Hochschule Esslingen
University of Applied Sciences

Bachelor-Thesis

Prototypische Realisierung und Test optischer Flusserkennung auf Basis von der OpenCV-Bibliothek

Anton Tratkovski

Fakultät Informationstechnik

Softwaretechnik und Medieninformatik

Sommersemester 2010

Zeitraum:	**1.04.2010 – 7.08.2010**
Prüfer:	**Prof. Dr. Jörg Friedrich**
Zweitprüfer:	**Prof. Dr. Andreas Rößler**
Betreuer:	**B. Eng. Dionysios Satikidis**

Kurzbeschreibung

Diese Bachelorarbeit beschäftigt sich mit dem Optischen Fluss und seiner Erfassung. Probleme des optischen Flusses werden aufgezeigt und erläutert. Methoden seiner Erfassung und deren Unterschiede werden untersucht und auf ihre Tauglichkeit erprobt. Die Offene Bibliothek OpenCV kommt dabei zum Einsatz und bildet damit die Grundlage der eingesetzten Software. Im Vordergrund steht die Nachfrage nach einem System, was einen Flugapparat stabilisiert und Regelt. Hierbei handelt es sich um den Quadrocopter der Hochschule Esslingen. Zur Stabilisierung solcher Flugkörper kommen unterschiedlichste Systeme zum Einsatz, die alle ihre Vor- und Nachteile besitzen. Vor allem die Nachteile der herkömmlichen Mechanismen lassen Platz für neue Überlegungen und Ansätze, die den Alten zwar nicht die ganze Arbeit abnähmen werden, aber in bestimmten Situationen diese erleichtern und unterstützen können. Solche wie GPS-Systeme die ihre vorteile vor allem im Außenbereich, auf der Straße haben, im Gebäude allerdings nur bedingt einsetzbar sind. Oder mechanische Lösungen, wie Beschleunigungssensoren deren Genauigkeit, vor allem bei sehr langsamen Bewegungen nicht den erforderlichen Stand erreicht. Zur Stabilisierung unterschiedlichster Systeme kommen optische Sensoren schon lange zum Einsatz, so liegt die Überlegung nahe auch bei dieser Aufgabenstellung nach optischen Lösungsmöglichkeiten zu suchen. Die nachfolgende Arbeit verfolgt die Methode der Bodenabtastung, die ähnlich wie bei einer optischen Computermaus, den Boden mit einer Kamera aufnimmt und anhand der Unterschiede zwischen den einzelnen Bildern die Bewegung schätzt. Die Ergebnisse dieser Arbeit sollen Aufschluss darüber geben welche Methoden und Algorithmen für so eine Aufgabe in Frage kommen, wie diese funktionieren, gesteuert und optimal ausgenutzt werden.

Inhaltsverzeichniss

Abbildungsverzeichnis

Formelnverzeichnis

1. Einleitung

1.1. Motivation

Im aktuellen Zeitalter, in dem nicht nur die Rohstoffe knapp sind, sondern zum Teil von der stets wachsenden Weltbevölkerung verursacht, auch der Platz. Besitzen immer mehr Menschen immer mehr Fortbewegungsmittel, es entstehen immer mehr Unfälle und Staus. Schon lange versuchen Hobbybastler und Ingenieure den Traum vom fliegenden Auto zu realisieren. Natürlich gib es längst Prototypen, die uns bequem vom Punkt A zum Punkt B durch die Luft befördern können. Doch ist dieser Industriezweig noch lange nicht auf dem Stand um Serien zu Produzieren und sie dem Verbraucher anzubieten. Um ein Flugapparat zu steuern bedarf es einer aufwendigen Ausbildung, und das bereits jetzt, wo es noch nahezu keinen Privaten Flugverkehr gibt. Natürlich zum größten Teil aus dem Grund, dass es nicht einfach ist im Flugverkehr teilzunehmen, es gibt viele Faktoren die man beachten muss und es gibt in der Luft keine Verkehrsschilder die einem sagen könnten wie man sich in der aktuellen Situation zu verhalten hat. Bevor das Fliegen so einfach wird wie das Autofahren müssen noch viele Sachen bedacht und entwickelt werden. Was aber schon jetzt klar ist, ist dass das Fortbewegungsmittel der Zukunft, vor allem ein Flugapparat sehr viele Aufgaben dem menschlichen Piloten abnehmen muss. Es ist bekannt, dass der Mensch ein Lauftier ist und sich an der zweidimensionalen Erdoberfläche am besten orientieren kann. So werden in der Zukunft die fliegenden Fortbewegungsmittel dem Menschen, aus dem dreidimensionalen Lüftraum eine virtuelle zweidimensionale Bewegungsfreiheit modellieren müssen, um uns das Zurechtfinden einfacher zu machen. Oder sie werden gar die gesamte Steuerung der Fortbewegung übernähmen, gesteuert von einem Zentralsystem, oder autonom sich an der Umwelt orientierend. In jedem Fall muss die Bewegung in der Lüft erst zu einer Selbstverständlichkeit werden, die nur wenige Signale benötigt um sicher, auch komplizierte Bewegungsmuster auszuführen, ohne dass Umweltfaktoren diese Bewegung beeinflussen können. Es ist ganz klar, dass diese Selbstverständlichkeit nur von einem komplizierten, sicherem System ermöglicht werden kann, das sich auf unterschiedlichste Sensoren und Mechanismen verlassen muss. Diese Sensorsysteme müssen sowohl im kombinierten Arbeitsablauf funktionieren können und stets richtig entscheiden können welcher der Sensoren die Realität genauer widerspiegelt. Doch auch bei möglichen Ausfällen müssen diese Sensoren genau genug sein um den ausgefallenen Partnermechanismus in vollem Masse zu ersetzen, um gegebenenfalls die Nächste Werkstatt zu erreichen. Es können GPS-Signale sein die dem Fahrzeug stets seine Position mitteilen, oder Beschleunigungssensoren, interessant sind auch optische Systeme zur Erfassung der Bewegung.

Die optische Variante des Bewegungssensors kann sehr unterschiedlich gestaltet werden. Man kann sich auf Markierungen verlassen, oder die Bewegung rein auf der Basis der gegebenen Umwelt schätzen. Dabei muss die Erfassung nicht unbedingt vom Apparat selber erfolgen, es kann auch von Außen, optisch erfasst und gesteuert werden, was sicherlich viel aufwändiger und kostspieliger wäre.

1.2. Aufbau der Arbeit

Diese Bachelorarbeit besteht aus 5 Kapiteln und ist folgendermaßen aufgebaut:

Kapitel 1

Enthält neben der Beschreibung des Aufbaus der Arbeit, die Motivation die den Leser auf die Thematik einstimmen soll.

Kapitel 2

Zeigt die Grundlagen der Flusserkennung, die das Verstehen der Problematik erleichtern sollen. Außerdem behandelt dieses Kapitel die Probleme, die bei der optischen Flusserkennung aufträten können. Und erläutert die Methoden, die dafür eingesetzt werden.

Kapitel 3

Dieses Kapitel beschreibt die Ausgangssituation, das Programm, was eingesetzt wurde um ein besseres Verständnis der OpenCV-Bibliothek zu kriegen. Des Weiteren Beschreibt dieses Kapitel das Programm, was für den Test der einzelnen OpenCV-Funktionen entwickelt wurde, und die getesteten Funktionen.

Kapitel 4

Im Kapitel 4 geht es um den Test der Methoden zur Erfassung des optischen Flusses, es wird die Testumgebung, sowie die Probleme die beim Testen aufgetreten sind beschrieben. Des Weiteren beschreibt dieses Kapitel die Testdurchführung und erläutert die dabei entstandenen Ergebnisse.

Kapitel 5

In diesem Kapitel werden die Ergebnisse und Eindrücke der Arbeit zusammengefasst und ein kurzer Ausblick gegeben.

2. Grundlagen zum optischen Fluss

2.1. Optischer Fluss

Als Definition für den optischen Fluss liefert die freie Onlineenzyklopädie, Wikipedia folgende Beschreibung:

Als Optischer Fluss (eng. Optical Flow) wird in der Bildverarbeitung und in der optischen Messtechnik ein Vektorfeld bezeichnet, das die Bewegungsrichtung und -Geschwindigkeit für jeden Bildpunkt einer Bildsequenz angibt. Der Optische Fluss kann als die auf die Bildebene projizierten Geschwindigkeitsvektoren von sichtbaren Objekten verstanden werden.

Optischer Fluss ist ein Phänomen das uns stets in alltäglichen Leben begleitet. Es ist nichts Anderes als die Bewegung unserer Umwelt relativ zu unseren Augen die wir wahrnehmen, wenn wir beispielsweise durch die Stadt laufen oder mit dem Auto unterwegs sind. Blicken wir aus dem Fenster eines Fahrenden Zuges, so sehen wir verschiedenste Landschaften, ob Gebäude in der Stadt oder Bäume und Felder auf dem Land egal ob überfüllt von Menschenmassen oder Tieren oder scheinbar leer und verlassen, alle diese Objekte rasen in einer scheinbar fast homogenen Masse an uns vorbei. Diese Bewegung ist der besagte optische Fluss. Zu den Funktionen des optischen Flusses gehört nicht nur das Erfassen von Bewegungsrichtungen, sondern auch das Erkennen von Entfernungen. So bewegen sich zum Beispiel große, weit entfernte Objekte wie Wolken oder Hochhäuser in Relation zu näheren und kleineren Objekten wie Bäumen nur sehr langsam.

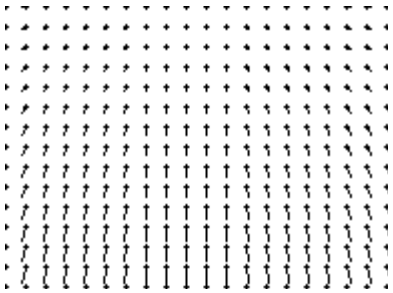


Abbildung 2-1: Vektorfeld für die Vorwertsbewegung

Es gibt eindeutige, mathematische Beziehungen Zwischen der Entfernung zum betrachteten Objekt und dem optischen Fluss. Fahren Sie die doppelte Geschwindigkeit so verdoppelt sich auch die Geschwindigkeit des Flusses. Platziert man jetzt das Objekt in einer doppelten Entfernung, so halbiert sich die sichtbare Geschwindigkeit wieder. Natürlich gibt es auch einen Zusammenhang zwischen dem optischen Fluss und dem Betrachtungswinkel relativ zur Bewegungsrichtung. Der Optische Fluss ist am größten, wenn das betrachtete Objekt sich auf Ihrer Höhe befindet, seitlich, ober- oder unterhalb von Ihnen, also Blickrichtung 90° zur Bewegungsrichtung. Vor oder hinter der 90° Marke nimmt der erkennbare optische Fluss ab.

Platziert man das betrachtete Objekt aber direkt in der Bewegungsrichtung, also vor dem Betrachter, so wird man keinen optischen Fluss erkennen, so ist direkt auf der Bewegungslinie der optische Fluss gleich Null. Allerdings werden die Kanten eines realen Objekts nie direkt auf der Linie der Bewegung liegen und haben so mit einem minimalen optischen Fluss, den wir als das Größerwerden des Objekts wahrnehmen können.

Die Abbildung 2-2 veranschaulicht die Verteilung des optischen Flusses bei unterschiedlicher Bewegung. Bei einer gradlinigen Bewegung in eine bestimmte Richtung verteilt sich der optische Fluss wie oben bereist beschrieben. Direkt auf der Bewegungsrichtung ist kein Fluss zu erkennen. Auf der Höhe des Betrachters sehen wir den größten optischen Fluss mit entgegen gesetzter Flussrichtung zur Bewegung des Betrachters. Anders bei einer Drehbewegung, dort sieht man auf den ersten Blick einen konstanten optischen Fluss, auf der zweidimensionalen Abbildung erkennt man allerdings nur die Pole des optischen Flusses. Die Nullpunkte, die sich bei einer gradlinigen Bewegung auf der Bewegungsgeraden befinden, sind bei einer Drehung ober- und unterhalb der Drehfläche, was logisch erscheint, wenn man sich an die Rechtes-Daumen-Regel oder die Umfassungsregel aus dem Physikunterricht erinnert. Mit der Umfassungsregel wird aus der Drehbewegung wieder eine Gerade, die senkrecht zur Drehfläche steht, als Drehachse bezeichnet werden kann und wie bei gradliniger Bewegung die Nullpunkte des optischen Flusses markiert.

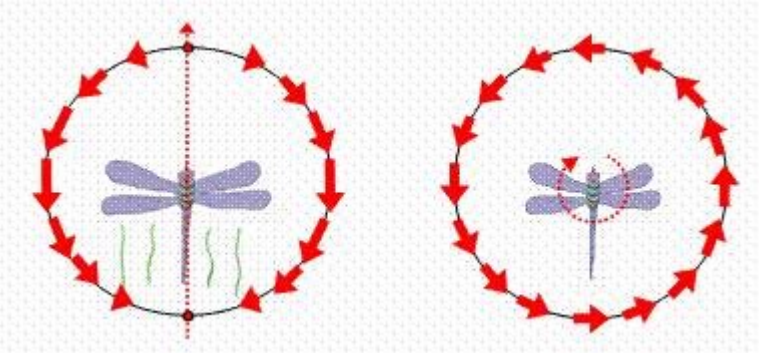


Abbildung 2-2: Charakteristische Vektorbilder für eine Vorwertsbewegung und eine Drehung

Die Eigenschaften der Drehbewegung werden klar wenn man die Abbildung 2-3 betrachtet. Es handelt sich dabei um eine Kugel im Raum, die in der Fläche um den Betrachter aufgerollt wurde. Die Pfeile repräsentieren hier mit ihrer Länge die Geschwindigkeit des optischen Flusses, wobei die Punkte oben und unten immer einen und denselben Punkt visualisieren, nämlich den oberen und unteren Punkt auf der Drehachse.

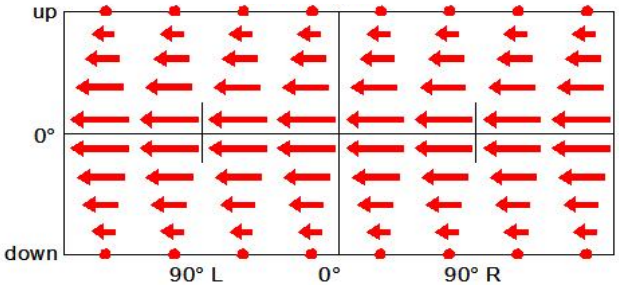


Abbildung 2-3: Aufgeklappte Vektorkugel für die Drehung nach rechts

Die Abbildung 2-4 zeigt eine solche Kugel, in einer gradlinigen Bewegung. Der Punkt in der Mitte markiert hier die Bewegungsrichtung mit den Vektoren des optischen Flusses, die von der Mitte ausgehen zur 90°-Marke immer größer werden, weiter hinten wieder an Länge verlieren und in der 180°-Marke in einem Punkt münden.

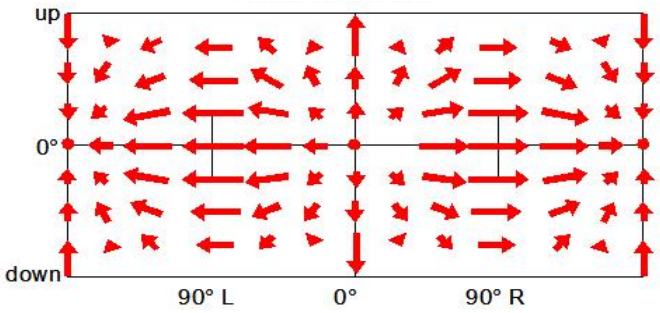


Abbildung 2-4: Aufgeklappte Vektorkugel für die Vorwertsbewegung

2.1.1. Bewegung zur Grauwertveränderung

In der Literatur zur Bildverarbeitung liest man oft den Begriff der Grauwertveränderung beziehungsweise, Grauwertverschiebung. Der Grauwert ist der Helligkeits- oder Intensitätswert eines Pixels, ohne die Berücksichtigung der Farben. In einem RGB-Bild könnte eine mögliche Bestimmung des Grauwerts wie in Formel 2-1 dargestellt aussehen.

$$\text{Grauwert} = 0{,}5 \cdot \text{Rot} + 0{,}3 \cdot \text{Grün} + 0{,}2 \cdot \text{Blau}$$

Formel 2-1: Bildung des Grauwerts

Anhand der Bewegung dieser Grauwerte lässt sich der optische Fluss berechnen. Was dabei aber fälschlicherweise angenommen werden kann, ist die Tatsache, dass die Grauwertveränderung mit der Bewegung in der Szene gleich zu setzen ist. Nun spielen in der Grauwertveränderung die Beleuchtung und die Objekte, eine gleich große Rolle wie die Bewegung selbst. Angenommen eine Kugel mit einer sehr glatten und gleichmäßigen Oberfläche dreht sich vor der Kamera, wären die Grauwerte für den Betrachter regungslos. Andererseits würde man eine solche Kugel von einer sich bewegenden Lichtquelle beleuchten

lassen, könnte man anhand des bewegten Schattens zu der falschen Erkenntnis kommen, dass die Kugel sich bewegt.

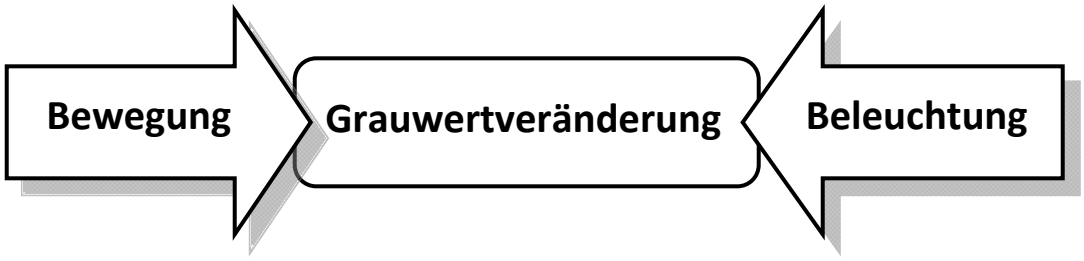


Abbildung 2-5: Zusammenstellung der Grauwertveränderung

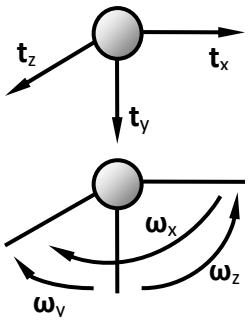
So kann man resultierend sagen, dass die Grauwertveränderung (siehe Abbildung 2-5) ein Ergebnis aus der Verbindung der Bewegung und der Beleuchtung ist, welches seine Uhrsprungskomponente nicht immer eindeutig widerspiegelt.

2.1.2. Perspektivisches Bewegungsmodell

Mit dem perspektivischen Bewegungsmodell lassen sich die Geschwindigkeiten der Szenenpunkte relativ zur Kamera, als Beobachter ziemlich anschaulich berechnen. Das Perspektivische Bewegungsmodell beruht auf der dreidimensionalen Starkörperbewegung mit sechs Freiheitsgraden. Zu einem sind es drei Bewegungsrichtungen der Translation und drei weitere Bewegungsrichtungen der Rotation Formel 2-2 (siehe Abbildung 2-6).

$$\vec{t} = \begin{bmatrix} t_x \\ t_y \\ t_z \end{bmatrix}; \ \vec{\omega} = \begin{bmatrix} \omega_x \\ \omega_y \\ \omega_z \end{bmatrix}$$

Formel 2-2



Translatorische Bewegungsvektoren

Rotatorische Bewegungsvektoren

Abbildung 2-6: Freiheitsgrade des Perspektivischen Bewegungsmodells

Ausgegangen von den sechs Freiheitsgraden und den Koordinaten des Szenepunktes $\vec{P}$, mit

$$\vec{P} = \begin{bmatrix} X \\ Y \\ Z \end{bmatrix}$$

Formel 2-3

läst sich für die Geschwindigkeit eines Punktes der Szene folgende Gleichung aufstellen:

$$\dot{\vec{P}} = -\vec{t} - (\vec{\omega} \times \vec{P}) = \begin{bmatrix} -t_x - (\omega_y \cdot Z) + (\omega_z \cdot Y) \\ -t_y - (\omega_z \cdot X) + (\omega_x \cdot Z) \\ -t_z - (\omega_x \cdot Y) + (\omega_y \cdot X) \end{bmatrix}$$

Formel 2-4

Die Koeffizienten des Vektors $\vec{P}$ sind negativ, da von der Bewegung der Kamera ausgegangen wird, und die Bewegungen der Objekte in der Szene dem genau entgegen wirken.

Um die Geschwindigkeit eines Punktes zu bestimmen, sind deren Koordinaten nach der Zeit abzuleiten. Dies kann auf einzelne Komponenten der Bewegungsrichtung explizit angewandt werden, wie in der Formel 2-5 für die X-Komponente angewandt wird, dabei ist f die Brennweite der Linse und Z der Abstand des Objekts. Abgeleitet nach der Zeit erhält man die Gleichung die in der Formel 2-6 dargestellt ist.

$$x = \frac{f}{Z} \cdot X$$

Formel 2-5

$$\frac{dx}{dt} = f \cdot \left(\frac{\partial x}{\partial Z} \cdot \frac{\partial Z}{\partial t} + \frac{\partial x}{\partial X} \cdot \frac{\partial X}{\partial t} \right) \quad \text{oder} \quad \dot{x} = f \cdot \left(\frac{X}{Z^2} \cdot \dot{Z} + \frac{1}{Z} \cdot \dot{X} \right)$$

Formel 2-6

Analog dazu kann man die gleiche Berechnung für die Y-Koordinate aufstellen (Formel 2-7).

$$\dot{y} = f \cdot \left(\frac{Y}{Z^2} \cdot \dot{Z} + \frac{1}{Z} \cdot \dot{Y} \right)$$

Formel 2-7

Durch Einsetzen der Formel 2-6 und Formel 2-7 in die Formel 2-4 , erhält man die Formel für das Geschwindigkeitsfeld $\dot{\vec{p}}$ (Formel 2-8).

$$\dot{\vec{p}} = \begin{bmatrix} \dot{x} \\ \dot{y} \end{bmatrix} = \frac{1}{Z} \cdot \begin{bmatrix} t_z \cdot x - f \cdot t_x \\ t_z \cdot y - f \cdot t_y \end{bmatrix} + \omega_x \cdot \begin{bmatrix} \frac{1}{f} \cdot x \cdot y \\ \frac{1}{f} \cdot y^2 + f \end{bmatrix} - \omega_y \cdot \begin{bmatrix} \frac{1}{f} \cdot x^2 + f \\ \frac{1}{f} \cdot x \cdot y \end{bmatrix} - \omega_z \cdot \begin{bmatrix} -y \\ x \end{bmatrix}$$

Formel 2-8

Diese Darstellung lässt sich durch Normierung weiter vereinfachen. Man setze die Brennweite f=1 und substituiert mit in Formel 2-9 und Formel 2-10 dargestellten Matrizen.

$$A = \begin{bmatrix} -1 & 0 & x \\ 0 & -1 & y \end{bmatrix}$$

Formel 2-9

$$B = \begin{bmatrix} x \cdot y & -x^2 - 1 & y \\ y^2 + 1 & -x \cdot y & -x \end{bmatrix}$$

Formel 2-10

so erhält man für das Perspektivische Bewegungsmodell in Formel 2-11 dargestellte Schreibweise.

$$\dot{p} = \begin{bmatrix} \dot{x} \\ \dot{y} \end{bmatrix} = A \cdot \frac{t}{Z} + B \cdot \omega = \begin{bmatrix} \frac{1}{Z} \cdot A & B \end{bmatrix} \cdot \begin{bmatrix} t \\ \omega \end{bmatrix}$$

Formel 2-11

Man merkt die Abhängigkeit der Geschwindigkeitsverteilung von der Entfernung Z des Szenenpunktes bis zur Linse der Kamera. Es bestätigt sich das Phänomen, dass weit entfernte Objekte sich langsamer zu bewegen scheinen, als näher gelegene. Dieser Zusammenhang Z(p) wird als Tiefenfunktion bezeichnet. [3]

2.2. Probleme der Flusserkennung

2.2.1. Das Blendenproblem (Aperturproblem)

Die Analyse des optischen Flusses beruht also auf räumlichen und zeitlichen Grauwertänderungen. Diese Grauwerte werden innerhalb bestimmter Radien, in zwei aufeinander folgenden Bildern wieder erkannt und einander zugeordnet. Nicht immer sind markante Merkmale dabei die sich sofort zuordnen lassen, doch auch wenn dies der Fall ist kann das Ergebnis mehrdeutig werden. Ein Solches Problem ist das Blendenproblem. Es taucht

dann auf wenn zwar ein Abschnitt des Suchfensters grauwertmäßig sich eindeutig vom Rest der Pixel unterscheidet, doch keine fassbaren Merkmale bietet an hand derer die Bewegung eindeutig erkannt werden kann, weil das Objekt zum Beispiel viel größer als das Suchfenster ist. (siehe Abbildung 2-7)

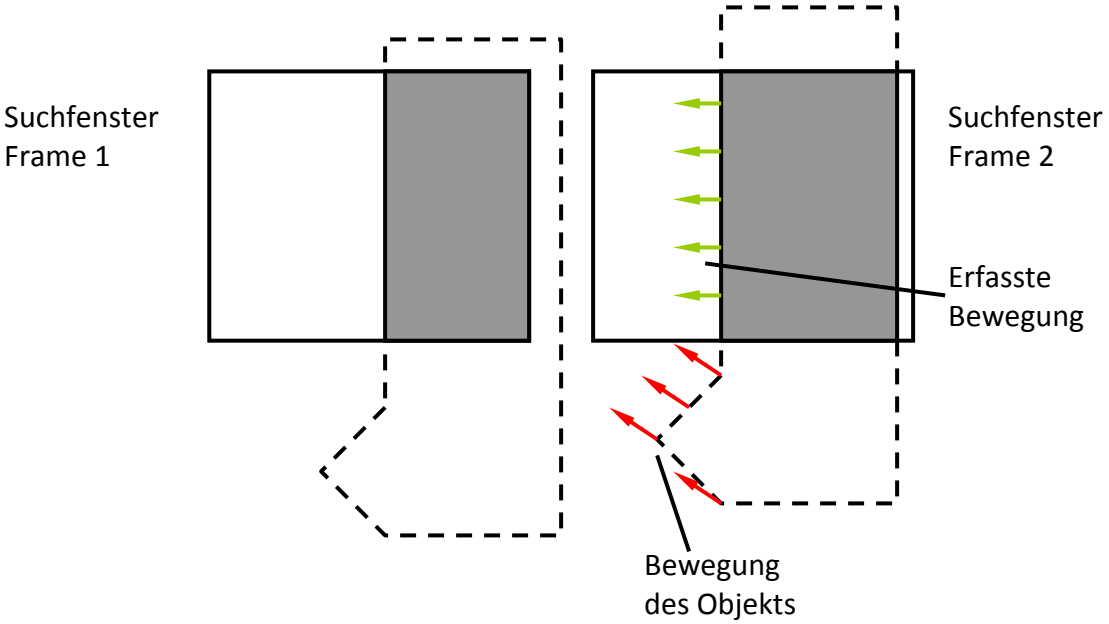


Abbildung 2-7: Veranschaulichung des Blendenproblems

Wir können lediglich die senkrecht zur Kante liegende Komponente des Verschiebungsvektors bestimmen, während die parallel zur Kante liegende unbekannt bleibt. Diese Mehrdeutigkeit kann z.B. aufgelöst werden, wenn die Operatormaske[1] die Ecke eines Objektes einschließt.

2.2.2. Korrespondenzproblem

Das Korrespondenzproblem beschreibt alle arten des Informationsverlustes einer Bewegung, das zuvor erwähnte Blendenproblem ist ein Spezialfall des Korrespondenzproblems. Das Korrespondenzproblem liegt dann vor, wenn wir keine eindeutig mit einander korrelierender Punkte, in zwei aufeinander folgenden Bildern einer Sequenz bestimmen können. In folgendem sind einige Beispiele des Korrespondenzproblems aufgeführt.

Die Abbildung 2-8 sieht man ein Beispiel für das Fehlen von eindeutigen Merkmalen an den der optische Fluss gemessen werden kann, es liegt dabei an der runden Form des Objekts bei der die rotatorische Komponente der Bewegung verloren geht.

[1] Eine Operatormaske oder das Suchfenster ist ein Block, in dessen Bereich der optische Fluss erfasst wird.

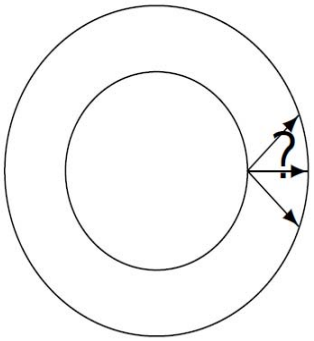


Abbildung 2-8: Korrespondenzproblem 1

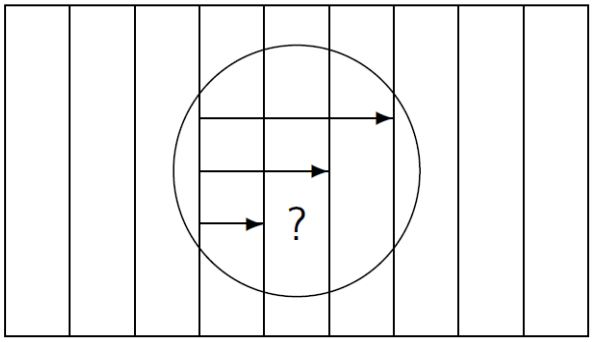


Abbildung 2-9: Korrespondenzproblem 2

Bei Periodischen Texturen, wie Netz oder Gitter(siehe Abbildung 2-9), kann die Bewegung nie eindeutig bestimmt werden, solange man nur ein abschnitt des Objekts betrachtet und die Bewegung ein Vielfaches der Maschenweite ist. Man erkennt die Bewegung erst dann eindeutig, wenn man den Rand des Gitters im Suchfenster hat.

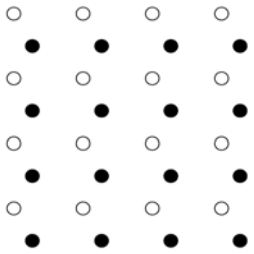


Abbildung 2-10: Korrespondenzproblem 3

Bei Bildern mit vielen Objekten (siehe Abbildung 2-10), die eine sehr ähnliche Form haben, kann das korrespondierende Teilchen im nächsten Bild nicht ermittelt werden.

Eine Lösung für Solche Probleme wäre das Verkürzen des Zeitinterwals, man würde einen solchen Interwal einstellen, bei dem man sich sicher sein kann, dass in der Zeit ein Teilchen nur einen kleinen Teil des mittleren Teilchenabstandes überbrücken kann (Abbildung 2-11). [7]

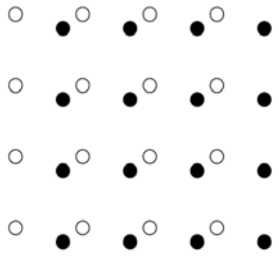


Abbildung 2-11: Lösung durch Verkürzen des Zeitinterwals

2.3. Erkennen des optischen Flusses

Die Erfassung des Optischen Flusses ist eine Disziplin der künstlichen Intelligenz. Eine Videokamera liefert ca. 30 Bilder in der Sekunde, der Unterschied zwischen den einzelnen Frames stellt hierbei die wichtigste Informationsquelle dar. Bewegt man die Kamera relativ zur Umgebung, so erkennt man am Bildschirm eine dementsprechende Bewegung der Umwelt, den zuvor erwähnten optischen Fluss. Den optischen Fluss stellt man oft als einen Vektorfeld dar, dabei beschreibt Länge und die Richtung jedes Vektors $\vec{V}(x_i, y_i)$ die derzeitige Bewegung, des Pixels $P(x_i, y_i)$. In der Abbildung 2-12 sieht man einige mögliche Vektorfelder und die dazugehörigen Transformationen wie Zoom, Drehung und die Kombination as den beiden.

Eine Bildrate von 30 Bilder/Sekunde entspricht einem Zeitintervall von $\frac{1}{30}$ Sekunde, zwischen zwei aufeinander folgenden Bildern. Es wird stets versucht die Intervalle so klein wie Möglich zu halten, damit die Bewegung der Bildinhalte nur wenige Pixel ausmacht[1]. Andererseits muss die Zeit, die dem Algorithmus bleibt, um den Fluss der Grauwerte zu berechnen, groß genug sein, um beim nächsten Frame das Ergebnis vorzulegen und mit der nächsten Berechnung fortzufahren.

[1] Die meisten Algorithmen zur Bestimmung des optischen Flusses, basieren auf der Annahme, dass die einzelnen Bewegungen im Bild nie eine große Distanz zurücklegen, somit sucht man ein bestimmtes Muster nicht in der gesamten Bildebene, sondern nur in der unmittelbaren Umgebung des gesuchten Grauwerts. Das heißt, umso kleiner der Zeitliche Abstand zweier auf einander folgender Bilder, desto kleiner kann der Suchbereich eingestellt werden, ohne den Verlust der Wahrscheinlichkeit den Vektor richtig zu bestimmen.

Abbildung 2-12: Vektorfelder des Zooms, der Drehung und ihrer Kombination

2.4. Kalman-Filter

Um möglichst genaue Messergebnisse vorlegen zu können, bedarf es eines Algorithmus der ohne viel Rechenaufwand, das Rauschen eines Signals heraus filtert und mit Statistischen Methoden an den tatsächlichen Wert möglichst nahe herankommt. Für eine solche Aufgabe bietet der heute wohl am weitesten verbreitete Algorithmus zur Zustandsschätzung linearer und nichtlinearer Systeme an, der Kalman-Filter.

Der Kalman-Filter ist ein nach seinem Entdecker Rudolf E. Kálmán benannter Satz von mathematischen Gleichungen. Obgleich die Benennung des Filters nach Rudolf E. Kálmán erfolgte, wurden bereits zuvor nahezu identische Verfahren durch Thorvald N. Thiele und Peter Swerling veröffentlicht. Auch existierten zur selben Zeit bereits allgemeinere, nichtlineare Filter von Ruslan L. Stratonovich, die das Kalman-Filter und weitere lineare Filter als Spezialfälle enthalten. Ebenso erwähnenswert sind Vorarbeiten und gemeinsame Publikationen von Kálmán mit Richard S. Bucy, insbesondere für den Fall zeitkontinuierlicher dynamischer Systeme. Daher wird häufig die Bezeichnung Kalman-Bucy-Filter und gelegentlich auch Stratonovich-Kalman-Bucy-Filter in der Fachliteratur benutzt.

Der erste erfolgreiche Einsatz des Filters erfolgte in Echtzeitnavigations- und Leitsystemen, die im Rahmen des Apollo-Programms der NASA unter Federführung von Stanley F. Schmidt entwickelt wurden und in der Navigation und Steuerung der Raumkapsel verbaut wurden.

Mithilfe dieses Filters sind bei Vorliegen lediglich fehlerbehafteter Beobachtungen Rückschlüsse auf den exakten Zustand von vielen der Technologien, Wissenschaften und des Managements zugeordneten Systemen möglich. Die Kernaussage ist: „Ist-Wert des Systems muss gleich Soll-Wert zu einer bestimmten Zeit sein.“ Vereinfacht dient das Kalman-Filter zum Entfernen der von den Messgeräten verursachten Störungen. Dabei müssen sowohl die mathematische Struktur des zugrundeliegenden dynamischen Systems als auch die der Messverfälschungen bekannt sein.

Eine Besonderheit des 1960 von Kálmán vorgestellten Filters bildet seine spezielle mathematische Struktur, die den Einsatz in Echtzeitsystemen verschiedenster technischer Bereiche ermöglicht. Dazu zählen u.a. die Auswertung von Radarsignalen zur Positionsverfolgung von sich bewegenden Objekten (Tracking) aber auch der Einsatz in elektronischen Regelkreisen allgegenwärtiger Kommunikationssysteme wie etwa Radio und Computer.

Mittlerweile existiert eine große Bandbreite von Kalman-Filtern für die unterschiedlichsten Anwendungsgebiete.

Im Gegensatz zu den klassischen FIR- und IIR-Filtern der Signal- und Zeitreihenanalyse basiert der Kalman-Filter auf einer Zustandsraummodellierung, bei der explizit zwischen der Dynamik des Systemzustands und dem Prozess seiner Messung unterschieden wird.

Bevor man einen Kalman-Filter erfolgreich einsetzen kann, sind vom System einige notwendige Bedienungen zu erfühlen, diese wären:

das System ist linear,

das Störsignal ist ein weises Rauschen und

das Störsignal ist gaußscher Natur.

die Erste bedeutet, dass jeder Zustand des Systems zum Zeitpunkt k, ähnlich wie eine Markow-Kette[1] erster Ordnung, nur vom Zustand k-1 abhängt, also aus der Multiplikation einiger Matrizen mit dem Zustand k-1 ermittelt werden kann.

$$X_k = F_{k-1} X_{k-1} + B_{k-1} u_{k-1} + v_{k-1}$$

Formel 2-12

[1] Eine Markow-Kette ist ein Stochastischer Prozess der aussagen über die Zukunft mache lässt. Das Spezielle einer Markow-Kette ist die Eigenschaft, dass durch Kenntnis einer begrenzten Vorgeschichte ebenso gute Prognosen über die zukünftige Entwicklung möglich sind wie bei Kenntnis der gesamten Vorgeschichte des Prozesses. [13]

Die Anderen zwei sagen aus, dass das Rauschen in unserer Messung nicht mit der Zeit korrelieren darf, also Zeitunabhängig ist und mit Zwei Werten exakt beschrieben werden kann, nämlich mit einem Durchschnittswert und einer Kovarianz.

$$v_k \approx WN(\bar{x}, Q_k)$$

Formel 2-13

Kurz zusammengefasst, erhöhen wir die Genauigkeit der Messwerte nach der eigentlichen Messung, mit Hilfe der vorangegangenen Ergebnisse. Wir berechnen dafür den aktuellen Zustand des Systems unter Berücksichtigung der aktuellen Messwerte, und der Werte von vorherigen Messungen und der dazugehörigen Wahrscheinlichkeiten und Varianzen. Somit bietet der Kalman-Filter eine gute Möglichkeit entweder von mehreren Quellen, oder von einer Quelle zur unterschiedlichen Zeiten erfasste Werte, mit einander zu kombinieren, um zu einem statistisch genaueren Ergebnis zu kommen.

Bis Jetzt war die Rede von Messungen eines ruhenden Systems, d.h. die einzelnen Messwerte wären im Idealfall genau gleich, was ist aber wenn sich das Objekt dessen Zustand erfasst wird, zwischen der ersten und der zweiten Messung bewegt. Um diesen Effekt im griff zu bekommen müssen wir in der so genannten „prediction phase“ eine Vorhersage treffen. In der „prediction phase“ verwenden wir alles was wir über dass System wissen um ihren Zustand zur Zeit der nächsten Messung zu ermitteln. Nehmen wir an unser Quadrokopter bewegt sich mit einer konstanten Geschwindigkeit, so können wir die einzelnen Messwerte (Lage im Raum) nicht einfach auf einander beziehen, doch mit der Kenntnis des Systems und des Zustands in dem es sich zur Zeit *t* befindet, können wir bestimmen wie sich der Ort des Quadrokopters bis zur zweiten Messung zur Zeit *t + dt* ändert. So vergleichen wir die Messung zur Zeit *t + dt* nicht einfach mit dem alten Zustand, sondern mit dem alten Zustand projiziert in die Zeit *t +dt.* Dabei unterscheidet kalmansche Lehre drei grundlegende Bewegungsarten, die in den weiteren Kapiteln genauer beschrieben werden.

2.4.1. Dynamical motion

Dynamical motion oder die Dynamische Bewegung, ist eine Bewegungsart bei der wir die Annahme treffen, dass das System eine gleich bleibende und seit der letzten Messung nicht veränderte Bewegung ausführt. Es sei, zum Beispiel x, die zu letzt gemessene Position des Systems zum Zeitpunkt t und v die aktuelle Geschwindigkeit, dann resultiert daraus, dass zur Zeit $\mathrm{t} + \mathrm{dt}$ das System sich an der Position $\mathrm{x} + \mathrm{v} * \mathrm{dt}$ befinden wird und sich mit der Geschwindigkeit v weiter bewegt.

2.4.2. Controll motion

Controll motion, eine Kontrollierte Bewegung. Wie der Name schon vermuten läst, ist es eine Bewegungsart die wir vorwiegend bei Systemen anwenden, die von Außen gestört werden und

geregelt werden müssen. Das bedeuten anders ausgeduckt, wir kriegen eine Reaktion des Systems auf eine beliebige Bewegungsart die wir selber initiieren. Am Beispiel unseres Quadrokopters heißt es, gibt man zum Zeitpunkt t dem Quadrokopter, der sich an der Position x befindet, ein Befehl zum Beschleunigen, so erwarten wir den Quadrokopter nicht an der Position $x + v * dt$, am Zeitpunkt $t + dt$, sondern ein Stück weiter, wegen der Beschleunigung die wir vorgeben und dem entsprechend auch mit einer höheren Geschwindigkeit $v + dv$.

2.4.3. Random motion

Der Name spricht auch in deisem Fall für sich, die Random motion oder die zufällige Bewegung, schließt alle die Bewegungen mit ein, die wir nicht vorhersehen können. Sei es ein Windstoß oder ein Rottorbruch.

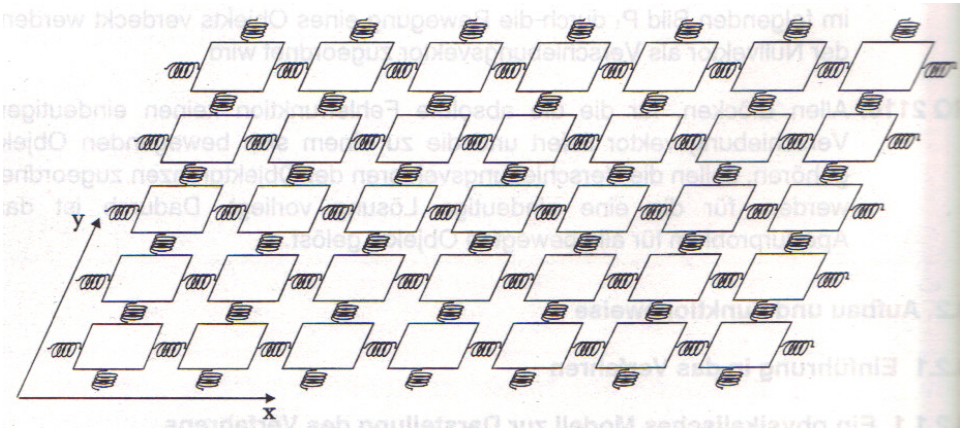
2.5. Korrelationsbasierte Verfahren

2.5.1. Block Matching

Die korrelationsbasierte Verfahren zur Erfassung des optischen Flusses, versuchen in zwei auf einander folgenden Bildern Bereiche zu finden, die einander eindeutig zuordenbar sind. Dabei vergleichen sie Muster mittels eines Ähnlichkeitsmaßes, aus hellen und dunkeln Pixels, in lokalen Bereichen der Bilder (Blöcke). Die zuvor definierten Blöcke müssen so gewählt werden, dass der Algorithmus bis zum nächsten Bildpaar das Ergebnis vorlegen kann, das heißt möglichst kleine Abschnitte mit wenigen Pixeln. Andererseits dürfen die Blöcke auch nicht zu klein gewählt werden, da sonst die Gefahr besteht dass das Suchmuster sich zum Zeitpunkt in dem das Zweite Bild gemacht wird schon außerhalb des Blocks befindet und nicht gefunden werden kann.

Da die verfolgten Objekte, an den der optische Fluss bestimmt werden soll, meist größer sind als die Blöcke, tritt beim Suchen der Ähnlichkeiten meist das Aperturproblem auf, da die Objekte nur an ihren Rändern eindeutige Bewegungen erkennen lassen. Dieses Problem wird dadurch gelöst, dass man einfach annimmt, dass die Blöcke die keine eindeutig zu erkennende Muster enthalten, zu einem Objekt mit daneben liegenden Blöcken zusammen gefasst werden können. Weiterhin wird angenommen, dass alle Blöcke die zu einem Objekt gehören gleiche Geschwindigkeiten besitzen, also in einem homogenen Vektorfeld liegen in dem alle Vektoren dieselbe Richtung und Länge haben.

Dazu existieren physikalische Verfahren zur besseren Vorstellung der Problematik. Dabei stellt man sich zwischen den benachbarten Blöcken gespannte Federn vor, deren Federkonstanten, jeweils davon abhängig sind wie ähnlich die Blöcke sich sind. Zwei ähnlich aussehende Blöcke haben demnach eine Verbindungsfeder mit einer großen Federkonstante und Blöcke die sich stark unterscheiden, oder zwischen denen sich eine Kante befindet, eine mit einer kleinen Federkonstante. [4]



Formel 2-14: Physikalisches Model

Das Ähnlichkeitsmaß ist in aller Regel, eine Korrelation zwischen zwei Folgebildern. Es gibt verschiedene Methoden um die Ähnlichkeit zweier Blöcke zu bestimmen.

SAD: Summe der Differenzen der Absolutbeträge („Sum of Absolut Differences“)

$$SAD = \sum_{r \in N} \left| I'(p' + r) - I(p + r) \right|$$

Formel 2-15: Sum of Absolut Differences

SSD: Summe der quadratischen Differenzen der Absolutbeträge („Sum of Squared Differences“)

$$SSD = \sum_{r \in N} \left(I'(p' + r) - I(p + r) \right)^2$$

Formel 2-16: Sum of Squared Differences

NCC: Normalisierte Kreuzkorrelation („Normalized Cross Correlation“)

$$NCC = \frac{\sum_{r \in N} I'(p' + r) \cdot I(p + r)}{\sqrt{\sum_{r \in N} I'^2(p' + r)} \cdot \sqrt{\sum_{r \in N} I'^2(p + r)}}$$

Formel 2-17: Normalized Cross Correlation

Dabei steht `I` für die Intensität des beobachteten Pixels, das `p` bezeichnet die Koordinaten des Punktes und `r` steht für die Strecke die der Punkt voraussichtlich zurückgelegt hat.

Dies sind drei am häufigsten eingesetzte Verfahren, um die Ähnlichkeit Zweier Blöcke zu bestimmen. Die Verfahren SAD und SSD zeigen die Unterschiede der Blöcke auf, also umso größer das Ergebnis, desto unterschiedlicher sehen sich die Blöcke. Um die beste Korrelation

der Blöcke zu finden sucht man daher nach dem minimalen Ergebnis. Im Gegensatz dazu sucht man mit dem NCC die Ähnlichkeit, so stellt der Maximalwert dieses Verfahrens die beste Übereinstimmung der Blöcke dar.

Methoden zur Ähnlichkeitsuntersuchung gibt es ebenfalls zu genüge, sie alle versuchen auf dem kürzestem Wege, die größte Übereinstimmung zweier Blöcke heraus zu finden. Das einfachste Vorgehen dabei wäre die Blöcke so in einander zu verschieben, dass am ende des Vorgangs jeder Pixel des ersten Blockes, jedem Pixel aus dem zweiten mindestens einmal gegenübergestellt worden wäre und zu jeder dieser Konstellation einen Ähnlichkeitsfaktor zu bestimmen. Mit dieser Methode wäre sichergestellt, dass das Ergebnis mit der größten Ähnlichkeit, tatsächlich die optimale Lösung darstellt. Betrachtet man aber die Tatsache, dass man dabei mindestens N mal einen Ähnlichkeitsfaktor bestimmen muss, dessen Komplexität, je nach dem wie weit die Blöcke in einander verschoben sind, variiert. Für einen solchen Vergleich zweier Blöcke, dessen Kante 8 Pixel groß ist, wären 224 Schritte nötig, wobei im Schnitt, bei jedem Schritt 16 Faktoren bestimmt werden müssen. Abbildung 2-13 beschreibt ein solches Verfahren, wobei man die Fläche unter der Kurve als Komplexität des Verfahrens betrachten kann.

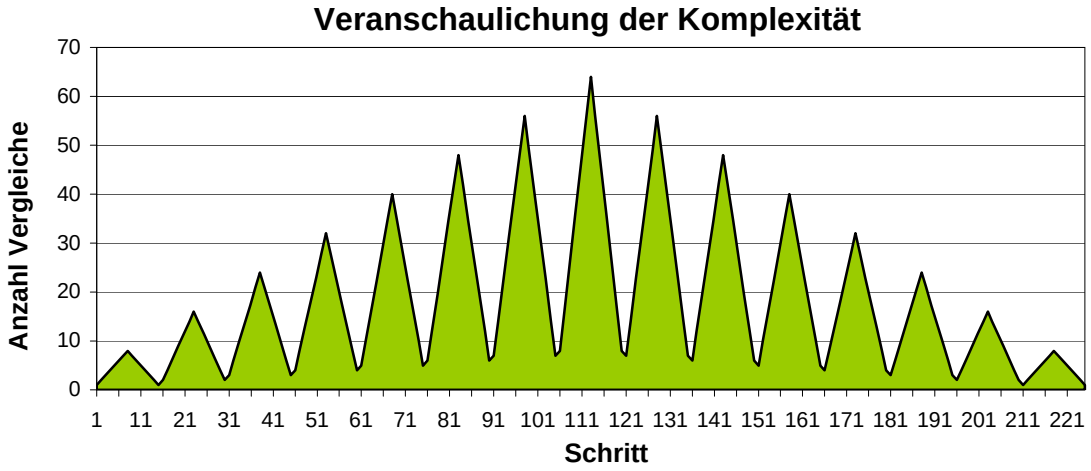


Abbildung 2-13: Komplexität der einfachen Ähnlichkeitsbestimmung

Zur Lösung dieses Problems gibt es allerdings einige Vorgehensweisen, die nicht so Zeit- und Rechenintensiv sind wie zum Beispiel der Three-Step Search (TSS) Algorithmus oder der Two Dimensional Logarithmic Search (TDL) Algorithmus, die im nächsten Kapitel dargestellt werden.

Three-Step Search (TSS)

Der Three-Step Search Algorithmus zeichnet sich dadurch aus, dass es sehr simpel und schnell ist, nur wenig Arbeitsspeicher benötigt, dabei aber nahezu optimale Ergebnisse liefert. Der Name Three-Step Search spricht in dem Fall für sich, die Suche des optimalen Blockausschnitts, mit dem größten Ähnlichkeitsfaktor, wird in drei Schritten ausgeführt.

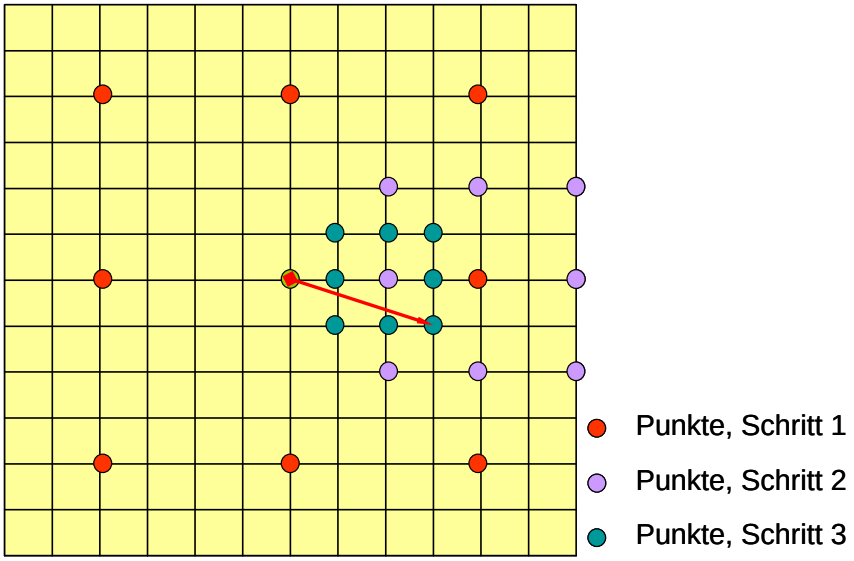


Abbildung 2-14: Three-Step Search

Zu Beginn werden die zu untersuchenden Frames in neun gleichgroße bereiche Geteilt. Die Mitten dieser Bereiche sind in der Abbildung 2-14 mit roten Punkten markiert. Im ersten Schritt wird Frame 1 mit seinem Zentrum jeweils zu jedem dieser Punkte angelehnt und aus den dabei entstandenen Konstellationen, jeweils die Ähnlichkeitsfaktoren bestimmt. So kriegt jeder, der Mittelpunkte einen eigenen Ähnlichkeitsfaktor zugewiesen. Es wird bestimmt welche dieser Konstellationen die geringste Abweichung ergibt und der dazugehörige Mittelpunkt des Bereiches wird nun als Ausgangspunkt für den zweiten Schritt. Um das, neu bestimmte Zentrum der Suche werden wieder acht Punkte verteilt und die Entfernung der Punkte von einander wird im Vergleich zum Schritt 1 halbiert. Die Vorgehensweise aus dem ersten Schritt wird diesmal mit den neuen Punkten wiederholt. Man erhält ein neues Zentrum und legt acht Punkte um diesen, wieder mit der Halben Entfernung zwischen den Punkten im Vergleich zum vorigen Schritt. Im Schritt drei wird analog vorgegangen, nur dass diesmal der Punkt mit dem größten Ähnlichkeitsfaktor die Position der Spitze, des Bewegungsvektors bezeichnet, ausgehend vom Zentrum des Blockes.

Two Dimensional Logarithmic Search (TDL)

Diese Suche ist der zuvor erklärten Three-Step Suche sehr ähnlich, dass dabei mehr als drei Schritte gemacht werden macht dieses Verfahren für größere Bildausschnitte geeigneter. Dazu kann man bei der TDL eine größere Genauigkeit erwarten, da dieser Algorithmus soweit rechnet bis keine Verkleinerung des Suchbereichs mehr möglich ist.

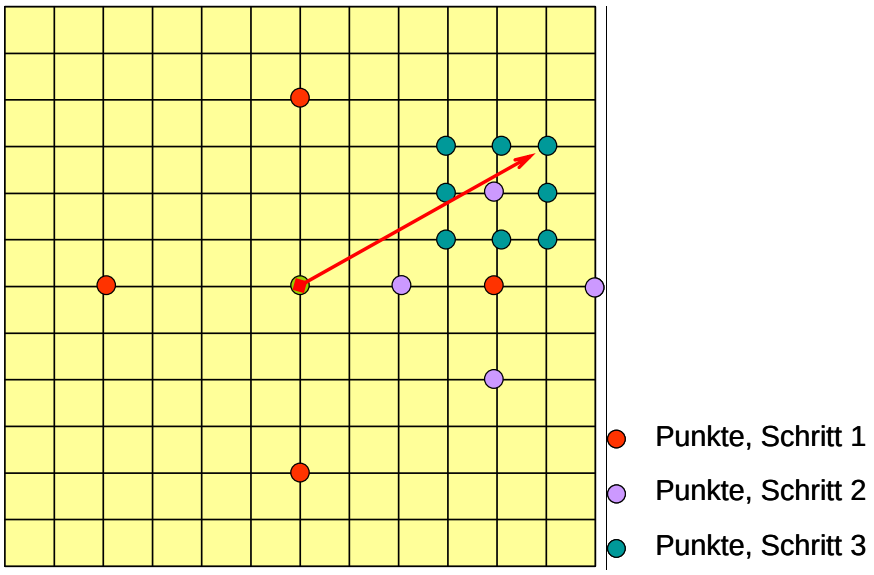


Abbildung 2-15: Two Dimensional Logarithmic Search

Im ersten Schritt bildet man ein Kreutz aus fünf Punkten in einem Bildabschnitt. Genau wie bei der Three-Step Suche werden diese Punkte jeweils mit dem Mittelpunkt des anderen Frames abgefahren und zu jedem dieser Punkte Ähnlichkeitsfaktoren bestimmt. Der Punkt mit der größten Ähnlichkeit wird wieder das neue Zentrum der Suche. Die Abstände der Punkte werden halbiert. Dieses Vorgehen wird so lange wiederholt bis der Abstand zwischen den Punkten ein Pixel beträgt, dann werden alle neun den Mittelpunkt umschließende Pixel abgefahren und der Pixel mit der größten Ähnlichkeit wird zur Spitze des Vektors der Bewegung.

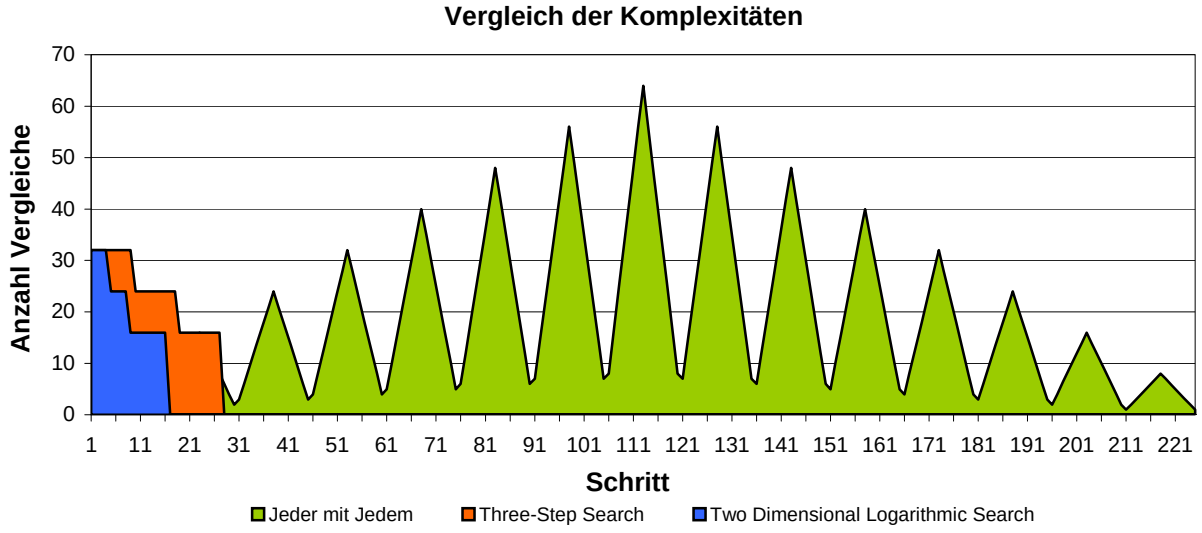


Abbildung 2-16: Komplexität der Methoden im Vergleich

In direktem Vergleich merkt man den unterschied der Suchalgorithmen Während die Methode „Jeder mit Jedem“ je nach Bildüberlappung, mehr oder weniger Berechnungen anstellen muss,

sind die beiden fortschrittlicheren Algorithmen gleich von Anfang an mit vielen Berechnungen belastet, da diese aber stichprobenartig durchgeführt werden, sind sie dementsprechend auch in wenigen Schritten beim gesuchten Pixel, oder in seiner unmittelbaren Nähe. [5][12]

2.5.2. Vor- und Nachteile der Korrelationsverfahren

Zu den größten Vorteilen der Korrelationsbasierten Verfahren zur Erfassung des optischen Flusses gehört ihre relativ gute Verständlichkeit, der unwesendliche mathematische Hintergrund und geringe Komplexität, was die Realisierung einfacher und weniger Fehleranfällig macht. Es ist aber gerade diese Einfachheit von Block Matching, das was für diese Methode zum Nachteil wird. Denn die Suche nach dem ähnlichen Block, im zweiten Bild ist sehr rechenintensiv und benötigt eine menge Zeit und Speicher. Was im Gegenzug einen relativ exaktes Ergebnis erwarten lässt. Doch auch das ist eine Täuschung, denn die blockweise Suche nach Ähnlichkeiten ist sehr unzuverlässig und anfällig für die bekannten Probleme der Bewegungsschätzung, Korrelations- und Blendenproblem lassen Fehleinschätzungen des Flusses sehr oft entstehen.

2.6. Gradientenverfahren

Die Annahme, die bei allen Gradientenverfahren gültig ist, ist die dass ein Objekt im Bild immer eine gleich bleibende Intensität besitzt. Ein Punkt zur Zeit `t`, an der Stelle `(x,y)`, befindet sich zur Zeit `t + dt` an der Stelle `(x+dx,y+dy)`. Würde sich die Intensität zeitlich verändern, würde die Formel 2-18 gelten.

$$I(x+dx, y+dy, t+dt) = I(x, y, t) + \frac{\partial I}{\partial x}\cdot dx + \frac{\partial I}{\partial y}\cdot dy + \frac{\partial I}{\partial t}\cdot dt$$

Formel 2-18

Da die Annahme, der Konstanten Intensität gilt, bedeutet dies, dass die Summe der Partiellen Differenzierung in `x`, `y`, `t` von `I` den Wert Null ergibt (siehe Formel 2-19).

$$\frac{\partial I}{\partial x}\cdot dx + \frac{\partial I}{\partial y}\cdot dy + \frac{\partial I}{\partial t}\cdot dt = 0$$

Formel 2-19

Mit der Substitution der beiden Geschwindigkeiten `u` und `v`, mit Formel 2-20 und Formel 2-21 Erhält man die Formel 2-22, die Horn & Schunk, 1981 der Welt präsentierten.

$$u = \frac{dx}{dt}$$

Formel 2-20

$$v = \frac{dy}{dt}$$

Formel 2-21

$$\frac{\partial I}{\partial x} \cdot u + \frac{\partial I}{\partial y} \cdot v = -\frac{\partial I}{\partial t}$$

Formel 2-22: Differentieller Ansatz zur Bestimmung des optischen Flusses

An dieser Stelle setzen die beiden Verfahren von Lucas & Kanade und Horn & Schunck an. Es gilt eine Gleichung zu lösen die Zwei Unbekannte besitzt, also brauch es noch eine zusätzliche Bedingung um eine eindeutige Lösung zu kriegen. [3]

2.6.1. Horn & Schunck Algorithmus

Die zusätzliche Bedingung, die zur Lösung beitragen soll ist bei Horn & Schunck die so genannte Glattheitsbedingung. Die besagt, dass das Lokale Vektorfeld eines kleinen Bildabschnitts fast konstant ist, d.h. Bewegungsvektoren unterscheiden sich vom Pixel zu Pixel nur geringfügig, in ihrer Richtung und Länge (glattes Vektorfeld). Zu Beginn nimmt man an, dass das Vektorfeld konstant ist und es einige Abweichungen vom allgemeinen Richtung und Länge des Feldes gibt, die wie in Formel 2-23 dargestellt beschrieben werden können.

$$e_s(u,v) = \int\int \|\Delta u\|^2 + \|\Delta v\|^2 dxdy = \int\int \left(\left(\frac{\partial u}{\partial x}\right)^2 + \left(\frac{\partial u}{\partial y}\right)^2 + \left(\frac{\partial v}{\partial x}\right)^2 + \left(\frac{\partial v}{\partial y}\right)^2 \right) dxdy$$

Formel 2-23: Abweichung von der H&S-Bedingung [3]

Die Helligkeitskonstanzgleichung, Formel 2-22 kann man hierfür auch aufschreiben als:

$$e_d(u,v) = \iint \left(\frac{\partial I}{\partial x} \cdot u + \frac{\partial I}{\partial y} \cdot v + \frac{\partial I}{\partial t} \right)^2 dxdy$$

Formel 2-24

Aus den Gleichungen in Formel 2-23 und Formel 2-24 ergibt sich das zu minimierende Gesamtfunktional der Methode nach Horn & Schunck:

$$e_{H\&S}(u,v) = e_d(u,v) + \lambda \cdot e_s(u,v)^{1}$$

Formel 2-25: Gesamtfunktional der Methode nach Horn & Schunck

2.6.2. Lucas & Kanade Algorithmus

Die Bedingung, die Lucas & Kanade in ihrer Berechnung des Optischen Flusses zur Annahme machen, ist die weiterführende Glattheitsbedingung des Horn & Schunck Algorithmuses. Diese besagt, dass der optische Fluss innerhalb eines kleinen Bildbereichs konstant ist. So erhalten wir eine Nachbarschaftsmatrix, mit der Kantenlänge m, mit Pixel für die alle die Helligkeitskonstanzgleichung gilt, was zu einem System von $N=m^2$ linearen Gleichungen führt. Für unsere bisherige Gleichung aus Formel 2-22 heißt es, wenn die Bewegungskomponenten u und v zu einem Vektor zusammengefasst werden, erhält man die Formel 2-26

$$\left[\frac{\partial I}{\partial x}, \frac{\partial I}{\partial y}\right] \cdot \begin{bmatrix} u \\ v \end{bmatrix} = -\frac{\partial I}{\partial t}$$

Formel 2-26

Wendet man darauf die Methode „Kleinste- Quadrat- Minimierung“[2] an, so erhält man eine Energiegleichung die es zu minimieren gilt.

$$E = \sum_{k \in N} \left\| \left[\frac{\partial I_k}{\partial x}, \frac{\partial I_k}{\partial y}\right] \cdot \begin{bmatrix} u \\ v \end{bmatrix} + \frac{\partial I_k}{\partial t} \right\|^2 \overset{!}{=} \min$$

Formel 2-27

Zur Lösung Führt dann folgende Formel:

[1] Die beispielhafte Lösung der Funktion findet man in der Präsentation, zur Veranstaltung „Bildverarbeitung und Computer Vision“, der Universität Münster, im Kapitel 13 – Optische Flüsse (Literatur [7]).

[2] Die Methode der kleinsten Quadrate (englisch: method of least squares) ist das mathematische Standardverfahren zur Ausgleichungsrechnung. Dabei wird zu einer Datenpunktwolke eine Kurve gesucht, die möglichst nahe an den Datenpunkten verläuft. Die Daten können physikalische Messwerte, wirtschaftliche Größen oder Ähnliches repräsentieren, während die Kurve aus einer parameterabhängigen problemangepassten Familie von Funktionen stammt. Die Methode der kleinsten Quadrate besteht dann darin, die Kurvenparameter so zu bestimmen, dass die Summe der quadratischen Abweichungen der Kurve von den beobachteten Punkten minimiert wird. [6]

$$\begin{bmatrix} \sum_{k\in N}\left(\frac{\partial I_k}{\partial x}\right)^2 & \sum_{k\in N}\frac{\partial I_k}{\partial x}\cdot\frac{\partial I_k}{\partial y} \\ \sum_{k\in N}\frac{\partial I_k}{\partial x}\cdot\frac{\partial I_k}{\partial y} & \sum_{k\in N}\left(\frac{\partial I_k}{\partial y}\right)^2 \end{bmatrix} \cdot \begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} \sum_{k\in N}\frac{\partial I_k}{\partial x}\cdot\frac{\partial I_k}{\partial t} \\ \sum_{k\in N}\frac{\partial I_k}{\partial y}\cdot\frac{\partial I_k}{\partial t} \end{bmatrix}$$

Formel 2-28: Lösungsgleichung zum Lucas & Kanade Algorithmus

2.7. Vor- und Nachteile der Gradientenverfahren

Gradientenverfahren zur Schätzung des optischen Flusses werden meistens mit ihrer, relativ hohen Genauigkeit in Verbindung gebracht. Tatsächlich liefern sowohl das Verfahren von Horn & Schunck als auch der Lucas & Kanade Algorithmus, sehr genaue Ergebnisse, was für sie spricht. Was aber schon die mathematische Komplexität vermuten läst, sind diese Methoden zu rechenintensiv, was eine Echtzeittauglichkeit fraglich macht. Zudem kommt noch die Tatsache, dass Gradientenverfahren nur eine lokal begrenzte Bewegungsschätzung erlauben, die meistens nur wenige Pixel groß ist.

Diese Probleme lasen sich aber mit relativ wenig Aufwand in den Griff kriegen. Zur Minimierung des zeitlichen und rechnerischen Aufwands, kommen so genannte Featuretracker zum Einsatz. Dies sind Verfahren, die bereits vor der eigentlichen Bewegungsschätzung Pixelmuster in den Bildern suchen und zur weiteren Verfolgung nominieren. Die Grauwertbewegung wird danach nur in der unmittelbaren Nähe der Features geschätzt, was eine erhebliche Ersparnis an Rechenleistung herbeiruft. Es muss natürlich bedacht werden, dass dabei kein zusammenhängendes Vektorfeld entsteht, was Bewegungen jedes Pixels erfasst. Was aber unseren Anforderungen an den optischen Fluss vollkommen genügt, wenn das Ergebnis genau genug ist.

2.7.1. Gaußpyramiden

Abbildung 2-17: Gaußpyramide

Zur Eliminierung des Problems, dass sich die Gradientenverfahren nicht über viele Pixel hinweg Objekte einander zuordnen können und so mit keinen optischen Fluss erkennen, kommen die Gaußpyramiden zum Einsatz.

Die Funktion der Gaußpyramiden ist sehr einfach aber effizient, durch die Verringerung der Auflösung wird das gesamte Bild mit weniger Pixel dargestellt, so mit haben die Gradientenverfahren keine Probleme auch Bewegungen zu erfassen, die über weite Strecken gehen.

2.7.2. Feature Selektion

Zur Verwendung von Features bei der Suche nach optischen Flüssen, griff man wegen mehreren Umstände. Der ausschlaggebende war das Aperturproblem und das allgemeine Korrelationsproblem. Zur Erinnerung, das Aperturproblem besagt dass bei einer geraden Kannte des Objekts, nur der Teil der Bewegung erfasst werden kann, der senkrecht zur Kannte verläuft. Die Idee der Forscher war, nur Ecken oder Regionen mit vielen, eindeutigen Texturen zu verfolgen, solche Regionen des Bildes nennt man “Trackabe Features“.

Das Verfahren der Featuresuche, welches mit der Funktion *cvGoodFeaturesToTrack(),* in der OpenCV-Bibliothek implementiert ist, wurde von Carlo Tomasi und Takeo Kanade, im April 1991, in dem Bericht „Detection and Tracking of Point Features“ (Literatur [8]) veröfentlicht. Dort werden mathematische Herleitung und die Zusammenhänge der Featuresuche erklärt. Im Grunde ist es eine Untersuchung, der in der Formel 2-28: Lösungsgleichung zum Lucas & Kanade Algorithmus vorgestellten Gleichung zur Erfassung des optischen Flusses. Angewandt auf ein Bildausschnitt, liefert die 2x2 Matrix Indikationswerte, die im Verhältnis zu einander, Aussagen über den Inhalt des Fensters treffen lassen. So kann man eine Kante von einer Ecke oder einen Bereich ganz ohne zuordenbare Features unterscheiden.

In Verbindung mit den Gaußpyramiden und den Featuretrackern zeigen die Gradientenverfahren sehr gute Ergebnisse, und erlauben ohne hochgesteckte Anforderungen an die Rechenleistung zu haben, auch Echtzeitproblematiken anzugehen. In der Literatur findet man Zahlreiche Aussagen darüber, wie radikal sich die Anwendung von Gaußpyramiden auf die Fähigkeiten und Anforderungen der Gradientenverfahren auswirkt. Und obwohl die meisten, die Meinung vertreten, dass der Algorithmus von Horn & Schunck in Kombination mit den Pyramiden bessere Ergebnisse liefert. In der OpenCV-Bibliothek wird nur der Ansatz von Lucas & Kanade um die Gaußpyramiden erweitert, wohl aus dem Grunde, dass der Lucas & Kanade Algorithmus eine Vereinfachung des von Horn & Schunck ist und schon in der unmodifizierten Version Weniger Zeit- und Speicheraufwändig ist. [8]

2.8. OpenCV

OpenCV ist eine quelloffene Programmbibliothek unter BSD-Lizenz[1], kann also auch kommerziell frei genutzt werden. Sie ist für die Programmiersprachen C und C++ geschrieben und enthält Algorithmen für die Bildverarbeitung und maschinelles Sehen. Das CV im Namen steht für Computer Vision. Die Entwicklung der Bibliothek wurde von Intel initiiert und wird heute hauptsächlich von Willow Garage gepflegt. Im September 2006 wurde die Version 1.0 herausgegeben, Ende September 2009 folgte nach längerer Pause die Version 2.0.0, welche die Bezeichnung "Gold" trägt. Die Stärke von OpenCV liegt in ihrer Geschwindigkeit und in der großen Menge der Algorithmen aus neuesten Forschungsergebnissen. Die Bibliothek umfasst unter anderem Algorithmen für Gesichtsdetektion, 3D-Funktionalität, Haar-Klassifikatoren, verschiedene sehr schnelle Filter (Sobel, Canny, Gauß) und Funktionen für die Kamerakalibrierung. [1]

OpenCV-Bibliothek ist frei im Internet erhältlich, und eignet sich für die unterschiedlichsten Bildverarbeitungsaufgaben.

Beschäftigt man sich mit OpenCV eine kurze Zeit, merkt man, dass es ein starkes Werkzeug ist, wenn es darum geht Informationen Bildern oder Videosequenzen zu entlocken. Folgende Disziplinen Zählen zu den wichtigsten OpenCV-Aufgaben:

- Mensch-Computer Interaction (HCI)
- Object Identifikation, Segmentierung und Erkennung
- Gesichtserkennung
- Gestenerkennung
- Objektverfolgung
- Structure From Motion (SFM)[2]

---

[1] BSD-Lizenz, steht für Berkeley Software Distribution - Lizenz und bezeichnet eine Gruppe von Lizenzen aus dem Open-Source-Bereich. Der Urtyp der Lizenz stammt von der University of California, Berkeley. Software unter BSD-Lizenz darf frei verwendet, kopiert, verändert und verbreiten werden. Einzige Bedingung ist, dass der Copyright-Vermerk des ursprünglichen Programms nicht entfernt werden darf. Somit eignet sich unter einer BSD-Lizenz stehende Software auch als Vorlage für kommerzielle Produkte.

[2] Structure From Motion, Nennt man in der Wissenschaft das Aufbauen von Dreidimensionalen Räumen, aus Zweidimensionalen Bildsequenzen. Dabei verlässt man sich im Wesentlichen auf die Tatsache, dass sich weit entfernte Gegenstände, optisch langsamer bewegen als diejenigen die sich in der Nähe der Kamera aufhalten.

- Mobile Robotics

2.8.1. Aufbau von OpenCV

Die OpenCV-Bibliothek läst sich in 4 wesendliche Teile aufgliedern.

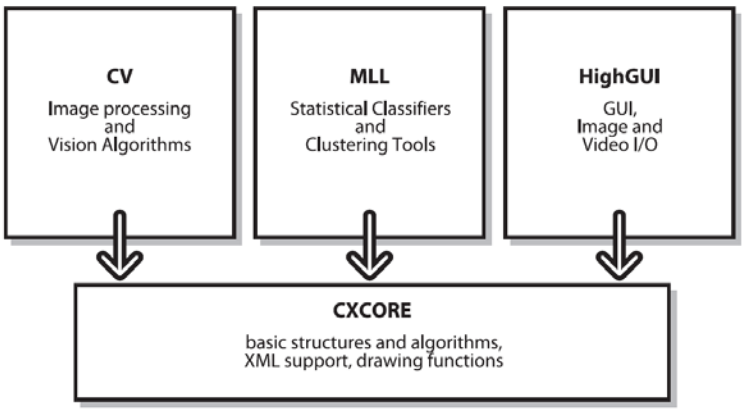


Abbildung 2-18: Aufbau der OpenCV-bibliothek (Quelle[2])

- CV steht für *Computer Vision* und benennt alle Algorithmen zur Bild- und Videoverarbeitung, so wie das Verzerren der Bilder (scheren, drehen, skalieren), oder Konvertierungen in verschiedene Bildformate.

- MLL heißt *Mashine Learning Library*, es beinhaltet statistische Methoden, Klassifikatoren, Clustering Tools und markiert somit den künstlichen Intellekt von OpenCV.

- HighGUI ist für die Kommunikation mit dem Benutzer zuständig, Fensterroutine, Geometrie, aber auch das Laden, das Lesen und Schreiben der Dateien gehört zu ihren Aufgaben.

- CXCORE stellt den Datenverkehr und Kommunikation zum Betriebssystem dar und sorgt für einen reibungslosen Arbeitsablauf.

2.8.2. Installation und Einstellungen von OpenCV

Es existieren sehr viele Beschreibungen der Installation von OpenCV, ob unter Windows oder Mac Systemen, auf Visual Studio oder Eclipse, für C oder C++. Die Meisten von ihnen sind sehr akkurat und detailliert beschrieben und enthalten genügend Bildmaterial um den Benutzer sicher zum Ziel zu führen.

Die Installation ist simpel und Schnell erledigt, bei der Vorbereitung braucht man nichts zu verstellen, auch den Installationspfad sollte man mit „C:\OpenCVX.X“ belassen. Mit rund 130 MB nimmt die Bibliothek nicht viel Platz auf der Festplatte ein und ist sofort einsatzbereit.

Beim Arbeiten mit Eclipse sollten folgende Enstellungen gemacht werden:

Nach dem ein C-Projekt erstellt ist, müssen in den Einstellungen des Projects, einige wenige Veränderungen vorgenommen werden. Über rechte Maustaste, „Properties“, auf dem Projektsymbol gelangt man ins Einstellungsmenü, dort wählt man die Rubrik „Settings“ und in dem Register „Tool Settings“, bei „GCC C Compiler“ -> „Directories“ ergänzt man, im Fenster „Include Paths“, die Einträge um den aktuellen Pfad zum OpenCV-Ordner, zum Beispiel „C:\OpenCV2.0\include\opencv“ (siehe Abbildung 2-19).

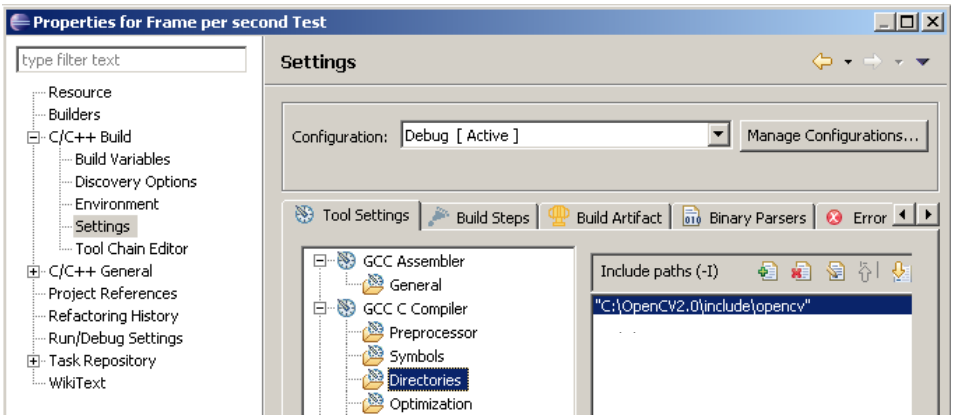


Abbildung 2-19: Einstellungen von Eclipse 1

Des Weiteren müssen noch die Bibliotheken, die man benutzen will definieren. Dies sind „libcv200“, „libcxcore200“, „libcxts200“, „libhighgui200“, „libcvaux200“, „libml200“, bei OpenCV2.0. Dafür schreibt man diese Bezeichnungen, unter „MinGW C Linker“ -> „Libraries“ in das Fenster „Libraries (-l)“. In dem Fenster „Library search path(-L)“ trägt man den Pfad zu den Bibliotheken ein, bei Standarteinstellungen lautet er „C:\OpenCV2.0\lib“ (siehe Abbildung 2-20).

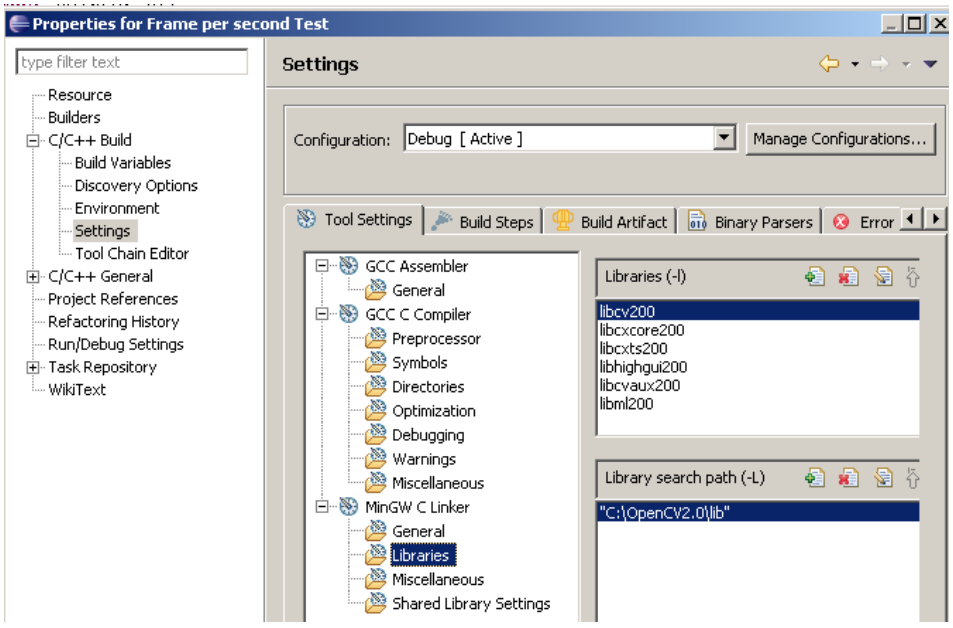


Abbildung 2-20: Einstellungen von Eclipse 2

Die Bezeichnungen können von Version zu Version variieren, deswegen ist man gezwungen diese Einstellungen zu aktualisieren, falls man eine andere Version von OpenCV verwenden möchte. [11]

Die Einstellungen in Visual Studio sehen so aus, Nach dem ein C-Projekt angelegt ist, geht man zu seinen Eigenschaften. Dort beim Punkt „C/C++“ -> „General“ im Fenster „Additional Include Directories“ fügt man den Pfad zum include-Ordener ein, er hat folgende Form „X:\OpenCVX.X\include\opencv“ (siehe Abbildung 2-21). Und unter dem Punkt „Linker“ -> „General“ fügt man im Feld „Additional Library Directories“ den Pfad zum lib-Ordner des OpenCV ein (siehe Abbildung 2-22). Im Punkt „Linker“ -> „Input“ fügt man die Bezeichnungen der *.lib Files, in das Fenster „Additional Dependencies“ (siehe Abbildung 2-23). [10]

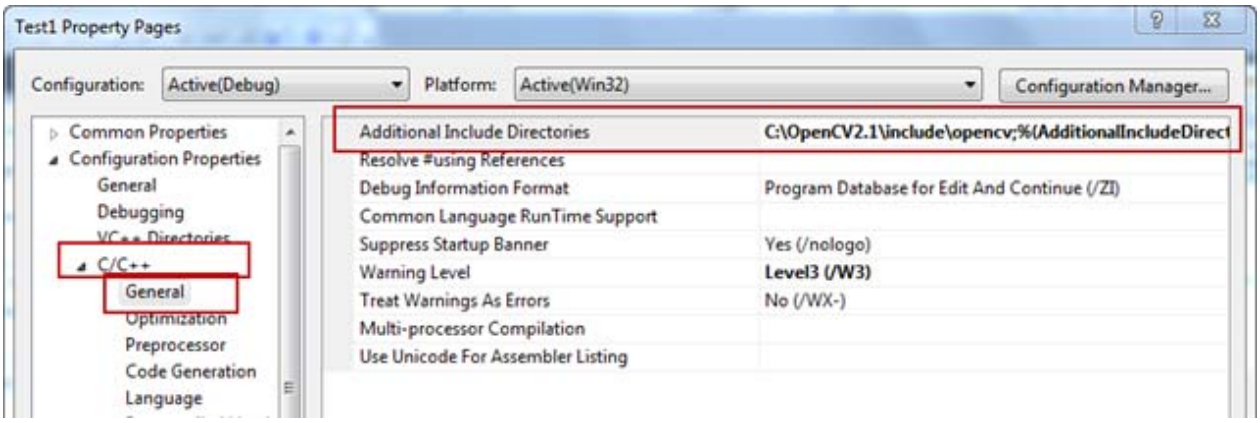


Abbildung 2-21: Entstellungen von Visual Studio 1

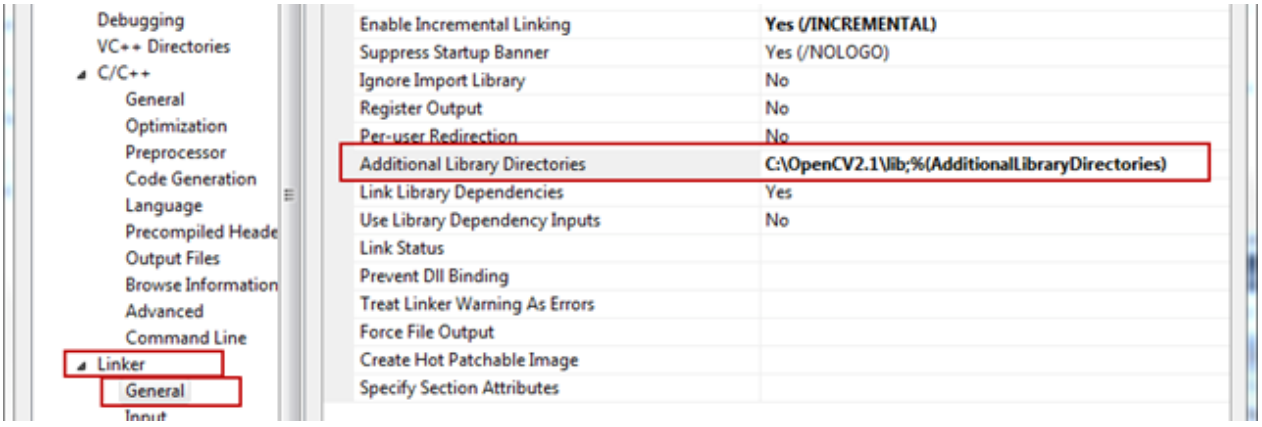


Abbildung 2-22: Einstellungen von Visual Studio 2

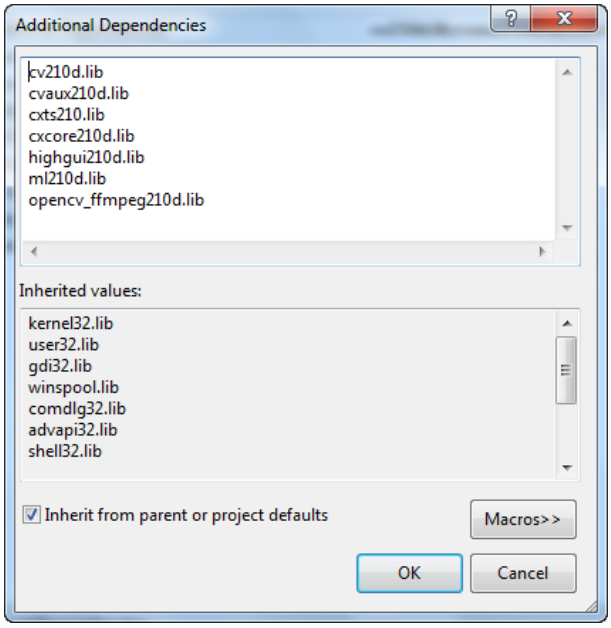


Abbildung 2-23: Einstellungen von Visual Studio 3

3. Ausgangssituation

3.1. Quadrocopterprojekt

Der Quadrocopter der Hochschule Esslingen Abbildung 3-1, ist ein Flugapparat an dem die Zukunft der Fortbewegung erprobt wird. wenn die Straßen immer überfühlter werden gibt es keine andere Möglichkeit, als in die Lüft auszuweichen, um aber der großen Masse die Möglichkeit zu geben solche Flugkörper zu steuern, muss die eigentliche Steuerung von der Maschine selber übernommen werden. Zu einem weil das Fliegen an sich, im Vergleich zu Autos die sich auf der zweidimensionalen Straße bewegen, einen Dimension mehr besitzt, die vom Pilot noch mehr Aufmerksamkeit und Konzentration verlangt. Zum Anderen muss so ein Apparat von einem herkömmlichen Autofahrer bedient werden können da man nicht davon ausgehen darf das die Ausbildung dieser Piloten länger dauert und damit teurer wird. Also muss die Maschine dem Fahrer Grenzen setzen, auf die er sich verlassen kann um sicher ans Ziel zu kommen. Dazu muss eine Fülle an Reglungsmechanismen entworfen werden, die alle im Einklang arbeiten, ohne sich gegenseitig zu stören.

Der Quadrocopter der Hochschule Esslingen Realisiert viele der geforderten Mechanismen, vom Betriebsystem bis hin zur Regelung und Steuerung solcher Apparate. In Zahlreichen Projekten, Studien- und Bachelorarbeiten untersuchen Studenten der Hochschule gemeinsam mit ihren Betreuern und Herrn Prof. Dr. Friedrich, die dafür notwendigen Hard- und Softwarelösungen. Um die bereits vorhandenen Methoden zu verstehen und neue Lösungsmöglichkeiten zu erarbeiten.

In dieser Arbeit untersucht man Methoden zur Erfassung des Optischen Flusses, diese sollen in Zukunft dafür eingesetzt werden um den Quadrocopter stabil fliegen zu lassen. Das heißt ohne großes Eingreifen des Piloten solche Manöver, wie auf einer Stelle schweben müssen realisierbar sein. Vier Solcher Methoden bietet die OpenCV-Bibliothek. Diese Werden in folgendem Abschnitt benannt, ihre Vor- und Nachteile durch diverse Tests ermittelt und aufgezeigt und ihre Bedienung erläutert.

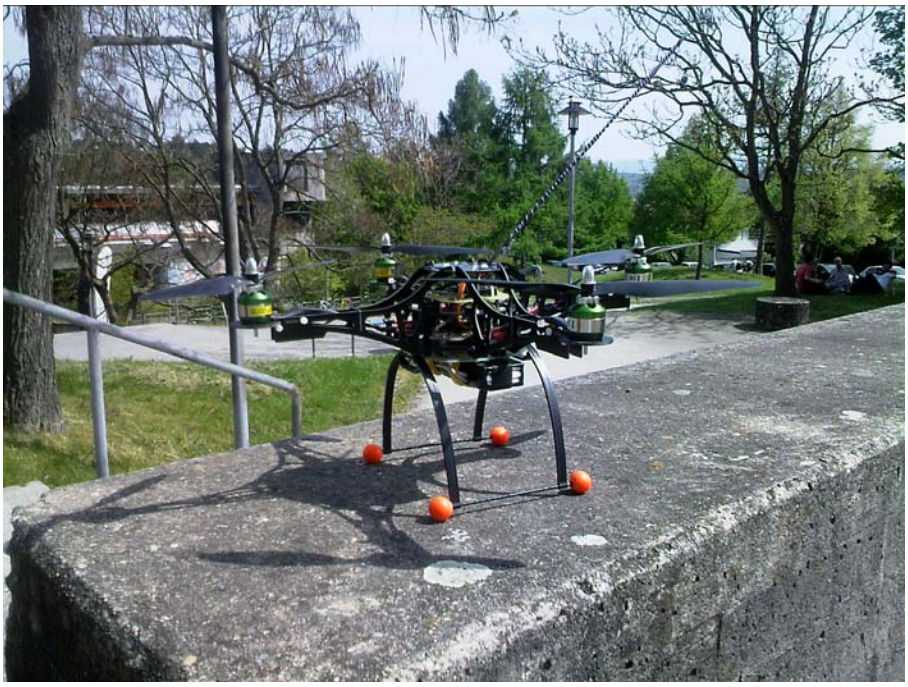


Abbildung 3-1: Quadrocopter der Hochschule Esslingen

3.2. Beschreibung des Uhrsprungprogramms

Das zugrunde liegende Programm, ist ein Beispielcode, des DARPA Grand Challenge[1] Teams von Standfort University. Die Programmiersprache ist C, mit Funktionen der Freien Bibliothek OpenCV. Es implementiert die Erfassung optischer Flüsse in einer AVI- Videodatei. Das Verfahren, das dabei zum Einsatz kommt ist der Lucas & Kanade Algorithmus, in Verbindung mit einem Feature Tracker und den Gauß-Auflösungspyramiden.

Der Ablauf des Programms sieht folgendermaßen aus. In der Anfangsphase werden Variablen deklariert, Pointer auf das zu untersuchende Video angelegt, je nach Auflösung der Frames, Platz im Speicher reserviert. Der wesentliche Teil des Ablaufs passiert in einer endlosen while-Schleife. Dort werden, pro Durchgang zwei Frames aus dem Videofile gelesen und in Schwarzweißbilder umgewandelt, spätere Schätzung des optischen Flusses erfolgt mittels Erfassung der Grauwertverschiebungen, dafür reichen Bilder mit einer Tiefe von acht Bit pro Pixel.

[1] Die DARPA Grand Challenge ist eine von der Technologieabteilung Defense Advanced Research Projects Agency des US-amerikanischen Verteidigungsministeriums gesponserter Wettbewerb für unbemannte Landfahrzeuge. Mit der Ausschreibung des Preises soll die Entwicklung vollkommen autonom fahrender Fahrzeuge vorangetrieben werden.

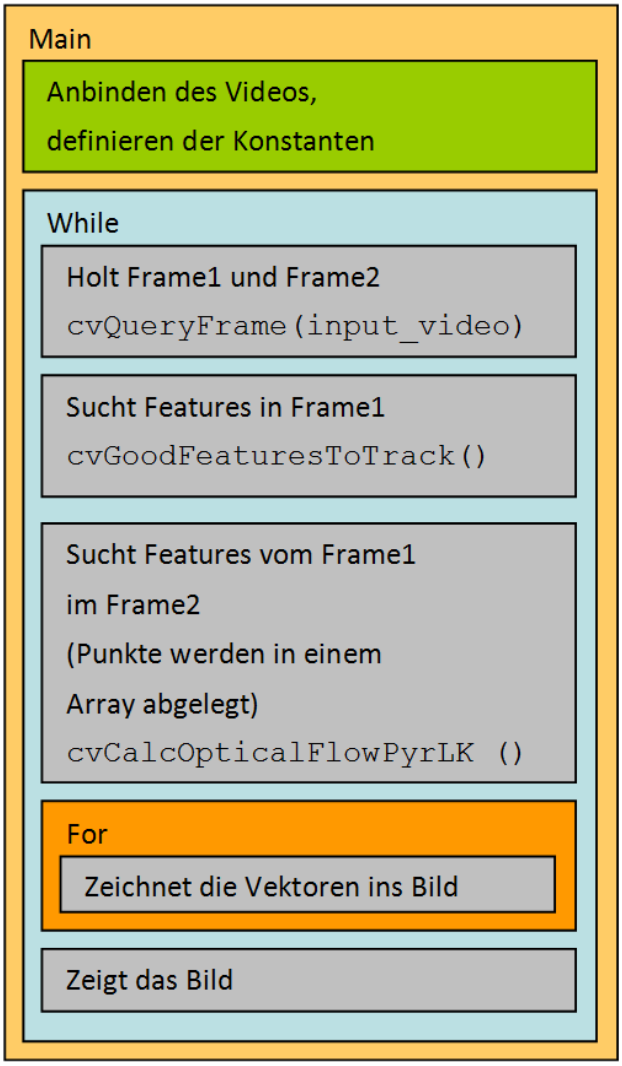


Abbildung 3-2: Grober Programmaufbau

Die Strategie, der Schätzung des optischen Flusses sieht wie folgt aus:

Ein Array von Koordinatenpunkten (CvPoint) wird angelegt, mit der zuvor festgelegten Anzahl an gesuchten Punkten (Features).

```
CvPoint2D32f frame1_features[Anzahl_Features];
```

Abbildung 3-3: Featurearray

Der nach dem Prinzip von Shi & Tomasi Algorithmus arbeitende Feature Tracker, `cvGoodFeaturesToTrack()` (siehe Abbildung 3-4)füllt dieses Array mit Koordinaten von Punkten, die die festgelegten Rahmenbedingungen erfühlen. Diese wären, das Feature darf eine bestimmte Intensitätsgrenze nicht unterschreiten und es muss ein minimaller Abstand zwischen den Features bestehen.

Die Funktion berechnet zunächst den minimalen Eigenwert für jeden Bildpunkt der Quelle, mit der Funktion `cvCornerMinEigenVal()` und legt ihn in `eig_image` ab.

```
public static void cvGoodFeaturesToTrack(
        IntPtr image,
        IntPtr eigImage,
        IntPtr tempImage,
        IntPtr corners,         // frame1_features
        ref int cornerCount,
        double qualityLevel,
        double minDistance,
        IntPtr mask,
        int blockSize,
        int useHarris,
        double k
)
```

Abbildung 3-4: Definition von cvGoodFeaturesToTrack

Danach wird alles, außer den lokalen Maxima in 3x3 Nachbarschaft, entfernt, so bleiben nur die Ecken und Stellen mit einem großen Wiedererkennungswert, zur weiteren Betrachtung. Im nächsten Schritt werden die Stellen entfern, die einen minimalen Eigenwert von weniger als `quality_level*max(eigImage(x,y))` besitzen. Schließlich sorgt die Funktion dafür, dass alle Ecken eine bestimmte Distanz von einander einhalten `min_distance`. Dabei lässt die Funktion bevorzugt die Features fallen, die später gefunden wurden, und aus diesem Grund als schwächer gelten. Nach einem erfolgreichen Durchlauf des Algorithmus, werden die Koordinaten von den gefundenen Featurepunkten in dem Punktearray `corners` abgelegt.

Ein weiteres Array von Punkten wird angelegt, diesmal für das zweite Frame `frame2_features[Anzahl_Features]`. Die zwei bestehenden Arrays werden mit den zu untersuchenden Bildern an die Funktion `cvCalcOpticalFlowPyrLK()` übergeben. Diese Funktion berechnet den optischen Fluss nach dem Lucas & Kanade und Gaußpyramiden Verfahren. Dabei such der Algorithmus eine Pixelwanderung unmittelbar in der Nähe der übergebenen Punkte des ersten Arrays, die Größe der Suchregion wird von `winSize` vorgegeben. Die Gaußpyramiden werden in zwei temporären Bildern aufgebaut (`prevPyr` und `currPyr`), die Anzahl der Pyramidenschichten wird von der Variable `level` vorgegeben. Das Bytearray `status` enthält immer bei den Elementen eine Null, bei denen der Algorithmus keine Übereinstimmung zum jeweiligen Feature aus dem ersten Bild, im Bild zwei finden konnte. Im Gegensatz dazu enthalten im Floatarray `trackError` alle nicht gefundenen Elemente eine Einz.

```
public static void cvCalcOpticalFlowPyrLK(
        IntPtr prev,
        IntPtr curr,
        IntPtr prevPyr,
        IntPtr currPyr,
        PointF[] prevFeatures, // frame1_features
        PointF[] currFeatures, // frame2_features
        int count,
        Size winSize,
        int level,
        byte[] status,
        float[] trackError,
        MCvTermCriteria criteria,
        LKFLOW_TYPE flags
)
```

Abbildung 3-5: Definition von cvCalcOpticalFlowPyrLK

Das `criteria` besteht aus zwei Variablen und sagt aus wann die suche erfolgreich ist, und wann nicht, die erste Variable CV_TERMCRIT_ITER besagt wie viele Widerholungen bei der Suche nach Features gemacht werden dürfen, bevor die suche abgebrochen wird, die zweite CV_TERMCRIT_EPS besagt wie genau die Übereinstimmung der Features sein soll, die erreicht werden muss, um sagen zu Können dass das Feature gefunden wurde. Die Funktion füllt, die dabei entstandene Ergebnisse in das Zweite Punktearray. Alles was noch zu tun bleibt, ist die Punkte aus den Arrays jeweils zu verbinden

$$v \approx \begin{bmatrix} p_{Arr1i} \\ p_{Arr2i} \end{bmatrix} \qquad \text{mit } 0 < i \leq \text{Anzahl der Features.}$$

Formel 3-1: Bildung der einzelnen Vektoren

In einer For-Schleife werden die einzelnen Vektoren ins Bild gezeichnet. So erhält man ein, mehr oder weniger dichtes Vektorfeld, mit einer größeren Konzentration an Stellen mit vielen Kontrasten und erkennbaren Bewegungen.

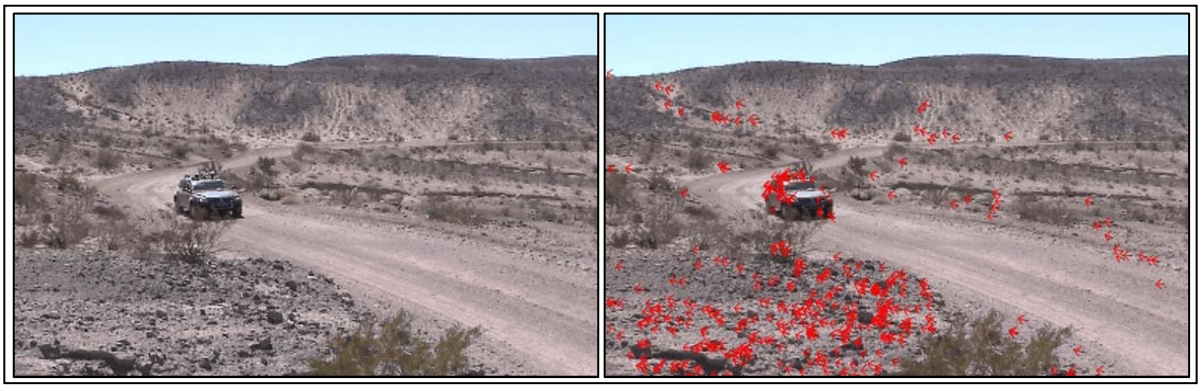


Abbildung 3-6: Input- und Output- Bilder des Programms, des Stanford DARPA Teams

Das Programm zeigt sehr gut die Funktion, der Flusserkennung. In zahlreichen Kommentaren werden die einzelnen Schritte, detailliert erklärt. Die Aufgabe, einen an die Problematik der Flusserkennung und an ihre Lösung mit der OpenCV-Bibliothek, möglichst schnell heran zu führen erfüllt das Beispielprogramm sehr gut. Man kriegt schnell ein Gefühl für den Aufwand der notwendig ist um den optischen Fluss zu erfassen (der zwar in einer niedrigen Qualität und relativ hohem Risiko belastet ist falsche Ergebnisse zu liefern, der allerdings ziemlich früh die eigenen Erfolge feiern läst, da man vieles vorgezeigt bekommt und nicht alles blind ausprobieren muss).

3.3. Weiterentwicklung des Programms

Gefordert ist eine Softwarelösung zur Bewegungsschätzung, um später eine darauf basierende Bewegungsregelung für den hochschuleigenen Quadrocopter zu entwickeln. Das bedeutet es bedarf kein flächendeckendes Vektorfeld, sondern nur einen möglichst genauen Vektor, genauer seine X- und Y-Komponente, die als Steuergrößen im Regler verarbeitet werden können.

So war mein erster Schritt zur Lösung, einen Durchschnittsvektor zu bilden. So rechnete ich in der For-Schleife alle X- und Y-Komponenten der Teilvektoren zusammen und teilte das Ergebnis durch die Anzahl der Features, oder Teilvektoren. Hierbei sollte man von der Anzahl der gefundenen Features ausgehen und nicht von der ursprünglich angesetzten Anzahl der Features. dazu dienen die beiden Arrays `status` und `trackError`, und die Variable `count`. Zur Erinnerung, das `i`-te Element des Arrays `status` enthält eine Null, falls das dazugehörige Feature nicht verfolgt werden konnte, das Array `trackError` wird in diesem Element eine Eins enthalten. Die Variable `count` enthält die Anzahl der Features, sie wird während der Laufzeit des Algorithmus stets nach unten korrigiert, falls ein Feature im zweiten Bild nicht gefunden wurde, so enthält die Variable immer die aktuelle Featureanzahl.

Um eine gewisse Übersicht im Code zu schaffen, wurde das bisherige Programm in mehrere kleinere Teile aufgetrennt und zur Vereinfachung und größerer Verständlichkeit, wurden neue Variablentypen eingeführt (siehe Abbildung 3-7 und Abbildung 3-8).

```
struct TwoFrames {
        IplImage *frame1;
        IplImage *frame2;
        };
```

Abbildung 3-7: Definition der Struktur TwoFrames

```
struct Pointer {
        CvPoint p1;
        CvPoint p2;
        };
```

Abbildung 3-8: Definition der Struktur Pointer

Diese Strukturen dienen nur einer besseren Übersicht und Kompaktheit des Codes. Der eigentliche Anstoß zur Einführung dieser Strukturen, war die Überlegung zwei anschauliche Objekte, als Eingang und Ausgang zu besitzen, die während dem Ablauf des Programms in einander umgerechnet werden. Aus zwei Bildern wird ein Pfeil.

Zur Testzwecken war es notwendig Längenangaben des Pointers erfassen zu können, dazu dienen die Funktionen `LaengePtr`, `LaengePtrX` und `LaengePtrY`, mit

```
int LaengePtr (struct Pointer P){
        return sqrt(square(LaengePtrX (P)) + square(LaengePtrY (P)));
        }
int LaengePtrX (struct Pointer P){
        return (P.p1.x > P.p2.x)? (P.p1.x - P.p2.x) : (P.p2.x - P.p1.x);
        }
int LaengePtrY (struct Pointer P){
        return (P.p1.y > P.p2.y)? (P.p1.y - P.p2.y) : (P.p2.y - P.p1.y);
        }
```

Abbildung 3-9: Funktionen zur Ausgabe der Längen eines Pfeils

und

```
inline static double square(int a){
        return a * a;
        }
```

Abbildung 3-10: Funktion square

Des Weiteren wanderte die gesamte Zeichen-Prozedur, die für das Zeichnen der Pfeile zuständig war, in eine eigene Funktion `FlowPointer()`, die ein Pfeil `ptr` in die Mitte des Übergebenen Frames `frame` zeichnet.

```
int FlowPointer(
        IplImage *frame,
        struct Pointer ptr,
        CvSize frame_size,
        double angle,
)
```

Abbildung 3-11: Definition von FlowPointer

Für die Berechnung des Bewegungsvektors wurden eigene Methoden eingeführt, die unterschiedliche OpenCV- Algorithmen zur Bestimmung des optischen Flusses benutzen, diese werden in einem folgenden Kapitel behandelt.

Es wurde eine Routine eingeführt zur Variierung der Testvariablen, die die Übergabeparameter der Methoden zur Erkennung des optischen Flusses steuern. Folgende Testvariablen werden

erst in ein TXT-File geschrieben, und später händisch ins Excel zur weiteren Untersuchung geladen:

```
Loop: Anzahl vom Testbeginn aufgenommener Datenreihen
Zeit[ms]: Tickzahlen der Funktion clock()
Länge_HS: Pfeillänge der Funktion cvcalcOptikalFlowHS
LängeX_HS: X-Komponente der Funktion cvcalcOptikalFlowHS
LängeY_HS: Y-Komponente der Funktion cvcalcOptikalFlowHS
Länge_LK: Pfeillänge der Funktion cvcalcOptikalFlowLK
LängeX_LK: X-Komponente der Funktion cvcalcOptikalFlowLK
LängeY_LK: Y-Komponente der Funktion cvcalcOptikalFlowLK
Länge_PyrLK: Pfeillänge der Funktion cvcalcOptikalFlowPyrLK
LängeX_PyrLK: X-Komponente der Funktion cvcalcOptikalFlowPyrLK
LängeY_PyrLK: Y-Komponente der Funktion cvcalcOptikalFlowPyrLK
Länge_BM: Pfeillänge der Funktion cvcalcOptikalFlowBM
LängeX_BM: X-Komponente der Funktion cvcalcOptikalFlowBM
LängeY_BM: Y-Komponente der Funktion cvcalcOptikalFlowBM
Param: Die im Aktuellem Test variierte Variable
t_HS: Benötigte Zeit für das cvcalcOptikalFlowHS
t_LK: Benötigte Zeit für das cvcalcOptikalFlowHS
t_PyrLK: Benötigte Zeit für das cvcalcOptikalFlowHS
t_BM: Benötigte Zeit für das cvcalcOptikalFlowHS
```

die Routine für den Test wird im Abschnitt 4.2 „Testdurchführung“ noch näher beleuchtet.

Des Weiteren wurde die Frameroutine so umgestaltet, dass nicht im jeden While-Durchlauf zwei Bilder geholt werden, sondern nur einer (siehe Abbildung 3-12). Die Flusserfassung geschieht jetzt zwischen dem aktuellen und dem vorherigen Bild. Dies bringt viel Zeitgewinn fürs Senden der Bilder und entlastet das spätere ZigBee-Modul.

Durch die genannten Modifikationen im Programmablauf wurde die `main`-Funktion entlastet und gewann an Übersichtlichkeit, was zum Beispiel die Zeitmessung wesentlich einfacher macht.

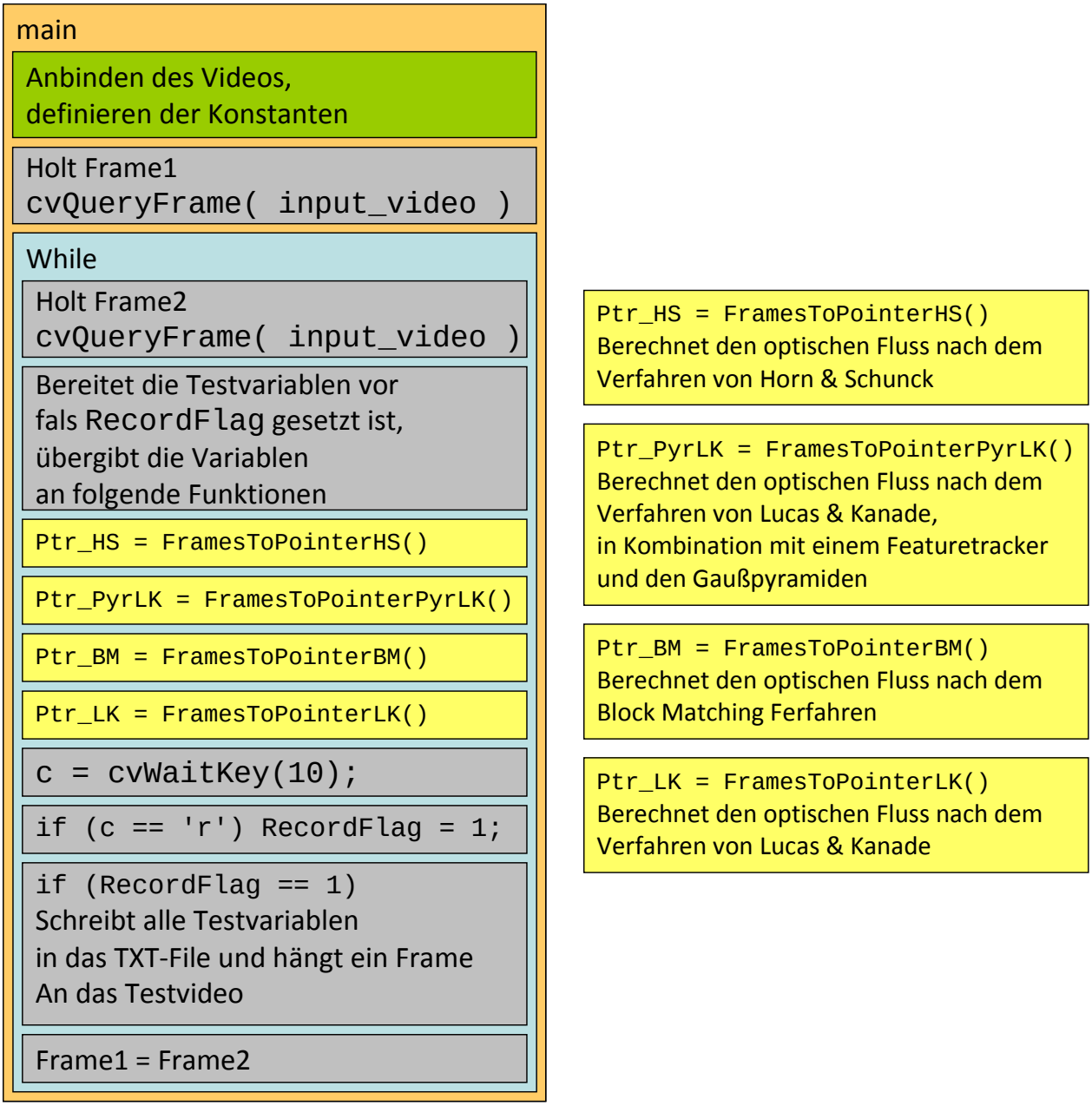


Abbildung 3-12: Grober Programmablauf des Testprogramms der optischen Flusserfassung

3.4. Benötigte Zeichenfunktionen

Die Zeichenfunktionen von OpenCV haben zwar nicht viel mit der Funktionalität der Algorithmen zur Schätzung des optischen Flusses zu tut, doch sind sie sehr Hilfreich wenn es darum geht, Ergebnisse und Zwischeninformationen anschaulich zu präsentieren.

Während der Arbeit mit OpenCV benötigte man im Wesendlichen, drei Zeichenmethoden, die im Folgenden Kapiteln beschrieben werden.

3.4.1. Funktion cvLine()

```
void cvLine(
CvArr * img,
CvPoint pt1,
CvPoint pt2,
CvScalar color,
int thickness=1,
int lineType=8,
int shift=0
)
```

Abbildung 3-13: Definition von cvLine

Die Funktion `cvLine()` zeichnet eine Linie in das übergebene Bild `img`, die beiden Punkte `pt1` und `pt2` stehen dabei für den Anfang und das Ende der Linie. Die Variablen `color`, `thickness`, `lineType` und `shift` verändern ihr Aussehen.

Da es in OpenCV keine explizite Funktion gibt, die Pfeile zeichnen könnte, müssen Pfeile aus Linien aufgebaut werden, dazu braucht man die Konstante pi als:

```
static const double pi = 3.14159265358979323846;
```

Abbildung 3-14: Definition von pi

Bildung der Pfeile funktioniert dann folgendermaßen:

```
angle2 = atan2( (double) b.y - d.y, (double) b.x - d.x );
cvLine( frame1, b, d, main_line_color, main_line_thickness, CV_AA, 0 );
b.x = (int) (d.x + 9 * cos(angle2 + pi / 4));
b.y = (int) (d.y + 9 * sin(angle2 + pi / 4));
cvLine( frame1, b, d, main_line_color, main_line_thickness, CV_AA, 0 );
b.x = (int) (d.x + 9 * cos(angle2 - pi / 4));
b.y = (int) (d.y + 9 * sin(angle2 - pi / 4));
cvLine( frame1, b, d, main_line_color, main_line_thickness, CV_AA, 0 );
```

Abbildung 3-15: Pfeil bilden aus Linien

3.4.2. Funktion cvCircle()

```
void cvCircle (
        CvArr * img,
        CvPoint Zentrum,
        int radius,
CvScalar color,
int thickness=1,
int lineType=8,
int shift=0
)
```

Abbildung 3-16: Definition von cvCircle

Diese Funktion zeichnet einen Kreis, ins Bild `img`, dabei ist der Punkt `Zentrum` der Mittelpunkt des Kreises und die Variable `radius` steht für den Radius. Die anderen Übergabeparameter steuern, analog zum `cvLine` das Aussehen der Linie, des Kreises.

Diese Methode verwendete ich um Features die von der `cvGoodFeatureToTrack()` Funktion gefunden wurden, zu markieren.

3.4.3. Funktion cvPutText()

```
void cvPutText(
CvArr* img,
const char* text, CvPoint
org,
const CvFont* font,
CvScalar color
)
```

Abbildung 3-17: Definition von cvPutText

Die Funktion `cvPutText()` schreibt einen beliebigen Text `text`, ins Bild `img`. Der Punkt `org`, definiert die Position an der, der untere, linke Rand des Textes beginnt, `cvPoint(0,0)` steht dabei für die linke obere Ecke des Bildes. Man kann die Farbe des Textes mit der Variable `color` einstellen, mit

```
CvScalar color = CV_RGB(0,255,0);
```

Abbildung 3-18: Beispiel zum CvScalar, für grüne Farbe

Diese Funktion bittet eine gute Möglichkeit unterschiedliche Ergebnisse auszugeben, so wie Zwischenzeiten des Algorithmus, oder X- und Y-Komponente des erfassten optischen Flusses. [9]

3.5. Methoden zur Erfassung des Optischen Flusses

Die OpenCV- Bibliothek Realisiert sowohl die Korrelationsbasierte Verfahren mit Block Matching, als auch Gradientenverfahren mit dem Horn & Schunck Algorithmus und dem Algorithmus von Lucas & Kanade, zur Erfassung des optischen Flusses. Insgesamt sind es vier Methoden, eine davon, `cvCalcOpticalFlowPyrLK()` wurde bereits in dem Beispielprogramm von Stanford University realisiert und im Abschnitt 3.2 vorgestellt. Weitere Methoden sind `cvCalcOpticalFlowBM()`, `cvCalcOpticalFlowLK()` und `cvCalcOpticalFlowHS()`, die in Folgendem näher beschrieben werden.

3.5.1. Funktion cvCalcOpticalFlowBM

Die Funktion `cvCalcOpticalFlowBM()` funktioniert nach dem Block-Matching Algorithmus und gehört somit zu den Korrelationsbasierten Verfahren der Flusserkennung.

```
void cvCalcOpticalFlowBM(
const CvArr* prev,
const CvArr* curr,
CvSize blockSize,
CvSize shiftSize,
CvSize max_range,
int usePrevious,
CvArr* velx,
CvArr* vely
)
```

Abbildung 3-19: Definition von cvCalcOpticalFlowBM

Die Variablen `prev` und `curr` sind zwei Bilder, an denen der Optische Fluss detektiert wird, `blockSize` ist die Größe der Blöcke, in die das Bild aufgeteilt wird. Variable `shiftSize` ist die Schrittweite, oder Häufigkeit, mit der die Blöcke im Bild verteilt werden, `max_range` gibt vor, wie weit um die Blöcke herum nach Ähnlichkeiten gesucht wird. Die Variable `usePrevious` sagt aus, ob vorher ermittelte Ergebnisse mitberücksichtigt werden sollen, dann sucht der Algorithmus zu erst in der Richtung, wo der letzte Vektor des Blocks hin gezeigt hat. Die 32-bit floating-point, single-channel Bildmatrizen `velx` und `vely` enthalten die Geschwindigkeitskomponenten, jeweils in X- und Y-Richtung, für jeden Block des Bildes.

```
velx = cvCreateImage(CvSize(120,160), IPL_DEPTH_32F, 1)
```

Abbildung 3-20: Beispiel der Allokation von velx

Für jeden Block gibt es also ein Pixel in `velx` und in `vely`. Die Bildung der Höhe und Breite von `velx` und `vely` funktioniert wie folgt:

$$Höhe_{VEL_{X/Y}} = \left[\frac{Höhe_{prev} - Höhe_{blockSize}}{Höhe_{shiftSize}} \right], \quad Breite_{VEL_{X/Y}} = \left[\frac{Breite_{prev} - Breite_{blockSize}}{Breite_{shiftSize}} \right]$$

Formel 3-2: Bildung der Größe, von velx und vely, bei Block Matching

3.5.2. Funktion cvCalcOpticalFlowLK

Die Funktion `cvCalcOpticalFlowLK()` funktioniert nach dem Algorithmus von Lucas & Kanade, gehört also zu den Gradientenverfahren.

```
void cvCalcOpticalFlowLK(
const CvArr* prev,
const CvArr* curr,
CvSize winSize,
CvArr* velx,
CvArr* vely
)
```

Formel 3-3: Definition von cvCalcOpticalFlowLK

Die Bilder `prev` und `curr`, sind das zu untersuchende Bildpaar. Die Variable `winSize` steht für die Größe der Blöcke, in den der Fluss als konstant angenommen wird. Die Parameter `velx` und `vely` sind ähnlich wie beim `cvCalcOpticalFlowBM()`, X- und Y-Komponente der Bewegung. Diese Bildmatrizen sind hier aber genauso groß wie die übergebenen Bilder, `prev` und `curr`, sind jedoch in jedem Fall 32-bit floating-point, single-channel Bilder. Denn auch wenn die Annahme herrscht, dass in einem bestimmten Block von der Größe `winSize` ein konstanter Fluss existiert, wird für jeden Pixel des Bildes ein eigener Vektor, in Form von seiner X- und Y-Komponente abgelegt.

3.5.3. Funktion cvCalcOpticalFlowHS

Die Funktion `cvCalcOpticalFlowHS()` funktioniert nach dem Algorithmus von Horn & Schunck und gehört neben der `cvCalcOpticalFlowLK()` zu den Gradientenverfahren.

```
void cvCalcOpticalFlowHS(
const CvArr* prev,
const CvArr* curr,
int usePrevious,
CvArr* velx,
CvArr* vely,
double lambda,
CvTermCriteria criteria
)
```

Abbildung 3-21: Definition von cvCalcOpticalFlowHS

Die Pointer `prev` und `curr` zeigen, wie bei den vorangegangenen Methoden auf die zwei übergebenen Bilder. Die Variable `usePervious` entscheidet ob die zuvor ermittelten Ergebnisse in die Rechnung miteinbezogen werden, oder nicht. `velx` und `vely` stehen auch hier für die in jedem Pixel ermittelte Geschwindigkeit, da der Algorithmus von Horn & Schunck wirklich für jeden Pixel ein Bewegungsvektor erstellt, müssen die Matrizen genauso groß sein wie die Bilder `prev` und `curr`. Die Variable `lambda` repräsentiert den, aus der Formel 2-25 bekanten Faktor des Rauschens.

$$e_{H\&S}(u,v) = e_d(u,v) + \boldsymbol{\lambda} \cdot e_s(u,v)$$

Formel 3-4: Rauschfaktor in der Formel von Horn & Schunck

Das `criteria` legt fest wann die Suche terminiert, es ist eine Struktur, bestehend aus zwei Variablen, die erste besagt wie viele Widerholungen gemacht werden, so lange es keine ausreichende Ähnlichkeit gibt, die andere ist ein Faktor der Ähnlichkeit, besagt also wie genau die Übereinstimmung sein muss, um sagen zu können das die Pixelwanderung gefunden wurde.

4. Test

4.1. Aufbau der Testumgebung

Die Testumgebung ist ein wenig modifizierter, schon aus dem Physiklabor bekannter Versuchsaufbau, mit dem sich Beschleunigungen und die Erdanziehungskraft messen lassen. An vorderster Stelle stand die Anforderung, die Tests möglichst reproduzierbar zu machen. Der Aufbau bittet eine sehr gute Möglichkeit eine identische Bewegung vom test zu test zu gewährleisten. Der Versuchsaufbau ist in Abbildung 4-1 dargestellt.

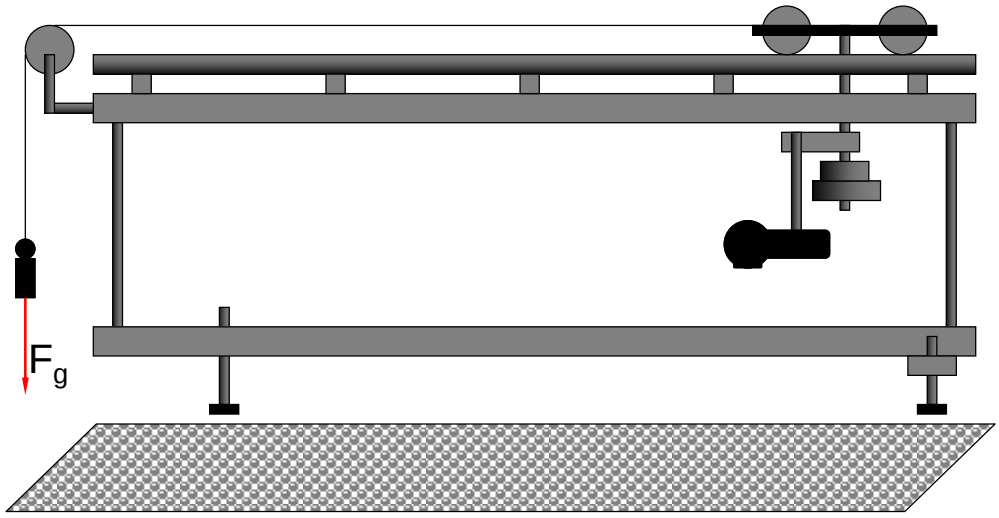


Abbildung 4-1: Skizze des Versuchaufbaus

Eine herkömmliche Digitalkamera, befestigt an den Beweglichen Wagen des Aufbaus, wird über eigens dafür angefertigte texturreiche Untergründe gefahren. Die Texturen sind sehr Symmetrisch, was ihre spätere Reproduktion leichter macht. Relativ viele Gewichte am Wagen sorgen dafür dass die Kamera möglichst stabil gelagert ist und die Bewegung nicht so anfällig ist für kleine Störfaktoren. Der Wagen ist über einen Faden mit einem Gewicht, am Ende der Testbahn befestigt und wird von der auf dieses Gewicht wirkenden Erdanziehungskraft Beschleunigt. Das beschleunigende Gewicht ist sehr klein gehalten, damit die Geschwindigkeit nicht übertrieben groß wird und die Bewegung längere Zeit andauert um möglichst viele Messwerte zu bekommen. So kriegen wir eine weitestgehend konstante Beschleunigung, die einen anschaulichen Vergleich der, von den Algorithmen gelieferten Ergebnisse erlaubt.

4.2. Testdurchführung

Für die Testdurchführung bieten sich zwei unterschiedliche Strategien an, die unterschiedliche Auswertungsmöglichkeiten anbieten. Zu einem das normale Durchfahrenlassen des Wagens. Wobei die eigentliche Bewegungserfassung, anhand von Mustern des Untergrunds erfolgt. Bei

dieser Testart lässt es sich sehr gut bestimmen, wie genau der eine oder der andere Algorithmus, beim Erfassen der Bewegung, während unterschiedlicher Einstellungen und Bedienungen ist.

Die andere Testmethode ist, den Wagen einfach in Ruhestellung zu belassen. Dabei filmt die Kamera den unbewegten Untergrund. So erfasst man ausschließlich das, von den Frames verursachte Rauschen und kann bequem versuchen, dieses durch variieren der Übergabeparameter, zu verringern. Auch bittet diese Testmethode eine aussagekräftige Ansicht der Rauschanfälligkeit, der Algorithmen im vergleich.

Zur Genauigkeitsuntersuchung der einzelnen Algorithmen variiere ich die Untergründe, dabei verwende ich unterschiedliche Muster. Zu einem Rechtecke und Kreise und für Aperturuntersuchungen schiefe und gerade Linien. Die Quadrate haben eine Kantenlänge von 2 cm, auch der Durchmesser der Kreise beträgt 2cm, sowie ihre Abstände von einander. Die Linienmuster sind einmal um 90°und einmal um 45° verdreht zur Bewegungsrichtung und enthalten genau gleich viele weiße, wie schwarze Fläche (). Damit lässt sich feststellen wie genau die Algorithmen die Richtung der Bewegung bestimmen können.

Die Testergebnisse werden in einem TXT-File stringweise abgelegt. Zu Beginnverfolgte ich die Strategie jedes, der Verfahren einzeln abzurufen und zu testen, was sich später allerdings als viel geschickter erwiesen hat war es, beim jedem Loop gleich alle Algorithmen abzurufen und ihre Werte abzulegen, dabei sind die Ergebnisse besser vergleichbar, als wenn man für jeden einzelnen Algorithmus, jedes Mal den Wagen fahren lässt.

Einige der Tests verlangen es, Übergabeparameter zu variieren, zum Beispiel die Untersuchung des winSize. Die Abbildung 4-2 zeigt Die dafür verwendete Routine. Beim Drücken des Buchstaben „c“ wird das `RecordFlag` gesetzt, ab dem nächsten Loop werden alle erforderlichen Variablen zur Variierung der Parameter und der Ablage der Ergebnisse in das TXT-File gesetzt. In den nachfolgenden Loops werden die Werte in das File geschrieben, eventuell wird das Video Frame per Frame Zusammengesetzt. Dabei werden auch die Übergabeparameter, mit Hilfe von der Variable `param`, sukzessive erhöht oder erniedrigt.

```
if(lk_F<1 && RecordFlag == 1){
        lk_F++;
}else if ( RecordFlag == 1){
        lk_F= 0;
        lk_S = lk_S +2 ;
        param++;
}
if (c == 'r')
        RecordFlag = 1;

if (RecordFlag == 1){
        i++;

        cvConvertImage(Two_F2.frame1,RecordFrame);
        cvWriteFrame(VW,RecordFrame);
        fprintf(datei,"%i;%i;%f;%i;%i;%f\n",
i, clock()-start2, LaengePtr(ptr), LaengePtrX(ptr),
LaengePtrY(ptr), param);
                }

c=cvWaitKey(10);

//LOOP
```

Abbildung 4-2: Routine zur Variierung der Übergabeparameter und Steuerung der Tests

4.3. Probleme beim Testen

Abbildung 4-3: Versuchsaufbau

Diverse Probleme erschwerten das genaue Durchführen der Tests, zu einem das Kabel der Kamera, was eine sauber beschleunigte Bewegung zu realisieren verhinderte, zu anderem die Lichtverhältnisse die stets Variiert haben und die Kamera dazu brachten, mal länger und mal kürzer zu belichten, dabei entstanden Bilder mit einem unterschiedlichen Zeitabstand und dementsprechend anderen Geschwindigkeitsgefühl.

Die Übergänge Zwischen den Blättern die, die Textur enthalten, machten auch Probleme, bei den Gradientenverfahren spielt die Anzahl der sich bewegenden Pixel eine große Rolle, bei den Übergängen gab es ein Texturverlust, also weniger bewegte Fläche, was zu einem Längenabfall führt den man in folgenden Diagrammen als Wellen wahrnehmen kann. Dabei war genau das die einzige Möglichkeit die Geschwindigkeit zu berechnen, davon ausgegangen das der Abstand zwischen den Wellen genau eine DIN-A4 Blattlänge ist, also 29,7 cm. Die Abbildung 4-4 zeigt Den grafischen Ansatz für die Berechnung der Geschwindigkeit. Dafür lautet die Berechnung dann folgendermaßen:

$$8323 Pixel \leftrightarrow \frac{29{,}7 cm}{2 \sec} = 14{,}85 \, cm/\sec \approx 0{,}1485 \, m/\sec \approx 0{,}5346 \, km/h$$

Formel 4-1: Zuordnung einer Geschwindigkeit zu einer Länge in Pixel

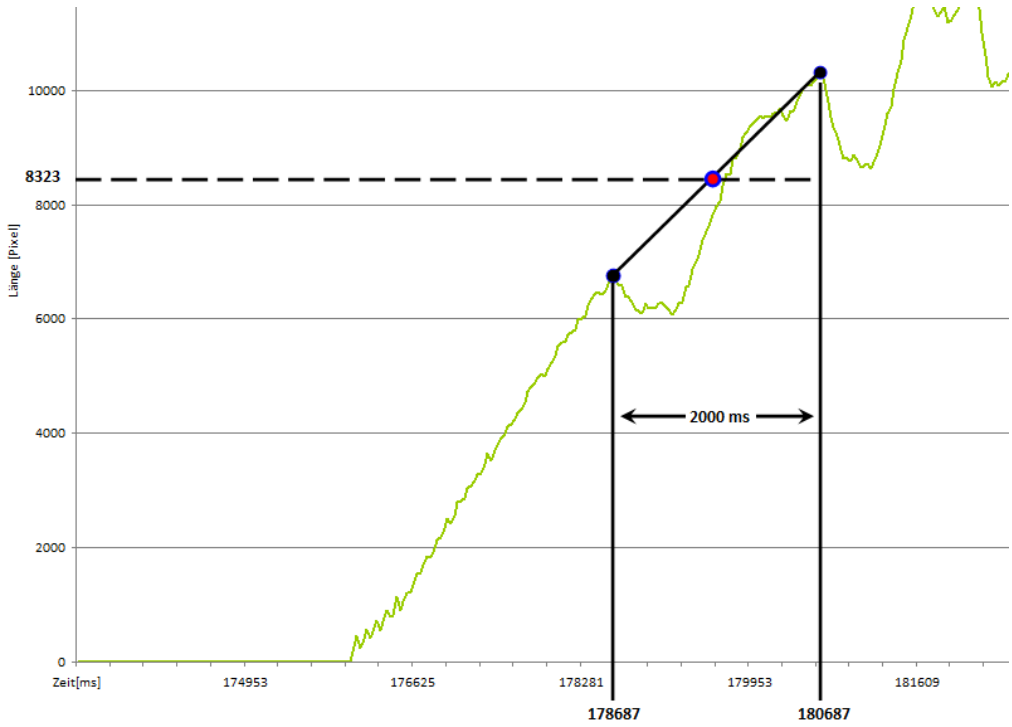


Abbildung 4-4: Bestimmung der Geschwindigkeit

Es war auch nicht immer Möglich Videos von der Testdurchführung aufzunehmen, da die Funktion `cvWriteFrame()` von Zeit zurzeit Speicherzugriffsfehler verursacht hat.

4.4. Ergebnisse der Tests

Während der Tests untersuchte ich die einzelnen Algorithmen auf ihre Tauglichkeit für die geforderte Aufgabe, den Quadrokopter zu stabilisieren untersucht. Jeder in OpenCV enthaltene Algorithmus zur Flusserkennung, besitzt eigene Eigenschaften in Form von Variablen, deren Einstellungen die Qualität des gelieferten Ergebnisses in unterschiedlicher Art und Weise beinträchtigen können. Diese Einflüsse werden im Ersten Teil der Tests untersucht, um gegebenenfalls eine optimale Einstellung der Algorithmen Vorschlagen zu können.

4.4.1. Parameter der Funktion cvCalcOpticalFlowPyrLK

Die Funktion `cvCalcOpticalFlowPyrLK()` besitzt folgende Parameter, die auf das Ergebnis Einfluss nehmen könnten:

- `int count,`
- `Size winSize,`
- `int level,`
- `MCvTermCriteria criteria`

Die Variable `count` steht für die Anzahl, der gesuchten Features, somit auch für die voraussichtliche Anzahl der gefundenen Bewegungsvektoren, deren Durchschnitt den endgültigen Bewegungsvektor definiert. Sind viele Features eingestellt so kann man eine korrekte Bewegungsschätzung, mit einer größeren Wahrscheinlichkeit erwarten. Ist es aber auch die Regel, dass die Anzahl der Ausreißer genauso zunimmt. Nach einigen Tests kann man sagen, dass 10 Features kein schlechteres, aber auch nicht ein besseres Ergebnis liefern, als 400, somit schließt sich `count`, als ein potenzieller Wert für die Genauigkeit der Schätzung aus. Man kann lediglich von einer Statistischen Verbesserung ausgehen, die sich allerdings in der realen Welt nicht bestätigt.

Die Variierung von `winSize`, `epsilon`[1] und `level` lieferten sehr ähnliche Ergebnisse (siehe Abbildung 4-5, Abbildung 4-6, Abbildung 4-7). Es zeigt sich, dass beim Erhöhen jedes dieser Werte das Rauschen zwar enger wird, dass es aber nicht wie erwartet sich zum Nullpunkt bewegt, sondern einen Bestimmten wert anstrebt, der sich im Bereich von 75 Pixel bewegt (siehe Abbildung 4-8). Es gibt also ein bestimmtes Grundrauschen, das sich mit keiner dieser Variablen in Griff kriegen lässt. Somit zeigt sich auch die Untauglichkeit der Methode `cvCalcOpticalFlowPyrLK()`, beim Regeln eines Systems, bei Kleinen werten.

[1] Die Variable `epsilon` ist die Zweite, aus den zwei Bestandteilen des Terminationcriteria *`criteria`*, sie steht für die Genauigkeit der Übereinstimmung, der gefundenen Features, also um so größer das *`epsilon`*, desto ähnlicher müssen sich die Features sein, um als gefunden zu gelten.

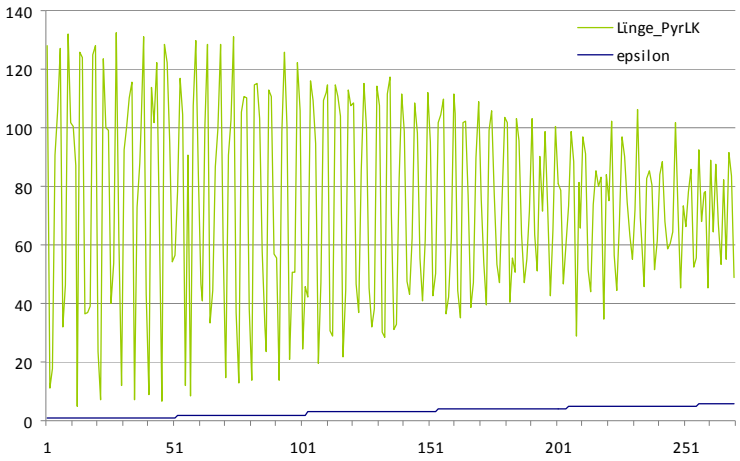


Abbildung 4-5: Variierung von epsilon beim Lucas & Kanade Pyramiden- und Feature-Algorithmus

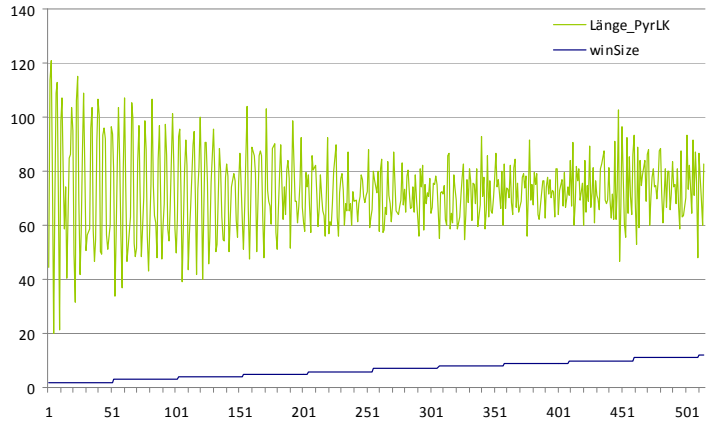


Abbildung 4-6: Variierung von winSize beim Lucas & Kanade Pyramiden- und Feature-Algorithmus

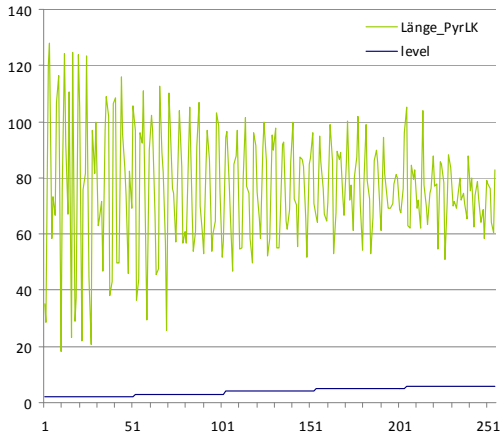


Abbildung 4-7: Variierung von level beim Lucas & Kanade Pyramiden- und Feature-Algorithmus

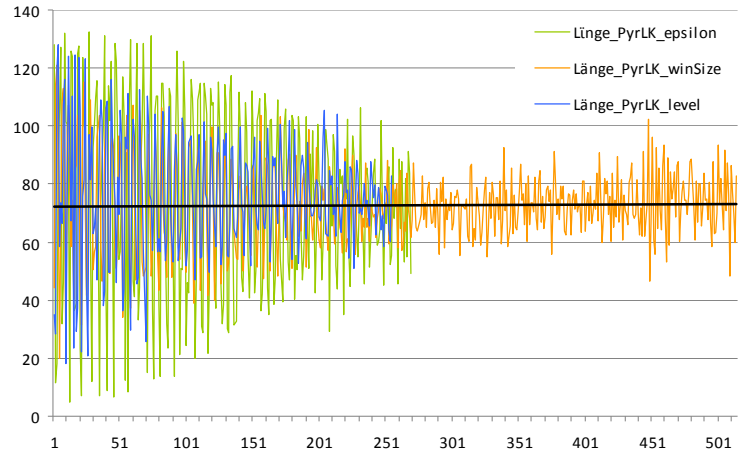


Abbildung 4-8: Überlappung der Kurven aus dem Test vom Lucas & Kanader Algorithmus mit Featuresuche und Gaußpyramiden, und ihre Trendlinie

4.4.2. Parameter der Funktion cvCalcOpticalFlowBM

Beim `cvCalcOpticalFlowBM()` hat man folgende Möglichkeiten um aufs Ergebnis einzuwirken:

- `CvSize blockSize,`
- `CvSize shiftSize,`
- `CvSize max_range`

Die Abbildung 4-9 zeigt den, experimentell ermittelten Zusammenhang zwischen dem Rauschen und dem `blockSize`. Es wird deutlich, dass der Block Matching Algorithmus erst bei Werten über 13 – 14 Pixel, Ergebnisse mit annehmbarem Rauschen liefert. Das Ergebnis ist wohl so zu erklären, dass wenn die Größe der Blöcke anwächst, werden automatisch die Muster eines solchen Blockes, entsprechend komplizierter und damit eindeutiger, flackernde Pixel erkennt der Algorithmus nicht mehr als Bewegungen des gesamten Blocks.

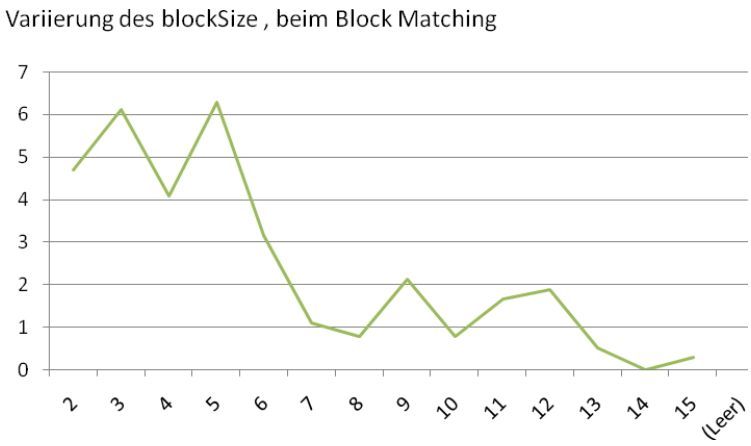


Abbildung 4-9: Zusammenhang zwischen blockSize und Rauschen, beim Block Matching

Ähnlich wie beim `blockSize`, verhält sich das Rauschen, wenn man das `shiftSize` variiert. Eine deutliche Verbesserung stellt sich bereits bei Werten über 10 Pixel ein. Das Testergebnis wird in der Abbildung 4-10 gezeigt.

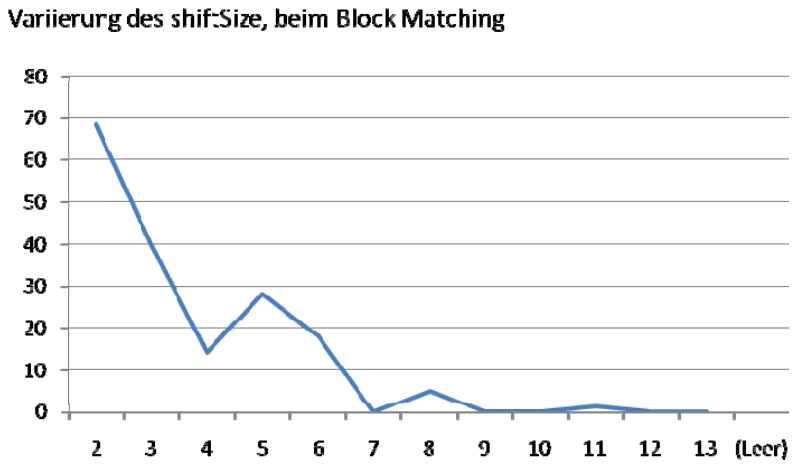


Abbildung 4-10: Zusammenhang zwischen shiftSize und Rauschen, beim Block Matching

Anders wie bei den Parametern `blockSize` und `shiftSize`, verhält sich die Rauschanfälligkeit, beim Variieren des `maxRange`. Die Abbildung 4-11 zeigt das Ergebnis des Tests, es wird deutlich, dass das Rauschen stetig zunimmt, beim zunehmenden `maxRange`.

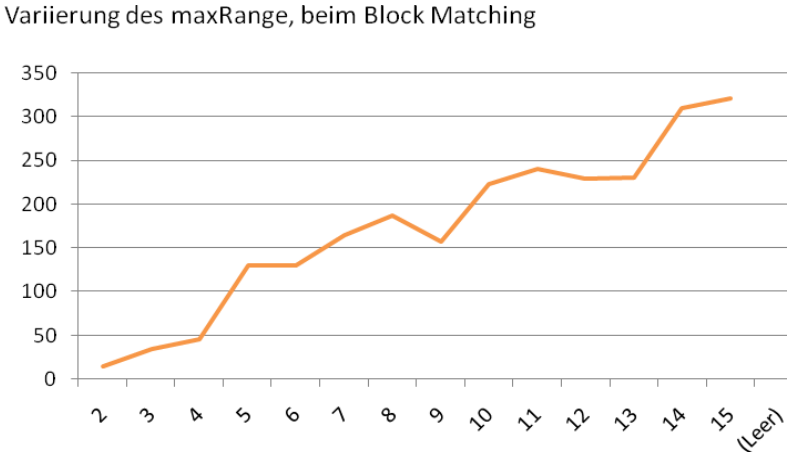


Abbildung 4-11: Zusammenhang zwischen maxRange und Rauschen, beim Block Matching

Die Tests, des Block Matching Algorithmuses zeigen, dass die optimale Einstellung dafür die folgende sein dürfte:

- `blockSize = cvSize(14,14);`

- `shiftSize = cvSize(10,10);`
- `max_range = cvSize(2,2);`

4.4.3. Parameter der Funktion cvCalcOpticalFlowLK

Die einzige Möglichkeit bei `cvCalcOpticalFlowLK()` den Algorithmus zu beeinflussen ist das:

- `CvSize winSize`

Zur Erinnerung, der Algorithmus von Lucas & Kanade setzt voraus das der optische Fluss in festen Blöcken, einen gleichen Fluss hat, und nicht wie beispielsweise, beim Horn & Schunck, für jeden Pixel berechnet wird. Das `winSize` steht genau für diese Blockgröße, wobei es ganz bestimmte Werte gibt, die man für die Kantenlänge der Blöcke einsetzen kann, es sind die 1, 3, 5, 7, 9, 11 und die 13, andere Werte geben einen Fehler aus.

Gleich zu Beginn Zeigte sich die relativ hohe Rauschanfälligkeit des Algorithmus, was mit nachfolgender Abbildung verdeutlicht wird.

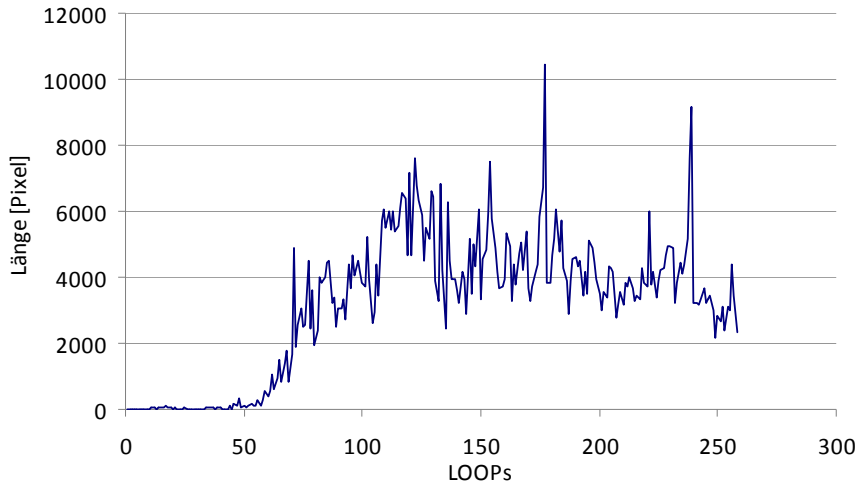


Abbildung 4-12: Pfeilänge beim Algorithmus von Lucas und Kanade, Bewegt durch rechteckiges Muster

beim Testen des `winSize` empfahl sich die Methode, stehende Messungen durchzuführen, also nicht die Kamera fahren zu lassen sondern in Ruhestellung das Rauschen aufzunähmen und zu analysieren. Folgende Kurve zeigt den dabei ermittelten Einfluss der Blockgröße, auf den Zeigerausschlag. Man schließt daraus, dass die optimale Einstellung für diese Funktion, das `winSize` = 13 ist.

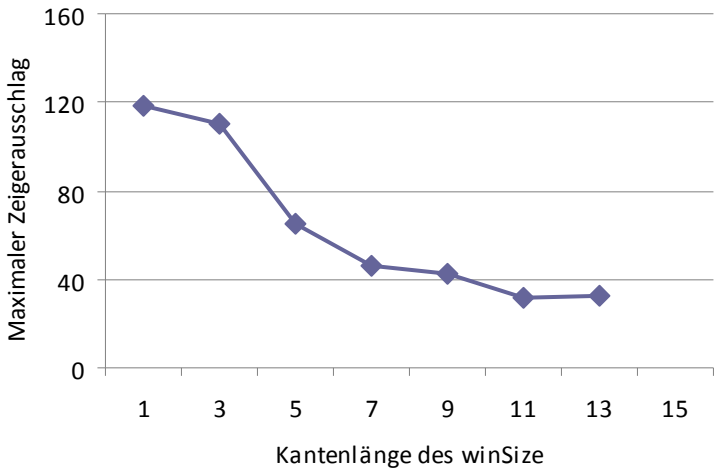


Abbildung 4-13: Verhältnis zwischen winSize und Rauschanfälligkeit, beim Lucas & Kanade Algorithmus

4.4.4. Parameter der Funktion cvCalcOpticalFlowHS

Die Methode `cvCalcOpticalFlowHS()` hat folgende Einstellmöglichkeiten:

- `int usePrevious`
- `double lambda,`
- `CvTermCriteria criteria`

Wie Oben bereits erwähnt ist `lamda` der Rauschfaktor, es bestätigt sich auch im Test, um so größer das `lamda` gemacht wird, desto unzuverlässiger wird auch der Ausschlag des Pfeils. Beim Test nach der Abhängigkeit von `lamda` werden die Werte von 1 bis 0,001 variiert, Das Ergebnis ist in Abbildung 4-14 dargestellt.

Dabei Steht die Y-Achse für die Länge des errechneten Pfeils, in Pixel und die X-Achse Zeigt die während dessen verstrichenen Loops. Was einem sofort ins Auge sticht ist, dass die Letzten zwei Kurven deutlich glätteren Verlauf haben als die anderen. Die Letzte Kurve, die einen Wert von `lamda`, von 0,001 entspricht, zeigt einen Kurvenverlauf der kaum realistisch zu sein scheint. Es besitzt kaum Rauscheffekte und sprungartige Bewegungen des Pfeils. Was auch auffällig ist, ist dass die letzte Kurve rund ein Drittel kürzere Maximallänge des Pfeils ausgibt, was einen zu dem Entschluss kommen lässt, dass das `lamda`, wenn es zu klein gewählt ist, Dämpfungseffekte besitzt, was auch durch das Abklingen der Kurve bestätigt wird.

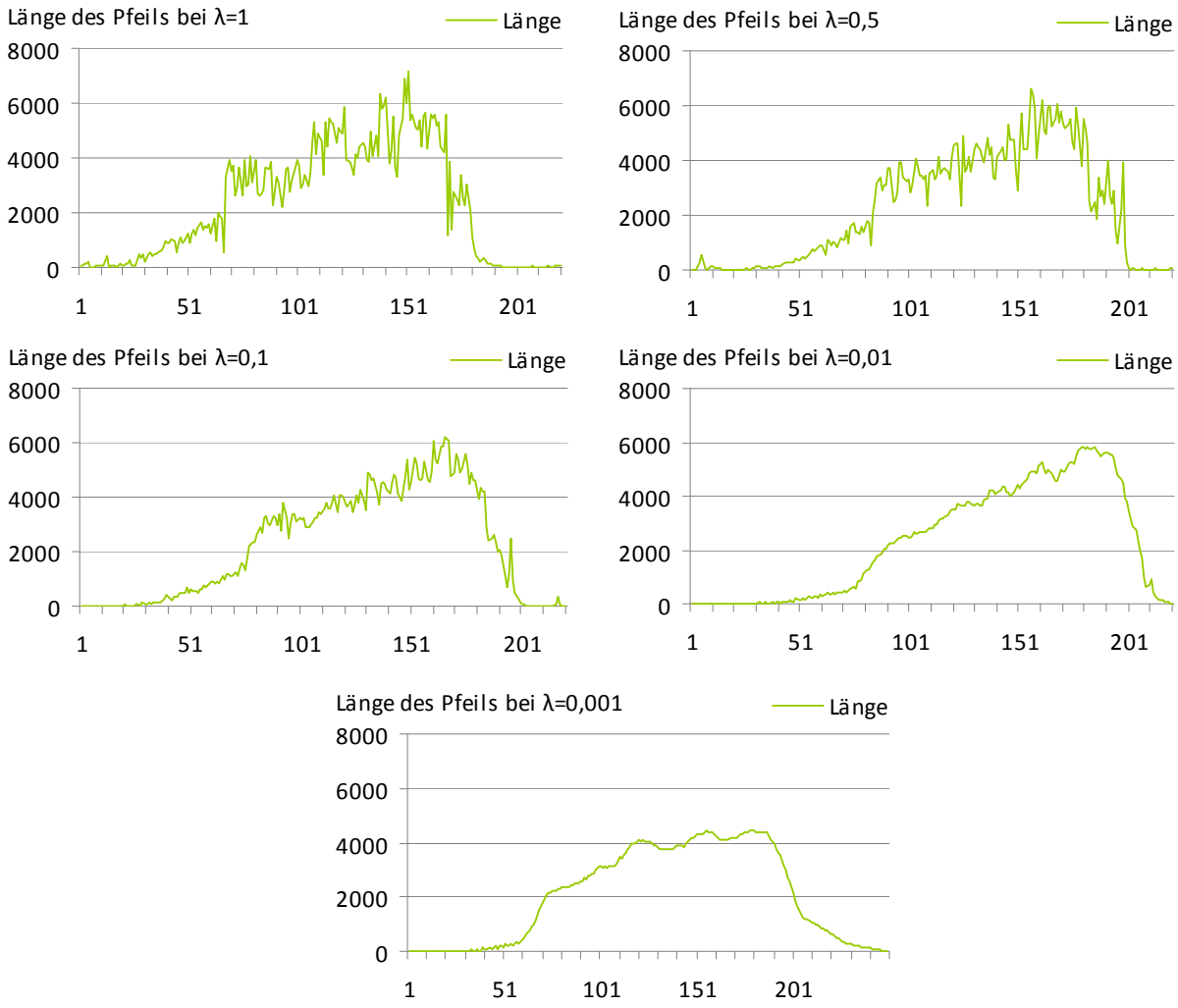


Abbildung 4-14: Variierung von lamda, im Algorithmus von Horn & Schunck

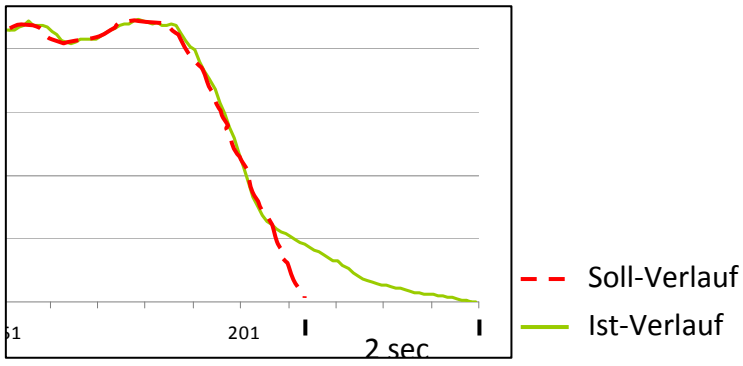


Abbildung 4-15: Abweichung des Kurvenverlaufs von der Wirklichkeit

Bei einem lamda von 0,001 klingt die Kurve nicht ab, wie man es erwarten würde, nämlich so wie der Wagen sich bewegt, sondern nur ganz langsam. Es Entsteht ein Nachschwingen von

genau 2 Sekunden (siehe Abbildung 4-15), was für eine Regelung des Flugapparats große Schwierigkeiten bereiten würde.

In Anbetracht dieses Ergebnisses kommt man zu dem Entschluss, dass es die beste Variante wäre den Algorithmus, mit einem lamda = 0,01 zu betreiben. Auch wenn, bei dieser Einstellung keine so glatte Kurve heraus kam, ist das Ergebnis trotzdem genau genug um damit weiter arbeiten zu können. Eine Länge des Pfeils von 6000 entspricht in dem Fall einer Geschwindigkeit von 0,106 m/s, das Rauschen was dabei entsteht, bewegt sich in Bereich von 20 – 40 Pixel, also 0,0005 m/s, verschwindend klein.

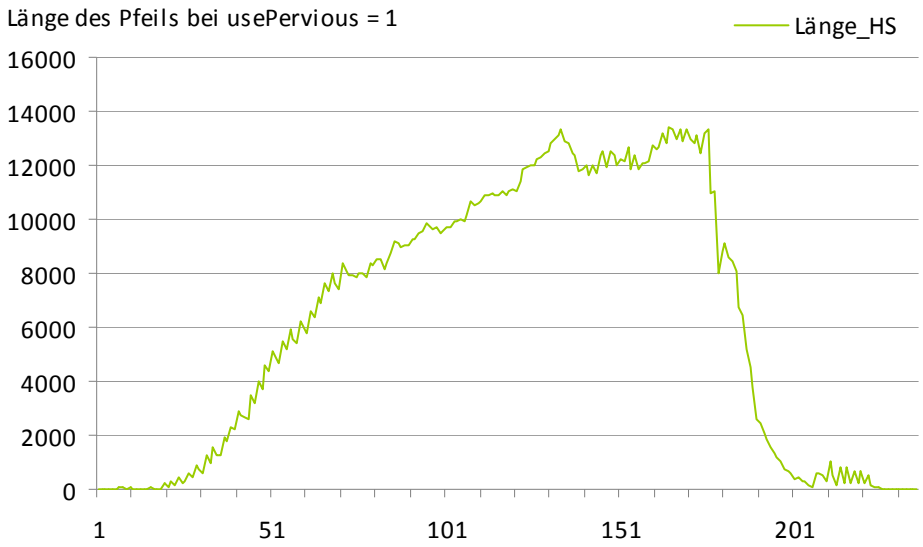


Abbildung 4-16: Horn & Schunck Algorithmus mit usePervious

Das `usePervious` gibt an ob die Ergebnisse von vorigen Berechnungen ins aktuelle Ergebnis mit einfließen oder nicht, nur wenn diese Variable einen Wahrheitswert liefert, also großer als Null ist, kann das oben Beschriebene Problem des Nachschwingens entstehen. Da wenn die Werte aus den vorigen Berechnungen nicht betrachtet werden, wird der Fluss wie beim Lucs & Kanade Algorithmus, rein nur aus den zwei aktuellen Bildern geschätzt. So liegt die Überlegung nahe, dass man einen, besser der Realität entsprechendes Ergebnis kriegt, wenn man auf die Funktion des `usePrvious` verzichtet. So bin ich dieser Überlegung nachgegangen, das Test Ergebnis sieht allerdings, wie in der Abbildung 4-16 und Abbildung 4-17 dargestellt, sehr ernüchternd aus.

Vergleicht man die beiden Diagramme in Abbildung 4-16 und Abbildung 4-17, merkt man sofort, dass der Algorithmus, ohne die vorhergegangenen Ergebnisse mit einzubeziehen, zwar auch auf ruckartige Bewegungen entsprechend reagiert, doch sind die Werte so vom Rauschen belastet, dass keine sinnvolle Regelung damit realisierbar wäre.

Was dabei aber auch noch interessant sein könnte, ist der Vergleich mit dem Lucas & Kanade Algorithmus. Die Kurve in Abbildung 4-18 zeigt, die während des usePervious – Tests

aufgenommene Werte des Lucas & Kanade Algorithmuses, was als erstes auffällt, ist der Dimensionsunterschied, die Maximalwerte von Lucas & Kanade sind rund um das Zehnfache größer als die, die von Horn & Schunk Algorithmus geliefert werden. Dabei ist das Rauschen des Lucas & Kanade Algorithmuses deutlich stärker, es eignet sich also noch weniger für die Aufgabe der Regelung.

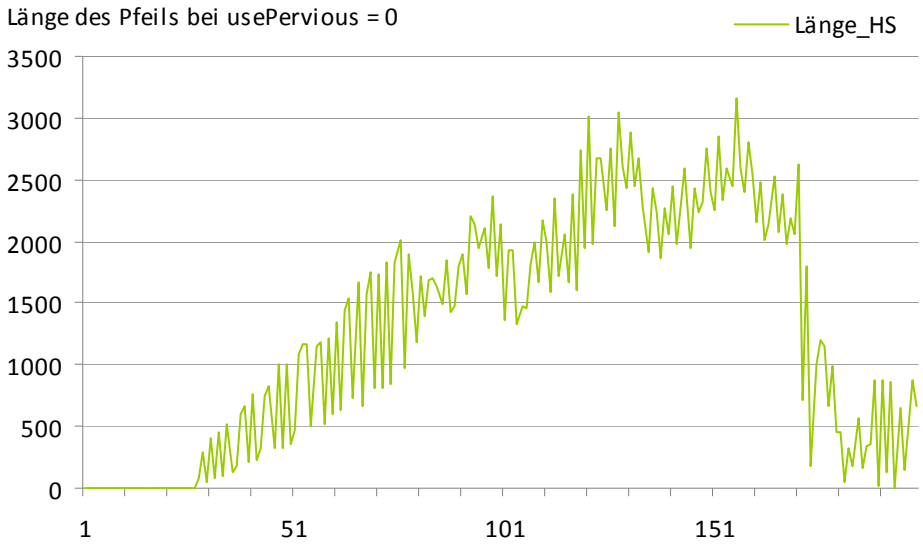


Abbildung 4-17: Horn & Schunck Algorithmus ohne usePervious

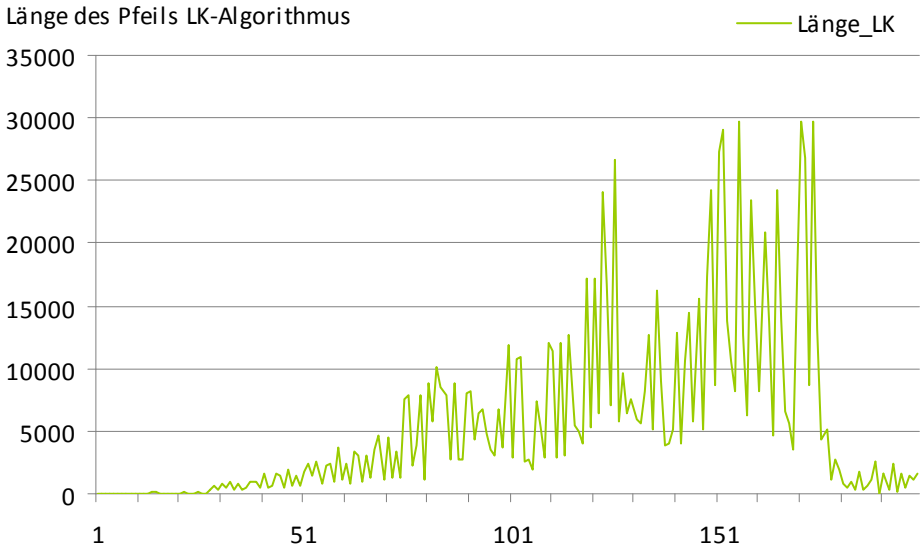


Abbildung 4-18: Lucas & Kanade Algorithmus zum Vergleich mit Horn & Schunck

4.5. Qualität der Bilder

Die Größe der Verwendeten Bilder spielt eine große Rolle, bei der Erkennung des optischen Flusses. Es ist aber auch logisch, dass wenn sich die Größe der Bilder ändert, dass sich auch ihre Qualität. Es gibt zwei Einstellungen fürs Qualitätsmaß der Bilder, zu Einem die JPG-Qualität und zum Anderem die PNG-Qualität. Wehrend die JPG-Qualität Bilder liefert die etwa 1KB groß sind, liefert die PNG-Qualität etwa das Achtfache, also 8 KB pro Frame.

Bei dem Test der Reaktion der Algorithmen auf die Qualitätsänderung wurde die Teststrecke mit Kreismustern abgefahren und die Ergebnisse mit einander verglichen. Die Abbildung 4-19 und Abbildung 4-20 zeigen die dabei entstandenen Ergebnisse. Als erstes fällt auf, dass die Ausschläge bei PNG-Qualität, beim Lucas & Kanade und Horn & Schunck Algorithmus fast um das sechsfache größer sind als bei JPG-Qualität. Dies kommt davon, da die Länge der ermittelten Pfeile beim Testen einfach zusammen addiert wird und Bilder der PNG-Qualität eine höhere Auflösung besitzen, also eine und die selbe Bewegung schließt mehr Pixel ein, zu den ein Vektor berechnen und zum Gesamtergebnis addiert wird. Damit ist auch zu erklären, dass der Block Matching Algorithmus (siehe Abbildung 4-21), längenmäßig betrachtet, ein sehr ähnliches Ergebnis liefert, da es in beiden Fällen die gleiche Anzahl an Blöcken verwendet. Fraglich ist das Ergebnis von dem Lucas & Kanade mit den Features und Gaußpyramiden (siehe Abbildung 4-22), der Längenunterschied der beiden Kurven sticht sofort ins Auge. Dabei ist dieselbe Featureanzahl eingestellt, und wenn es danach ginge, sollten die Längen der beiden Durchläufe ähnlich sein. Eine Vermutung ist, dass sehr viele Features nicht gefunden werden bei der JPG-Qualität. Es ist aber auch eindeutig zu sehen, dass die Frames mit der schlechteren Qualität Rauschbelasteter sind, als die mit PNG-Qualität.

Das Ergebnis dieses Tests war entsprechend auch erwartet worden, die Bilder der JPG-Qualität weisen viel weniger Anhaltspunkte auf an denen der optische Fluss gemessen werden kann, dazu stört das permanente Flackern einzelner Pixel und schlägt sich, als Rauschen im Ergebnis nieder.

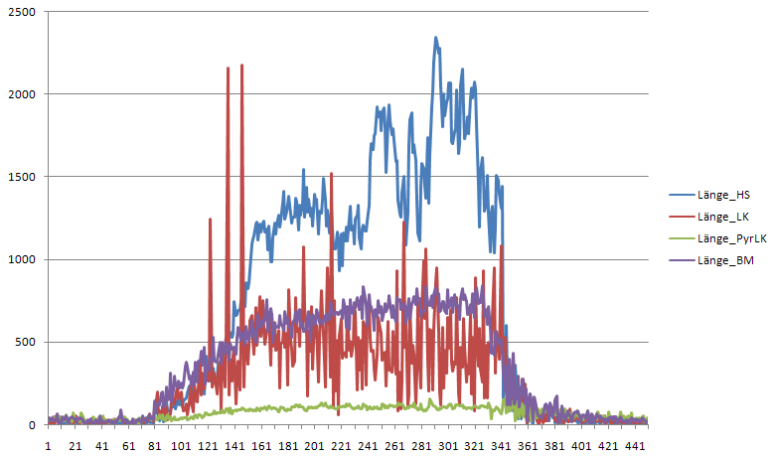


Abbildung 4-19: OpenCV-Algorithmen mit JPG-Qualität

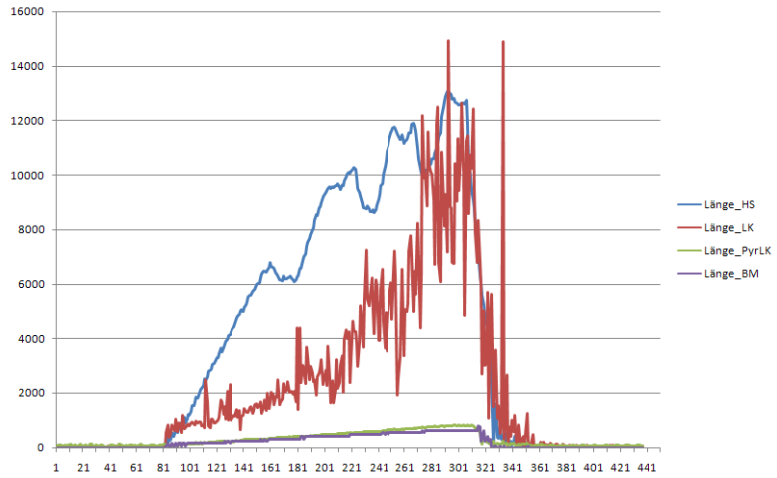


Abbildung 4-20: OpenCV-Algorithmen mit PNG-Qualität

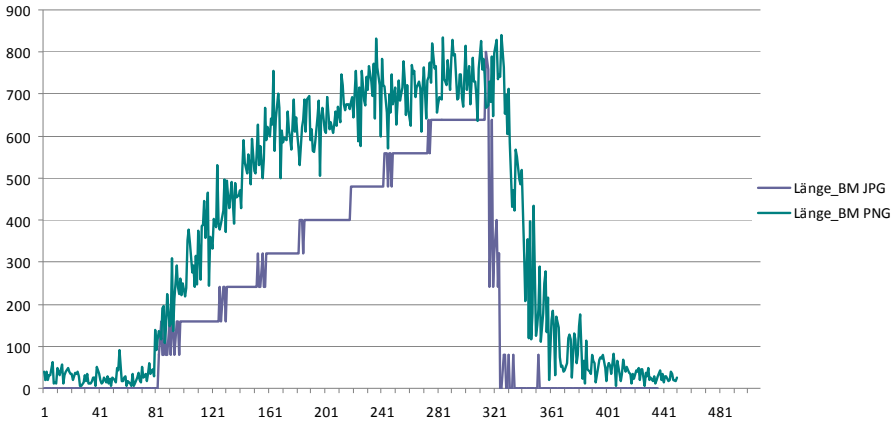


Abbildung 4-21: Vergleich JPG- und PNG-Qualität beim Block Matching

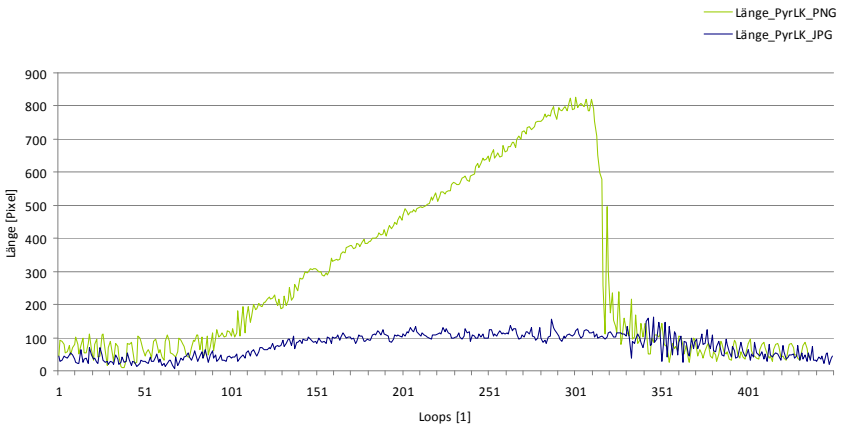


Abbildung 4-22: Vergleich JPG- und PNG-Qualität beim Algorithmus von Lucas & Kanade mit Featuresuche und Gauspyramiden

4.6. Texturen der Untergründe

Die Untersuchung der Reaktion der Algorithmen, auf die unterschiedlichen Untergründe soll uns mehr Aufschluss darüber geben, was das Ausschlaggebende ist, in der Untergrund Textur und wie es das Ergebnis beeinflusst.

In Abbildung 4-23 und Abbildung 4-24 sind Kurven dargestellt, es sind Längenmasse der X- und Y-Komponenten der einzelnen Algorithmen, jeweils beim Untergrund Textur aus Kreisen und Rechtecken. Es wird eine Bewegung in X-Richtung durchgeführt, so ist zu erwarten, dass im Idealfall die Algorithmen keinen Ausschlag bei der Y-Komponente aufweisen. Das Rausche oder Flackern der Pixel stört allerdings die Bewegungserkennung. Was dazu führt, dass doch eine Y-Komponente zu sehen ist, das Aperturproblem ist in diesen Fällen auszuschließen, da es in dem Fall eindeutige Figuren gibt die verfolgt werden Können, unabhängig davon, ob die Texturträger gerade zur Apparatur des Tests liegen oder nicht. Es kann aber auch daran liegen, dass die Kamera nicht exakt Rechtwinklig zur Bewegung ausgerichtet werden konnte. Dieser enfluss ist aber eher minimal.

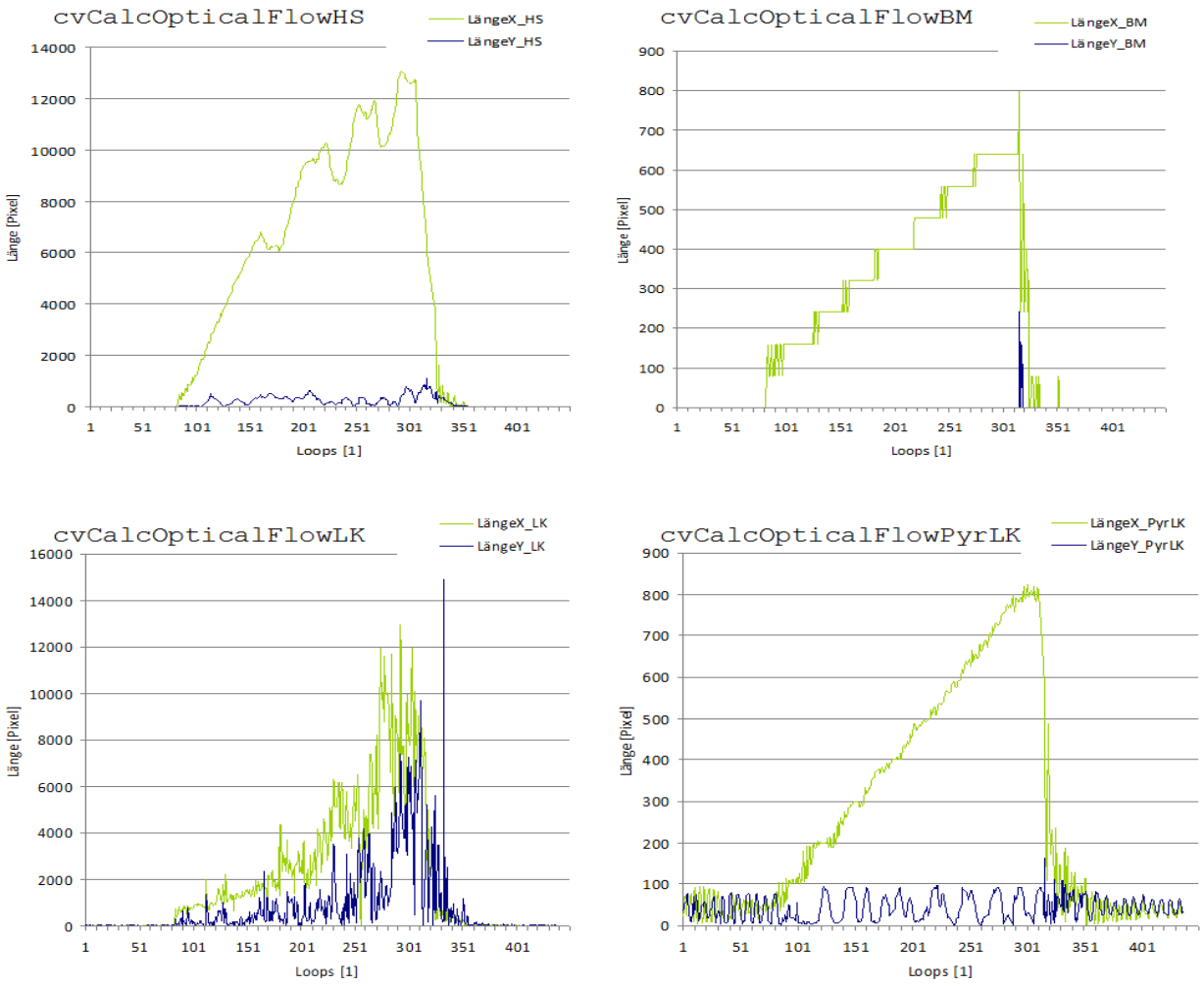


Abbildung 4-23: Ergebnisse des Tests der Untergründe (Kreise)

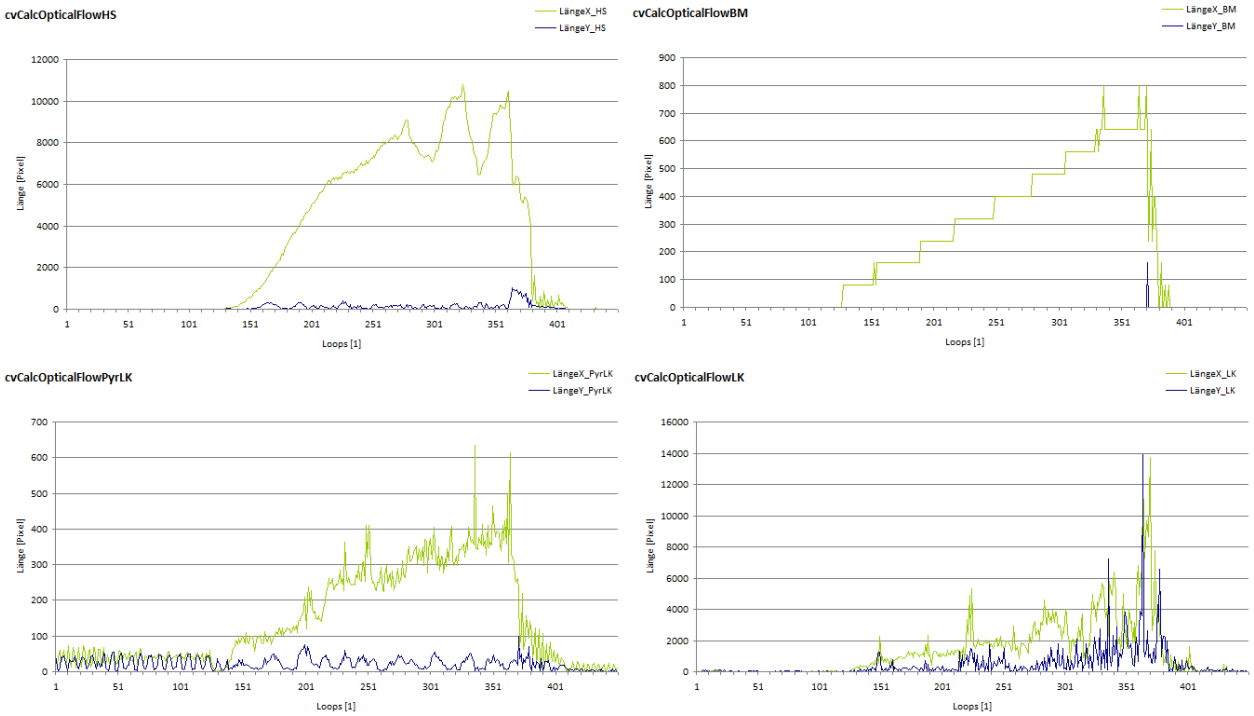


Abbildung 4-24: Ergebnisse des Tests der Untergründe (Rechtecke)

Das zu erwartende Ergebnis des Lukas & Kanade Algorithmuses mit Featuresuche wäre, dass die Rechteckigen Muster leichter zu verfolgen sind da die Features nach Ecken suchen die im Kreismuster gar nicht vorhanden sind, was aber der Test widerlegt. Es liegt aber daran, dass die Kreise mit einer kleinen Auflösung dargestellt werden wie Figuren die aus vielen Quadraten bestehen und bieten damit dem Algorithmus viel mehr Ecken die verfolgt werden können.

Der Algorithmus von Horn & Schunk und Lucas & Kanade liefern in beiden Fällen ein sehr ähnliches Ergebnis, lediglich die Maximalausschläge unterscheiden sich um c.a. 10 %. Dieses Phänomen kann mit der unterschiedlichen Flächenbewegung erklärt werden, die Algorithmen detektieren Bewegungen jedes Pixels, umso mehr Fläche sich bewegt, umso mehr Pixel sind von ihr eingeschlossen.

Der Algorithmus von Lucas & Kanade liefert bei diesem Test das Schlechteste Ergebnis, man kann hier nicht eindeutig sagen ob sich der Wagen in die X- oder die Y-Richtung bewegt, da die Ausschläge beider Kurven fast gleich sind. Dazu ist das Ergebnis sehr verrauscht, was nicht so sehr an den Frames liegt, was uns der Algorithmus von Horn & Schunck beweist.

Der Block Matching Algorithmus liefert dabei, für seine Verhältnisse gutes Ergebnis, die Pfeillängen wachsen und fallen zwar immer in Stufen von 80 Pixel, zeigt dieser Algorithmus aber als Einziger keinen Ausschlag bei der Y-Komponente, außer am Ende der Bewegung, was man mit dem Aufschlag erklären kann den der Wagen macht am ende der Strecke. Der Algorithmus zeigt auch eine sehr gute Geschwindigkeitserkennung, da die Maximalausschläge in beiden Fällen sehr ähnlich sind.

4.7. Aperturtest

Mit dem Aperturtest wird klar welcher Algorithmus am besten auf das Aperturproblem reagiert. Dazu werden strecken mit geraden und schiefen Linien abgefahren, wobei ganz klar ist, dass die schiefen Linien unweigerlich zum Aperturproblem führen, was Aufschluss darüber machen lässt welcher der Algorithmen am besten die Bewegungsrichtung identifizieren kann. Im Idealfall müssen die Algorithmen dabei, von der Länge identische X- und Y-Komponente liefern. Die Abbildung 4-25 und Abbildung 4-26 zeigen die bei diesem Test entstandenen Ergebnisse.

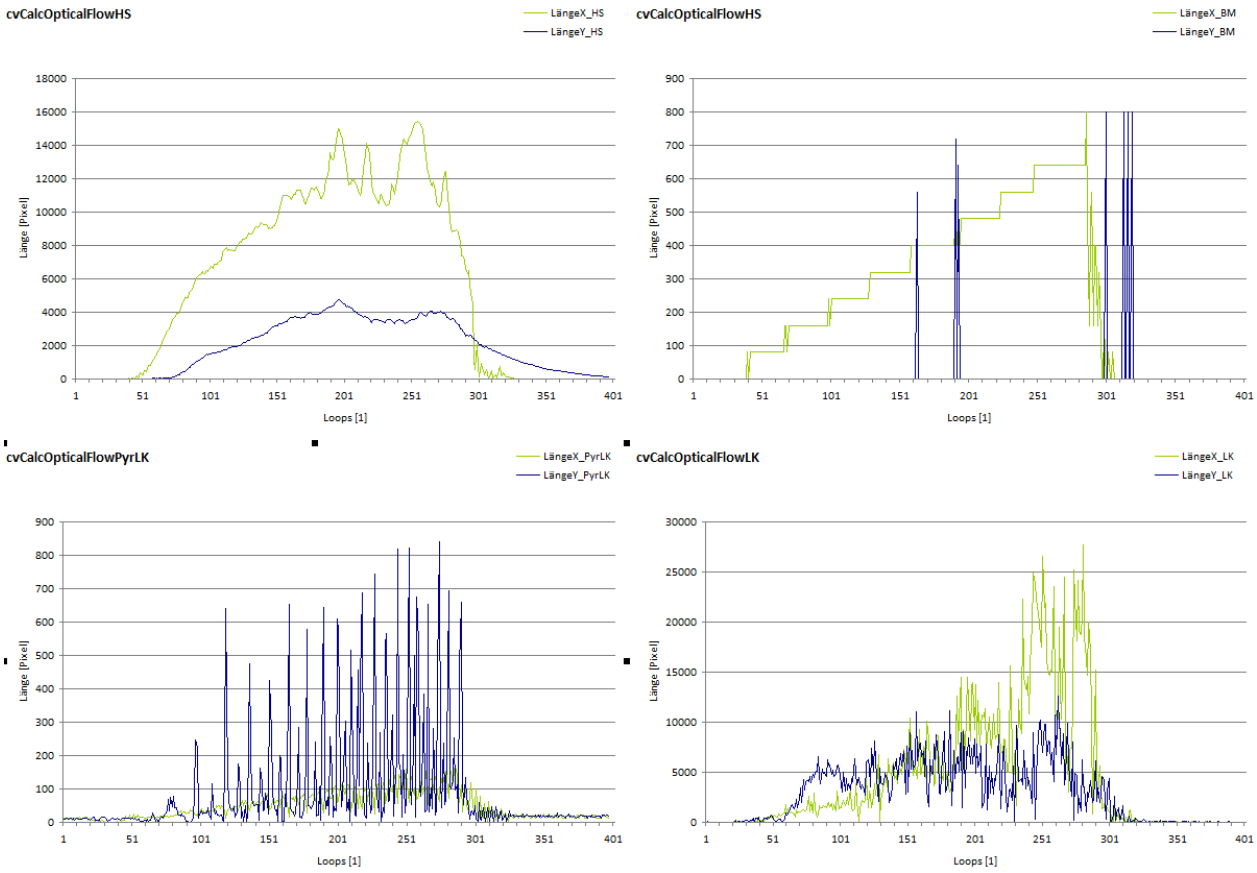


Abbildung 4-25: Aperturtest (grade Linien)

Die Funktion `cvCalcOpticalFlowPyrLK` versagte beim Test auf gerade Apertur, der Algorithmus konnte keine eindeutigen Features finden, deswegen auch keine eindeutige Bewegung erfassen, Das rauschen überwiegt im gesamten Verlauf der Kurve, wobei es die Bewegungsrichtung um 90° verfehlt. Auch bei schiefer Apertur ist das Ergebnis nicht zufrieden stellend, sondern sehr verrauscht, obwohl hier der Algorithmus im Gegensatz zur geraden Apertur, die Richtung fast richtig Detektierte.

Auch der Algorithmus von Horn Und Schunck lieferte kein sehr überzeugendes Ergebnis, bei der geraden Apertur, es wurde Bewegung auch in die Y-Richtung detektiert, vermutlich lag es zum

teil an der Befestigung der Kamera, die Möglicherweise nicht perfekt auf die Bewegungsrichtung ausgerichtet war. Bei der schiefen Apertur gab es auch Abweihungen von dem Idealen Ergebnis. Doch zeigt sich trotzdem die Methode `cvCalcOpticalFlowHS` als zuverlässig, wenn es um Richtungserkennung geht.

Der Block Matching Algorithmus detektierte in beiden Fällen nur die X-Komponente der Bewegung, was bei geraden Linie erwünscht war, verfehlt das Ergebnis bei der schiefen Apertur. Wobei es auch sein kann, dass der Block Matching Algorithmus mit allerart von Apertur sehrgut umgehen kann, da die Bewegung tatsächlich in die X-Richtung ging.

Die Funktion `cvCalcOpticalFlowLK` lieferte ein verrauschtes Ergebnis. Wobei es trotzdem die Richtung der sichtbaren Bewegung [1] richtig erkannte.

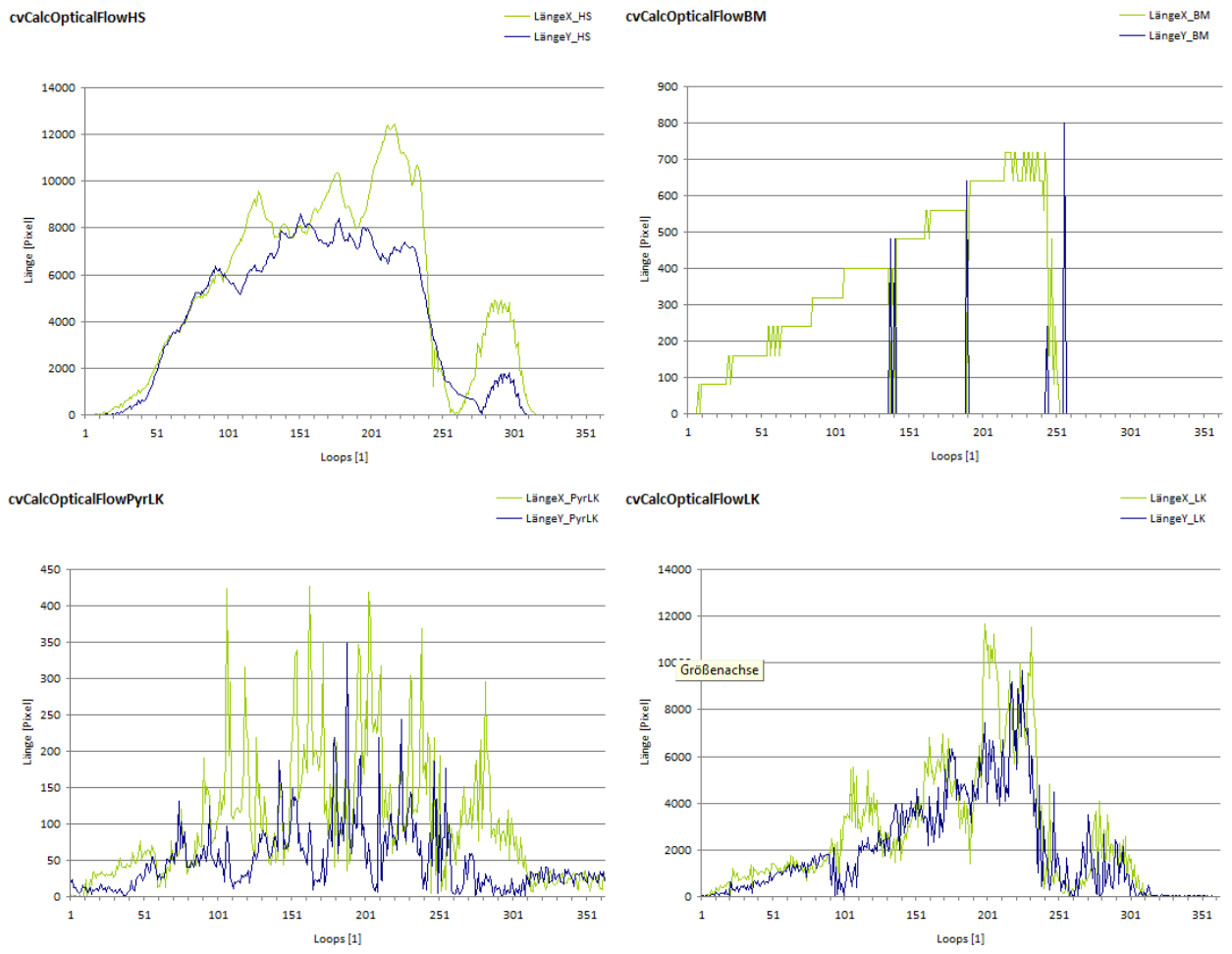


Abbildung 4-26: Aperturtest (schiefe Linien)

[1] In Anbetracht des Blendenproblems, muss sichtbare Bewegung nicht die tatsächliche Bewegung sein, sondern nur ein Teil davon.

4.8. Zusammenfassung der Tests

Die Tests zeigten die Schwächen und die Stärken der Algorithmen. Es ist kein eindeutiger Sieger hervorgegangen, was von Begin an zu erwarten war, doch in einzelnen Disziplinen haben die Algorithmen sehr unterschiedliche Ergebnisse erzielt.

Die langsame Bewegung wird vom Horn & Schunck Algorithmus am besten detektiert, dieser ist auch der, der am wenigsten vom Rauschen geplagt ist, die Länge des errechneten Pfeils hängt aber bei diesem Algorithmus von der Anzahl der Objekte ab die sich im Bild bewegen. Deswegen eignet sich dieser Algorithmus nur bedingt zur Geschwindigkeitsmessung. Ganz ausgeschlossen ist dies aber nicht, denn man kann herausfinden wie fiele Pixel sich in einem Bildpaar bewegen, diese Information kann man dazu nutzen um den Vektor zu normieren, dann wäre der Vektor zwar zur Geschwindigkeitsberechnung einsetzbar, allerdings trotzdem weniger genau bei weniger bewegten Pixels in einem Bildpaar. Bei der Berechnung der Richtung allerdings benötigt man keine normierten Ergebnisse, sondern nur exakte Verhältnisse zwischen der X- und der Y-Komponente, die Methode von Horn & Schunck bietet einen sehr gute Möglichkeit der Richtungsbestimmung, obgleich sie anfällig für das Apperturproblem ist, ist der von der Methode gelieferte Richtung stets sehr genau.

Der Block Matching Algorithmus zeigte seine Vorteile bei der Geschwindigkeitserfassung und Könnte gut dafür eingesetzt werden um die anzuzeigende Geschwindigkeit zu ermitteln, zur Regelung wären die Ergebnisse des Block Matching voraussichtlich zu unpräzise, da er in 80-Pixelstritten springt.

Die beiden Algorithmen von Lucas & Kanade sind im Gegensatz zu den oben genanten Verfahren, sehr rauschanfällig und bieten kaum Möglichkeiten es zu beseitigen, oder gar nicht entstehen zu lassen. Allerdings hat die Methode `cvCalcOpticalFlowPyrLK` ihre Vorteile bei der Geschwindigkeitsschätzung, mit der Voraussetzung die Featurezahl und die Auflösung wird nicht variiert. Der einfache Lucas & Kanade Algorithmus hat eher Vorteile in der einfachen Bedienbarkeit, dessen Ergebnisse im Test allerdings sind eindeutig zu schwach um diesem Algorithmus weitere Beachtung zu schenken.

5. Zusammenfassung und Ausblick

Abschließend lässt sich mit fester Überzeugung sagen, dass die Methoden der OpenCV-Bibliothek zur Regelung des Quadrokopters der Hochschule Esslingen geeignet sind. Die durchgeführten Tests haben Vor- und Nachteile der einzelnen Methoden aufgedeckt und gezeigt dass die Methoden ohne Probleme parallel, zur gleichen Zeit arbeiten können. Das heißt man braucht sich nicht nur auf eine der Methoden zu verlassen sondern kann diese beliebig kombinieren und lediglich die Vorteile der einzelnen Algorithmen ausnutzen. Es hat sich gezeigt dass fast alle Übergabeparameter der Methoden, die auf die Bewegungsfindung Einfluss nehmen können zur Laufzeit variiert werden können, dies wäre beispielsweise bei wechselnden Lichtverhältnissen vom großen Vorteil. Lediglich die Variable der Featureanzahl bei der Funktion `cvCalcOptocalFlowPyrLK`() läst sich nicht variieren und muss zu Beginn einen für den gesamten Ablauf festen Wert annehmen.

Natürlich muss noch einiges getan werden damit diese Methoden zur Regelung des Quadrocopters eingesetzt werden können. Zu einem müssen die Methoden einen normierten Vektor ausgeben können, der mittels Mittelwert- oder Medianbildung realisiert werden kann. Dieser Vektor muss dann mit der Höhe verrechnet werden, um daraus ein Maß für die Geschwindigkeit zu gewinnen. Zum Anderen müssen Kompromisse eingegangen werden in Fragen, was und in welchem Masse die eine oder die andere Methode, in der Ergebnisbildung übernimmt. Wobei die nach dem Horn & Schunck Algorithmus arbeitende Methode eine sehr große Rolle spielen wird, da ihre Testergebnisse im vergleich zu den restlichen Algorithmen überragend waren. Auch wird eine wichtige Frage sein ob die Regelung nicht mit einem anderem Ansatz besser lösbar ist, nämlich dem nicht den optischen Fluss, also die Bewegung zu messen, sondern mittels Musterbildung, beispielsweise mit Hilfe des Featuretrackers, direkt die Lage zu erfassen. Dabei könnten am Boden detektierte Features zu Mustern zusammengefasst werden, die man als einer Art Anker verwendet. In dem man diese Muster versucht im selben Bildbereich zu halten, mittels Reglung des Quadrocopters, wäre der Schwebeflug vermutlich leichter realisierbar.

6. Literaturverzeichnis

[1] http://de.wikipedia.org/wiki/OpenCV

[2] Learning OpenCV (Gary Bradski und Adrian Kaehler)

[3] Navigation und Regelung eines Luftschiffes mittels optischer, inertialer und GPS Sensoren, Doktorarbeit, Universität Stuttgart (Martin Fach)

[4] Bewegungserkennung in Videosequenzen zur Erfassung von Verkehrsteilnehmern, Diplomarbeit, Hochschule Esslingen (Ralph Scharpf)

[5] http://www.ece.cmu.edu/~ee899/project/deepak_mid.htm

[6] http://de.wikipedia.org/wiki/Methode_der_kleinsten_Quadrate

[7] Vorlesungspräsentation, der Veranstaltung „Bildverarbeitung und Computer Vision, Kapitel 13 – Optische Flüsse, Uni-Münster (von Prof. Dr. Xiaoyi Jiang, Michael Schmeing und Lucas Franek)

[8] Detection and Tracking of Point Features, (Carlo Tomasi und Takeo Kanade)

[9] http://opencv.willowgarage.com/documentation/drawing_functions.html

[10] http://blog.aguskurniawan.net/post/OpenCV-210-with-Visual-Studio-2010.aspx , Installation von OpenCV und Integration in Visual Studio 2010

[11] http://www.jestinstoffel.com/?q=node/112 , Integration von OpenCV in Eclipse

[12] http://www.ece.cmu.edu/~ee899/project/deepak_mid.htm , Search Algorithms for Block-Matching in Motion Estimation

[13] http://de.wikipedia.org/wiki/Markow-Kette