DD2458, Problemlösning och programmering under press

Föreläsning 11: Beräkningsgeometri

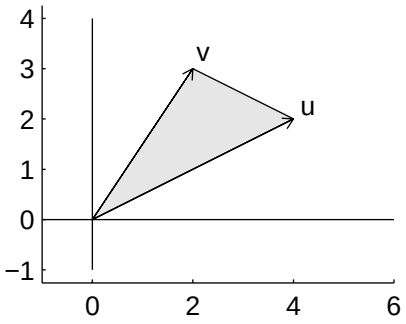
Datum: 2007-11-27
Skribent(er): Fredrik Atmer
Föreläsare: Mikael Goldmann

Den här föreläsningen skrapar lite på ytan av området beräkningsgeometri. Vi går igenom hur man beräknar arean av polygoner, hittar konvexa höljen till punktmängder, avgör huruvida linjer skär varandra och var punkter ligger i förhållande till varandra.

1 Vektorgeometri

Först en användbar vektoroperation. Punkten (x, y) i planet kan beskrivas som en vektor $(x, y, 0)$ i rummet. Om vi tar kryssprodukten $u \times v$ mellan vektorerna $u = (u_x, u_y, 0)$ och $v = (v_x, v_y, 0)$ får vi en vektor $w = (0, 0, u_x v_y - v_x u_y)$ där z-komponenten är dubbelt så stor som arean på triangeln som u och v spänner upp (Figur 1). Tecknet på z-komponeneten kommer att markera om vinkeln mellan u och v är positiv eller negativ.

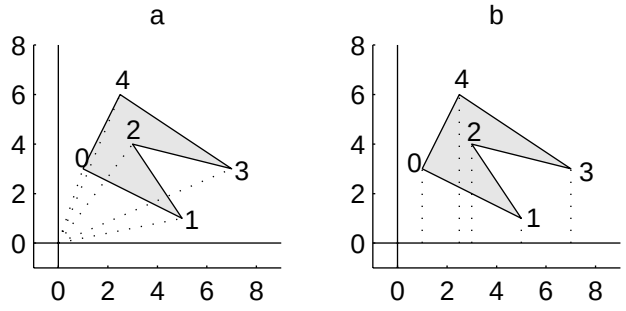
$$\begin{aligned} w &= (u \times v) = \begin{vmatrix} \hat{x} & \hat{y} & \hat{z} \\ x_1 & y_1 & 0 \\ x_2 & y_2 & 0 \end{vmatrix} = (x_1 y_2 - x_2 y_1)\hat{z} \\ A &= \frac{1}{2}(x_1 y_2 - x_2 y_1) \end{aligned}$$



Figur 1: Vektorerna $u = (4, 2)$ och $v = (2, 3)$. $A = 0.5(4 \cdot 3 - 2 \cdot 2) = 4$

Resonemanget går att generalisera till n dimensioner och arean för hypertriangeln som vektorerna $u_1, u_2, \ldots u_n$ spänner upp blir

$$w = \begin{vmatrix} \hat{x}_1 & \hat{x}_2 & \cdots & \hat{x}_{n+1} \\ u_{11} & u_{12} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ u_{n1} & u_{n2} & \cdots & 0 \end{vmatrix} \qquad A = \frac{1}{n!}\|w\|$$



Figur 2: Beräkning av arean mha (a) trianglar utgående från origo. (b) parallellogram under polygonens kanter (Greens formel)

vilket är vad vi brukar kalla för volym när $n = 3$.

2 Polygonarea

Nu kan vi använda vår nyvunna kunskap om vektorer för att beräkna arean av en godtycklig, enkel, polygon. Givet en ordnad mängd koordinater (x_i, y_i) för polygonens n st hörn blir arean lika med summan av alla trianglar som spänns upp av vektorerna (x_i, y_i) och (x_j, y_j) där $j = (i + 1) \bmod n$ (Figur 2a). (Vi utelämnar z-koordinaterna av praktiska skäl.)

$$A = \frac{1}{2} \sum_{i=0}^{n-1} \|v_i \times v_j\|, \quad j = (i + 1) \bmod n$$

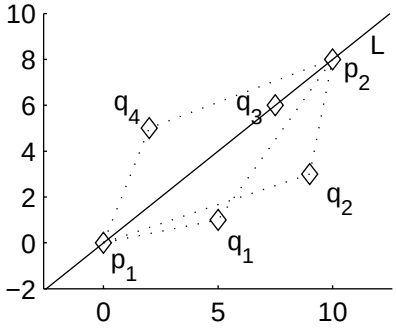
Arean på trianglarna vi lägger till kommer att få olika tecken beroende på om vi är på bak- eller framsidan av polygonen från origo sett. Detta gör att den yta vi eventuellt lagt till extra på vägen bakom polygonen dras av igen när vi går förbi åt andra hållet på framsidan. Tecknet på den totala arean kommer att bli positiv om polygonens hörn är givna i moturs ordning och negativ om dom är givna i medurs ordning.

Ett annat sätt att beräkna polygonens area är att använda Greens formel. Vi beräknar arean mellan alla linjesegment och x-axeln (Figur 2b). Beroende på om x_j är större eller mindre än x_i kommer vi att få ett positivt eller negativt bidrag till arean. Detta gör att kanter på undersidan av polygonen bidrar med areor av motsatt tecken mot kanter på ovansidan.

$$A = \frac{1}{2} \sum_{i=0}^{n-1} (y_i + y_j)(x_i - x_j), \quad j = (i + 1) \bmod n$$

Även här kommer tecknet på arean återspegla om hörnen är givna i negativ eller positiv ordning.

Om polygonens hörn har heltalskoordinater kan vi även beräkna dess area med hjälp av *Picks sats* genom att räkna antalet heltalspunkter som ligger inuti polygonen, I, och antalet punkter som ligger på randen till polygonen, R. Arean blir nu $A = I + \frac{1}{2}R - 1$. Mer användbart är antagligen omvändningen av sambandet, att



Figur 3: Areorna $area(p_1, p_2, q_1)$ och $area(p_1, p_2, q_2)$ är båda mindre än noll dvs punkterna q_1 och q_2 ligger på samma sida om L. Arean $area(p_1, p_2, q_4)$ däremot är positiv och ligger mao på andra sidan L jämfört med q_1 och q_2. Arean $area(p_1, p_2, q_3)$ är lika med noll och q_3 ligger alltså på L.

kunna räkna ut hur många heltalspunkter som ligger inuti en polygon utifrån dess area.

3 På samma sida av linjen?

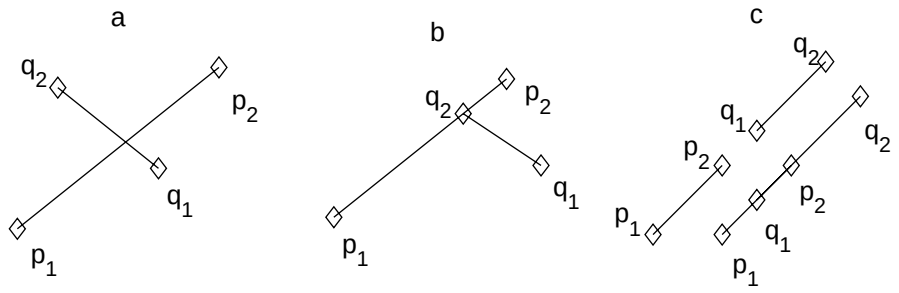
För att undersöka om två punkter ligger på samma sida om en linje kan vi använda vårt uttryck för polygonareor. Antag att vi har en linje L bestämd av två punkter p_1 och p_2 och två punkter q_1 och q_2 som vi vill undersöka om dom ligger på samma sida om L (Figur 3). Vi beräknar arean av triangeln $p_1p_2q_1$ och triangeln $p_1p_2q_2$. Om areorna har samma tecken ligger punkterna på samma sida om linjen.

På samma sätt kan vi enkelt kontrollera om tre punkter p_1, p_2 och p_3 ligger på linje genom att beräkna arean av triangeln $p_1p_2p_3$. Om arean är noll ligger punkterna antingen på samma linje eller så är $p_1 = p_2 = p_3$ och punkterna ligger på alla linjer som går igenom dom.

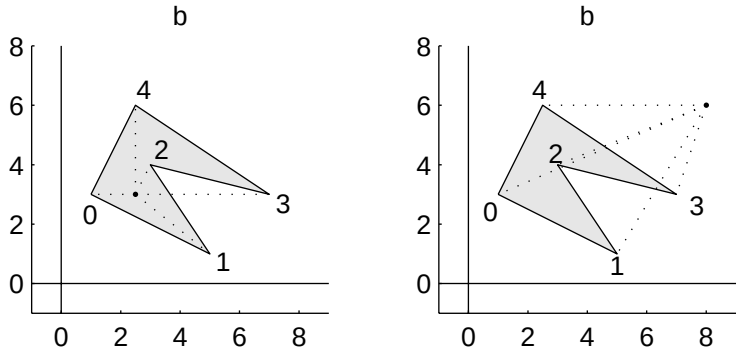
Ett något mer invecklat problem som kan lösas med samma metod är att bestämma om två ändliga linjesegment skär varandra eller inte. Antag att p_1, p_2 respektive q_1, q_2 är ändpunkter på linjesegmenten L_p och L_q (Figur 4). Om p_1 inte ligger på samma sida av L_q som p_2 **och** q_1 inte ligger på samma sida av L_p som q_2 skär segmenten med säkerhet varandra (punkter på linjerna ligger inte på någon sida av dom). I specialfallet att alla punkter ligger på samma linje måste vi kontrollera om segmenten överlappar varandra eller inte. För att göra det kan vi kontrollera om någon av punkterna q_1 eller q_2 ligger mellan p_1 och p_2 **eller** någon av punkterna p_1 eller p_2 ligger mellan q_1 och q_2. I det första fallet är detta sant om båda olikheterna $\|p_2 - p_1\| \geq \|p_1 - q_i\|$ **och** $\|p_2 - p_1\| \geq \|p_2 - q_i\|$ är uppfyllda och det andra fallet helt analogt.

4 Inuti polygonen?

Om vi vill kontrollera om en punkt p ligger inuti eller utanför en polygon finns det ett par olika tillvägagångssätt (Figur 5). Kalla hörnen i polygonen för $q_0, q_1, \ldots q_{n-1}$



Figur 4: (a) Segmenten skär varandra, $area(p_1, p_2, q_1)$ och $area(p_1, p_2, q_2)$ har skilda tecken **och** $area(q_1, q_2, p_1)$ och $area(q_1, q_2, p_2)$ har skilda tecken. (b) Här är visserligen $area(p_1, p_2, q_1) = 0$ men $area(p_1, p_2, q_2) \neq 0$. Så segmenten skär fortfarande varandra. (c) Två av specialfallen där alla punkter ligger på samma linje.



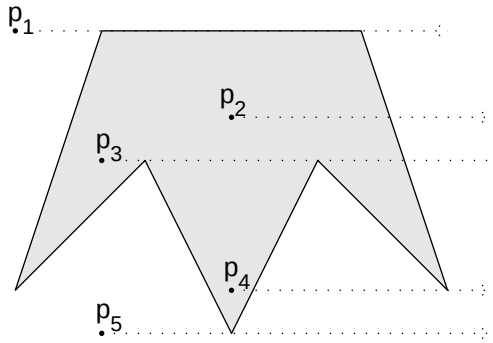
Figur 5: (a) Punkten ligger inuti polygonen, summan av vinklarna mellan två på varandra följande hörn är 2π (b) Punkten ligger utanför polygonen och summan av vinklarna är 0.

och beräkna summan av vinklarna mellan q_i och q_j med spets i p där $j = (i + 1) \bmod n$.

$$\sum_{i=0}^{n} angle(q_i - p, q_j - p)$$

Om punkten ligger inuti polygonen kommer linjen pq_i att ha svept ett helt varv runt p när vi kommer tillbaks till hörnet vi startade i och summan blir $\pm 2\pi$. Ligger punkten däremot utanför polygonen kommer linjen att göra en svepande rörelse lika mycket medurs som moturs och summan blir 0. Om p, q_i och q_j ligger på samma linje måste vi vara lite försiktiga. Är skalärprodukten $\langle q_i - p, q_j - p \rangle \leq 0$ så ligger p på randen till polygonen. Annars kan vi sätta vinkeln mellan q_i och q_j till 0 och fortsätta som vanligt.

Ett annat sätt att se om en punkt ligger inuti en polygon är att dra en linje från punkten parallellt med x-axeln bort mot oändligheten och räkna hur många av polygonens segment $q_i q_j$ som linjen korsar (Figur 6). Är det ett jämnt antal så ligger



Figur 6: p_1 har 2 passager, en för varje ändpunkt på sidosegmenten (det horisontella segmentet räknas inte), p_2 har en enkel passage, p_3 har 5 passager, två i varje spets och en i sidosegment. För p_4 kommer endast den första passagen räknas och för p_5 kommer ingen passage att räknas.

punkten utanför polygonen och är det ett udda antal så ligger den inuti. Här kan det uppstå problem om linjen råkar gå precis genom ett hörn q_i eller sammanfaller med ett segment q_iq_j. För att komma till rätta med dessa fall ignorerar vi horisontella kanter och räknar bara den ändpunkt med störst y-koordinat som tillhörandes varje segment.

5 Konvext hölje

Givet en mängd med n st punkter kan vi med hjälp av en *Graham scan* i tid $O(n \log n)$ hitta den minsta konvexa polygon som omsluter alla punkter. Börja med att hitta den punkt p_0 som har minst y-koordinat. Om det är fler än en så ta den med minst x-koordinat. Sortera sedan punkterna med avseende på vinkeln $angle(\hat{x}, p_i - p_0)$. Om två punkter ligger på samma linje relativt p_0 så bestämmer vi att den som ligger närmst p_0 ska sorteras först. Det vi vill åstadkomma är en rand som precis innesluter alla punkter och hela tiden "svänger åt vänster". Det kan vi nu avgöra om den gör genom att titta på arean av triangeln $p_{i-2}p_{i-1}p_i$. Blir den ickepositiv har vi inte svängt åt vänster och punkten p_{i-1} kan förkastas. Uttryckt i algoritmform skulle det kunna se ut som Algoritm 1.

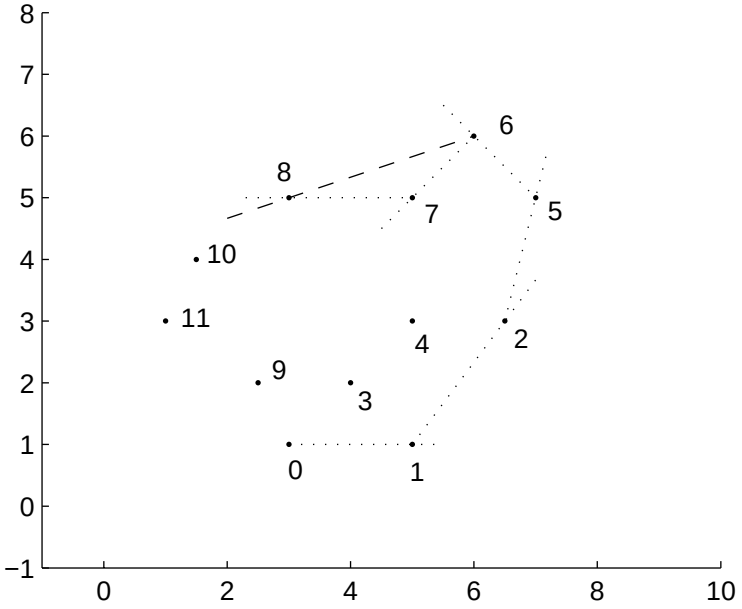
Algoritm 1:
Input: En punktmängd med ett minsta konvext hölje med area större än noll.
Output: Det minsta konvexa höljet till punktmängden.

```
convexHull(pointSet)
(1)   p2 ← pointSet.findLowestY().findLowestX()
(2)   stack.push(p2)
(3)   queue ← pointSet.sortByAngleAndDistance().toQueue()
(4)
(5)   p1 ← queue.poll()
(6)   while area(p2, p1, queue.peek()) = 0
(7)      p1 ← queue.poll()
(8)    stack.push(p1)
(9)
(10)  while not queue.empty()
(11)     p0 ← queue.poll()
(12)     while area(p2, p1, p0) ≤ 0
(13)        p1 ← p2
(14)        p2 ← stack.pop()
(15)     stack.push(p2)
(16)     p2 ← p1
(17)     p1 ← p0
(18)
(19)  stack.push(p2)
(20)  stack.push(p1)
(21)
(22)  return stack.toPolygon()
```

6 Närmsta par av punkter

Om vi vill hitta dom två punkter i en punktmängd i planet som ligger närmst varandra kan vi göra på följande sätt. Sortera punkterna efter deras x-koordinat och dela upp mängden så att hälften av punkterna hamnar i den vänstra delen och hälften i den högra. Upprepa rekursivt tills sidorna bara innehåller en enda punkt. Eftersom punkten nu är ensam säger vi att det minsta avståndet mellan två punkter i delmängden är oändligt.

När vi nu tar ett steg upp i rekursionen passar vi på att ”merga” ihop punktmängderna i de båda delarna så att punkterna blir sorterade efter deras y-koordinat. Sedan går vi igenom mängden vi får och plockar ut de punkter som ligger på avstånd högst $d = min(d_{Vmin}, d_{Hmin})$ från delningslinjen (det minsta avståndet mellan två punkter i vänster respektive höger del). Det smarta är att vi nu för varje av de utvalda punkterna bara behöver kontrollera avståndet till de sju direkt följande punkterna (sorterade efter y-koordinat). Om det ska finns någon punkt som ligger närmre än



Figur 7: Här ser vi ett steg i algoritmen. Hörn 0,1,2 och 5 ligger på stacken, 6 och 7 hänger i luften och vi har tagit 8 från kön. Eftersom vi just svängt åt höger kommer 7 att kastas och 5 plockas upp från stacken igen.

d från punkten vi tittar på måste den ligga på andra sidan delningslinjen och den måste dessutom ligga inuti en kvadrat med sidan d och i den får det plats högst fyra punkter med inbördes avstånd minst d. På samma sätt kan det finnas ytterligare tre punkter i en likadan kvadrat på samma sida av delningslinjen som punkten vi tittar på ligger. Alltså totalt sju punkter direkt följande i y-ordning.

Sedan är det bara att fortsätta upp i rekursionsträdet och hitta varje d_{VHmin} och jämföra med d_{Vmin} och d_{Hmin}. Algoritm 2 beskriver idén. Det tar $O(n \log n)$ tid att sortera punkterna efter deras x-koordinat. Tiden, $T(n)$, för ett anrop till D&C med en instans av storlek n är $O(n) + 2T(n/2)$ vilket ger att $T(n) \in O(n \log n)$. Totalt blir komplexiteten alltså $O(n \log n)$.

Algoritm 2:
Input: En punktmängd.
Output: Dom två punkter som ligger närmst varandra samt avståndet mellan dom.

```
closestPair(pointSet)
(1)   p1 ← null
(2)   p2 ← null
(3)   pointSet.sortByX()
(4)   [mindist, null] ← D&C(pointSet)
(5)   return [mindist, p1, p2]

      D&C(set)
(1)   if set.size() = 1
(2)       return [∞, set]
(3)
(4)   [d1, S1] ← D&C(set.lowerHalf())
(5)   [d2, S2] ← D&C(set.upperHalf())
(6)   d ← min(d1, d2)
(7)   S ← mergeByY(S1, S2)
(8)
(9)   m ← (set.lowerHalf().maxX()+set.upperHalf().minX())/2
(10)  T ← {p : p ∈ S ∧ m−d < p.x < m+d}
(11)
(12)  for i = 1 to max(T.size()−7, 1)
(13)      for j = i+1 to max(i+7, T.size())
(14)          if dist(pi, pj) < d
(15)              p1 ← pi
(16)              p2 ← pj
(17)              d ← dist(pi, pj)
(18)
(19)  return [d, s]
```

Om de n st punkterna är jämnt fördelade över ett delområde av planet kan vi istället dela in området i $\Theta(n)$ st lika stora kvadrater. Då får vi i genomsnitt en punkt per kvadrat. Sedan kontrollerar vi vad det minsta avståndet mellan två punkter i samma kvadrat är. Vi måste även kontrollera vad det minsta avståndet mellan två punkter från olika kvadrater är. Eftersom punkterna är jämnt fördelade vet vi att det minsta avståndet högst kan vara i storleksordningen av diagonalen på våra små kvadrater. Då räcker det att för varje punkt kontrollera avståndet till dom punkter som ligger i de 20 st närmsta kvadraterna. Vi har $\Theta(n)$ st kvadrater och eftersom det är ett konstant antal punkter i varje kvadrat kommer det att krävas konstant tid att gå igenom varje kvadrat. Totalt kommer algoritmen alltså att ta $O(n)$ tid.