Softwareprojekt Algorithmen der Computervision

Waggledance Detection

Roman Schulte-Sasse, Simon Auch, Johannes Sauer

12. Oktober 2015

1 Hardwaresetup

Bevor wir überhaupt mit dem algorithmischen Teil des Projekts, der Verbesserung bei der Erkennung der Schwänzeltänze, anfangen konnten, war es zunächst notwendig die passende Hardware zu beschaffen. Die verwendete Kamera stammte aus dem PlayStation Eye System und wurde schon vor unserem Projektstart ausgewählt um die Tänze zu detektieren, da sie eine außergewöhnliche hohe Aufzeichnungsrate von 100 Hz bei gleichzeitig niedrigem Preis bietet.
Leider ist das Objektiv dieser Kamera nicht auf die Maße der Aufbauten um den Bienenstock ausgerichtet. Bei dem Setup zur Zeit unseres Projekts war geplant die Kamera ca. 70 bis 90 cm von dem Stock entfernt zu installieren. Für diese Objektweite wurde ein Objektiv mit anderer Brennweite notwendig. Da bei der PS Eye Kamera kein Austausch von Objektiven vorgesehen ist, musste der Aufbau über dem Sensor mit dem ursprünglichen Objektiv zusammen entfernt werden und durch einen Neuen ersetzt werden.

1.1 Kameramount

Da die PS Eye Kamera allgemein bei Bastlern beliebt ist, waren schon Mehrere vor uns auf die Idee gekommen das Kameragehäuse auszutauschen. Deswegen sind CAD-Modelle zum 3D-Drucken für solche Gehäuse öffentlich verfügbar. Bei unserem Projekt verwendeten wir ein Gehäuse für M12-Objektive. Wir konnten das Gehäuse mit dem 3D-Drucker der AG Secure Identity ausdrucken, der Druck selbst war problemlos. Allerdings stellten wir später bei der Verwendung der modifizierten Kamera fest, dass das Schmelzschichtverfahren bei dem Druck etwas an Genauigkeit zu wünschen übrig liess. Das Objektiv hatte deshalb in dem Gewinde etwas zu viel Spiel und musste ab und zu nachjustiert werden, wenn das Licht nicht richtig auf den Sensor fokussiert wurde. Alles in allem war die Qualität des Gehäuses für unsere Zwecke jedoch ausreichend.

1.2 Objektive

Die Wahl unserer Objektive war zum einen durch den schon beschriebenen Abstand zum Bienenstock und den M12-Mount beschränkt. Da der Stock Ausmaße von ca. 30 cm × 40 cm hatte, ergab sich eine Brennweite von ca. 6 bis 7 mm. Außerdem sollte das Objektiv das Bild möglichst wenig verzerren und keinen IR-Filter besitzen. Da die Kombination dieser Eigenschaften etwas speziell ist hatten wir einige Zeit benötigt um eine gute Auswahl zu finden. Letztendlich wurden wir von dem Versandhändler Lensation gut beraten und bekamen sogar mehrere Objektive zum Testen ausgeliehen. Eins davon passte gut zu den Anforderungen und wurde deswegen gekauft.

2 Dotdetector

Nachdem wir uns das erste mal in den Code und die Arbeit eingearbeitet und profiled haben wurde uns schnell klar, dass:

a) Der Dotdetector einen Fehler hat

b) eine Formel genutzt wird welche sehr unintuitiv ist

c) 90% der Ausführungszeit auf den Dotdetector fällt

d) Die restlichen Layers das tun was Sie sollen

Aufgrund dieser zwei Punkte haben wir uns entschlossen uns hauptsächlich auf den Dotdetector zu konzentrieren.

2.1 Fehler der Frequenzanalyse-Berechnung

Die Berechnung der Frequenzanalyse geschieht mit folgender Formel:

$$score(\bar{B}_j^n, r) = \sum_{1 \le m \le b} (\bar{B}_j^n(m) * sin_r^m)^2 + (\bar{B}_j^n(m) * cos_r^m)^2 \tag{1}$$

$$score(\bar{B}_j^n, r) = \sum_{1 \le m \le b} (\bar{B}_j^n(m))^2 * (sin_r^m)^2 + (\bar{B}_j^n(m))^2 * (cos_r^m)^2 \tag{2}$$

$$score(\bar{B}_j^n, r) = \sum_{1 \le m \le b} (\bar{B}_j^n(m))^2 * ((sin_r^m)^2 + (cos_r^m)^2) \tag{3}$$

$$score(\bar{B}_j^n, r) = \sum_{1 \le m \le b} (\bar{B}_j^n(m))^2 \tag{4}$$

Das diese Formel keine Frequenzanalyse darstellt dürfte einleuchtend sein, weswegen wir die Formel folgendermaßen korrigieren:

$$score(\bar{B}_j^n, r) = \Big(\sum_{1 \le m \le b} \bar{B}_j^n(m) * sin_r^m \Big)^2 + \Big(\sum_{1 \le m \le b} \bar{B}_j^n(m) * cos_r^m \Big)^2 \tag{5}$$

Durch diese einfache Änderung stellten wir bereits eine deutlich reduzierte Erkennungsrate von Pixel fest, welche in dem betrachteten Zeitraum ***nur*** eine große Amplitude hatten (z.B. dunkle Biene auf hellem Holz).

2.2 Neues Feature zur Bewertung der Dots

Da die Formel zur Bewertung eines Pixel mit all seinen berechneten Frequenzen sehr unintuitiv ist, dachten wir uns vielleicht das man hier eine einfachere und womöglich auch bessere Formel finden können sollte.
Die alte Formel:

$$potential(D_j) = A_j^n * \sum_{r \in FREQS} w(score(\bar{B}_j^n, r)) * score(\bar{B}_j^n, r) \tag{6}$$

Wobei die Funktion w angibt, wieviele Scores kleiner sind, d.h. höhere Scores werden stärker, niedrigere schwächer bewertet. Da in keiner Stelle der Arbeit darauf eingegangen wird, wieso diese Formel an dieser Stelle ihren zweck erfüllt, haben wir uns etwas einfacheres ausgedacht.
Zunächst haben wir uns entschieden, die Amplitude aus dieser Rechnung zu entfernen. Denn sie wird bereits genutzt, um zu entscheiden ob der Pixel überhaupt als Dot in Betracht gezogen wird.
Weiterhin ist die Multiplikation des Scores mit seiner Position in der geordneten Liste der Scores seltsam unintuitiv und führt weiterhin zu einem einzustellenden Hyperparameter x, der nur mit sehr viel Code-Kenntnis gesetzt werden kann. Wir entschlossen uns, diese Berechnung durch einen simplen Grenzwert x zu ersetzen und die Funktion w ersatzlos zu entfernen.

$$\exists r \in FREQS : x \leq score(\tilde{B}_j^n, r) \tag{7}$$

Da diese Funktion schon sehr vielversprechend arbeitete, jedoch nicht immer alle Waggles erkannt hat, haben wir uns entschieden über jeweils zwei benachbarte Frequenzen die entsprechenden Scores zu summieren und für diese testen ob Sie größer x sind. Diese Änderung hatte den Effekt, das nun auch die Bienen erkannt werden welche genau zwischen zwei Frequenzen tanzen.

2.3 Probleme des neuen Features und der Frequenanalyse

Das einzige gravierende Problem welches wir mit dem neuen Feature und der korrigierten Frequenanalyse festgestellt haben ist, dass bei ungünstiger Beleuchtung die Bienen beim sogenannten *Ventilieren* erkannt werden, da der Wechsel von reflektiertem Licht und dunkler Wabe auch in den betrachten Frequenzbereich fällt (bzw. ein vielfachens davon). Ventilieren bezeichnet einen intensiven Flügelschlag der Biene innerhalb des Stocks, welcher zwar eigentlich deutlich schneller als die Frequenzen der Schwänzeltänze ist, aber teilweise einn Vielfaches dieser Schwingungen erreicht.
Die einfachste Möglichkeit dieses Problem zu lösen, welche sehr zuverlässig funktionierte, war die Beleuchtung zu ändern. Als Ideal hat sich diffuse Beleuchtung (z.B. Raumbeleuchtung mit Leuchtröhren) herausgestellt.
Jedoch glauben wir auch dass es ansonsten keine triviale Möglichkeit gibt, dieses Ventilieren und das Tanzen auf dieser Ebene zu unterscheiden, da beides auf den gleichen Frequenzraum fällt (bzw. vielfache davon). Eine mögliche Lösung in der Nachbearbeitung wäre womöglich die euklidischen Abstände erkannter Waggles im Stock zu messen, da sich die Bienen während des Ventilierens nicht bewegen und die erkannten Waggles sich entprechend immer an genau der gleichen Stelle befinden.
Um an dieser Stelle anzusetzen existiert bereits ein Python Prototyp, siehe Python Prototyp.

2.4 Parallelisierung der Dotdetection

Während der Laufzeitanalyse des bestehenden Codes ist uns aufgefallen, dass die große Verarbeitungsdauer des Dotdetectorlayers schon bei kleineren Abweichung von den üblichen Parametern (z.B. bei einer höheren Auflösung) zu Problemen führen kann, insbesondere weil dadurch die engen Echtzeitanforderungen an das System nicht eingehalten werden können. Denn um die hohe Framerate von 100 hz zu bedienen, hat das gesamte System nur zehn ms zur Verarbeitung eines Frames zur Verfügung. Da wir nicht einmal eine GPU zur Verfügung hatten, fiel auch die sonst wahrscheinlich naheliegenste Möglichkeit eines hoch-parallelisierten Waggle-Detectors weg.
Um dieses Problem anzugehen, versuchten wir zunächst kleinere, lokale Änderungen einzuführen um zu sehen ob sich das Problem ohne aufwendiges Refactoring beheben ließ.
Wie oben bereits angedeutet, war uns nach mehrern Profilingdurchgängen aufgefallen, dass vor allem die Dotdetection zeitraubend ist.
Da eine Schleife durch alle Pixel des Frames durchiteriert, dachten wir hier an eine einfache Möglichkeit zur Parallelisierung. Hierzu verwendeten wir eine `parallel_for` Schleife; die kritischen Programmteile innerhalb der Schleife wurden durch Locks geschützt. Diese Verbesserung war zwar sehr schnell durchgeführt und änderte nichts an der Funktionsweise des Programms, leider war der Geschwindigkeitszugewinn jedoch zu klein um das Problem zu lösen.

2.5 Dotdetector mit Matrizen

Der hohe Zeitverbrauch liegt vor Allem an der sehr aufwändigen Projektion jedes einzelnen Pixels auf die Sinus bzw. Cosinus-Funktion. Der bisherige Ansatz zur Lösung dieses Problems bestand aus einem Objekt pro Pixel, in welches die Helligkeitswerte, die dieser Pixel besaß, kopiert wurden.
Für einen neuen Frame wurden also zunächst die Werte aktualisiert und anschließend wurde für jeden einzelnen Pixel die Projektion vorgenommen und entschieden, ob ein Schwellwert überschritten wurde. Da

die Bienen meist in nur 3-4 Frequenzen tanzen, muss dementsprechend jeder Pixel achtmal projeziert (Sinus und Cosinus) und anschließend evaluiert werden.
Um diesen Prozess effizienter zu gestalten, wollten wir diesen Ansatz ändern. Anstatt eines Dot-Detectors pro Pixel sollte die gesamten Bildinformationen über die Zeit in einer Matrix gespeichert werden. Die Projektion der gesamten Pixel auf einen Sinus oder Cosinus entspricht damit einer Matrix-Multiplikation. Die Projektion der einzelnen Bildpunkte auf Cosinus/Sinus lässt sich also folgendermaßen formulieren:

$$\mathcal{D} \cdot F_{sin} = p_{sin} \tag{8}$$
$$\mathcal{D} \cdot F_{cos} = p_{cos} \tag{9}$$
$$\tag{10}$$

wobei

$$\mathcal{D} \in (resX \cdot resY) \times bufferSize$$
$$F_{sin/cos} \in bufferSize \times freqNumber$$

Dabei steht *resX* die Auflösung des Bildes in horizontaler Richtung, *resY* die Auflösung in vertikaler Richtung und *bufferSize* die Anzahl der im Ringbuffer enthaltenen Bilder. *freqNumber* bezeichnet die Anzahl der Frequenzen, auf die projeziert werden soll.
Für alle Frequenzen sind damit nur noch zwei Matrix-Multiplikationen nötig. Dies bietet zwar keine Änderung in der Anzahl der Berechnungen, allerdings sind die modernen Bibliotheken für Lineare Algebra sehr stark optimiert, so dass die Berechnung mittels Matrizen deutlich schneller geht, als eine selbst implementierte Variante.
Je nach Prozessor ist der Grad der Optimierung auch unterschiedlich, allerdings sind mittels Armadillo Beschleunigungen im 10-100 fachen erwartbar, da SSE Register und spezielle Makro-Befehle genutzt werden können.
Weiterhin bietet diese Form der Berechnung der Projektion eine Möglichkeit auch in der Zukunft weiter zu optimieren, beispielsweise durch Auslagerung der Rechnungen auf eine GPU.
In Performance-Tests war die Implementierung mit Matrix-Multiplikation ca. ein Drittel schneller, als die auf den Zweck hochoptimierte alte Variante des Dot-Detectors.
Zukünftig könnten auch die Sinus- bzw. Cosinus Matrizen einmal vorberechnet werden, anstatt wie momentan jeden Frame neu. Dadurch ließe sich ein weiterer Speedup der Berechnungen erreichen.

3 Winkelanalyse

Der dritte Layer der *Waggle Detection* beschäftigt sich mit der Extraktion von Winkel und Länge des Tanzes. In der bisherigen Implementierung wurde dafür der Winkel des letzten und ersten erkannten Punktes, welcher zum Tanz gehört, berechnet. Für die Länge wurde ebenso die euklidische Distanz dieser beiden Punkte zueinander berechnet. Mathematisch lässt sich die alte Winkelextraktion folgendermaßen aufschreiben:

$$\theta = atan2((last.y - first.y), (last.x - first.x)) \tag{11}$$
$$r = \sqrt{(last.x - first.x)^2 + (last.y - first.y)^2} \tag{12}$$

Dieser Ansatz ist zwar mathematisch korrekt, allerdings gegenüber Ausreißern in der Detektion der Pixel nicht robust. Weiterhin kann es durchaus sein, dass sich eine Biene immer noch tanzt, während sie bereits dabei ist, sich umzudrehen. In diesem Fall wäre zwar die Richtung des Großteils des Tanzes in einer Richtung, allerdings könnten diese letzten Punkte die extrahierte Richtung sehr stark beeinflussen.

Aus diesem Grund haben wir uns entschieden, auch hier Verbesserungen vorzunehmen, um die endgültige Extraktion der Positionen der Nahrungsquellen für die Bienen zu verbessern.

3.1 Fitline und Lineare Regression

Um die Winkel-Extraktion zu verbessern, haben wir zunächst einen klassischen Algorithmus aus der Mustererkennung verwendet. Die *lineare Regression* berechnet aus verschiedenen Datenpunkten eine Ausgleichsgerade (im $\mathbf{R}^2$). Diese ist zwar nicht sehr robust gegenüber Ausreißern, allerdings schon deutlich besser, als die einfache Extraktion via erstem und letztem Punkt. Bei der linearen Regression wird der quadratische Fehler einer Gerade durch die Datenpunkte minimiert, was zu einer Gerade führt, die den tatsächlichen Verlauf des Tanzes sehr viel besser approximiert, als der bisherige Algorithmus.
Da allerdings auch dieses Verfahren aus oben genannten Gründen nicht völlig zufriedenstellend war, haben wir mittels der Fitline-Funktion von OpenCV versucht, bessere Ergebnisse zu bekommen. Diese Funktion basiert auf einer Klasse von Schätzern, sog. *M-Estimators* und verwendet zur Approximation der Linie einen *RANSAC*-ähnlichen Algorithmus. Durch die Verwendung dieser Funktion ließ sich der Winkel deutlich besser extrahieren als in der Vorgänger-Funktion.

3.2 Darstellung in Videos

Um die Extraktoren gut und angewandt vergleichen zu können, war ein neuer Modus im Programm vonnöten. Durch eine neue Funktion ist es möglich, den Verlauf des Videos für eine bestimmte Zeit automatisch anzuhalten.
Während dieses *freezes* des Videos werden die berechneten Winkel der verschiedenen Extraktoren in Form von verschiedenfarbigen Geraden angezeigt. Dadurch wird eine vergleichsweise einfache Validierung der Winkel-Extraktion ermöglicht. Natürlich ist es möglich, diese Visualisierung auszuschalten, wenn das Programm ativ läuft.

4 Python Prototyp

Gegen Ende des Projektes haben wir noch an einem kleinen Python-Skript gearbeitet, um die gefundenen Waggles zu visualisieren und anhand von Bedingungen (z.B. der maximale/minimale, zeitliche und örtliche Abstand von einzelnen Waggles) zu clustern. Die Visualisierung erfolgt hierbei mithilfe von gnuplot. Zu beachten ist, dass das Skript zur Zeit noch *alle* Waggles die es im angegebenen Ordner finden kann, lädt. Dies kann bei vielen erkannten Tänzen entsprechend lange dauern.
Dieses Projekt ist vermutlich eine gute Stelle um Probleme wie die des neuen Features zur Dotdetection mit ungünstigen Lichverhältnissen in der Nachbearbeitung zu lösen.

4.1 Funktionsübersicht

a) eine Klasse Waggle, welche alle Informationen zu einzelnen Waggles speichert

b) laden aller gefundener Waggles des WDD

c) Clustern der gefunden Waggles anhand von bedingungen (siehe getAllClusters.py, Funktion match)

d) Ausgabe aller gefundenen Waggles/Cluster in eine .csv

e) Visualisierung von Waggles/Clustern

5 Korrektur des Sanitychecks

Während einiger Tests ist uns aufgefallen, dass man nicht auf der kompletten Auflösung von Videos arbeiten konnte da ein fehlerhafter Sanity-check auf eine Reduzierung auf mindestens 1/4 der originalen Bildgröße bestanden hat.