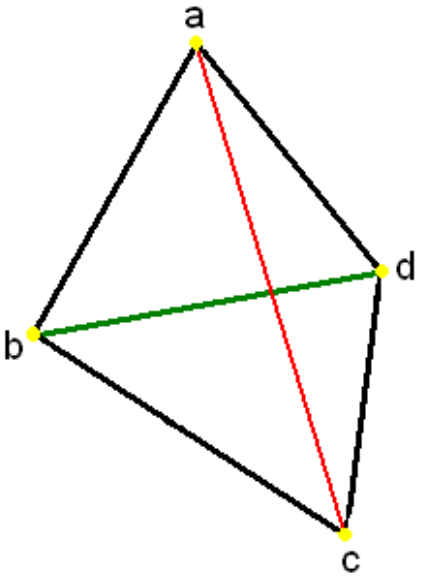
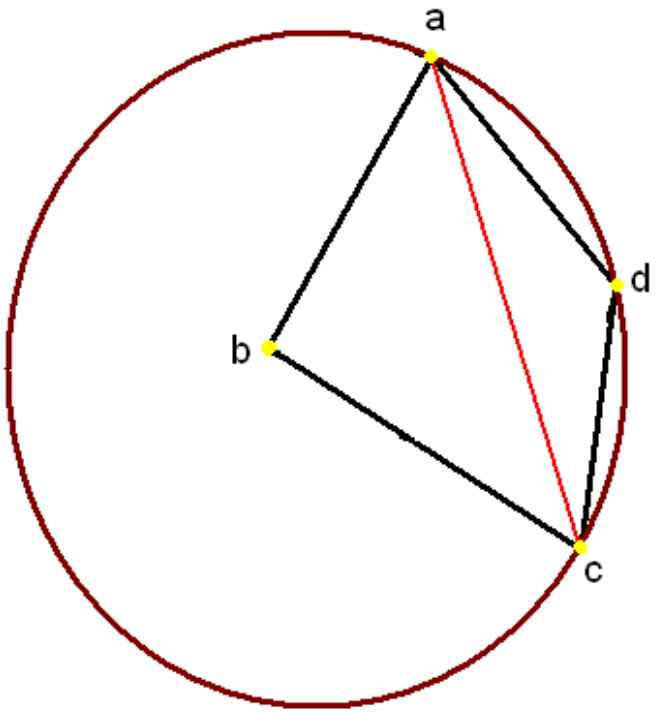
Delaunay-Triangulierung

- Gegeben ist eine Punktmenge $L = \{ l_1, \ldots, l_n \}$.
- Triangulierung T der konvexen Hülle:
- $\forall$ Dreiecke l_i, l_j, l_k aus T
  $\Rightarrow$ kein weiterer Punkt l_m aus L, innerhalb des Kreises durch l_i, l_j, l_k.



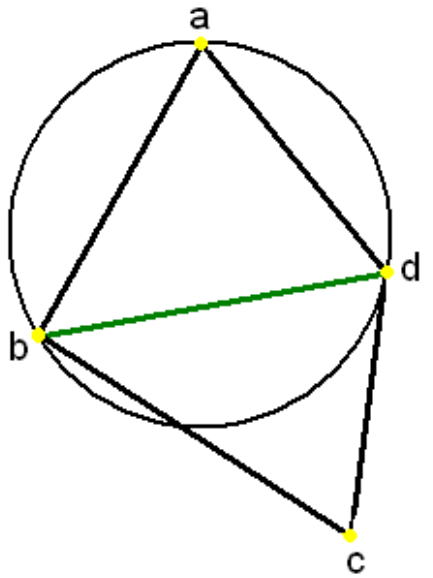
Delaunay-Triangulierung

- Umkreis um Dreieck acd enthält Punkt b
- Delaunay-Bedingung verletzt
- Seite „flippen“: ac ⇒ bd



Delaunay-Triangulierung

- Umkreis um abd enthält c nicht
- Korrekte Triangulierung



- Die Triangulierung T ist planarer Graph eingebettet in L

Vorgabe

- Benutzung der LEDA-Bibliothek
- Viele Funktionen schon bereitgestellt
  - Interaktive Fenster
  - Möglichkeiten zur Animation
  - Graph-Datenstruktur
  - Punkte in der Ebene

Orientation- / Incircle-Test

- Gerade durch q und r
- Test, auf welcher Seite s liegt

$$det(A) = \begin{vmatrix} q_x & q_y & 1 \\ r_x & r_y & 1 \\ s_x & s_y & 1 \end{vmatrix}$$

  - det(A) > 0 ⇒ s links
  - det(A) = 0 ⇒ s kollinear
  - det(A) < 0 ⇒ s rechts

- Kreis durch q, r, s
- O.B.d.A. q,r,s im Uhrzeigersinn
- Test, ob p enthalten ist

$$det(B) = \begin{vmatrix} 1 & q_x & q_y & q_x^2+q_y^2 \\ 1 & r_x & r_y & r_x^2+r_y^2 \\ 1 & s_x & s_y & s_x^2+s_y^2 \\ 1 & p_x & p_y & p_x^2+p_y^2 \end{vmatrix}$$

  - det(B) ≥ 0 ⇒ p in Kreis

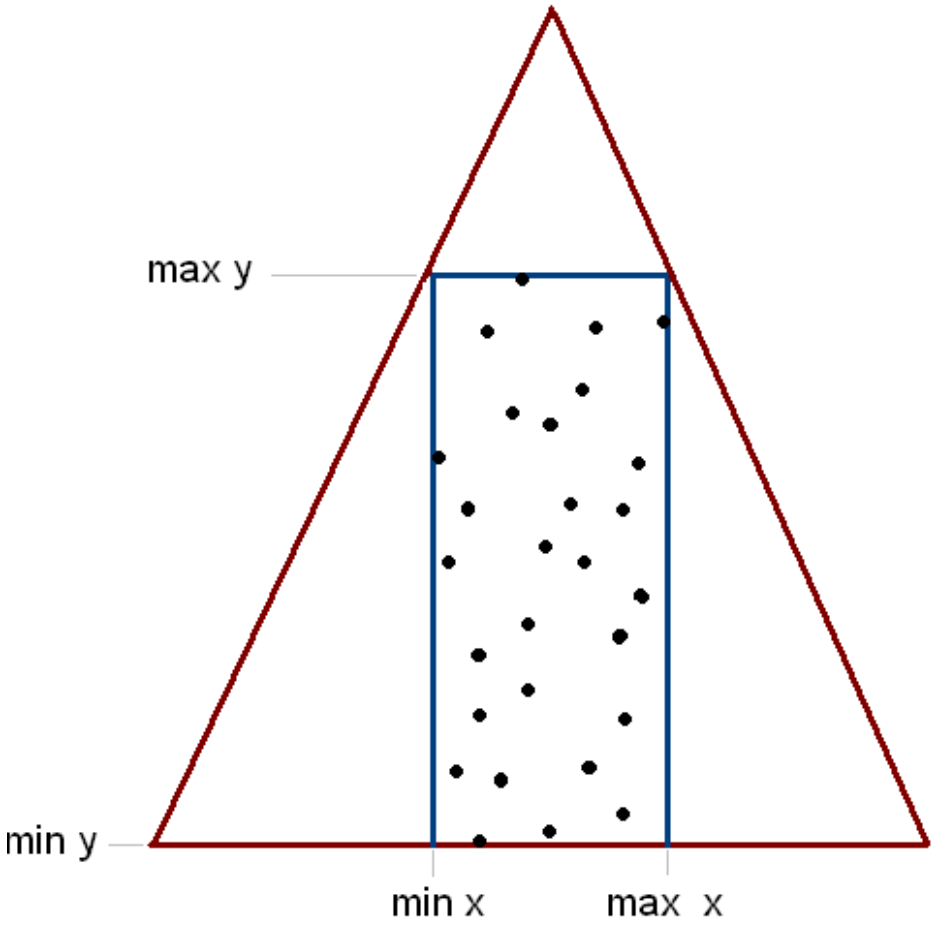
Algorithmus

```
set<point> L;     /* Eingabemenge */
Graph G;          /* initial leer */

point s = init(L);
while (!L.empty()) {
  point p = L.pop();
  triangle D_dest = find_face(s, p);
  triangulate(D_dest, p);
  s = p;
}
```

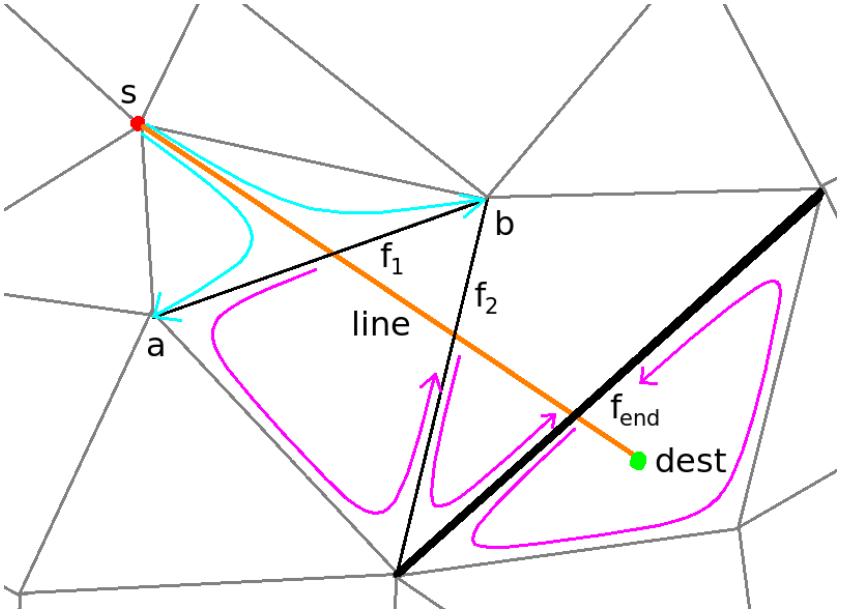
Initialisierung

- Durchsuche Punktmenge nach Min-/Max-Werten in X-/Y-Richtung
- Dreieck um Punkte
- Grundfläche = $2 \bullet \Delta x$
- Höhe = $max_y + \Delta y/2$
- Spitze des Dreiecks ist Startpunkt s

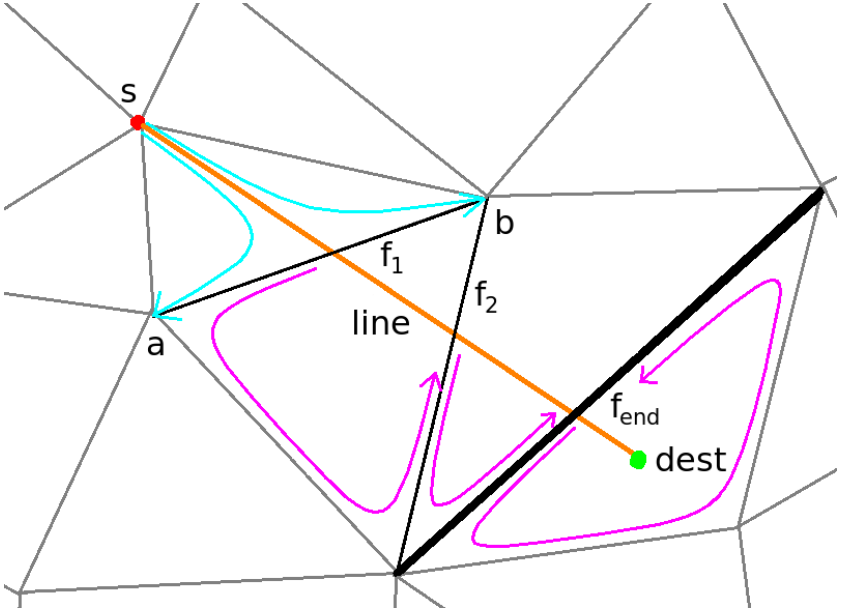


Suche des neuen Punkts

- bilde Strecke von Startpunkt s zum Punkt dest
- finde Startdreieck (mit 2 Orientation-tests)
- f_1:= Kante des Startdreiecks, die gegenüber von s liegt



- Repeat
  - laufe das an f_i angrenzende Dreieck entgegen des Uhrzeigersinns ab, bis eine Kante die Strecke schneidet
  - setze f_{i+1} auf diese Kante
- until f_{i+1} existiert nicht
- return $f_{end} := f_i$



Bemerkung: Da in jedem Face höchstens zwei Seiten von der Strecke geschnitten werden, ist der Weg eindeutig bestimmt.

Triangulierungsschritt

- Füge Knoten n für dest ein
- ∀ Kanten von X → Stack s
- while (!s.empty())
  - (a,b) ← s.pop()
  - Finde ggü.-liegenden Knoten c
  - if (indest(dest, a, b, c)
    - Lösche (a,b) aus G
    - s.push((c,b))
    - s.push((a,c))
  - else Füge (a,n) in G ein