Testprotokoll

Die Visualisierung des Programms wurde mit OpenCV gemacht. Diese Library ist notwendig um von die Source Files zu bilden. Eine Anleitung je nach OS ist zu finden unter: https://docs.opencv.org/4.x/d0/d3d/tutorial_general_install.html.
Eine Makefile ist vorhanden, jedoch wurde dies nur von 1 der 3 Gruppenmitglieder getestet.
Die Executable des x64 Release Builds ist ebenfalls beiliegend.

Usage

In der Konsole ConvexHull.exe aufrufen und über das CLI-Menu spezifizieren ob Visualisierung mit RNG punkten dargestellt werden soll oder ein Performance Test für eine Datei durchgeführt werden soll. Bei zweiterem muss ein Pfad zum Filename angegeben werden und anschließend noch bestätigt werden ob die Hüllenpunkte in der Konsole ausgegeben werden sollen. *Achtung:* DivideAndConquer funktioniert nur langsam!

z.B.:

```
./Testfiles/Circle/50000.txt
```

Testfiles

Die Testfiles wurden mit generate\.py gebildet. Es wurden je 5 Files für jede Kategorie mit N-Points in range(50k, 300k, 50k) generiert, welche die floats im Format x,y\n beinhalten.

Die Kategorien sind:

- Circle: Alle Punkte befinden sich auf einem Kreis
- Rectangle: Alle Punkte befinden sich eingegrenzt durch 4 Punkte
- Random: Alle Punkte sind uniform verteilt

Zudem wurde für den Giftwrapping Algorithmus zum testen der Optimierung eine Punktemenge gebildet mit n = 10^6 Punkte, die alle auf der Hülle sitzen + n/4 = 250k random verteilte Punkte innerhalb. Zu finden unter DenseLines.txt

Giftwrapping

Da der Algorithmus anfangs durch alle Punkte geht um den linkesten Punkt zu finden, ist der Aufwand am Anfang einmalig O(n).

Additiv dazu wird für jeden Punkt auf der Konvexen Hülle die Funktion *orient2d* aufgerufen um den rechtesten Punkt zwischen einer aktuellen Testgerade zu finden.

Eine kleine Optimierung wurde implementiert: Es werden die Punkte der aktuellen Testgerade, von der aus bestimmt wird ob ein Punkt C weiter Rechts liegt oder nicht, nicht mit sich selbst getestet, also n-2.

Zudem wurde noch eine, sich potentiell erheblich auswirkende Optimierung, implementiert: Wenn Punkt C auf der selben gerade liegt und weiter entfernt ist von A als B, dann wird dieser sofort übernommen, so wie wenn er weiter rechts wäre. Also je nach dem wieviele Punkte u auf den Geraden der Konvexen Hülle liegen, die selbst keine Eckpunkte sind ~-u * (n - 1) .

Insgesamt muss der Algorithmus für jeden Punkt der Konvexen Hülle k alle anderen n-2 (-2:Punkt auf der Hülle und der nächste im Array ausgenommen) Punkte prüfen.

D.h. der Allgemeine aufwand Beträgt etwa
O(n + k * (n - 2) - u * (n - 1)) = O((k-u) * n)= O(k * n) für kleine u.

Die Testfiles wurden alle für diesen Algorithmus getestet:

Form\N	50k	100k	150k	200k	250k
Circle	270.776ms	626.794ms	1027.21ms	1509.47ms	1974.87ms
Rectangle	0.9213ms	2.0919ms	2.8214ms	3.7676ms	5.0053ms
Random	1.8898ms	4.1657ms	5.7662ms	7.6738ms	10.4857ms

Wie erwartet performt Rectangle am besten, während Circle exponential wächst.

Zudem wurde noch das File DenseLines.txt getestet, welches auf dem Rechteck [0, 10^6] x [0, 10^6] an den Kanten je 10^6 Punkte hat und innerhalb zusätzlich uniform verteilte 250 * 10^3 Punkte:

Die Laufzeit betrug t = 25.3869ms. Bei der nicht optimierten Version war dies nicht in einem sinnvollen Zeitrahmen zu testen, da hierbei ~O(n^2) bei 4,25 * 10^6 Punkten gilt.

Divide And Conquer

Wurde unter den gleichnamigen Cpp Files implementiert, jedoch nicht auf Performance getestet, da dieser Teil lange buggy war und bei höheren Punktezahlen >1000 nur mehr sehr langsam durchläuft.

Eine allgemeine Beschreibung aus der Präsentations-Einheit ist beiliegend in CH-DAC.html.