle Segmente des Polygons durchlaufen und je vier verschiedene Fälle des Schnitts vom Polygon mit dem Strahl *ray* und dem Strahl *ray_reverse* überprüft.
Vorweg sei noch erwähnt, dass wir zu jedem Segment den Startpunkt *a* und den Endpunkt *b* definieren (mit Hilfe der LEDA-Funktionen *.start()* und *.end()* über dem Typ *segment*):

```
segment s;

a = s.start();

b = s.end();
```

Nun überprüfen wir folgende Fälle:

- Schnitt des Segments *s* mit *ray*:
  - *b* liegt auf horizontalem Strahl durch *q* und *a* darüber
  - *a* liegt auf horizontalem Strahl durch *q* und *b* darüber
  - *a* liegt unter horizontalem Strahl durch *q* und *b* darüber
  - *b* liegt unter horizontalem Strahl durch *q* und *a* darüber

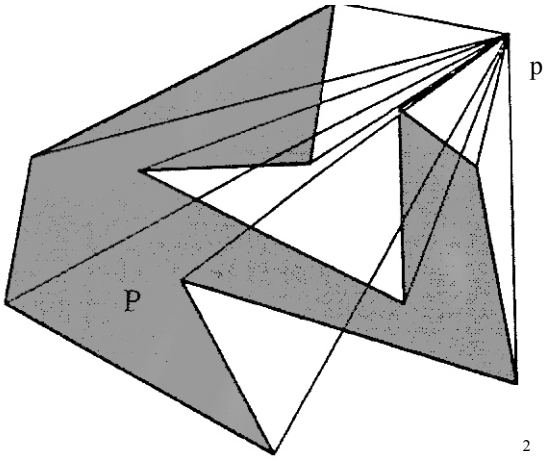
- Schnitt des Segments *s* mit *ray_reverse*:
  - *b* liegt auf horizontalem Strahl durch *q* und *a* darunter
  - *a* liegt auf horizontalem Strahl durch *q* und *b* darunter
  - *a* liegt unter horizontalem Strahl durch *q* und *b* darüber
  - *b* liegt unter horizontalem Strahl durch *q* und *a* darüber

Für jeden eingetretenen Fall wird der Counter (*crossings* bzw. *crossings2*) erhöht. Erhalten wir insgesamt eine ungerade Anzahl Schnitte, gibt die Funktion *ray_crossings(...)* den Wert *true* zurück, andernfalls *false*. *True* bedeutet, der Testpunkt liegt innerhalb, *false* bedeutet das Gegenteil. Innerhalb der *main*-Methode findet nun die passende graphische Ausgabe statt.

(2.2) Winding Number

Die zweite Methode, die wir implementiert haben, um zu überprüfen, ob ein Punkt innerhalb oder nicht innerhalb eines Polygons liegt, basiert auf dem Algorithmus „Winding Number“.

Sei *p* unser Testpunkt. Unser „Standpunkt“ sei ebenso *p*. Von dort aus betrachten wir nacheinander die Ecken des beliebigen Polygons *P*. Alle Ecken werden nacheinander durchlaufen, allerdings muss dies gegen den Uhrzeigersinn geschehen (*counterclockwise*). Je nachdem, in welcher Richtung das Polygon erzeugt wurde, müssen wir vor der Anwendung des Algorithmus garantieren, dass die Orientierung des Polygons *counterclockwise* ist.
Die Funktion *orientation()* in LEDA, welche auf einem Objekt des Typs *Polygon* arbeitet, liefert dabei entweder den Wert *1* für counterclockwise oder *-1* für nicht counterclockwise zurück. Liefert sie *-1* zurück, müssen wir die Orientierung umkehren, indem wir auf dem Polygon die Funktion *complement()* einmal anwenden.



[2]

Fig. 2. Dieser äußere Testpunkt p hat Winding Number 0

Für den Fall, dass $p \in P$, würde man sich einmal ganz im Kreis bewegen, während man sich von *p* aus nacheinander alle Ecken ansieht, dh. eine Drehung um 360° macht bzw. um 2 Pi, für den Fall, dass wir nicht in Grad umrechnen.
Gilt nicht $q \in P$, dann machen wir eine Drehung um 0. Die Summe von 0 kann durch die Konvention entstehen, dass Drehungen im Uhrzeigersinn positiv und Drehungen gegen den Uhrzeigersinn negativ gewertet werden. Wir erhalten also einen positiven bzw. negativen Winkelwert. Eine Aufsummierung kann dann dazu führen, dass die Winkelwerte sich gegenseitig aufheben und wir Null erhalten.

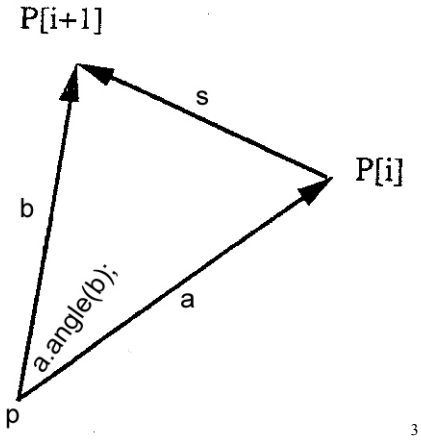
Nun werfen wir einen genaueren Blick auf den Weg der Berechnung dieser *Winding Number.*
Dazu betrachten wir den Winkel, den p verbunden mit dem Eckpunkt P[i] (i=1, ..., n mit n = Anzahl der Eckpunkte bzw. Segmente des Polygons) und *p* verbunden mit dem Eckpunkt P[i+i] einschließt. Die beiden so entstehenden Segmente seien *a* und *b*. Es handelt sich also genau um den Winkel zwischen diesen beiden.

```
segment a (p, s.start());

segment b (p, s.end());

angle_value = a.angle(b) * 180 / PI;
```

[2] Quelle: Buch “Computational Geometry in C” von Joseph O’Rourke, Umbenennung des Testpunkts (Original „q“)



[3]

Fig. 3. Berechnung des i-ten Winkels zwischen a und b von insgesamt n Winkeln

Die Funktion *a.angle(b)* bestimmt nun den Winkel. Der Teil *„* 180 / PI“* dient zur Umwandlung des double-Wertes in eine Grad-Zahl.

Hat *P* n Eckpunkte, was n Segmenten entspricht, dann reicht nicht die Berechnung dieses einen Winkels, sondern wir müssen n solcher Winkel berechnen.
In unserem Programm durchlaufen wir dazu *counterclockwise* alle Segmente nacheinander und bestimmen immer wieder den Winkel zwischen Testpunkt und Start- bzw. Endpunkt des aktuellen Segments.
Dazu nutzen wir die LEDA-Schleife

```
forall_segments (segment s, polygon P) { ... } .
```

Bei der Summe aller Einzelwinkel spricht man von der *Winding Number*.
Wie bereits angesprochen ist es von großer Bedeutung zu beachten, ob es sich um einen negativen oder einen positiven Winkelwert handelt (orientierungsabhängig).
Die Orientierung bestimmt man aus der Lage des aktuell betrachteten Segments, bzw. dessen Anfangs- und Endpunkt, sowie der Lage des Testpunkts p im Verhältnis zum Segment:

```
int orientation (segment s, point p)
```

Da wir mit Fließkomma-Zahlen rechnen, erhält man nicht immer exakt 0 (= 0°) bzw. 2 Pi (= 360°). Es tauchen also während der Winkelberechnung und Summierung derer Rundungsfehler auf.

[3] Quelle: Buch “Computational Geometry in C” von Joseph O’Rourke (jedoch Änderung einiger Beschriftungen im Vgl. Zum Original)

```
anglesum = anglesum + orientation (s, p) * angle_value;
```

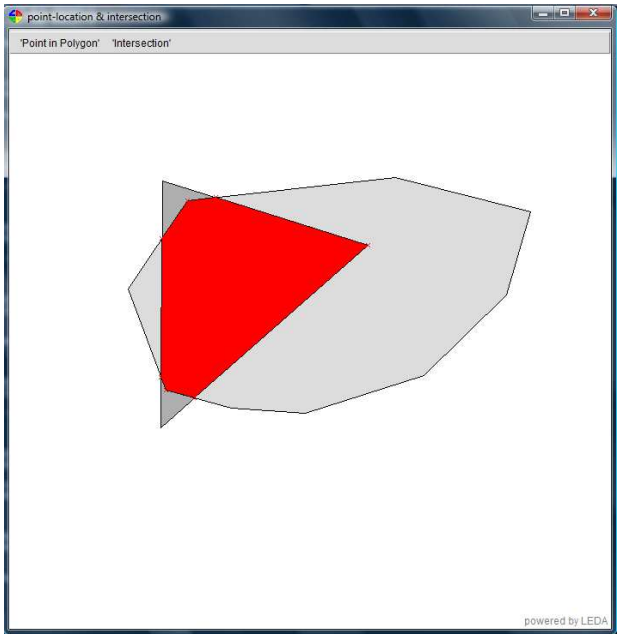
Daher wird im Programm am Ende nur getestet, ob der Wert in einem Bereich zwischen 359° und 361° liegt, somit sind kleinere Ungenauigkeiten abgedeckt.

```
if (anglesum > 359 && anglesum < 361) return true;

else return false;
```

Punkte außerhalb dieses Wertebereichs liegen außerhalb des Polygons.

(3) Schnitt von konvexen Polygonen



(3.1) Erklärung des Algorithmus und seiner Implementierung

Der Schnitt von zwei konvexen Polygonen hat eine lineare Zeitkomplexität von O(n+m), wobei n bzw. m die Anzahl der Eckpunkte/Segmente von dem jeweiligen Polygon ist.
Es gibt eine Reihe von Algorithmen zur Schnittberechnung, wir nutzen für unser Programm wieder einen Algorithmus von O'Rourke, Chien, Olson und Naddor aus dem Jahr 1982.
Seien die beiden Polygone mit *P* und *Q* bezeichnet. Grundbedingung ist wiederum, dass sie eine Orientierung gegen den Uhrzeigersinn besitzen (*counterclockwise*).
In unserer Klasse *Intersection* ist dies der erste Punkt innerhalb der Hauptfunktion

```
polygon_intersection(const polygon& X,const polygon& Y),
```

den wir sicherstellen, indem wir die Orientierung umkehren, wenn nötig.

A und *B* seien jeweils die aktuell gewählten gerichteten Kanten/Segmente der beiden Polygone. *a* ist der Kopf des Segments, *a1* der Beginn des Segments. Entsprechend *b* und *b1* für Segment *B*.

Im Programm werden die Segmente beider Polygone in getrennten Segment-Listen abgespeichert. *A* und *B* werden nun durch Iteratoren über der jeweiligen Liste dargestellt. *a, a1, b, b1* sind Objekte vom LEDA-Typ *point*.

```
list<segment> segments_p1;

list<segment>::iterator A = segments_p1.begin();

list<segment> segments_p2;

list<segment>::iterator B = segments_p2.begin();

…

point a, b;

point a1, b1;
```

A und *B* "verfolgen" sich gegenseitig, nach gewissen Regeln wird stets *A* oder *B* um eins vorgerückt. Sie treffen sich auf alle Fälle aber an jedem Schnittpunkt von *A* und *B*.
Die Grundidee des Algorithmus ist in Fig. 4 veranschaulicht. Das Hauptelement des Algorithmus sind die sog. *„Advance rules"*, also, wann *A* und wann *B* vorgerückt werden.

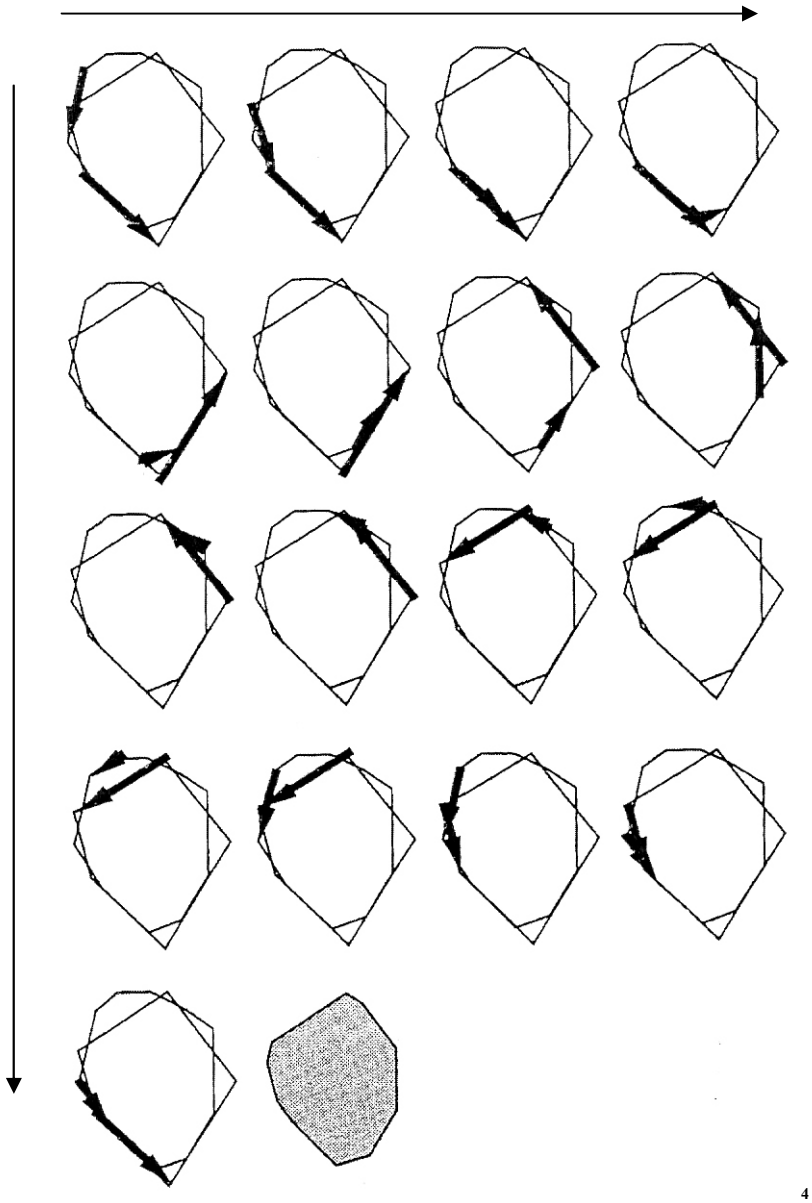


Fig. 4. Veranschaulichung des Algorithmus zur Berechnung des Schnittes von zwei konvexen Polygonen

[4] Quelle: Buch “Computational Geometry in C” von Joseph O’Rourke

Wenn *B* auf *A* zeigt, aber *A* nicht schneidet, möchten wir *B* soweit annähern, dass wir einen möglichen Schnitt erreichen.
H(A) sei die Halbebene links von *A*, gekennzeichnet durch die Notation *„A x B > 0“*. Das bedeutet, dass die geringste Drehung von *A* auf *B* gegen den Uhrzeigersinn wäre.

Dies mündet in folgenden Regeln zum Vorrücken einer der Vektoren:

- Vorrücken von *B*: falls
  - A x B > 0 und ¬ (b ϵ H(A))
  - A x B < 0 und b ϵ H(A)
- Vorrücken von *A*: falls
  - A x B < 0 und ¬ (a ϵ H(B))
  - A x B > 0 und a ϵ H(B)

Sollten beide Vektoren auf den jeweils anderen zeigen, kann man einen beliebigen vorrücken.
Wenn keiner der beiden auf den anderen zeigt, rücken wir den Vektor vor, welcher sich außerhalb der Halbebene des anderen befindet, sind beide außerhalb, dann kann man wieder frei wählen, welcher vorrückt.
Die Zusammenfassung aller Fälle ergibt folgende Tabelle:

A x B	a ϵ H(B)	b ϵ H(A)	Vorrücken von ...
> 0	T	T	A
> 0	T	F	A oder B
> 0	F	T	A
> 0	F	F	B
< 0	T	T	B
< 0	T	F	B
< 0	F	T	A or B
< 0	F	F	A

Fig. 5. Tab. 1

Gekürzt ergibt dies folgende Tabelle, welche die Sammlung der Regeln darstellt, die wir im Programmcode umsetzen:

A x B	Halbebenen – Bedingung	Vorrücken von ...
> 0	b ϵ H(A)	A
> 0	¬ (b ϵ H(A))	B
< 0	a ϵ H(B)	B
< 0	¬ (a ϵ H(B))	A

Fig. 6. Tab.2

(3.2) Die Grundvariablen

In der Funktion

```
polygon_intersection(const polygon& X,const polygon& Y)
```

deklarieren wir vor der Schleife, die die eigentliche Berechnung durchführt, noch einige benötigte Variablen (neben den bereits erwähnten Segment-Listen, deren Iteratoren, sowie deren Start- und Endpunkten).

```
list<point> poly_result;

double cross;

bool bHA, aHB;

point p;

tInFlag inflag = Unknown;

int aadv = 0, badv = 0;

int n = P.size();

int m = Q.size();
```

1. Die Liste *poly_result*, welche Objekte vom Typ *point* enthält, dient zum Abspeichern der Ergebnispunkte, welche den Eckpunkten des Schnitts von *P* und *Q* entsprechen.

2. Der Double-Wert *cross* steht dient lediglich zum Speichern und Auswerten des Ergebnisses aus der Kreuzproduktberechnung der Hauptschleife:

   ```
   cross =   (A_vector.xcoord() * B_vector.ycoord()) -
             (A_vector.ycoord() * B_vector.xcoord());
   ```

3. Die beiden Boolean-Variablen *bHA* und *aHB* erhalten das Ergebnis aus dem Test, ob $b \in H(A)$ oder $a \in H(B)$, welchen wir mit Hilfe eines Orientation-Tests (*orientation()*-Funktion aus LEDA auf einem *segment* und einem *point*) durchführen. Dieser Test ist aus der Funktion *polygon_intersection(...)* ausgegliedert in der Funktion:

   ```
   bool LeftOn(point x, point y, point z);
   ```

   Diese Funktion liefert das Ergebnis *true* bzw. *false* zurück.

4. *point p* speichert den aktuellen Schnittpunkt der Segmente und wird in der großen While-Schleife berechnet durch die LEDA-Funktion

   ```
   (*A).intersection((*B), p)
   ```

5. Die Variable *inflag* vom benutzerdefinierten Typ *tInFlag* kann die Werte *Unknown*, *Pin* und *Qin* annehmen. *Pin* bedeutet, Polygon `P` ist das innere der beiden Polygone, *Qin* bedeutet entsprechendes für *Q*. *Unknown* ist selbsterklärend.

6. *aadv* und *badv* sind lediglich Integer counter-Variablen, die die Anzahl der durchlaufenen Polygonsegmente zählen. Sie dienen als Abbruchkriterium für die While-Schleife, sobald für ein Polygon alle Segmente durchlaufen sind.

   ```
   while ((aadv < n) || (badv < m)) { … }
   ```

7. *int n* und *int m* speichern die Anzahl der Eckpunkte/Segmente der beiden Polygone. Bestimmt wird diese mit der Funktion *size()* auf einem Polygon-Typ.

(3.3) Die Hauptschleife „while“

Die bereits angesprochene While-Schleife startet mit einer Abbruchbedingung für den Fall, dass beide Polygone bereits einmal komplett durchlaufen sind, es aber noch nicht zum Setzen des inflags *Pin* oder *Qin* gekommen ist. Das bedeutet dann, dass es sich bei dem aktuell berechneten Beispiel um einen der drei folgenden Sonderfälle handelt:

- *P* enthält *Q* komplett
- *Q* enthält *P* komplett
- Die beiden Polygone schneiden sich in keinem Punkt

Sollte ein solcher Fall erkannt werden, wird die while-Schleife sofort mit folgender Bedingung verlassen und die Sonderfälle werden später behandelt.

```
if (((aadv == n) || (badv == m)) && (inflag ==
Unknown)) break;
```

Im Folgenden werden den Variablen *a, a1, b,* und *b1* Ihre aktuellen Werte zugewiesen werden, abhängig davon, welches gerade das aktuelle Segment von Polygon *P* bzw. *Q* ist, dh. auf welches Element Iterator *A* und *B* gerade zeigen:

```
a1 = (*A).start();

a  = (*A).end();

b1 = (*B).start();

b  = (*B).end();
```

Um das in 2. von (3.2) genannte Kreuzprodukt berechnen zu können, werden die beiden aktuellen Segmente zu Vektoren umgewandelt und den beiden Variablen *A_vector* und *B_vector* vom Typ *vector* zugewiesen. Die Umwandlung findet durch die LEDA-Funktion *to_vector()* statt.

```
vector A_vector = (*A).to_vector();

vector B_vector = (*B).to_vector();
```

Das Kreuzprodukt berechnet sich wie folgt:

```
cross = (A_vector.xcoord() * B_vector.ycoord()) -

        (A_vector.ycoord() * B_vector.xcoord());
```

Nun wird der in 3. von (3.2) gezeigte Halbebenentest durchgeführt:

```
bHA = LeftOn(a1, a, b);

aHB = LeftOn(b1, b, a);
```

In der darauf folgenden If-Abfrage wird getestet, ob sich die beiden aktuellen Segmente in einem Punkt schneiden. Sollte dem so sein, werden die beiden Counter-Variablen *aadv* und *badv* auf 0 gesetzt. Sollte nicht sofort im ersten Schleifendurchlauf ein Schnitt stattfinden, werden die Variablen im x-ten Durchlauf genullt, sobald die Segmente sich angenähert haben.
Damit wir den gefundenen Ergebnispunkt *p* (in dem Fall ein konkreter Schnittpunkt) abspeichern, „pushen“ wir ihn in unsere Ergebnisliste *poly_result*.

Desweiteren wird hier mit Hilfe der Funktion

```
tInFlag InOut(tInFlag inflag, bool aHB);
```

getestet, welches der beiden Segmente aktuell innerhalb der Halbebenen der anderen liegt.

Die Hauptaufgabe, die in der Schleife erledigt wird, ist die Bestimmung, welches der beiden Segmente um eins vorgerückt wird (Advance rules).
Hier findet man im Groben eine if-else-Anweisung, welche im if- und im else-Teil nochmals in if-else aufgeteilt ist. Dies entspricht der Tabelle in Fig.6. Hier werden also die verschiedenen Fälle überprüft.
Es wird in jedem der Fälle die Variable *inflag* daraufhin überprüft, ob sie auf *Pin* bzw. *Qin* gesetzt ist. Ist dies der Fall, wird der so ermittelte Ergebnispunkt (*a* oder *b*) in der Ergebnisliste abgespeichert.
Hier wird also dafür Sorge getragen, dass Punkte des Schnitt-Polygons abgespeichert werden, welche nicht durch einen direkten Schnitt von zwei Segmenten entstehen.

Desweiteren wird innerhalb der if-Abfragen *aadv* bzw. *badv* um eins erhöht und das jeweilige Segment (siehe Tabelle) um eines vorgerückt durch *++A* bzw. *++B*.

Sollte ein Iterator nun auf das Ende seiner Liste (*.end()*) zeigen, wird er auf das erste Listenelement gesetzt (*.begin()*).

(3.4) Sonderfallbehandlung

Ist die While-Schleife komplett abgearbeitet, stehen alle Ergebnispunkte in *poly_result*, es sei denn, bei dem ausgewerteten Beispiel handelt es sich um einen der bereits angesprochenen drei Sonderfälle.

- **Sonderfall 1:**
  Überprüfung, ob Polygon *Q* das Polygon *P* enthält. Dies geschieht in folgender *forall_vertices*-Schleife aus LEDA.

```
forall_vertices(exc, P) {

   intersection1 = exception.ray_crossings(Q,exc);

   if (intersection1 == false) break;

}
```

  Es wird mit Hilfe des Ray-Crossing-Algorithmus (auf dem Objekt *exception*) getestet, ob alle Punkte von *P* in *Q* liegen. Beim ersten Punkt, für den diese Bedingung nicht zutrifft, wird die Schleife abgebrochen, was bedeuten würde, *P* kann nicht mehr komplett in *Q* liegen.
- **Sonderfall 2:**
  Entspricht exakt Nr. 1, außer dass hier getestet wird, ob *Q* in *P* liegt.
- **Sonderfall 3:**
  Liegt keiner der beiden ersten Fälle vor, handelt es sich um den Fall, dass beide Polygone sich in keinem Punkt schneiden.

Für Fall 1 und 2 werden noch alle Eckpunkte des inneren Polygons in die Ergebnisliste gepusht, um sie am Ende der Funktion ausgeben zu können.

(3.5) Das Ergebnis

Das Ergebnis wird am Ende der Funktion *polygon_intersection* auf zweierlei Arten dargestellt.
Einmal werden alle Ergebnispunkte auf der Konsole ausgegeben, mit dem kleinen Zusatz, um wie viele Punkte es sich im Gesamten handelt.

Zweitens wird durch

```
return poly_result;
```

die Ergebnisliste an den Aufrufer der Funktion zurückgegeben, wo das Ergebnis dann auch graphisch dargestellt werden kann.

(4) Die main-Methode

(4.1) Allgemeines zum Programm

Die in Kapitel (2) und (3) beschriebenen Algorithmen zu ’Point-in-Polygon’ und zum ’Schnitt konvexer Polygone’ können einerseits mit benutzerdefinierten, andererseits aber auch mit vorgegebenen Polygonen getestet werden. Eine entsprechende Auswahl findet der Benutzer unter den Menüpunkten ’Point in Polygon’ und ’Intersection’. Je nach Auswahl wird der Benutzer ggf. aufgefordert weitere Parameter wie z.B. Anzahl der Testpunkte, Rekursionstiefe für Hilbert-Polygone und die Eckenanzahl der n-Gone über ein LEDA-Panel einzugeben.

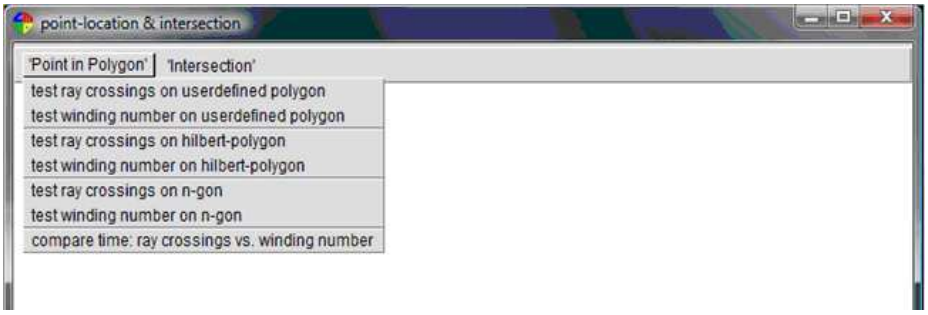


Fig. 7. Screenshot des ersten Menüs

Fig. 8. Screenshot des zweiten Menüs

Im Wesentlichen besteht unsere main-Methode aus einer While-Schleife, in der wir alle möglichen Benutzereingaben abfangen und bearbeiten.

```
while ((but = W.read_mouse()) != 0)     {

   switch (but)

      { case 1: … }

}
```

Es gibt neun verschiedene Auswahlmöglichkeiten für den Benutzer (case 1 bis 9).

(4.2) Point-in-Polygon auf benutzerdefinierten Polygonen

In den ersten beiden Fällen besteht für den Benutzer die Möglichkeit, den Ray Crossings – Algorithmus bzw. den Winding Number – Algorithmus auf einem beliebigen, per Mausklick eingegebenen, Polygon zu testen. Die Anzahl der Testpunkte kann mit Hilfe eines LEDA-Panels eingegeben werden.

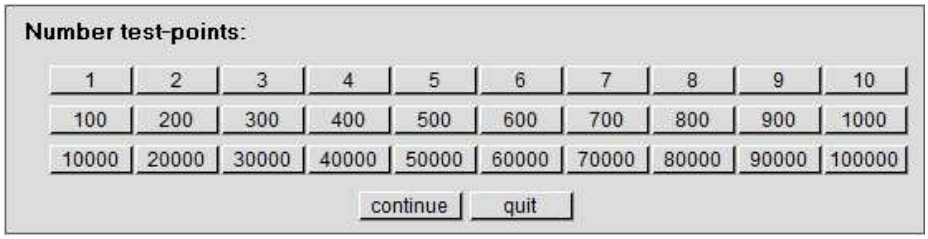


Fig. 9. Auswahl der Punktanzahl

Die Funktion

```
polygon create_polygon()
```

erstellt aus den vom Benutzer eingegebenen Punkten ein Polygon.

Die Zufallspunkte erzeugen wir mit der Funktion

```
list<point> generate_points(int n)
```

und speichern diese in einer Punktliste ab. Diese Funktion basiert auf einer C++ random-Funktion.

Anschließend lassen wir den Iterator *it* über diese Liste laufen und rufen mit

```
check = demo1.ray_crossings(P,*it);
```

bzw.

```
check = demo2.winding_number(P,*it);
```

für jeden dieser Punkte den entsprechenden Point-in-Polygon-Algorithmus auf. Dadurch erhält man für jeden Testpunkt einen Wahrheitswert, den wir in der Variable *check* speichern. Anschließend werden die Punkte, je nach Wert der Variable *check*, unterschiedlich farbig eingefärbt und im LEDA-Fenster ausgegeben.

```
if (check==true) W.set_color(LEDA::green);

else W.set_color(LEDA::red);

W << *it;
```

In *Case 7* kann man gezielt Tests zum Geschwindigkeitsvergleich der Algorithmen durchführen lassen.

Die Messung der Zeit läuft folgendermaßen ab:
Zuerst sind drei Variablen *time1*, *time2* und *difference* vom Typ float definiert. Die Variable *time1* speichert die aktuelle Zeit, nach dem Durchlauf von Winding Number bzw. Ray Crossings wird die dann aktuelle Zeit in *time2* gespeichert und anschließen in *difference* die Differenz der beiden Werte gebildet. Somit erhalten wir die vom jeweiligen Algorithmus benötigte Zeit, die wir dann auf der Konsole ausgeben.

```
float time1, time2, difference;

time1 = used_time();

//… Aufruf des Winding Number bzw. Ray Crossings …

time2 = used_time();

difference = time2-time1;

cout << "\ntime in seconds:\t" << difference;
```

(4.3) Point-in-Polygon auf komplexen Polygonen

In den Fällen 3 bis 6 ermöglichen wir dem Benutzer die Point-in-Polygon Algorithmen auch auf Hilbert- und n-Gonen zu testen.

Anders als in den ersten beiden Fällen, bei denen der Benutzer das Polygon eingeben konnte, erzeugen wir die Polygone hier mit Hilfe von in LEDA vordefinierten Funktionen:

```
POLYGON hilbert(int n, RAT_TYPE x1, RAT_TYPE x2,
RAT_TYPE y1, RAT_TYPE y2 )
```

bzw.

```
POLYGON n_gon( int n, CIRCLE C, double epsilon);
```

Die Parameter, wie z.B. Rekursionstiefe für Hilbert-Polygone, Anzahl der Ecken für n-Gone und Anzahl der Testpunkte können über ein LEDA-Panel variiert werden.

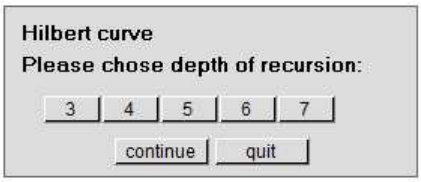


Fig. 10. Auswahl der Rekursionstiefe

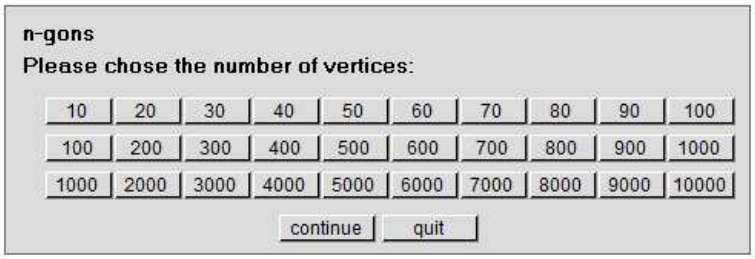


Fig. 11. Auswahl der Ecken-Anzahl

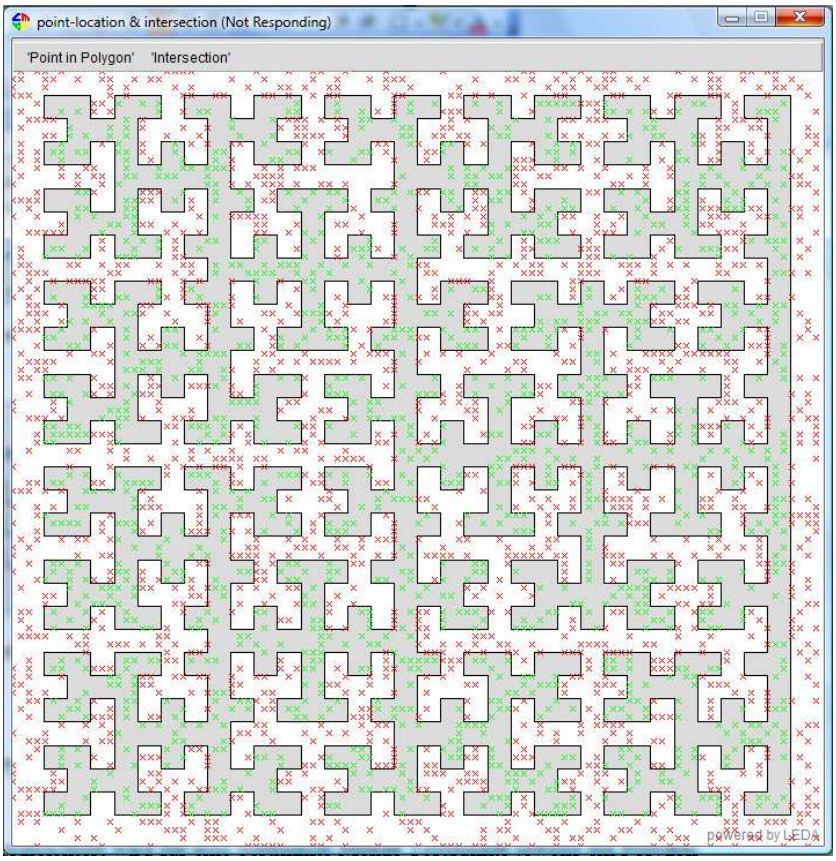


Fig. 12. Beispiel für ein Hilbert-Polygon der Tiefe 5

(4.4) Vergleichstests „Ray Crossings“ vs. „Winding Number“

Für den Vergleichtest wird für eine bestimmte Menge Testpunkte jeweils ein Polygon erzeugt und dann mit Hilfe des „Ray Crossings“- bzw. „Winding Number“-Algorithmus getestet, wie lange der jeweilige braucht, um für alle Punkte zu entscheiden, ob sie innerhalb oder außerhalb des Polygons liegen.

Den ersten Teil des Tests führen wir auf n-Gonen durch, deren Anzahl an Eckpunkten sich immer weiter erhöht, von 10 auf 1000 in 10er-Schritten.
Der nächste Test findet auf Polygonen statt, die mit Hilfe der rekursiven Hilbert-Funktion erzeugt werden (Komplexität 1 bis 8).

Alle gemessenen Zeiten werden in die Datei *times.txt* (im Programm-Ordner) geschrieben. Am Ende des gesamten n-Gon- bzw. Hilbert-Polygon-Tests wird die Durchschnittszeit ausgegeben, die für die Berechnung benötigt wurde.

Um nicht unnötige Zeit zu verbrauchen, wird hier natürlich komplett auf eine graphische Ausgabe verzichtet.
Obwohl beide Algorithmen theoretisch die Laufzeit O(n) haben, ist der Ray Crossings-Algorithmus in der Praxis etwas schneller als Winding Number. Dies liegt daran, dass Ray Crossings im Wesentlichen nur auf Vergleichen beruht, während Winding Number Fließkomma- und trigonometrische Berechnungen durchführt.

(4.5) Schnitt von konvexen Polygonen

Der Benutzer hat hier die Möglichkeit die beiden Polygone *P* und *Q* selbst per Mausklick einzugeben. Da der von uns verwendete Algorithmus nur auf konvexe Polygone ausgelegt ist, stellen wir mit der LEDA-Funktion *Q.is_convex()* bzw. *P.is_convex()* sicher, dass nur konvexe Polygone vom Benutzer eingegeben werden. Notfalls wird er solange zur Neueingabe aufgefordert, bis die Eingabe den Anforderungen entspricht.

Desweiteren kann ein auf n-Gonen basierendes Demo gestartet werden. Hier wird das erste Polygon *P* durch die LEDA-Funktion für n-Gone erzeugt und das Polygon *Q* ergibt sich durch eine Drehung von *P* um seinen Mittelpunkt.

```
point center(50,50);

circle C(50,50,35);

P = n_gon(ecken, C, 0.1);

Q = P.rotate(center,3.3);
```

Anschließend wird die Schnittfunktion aufgerufen. Diese liefert die Liste mit den Schnittpunkten, aus der dann das Ergebnispolygon erzeugt wird, welches wiederum rot eingefärbt wird.

```
Intersection schnitt;

list<point> schnittpunkte =
schnitt.polygon_intersection(P,Q);

polygon result(schnittpunkte);

W.draw_filled_polygon(result,red);

W << result;
```

(5) Die Dateien

- 2d_polygons.cpp: Einlesen von Polygonen, Generieren von Zufallspunkten, Menüsteuerung, graphische Ausgabe, ...
- Ray_crossings.h: Implementierung des Ray Crossing-Algorithmus
- Winding_number.h: Implementierung des Winding Number-Algorithmus
- Intersection.h: Berechnung des Schnittes von Polygonen