Convex Hull - Gift Wrapping Algorithmus

Hannes Höttinger und Josua Kucher

FH Technikum Wien, Game Engineering und Simulation, Wien, AUT

1 Aufgabenstellung

Ziel ist die Implementierung und Visualisierung eines Algorithmus zur Berechnung von Konvexen Hüllen. Es wurde der *Jarvi's March* oder der sogenannte *Gift Wrapping* Algorithmus gewählt. Die Eingabedaten werden zufällig generiert (x,y Koordinaten). Es sollen folgende Varianten implementiert werden:

1. performanceoptimiert mit Ausgabe der konvexen Hüllen und Zeitmessung
2. visueller Einzelschrittmodus mit grafischer Darstellung der Punktmenge und der einzelnen Schritte wie die Hülle berechnet wird

2 Jarvi's-March (Gift-Wrapping) Algorithmus

Die Idee dieses Algorithmus ist es, die Menge der Punkte in der Ebene wie mit einer Schnur zu umwickeln. Begonnen wird mit einem Extrempunkt, der sicher auf der konvexen Hülle liegt, also zum Beispiel der Punkt mit der minimalen x-Koordinate. Von diesem Punkt ausgehend wird der "am weitesten rechts liegende" Punkt gesucht. Alle Punkte müssen links von der Verbindungsgeraden liegen. Von diesem Punkt ausgehend wird dieses Verfahren wiederum angewandt, solange bis wieder der Ausgangspunkt erreicht ist.

Analysiert man dieses Verfahren, wird zuerst ein Extrempunkt für den Start bestimmt. Diese Berechnung dauert für n gegebene Punkte $\mathcal{O}(n)$. Um nun die konvexe Hülle zu finden muss wie oben beschrieben der am weitesten rechts liegende Punkt für jeden Punkt auf der konvexen Hülle bestimmt werden. Die Berechnung für diesen Punkt ist mit $\mathcal{O}(n)$ definiert, da jedes mal n Punkte überprüft werden. Entscheidend für die Laufzeit ist weiters die Anzahl der Punkte h auf der konvexen Hüllen. Der Algorithmus wird somit h mal aufgerufen und ergibt somit insgesamt eine Laufzeit von:

$$Averagecase : \mathcal{O}(n \cdot h) \tag{1}$$

$$Worstcase : \mathcal{O}(n^2) \tag{2}$$

Die worstcase Laufzeit ergibt sich aus der Überlegung, dass die konvexe Hülle aus n Eckpunkten bestehen kann, wenn z.B. alle Punkte auf einem Kreisumfang liegen. Der Algorithmus würde somit n-mal aufgerufen werden.

3 Implementierung

In dieser Implementierung wurde auf den Einsatz von Winkelfunktionen aufgrund von Performanceoptimierungen verzichtet (Jarvi's March → der am weitesten rechts liegende Punkt = kleinster Winkel). Stattdessen wird eine Methode mittels Kreuzprodukt verwendet (Thomas H. Cormen, 2009).

Kreuzprodukt zweier Vektoren

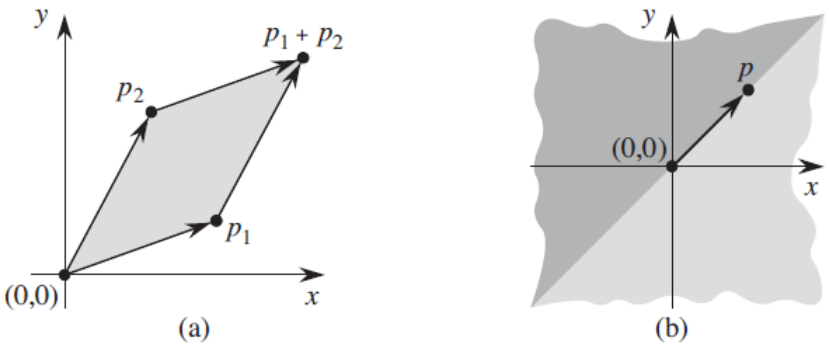

Abbildung 1: Das Kreuzprodukt zweier Vektoren. (**b**) Die helle Fläche beinhaltet Vektoren die im Uhrzeigersinn von p aus liegen. Die dunkle Fläche gegen den Uhrzeigersinn. (Thomas H. Cormen, 2009)

$$p_1 \times p_2 = det \begin{pmatrix} x_1 & x_2 \\ y_1 & y_2 \end{pmatrix} \tag{3}$$

$$= x_1 y_2 - x_2 y_1 \tag{4}$$

Wenn nun das Kreuzprodukt einen positiven Wert ergibt, dann liegt p_1 im Uhrzeigersinn von p_2. Bei einem negativen Wert entspricht dies einer Orientierung gegen den Uhrzeigersinn. 0 bedeutet die Vektoren sind kollinear. Um dies nun effektiv einsetzen zu können, wird nun ein Punkt p_0 als Ursprungspunkt definiert. In Abb. 2 wird die Orientierung nochmals veranschaulicht. Es werden drei Punkt gewählt, wobei die Orientierung von p_1 in Bezug auf p_0 und p_1 gefragt ist.

Es ergibt sich also:

$$(p_1 - p_0) \times (p_2 - p_0) = (x_1 - x_0)(y_2 - y_0) - (x_2 - x_0)(y_1 - y_0) \tag{5}$$

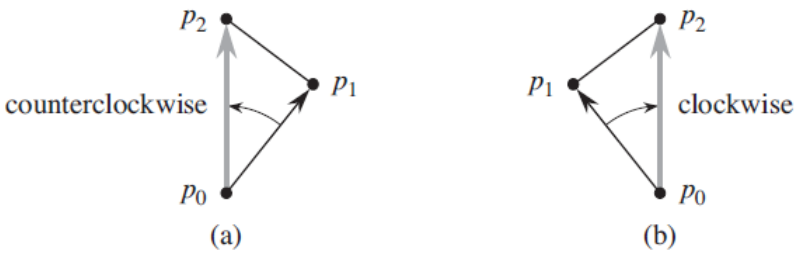

Abbildung 2: Das Kreuzprodukt zur Bestimmung der Orientierung von den angeordneten Segmenten $\overline{p_0p_1}$ und $\overline{p_1p_2}$ an dem Punkt p_1. (a) gegen den Uhrzeigersinn → Linksabbiegen. (b) im Uhrzeigersinn → Rechtsabbiegen (Thomas H. Cormen, 2009)

Code

In folgendem Codebeispiel 1 wird die implementiere Version der Orientierungsbestimmung gezeigt. Gefragt ist die Orientierung der Vektoren p,q und r (entspricht: $p_0 = p$, $p_1 = q$, $p_2 = r$), wobei das Rechtsabbiegen für die Hauptschleife des Algorithmus entscheidend ist (dieser Wert wird gespeichert, also wenn das Kreuzprodukt > 0 ist).

Listing 1: Orientierungsbestimmung mittels Kreuzprodukt

```
double Hull::Rightturn(Vector2 p, Vector2 q, Vector2 r)
{
    // cross product to determine polygon orientation
    double x1 = (q.x - p.x) * (r.y - p.y);
    double x2 = (r.x - p.x) * (q.y - p.y);
    double val = x1 - x2;

    if (val == 0)   return 0;
    if (val>0)      return 1;   // CW orientation
    else            return -1;
}
```

Die Hauptschleife hat als Abbruchbedingung, dass der nächste Punkt auf der Hülle der Anfangspunkt ist. Sobald eine Rechtsabbiegung gefunden wurde, wird p_1 mit p_2 überschrieben (da weiter rechts liegt). Dies wird für jeden Punkt der Menge durchgeführt und liefert somit den am meisten rechts liegenden Punkt von p_0 aus gesehen. Wurden alle Punkte verglichen wird p_0 auf den gefundenen Punkt p_1 gesetzt ($p = q$) und p_1 wird auf den nächst folgenden Punkt von p_0 in der Datenstruktur gesetzt. Dies geschieht solange bis $p_0 = Anfangspunkt$ (= konvexe Hülle geschlossen). Sind die Punkte kollinear, muss der Punkt p_2 weiter entfernt von p_0 als p_1 liegen.

Listing 2: Hauptschleife des Gift-Wrappings

```
do
{   // add left most point and the points to follow
    m_pointsHull.push_back(p);
    for (int r = 0; r<m_pointsCloud.size(); r++) {
        int ccw = Rightturn(m_pointsCloud[p], ↩
            m_pointsCloud[q], m_pointsCloud[r]);
        // if turn right -> set new point
        if (ccw == 1 || ccw == 0 && m_pointsCloud[p↩
            ].GetAbsDistance(m_pointsCloud[r]) > ↩
            m_pointsCloud[p].GetAbsDistance(↩
            m_pointsCloud[q]))
        {
            q = r;
        }
    }
    // set final q to next point on hull
    p = q;
    // go to next point in array after hullpoint
    q = (p + 1) % m_pointsCloud.size();
} while (p != l);
```

4 Ergebnisse

Zur Durchführung des Tests werden eine Reihe an Zufallszahlen (x,y Koordinaten) mit Hilfe der rand() Funktion der C++ Standard-Library generiert. Diese Zahlen werden als Vektoren gespeichert. Zwei weitere Testfälle sind implementiert, indem das Worstcase Szenario aufgezeigt werden soll (x-Koordinaten konstant bzw. y-Koordinaten konstant, somit jeder Punkt auf der konvexen Hülle). Für die Berechnung der Testzeiten der Algorithmen wurde eine eigene Klasse implementiert (QueryperformanceCounter). Die Laufzeiten der Testfälle werden in einem .txt File gespeichert (*results.txt*) gespeichert. Das Format ist wie folgt: **hh:mm:ss.ms**.

Des Weiteren besteht die Möglichkeit bei Programmausführung zwischen den gefragten Modi umzuschalten. Dies wird mittels Konsolenparameter ermöglicht. Es besteht die Möglichkeit zwischen dem Step-Modus (Veranschaulichung der Einzelschritte) und dem Test-Modus (sofortige Ausgabe der konvexen Hülle) umzuschalten. Weiters kann die Anzahl der Zufallszahlen eingestellt werden.

Ein .bat File befindet sich im Verzeichnis *Executable* und enthält beide Modi:

ConvexHull.exe −−stepbystep 1 −−points 40
pause
ConvexHull.exe −−stepbystep 0 −−points 2000000
pause

Step-Modus

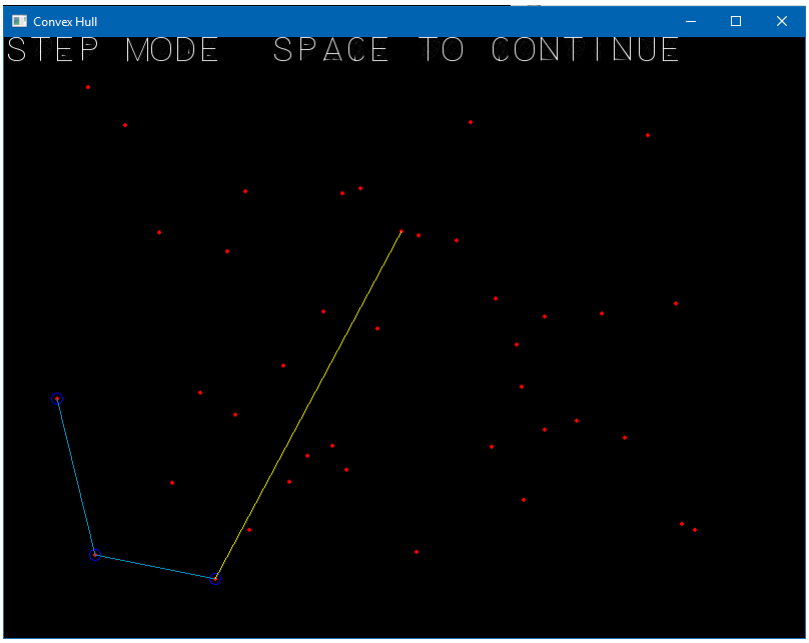

Abbildung 3: Gift-Wrapping Visualisierung mittels SFML. Blau eingekreist: Punkt auf konvexer Hülle; Blaue Linie: gefundene konvexe Hülle; Gelbe Linie: Vektor $\overline{p_0p_2} = \overline{pr}$

Test-Modus

Ausgehend von $2{\cdot}10^6$ Zufallszahlen und den drei Testszenarien ergibt sich folgende Laufzeit für den implementierten Gift-Wrapping Algorithmus um die konvexe Hülle zu finden:

Algorithmus	Laufzeit [s] für $n = 2 \cdot 10^6$
Random x und y	0.044
Random x und y konstant	0.454
Random y und x konstant	0.436

Tabelle 1: Ergebnis der Testszenarien

In Tab. 1 zeigt sich deutlich die Schwäche des Algorithmus. Bei normalverteilten Werten liefert das Gift-Wrapping sehr schnell eine exakte Lösung der konvexen Hülle. Liegen jedoch alle Werte der Eingangsdaten auf der konvexen Hülle führt dies zu einem drastischen Anstieg der Laufzeit.

Literatur

Thomas H. Cormen, Charles E. Leiserson, R. L. R. C. S. (2009). *Introduction to Algorithms* . MIT, 3rd edition edition.