Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Andrej Hofmann und Baljinder Singh

Shared Guide Dog - Hinderniserkennung / Erkennung von herabhängenden Objekten mit einer monokularen Kamera

Fakultät Technik und Informatik
Department Fahrzeugtechnik und Flugzeugbau

Faculty of Engineering and Computer Science
Department of Automotive and Aeronautical Engineering

Andrej Hofmann und Baljinder Singh

Shared Guide Dog - Hinderniserkennung / Erkennung von herabhängenden Objekten mit einer monokularen Kamera

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

im Studiengang Bachelor of Science Mechatronik
am Department Fahrzeugtechnik und Flugzeugbau
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Erstprüfer: Prof. Dr. Jochen Maaß
Zweitprüfer: Prof. Dr. Henner Gärtner

Abgabedatum: 26.08.2021

Zusammenfassung

Andrej Hofmann und Baljinder Singh

Thema der Bachelorthesis

Shared Guide Dog – Hinderniserkennung / Erkennung von herabhängenden Objekten mit einer monokularen Kamera

Stichworte

Hindernisdetektion, Bildverarbeitung, OpenCV, Optischer Fluss, Template Matching

Kurzzusammenfassung

In dieser Arbeit werden Ansätze basierend auf optischem Fluss und Template Matching für eine Hindernisdetektion mittels einer monokularen Kamera implementiert und getestet. Dabei liegt der Fokus auf der Detektion von herabhängenden Objekten. Die Implementierung erfolgt mithilfe der OpenCV-Bibliothek.

Andrej Hofmann and Baljinder Singh

Title of the paper

Shared Guide Dog – obstacle detection / detection of hanging objects using a monocular camera

Keywords

obstacle detection, image processing, OpenCV, optical flow, template matching

Abstract

In this thesis, approaches for obstacle detection based on optical flow and template matching are implemented and tested. The focus is the detection of hanging objects. These approaches are implemented using the OpenCV-Library.

Inhaltsverzeichnis

1 Einleitung 1

1.1 Problemstellung 2
1.2 Vorgehen 3

2 Stand der Technik 4

2.1 Tiefe durch relative Höhe 5
2.2 Tiefe durch Schärfe / Unschärfe 1 7
2.3 Tiefe durch Schärfe / Unschärfe 2 9
2.4 Tiefe durch Schattierung 11
2.5 Form durch Textur 12
2.6 Tiefe durch Kantenverdeckung 13
2.7 Tiefe durch Perspektive 15
2.8 Hinderniserkennung durch Aussehen 17
2.9 Hinderniserkennung durch optischen Fluss / Template Matching 19
2.10 Hinderniserkennung durch Template Matching 21
2.11 Tiefe durch parametrisches Lernen 23
2.12 Tiefe durch nicht-parametrisches Lernen 25
2.13 Tiefe durch unüberwachtes maschinelles Lernen 26
2.14 Tiefe durch überwachtes maschinelles Lernen 28

3 Konzeption 30

3.1 Bewertungskriterien 31
3.2 Festlegung der zu implementierenden Ansätze 32
3.2.1 Detektion von herabhängenden Hindernissen 32
3.2.2 Verwendung einer monokularen Kamera 32
3.2.3 Trennung von herannahenden und fernen Objekten 32
3.2.4 Niedrige Berechnungsdauer 34
3.2.5 Auswahl 34
3.3 Generierung der Testdaten 35

4 Methoden 38

4.1 Optischer Fluss 39
4.1.1 Lucas-Kanade-Algorithmus 39
4.1.2 Farnebäck-Algorithmus 40
4.1.3 Tiefe durch optischen Fluss 41
4.2 Template Matching 45
4.3 Merkmalsvergleich 47
4.4 Clustering 49

4.5 Kamerakalibrierung 50

5 Implementierung 51

5.1 Software 52
5.2 OpenCV-Bibliothek 52
5.3 Ansatz 1: Optischer Fluss und Template Matching 53
5.3.1 Hintergrundsubtraktion und Filterung 53
5.3.2 Merkmalsdetektion und Clustering 55
5.3.3 Template Matching und Optischer Fluss 56
5.3.4 Testergebnisse 59
5.4 Ansatz 2: Template Matching 62
5.4.1 Merkmalsdetektion und -vergleich 62
5.4.2 Template Matching 64
5.4.3 Testergebnisse 66
5.5 Ansatz 3: Tiefe durch optischen Fluss 68
5.5.1 Testergebnisse 75

6 Auswertung 83

6.1 Auswertung des Ansatzes 1 84
6.2 Auswertung des Ansatzes 2 85
6.3 Auswertung des Ansatzes 3 86
6.4 Fazit und Ausblick 88

Literaturverzeichnis 90

Abbildungsverzeichnis 94

Tabellenverzeichnis 96

Kennzeichnung der verfassten Kapitel / Seiten 97

Anhang A: CD 98

Anhang B: Quellcode 99

Ansatz 1 99
Ansatz 2 109
Ansatz 3 Variante 1 114
Ansatz 3 Variante 2 121
Ansatz 3 Variante 3 128

1 Einleitung

Diese Arbeit ist Teil des Projekts *Shared Guide Dog 4.0*, welches an der HAW durchgeführt wird. Bei dem Projekt *Shared Guide Dog 4.0* geht es um die Entwicklung eines technischen Assistenzsystems, welches die Aufgaben eines Blindenhundes übernimmt und erweitert. Dieses Assistenzsystem hat die Aufgabe Menschen mit Sehbehinderung sicher und autonom zu Navigieren. Bei dem aktuellen Prototyp handelt es sich um einen Rollator, welcher mit verschiedenen Sensoren und Motoren für ein eigenständiges Fortbewegen ausgestattet ist.

Das Ziel dieser Bachelorarbeit ist es, für das selbstfahrende Assistenzsystem eine Hinderniserkennung mithilfe einer monokularen Kamera zu implementieren, um ein sicheres und eigenständiges Fortbewegen zu gewährleisten.

1.1 Problemstellung

Die Ausbildung von Blindenführhunden ist zeitintensiv und muss individuell auf die speziellen Einschränkungen der Person angepasst werden. Die Folge daraus ist, dass ein ausgebildeter Blindenhund lediglich von einer Person genutzt werden kann.

Aufgrund der zeitintensiven Ausbildung und der hohen Kosten, die solch eine Ausbildung mit sich bringt, ist die Verfügbarkeit von Blindenführhunden beschränkt und nicht für jeden Menschen mit Sehbehinderung steht ein Blindenführhund zur Verfügung. Auch sind herabhängende Gegenstände wie beispielsweise der Ast eines Baumes, Wegbegrenzungen wie Schranken oder Straßenschilder mit einem Blindenstock nur schwer oder nicht erkennbar.

Die Problematik um herabhängende Gegenstände und die begrenzte Verfügbarkeit von Blindenführhunden soll das Assistenzsystem *Shared Guide Dog 4.0* beheben. Speziell der Aspekt der Erkennung von herabhängenden Objekten und anderen Hindernissen ist das Ziel dieser Arbeit. In der folgenden Abbildung 1 wird der aktuelle Prototyp des *Shared Guide Dog 4.0* dargestellt (Stand 14.06.21).

Abbildung 1: Prototyp des *Shared Guide Dog 4.0*

1.2 Vorgehen

Um das Ziel der Implementierung einer optischen Hinderniserkennung zu erreichen, wird zunächst der aktuelle Stand der Technik der Hindernisdetektion mithilfe einer monokularen Kamera recherchiert. Außerdem werden aktuelle Ansätze für die Generierung von Tiefendaten durch monokulares Video ermittelt. Die einzelnen Methoden bzw. Ansätze werden mit den jeweiligen Vor- und Nachteilen vorgestellt. Die vielversprechendsten Ansätze werden durch die Betrachtung der jeweiligen Vor- und Nachteile ausgewählt. Daraufhin werden die ausgewählten Ansätze mithilfe der Bildverarbeitungsbibliothek *OpenCV* [1] implementiert. Nach der Implementierung werden Beispielaufnahmen mithilfe des Prototyps des *Shared Guide Dog 4.0* erstellt, um den jeweiligen Ansatz zu testen. Die Ergebnisse des Tests werden vorgestellt und es wird ein Fazit zu den einzelnen Methoden abgegeben. Am Ende der Arbeit wird ein Ausblick für mögliche Verbesserungen oder weitere Arbeiten beschrieben.

Das Vorgehen wird in der folgenden Auflistung verdeutlicht:

1. Ermittlung des aktuellen Stands der Technik zur Hindernisdetektion durch eine monokulare Kamera
2. Beschreibung der einzelnen Ansätze
3. Auswahl der zu verwendenden Methoden
4. Implementierung der Methoden
5. Test und Ergebnisvorstellung
6. Fazit

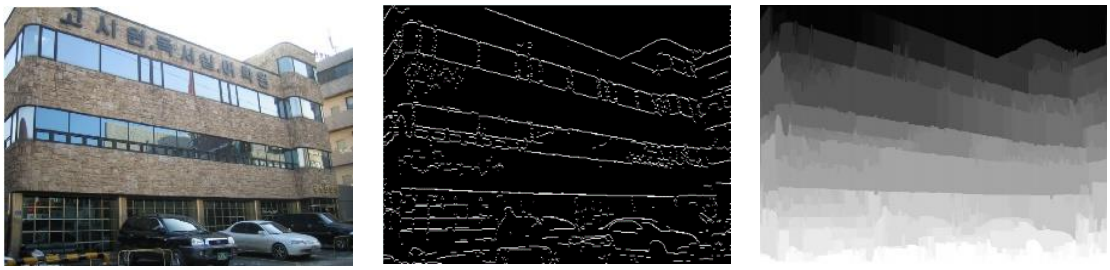
2 Stand der Technik

Um den aktuellen Stand der Technik zu ermitteln, werden verschiedene Ansätze und Methoden für die Generierung von Tiefeninformationen und Vorgehensweisen für eine Hindernisdetektion mithilfe von monokularem Video recherchiert. Die Methoden dieser Ansätze werden erläutert und deren Vor- und Nachteile vorgestellt. Anhand der Vor- und Nachteile werden in dem Kapitel Konzeption mittels einer Bewertungsanalyse die für den Anwendungsfall am besten geeigneten Ansätze ermittelt. Die Ansätze werden in den folgenden Unterkapiteln unterteilt.

2.1 Tiefe durch relative Höhe

In dem Artikel von Jung et al. [2] werden Bilder, die über eine monokulare Kamera aufgenommen werden, durch einen Algorithmus in Stereo-Bilder konvertiert. Um die Stereo-Bilder zu generieren, wird zunächst eine Tiefenkarte des Eingangsbildes erstellt. Da für den Anwendungsfall dieser Arbeit nur der Aspekt der Generierung von Tiefeninformationen von Bedeutung ist, wird nur dieser Teil genauer beschrieben.

Um eine Tiefenkarte zu generieren, werden zunächst Kanten durch einen Kanten-Detektor auf dem Eingangsbild detektiert. Daraufhin wird ein Linien-Verfolgungs-Algorithmus verwendet, welcher mit den Kanteninformationen die ausgeprägtesten Kanten auswählt und verfolgt. Mit den ausgewählten Kanten werden Linien generiert, die vom linken bis zum rechten Bildrand reichen. Durch die Anwendung des Algorithmus entsteht eine sogenannte Linienkarte mit festgelegter Linienanzahl. Die Linienkarte besitzt nur die Informationen der ausgeprägtesten Kanten. Die für den Algorithmus irrelevanten Kanten werden verworfen. Ein Beispiel für solch eine Linienkarte und das dazugehörige Eingangsbild ist in der Abbildung 2 (a und b) zu sehen. Den durch die Linien abgegrenzten Bereiche werden Tiefen in steigender Reihenfolge zugewiesen. So werden Bereiche im unteren Teil des Bildes als nah und Bereiche weiter oben im Bild als fern deklariert. Die zugewiesene Tiefe in den einzelnen Bereichen bezieht sich somit nur auf die Höhe der Bereiche im Eingangsbild. In der Abbildung 2 (c) ist ein Beispiel für eine generierte Tiefenkarte zu sehen.



(a) (b) (c)

Abbildung 2: Tiefenberechnung durch relative Höhe a) das Eingangsbild b) die dazugehörige Linienkarte c) die dazugehörige Tiefenkarte [2]

Durch die vorgestellte Methode wird eine grobe Tiefenkarte in Abhängigkeit der relativen Höhe von Objekten im Eingangsbild erzeugt. Die generierten Tiefeninformationen repräsentieren jedoch nicht die tatsächliche Tiefe einzelner Objekte. Die Methode findet, wie auch in dem Artikel beschrieben, ihren Einsatz in der Generierung von Stereobildern für 3D-Mediageräte.

2.2 Tiefe durch Schärfe / Unschärfe 1

Tang et al. [3] stellen eine Methode für die Generierung einer Tiefenkarte vor, welche die Tiefeninformationen über die Unschärfe abschätzt. Zudem wird in dem Artikel eine Methode für die Optimierung der generierten Tiefenkarten vorgestellt. Die Grundidee beruht auf dem Prinzip, dass abhängig von der Distanz zwischen der Kamera und einem Objekt der Bildsensor der Kamera unterschiedliche Werte liefert. Das Prinzip wird in Abbildung 3 visualisiert. Jede farbig dargestellte Linie stellt eine unterschiedliche Entfernung zwischen der Kamera und einem Objekt dar.

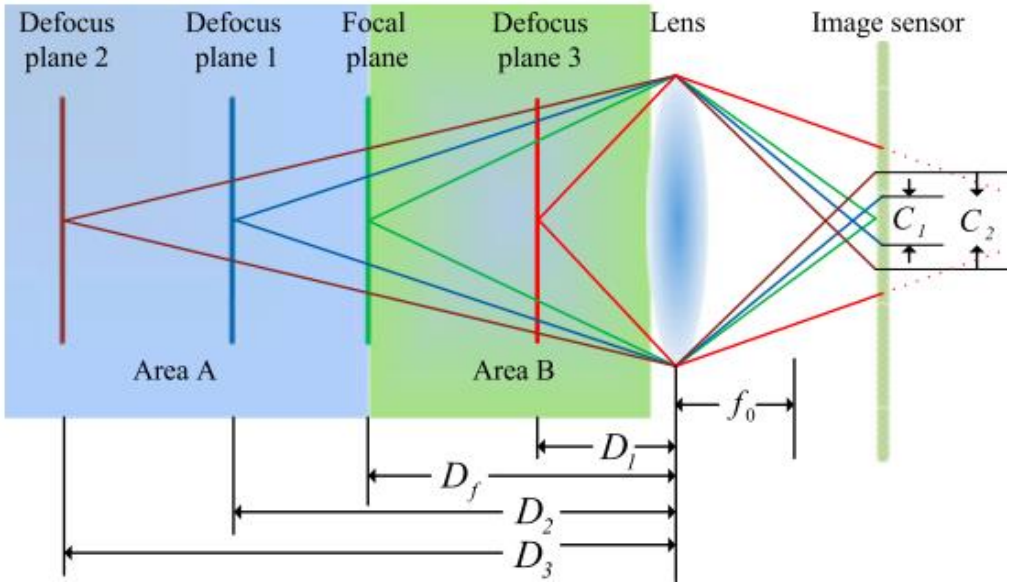


Abbildung 3: Zusammenhang zwischen Schärfe / Unschärfe und Distanz [3]

Für die Distanzen in den blau und grün hinterlegten Flächen liefert der Bildsensor messbare Werte für die Unschärfe. Anhand der gemessenen Werte für die Unschärfe lässt sich die Tiefe abschätzen. Da sich die Unschärfe am besten in Regionen mit hohem Kontrast wie Ecken oder Kanten schätzen lässt, wird vorweg eine Kantendetektion am Eingangsbild durchgeführt. Hierfür wird der Kontrast zwischen zwei benachbarten Pixeln verglichen. Nach der Kantendetektion wird an den Kanten mithilfe der in dem Artikel beschriebenen

Funktion die Tiefe abgeschätzt. Diese Funktion beschreibt die Verhältnismäßigkeit zwischen der Tiefe und der Unschärfe eines Objekts bzw. eines Pixels. Aus diesen Tiefenwerten wird eine vorläufige Tiefenkarte generiert, die mithilfe eines Algorithmus vervollständigt wird. Dieser Algorithmus wird in dem Artikel von Levin et al. [4] genauer erläutert. Anschließend wird die vervollständigte Tiefenkarte mit weiteren Optimierungsmethoden überarbeitet. Für die Recherche dieser Arbeit ist jedoch der Optimierungsaspekt nicht von Bedeutung, daher wird auf diesen im Folgenden nicht weiter eingegangen und es folgt keine genauere Beschreibung. Die Abbildung 4 stellt das Eingangsbild sowie die daraus generierte Tiefenkarte dar. Die hierbei angewandte Methode ist mit handelsüblichen Kameralinsen ausschließlich für Nah- und Detailaufnahmen anwendbar. Wie in der Abbildung 4 zu sehen ist, werden Details wie die Pollen der Blüte bei der Tiefenkarte genau wiedergegeben bzw. aufgezeichnet.

(a) (b)

Abbildung 4: Tiefenberechnung durch Schärfe / Unschärfe 1
a) das Eingangsbild b) die dazugehörige Tiefenkarte [3]

2.3 Tiefe durch Schärfe / Unschärfe 2

Die in dem Artikel von Martinello und Favaro [5] beschriebene Methode für die Generierung von Tiefendaten durch Videoaufnahmen einer monokularen Kamera basiert auf einem codierten Blendensystem.
Der verwendete Algorithmus schätzt Tiefe anhand der Schärfe bzw. Unschärfe einzelner Objekte im Bild. Wenn ein Objekt im Bild im Fokus ist, also scharf abgebildet wird, ist ein anderes Objekt, welches nicht die gleiche Entfernung zur Kamera hat, unschärfer im Bild abgebildet. Über den Grad der Unschärfe kann die relative Tiefe bzw. die relative Entfernung zur Kamera einzelner Objekte berechnet werden. Bei bekannten Kamera-Parametern kann auch die exakte Tiefe bestimmt werden. Die hier verwendete codierte Maske, welche auf die Linse der Kamera gelegt wird, verbessert die Schärfen- bzw. Unschärfen-Identifikation und führt dadurch auch zu einer verbesserten Tiefenberechnung. Die in dem Artikel verwendete Maske wird in der Abbildung 5 dargestellt.

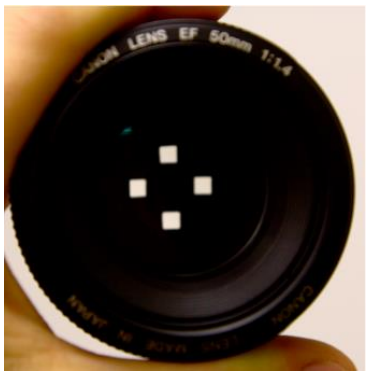
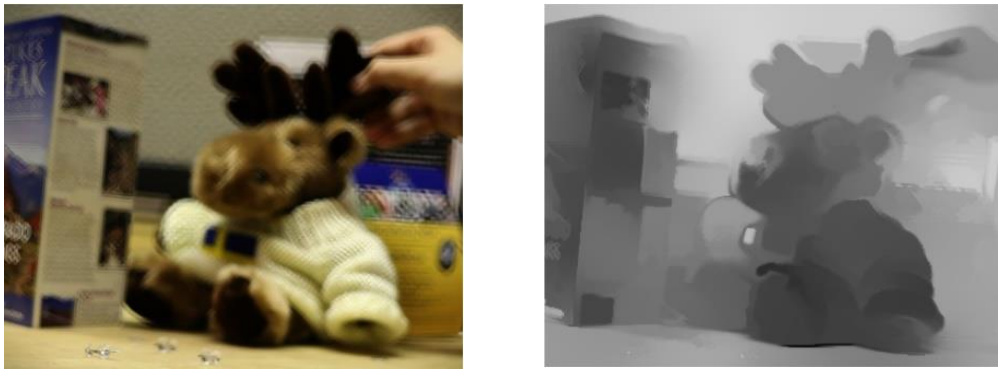


Abbildung 5: Codierte Maske [5]

Bei der Berechnung wird angenommen, dass Regionen mit ähnlicher Farbe bzw. Textur auch eine ähnliche Tiefe besitzen und wahrscheinlich zum gleichen Objekt gehören. Es werden sowohl Regionen innerhalb eines Bildes als auch Regionen in aufeinanderfolgenden Bildern des Videos betrachtet. Je weiter Regionen voneinander entfernt sind, desto unwahrscheinlicher ist es, dass diese zum gleichen Objekt gehören. Die Tiefe der einzelnen Bilder wird durch die Minimierung einer entworfenen Kostenfunktion berechnet. Ein Beispiel für eine berechnete Tiefenkarte mit dem dazugehörigen Eingangsbild ist in Abbildung 6 zu sehen.



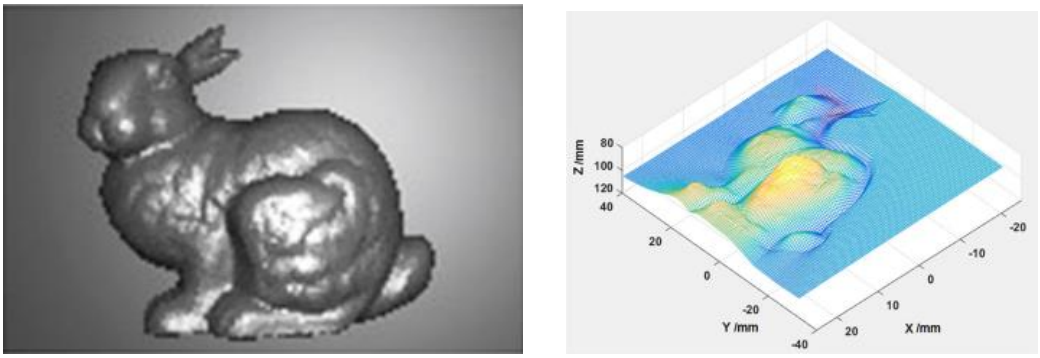
(a) (b)

Abbildung 6: Tiefenberechnung durch Schärfe / Unschärfe 2
a) das Eingangsbild b) die dazugehörige Tiefenkarte [5]

Die vorgestellte Methode ist nicht von Bewegungen innerhalb eines Videos abhängig und die Tiefe kann durch Betrachtung einzelner Bilder berechnet werden. Die Berechnungszeit eines Bildes der Größe von 500 x 600 Pixeln liegt bei ca. 4 Sekunden. Der Artikel wurde im Jahr 2012 veröffentlicht und es kann davon ausgegangen werden, dass die Berechnung auf moderner Hardware weniger Zeit in Anspruch nimmt.

2.4 Tiefe durch Schattierung

Durch Variationen von Schattierungen auf der Oberfläche von Objekten kann die Form bzw. die Tiefe eines Objektes berechnet werden. In dem Artikel von Fan et al. [6] wird eine Methode für die Tiefenberechnung von monokularen Bildern durch Schattierung beschrieben.
Die Tiefenberechnung wird hier über die Minimierung einer Gleichung gelöst, welche die Reaktion von Licht auf Oberflächen beschreibt. Diese Gleichung wird mithilfe des Modells von Schlick [7] formuliert. In diesem Modell werden sowohl die Reflexionen von diffusen als auch von spiegelnden Oberflächen beschrieben. Dabei muss die Lichtquellenposition nicht definiert werden. Für die Lösung der Gleichung wird ein Startwert für die Tiefe Z vorgegeben.
In der folgenden Abbildung 7 ist ein Beispiel für eine generierte Tiefenkarte und das dazugehörige Eingangsbild dargestellt. Bei dieser Tiefenkarte wurde ein Bild eines Hasen-Modells mit einer spiegelnden Oberfläche als Eingangsbild verwendet.



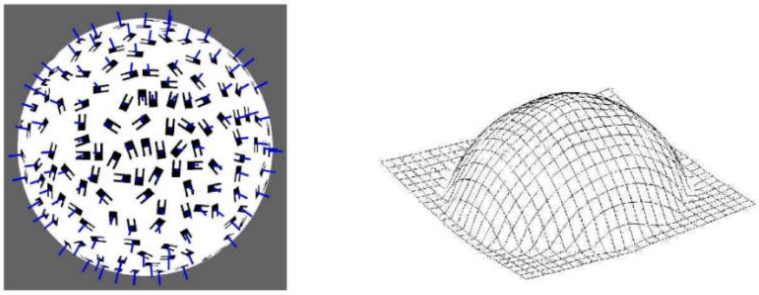
(a) (b)

Abbildung 7: Tiefenberechnung durch Schattierung a) das Eingangsbild b) die dazugehörige Tiefenkarte [6]

Der beschriebene Algorithmus konvergiert bei einem Startwert von Z = 50 mm und einer Bildgröße von 100 x 200 Pixeln nach 251 Iterationen. Dies nimmt ca. 31 s in Anspruch, jedoch kann die Berechnungszeit durch Verwendung einer Bildpyramide verringert werden.
Der vorgestellte Ansatz wird meist bei medizinischen Anwendungen wie z. B. bei endoskopischen Operationen verwendet.

2.5 Form durch Textur

Loh und Hartley [8] präsentieren einen Ansatz für die Rekonstruktion der Form eines Objektes in einem Bild. Bei dem Ansatz werden zunächst Texturmerkmale auf der Oberfläche eines Objektes gesucht. Aus diesen Merkmalen wird die räumliche Form bzw. Struktur des Objektes mithilfe einer affinen Transformation berechnet. Die affine Transformation beschreibt die Änderung der geometrischen Verschiebung (Position), Drehung (Orientierung) und Skalierung (Größe).
Zunächst werden mit der Methode von Matas et al. [9] Regionen mit ähnlichen Eigenschaften im Eingangsbild gesucht und als Texturmerkmale gekennzeichnet. Um die affine Transformation durchführen zu können, wird die Position, Orientierung und Größe eines Texturmerkmals als Referenz festgelegt. Mit der Methode „*Scale-Invariant Feature Transform*" (SIFT) von Lowe [10] werden weitere Texturmerkmale im Bild gesucht. Für jedes gefundene Texturmerkmal wird die affine Transformation durchgeführt. Durch die berechneten Transformationen lässt sich die Form des Objektes rekonstruieren. Ein Beispiel für eine texturierte Oberfläche und die dazugehörige Form, welche nach der oben beschriebenen Methode berechnet wurde, ist in Abbildung 8 zusehen.



(a) (b)

Abbildung 8: Berechnung einer Form durch Textur a) das Eingangsbild b) die berechnete Form [8]

Der vorgestellte Ansatz benötigt detektierbare Texturmerkmale auf der Oberfläche eines Objektes, um daraus die Form bzw. Struktur rekonstruieren zu können. Die Texturmerkmale müssen alle die gleiche Größe aufweisen, da dies sonst zu Fehlern in der Berechnung der Form führt.

2.6 Tiefe durch Kantenverdeckung

Der im Folgenden beschriebene Ansatz aus dem Artikel von Ming et al. [11] basiert auf der Berechnung einer Tiefen-Reihenfolge durch Kantendetektion. Die verwendeten Eingangsbilder werden durch eine monokulare Kamera aufgenommen. Jedes Eingangsbild wird in sich nicht überlappende Regionen unterteilt. Die Unterteilung dieser Regionen wird über die Farbe, Textur und detektierten Kanten realisiert. Dadurch ergibt sich ein 91-dimensionaler Merkmalsvektor für jede der unterteilten Regionen. Da die Merkmalsvektoren redundante Informationen enthalten, wird die Dimension dieser Vektoren über einen Lernprozess reduziert. In dem Lernprozess werden vor der eigentlichen Anwendung der Methode Gewichte für die Merkmalsvektoren gelernt. Nach der Dimensionsreduktion wird eine Kanten-Klassifizierung angewandt, welche triviale von sogenannten Verdeckungskanten trennt. Verdeckungskanten sind Kanten, welche aus dem Betrachtungswinkel der Kamera über oder hinter anderen Kanten liegen. Nach der Klassifizierung der detektierten Kanten wird ein sogenannter „*triple descriptor*" verwendet. Durch diesen werden die Beziehungen zwischen einzelnen Regionen mittels Kreuzungen und Kurven der Kanten beurteilt. Durch den „*triple descriptor*" wird eine ungefähre Tiefen-Reihenfolge der einzelnen Regionen berechnet. Diese Einschätzung wird im nächsten Schritt verwendet, um die tatsächliche Tiefen-Reihenfolge zu bestimmen. Als Nächstes wird die Tiefen-Reihenfolge der Regionen über einen gerichteten Graphen bestimmt. In diesem Graphen wird nach einem validen Pfad gesucht, der auch mit der Berechnung des „*triple descriptors*" übereinstimmt. Dieser Pfad gibt die tatsächliche Tiefen-Reihenfolge wieder. In der Abbildung 9 werden ein Eingangsbild und die dazugehörige Tiefenkarte abgebildet.

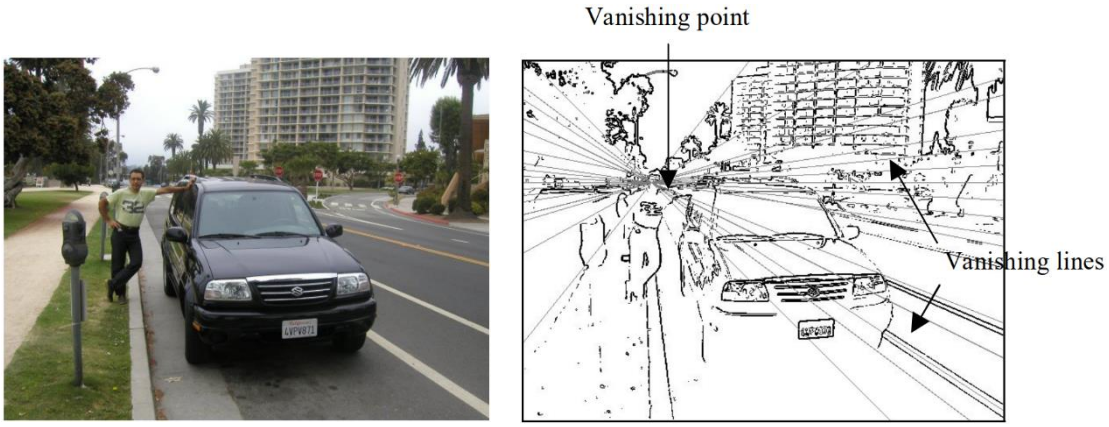
(a) (b)

Abbildung 9: Tiefenberechnung durch Kantenverdeckung
a) das Eingangsbild b) die dazugehörige Tiefenkarte [11]

Wie in der Abbildung 9 zu sehen ist, werden keine exakten Tiefenwerte berechnet. Die einzelnen Objekte werden in verschiedenen Grautönen anhand ihrer relativen Tiefe dargestellt.
Die in dem Artikel beschriebenen Experimente wurden mit zwei verschiedenen Datensätzen vollzogen. Die Trainingsdauer bei diesen Datensätzen lag bei 20 - 35 Minuten. Die eigentliche Berechnung eines Bildes nahm ca. 3 Sekunden in Anspruch. Da der Artikel aus dem Jahr 2015 stammt, kann davon ausgegangen werden, dass die Berechnungszeit auf moderner Hardware weniger Zeit in Anspruch nimmt.

2.7 Tiefe durch Perspektive

In dem Artikel von Battiato et al. [12] wird ein Ansatz zur Generierung von Tiefendaten vorgestellt. Die Tiefe wird basierend auf den geometrischen und farblichen Informationen in einem Bild geschätzt und dadurch eine grobe Tiefenkarte erstellt.
Mit den farblichen Informationen werden Regionen ähnlicher Farbe wie z. B. Wolken, Wiesen und Berge segmentiert. Die segmentierten Regionen werden mit einem angelernten System regelbasiert klassifiziert. Die Klassifizierung erfolgt abhängig von der Farbe und Lage des Objektes im Bild. Ein Beispiel für solch eine Regel ist, dass wenn sich ein Objekt in der oberen Hälfte des Bildes befindet und eine bläuliche Farbe aufweist, dieses Objekt als Himmel klassifiziert wird. Die Regeln für die Klassifizierung werden anhand einer Datensammlung (Bilder) angelernt. Zudem wird mithilfe der Datensammlung eine grobe Tiefenkarte des Eingangsbildes erstellt, welche die Segmentierung und eine abgeschätzte Reihenfolge der Objekte beinhaltet.
Um die geometrischen Informationen zu verwenden, wird im Bild nach Fluchtlinien und deren Schnittpunkt, dem sogenannten Fluchtpunkt, gesucht. Hierfür werden mit dem Sobel-Kantendetektor die Kanten bzw. Geraden im Bild gesucht und bis zu dem Schnittpunkt verlängert. Ein Beispiel für ein Eingangsbild und die darin detektierten Fluchtlinien (Vanishing lines) werden mit dem Fluchtpunkt (Vanishing point) in Abbildung 10 dargestellt.



(a) (b)

Abbildung 10: Fluchtlinien und Fluchtpunkt a) das Eingangsbild b) detektierte Fluchtlinien und Fluchtpunkt [12]

Die Tiefe wird mithilfe der Fluchtlinien und dem Fluchtpunkt geschätzt. Der Fluchtpunkt wird in einem Bild als die entfernteste Stelle angenommen. Daraus resultiert, dass je näher ein Objekt an dem Fluchtpunkt ist, desto weiter entfernt ist es von der Kamera.
Zum Schluss werden die beiden Tiefenkarten, die durch Klassifizierung und Geometrie berechnet wurden, zusammengefasst. Ein Beispiel für eine zusammengesetzte Karte ist in Abbildung 11 zu sehen.

(a) (b)

Abbildung 11: Tiefenberechnung durch Perspektive a) das Eingangsbild b) die dazugehörige Tiefenkarte [12]

Der vorgestellte Ansatz liefert eine sehr grobe Tiefenkarte ohne exakte Werte für die Tiefe. Zudem werden Bilder mit großen Distanzen zwischen der Kamera und den Objekten benötigt.

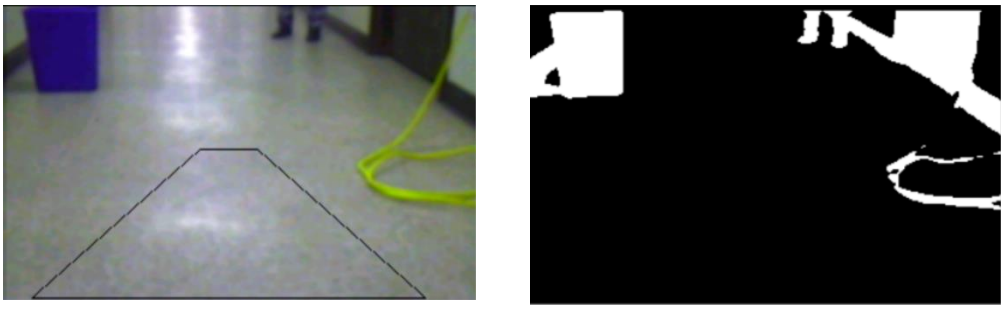
2.8 Hinderniserkennung durch Aussehen

In dem Artikel von I. Ulrich und I. Nourbakhsh [13] werden Hindernisse durch Farbe detektiert. In diesem Ansatz wird jeder Pixel des Eingangsbildes entweder als Boden oder als Hindernis klassifiziert.
In dem aufgenommenen Eingangsbild wird eine trapezförmige Referenzfläche im unteren Teil des Bildes definiert. Diese Referenzfläche wird als Boden behandelt. Bei diesem Ansatz werden drei Annahmen getroffen: Hindernisse unterscheiden sich in ihrem Aussehen von dem Boden, der Boden ist flach und es gibt keine herabhängenden Hindernisse. Damit der Boden von Hindernissen unterschieden werden kann, müssen erst Referenzaufnahmen gemacht werden. Mit diesen Aufnahmen wird das Aussehen des Bodens gelernt.
Im Folgenden wird das Vorgehen genauer beschrieben. Als erstes wird auf das Eingangsbild ein 5 x 5 Gauß-Filter angewandt. Daraufhin wird das gefilterte Bild von dem RGB- in den HSI-Farbraum transformiert. Anschließend wird jeder Pixel des transformierten Bildes einzeln betrachtet und es wird mittels eines Schwellwerts für Intensität und Sättigung ein neues Bild generiert. Dazu werden dem Farbton und der Sättigung nur Werte im neuen Bild zugeteilt, falls die dazugehörige Intensität über einem bestimmten Wert liegt. Gleichermaßen wird dem Farbton nur ein Wert im neuen Bild zugeteilt, falls die dazugehörige Sättigung über einem bestimmten Wert liegt.
In dem neu generierten Bild werden Farbton- und Intensitätshistogramme für die definierte Referenzfläche erstellt. Nach dem Training kann über diese Histogramme der Boden von Hindernissen in neu aufgenommenen Bildern getrennt werden. Dazu wird jedes Eingangsbild wie auch bei den Referenzaufnahmen gefiltert und in den HSI-Farbraum transformiert. Jeder Pixel des gefilterten Eingangsbildes wird den Farbton- und Intensitätshistogrammen gegenübergestellt. Ein Pixel wird als Hindernis gesehen, sollte eine der folgenden Bedingungen stimmen:

1. Der Histogrammwert des Farbtons des Eingangs-Pixels ist niedriger als ein definierter Schwellwert.
2. Der Histogrammwert der Intensität des Eingangs-Pixels ist niedriger als ein definierter Schwellwert.

Sollte keine der beiden Bedingungen stimmen, wird der Pixel als Boden klassifiziert. Damit dieses Vorgehen auch bei verschiedenen Arten von Böden funktioniert, wird das Aussehen des Bodens auch nach dem eigentlichen Training gelernt. Dazu werden unter bestimmten Bedingungen weiter

Histogramme von der Referenzfläche erstellt. Diese Histogramme werden gespeichert und mit den vorherigen verodert. In der folgenden Abbildung 12 wird ein Eingangsbild mit einer trapezförmigen Referenzfläche und das dazugehörige Ausgangsbild dargestellt. Auf dem Ausgangsbild sind die Bereiche, welche als Boden klassifiziert sind, schwarz dargestellt. Die weißen Bereiche sind Objekte, welche als Hindernis klassifiziert sind.



(a) (b)

Abbildung 12: Hinderniserkennung durch Aussehen a) das Eingangsbild b) das dazugehörige Ausgangsbild [13]

Mit dieser Methode werden, falls die drei oben beschriebenen Bedingungen für die Szene zutreffen, Hindernisse von dem Boden unterschieden. Eine Aussage darüber, wie weit ein Hindernis von der Kamera entfernt ist oder wann es zur Kollision kommen würde, kann nicht getroffen werden.

2.9 Hinderniserkennung durch optischen Fluss / Template Matching

In der Masterarbeit von J. M. Sagar [14] wird eine Methode der Hinderniserkennung für Drohnen beschrieben. Für die Hinderniserkennung werden Bilder während des Fluges mit einer monokularen Kamera aufgenommen. Im Folgenden wird die Hinderniserkennung beschrieben.
Als erstes wird eine Hintergrundsubtraktion durchgeführt, um die Anzahl der zu beobachtenden Objekte zu reduzieren. Durch die Hintergrundsubtraktion werden nur die Objekte betrachtet, die sich in den aufgenommenen Bildern bewegen. Nach diesem Schritt wird in dem gefilterten Bild nach Konturen gesucht, deren Größe über 1 % der Bildauflösung liegt. Anschließend wird ein Bild erstellt, in dem nur die gefilterten Konturen abgebildet sind. Durch die Filterung ergibt sich ein rauschfreies Bild. In Abbildung 13 sind Bilder nach der Hintergrundsubtraktion mit und ohne die Filterung zu sehen.

Abbildung 13: Konturendetektion und Filterung a) detektierte Konturen b) detektierte Konturen gefiltert [14]

Daraufhin werden markante Bildmerkmale aufgenommen. Für die Merkmalsdetektion wird der Algorithmus von Shi und Tomasi [15] verwendet, mit dem Ecken von Objekten detektiert werden. Die Detektion von den Ecken erfolgt für jedes aufgenommene Bild.
Als Nächstes wird die Bewegung von Merkmalen in aufeinanderfolgenden Bildern mittels optischen Flusses berechnet, um nahe von fernen Objekten basierend auf zurückgelegter Pixeldistanz zu segmentieren.
Die Segmentierung erfolgt unter der Annahme, dass Objekte, welche sich bei unterschiedlicher Entfernung zur Kamera mit gleicher Geschwindigkeit

fortbewegen, eine unterschiedliche Pixeldistanz zurücklegen. Nähere Objekte legen aus Sicht der Kamera eine größere Pixeldistanz zurück.
Ein Beispiel für solch eine Segmentierung ist in Abbildung 14 zu sehen.

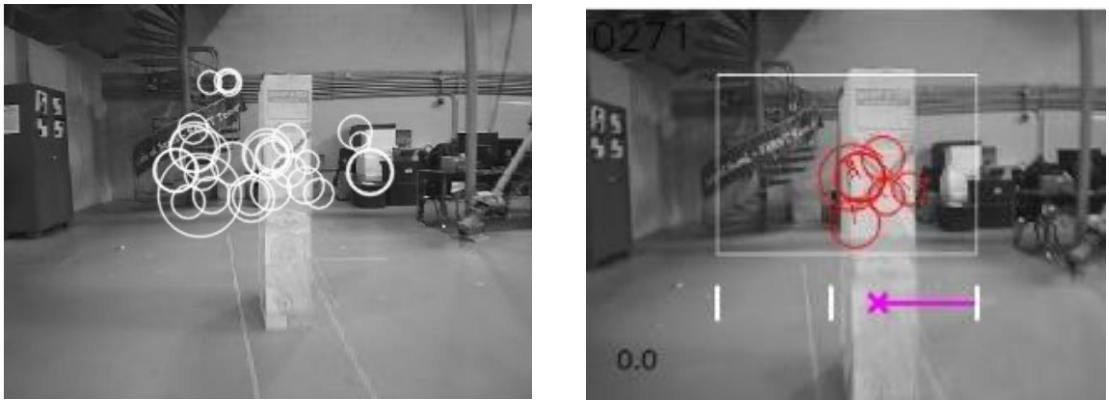
Abbildung 14: Segmentierung mittels optischen Flusses
(rot: nah und blau: fern) [14]

In der Abbildung 14 legen die roten Merkmale eine größere Pixeldistanz als die blauen zurück. Daraus lässt sich schließen, dass diese eine geringere Distanz zur Kamera aufweisen. Neben der Berechnung des optischen Flusses wird mittels Template Matching das Ausmaß der Expansion von Objekten berechnet. Hierfür werden von dem vorherigen Bild Ausschnitte der detektierten Objekte in verschiedenen Skalierungsgrößen erstellt. Diese werden mit dem aktuellen Bild verglichen. Bei den gefundenen Bildausschnitten spiegelt die Skalierungsgröße das Ausmaß der Expansion wider. Durch die Berechnung des optischen Flusses und der Berechnung der Expansion von einzelnen Objekten lassen sich Hindernisse erkennen. Ein Beispiel hierfür ist in Abbildung 15 zu sehen. Der oben beschriebene Ansatz funktioniert nur bei Bewegung der Kamera bzw. bei Bewegung einzelner Objekte. Bei einzelnen oder statischen Bildern können keine Hindernisse detektiert werden.

Abbildung 15: Kennzeichnung der Hindernisse
(rötlich: nah und bläulich: fern) [14]

2.10 Hinderniserkennung durch Template Matching

In dem Artikel von Mori und Scherer [16] wird eine Methode vorgestellt, mit welcher durch die Expansion detektierter Merkmale Hindernisse detektiert werden, um einer Drohne ein Ausweichen zu ermöglichen. Für diese Methode werden Bilder während des Fluges aufgenommen. Die Kamera ist an der Drohne befestigt und zeigt in dessen Flugrichtung.
Bei der beschriebenen Methode wird der SURF-Merkmalsdetektor von Bay et al. [17] verwendet, um markante Pixel in einer Szene zu erkennen. Diese Merkmale werden in jedem Bild, welches die Kamera aufnimmt, gesucht. Nachdem die Merkmale in zwei aufeinanderfolgenden Bildern detektiert wurden, werden Merkmale von dem jetzigen Bild mit den Merkmalen von dem vorherigen Bild gepaart. Es wird nach identischen Merkmalen in beiden Bildern gesucht. Anschließend werden alle Merkmale, welche nicht größer geworden sind, entfernt. Daraufhin werden Bildausschnitte um die übrigen Merkmalspaare von dem vorherigen und von dem jetzigen Bild erstellt. Die Bildausschnitte des vorherigen Bildes werden in verschiedenen vordefinierten Größen skaliert.
Dieser wird in jeder Skalierung mit dem passenden Bildausschnitt des jetzigen Bildes mittels Template Matching verglichen. Die Skalierung mit der höchsten Übereinstimmung wird als Maß für die Expansion dieses Merkmals verwendet. Sollte die Expansion über einem bestimmten Wert liegen, wird das Merkmal als Hindernis betrachtet.
In der folgenden Abbildung 16 sind Beispielaufnahmen der Hindernisdetektion abgebildet. Die gefundenen Merkmale werden als Kreise dargestellt. Die Größe der Kreise gibt die Größe der Merkmale wieder. Rote Kreise sind Merkmale, deren Expansion über dem Schwellwert liegt. Diese Kreise werden als Hindernis betrachtet.



(a) (b)

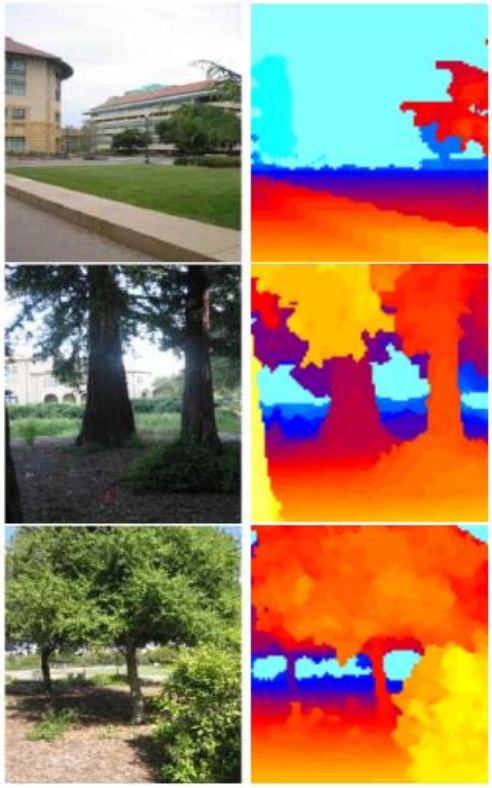
Abbildung 16: Hinderniskennzeichnung a) gefilterte Merkmale (weiße Kreise) b) als Hindernisse eingestufte Merkmale (rote Kreise) [16]

In dem Artikel wird ein Algorithmus zum Ausweichen von detektierten Hindernissen beschrieben. Auf diesen wird jedoch nicht weiter eingegangen.

Der Algorithmus wurde getestet, indem die Drohne durch einen Parkour geflogen wurde. In diesem Parkour wurden mehrere Zylinder aufgestellt, welchen die Drohne ausweichen sollte. Die Drohne konnte insgesamt 104 von 107 Hindernissen ausweichen.
Durch diese Methode werden Hindernisse detektiert und die Zeit bis zu einer möglichen Kollision über die Expansion detektierter Merkmale geschätzt. Es werden jedoch keine exakten Werte für die Tiefe berechnet. Die Genauigkeit dieser Methode hängt von dem Finden von genügend passenden Merkmalen in aufeinanderfolgenden Bildern ab.

2.11 Tiefe durch parametrisches Lernen

Saxena et al. [18] stellen einen Ansatz für die Generierung von Tiefendaten durch parametrisches Lernen vor. Der Ansatz verwendet einzelne monokulare Bilder und erstellt für diese eine Tiefenkarte. Die Grundidee ist, dass für die Berechnung der Tiefe zum einen lokale Information an bestimmten Stellen im Bild verwendet werden und zum anderen die Beziehungen zwischen den einzelnen Stellen betrachtet werden. Beispiele für die mit dem Ansatz generierten Tiefenkarten und die dazugehörigen Eingangsbilder sind in Abbildung 17 zu sehen.



(a) (b)

Abbildung 17: Tiefenberechnung durch parametrisches Lernen
a) Eingangsbilder b) die dazugehörigen Tiefenkarten [18]

Um die lokalen Informationen zu verwerten, wird das Eingangsbild in kleine Bildausschnitte, auch „*patches*" genannt, unterteilt. Diese werden in unterschiedlichen Auflösungen bzw. Skalierungsgrößen generiert. In jeder Skalierungsgröße werden Bildmerkmale von unterschiedlichen Merkmalstypen aufgenommen. Beispiele für die verwendeten Merkmalstypen sind Texturen, Verdeckungen und Ecken. Die aufgenommenen Bildmerkmale und die dazugehörigen Skalierungsgrößen werden in Vektoren gespeichert.

Um die globalen Informationen zu verarbeiten, wird das „*Markov Random Field"* - Modell (MRF-Modell) verwendet. Mit diesem Modell werden die Beziehungen bzw. Zusammenhänge zwischen direkten Nachbarbildausschnitten und die Beziehungen zwischen nicht-direkten Nachbarbildausschnitten in unterschiedlichen Auflösungen berechnet. Hierdurch lassen sich komplexe Zusammenhänge wie z. B. die Struktur eines Baumes zusammenfassen. Unter der alleinigen Verwendung von lokalen Informationen ist dies nicht möglich. Es werden zwei Varianten des MRF-Modells verwendet, das Gaußsche MRF- und das Laplace'sche MRF-Modell.

Der beschriebene Ansatz wurde durch überwachtes maschinelles Lernen trainiert. Die Trainingsdaten beinhalten 425 monokulare Bilder mit den dazugehörigen Tiefenkarten.

2.12 Tiefe durch nicht-parametrisches Lernen

Karsch et al. [19] beschreiben ein Verfahren zur Generierung einer Tiefenkarte eines 2D-Bildes oder 2D-Videos für die 3D-Visualisierung. Der Ansatz basiert auf nicht-parametrischem Lernen. Für die beschriebene Methode müssen keine Parameter oder explizite Annahmen vordefiniert werden. Um das System zu trainieren, werden Trainingsdaten mit zwei Microsoft-Kinect-Kameras aufgenommen. Die Trainingsdaten beinhalten Videos und die dazugehörigen Tiefenkarten.

Grundsätzlich beruht der Ansatz auf der Idee, dass ähnliche Bilder auch ähnliche Tiefen besitzen. Daher werden aus der Datensammlung sieben Bilder, die dem Eingangsbild am ähnlichsten sind, mithilfe der „*GIST*"-Methode von Oliva und Torralba [20] ausgewählt. Da die ausgewählten Bilder aus der Datensammlung perspektivisch nicht in Gänze mit dem Eingangsbild übereinstimmen, wird die „*SIFT Flow*"-Methode von Liu et al. [21] auf alle ausgewählten Bilder angewendet. Die „*SIFT Flow*"-Methode richtet ein Bild unabhängig von der Perspektive und der Beleuchtung aus. Dabei werden gemeinsame Merkmale in zwei Bildern verglichen. Die Tiefenkarten der ausgerichteten sieben Bilder werden mithilfe eines Optimierungsprozesses zu einer dem Eingangsbild entsprechenden Tiefenkarte interpoliert und geglättet. In Abbildung 18 ist ein Eingangsbild und die dazugehörige Tiefenkarte zu sehen.

Mit dem vorgestellten Ansatz werden von Bildern und Videos Tiefenkarten erstellt. Diese Methode berechnet jedoch keine exakten Werte für die Tiefe.

(a) (b)

Abbildung 18: Tiefenberechnung durch nicht-parametrisches Lernen
a) Eingangsbild b) die dazugehörige Tiefenkarte [19]

2.13 Tiefe durch unüberwachtes maschinelles Lernen

In dem Artikel von Godard et al. [22] wird ein maschinelles Netzwerk vorgestellt, welches mittels unüberwachtem Lernen trainiert wird. Dieses Netzwerk erstellt aus Bildern, welche über eine monokulare Kamera aufgenommen werden, Tiefenkarten.
Bei dem Training dieses Netzwerks werden zwei einzelne Netze verwendet. Ein Netzwerk, welches die Tiefenkarte *Id* eines Eingangsbildes *It* ausgibt und ein anderes Netz, welches das gleiche Eingangsbild *It* und zusätzlich die nahe liegenden Bilder *It-1* und *It+1* verwendet, um die jeweilige Pose der Kamera auszugeben. Es wird die Änderung der Pose vom Zeitpunkt *t* zum Zeitpunkt *t+1* und vom Zeitpunkt *t-1* zum Zeitpunkt *t* ausgegeben. Mithilfe der generierten Tiefenkarte *Id* und der jeweiligen Pose werden die Eingangsbilder *It-1* und *It+1* rekonstruiert.
Die beiden Netzwerke werden durch die Minimierung des Rekonstruktionsfehlers trainiert. Das Netzwerk, welches die Posen ausgibt, wird nur während des Trainings benötigt.
In dem Artikel werden drei Arten von Problemen und die dazugehörigen Lösungen beschrieben. Im Folgenden wird kurz auf diese eingegangen.
Durch Pixel, welche aufgrund der Bewegung der Kamera oder einer auftretenden Verdeckung nicht in allen der Eingangsbilder zu sehen sind, kann der Rekonstruktionsfehler verfälscht werden. Um dem entgegenzuwirken, wird eine neu entworfene Fehler-Funktion vorgestellt. Diese verringert die Bildung von Artefakten, verbessert die Schärfe von verdeckten Kanten und führt zu einer verbesserten Genauigkeit.
Ein weiteres Problem stellen stillstehende Objekte dar. Falls solche Objekte während des Trainings in Bewegung waren, wird für diese eine unendliche Tiefe berechnet. Deswegen werden Pixel ignoriert, welche sich von einem Bild zum nächsten nicht ändern. Dadurch können Objekte ignoriert werden, welche sich mit der gleichen Geschwindigkeit der Kamera fortbewegen. Zusätzlich werden ganze Bilder ignoriert, falls sich die Kamera nicht mehr bewegt.
Zum Schluss wird eine Methode vorgestellt, welche die Verzögerung des Trainings durch lokale Minima verhindert. In der folgenden Abbildung 19 wird ein Beispiel für eine generierte Tiefenkarte und das zugehörige Eingangsbild dargestellt.

(a)

(b)

Abbildung 19: Tiefenberechnung durch unüberwachtes maschinelles Lernen a) Eingangsbild b) die dazugehörige Tiefenkarte [22]

In der Tiefenkarte werden nahe Bereiche hell und ferne Bereiche dunkel dargestellt.

Bei dem vorgestellten Ansatz werden für reflektierende Objekte und Objekte mit komplizierten Formen ungenaue Tiefenwerte berechnet.

2.14 Tiefe durch überwachtes maschinelles Lernen

In dem Artikel von Chiu et al. [23] wird die Berechnung von Tiefenkarten über ein maschinelles Netzwerk, welches durch überwachtes Lernen trainiert wird, realisiert. Bei dem Netzwerk handelt es sich um ein „*Convolutional Neural Network*“ mit einer Encoder-Decoder Struktur. Das gesamte Netzwerk besitzt 0,32 Millionen Parameter und wird mit dem öffentlich zugänglichen „*KITTI“*-Datensatz trainiert.
Es wird aus dem Bild einer monokularen Kamera sowohl eine Tiefenkarte als auch eine Segmentierungskarte generiert. In der Segmentierungskarte werden einzelne Objekte mit unterschiedlichen Farben dargestellt. Bei der Tiefenkarte wird eine Farbpalette verwendet, in der helle Farben zu nahen und dunkle Farben zu fernen Bereichen korrespondieren.
Die Bilder, welche für das Training verwendet werden, wurden in Straßen und Städten aufgenommen. Für das Training des Netzwerks werden Bilder einer monokularen Kamera als Modell-Eingang und die dazugehörigen Tiefen- und Segmentierungskarten, mit welchen die Ausgänge des Netzwerks verglichen werden, verwendet. In der verwendeten Fehler-Funktion werden sowohl der Tiefenfehler als auch der Segmentierungsfehler als gewichtete Summe berücksichtigt.
Als Encoder wird der bereits trainierte Encoder des “*MobileNetV2*” von Sandler et al. [24] verwendet. Der Encoder besteht aus einem 3x3 „*Convolution-Layer*“, gefolgt von 17 „*Bottlenecks*“.
Der Decoder des Netzwerks besitzt mehrere „*Upscaling-Layer*“, um die „*Feature-Maps*“ zu vergrößern und die Auflösung der Tiefenkarten zu erhöhen. Der Decoder besteht aus insgesamt vier Regionen. Jede dieser Regionen besteht aus einem „*Pixel-Shuffle-Layer*“ und zwei bis vier „*Bottlenecks*“. In der folgenden Abbildung 20 wird ein Eingangsbild und die dazugehörige Tiefen- sowie Segmentierungskarte dargestellt.

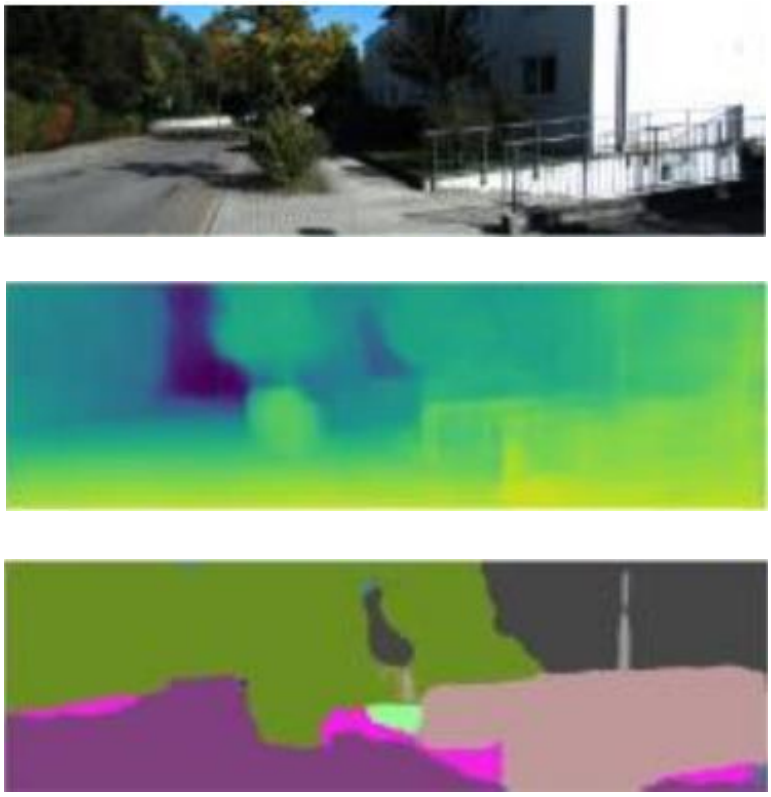


Abbildung 20: Tiefenberechnung und Segmentierung durch überwachtes maschinelles Lernen a) Eingangsbild b) die dazugehörige Tiefenkarte c) die dazugehörige Segmentierungskarte [23]

Aufgrund der geringen Anzahl an Parametern erreicht das Netzwerk bei einer Eingangs-Bildgröße von 480 x 320 und einer Ausgangs-Bildgröße von 240 x 160 Pixeln bei Verwendung einer „*NVIDIA GTX 1060*“ 121,6 Bilder pro Sekunde und bei der Berechnung auf dem „*NVIDIA Jetson TX2*“ 33,5 Bilder pro Sekunde.

3 Konzeption

Nachdem die recherchierten Ansätze vorgestellt sind, werden in diesem Kapitel die Ansätze ausgewählt, die am besten für den Anwendungsfall dieser Arbeit geeignet sind. Dazu werden zunächst Kriterien festgelegt, die durch die Aufgabenstellung gegeben sind und mit den Randbedingungen übereinstimmen. Anschließend wird aus den recherchierten Ansätzen durch Betrachtung der jeweiligen Vor- und Nachteile eine passende Auswahl getroffen. Die Methoden der ausgewählten Ansätze werden in dem Kapitel 4 näher erläutert.

3.1 Bewertungskriterien

Um eine passende Auswahl an Ansätzen auszuwählen, werden im Folgenden Kriterien beschrieben, nach welchen diese zu bewerten sind.
Mithilfe der ausgewählten Ansätze sollen herabhängende Hindernisse, wie in Kapitel 1.1 beschrieben, detektiert werden. Für die Detektion dieser Hindernisse soll eine einfache monokulare Kamera verwendet werden. Diese Kamera wird auf den Prototypen montiert und muss möglichst klein und kompakt sein, um bei der Benutzung des „*Shared Guide Dog 4.0“* nicht zu stören. Damit detektierten Hindernissen ausgewichen werden kann, ist eine Trennung von herannahenden und fernen Hindernissen erforderlich. Hierfür sind entweder exakte Tiefenwerte oder eine Detektion von herannahenden Objekten erforderlich. Die Berechnung soll lokal auf dem Prototypen erfolgen. Der Rechenaufwand und die daraus resultierende Berechnungszeit muss niedrig sein, um ein rechtzeitiges Ausweichen zu ermöglichen. Es wird davon ausgegangen, dass die Berechnung von 10 Bildern pro Sekunde hierfür ausreicht. Im Folgenden werden die festgelegten Kriterien in einer Liste zusammengefasst.

- Detektion von herabhängenden Hindernissen
- Verwendung einer monokularen Kamera
- Trennung von herannahenden und fernen Objekten
- niedrige Berechnungsdauer von 10 Bildern pro Sekunde

3.2 Festlegung der zu implementierenden Ansätze

Nachdem die einzelnen Bewertungskriterien festgelegt sind, werden die recherchierten Ansätze aus dem Kapitel 2 hinsichtlich dieser und der jeweiligen Vor- und Nachteile bewertet. Daraufhin werden die geeignetsten Ansätze ausgewählt.

3.2.1 Detektion von herabhängenden Hindernissen

Bei dem beschriebenen Ansatz [13] werden Randbedingungen für die Detektion von Hindernissen festgelegt. Zu diesen Randbedingungen gehört unter anderem, dass keine herabhängenden Hindernisse vorkommen dürfen. Aus diesem Grund wird dieser Ansatz nicht weiter betrachtet.
In keinem anderem der recherchierten Ansätze wird die Detektion oder die Berechnung der Tiefe von herabhängenden Hindernissen evaluiert. Deshalb kann die Eignung hinsichtlich dieses Bewertungskriteriums bei den übrigen Ansätzen nur durch Testen überprüft werden.

3.2.2 Verwendung einer monokularen Kamera

In jedem der recherchierten Ansätze werden Bilder oder Videos einer monokularen Kamera verwendet, um Hindernisse zu detektieren oder um Tiefeninformationen zu berechnen.

3.2.3 Trennung von herannahenden und fernen Objekten

Der Ansatz aus dem Artikel [2] berechnet keine exakte Tiefe. Objekten werden anhand der Position im Bild vordefinierte Tiefen zugewiesen. Die Objekte im unteren Teil des Bildes werden als nah und die Objekte im oberen Teil des Bildes als fern deklariert.
Die Methode aus [3] eignet sich nur für die Tiefenberechnung bei Nahaufnahmen, da nur bei solchen der Unterschied der Unschärfe im Bild groß genug ist. Bei der Methode aus [5] wird eine kodierte Maske verwendet, die in

die Linse gelegt wird, um einen ausreichend großen Unterschied der Unschärfe im Bild zu erreichen. Dadurch kann eine exakte Tiefenkarte berechnet werden.
Fan et al. [6] berechnen die Tiefe bzw. die Form von Objekten anhand von Variationen in der Schattierung. Diese Methode wird meist bei endoskopischen Operationen verwendet, bei denen sehr gute Lichtverhältnisse vorhanden sind. Eine Evaluierung im Freien wird nicht beschrieben.
In [8] wird die Form eines Objektes anhand der Texturmerkmale auf der Oberfläche rekonstruiert. Die Tiefe kann nur bei Objekten mit ausreichender Anzahl an Texturmerkmalen berechnet werden.
Mit der Methode aus [11] können nahe und ferne Objekte anhand von Kantenverdeckungen getrennt werden. Eine Aussage über herannahende Objekte kann nicht getroffen werden.
In der Methode aus Artikel [12] wird eine grobe Tiefenkarte ohne exakte Tiefenwerte durch geometrische Perspektive und farbliche Segmentierung generiert. Für die Berechnung der Tiefe ist eine hohe Distanz zwischen der Kamera und den Objekten erforderlich.
J. M. Sagar [14] und Mori und Scherer [16] beschreiben jeweils eine Methode für die Hinderniserkennung einer Drohne. In [16] werden herannahende Objekte durch das Ausmaß der Expansion von Bildausschnitten detektiert. Bei [14] wird neben dem Ausmaß der Expansion zusätzlich der optische Fluss berechnet.
Der Ansatz aus [18] berechnet Tiefenkarten mit exakten Tiefenwerten durch parametrisches Lernen. Dabei werden zwei verschiedene Modelle verglichen.
In dem Ansatz aus [19] wird eine Methode für die Generierung von Tiefenkarten durch nicht-parametrisches Lernen vorgestellt. Bei diesem Ansatz werden jedoch keine exakten Werte für die Tiefe berechnet.
Die Ansätze aus den Artikeln [22] und [23] generieren Tiefenkarten basierend auf maschinellem Lernen. Der Ansatz aus [22] verwendet dabei unüberwachtes und der Ansatz aus [23] überwachtes maschinelles Lernen. Beide Ansätze generieren exakte Tiefenwerte.
Die Ansätze aus den Artikeln [2], [3], [11], [12] und [19] können aufgrund der zuvor genannten Gründe nicht für den Anwendungsfall dieser Arbeit verwendet werden. Diese Ansätze werden bei den weiteren Kriterien nicht weiter betrachtet.

3.2.4 Niedrige Berechnungsdauer

Die Ansätze aus [5] und [6] weisen eine hohe Berechnungsdauer auf, dadurch ist ein rechtzeitiges Ausweichen nicht möglich. Die Methode aus [5] wurde zwar auf Hardware aus dem Jahr 2012 evaluiert, jedoch muss die Einsparung der Berechnungsdauer auf moderner Hardware überprüft werden. In [6] wird die Verwendung einer Bildpyramide empfohlen, um die Berechnungszeit zu verringern. Inwieweit sich diese dadurch verringern lässt, ist nicht beschrieben. Die Berechnungszeit der Methoden aus den Artikeln [14], [16] und [23] ist niedrig genug, um ein rechtzeitiges Ausweichen zu ermöglichen.
In den Artikeln [8], [18] und [22] wird die Berechnungszeit nicht beschrieben.
Aufgrund der hohen Berechnungsdauer der Ansätze [5] und [6] können diese nicht für den Anwendungsfall dieser Arbeit verwendet werden.

3.2.5 Auswahl

Im Folgenden werden aus den verbleibenden Ansätzen aus den Artikeln [8], [14], [16], [18], [22] und [23] die geeignetsten Methoden ausgewählt.
Für den Ansatz aus dem Artikel [8] werden genügend gleiche Texturen auf den zu detektierenden Objekten benötigt. Diese kommen jedoch im Freien nicht vor. Außerdem wird bei dieser Methode nur die Form einzelner Objekte rekonstruiert.
Die Ansätze aus den Artikeln [18], [22] und [23] benötigen Trainingsdaten, um die Tiefe in Bildern zu berechnen. Eine genügende Anzahl an Trainingsdaten zu generieren, ist in dem Zeitrahmen dieser Arbeit nicht möglich.
Abhängig der festgelegten Kriterien und den erfolgreich durchgeführten Experimenten in den Artikeln [14] und [16] eignen sich diese Ansätze am besten für den Anwendungsfall dieser Arbeit.
Da die Ansätze aus [14] und [16] lediglich auf einer Schätzung der Distanz zu Hindernissen basieren, wird zusätzlich ein dritter Ansatz implementiert, durch den exakte Tiefeninformationen mittels optischen Flusses generiert werden. Diese Methode wird in dem Kapitel 4.1.3 genauer erläutert.

3.3 Generierung der Testdaten

Weitere Aspekte zur Realisierung der Ansätze werden nachstehend näher erläutert. Für die Implementierung der vorgestellten Ansätze werden monokulare Bilder bzw. Videos benötigt. Zusätzlich sind für den dritten Ansatz die extrinsischen und intrinsischen Parameter der Kamera erforderlich. Für die Berechnung der Tiefe sind Informationen für die translatorische und rotatorische Bewegung der Kamera bzw. des Prototypen sowie die Brennweite der Kamera nötig.

Die Videos, die als Bildsequenzen in den Programmabläufen verarbeitet werden, werden parallel mit einer Webcam und einer Mobiltelefonkamera aufgenommen. Hierbei wird eine handelsübliche Webcam des Modells „*Jelly Comd W06*“ mit einer Auflösung von 1920 x 1080 Pixeln und einer Bildrate von 30 Bildern pro Sekunde genutzt. Dieses Modell besitzt keine eigenständige optische Bildstabilisierung.
Bei der Mobiltelefonkamera handelt es sich um eine Kamera mit einer Auflösung von bis zu 3840 x 2160 Pixeln und einer Bildrate von 60 Bildern pro Sekunde. Diese ist im „*Apple iPhone 12*“ mit einer integrierten optischen Bildstabilisierung verbaut. Die Videos werden jedoch mit einer Auflösung von 1920 x 1080 Pixeln und einer Bildrate von 30 Bildern pro Sekunde aufgenommen. Beide Aufnahmegeräte werden, wie auf der Abbildung 21 ersichtlich, an den Prototypen in Laufrichtung befestigt. Die Aufnahmen werden im Lohmühlenpark in 20099 Hamburg gemacht. Dabei werden insbesondere herabhängende Gegenstände, welche in Kapitel 1.1 beschrieben werden, aufgenommen.

Aus Vereinfachungsgründen und zu Testzwecken werden Live-Aufnahmen des aufgenommenen Bildmaterials mithilfe der Software „*OBS Studio*“ der Version 26.1.1 simuliert. Hierdurch können die Implementierung sowie das Testen unabhängig von dem Ort, an dem die Videos aufgenommen wurden, durchgeführt werden. Zudem wird sichergestellt, dass alle Ansätze mit den gleichen Bildsequenzen getestet werden.

Eine Messung der rotatorischen und translatorischen Bewegung wird mit einer inertialen Messeinheit (IMU) des Modells „*Adafruit 9-DOF Absolute Orientation IMU Fusion Breakout - BNO055*“ durchgeführt. Dieser Sensor wird ebenfalls an den Prototypen angebracht und ist in Abbildung 21 dargestellt.

Abbildung 21: Prototyp des „*Shared Guide Dog 4.0*“

Im Folgenden werden die Daten, die die inertiale Messeinheit aufnehmen kann, und die dazugehörigen Aufnahmeraten angegeben.

- **Absolute Orientierung** (Euler-Winkel, 100 Hz)
- **Absolute Ausrichtung** (Quaternion, 100 Hz)
- **Winkelgeschwindigkeitsvektor** (100 Hz)
- **Beschleunigungsvektor** (100 Hz)
- **Magnetfeldstärkevektor** (20 Hz)
- **Beschleunigungsvektor (ohne Erdbeschleunigung)** (100 Hz)
- **Gravitationsvektor** (100 Hz)
- **Temperatur** (1 Hz)

Für den Ansatz der Berechnung von Tiefenkarten sind nur die Informationen aus dem Winkelgeschwindigkeitsvektor und aus dem Beschleunigungsvektor (ohne Erdbeschleunigung) erforderlich. Der Sensor liefert Werte für die Winkelgeschwindigkeit in der Einheit rad/s. Für die Berechnung ist jedoch die Einheit 1/s erforderlich. Die Werte werden umgerechnet, indem diese durch 2π dividiert werden. Für die Berechnung ist außerdem die translatorische Geschwindigkeit erforderlich. Deshalb werden die Werte des Beschleunigungsvektors mit der Formel $v = a * \Delta t + v_0$ umgerechnet. Dabei

entspricht v der Geschwindigkeit zum Zeitpunkt t, a der Beschleunigung zum Zeitpunkt $t-1$, Δt der Zeitdifferenz zwischen beiden Aufnahmen der Werte und v_0 der Geschwindigkeit zum Zeitpunkt $t-1$.
Die aufgenommenen Messungen des Sensors und die aufgenommenen Videos werden synchronisiert, indem ein akustisches Startsignal erzeugt wird. Dieses Signal wird beim Beginn der Messung erzeugt und kennzeichnet somit im aufgenommenen Video eindeutig den Zeitpunkt des Messbeginns.
Die Brennweite der Kamera wird mithilfe des Programms „*calibrate_camera*“ aus der OpenCV-Bibliothek ermittelt. Die Kamerakalibrierung wird in Kapitel 4.5 näher erläutert.

4 Methoden

In Kapitel 3 werden die zu implementierenden Ansätze aufgeführt. Bei der Implementierung werden hauptsächlich Methoden aus der OpenCV-Bibliothek verwendet. Die Methoden, welche für das Verständnis der zuvor ausgewählten Ansätze erforderlich sind, werden in den folgenden Unterkapiteln beschrieben. Eine nähere Erläuterung aller verwendeten OpenCV-Methoden ist in der OpenCV-Dokumentation [1] vorhanden.

4.1 Optischer Fluss

Im Folgenden werden zwei der zu implementierenden Methoden erläutert. Es wird auf den Lucas-Kanade-Algorithmus und auf den Farnebäck-Algorithmus zu der Berechnung des optischen Flusses von Pixeln zwischen aufeinanderfolgenden Bildern eingegangen. Zusätzlich wird eine Formel umgestellt, die es ermöglicht, mithilfe der Informationen des optischen Flusses Tiefe von Objekten in Bildern zu berechnen.

4.1.1 Lucas-Kanade-Algorithmus

Die Implementierung des Lucas-Kanade-Algorithmus zu der Berechnung des optischen Flusses wird in dem Artikel [25] beschrieben. Ziel dieses Algorithmus ist es, einen bestimmten Punkt in einem Bild in einem darauffolgenden Bild wiederzufinden. Es wird angenommen, dass sich dieser Punkt zwischen diesen beiden Bildern verschiebt und eine affine Transformation durchläuft. Um den Punkt in dem nachfolgenden Bild wiederzufinden, wird eine Funktion minimiert. Die Unbekannten in dieser Funktion sind die Verschiebung des Punktes zwischen den Bildern und die Parameter der affinen Transformationsmatrix. In dieser Funktion kann die Größe eines Integrationsfensters frei bestimmt werden. Es wird beschrieben, dass ein kleines Integrationsfenster verhindert, dass Details im Bild verloren gehen und dadurch die Genauigkeit erhöht wird. Ein größeres Integrationsfenster hingegen führt dazu, dass der Punkt auch bei größeren Bewegungen wiedergefunden werden kann. Damit nicht zwischen diesen Eigenschaften gewählt werden muss, wird in [25] der Lucas-Kanade-Algorithmus iterativ mithilfe einer Bildpyramide implementiert.

Die Implementierung dieses Algorithmus ist in der OpenCV-Bibliothek unter der Methode „*calcOpticalFlowPyrLK*“ vorhanden. Diese Methode berechnet durch Eingabe von zwei aufeinanderfolgenden Bildern sowie detektierten Merkmalen auf dem vorherigen Bild die Positionen dieser Merkmale auf dem nachfolgenden Bild. Die aufeinanderfolgenden Bilder werden der Methode mit den Eingangsparametern „*prevImg*“ und „*nextImg*“ übergeben und die detektierten Merkmale werden mit dem Eingangsparameter „*prevPts*“ übergeben. Über zusätzliche Parameter werden die Eigenschaften der Berechnung festgelegt. Durch den Parameter „*maxLevel*“ wird festgelegt, wie

viele Ebenen einer Bildpyramide verwendet werden. Über den Parameter „*winSize*“ wird die Größe des Integrationsfensters in jeder Stufe der Bildpyramide bestimmt. Mit dem Parameter „*criteria*“ werden Abbruchkriterien wie z. B. die maximale Anzahl an Schleifendurchgängen für die Berechnung der Position der Merkmale auf dem nachfolgenden Bild festgelegt. Durch den Eingangsparameter „*flags*“ wird festgelegt, welches Fehlermaß verwendet wird. Der letzte Eingangsparameter „*minEigThreshold*“ legt einen Schwellwert für die in dem Algorithmus berechneten Eigenwerte fest.

Die Ausgabe der wiedergefundenen Merkmale im nachfolgenden Bild erfolgt über die Parameter „*nextPts*“ und „*status*“. Die Parameter „*nextPts*“ und „*prevPts*“ sind Arrays von 2D-Koordinaten. Jeder Ausgangsparameter ist ein Array und besitzt die gleiche Größe wie „*prevPts*“. Ein Merkmal, welches im vorherigen Bild detektiert wurde, wird an einer Stelle in „*prevPts*“ gespeichert. Sollte die Berechnung der Position dieses Merkmals im nachfolgenden Bild erfolgreich sein, ist an der gleichen Stelle in „*status*“ eine „1“ vorhanden und die Position des Merkmals im nachfolgenden Bild wird an der gleichen Stelle in „*nextPts*“ gespeichert. Sollte die Berechnung nicht erfolgreich sein, steht an dieser Stelle in „*status*“ eine „0“. In dem Ausgangsparameter „*err*“ wird zu jeder erfolgreich berechneten Merkmalsposition der Berechnungsfehler, dessen Typ durch „*flags*“ festgelegt ist, an der korrespondierenden Stelle gespeichert. Sollte die Berechnung der Position eines Merkmals nicht erfolgreich sein, wird kein Wert angegeben.

4.1.2 Farnebäck-Algorithmus

Der im Folgenden beschriebene Algorithmus aus dem Artikel [26] berechnet anders als der im Kapitel 4.1.1 beschriebene Algorithmus den optischen Fluss in aufeinanderfolgenden Bildern für alle Pixel. Dazu wird für jeden Pixel in einem Bild ein quadratisches Polynom gebildet. Dieses Polynom beschreibt die Beziehung eines Pixels zu den benachbarten Pixeln. Die Koeffizienten der Polynome werden über die Methode der kleinsten Quadrate geschätzt. Es werden Polynome für alle Pixel in zwei aufeinanderfolgenden Bildern gebildet. Anschließend können die Polynome in den zwei Bildern miteinander verglichen werden. Die Verschiebung von der Position eines Pixels in dem ersten Bild zu einer anderen Position im zweiten Bild wird iterativ über die Minimierung einer Funktion berechnet. Um den Berechnungsfehler bei großen Bewegungen zu reduzieren, werden die zuvor berechneten Verschiebungen in der

nachfolgenden Berechnung als Schätzung für die Positionen von Pixeln verwendet. Um die Genauigkeit bei größeren Bewegungen zusätzlich zu erhöhen, wird eine Bildpyramide verwendet. Die Ergebnisse der Berechnung von kleineren Skalierungen werden als Startwerte für die Berechnung von größeren Skalierungen verwendet.

Die Implementierung dieses Algorithmus ist in der OpenCV-Bibliothek vorhanden. Der Algorithmus wird über das Aufrufen der Methode „*calcOpticalFlowFarneback*“ verwendet. Mit den Eingangsparametern „*prev*“ und „*next*“ werden der Methode zwei aufeinanderfolgende Grauwertbilder übergeben. Diese Bilder müssen die gleiche Größe besitzen. In dem Parameter „*flow*“ werden die berechneten Verschiebungen der Pixel gespeichert. Der Parameter „*levels*“ legt die Anzahl an Ebenen in der Bildpyramide fest und der Parameter „*pyr_scale*“ die dazugehörige Skalierung der einzelnen Ebenen. Mit dem Parameter „*winsize*“ wird die Größe des Integrationsfensters festgelegt. Die Anzahl der Iterationen, die der Algorithmus in jeder Ebene der Bildpyramide durchläuft, wird über den Parameter „*iterations*“ festgelegt. Die Anzahl von benachbarten Pixeln, die bei der Bildung der Polynome betrachtet werden, wird über den Parameter „*poly_n*“ bestimmt. Mit dem Parameter „*poly_sigma*“ wird der Wert der Standardabweichung des Gauß-Filters festgelegt. Durch den Parameter „*flags*“ wird bestimmt, ob die in dem Parameter „*flow*“ gespeicherten Verschiebungen als Startwerte für nachfolgende Berechnungen verwendet werden. Zudem kann auch festgelegt werden, ob ein Gaußfilter statt eines Boxfilters verwendet wird.

4.1.3 Tiefe durch optischen Fluss

Im Folgenden wird die Berechnung von Tiefenwerten durch optischen Fluss vorgestellt. Diese Tiefenwerte entsprechen der Distanz zwischen der Kamera und Punkten im drei-dimensionalen Raum, die durch Pixel auf der Bildebene abgebildet werden. Für diese Berechnung wird die folgende Gleichung (1) aus dem Artikel [27] umgestellt.

$$\begin{pmatrix} u_x \\ u_y \end{pmatrix} = \frac{1}{Z(x,y)} \cdot \boldsymbol{A}(x,y) \cdot \boldsymbol{t} + \boldsymbol{B}(x,y) \cdot \boldsymbol{\omega} \tag{1}$$

Hierfür werden die rechtsdrehenden Koordinatensysteme aus [27] verwendet. Diese sind in der folgenden Abbildung 22 dargestellt.

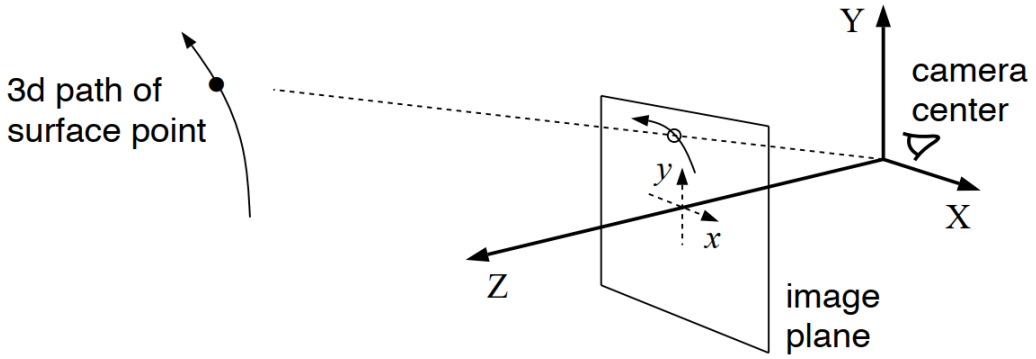


Abbildung 22: Verwendete Koordinatensysteme [27]

Die Gleichung (1) gibt den Zusammenhang zwischen der Geschwindigkeit eines dreidimensionalen Punktes und der korrespondierenden Geschwindigkeit eines Pixels, der diesen Punkt auf der Bildebene abbildet, wieder. Die Parameter u_x und u_y sind die Geschwindigkeiten eines Pixels. u_x ist dabei die x-Komponente und u_y die y-Komponente der Geschwindigkeit. Die Einheit dieser Geschwindigkeiten ist Pixel/s. Die Pixelgeschwindigkeit kann als die zeitliche Positionsänderung eines Merkmals auf einem Bild zu einem nachfolgenden Bild betrachtet werden. Die Pixelgeschwindigkeit kann berechnet werden, wenn die Positionsänderung eines Merkmals zwischen zwei aufeinanderfolgenden Bildern und die Zeit zwischen dem Aufnehmen dieser Bilder bekannt sind. Die Pixelgeschwindigkeit ist die Geschwindigkeit, die der Pixel zum Zeitpunkt der Aufnahme des nachfolgenden Bildes besitzt.

Der Parameter $Z(x, y)$ entspricht der Distanz zwischen der Kamera und einem Punkt im drei-dimensionalen Raum, welcher durch einen Pixel auf der Position (x, y) in der Bildebene des nachfolgenden Bildes abgebildet wird. Wenn zwei Objekte mit unterschiedlicher Distanz zu einer Kamera sich mit identischer Geschwindigkeit in die gleiche Richtung im dreidimensionalen Raum bewegen, legt das nähere Objekt eine größere Distanz auf der Bildebene zurück als das

weiter entfernte Objekt. Aus diesem Grund ist die Distanz umgekehrt proportional zur Pixelgeschwindigkeit. Die Einheit der Distanz ist m.

Der Vektor $\boldsymbol{t} = (T_x \quad T_y \quad T_z)^T$ gibt die Geschwindigkeit der Kamera bzw. die Geschwindigkeit von den Punkten im dreidimensionalen Raum wieder, die zu den betrachteten Pixeln korrespondierenden. T_x entspricht dabei der Kamera-Geschwindigkeit in X-, T_y in Y- und T_z in Z-Richtung zum Zeitpunkt der Aufnahme des zweiten Bildes. Die Einheit von T_x, T_y und T_z ist m/s. Gleichermaßen gibt $\boldsymbol{\omega} = (\boldsymbol{\omega}_x \quad \boldsymbol{\omega}_y \quad \boldsymbol{\omega}_z)^T$ die Rotation der Kamera wieder. $\boldsymbol{\omega}_x$ entspricht der Kamera-Rotation um die X-, $\boldsymbol{\omega}_y$ um die Y- und $\boldsymbol{\omega}_z$ um die Z-Achse zum Zeitpunkt der Aufnahme des zweiten Bildes. Die Einheit von $\boldsymbol{\omega}_x, \boldsymbol{\omega}_y$ und $\boldsymbol{\omega}_z$ ist 1/s.

Bei der Berechnung der Tiefe wird davon ausgegangen, dass die aufgenommenen Szenen statisch sind und sich nur die Kamera bewegt. Wenn sich die Kamera in positive Z-Richtung auf ein Objekt zu bewegt, bewegt sich aus der Sicht der Kamera das Objekt in negative Z-Richtung auf die Kamera zu. Gleichermaßen ist es bei Bewegungen der Kamera in X- und Y-Richtung und der Rotation. Wenn die Geschwindigkeit der Kamera aufgenommen wird, muss das Vorzeichen für die Berechnung umgekehrt werden.

Die im Folgenden dargestellten Matrizen $\boldsymbol{A}(x, y)$ und $\boldsymbol{B}(x, y)$ besitzen die Parameter $f, x,$ und y. f entspricht der Brennweite der verwendeten Kamera. Die Brennweite wird in Pixeln angegeben. x und y sind die Koordinaten des betrachteten Punktes auf der Bildebene des zweiten Bildes.

$$\boldsymbol{A}(x, y) = \begin{pmatrix} -f & 0 & x \\ 0 & -f & y \end{pmatrix}$$

$$\boldsymbol{B}(x, y) = \begin{pmatrix} \frac{x \cdot y}{f} & -\left(f + \frac{x^2}{f}\right) & y \\ f + \frac{y^2}{f} & -\frac{x \cdot y}{f} & -x \end{pmatrix}$$

Im Folgenden wird die Formel (1) nach $Z(x, y)$ umgestellt. Dazu wird zunächst der Term $\boldsymbol{B}(x, y) \cdot \boldsymbol{\omega}$ subtrahiert:

$$\begin{pmatrix} u_x \\ u_y \end{pmatrix} - \boldsymbol{B}(x, y) \cdot \boldsymbol{\omega} = \frac{1}{Z(x, y)} \cdot \boldsymbol{A}(x, y) \cdot \boldsymbol{t}$$

Aufgrund besserer Lesbarkeit wird der Term links vom Gleichheitszeichen als p geschrieben:

$$p := \begin{pmatrix} u_x \\ u_y \end{pmatrix} - \boldsymbol{B}(x,y) \cdot \boldsymbol{\omega}$$

$$p = \frac{1}{Z(x,y)} \cdot \boldsymbol{A}(x,y) \cdot \boldsymbol{t}$$

Als Nächstes wird die Gleichung mit der Transponierten des Vektors p multipliziert.

$$p^T p = p^T \left(\frac{1}{Z(x,y)} \cdot \boldsymbol{A}(x,y) \cdot \boldsymbol{t} \right)$$

$$p^T p = \frac{1}{Z(x,y)} \cdot p^T \cdot \boldsymbol{A}(x,y) \cdot \boldsymbol{t}$$

Da es sich bei $p^T p$ und $p^T \cdot \boldsymbol{A}(x,y) \cdot \boldsymbol{t}$ um Skalare handelt, kann die Gleichung durch Division nach $Z(x,y)$ umgestellt werden. Die umgestellte Gleichung lautet:

$$Z(x,y) = \frac{p^T \cdot \boldsymbol{A}(x,y) \cdot \boldsymbol{t}}{p^T p} \qquad (2)$$

Durch die Formel (2) kann die Tiefe eines Pixels bei bekannter Pixelgeschwindigkeit und bekannten Kameraparametern berechnet werden.

4.2 Template Matching

Template Matching ist eine gängige Methode in der Bildverarbeitung zum Wiederfinden von bekannten Objekten in einem Bild. Mit dieser Methode lassen sich Objekte in einem Bild lokalisieren oder in einer Bildersequenz verfolgen.

Bei dieser Methode wird ein Bildausschnitt, nachstehend Template genannt, in einem größeren Bild, nachstehend Zielbild genannt, gesucht. Hierbei wird das gesuchte Template durch das Zielbild verschoben. Beim Verschieben wird das Template mit dem Bildausschnitt an der korrespondierenden Stelle im Zielbild verglichen. Dabei werden Werte für die Ähnlichkeit dieser Ausschnitte mittels einer Funktion berechnet. Bei einer Übereinstimmung werden die Position sowie der Ähnlichkeitswert aufgenommen [28]. Das Prinzip wird in Abbildung 23 visualisiert.

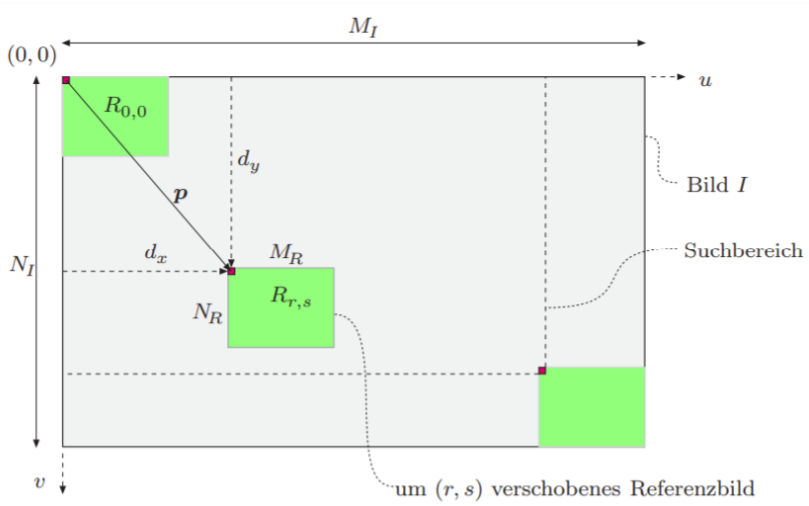


Abbildung 23: Suchprinzip des Template Matchings [29]

Die Position wird bestimmt, indem beim Verschieben des Templates der Abstand zum Koordinatenursprung des Zielbildes gemessen wird. Die Größe des Suchbereichs ist dabei von der Größe des Templates sowie von der Größe des Zielbilds abhängig [29]. Der Ähnlichkeitswert beschreibt das Maß für die Übereinstimmung zwischen den zu vergleichenden Bildausschnitten bei einer bestimmen Position im Zielbild. Um den Ähnlichkeitswert zu berechnen, stehen

verschiedene Möglichkeiten zur Verfügung. Zum Beispiel kann die Differenz der Grauwerte zwischen dem Template und dem Ausschnitt des Zielbilds berechnet werden. Je ähnlicher die beiden Bildausschnitte (Template und die zu untersuchende Stelle im Zielbild) sind, umso kleiner ist die Differenz der Grauwerte. Bei dieser Berechnungsmethode ist die höchste Übereinstimmung bei der Position mit der geringsten Differenz der Grauwerte [30].

In der OpenCV-Bibliothek ist die Methode des Template Matchings unter dem Namen „*matchTemplate“* definiert. Diese Methode benötigt das Zielbild, das Template und die Vergleichsmethode als Eingang und liefert als Ausgang die Ähnlichkeitswerte mit der dazugehörigen Position. Optional kann eine zusätzliche Bildmaske mit den Eingangsdaten angegeben werden. Es sind sechs Vergleichsmethoden (Stand 24.07.2021) zur Berechnung der Ähnlichkeitswerte vorhanden. Je nach Vergleichsmethode liefern entweder die größten oder die kleinsten Ähnlichkeitswerte die höchste Übereinstimmung der zu vergleichenden Bildausschnitte. Mithilfe der Methode „*minMaxLoc*“ aus der OpenCV-Bibliothek können die maximalen und die minimalen Ähnlichkeitswerte mit der jeweiligen Position ermittelt werden. Die Methode ermittelt aus einem gegebenen Datensatz das globale Minimum mit dazugehöriger Position sowie das globale Maximum mit der dazugehörigen Position der Werte.

4.3 Merkmalsvergleich

Im Folgenden wird auf die Methode „*Feature Matching*“, auch Merkmalsvergleich genannt, eingegangen. Bei dieser Methode werden Bildmerkmale verglichen, um Objekte in verschiedenen Bildern wiederzufinden. Die Methode besteht aus den folgenden drei Schritten [31]:

- Merkmalsextraktion: Finden der Merkmale
- Merkmalsdeskription: Berechnung der Merkmalsvektoren
- Merkmalsvergleich: Vergleichen der Merkmalsvektoren

Das Ziel der Merkmalsextraktion ist die Detektion von Bildmerkmalen und das Ziel der Merkmalsdeskription ist die eindeutige Beschreibung dieser Merkmale mit Werten oder Attributen [32]. Bildmerkmale sind markante Orte im Bild, welche sich durch ihre Form, Helligkeit, Auflösung oder anderen Faktoren herausheben [33].
Ein Vorteil der Charakterisierung durch Werte oder Attribute ist, dass die Berechnungsdauer des Vergleichens von Objekten deutlich geringer ist gegenüber anderen Vergleichsmethoden wie dem Template Matching [32].
Bei dem letzten Schritt der Bildmerkmalssuche werden Merkmalsvektoren verglichen, um dasselbe Merkmal in verschiedenen Bildern wiederzufinden.
Für die Merkmalsdetektion und Deskription stehen in der OpenCV-Bibliothek mehrere Algorithmen zur Verfügung, einige dieser Algorithmen werden im Folgenden aufgelistet:

Detektionsalgorithmen:

- **SIFT:** scale-invariant feature transform
- **FAST:** features from accelerated segment test
- **GFTT:** good featurest to track
- **MSER:** maximal stable extermal regions
- **ORB:** oriented FAST and rotated BRIEF
- **STAR:** basiert auf CenSurE – center surrounded extremae
- **SURF:** speeded up robust features

Deskriptoralgorithmen:

- **SIFT:** scale-invariant feature transform
- **SURF:** speeded up robust features
- **BRIEF:** binary robust independant elemantary features
- **BRISK:** binary robust invariant scalable keypoints
- **FREAK:** fast retina keypoints
- **ORB:** oriented FAST and rotated BRIEF

Im Folgenden werden Merkmalsvergleichsalgorithmen aufgelistet, welche in der OpenCV-Bibliothek vorhanden sind.

- **Brute-Force Matcher**
- **FLANN based Matcher:** Fast Approximate Nearest Neighbor Search Library

In der Masterarbeit [31] werden die oben aufgelisteten Merkmalsdetektoren untereinander verglichen. Hierbei wird für jeden Merkmalsdetektor der SURF-Algorithmus als Merkmalsdeskriptor und die „*Brute-Force*“-Methode für den Merkmalsvergleich verwendet. Abhängig der durchgeführten Experimente ist festzustellen, dass der SURF-Merkmalsdetektor in nahezu allen Kriterien die besten Ergebnisse erzielt. In der Arbeit werden ebenfalls die Merkmalsdeskriptoren untereinander verglichen. Hierzu wird der SURF-Algorithmus als Merkmalsdetektor und die „*Brute-Force*“-Methode für den Merkmalsvergleich verwendet. Die Merkmalsdeskriptoren FREAK, ORB und SURF haben bessere Ergebnisse als die restlichen Merkmalsdeskriptoren erzielt.

In der OpenCV-Bibliothek wird zwischen zwei Arten von Algorithmen für den Merkmalsvergleich unterschieden. Der „*Brute-Force*“ basierte Ansatz vergleicht den gesuchten Merkmalsdeskriptor mit allen vorhandenen Deskriptoren. Der Vergleich wird über verschiedene Methoden realisiert wie z. B. der Berechnung der euklidischen Distanz.

Die „*FLANN*“ basierten Algorithmen vergleichen nicht alle Merkmalsvektoren miteinander und haben deshalb bei großen Datensätzen eine deutlich niedrigere Berechnungsdauer als der „*Brute-Force*“ basierte Algorithmus. Zu einem gesuchten Merkmalsdeskriptor wird ein möglichst ähnlicher Deskriptor gesucht, jedoch kann nicht garantiert werden, dass dieser der Deskriptor mit der höchsten Übereinstimmung ist.

4.4 Clustering

Im Folgenden wird die Methode des Clusterings beschrieben. Das Clustering ist ein Verfahren zum Gruppieren von Objekten oder Merkmalen abhängig von ihren Eigenschaften und Beziehungen. Eine zusammengesetzte Gruppe, welche ähnliche Eigenschaften aufweist, wird als Cluster bezeichnet. Das Verfahren oder die Methode zur Gruppierung dieser Cluster wird Clustering genannt [34].

Bei der Methode der Klassifizierung werden ebenfalls ähnliche Objekte in Gruppen oder Klassen eingeordnet, jedoch sind diese Klassen vordefiniert. Bei dem Clustering sind keine Gruppierungen vordefiniert. Die Gruppen werden eigenständig von den Clusteralgorithmen abhängig von den zu untersuchenden Daten generiert [34].

In der OpenCV-Bibliothek werden die zwei Methoden *„kmeans“* und *„partition“* für das Clustering beschrieben. Die Methode *„kmeans“* ist für Anwendungen mit kleiner Anzahl an Clustern geeignet. Die Anzahl der Cluster muss bei dieser Methode vorgegeben werden. Bei der Methode *„partition“* muss die Anzahl der zu berechnenden Cluster nicht vorgegeben werden. Da in dieser Arbeit die Methode „*partition*“ zum Gruppieren von Bildmerkmalen verwendet wird, wird auf diese im Folgenden näher eingegangen.

Die Methode „*partition*“ besitzt die nachstehend genannten Parameter: „*_vec*“ „*labels*“ und „*predicte*“. Mit dem Parameter „*_vec*“ wird der Methode ein Vektor übergeben, welcher die Daten für die Gruppierung beinhaltet. Mit dem Parameter „*predicte*“ wird das Ergebnis einer booleschen Funktion übergeben. Die Funktion beinhaltet die Bedingung für die Zuordnung einzelner Elemente in die Gruppierungen. Der Parameter „*labels*“ beinhaltet die berechneten Cluster.

4.5 Kamerakalibrierung

Für die Implementierung des dritten Ansatzes wird die Brennweite der Kamera benötigt. Um die Brennweite zu ermitteln, wird eine Kamerakalibrierung durchgeführt. Im Folgenden wird auf den Ansatz zur Kamerakalibrierung eingegangen. Mit einer Kamerakalibrierung werden die Abbildungsparameter einer Kamera bestimmt. Für die Kalibrierung werden Referenzpunkte eines Kalibrierungsmusters verwendet. Hierfür werden von dem Kalibrierungsmuster mehrere Aufnahmen in unterschiedlichen Entfernungen und Drehungen zu der Kamera aufgenommen.

In dieser Arbeit wird das Programm „*calibrate_camera*“ aus dem ArUco-Modul der OpenCV-Bibliothek für die Kalibrierung der Mobiltelefonkamera und der Webcam verwendet. Das verwendete Kalibrierungsmuster wird in Abbildung 24 dargestellt. Dieses Muster wird mit dem Programm „*create_board*“ aus dem ArUco-Modul der OpenCV-Bibliothek erstellt. Dafür werden folgende Parameter verwendet:

- Anzahl Marker in X-Richtung: w = 5
- Anzahl Marker in Y-Richtung: h = 7
- Kantengröße der Marker in Pixeln: l = 100
- Abstand zwischen zwei Markern in Pixeln: s = 10
- Verwendete Markerbibliothek: d = 10

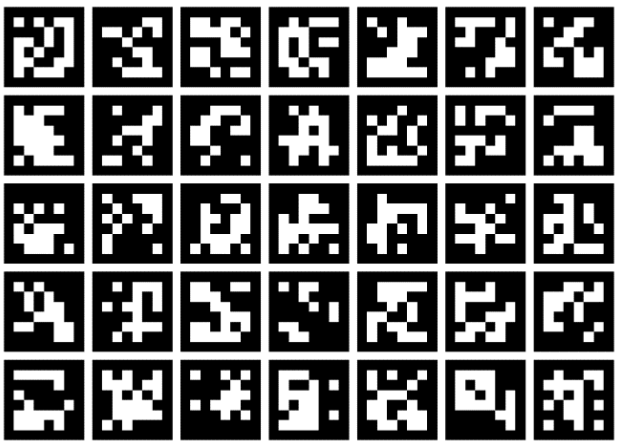


Abbildung 24: Verwendetes Kalibrierungsmuster für die Kamerakalibrierung

5 Implementierung

Nachdem die wesentlichen Methoden in Kapitel 4 vorgestellt sind, wird in den folgenden Unterkapiteln auf die Implementierung der einzelnen Ansätze eingegangen. Es werden der Aufbau der zu implementierenden Programme und Anpassungen an diesen erläutert. Zudem wird die verwendete Software sowie die OpenCV-Bibliothek beschrieben. Die aus der Implementierung resultierenden Testergebnisse werden im Anschluss vorgestellt. Die Testergebnisse werden in Kapitel 6 ausgewertet. Der Code der einzelnen implementierten Ansätze befindet sich im Anhang.

Die Aufnahmen der Webcam ohne optische Bildstabilisierung sind aufgrund der Vibrationen durch die Unebenheiten im Boden nicht verwendbar. Ein Beispiel für solch eine Aufnahme ist in Abbildung 25 zu sehen. Aus diesem Grund werden im weiteren Verlauf nur die Aufnahmen verwendet, die mit dem „*Apple iPhone 12*“ gemacht wurden. Die Aufnahmen werden für die Berechnung auf eine Größe von 320 x 240 Pixeln herunterskaliert.

Abbildung 25: Beispielaufnahme der Webcam

5.1 Software

Wie in dem Kapitel 1.2 erwähnt, erfolgt die Implementierung der Ansätze mithilfe der Methoden aus der OpenCV-Bibliothek. Die Programmiersprache wurde auf C++ festgelegt. In dieser Arbeit wird die Version 4.5.2 der OpenCV-Bibliothek verwendet. Die Installation erfolgt mithilfe der Softwareprogramme „*Visual Studio Community 2019*“ der Version 16.9.31205.134 und „*CMAKE“* der Version 3.20.3. Die Implementierung der Ansätze erfolgt ebenfalls mit „*Visual Studio Community 2019*“. Um die Live-Aufnahmen zu simulieren, wird die Software „*OBS Studio*“ der Version 26.1.1 verwendet. Alle verwendeten Softwareprogramme und die Implementierung der Ansätze werden auf dem Betriebssystem Windows 10 Pro der Version 20H2 realisiert.

5.2 OpenCV-Bibliothek

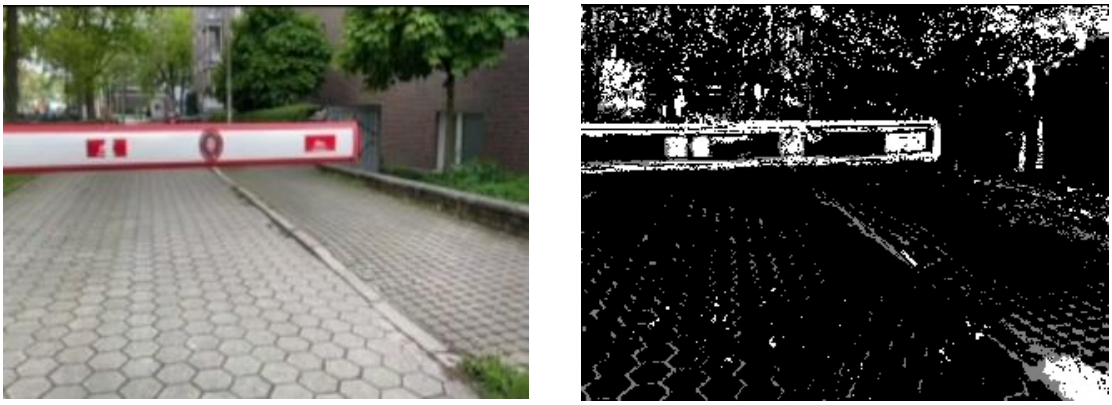
Die OpenCV-Bibliothek (Open Source Computer Vision Library) ist eine frei verfügbare Programmbibliothek für Computer Vision Wissenschaft sowie für das maschinelle Lernen. Die Bibliothek verfügt über mehr als 2500 optimierte Algorithmen und hat über 18 Mio. Downloads weltweit. Die meisten Algorithmen sind in der Programmiersprache C++ geschrieben, jedoch werden Schnittstellen für die Programmiersprachen Python, Java und MATLAB angeboten. Unterstützte Betriebssysteme sind Windows, Linux, Android und macOS.

5.3 Ansatz 1: Optischer Fluss und Template Matching

An dieser Stelle wird ein angepasster Ansatz nach der Masterarbeit [14] näher erläutert. Mit diesem Ansatz wird eine Hinderniserkennung mittels optischen Flusses und Template Matching realisiert. Der Ansatz besteht aus drei Schritten, welche im Folgenden einzeln beschrieben werden.

5.3.1 Hintergrundsubtraktion und Filterung

In dem ersten Schritt werden eine Hintergrundsubtraktion und eine Filterung der gefundenen Konturen durchgeführt. Hierfür werden zunächst Bilder über eine monokulare Kamera aufgenommen. Auf die aufgenommenen Bilder wird mithilfe der OpenCV-Methode „*BackgroundSubtractorMOG2*“ eine Hintergrundsubtraktion angewandt, um Objekte, welche sich innerhalb der aufgenommenen Szenen nicht bewegt haben, herauszufiltern. Die verwendete Lern-Rate dieser Methode wird nach [14] auf $\alpha = 0{,}01$ festgelegt. In der folgenden Abbildung 26 wird das aufgenommene Bild sowie das Bild nach Anwendung der Methode dargestellt.



(a) (b)

Abbildung 26: Hintergrundsubtraktion (a) Eingangsbild (b) Ausgangsbild nach Anwendung der Hintergrundsubtraktion

Bei diesem Schritt werden nicht alle irrelevanten Bereiche entfernt. Deshalb werden nach der Hintergrundsubtraktion zunächst ausgeprägte Konturen mit der Methode „*findContours*“ aus der OpenCV-Bibliothek detektiert. Dabei werden Konturen, welche eine Fläche kleiner als 0,5 % der Bildauflösung aufweisen, herausgefiltert, um ein rauschfreies Bild zu erzeugen. Als Konturen werden Linien oder Punkte bezeichnet, welche die Abgrenzung eines Objektes wiedergeben. Mit Hilfe der OpenCV-Methode „*drawContours*“ wird ein Bild erstellt, welches die gefilterten Konturen und deren Position im Bild beinhaltet. Diese Vorfilterung des Eingangsbildes soll die Detektion von Merkmalen im nächsten Schritt ausschließlich auf mögliche Hindernisse beschränken. Hierdurch kann z. B. der Boden, welcher kein Hindernis darstellt, herausgefiltert werden. Ein Beispiel für das Ergebnis der Vorfilterung des Eingangsbildes ist in der folgenden Abbildung 27 dargestellt.

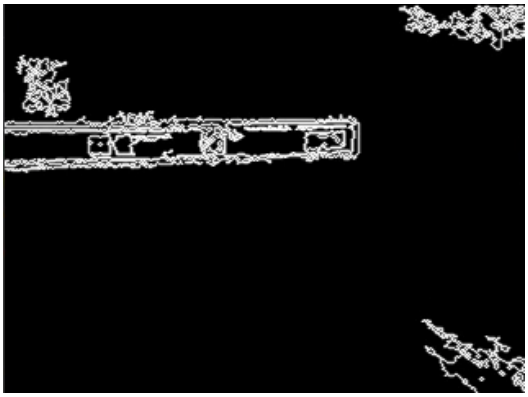
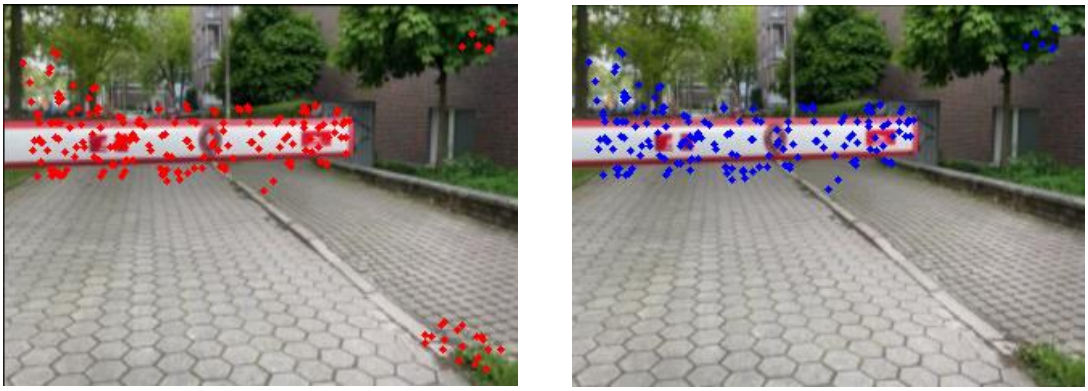


Abbildung 27: Ausgangsbild nach der Filterung

Zwar werden nicht alle Bereiche des Bodens oder des Hintergrunds entfernt, jedoch wird das Bild insgesamt rauschärmer und der relevante Bereich der Schranke bleibt erhalten.

5.3.2 Merkmalsdetektion und Clustering

Im nächsten Schritt werden Bildmerkmale in dem vorgefilterten Bild detektiert. Zudem werden die detektierten Bildmerkmale mit dem aus [14] beschriebenen Algorithmus gefiltert. Die Bildmerkmale in [14] werden mit dem Merkmalsdetektor „goodFeaturesToTrack“ generiert. Aufgrund der Ausführungen in der Masterarbeit [31] wird jedoch in der folgenden Implementierung der Merkmalsdetektor „SURF“ verwendet, da in den durchgeführten Experimenten der SURF-Merkmalsdetektor bessere Ergebnisse erzielen konnte. Beide Merkmalsdetektoren sind in der OpenCV-Bibliothek vorhanden. Bei dem verwendeten SURF-Merkmalsdetektor wird der Schwellwert für die Berechnung der Determinante der Hesse-Matrix („*hessianThreshold*“) auf 800 festgelegt. Anschließend werden die detektierten Merkmale mithilfe des Algorithmus aus [14] („*Algorithm 2*“) gefiltert und gruppiert, um isolierte Bildmerkmale, welche nicht zu einem Objekt gehören und überlappende Bildmerkmale zu verwerfen. Bei dieser Methode werden Objekte anhand zusammenhängender Bildpunkte und deren Distanz zueinander verwertet. Die Methoden der Merkmalsdetektion und Merkmalsdeskription werden in dem Kapitel 4.3 näher erläutert. In der folgenden Abbildung 28 werden die detektierten sowie die gefilterten Merkmale nach dem ersten Schritt der Vorfilterung dargestellt.



(a) (b)

Abbildung 28: Merkmalsdetektion und Gruppierung a) detektierte Merkmale b) gruppierte Merkmale

5.3.3 Template Matching und Optischer Fluss

Nachdem die Merkmale detektiert und gruppiert sind, werden im letzten Schritt parallel die Methoden für die Berechnung des optischen Flusses und die des Template Matchings angewandt. Für beide Methoden werden zwei aufeinanderfolgende Bilder betrachtet.

Mit der Methode Template Matching wird das Ausmaß der Expansion von Objekten, die durch die Ansammlung von zuvor gefilterten Merkmalen beschrieben werden, berechnet, um herannahende Objekte zu detektieren. Diese Methode wird in Kapitel 4.2 genauer erläutert.
Bei der Methode wird um jede Ansammlung von Bildmerkmalen in dem vorherigen Bild ein Bildausschnitt generiert. In dem aktuellen Bild werden Bildausschnitte an den gleichen Positionen mit gleicher Größe erstellt. Die Bildausschnitte aus dem vorherigen Bild werden zusätzlich zu der originalen Größe in acht verschiedenen Größen skaliert. Dabei wird die Kantengröße des Bildausschnitts betrachtet. Die kleinere Kantengröße wird um jeweils einen Pixel erhöht und die größere Kantengröße um einen Wert, bei dem das originale Seitenverhältnis beibehalten wird. Die skalierten Bildausschnitte des vorherigen Bildes werden mit dem jeweiligen Bildausschnitt aus dem aktuellen Bild verglichen. Hierfür werden die OpenCV-Methoden „*matchTemplat*e“ und „*minMaxLoc*“ verwendet. Bei den detektierten Bildausschnitten gibt die Skalierung mit der höchsten Übereinstimmung das Ausmaß der Expansion der Objekte wieder. In der Abbildung 29 ist eine Beispielaufnahme für das Ergebnis des Template Matchings dargestellt.

Abbildung 29: Ergebnisse nach Anwendung des Template Matchings

Die abgebildeten Punkte entsprechen dabei den Mittelpunkten der in Schritt 2 berechneten Ansammlungen von Merkmalen. Bildausschnitte der Ansammlungen, bei denen sich die kleinere Kantengröße um min. 5 Pixel erhöht hat, werden rot dargestellt. Diese Punkte entsprechen Hindernissen. Die restlichen Ansammlungen werden mit blauen Punkten dargestellt.
Um zusätzlich nahe von fernen Objekten zu trennen, wird der optische Fluss durch die Bewegung der ungefilterten Bildmerkmale berechnet. Für die Berechnung des optischen Flusses wird die Methode „calcOpticalFlowPyrLK“ aus der OpenCV-Bibliothek verwendet. Diese Methode berechnet die Positionen der Merkmale, die im vorherigen Bild detektiert wurden, im aktuellen Bild. Die Methode wird in Kapitel 4.1.1 näher erläutert. Nach der Berechnung des optischen Flusses wird der Mittelwert der zurückgelegten Distanzen von Merkmalen vom vorherigen zum aktuellen Bild berechnet. Durch die Trennung von Punkten mithilfe des Mittelwerts werden nahe von fernen Objekten unterschieden. Als Schwellwert für die Klassifizierung von nahen und fernen Objekten wird das 1,2-Fache des Mittelwerts festgelegt. In der Abbildung 30 ist eine Beispielaufnahme für den optischen Fluss dargestellt. Die grünen Punkte entsprechen den Merkmalen, deren Distanz unter dem Schwellwert liegt. Diese Merkmale werden als fern eingestuft. Die roten Punkte entsprechen Merkmalen, deren Distanz über dem Schwellwert liegt. Diese Punkte werden als nah eingestuft.

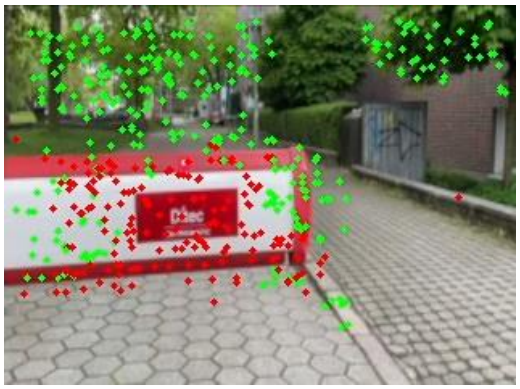


Abbildung 30: Ergebnisse nach Anwendung der Methode des optischen Flusses

Die Hinderniserkennung erfolgt anschließend durch die gewonnenen Informationen aus den angewandten Methoden des Template Matchings und des optischen Flusses.

Eine Übersicht des Aufbaus ist in Abbildung 31 dargestellt.

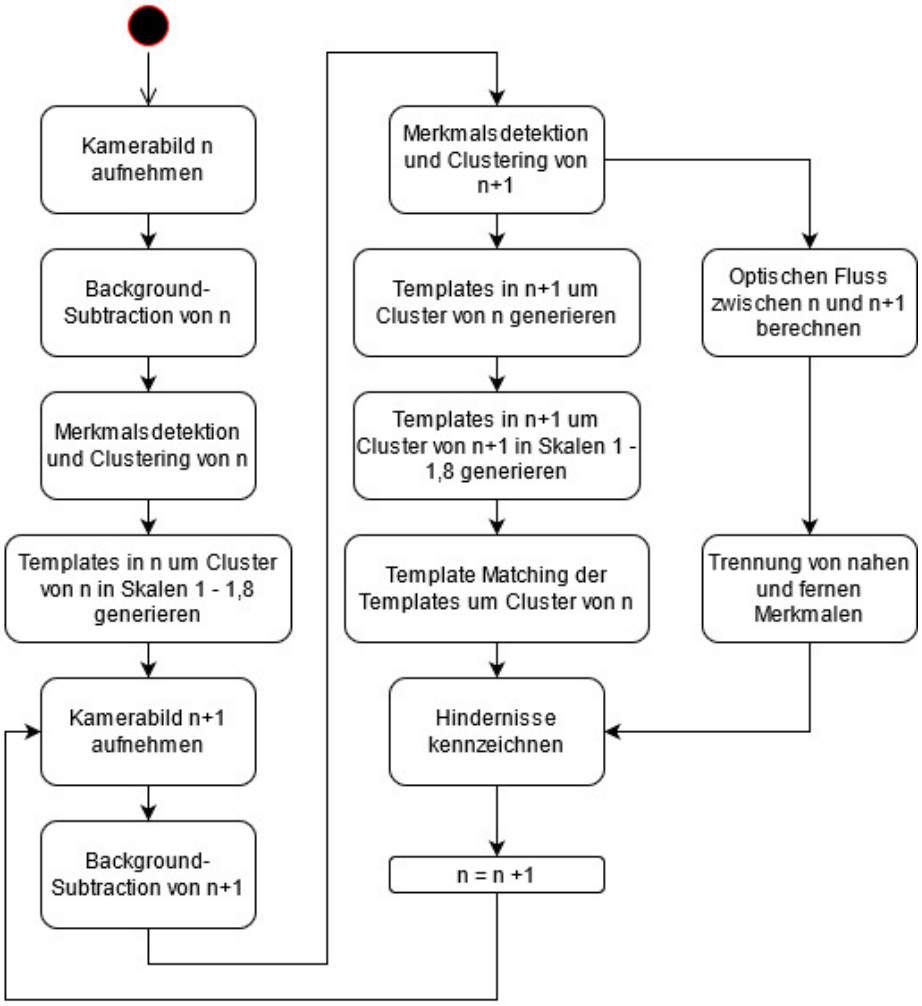


Abbildung 31: Programmaufbau Ansatz 1

5.3.4 Testergebnisse

Im Folgenden wird auf die Testergebnisse des zuvor beschriebenen Ansatzes eingegangen.

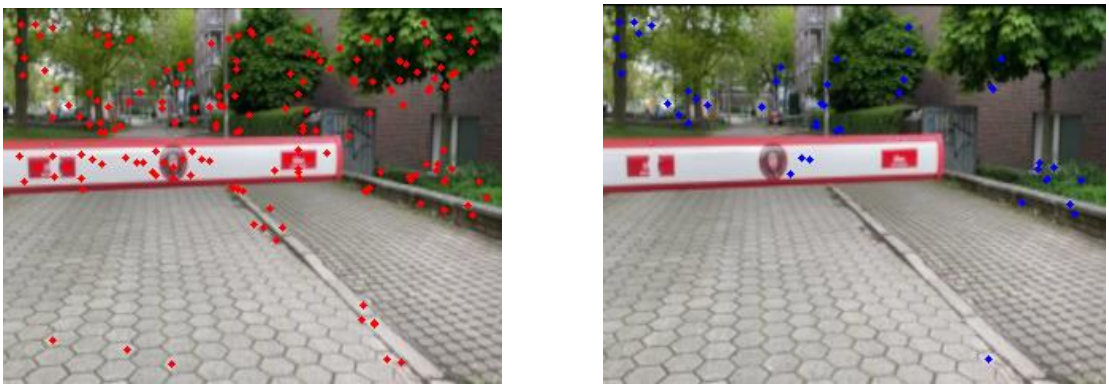
Bei dem Testen des Ansatzes mit den aufgenommenen Bildern ist auffällig, dass Objekte mit einer einfarbigen oder spiegelnden Oberfläche nicht detektiert werden. Dies liegt an der zuvor beschriebenen Filterung durch die Hintergrundsubtraktion. Ein Beispiel dafür ist die in Abbildung 32 dargestellte Schranke, welche zum Großteil aus dem Bild entfernt wird.

(a) (b)

Abbildung 32: Fehlerhafte Hintergrundsubtraktion a) Eingangsbild b) Beispielaufnahme für eine fehlerhafte Filterung

Durch die Entfernung des Schrittes der Hintergrundsubtraktion bleiben solche Objekte im Bild erhalten, jedoch werden auch mehr Merkmale auf dem Boden und im Hintergrund detektiert. Dies hat zur Folge, dass insgesamt weniger Merkmale auf zuvor detektierten Objekten erkannt werden. Der zusätzliche Schritt der Filterung von Merkmalen führt zu einer weiteren Reduzierung von Merkmalen auf relevanten Objekten.

Ein Beispiel für eine Aufnahme ohne die Filterung mittels Hintergrundsubtraktion ist in Abbildung 33 dargestellt.



(a) (b)

Abbildung 33: Merkmalsdetektion und Gruppierung ohne Hintergrundsubtraktion a) alle detektierten Merkmale b) gruppierte Merkmale

Die Trennung von nahen und fernen Objekten mittels optischen Flusses durch die Berechnung des Mittelwerts von zurückgelegten Distanzen ist, wie in Abbildung 30 dargestellt, nicht eindeutig. Da die Trennung bei Objekten, auf denen sich annähernd die Hälfte aller detektierten Merkmale des Bildes befinden, dazu führt, dass Objekte sowohl als nah und als fern gekennzeichnet werden. Die Festlegung eines festen Schwellwertes für die Unterteilung bei bekannter Geschwindigkeit führt zu einer eindeutigen Trennung. Jedoch ist die Trennung mit einem festen Schwellwert bei Drehungen der Kamera oder Vibrationen im Bild fehlerhaft.

Als Nächstes wird auf die Ergebnisse des Schrittes des Template Matchings eingegangen. Die Vorgehensweise zur Detektion von herannahenden Objekten liefert zum Großteil falsche Ergebnisse. Dies liegt daran, dass die Positionen der Bildausschnitte aus dem aktuellen Bild von den Positionen der Ansammlungen von Merkmalen im vorherigen Bild abhängig sind. Nur bei Bewegungen der Kamera, bei der diese sich mittig auf ein Objekt zu bewegt, werden herannahende Objekte korrekt detektiert.

Ein Beispiel für eine fehlerhafte Detektion ist in Abbildung 34 dargestellt.

Abbildung 34: Beispielaufnahme einer fehlerhaften Hindernisdetektion

Die Berechnungsdauer eines Bildes der Größe 320 x 240 Pixeln beträgt ca. 72 Millisekunden. Somit ergibt sich eine Bildrate von 14 Bildern pro Sekunde.

5.4 Ansatz 2: Template Matching

Im Folgenden wird ein Ansatz nach [16] beschrieben. Bei diesem Ansatz werden Hindernisse auf Bildern einer monokularen Kamera durch detektierte Merkmale erkannt. Jede der verwendeten Methoden ist in der OpenCV-Bibliothek vorhanden.

5.4.1 Merkmalsdetektion und -vergleich

Bei dieser Vorgehensweise werden identische Merkmale in aufeinanderfolgenden Bildern benötigt. Deshalb werden bei jedem aufgenommenen Bild Merkmale detektiert und zu jedem der detektierten Merkmale ein Deskriptor erstellt. In diesem Vektor werden die Eigenschaften eines Merkmals beschrieben. Durch Vergleichen dieser Vektoren werden gleiche Merkmale in aufeinanderfolgenden Bildern ermittelt. Für die Merkmalsdetektion und Deskription wird aufgrund der Ergebnisse aus [31] anders als in [16] der SURF-Algorithmus verwendet. Bei dem verwendeten SURF-Merkmalsdetektor wird der Schwellwert für die Berechnung der Determinante der Hesse-Matrix („*hessianThreshold*“) wie bei der Implementierung des Ansatzes 1 auf 800 gesetzt. In der Abbildung 35 werden die detektierten Bildmerkmale einer Beispielaufnahme durch rote Kreise dargestellt.

Abbildung 35: Detektierte Merkmale im Eingangsbild

Nach der Detektion und Deskription von Merkmalen in zwei aufeinanderfolgenden Bildern werden die Deskriptoren der Merkmale durch die „*FLANN*“-basierte Methode „*knnMatch*“ miteinander verglichen. Dabei werden für jedes Merkmal aus dem aktuellen Bild die zwei Merkmale mit der höchsten Übereinstimmung aus dem vorherigen Bild berechnet. Diese zwei Merkmalspaare werden anschließend nach der Methode aus [10], bei der das Verhältnis der euklidischen Distanzen betrachtet wird, gefiltert, um schlechte Paare auszusortieren. Zusätzlich werden Merkmalspaare aussortiert, bei denen die Merkmale des aktuellen Bildes nicht größer sind als die Merkmale des vorherigen Bildes. Dazu wird bei jedem Merkmalspaar die Fläche der einzelnen Merkmale berechnet. Sollte die Fläche des Merkmals in dem nachfolgenden Bild nicht größer sein als die Fläche des Merkmals im vorherigen Bild, wird dieses Merkmalspaar aussortiert und nicht weiter betrachtet.

In der folgenden Abbildung 36 werden die gefilterten Merkmale aus zwei aufeinanderfolgenden Bildern abgebildet. Hierbei werden Merkmale als Kreise dargestellt. Die gepaarten Merkmale sind durch eine Linie miteinander verbunden.

Abbildung 36: Gepaarte Merkmale in aufeinanderfolgenden Bildern

5.4.2 Template Matching

Bei den verbleibenden Merkmalspaaren handelt es sich um potenzielle Hindernisse. Um herannahende Objekte zu detektieren, wird die Methode Template Matching verwendet. Hierzu werden die gefilterten Merkmale aus dem vorherigen Schritt betrachtet. Es werden quadratische Bildausschnitte sowohl im vorherigen als auch im nachfolgenden Bild um die verbleibenden Merkmalspaare generiert. Der Bildausschnitt aus dem vorherigen Bild wird dabei zusätzlich zu der originalen Größe in fünf verschiedenen Skalierungen generiert. Die Kantengröße des Bildausschnitts wird jeweils um 10, 20, 30, 40 und 50 % erhöht. Nach der Skalierung liegt der Bildausschnitt in der originalen Größe und mit bis zu 50 % größerer Kantengröße vor. Für die Generierung der Skalen wird die Methode „*resize*“ verwendet.

Als Nächstes werden die sechs Skalierungen des Bildausschnitts des vorherigen Bildes mit dem korrespondierenden Bildausschnitt aus dem nachfolgenden Bild mithilfe der Methode „*matchTemplate*“ verglichen. Dabei wird die Skalierung des Bildausschnitts mit der höchsten Übereinstimmung zu dem Bildausschnitt aus dem nachfolgenden Bild mit der Methode „*minMaxLoc*“ ermittelt. Sollte die Kantengröße des Bildausschnitts mit der höchsten Übereinstimmung über 120 % der originalen Kantengröße betragen, wird das jeweilige Merkmal als Hindernis eingestuft. Durch Anpassen der Skalierungsgrößen und durch Anpassen des Schwellwerts kann die Entfernung eingestellt werden, in der Objekte als Hindernisse eingestuft werden. In der folgenden Abbildung 37 werden Merkmale, welche als Hindernis eingestuft sind, als rote Punkte dargestellt.

Abbildung 37: Ergebnis der Hindernisdetektion

Der Aufbau dieser Implementierung ist in der folgenden Abbildung 38 dargestellt.

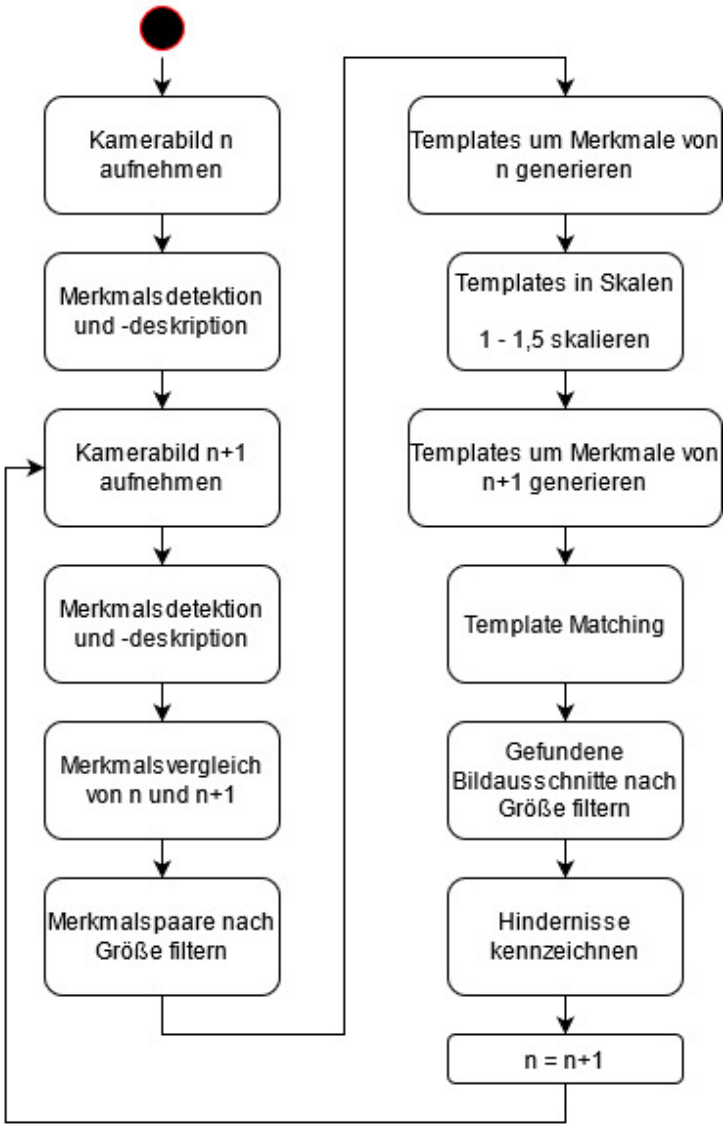
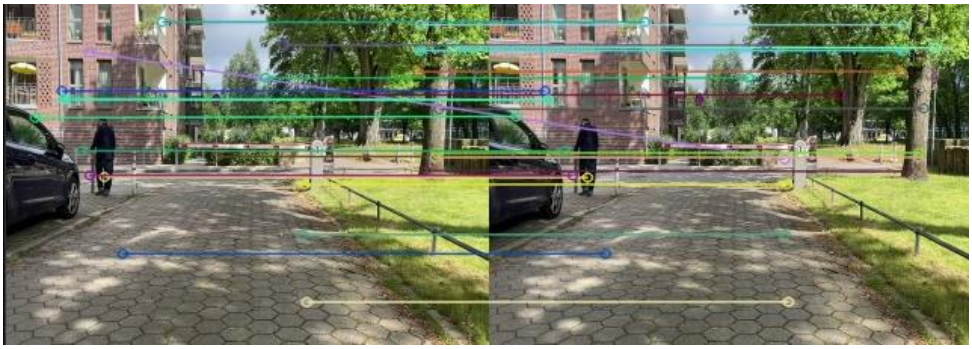


Abbildung 38: Programmaufbau Ansatz 2

5.4.3 Testergebnisse

Nach der Implementierung des Ansatzes 2 werden im Folgenden die Testergebnisse vorgestellt.

Es werden häufig fälschlicherweise weit entferne Objekte als Hindernis eingestuft. Dies liegt an Fehlern in der Paarung von Merkmalen. In der folgenden Abbildung 39 wird eine fehlerhafte Hindernisdetektion und die dazu korrespondierende Merkmalspaarung dargestellt. Da die zwei betrachteten Bilder zeitlich sehr nah aneinander liegen und die abgebildeten Objekte weit von der Kamera entfernt sind, sollten die Linien, die die Merkmalspaare verbinden, nahezu parallel zueinander sein. Jedoch ist auf dem Bild durch die nicht parallel laufende Linie zu erkennen, dass die Bildung eines Merkmalspaars fehlerhaft ist. Dies führt zu der fehlerhaften Detektion eines Hindernisses.



(a)

(b)

Abbildung 39: Fehlerhafte Merkmalspaarung und die daraus resultierende Hindernisdetektion a) Paarung von Merkmalen b) Hindernisdetektion

Nach der Bildung und Filterung von Merkmalspaaren auf Objekten ohne viel Textur stehen sehr wenige Merkmalspaare für das anschließende Template Matching zur Verfügung. Dies ist auf der Schranke in Abbildung 36 zu erkennen.

Abgesehen von der Detektion von falschen Hindernissen werden Hindernisse wie in Abbildung 37 zu sehen ist, erst bei einer geringen Distanz zwischen Hindernissen und Kamera erkannt. Eine Anpassung der Skalierung der Bildausschnitte des vorherigen Bildes und die Anpassung des Schwellwerts führen nur zu einer insignifikanten Vergrößerung des Abstands, ab dem Hindernisse als solche detektiert werden. Bei dieser Anpassung wurde die Größe des Bildausschnitts um jeweils einen Pixel pro Skalierungsebene erhöht. Eine weitere Vergrößerung dieses Abstands wird erreicht, indem Bilder mit einem größeren zeitlichen Abstand zueinander betrachtet werden. Hierdurch ist die Vergrößerung einzelner Merkmale deutlicher. Jedoch führt dies dazu, dass die Berechnung insgesamt langsamer wird.

Die Berechnungsdauer eines Bildes, bei der zwei aufeinanderfolgende Bilder der Größe 320 x 240 Pixeln betrachtet werden, beträgt ca. 39 Millisekunden. Somit ergibt sich eine Bildrate von 26 Bildern pro Sekunde.

5.5 Ansatz 3: Tiefe durch optischen Fluss

Der im Folgenden beschriebene Ansatz generiert durch Informationen des optischen Flusses Tiefendaten. Es werden Bilder verwendet, die durch eine monokulare Kamera aufgenommen werden. Als optischer Fluss wird hier die Bewegung eines Pixels in einem Bild zu einer anderen Position in einem darauffolgenden Bild bezeichnet. Bei dieser Vorgehensweise werden zwei verschiedene Algorithmen für die Berechnung des optischen Flusses verwendet. Zum einen wird der Lucas-Kanade- [25] und zum anderen der Farnebäck-Algorithmus [26] verwendet. Beide dieser Algorithmen sind in der OpenCV-Bibliothek vorhanden. Der Lucas-Kanade-Algorithmus wird über die OpenCV-Methode „*calcOpticalFlowPyrLK*“ und der Farnebäck-Algorithmus über die OpenCV-Methode „*calcOpticalFlowFarneback*“ aufgerufen. Beide Algorithmen werden in dem Kapitel 4.1 beschrieben.

Bei der ersten Variante wird der Farnebäck-Algorithmus verwendet. Bei dieser Methode werden zwei aufeinanderfolgende Bilder betrachtet. Nach dem Aufnehmen der Bilder werden diese durch die OpenCV-Methode „*cvtColor*“ in Graubilder konvertiert. Es wird die Bewegung jedes Pixels im vorherigen Bild zu der Position im nachfolgenden Bild berechnet. Jedoch wird bei der Berechnung der Tiefe nur jeder 5. Pixel in x- sowie in y-Richtung betrachtet, um den Rechenaufwand möglichst gering zu halten.

Der Lucas-Kanade-Algorithmus wird bei der zweiten Variante verwendet. Hierfür werden, wie bei der ersten Variante, zwei aufeinanderfolgende Bilder betrachtet. Im vorherigen Bild werden Merkmale mithilfe des SURF-Algorithmus detektiert. Bei dem verwendeten SURF-Merkmalsdetektor wird der Schwellwert für die Berechnung der Determinante der Hesse-Matrix („*hessianThreshold*“) wie bei den Ansätzen zuvor auf 800 festgelegt. Für die Merkmale werden, anders als beim Ansatz 2 keine Deskriptoren benötigt. Diese Merkmale werden anschließend für die Berechnung des optischen Flusses zum nachfolgenden Bild verwendet.

Zusätzlich wird eine dritte Variante implementiert, die auch auf dem Lucas-Kanade-Algorithmus basiert. Mit dieser Variante wird ermittelt, ob es zu einer Verbesserung führt, wenn der optische Fluss zwischen zwei zeitlich entfernteren Bildern berechnet wird. Dazu werden, wie bei der zweiten Variante, Merkmale in einem Bild detektiert und der optische Fluss zu einem nachfolgenden Bild berechnet. Die Merkmale des vorherigen Bildes werden

gespeichert und für die Berechnung zu weiteren nachfolgenden Bildern verwendet. Erst wenn die Anzahl der Pixel, bei denen der optische Fluss berechnet werden konnte, unter einem definierten Schwellwert liegt, werden neue Bildmerkmale aufgenommen. Der Schwellwert wird hier auf 90 % der Anzahl der zuvor aufgenommenen Merkmale festgelegt.

Nachdem der optische Fluss berechnet ist, werden bei der ersten Variante über die Formel (2) die Tiefenwerte von jedem 5. Pixel in x- sowie in y- Richtung berechnet. Gleichermaßen werden bei der zweiten und dritten Variante die Tiefenwerte jedes Pixels, für die der optische Fluss berechnet ist, im nachfolgenden Bild berechnet. Die Formel wird in dem Kapitel 4.1.3 näher erläutert. Um die Tiefe zu berechnen, wird die Brennweite der Kamera, die Translation und Rotation der Kamera zwischen den zwei betrachteten Bildern benötigt. Das Aufnehmen der Daten wird in Kapitel 3.3 näher beschrieben.

Nach der Berechnung der Tiefenwerte werden in allen drei Varianten Pixel mit Tiefenwerten unter einem definierten Schwellwert durch die Methode „*partition*“ gruppiert. Der Schwellwert wird hier auf 1 m festgelegt. Um die Gruppierungen, die mindestens 10 Pixel beinhalten, wird im Bild durch die Methoden „*convexHull*“ und „*drawContours*“ eine Linie dargestellt. Die Bereiche, die durch diese Linie markiert sind, kennzeichnen Hindernisse im Bild. Eine nähere Beschreibung zu der Zusammenfassung der Pixel ist in Kapitel 4.4 beschrieben.

Zusätzlich werden zur Veranschaulichung Richtungsanweisungen bei Hindernissen generiert. Dazu wird das Eingangsbild in x-Richtung in drei gleichgroße Bereiche, wie in der Abbildung 40 dargestellt, unterteilt.

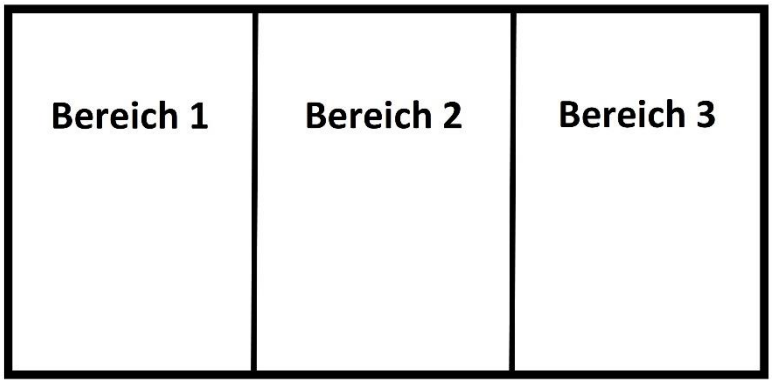


Abbildung 40: Aufteilung des Eingangsbildes in drei Bereiche

Es werden die x-Werte der zuvor gruppierten Pixel mit Tiefenwerten kleiner als 1 m betrachtet. Abhängig von diesen x-Werten werden Richtungsanweisungen erzeugt. Die verwendeten Richtungsanweisungen werden in der folgenden Tabelle 1 beschrieben.

Richtungsanweisung	**Beschreibung**
„Stopp“	Der „*Shared Guide Dog*“ muss anhalten.
„Links“	Der „*Shared Guide Dog*“ muss nach links ausweichen.
„Rechts“	Der „*Shared Guide Dog*“ muss nach rechts ausweichen.

Tabelle 1: Verwendete Richtungsanweisungen

Die Anweisungen „Rechts“ und „Links“ werden durch grüne Pfeile und die Anweisung „Stopp“ durch roten Text im Ergebnisbild dargestellt. Diese Anweisungen werden durch die in der folgenden Tabelle 2 dargestellten Bedingungen erzeugt. Für Fälle, die nicht in dieser Tabelle beschrieben sind, werden keine Richtungsanweisungen generiert.

Hindernis in Bereich 1	**Hindernis in Bereich 2**	**Hindernis in Bereich 3**	**Richtungsanweisung**
Ja	Ja	Ja	Stopp
Ja	Ja	Nein	Rechts
Nein	Ja	Ja	Links
Nein	Ja	Nein	Links und Rechts

Tabelle 2: Bedingungen für die Zuweisung der Richtungsanweisungen

Im Folgenden wird der Aufbau der drei Varianten der Implementierung des Ansatzes 3 in Abbildung 41, Abbildung 42 und Abbildung 43 dargestellt.

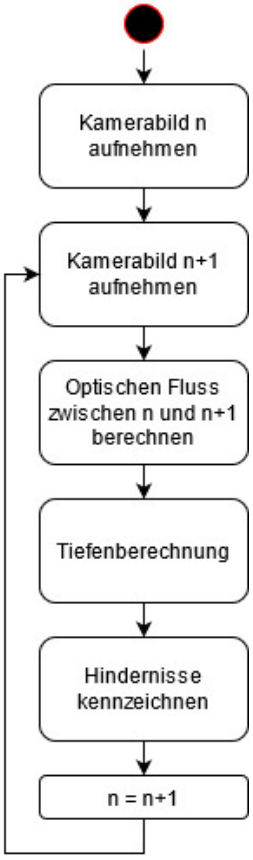


Abbildung 41: Programmaufbau Ansatz 3 Variante 1

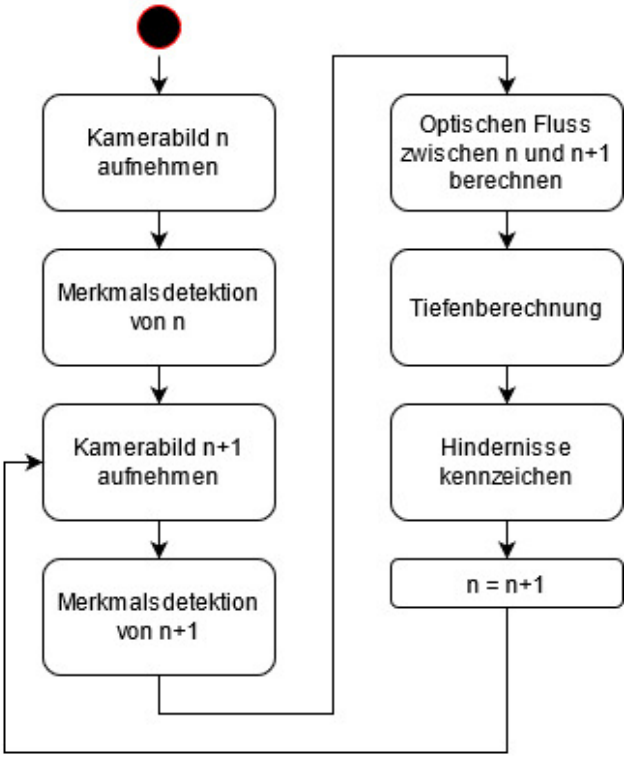


Abbildung 42: Programmaufbau Ansatz 3 Variante 2

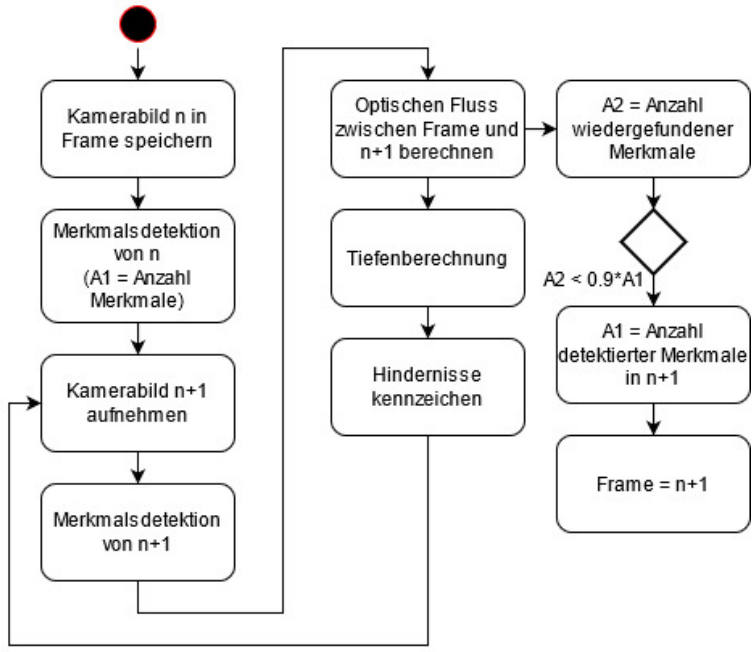


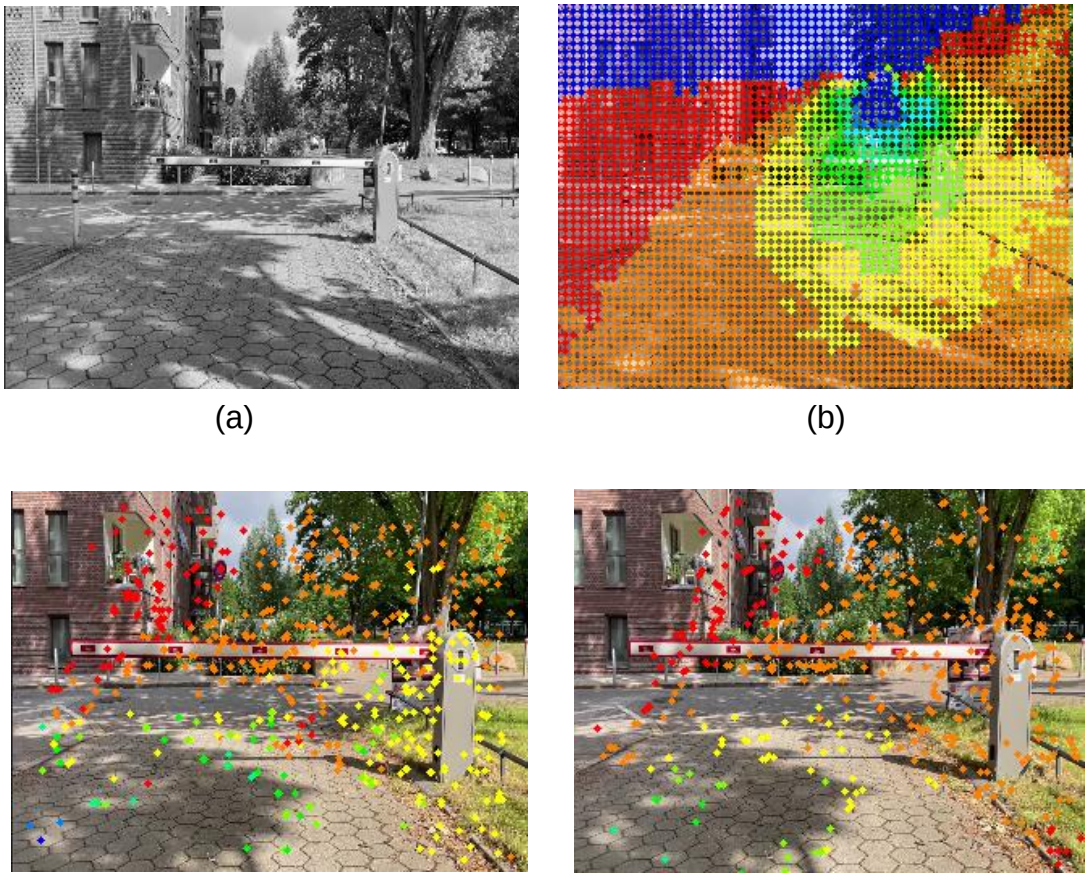
Abbildung 43: Programmaufbau Ansatz 3 Variante 3

5.5.1 Testergebnisse

Die verwendeten Bilder und Bewegungsdaten werden, wie in Kapitel 3.3 beschrieben, aufgenommen. Bei der Umrechnung der Beschleunigungsdaten in Geschwindigkeiten, fällt auf, dass die aufgenommenen Werte der IMU sehr ungenau sind. Im Folgenden werden Beispielaufnahmen der Ergebnisse der drei Varianten dargestellt, bei denen die Bewegungsdaten durch die IMU aufgenommen wurden. Der Prototyp wurde dabei, abgesehen von kleinen Abweichungen durch Unebenheiten im Boden, ausschließlich in die Z-Richtung bewegt.
Es wird eine Farbskala für die Darstellung der Tiefenwerte verwendet. Diese Farbskala mit den dazugehörigen RGB-Werten wird in der Tabelle 3 dargestellt.

Tiefenwerte Z in m	Farbe	RGB-Werte
0 – 1		255, 0, 0
1 – 2		255, 128, 0
2 – 3		255, 255, 0
3 – 4		128, 255, 0
4 – 5		0, 255, 0
5 – 6		0, 255, 128
6 – 7		0, 255, 255
7 – 8		0, 128, 255
> 8		0, 0, 255
< 0		0, 0, 255

Tabelle 3: Verwendete Farbskala für die Tiefenwerte



(a) (b)

(c) (d)

Abbildung 44: Fehlerhafte Tiefenberechnung Ansatz 3 a) Eingangsbild Variante 1 b) Tiefenberechnung mittels Variante 1 c) Tiefenberechnung mittels Variante 2 d) Tiefenberechnung mittels Variante 3

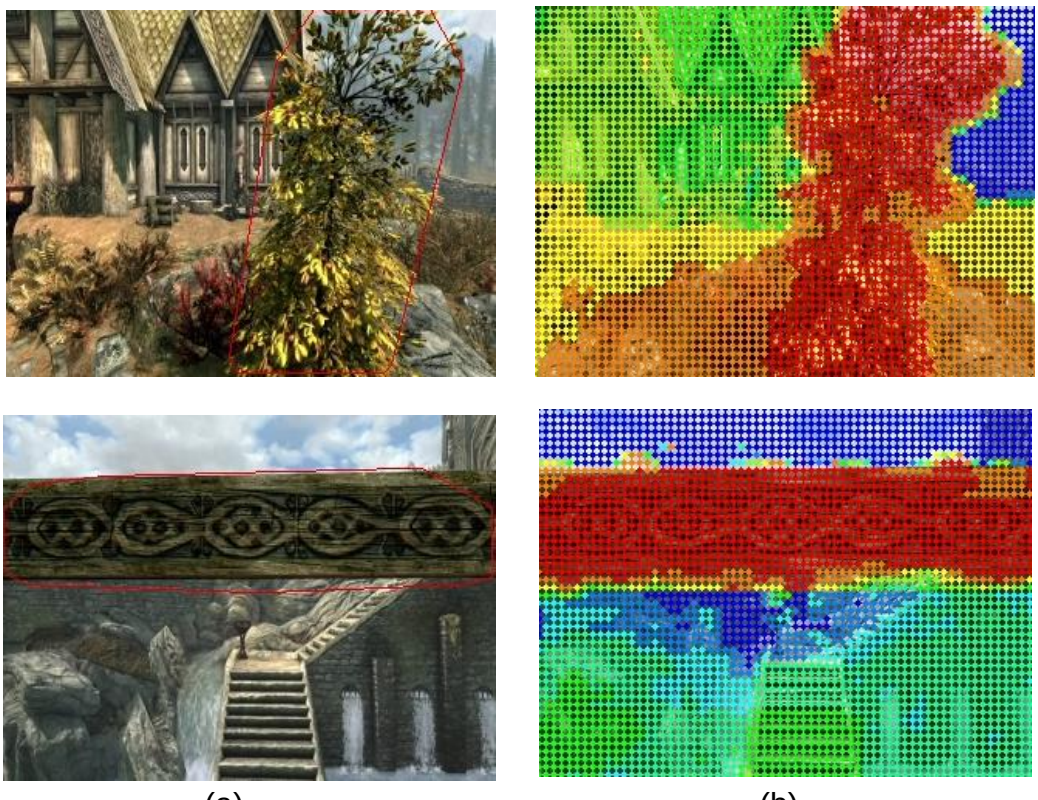
Wie auf den Bildern ersichtlich ist, sind die berechneten Tiefenwerte der Pixel fehlerhaft. Diese fehlerhafte Berechnung ist auf die ungenauen Bewegungsdaten, die durch die IMU aufgenommen wurden, zurückzuführen. Um die vorgestellten Varianten evaluieren zu können, werden virtuelle Aufnahmen verwendet. Die Aufnahmen werden in dem Computerspiel „*The Elder Scrolls: Skyrim*“ mit Hilfe des eingebauten Kamera-Modus aufgenommen. Dabei wird die Kamera ausschließlich gleichförmig in Z-

Richtung bewegt. In der Implementierung werden die Werte für die Rotation sowie die Werte für die Geschwindigkeit in X- und in Y-Richtung auf 0 gesetzt. Für die Geschwindigkeit in Z-Richtung wird ein fester Wert von 1 m/s festgelegt. Zwar stimmt die Geschwindigkeit in Z-Richtung nicht mit der tatsächlichen überein, doch führt dies nur zu Verschiebungen innerhalb der verwendeten Skala.

Im Folgenden werden die Testergebnisse der drei Varianten bei Verwendung der virtuellen Aufnahmen dargestellt. Es handelt sich um Aufnahmen von einem Baum und einem Tor. Es wird die zuvor beschriebene Farbskala für die berechneten Tiefenwerte verwendet. Hindernisse entsprechen Objekten mit einer maximalen Entfernung von 1 m zur Kamera und werden durch eine rote Linie umrandet.

Variante 1 (Tiefenberechnung mit dem Farnebäck-Algorithmus):

Bei Verwendung der virtuellen Aufnahmen werden plausible Tiefenwerte berechnet. Dabei ist eine eindeutige Trennung von herannahenden und fernen Objekten möglich. Bei dieser Variante werden Tiefenwerte für jeden 5. Pixel berechnet. Jedoch kann, wenn nötig, auch für jeden Pixel ein Tiefenwert berechnet werden. In der Abbildung 45 werden Beispielaufnahmen von den berechneten Tiefenwerten und der dazugehörigen Kennzeichnung von Hindernissen dargestellt. Da diese Objekte Tiefenwerte von weniger als 1 m besitzen, werden diese als Hindernis markiert.



(a) (b)

Abbildung 45: Tiefenberechnung Ansatz 3 Variante 1 a) Eingangsbilder mit Kennzeichnung der Hindernisse b) die dazugehörigen Tiefenwerte

Die Berechnungsdauer eines Bildes, bei der zwei aufeinanderfolgende Bilder der Größe 320 x 240 Pixeln betrachtet werden, beträgt ca. 31 Millisekunden. Somit ergibt sich eine Bildrate von 32 Bildern pro Sekunde.

Variante 2 (Tiefenberechnung mit dem Lucas-Kanade-Algorithmus):

Bei der Variante 2 wird der Lucas-Kanade-Algorithmus für die Berechnung des optischen Flusses zwischen zwei aufeinanderfolgenden Bildern verwendet. Die Tiefenwerte werden wie bei der Variante 1 berechnet. Die Ergebnisse der Tiefenberechnung sowie die Kennzeichnung der Hindernisse werden in der folgenden Abbildung 46 dargestellt. Obwohl für die Kennzeichnung von Hindernissen insgesamt weniger Tiefenwerte als in Variante 1 vorhanden sind, werden diese dennoch ausreichend gekennzeichnet. Eine Detektion von herannahenden Objekten ist möglich.

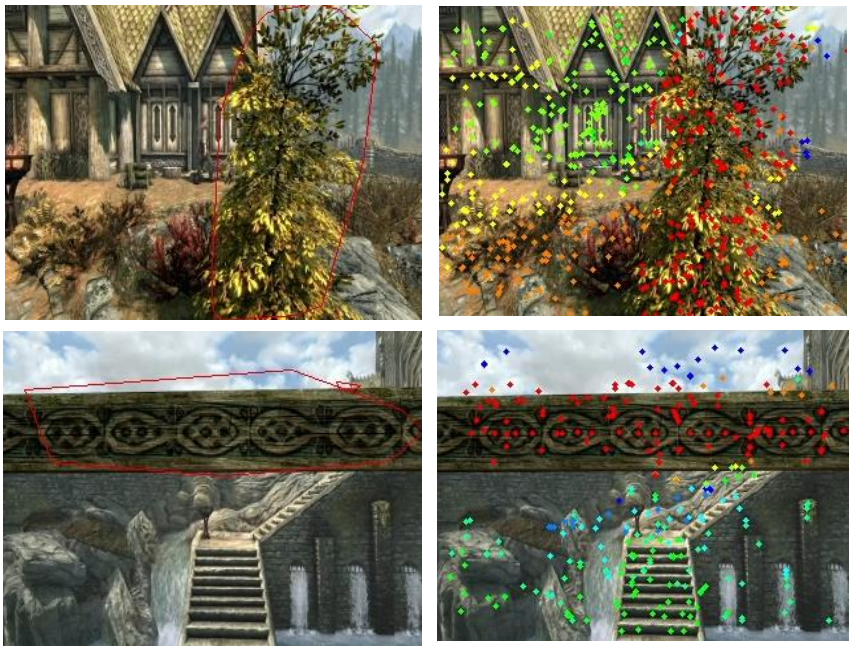
(a) (b)

Abbildung 46: Tiefenberechnung Ansatz 3 Variante 2 a) Eingangsbilder mit Kennzeichnung der Hindernisse b) die dazugehörigen Tiefenwerte

Die Berechnungsdauer eines Bildes, bei der zwei aufeinanderfolgende Bilder der Größe 320 x 240 Pixeln betrachtet werden, beträgt ca. 31 Millisekunden. Somit ergibt sich eine Bildrate von 32 Bildern pro Sekunde.

Variante 3 (Tiefenberechnung mit dem Lucas-Kanade-Algorithmus zwischen zeitlich weiter entfernten Bildern):

In dieser Variante wird ebenfalls der Lucas-Kanade-Algorithmus für die Berechnung des optischen Flusses verwendet. Dabei werden jedoch, wie in Kapitel 5.5 beschrieben Bilder verwendet, die einen größeren zeitlichen Abstand zueinander haben. Die Tiefenwerte werden auf die gleiche Weise wie in den Varianten zuvor berechnet. In der folgenden Abbildung 47 sind die Ergebnisse der Evaluierung der Variante 3 bei den virtuellen Aufnahmen dargestellt. Die Anzahl der berechneten Tiefenwerte entspricht ungefähr der von Variante 2. Auch hier ist eine Trennung von herannahenden Objekten möglich.



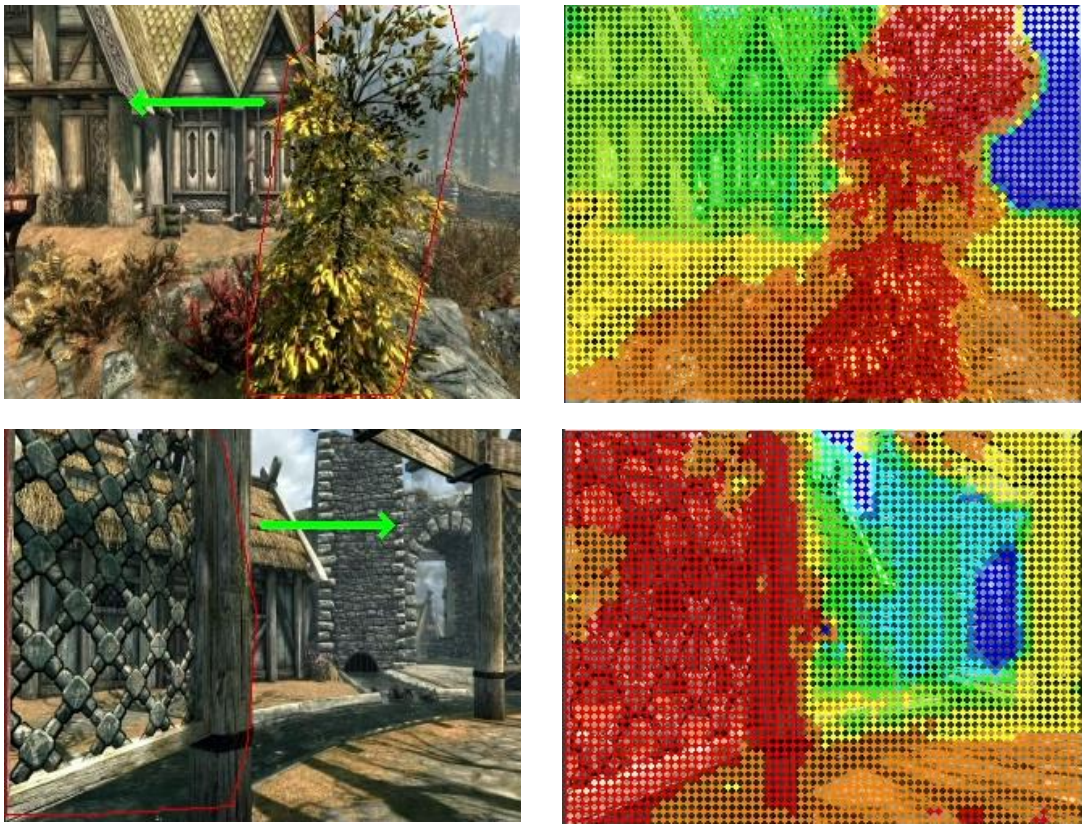
(a) (b)

Abbildung 47: Tiefenberechnung Ansatz 3 Variante 3 a) Eingangsbilder mit Kennzeichnung der Hindernisse b) die dazugehörigen Tiefenwerte

Die Berechnungsdauer eines Bildes, bei der zwei aufeinanderfolgende Bilder der Größe 320 x 240 Pixeln betrachtet werden, beträgt ca. 31 Millisekunden. Somit ergibt sich eine Bildrate von 32 Bildern pro Sekunde.

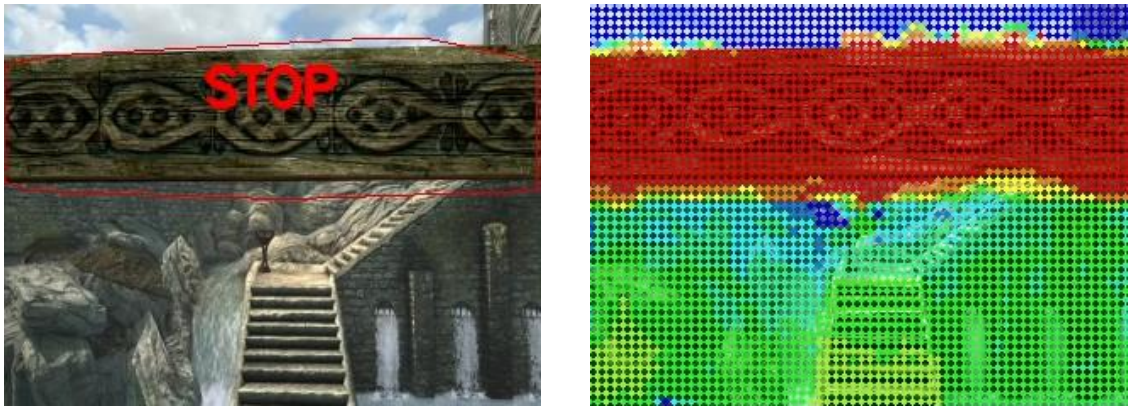
Richtungsanweisungen:

Im Folgenden werden die Ergebnisse der Generierung von Richtungsanweisungen vorgestellt. Dazu werden in der Abbildung 48 und Abbildung 49 Beispielaufnahmen der Variante 1 mit Richtungsanweisungen bei detektierten Hindernissen und den dazugehörigen Tiefenbildern dargestellt.



(a) (b)

Abbildung 48: Generierte Richtungsanweisungen a) Eingangsbilder mit Kennzeichnung der Hindernisse und Richtungsanweisungen b) die dazugehörigen Tiefenwerte



(a) (b)

Abbildung 49: Generierte Richtungsanweisung a) Eingangsbild mit Kennzeichnung des Hindernisses und Richtungsanweisung b) die dazugehörigen Tiefenwerte

6 Auswertung

In den Kapiteln 5.3 bis 5.5 werden die Testergebnisse der implementierten Ansätze vorgestellt und beschrieben. In den folgenden Unterkapiteln werden diese Testergebnisse anhand der in Kapitel 3.1 festgelegten Kriterien ausgewertet. Auf das Kriterium der monokularen Kamera wird nicht eingegangen, da alle implementierten Ansätze auf die Verwendung der Aufnahmen einer solchen Kamera ausgelegt sind. Nach der Auswertung wird der Ansatz ausgewählt, der für die Anwendung am besten geeignet ist. Zudem werden weitere Testergebnisse vorgestellt. Zum Schluss wird ein Ausblick auf Verbesserungsmöglichkeiten gegeben.

6.1 Auswertung des Ansatzes 1

Im Folgenden werden die Testergebnisse des Ansatzes 1 anhand der Kriterien aus Kapitel 3.1 ausgewertet. Hierzu wird auf die Aspekte der verwendeten Hintergrundsubtraktion, des Template Matchings und der Berechnung des optischen Flusses eingegangen.

Die Detektion von herannahenden Hindernissen durch das verwendete Template Matching ist grundsätzlich möglich, wie in Abbildung 29 zu sehen ist. Jedoch werden Objekte mit glatten und einfarbigen Oberflächen aufgrund des Schrittes der Hintergrundsubtraktion und Filterung nicht zuverlässig detektiert. Außerdem ist die Detektion bei anderen Objekten aufgrund der Vorgehensweise, wie einzelne Bildausschnitte miteinander verglichen werden, häufig falsch. Dies ist in Abbildung 34 verdeutlicht. Die Trennung von nahen und fernen Objekten wird über die Berechnung des optischen Flusses realisiert. Jedoch ist das Ergebnis dieser Berechnung, wie in Abbildung 30 dargestellt, häufig nicht eindeutig genug, um eine zuverlässige Trennung zu gewährleisten.

Herabhängende Hindernisse sowie feststehende Hindernisse werden aufgrund der zuvor genannten Gründe nicht zuverlässig detektiert. Dabei ist kein Unterschied der Ergebnisse bei verschiedenen Arten von Hindernissen festzustellen.

Die Berechnungsdauer dieser Implementierung beträgt 14 Bilder pro Sekunde und stimmt somit mit dem festgelegten Kriterium von mindestens 10 Bildern pro Sekunde überein.

6.2 Auswertung des Ansatzes 2

An dieser Stelle werden die Ergebnisse der durchgeführten Tests des Ansatzes 2 anhand der Kriterien aus dem Kapitel 3.1 ausgewertet.

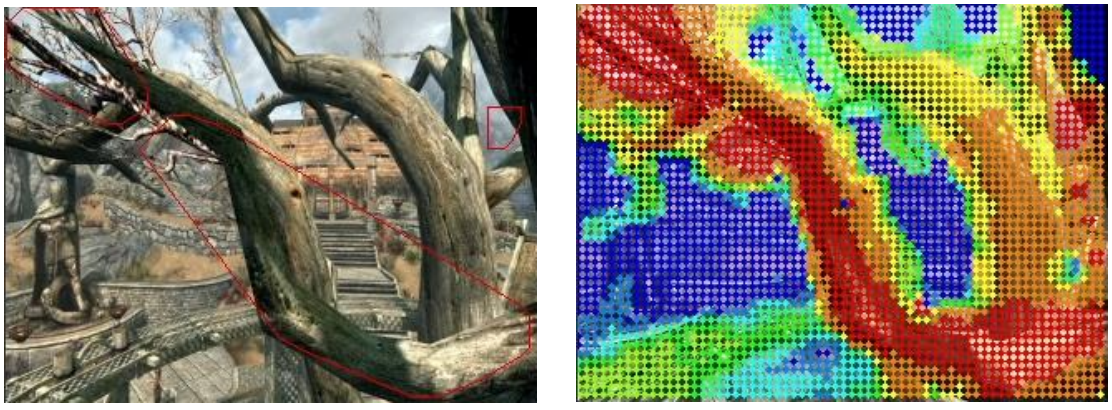
Die grundsätzliche Detektion von herannahenden Hindernissen ist möglich. Dabei werden feststehende sowie herabhängende Objekte erkannt. Der Abstand, ab dem Hindernisse als solche klassifiziert werden, ist jedoch klein, sodass ein rechtzeitiges Ausweichen nicht möglich ist. Eine Anpassung der für das Template Matching verwendeten Skalierungsgrößen führt zu keiner ausreichenden Verbesserung. Die Verwendung von zeitlich weiter entfernten Bildern führt zwar zu einem vergrößerten Abstand, jedoch steigt dadurch auch die Berechnungsdauer an.
Außerdem ist die Anzahl an falsch-positiven Ergebnissen durch die fehlerhafte Paarung von Merkmalen in aufeinanderfolgenden Bildern hoch. Somit ist eine zuverlässige Trennung von herannahenden und fernen Objekten nicht möglich.

Die Berechnungsdauer dieser Implementierung führt zu einer Bildrate von 26 Bildern pro Sekunde und entspricht somit dem geforderten Kriterium von mindestens 10 Bildern pro Sekunde.

6.3 Auswertung des Ansatzes 3

In diesem Unterkapitel werden die Testergebnisse der einzelnen Varianten der Implementierung des Ansatzes 3 aus dem Kapitel 5.5.1 im Hinblick auf die in Kapitel 3.1 festgelegten Kriterien ausgewertet.

Anders als in den Ansätzen zuvor werden in diesem Ansatz exakt berechnete Tiefenwerte für die Detektion von Hindernissen verwendet. Bei allen implementierten Varianten werden herabhängende sowie feststehende Objekte zuverlässig detektiert. Ein Unterschied bei der Detektion von verschiedenen Arten von Objekten ist nicht festzustellen. Durch die Berechnung einzelner Tiefenwerte ist eine zuverlässige Trennung von herannahenden und fernen Objekten bei allen Varianten möglich. Der Unterschied zwischen Variante 2 und 3 liegt in dem zeitlichen Abstand der betrachteten Bilder. Dabei ist kein signifikanter Unterschied zwischen den Ergebnissen der beiden Varianten festzustellen. Die Variante 1 generiert verglichen mit den anderen Varianten die meisten Tiefeninformationen, da bei dieser die Tiefe für jeden einzelnen Bildpixel berechnet werden kann. Dadurch ist eine genaue Trennung zwischen Hindernissen möglich. Bei dieser Trennung bleiben Informationen feiner Strukturen, wie z. B. einzelner Äste eines Baumes erhalten. Ein Beispiel hierfür ist in der Abbildung 50 dargestellt.



(a) (b)

Abbildung 50: Tiefenberechnung Ansatz 3 Variante 1 bei feinen Strukturen a) Eingangsbild mit Kennzeichnung der Hindernisse b) die dazugehörigen Tiefenwerte

Anzumerken ist, dass bei den drei Varianten die Berechnung der Tiefe von Pixeln, die sich genau in der Bildmitte befinden, beim Fortbewegen in eine bestimmte Richtung oft fehlerhaft ist. Dies liegt daran, dass diese Pixel in aufeinanderfolgenden Bildern eine sehr geringe Distanz zurücklegen und hierdurch die Tiefe falsch berechnet werden kann. Da die Anzahl dieser Pixel jedoch gering ist, werden diese durch den Schritt der Gruppierung gefiltert.

Die Berechnungsdauer der drei Varianten ist nahezu identisch und führt zu einer Bildrate von ca. 31 Bildern pro Sekunde. Diese liegt somit über der festgelegten Bildrate von mindestens 10 Bildern pro Sekunde.

6.4 Fazit und Ausblick

Das Ziel dieser Arbeit ist, eine Hindernisdetektion mittels einer monokularen Kamera zu implementieren. Dazu wurde der Stand der Technik zur Hindernisdetektion ermittelt. Nach einer Auswertung anhand zuvor definierter Kriterien wurden Ansätze für die Implementierung festgelegt. Im Anschluss der Erläuterung der zu implementierenden Methoden wurden die ausgewählten Ansätze implementiert und getestet.

Die Ergebnisse der Ansätze 1 und 2, bei denen die Methoden Template Matching und optischer Fluss verwendet werden, zeigen, dass diese unzuverlässig in der Detektion von Hindernissen sind. Diese Ansätze werden nicht für die Hindernisdetektion des „*Shared Guide Dog 4.0*“ empfohlen. Der Ansatz 3, bei dem Tiefeninformationen mittels optischen Flusses generiert werden, liefert insgesamt die besten Ergebnisse bei virtuellen Aufnahmen. Bei den verwendeten Aufnahmen wurden die Hindernisse zuverlässig detektiert.
Die Varianten 2 und 3 des Ansatzes 3 sind dabei von der Detektion von Merkmalen abhängig. Durch die Ergebnisse der Ansätze 1 und 2 ist deutlich, dass eine Detektion von Merkmalen auf spiegelnden Oberflächen oder Oberflächen ohne ausgeprägte Textur nicht gewährleistet ist. Deshalb kann davon ausgegangen werden, dass die Detektion bei solchen Objekten bei Verwendung der Varianten 2 und 3 unzuverlässig ist.
Da die Variante 1 unabhängig von der Detektion von Merkmalen ist, kann davon ausgegangen werden, dass die Erkennung von solchen Objekten auch bei realen Aufnahmen zuverlässig ist. Dabei liefert die Variante 1 dieses Ansatzes die meisten Tiefeninformationen bei gleichbleibender Berechnungsdauer.

Für die Anwendung der Variante 1 des Ansatzes 3 als Hindernisdetektion des „*Shared Guide Dog 4.0*“ muss die Zuverlässigkeit bei realen Aufnahmen und Verwendung einer inertialen Messeinheit, die genaue Bewegungsdaten liefert, evaluiert werden. Dabei sollte der Zusatznutzen der Ergebnisse dieser Implementierung bei Verwendung einer solchen Messeinheit gegenüber den Ergebnissen bei Verwendung einer 3D-Kamera hinsichtlich Anschaffungskosten und Mehrwert überprüft werden.
Da dieser Ansatz auf der Berechnung des optischen Flusses basiert, kann eine Hindernisdetektion nur bei einer Bewegung gewährleistet werden. Deshalb wird die Verwendung eines zusätzlichen Sensors, wie z. B. eines Ultraschallsensors für die Hindernisdetektion im Stillstand empfohlen.

Aufgrund fehlender Trainingsdaten und begrenzter Bearbeitungszeit dieser Arbeit wurde keine Hindernisdetektion basierend auf maschinellem Lernen implementiert. Deshalb sollte die Eignung für die Hindernisdetektion des „*Shared Guide Dog 4.0*“ eines solch basierenden Ansatzes, sofern Trainingsdaten vorhanden sind, ebenfalls evaluiert werden.

Literaturverzeichnis

1. OpenCV Documentation 4.5.2. https://docs.opencv.org/4.5.2/. Abgerufen 26 Jul 2021

2. Jung YJ, Baik A, Kim J et al. (2009) A novel 2D-to-3D conversion technique based on relative height-depth cue. In: Woods AJ, Holliman NS, Merritt JO (eds) Stereoscopic Displays and Applications XX. SPIE, 72371U

3. Tang C, Hou C, Song Z (2015) Depth recovery and refinement from a single image using defocus cues. Journal of Modern Optics 62:441–448. https://doi.org/10.1080/09500340.2014.967321

4. Levin A, Lischinski D, Weiss Y (2004) Colorization using optimization. ACM Trans Graph 23:689–694. https://doi.org/10.1145/1015706.1015780

5. Martinello M, Favaro P (2012) Depth estimation from a video sequence with moving and deformable objects. In: IET Conference on Image Processing (IPR 2012). IET, p 131

6. Fan J, Chen M, Mo J et al. (2021) Variational formulation of a hybrid perspective shape from shading model. Vis Comput. https://doi.org/10.1007/s00371-021-02081-x

7. Schlick C (1994) An Inexpensive BRDF Model for Physically-based Rendering. Computer Graphics Forum 13:233–246. https://doi.org/10.1111/1467-8659.1330233

8. Loh AME, Hartley R (2005) Shape from non-homogeneous, non-stationary, anisotropic, perspective texture. In: Clocksin WF, Fitzgibbon AW, Torr PHS (eds) Procedings of the British Machine Vision Conference 2005. British Machine Vision Association, 8.1-8.10

9. Matas J, Chum O, Urban M et al. (2004) Robust wide-baseline stereo from maximally stable extremal regions. Image and Vision Computing 22:761–767. https://doi.org/10.1016/j.imavis.2004.02.006

10. Lowe DG (2004) Distinctive Image Features from Scale-Invariant Keypoints. International Journal of Computer Vision 60:91–110. https://doi.org/10.1023/B:VISI.0000029664.99615.94

11. Ming A, Wu T, Ma J et al. (2016) Monocular Depth-Ordering Reasoning with Occlusion Edge Detection and Couple Layers Inference. IEEE Intell Syst 31:54–65. https://doi.org/10.1109/MIS.2015.94

12. Battiato S, Curti S, La Cascia M et al. (2004) Depth map generation by image classification. In: Corner BD, Li P, Pargas RP (eds) Three-Dimensional Image Capture and Applications VI. SPIE, p 95

13. I. Ulrich, I. Nourbakhsh (2000) Appearance-Based Obstacle Detection with Monocular Color Vision. In: AAAI/IAAI

14. J. M. Sagar (2014) Obstacle Avoidance using Monocular Vision on Micro Aerial Vehicles. Masterarbeit, University of Amsterdam

15. Shi J, Tomasi (1994) Good features to track. In: Proceedings of IEEE Conference on Computer Vision and Pattern Recognition CVPR-94. IEEE Comput. Soc. Press, pp 593–600

16. Mori T, Scherer S (2013) First results in detecting and avoiding frontal obstacles from a monocular camera for micro unmanned aerial vehicles. In: 2013 IEEE International Conference on Robotics and Automation. IEEE, pp 1750–1757

17. Bay H, Ess A, Tuytelaars T et al. (2008) Speeded-Up Robust Features (SURF). Computer Vision and Image Understanding 110:346–359. https://doi.org/10.1016/j.cviu.2007.09.014

18. Saxena A, Chung S, Ng A (2006) Learning Depth from Single Monocular Images. In: Y. Weiss, B. Schölkopf, J. Platt (eds) Advances in Neural Information Processing Systems, vol 18. MIT Press

19. Karsch K, Liu C, Kang SB (2014) Depth Transfer: Depth Extraction from Video Using Non-Parametric Sampling. IEEE Trans Pattern Anal Mach Intell 36:2144–2158. https://doi.org/10.1109/TPAMI.2014.2316835

20. Oliva A, Torralba A (2001) Modeling the Shape of the Scene: A Holistic Representation of the Spatial Envelope. International Journal of Computer Vision 42:145–175. https://doi.org/10.1023/A:1011139631724

21. Liu C, Yuen J, Torralba A (2011) SIFT flow: dense correspondence across scenes and its applications. IEEE Trans Pattern Anal Mach Intell 33:978–994. https://doi.org/10.1109/TPAMI.2010.147

22. Godard C, Aodha OM, Firman M et al. (2018) Digging Into Self-Supervised Monocular Depth Estimation

23. Chiu M-J, Chiu W-C, Chen H-T et al. (2021) Real-time Monocular Depth Estimation with Extremely Light-Weight Neural Network. In: 2020 25th International Conference on Pattern Recognition (ICPR). IEEE, pp 7050–7057

24. Sandler M, Howard A, Zhu M et al. (2018) MobileNetV2: Inverted Residuals and Linear Bottlenecks. In: 2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition. IEEE, pp 4510–4520

25. Bouguet J-Y (2001) Pyramidal implementation of the affine lucas kanade feature tracker description of the algorithm. Intel corporation 5:4

26. Farnebäck G (2003) Two-Frame Motion Estimation Based on Polynomial Expansion. In: Bigun J, Gustavsson T (eds) Image Analysis. Springer Berlin Heidelberg, Berlin, Heidelberg, pp 363–370

27. David J. Heeger (1996) Notes on Motion Estimation. https://www.cns.nyu.edu/~david/handouts/motion.pdf. Abgerufen 21 Jul 2021

28. Template Matching. https://docs.opencv.org/4.5.2/d4/dc6/tutorial_py_template_matching.html. Abgerufen 27 Jul 2021

29. Jähne B (2005) Digitale Bildverarbeitung, 6., überarbeitete und erweiterte Auflage. Springer-Verlag Berlin Heidelberg, Berlin, Heidelberg

30. R. Neubecker 1 Bildverarbeitung I Bildvergleich. https://docplayer.org/187576047-Bildverarbeitung-i-bildvergleich.html. Abgerufen 27 Jul 2021

31. M. Riegler (2015) A study and comparison of feature matching. Masterarbeit, Hochschule München

32. Dr. Bernard Haasdonk (2021) Digitale Bildverarbeitung. https://lmb.informatik.uni-freiburg.de/people/haasdonk/DBV_FHO/DBV_FHO_SS08_E10.pdf

33. Andre Sanders (2017) Ein mobiler Augmented-Reality-Ansatz zur Erkennung kontextabhängiger Teilbilder mittels Feature-Matching. Bachelorarbeit, Universität Osnabrück

34. Dirk Loss (2002) Data Mining: Klassifikations- und Clusteringverfahren. Ausarbeitung, Westfälische Wilhelms-Universität Münster

Abbildungsverzeichnis

Abbildung 1: Prototyp des Shared Guide Dog 4.0 2
Abbildung 2: Tiefenberechnung durch relative Höhe 5
Abbildung 3: Zusammenhang zwischen Schärfe / Unschärfe und Distanz 7
Abbildung 4: Tiefenberechnung durch Schärfe / Unschärfe 1 8
Abbildung 5: Codierte Maske 9
Abbildung 6: Tiefenberechnung durch Schärfe / Unschärfe 2 10
Abbildung 7: Tiefenberechnung durch Schattierung 11
Abbildung 8: Berechnung einer Form durch Textur 12
Abbildung 9: Tiefenberechnung durch Kantenverdeckung 13
Abbildung 10: Fluchtlinien und Fluchtpunkt 15
Abbildung 11: Tiefenberechnung durch Perspektive 16
Abbildung 12: Hinderniserkennung durch Aussehen 18
Abbildung 13: Konturendetektion und Filterung 19
Abbildung 14: Segmentierung mittels optischen Flusses 20
Abbildung 15: Kennzeichnung der Hindernisse 20
Abbildung 16: Hinderniskennzeichnung 22
Abbildung 17: Tiefenberechnung durch parametrisches Lernen 23
Abbildung 18: Tiefenberechnung durch nicht-parametrisches Lernen 25
Abbildung 19: Tiefenberechnung durch unüberwachtes maschinelles Lernen 27
Abbildung 20: Tiefenberechnung und Segmentierung durch überwachtes maschinelles Lernen 29
Abbildung 21: Prototyp des Shared Guide Dog 4.0 36
Abbildung 22: Verwendete Koordinatensysteme 42
Abbildung 23: Suchprinzip des Template Matchings 45
Abbildung 24: Verwendetes Kalibrierungsmuster für die Kamerakalibrierung 50
Abbildung 25: Beispielaufnahme der Webcam 51
Abbildung 26: Hintergrundsubtraktion 53
Abbildung 27: Ausgangsbild nach der Filterung 54
Abbildung 28: Merkmalsdetektion und Gruppierung 55

Abbildung 29: Ergebnisse nach Anwendung des Template Matchings 56
Abbildung 30: Ergebnisse nach Anwendung der Methode des optischen Flusses 57
Abbildung 31: Programmaufbau Ansatz 1 58
Abbildung 32: Fehlerhafte Hintergrundsubtraktion 59
Abbildung 33: Merkmalsdetektion und Gruppierung ohne Hintergrundsubtraktion 60
Abbildung 34: Beispielaufnahme einer fehlerhaften Hindernisdetektion 61
Abbildung 35: Detektierte Merkmale im Eingangsbild.....62
Abbildung 36: Gepaarte Merkmale in aufeinanderfolgenden Bildern.....63
Abbildung 37: Ergebnis der Hindernisdetektion 64
Abbildung 38: Programmaufbau Ansatz 2 65
Abbildung 39: Fehlerhafte Merkmalspaarung und die daraus resultierende Hindernisdetektion 66
Abbildung 40: Aufteilung des Eingangsbildes in drei Bereiche 70
Abbildung 41: Programmaufbau Ansatz 3 Variante 1 72
Abbildung 42: Programmaufbau Ansatz 3 Variante 2 73
Abbildung 43: Programmaufbau Ansatz 3 Variante 3 74
Abbildung 44: Fehlerhafte Tiefenberechnung Ansatz 3 76
Abbildung 45: Tiefenberechnung Ansatz 3 Variante 1 78
Abbildung 46: Tiefenberechnung Ansatz 3 Variante 2 79
Abbildung 47: Tiefenberechnung Ansatz 3 Variante 3 80
Abbildung 48: Generierte Richtungsanweisungen 81
Abbildung 49: Generierte Richtungsanweisung 82
Abbildung 50: Tiefenberechnung Ansatz 3 Variante 1 bei feinen Strukturen .86

Tabellenverzeichnis

Tabelle 1: Verwendete Richtungsanweisungen 70
Tabelle 2: Bedingungen für Zuweisung der Richtungsanweisungen 71
Tabelle 3: Verwendete Farbskala für die Tiefenwerte 75

Kennzeichnung der verfassten Kapitel / Seiten

Andrej Hofmann verfasste folgende Kapitel: 2.1, 2.3, 2.4, 2.6, 2.8, 2.10, 2.13, 3.1, 3.2, 3.2.1, 3.2.2, 3.2.3, 3.2.4, 4.1, 5.4, 6.2, 6.4 und folgende Seiten: 30, 51, 68, 69, 75 – 78.

Baljinder Singh verfasste folgende Kapitel: 1, 2.2, 2.5, 2.7, 2.9, 2.11, 2.12, 2.14, 3.2.5, 3.3, 4.2, 4.3, 4.4, 4.5, 5.1, 5.2, 5.3, 6.1, 6.3 und folgende Seiten: 4, 38, 70 – 74, 79 – 83.

Anhang A: CD

Ordner *Bachelorarbeit:*

- *Hinderniserkennung_AH_BS.pdf*

Ordner *Quellcode*:

- *Ansatz1.cpp*
- *Ansatz2.cpp*
- *Ansatz3Variante1.cpp*
- *Ansatz3Variante2.cpp*
- *Ansatz3Variante3.cpp*

Anhang B: Quellcode

Ansatz 1

```
1.  /*
2.  Shared Guide Dog - Hinderniserkennung / Erkennung von herabhängenden Objekten mi
    t einer monokularen Kamera
3.  Andrej Hofmann und Baljinder Singh
4.  15.08.2021
5.  Ansatz 1 nach Jaysinh Sagar: Obstacle Avoidance Using Monocular Vision In Micro
    Aerial Vehicles
6.  Link: https://staff.fnwi.uva.nl/a.visser/education/masterProjects/Obstacle%20Avo
    idance%20using%20Monocular%20Vision%20on%20Micro%20Aerial%20Vehicles_final.pdf
7.  */
8.
9.  #include <iostream>
10. #include <windows.h>
11. #include <opencv2/opencv.hpp>
12. #include <opencv2/core.hpp>
13. #include "opencv2/features2d.hpp"
14. #include "opencv2/xfeatures2d.hpp"
15.
16. using namespace cv;
17. using namespace std;
18. using namespace cv::xfeatures2d;
19.
20.
21. bool pointComp(const cv::Point2f& a, const cv::Point2f& b) { //Methode, die zwei
     Punkte hinsichtlich der nähe zum Ursprung vergleicht
22.   return ((a.x * a.x + a.y * a.y) < (b.x * b.x + b.y * b.y)); //Methode von: htt
    ps://stackoverflow.com/questions/16939147/how-to-organize-or-sort-a-stdvector-
    cvpoint2f/16939266#16939266, Zul. aufgerufen am 15.08.2021
23. }
24.
25. void bgSubtractor(Mat inputFrame, Ptr<BackgroundSubtractor>& BackSub1, Mat& outp
    utFrame) { //Methode für das Entfernen des Hintergrunds des Eingangsbildes
26.   vector<vector<Point>> contours; //Vektor um gefundene Konturen zu speichern
27.   vector<Vec4i> hierarchy; //Vektor um Hierarchie der Konturen zu speichern
```

```
28.   vector<int> index; //Vektor um Stelle der Konturen zu speichern, deren Flaeche
     groesser 0.5% der Eingangsbildgroesse ist
29.   Mat fgMask = Mat::zeros(inputFrame.size(), CV_8U); //Vordergrundmaske in gleic
    her Groeße wie Eingangsbild erstellen
30.
31.   BackSub1->apply(inputFrame, fgMask, 0.01); //Background-
    Subtractor anwenden und Ergebnis in fgMask speichern, mit einer Lernrate von 0.0
    1
32.
33.   findContours(fgMask, contours, hierarchy, RETR_TREE, CHAIN_APPROX_NONE); //Kon
    turen in Vordergrundmaske suchen
34.   for (int i = 0; i < contours.size(); i++) { //Schleife für jede gefundene Kont
    ur
35.
      if (contourArea(contours[i]) > 0.005 * fgMask.size().area()) { //Index der Kon
    turen mit einer Flaeche groesser 0.5% der Eingangsbildgroesse speichern
36.                       index.push_back(i);
37.             }
38.   }
39.
40.   if (!index.empty()) { //Wenn Konturen groesser 0.5% der Eingangsbildgroesse vo
    rhanden sind
41.
      fgMask = Mat::zeros(inputFrame.size(), CV_8U); //fgMask mit Nullen ueberschrei
    ben
42.
      for (int i = 0; i < index.size(); i++) { //Nur die Konturen mit einer Flaeche
    groesser 0.5% der Eingangsbildgroesse in Vordergrundmaske speichern
43.
      drawContours(fgMask, contours, index[i], CV_RGB(255, 255, 255), 1, LINE_AA, hi
    erarchy, 4);
44.             }
45.             index.clear();
46.
47.
      outputFrame = Mat::zeros(inputFrame.size(), CV_8U); //gefilterte Vordergrundma
    ske der Variable outputFrame uebergeben
48.             outputFrame = fgMask;
49.   }
50.   else { //Wenn keine Konturen groesser 0.5% der Eingangsbildgroesse vorhanden s
    ind
51.
      outputFrame = Mat::zeros(inputFrame.size(), CV_8U); //Variable outputFrame mit
     Nullen auffuellen
52.   }
53. }
54.
55. //Methode die Merkmale auf von bgSubtractor gefilterten Bild detektiert,
56. //gruppiert und gefundene Gruppierungen, gefundene Merkmale,
57. //ein Bild mit allen gefundenen Merkmalen
58. //und ein Bild mit den gruppierten Merkmalen ausgibt
59. void featurePoints(Mat inputFrame, Mat filteredFrame, vector<Point2f>& featurePo
    ints, Mat& frameFeatures, Mat& frameCluster, vector<vector<Point2f>>& outputClus
    ter) {
60.   vector<Point2f> cluster, clusterPoints; //Vektor um gefundene Gruppierungen un
    d Punkte dieser Gruppierungen zu speichern
```

```
61.   int minHessian = 800; //Threshold-
    Wert um zu entscheiden ob es sich um einen Feature-Punkt handelt


62.   Ptr<SURF> detector = SURF::create(minHessian); //SURF-
    Merkmalsdetektor initialisieren


63.   vector<KeyPoint> keypoints; //Vektor um alle gefundenen Merkmale zwischenzuspe
    ichern
64.
65.   detector-
    >detect(filteredFrame, keypoints, noArray()); //Merkmale auf gefiltertem Eingang
    sbild detektieren
66.
67.   for (int i = 0; i < keypoints.size(); i++) { //Gefundene Feature-
    Punkte in featurePoints speichern
68.             featurePoints.push_back(keypoints[i].pt);
69.   }
70.
71.   if (!featurePoints.empty()) { //Gruppierung der Merkmale
72.
      sort(featurePoints.begin(), featurePoints.end(), pointComp); //Punkte in featu
    rePoints nach Abstand zum Ursprung in steigender Reihenfolge sortieren
73.             for (int i = 0; (i + 1) < featurePoints.size(); i++) {
74.
      double distance = norm(featurePoints[i + 1] - featurePoints[i]); //Distanz zwi
    schen Merkmalen berechnen
75.
      if (distance < 30) { //Wenn Distanz unter threshold ist, wird das Merkmal in c
    luster gespeichert
76.                                 cluster.push_back(featurePoints[i]);
77.                         }
78.
      if (distance > 30) { // Wenn die Anzahl der Merkmale in der Gruppierung über e
    inem Schwellwert ist, wird die aktuelle Gruppierung gespeichert
79.                                 if (cluster.size() > 3) {
80.
      for (int y = 0; y < cluster.size(); y++) {
81.
      outputCluster.push_back(cluster);
82.
      clusterPoints.push_back(cluster[y]);
83.                                                 }
84.                                 }
85.                                 cluster.clear();
86.                         }
87.             }
88.   }
89.
90.   inputFrame.copyTo(frameFeatures); //Bild mit allen detektierten Merkmalen gene
    rieren
91.   for (int i = 0; i < featurePoints.size(); i++) {
92.             circle(frameFeatures, featurePoints[i], 2, CV_RGB(255, 0, 0), -1);
93.   }
94.
95.   inputFrame.copyTo(frameCluster); //Bild mit gruppierten Merkmalen generieren
96.   for (int i = 0; i < clusterPoints.size(); i++) {
```

```
97.             circle(frameCluster, clusterPoints[i], 2, CV_RGB(0, 0, 255), -1);
98.     }
99.
100.            clusterPoints.clear();
101.    }
102.
103.    int main() {
104.            try {
105.
        Mat frame, filteredFgMask, frameFP, frameClustFP, frameOF, frameOld, frameTM;
106.
        bool firstRound = true; //Variable zur Ueberpruefung ob es sich um den ersten
    Durchlauf handelt
107.
108.
        Ptr<BackgroundSubtractor> BackSub = createBackgroundSubtractorMOG2(500, 16, tr
    ue); //Background Subtractor initialisieren
109.
110.
        vector<Point2f> featurePunkte; //Vektor für alle detektierten Merkmale
111.
        vector<vector<Point2f>> cluster; //Vektor um die durch die Methode featurePoin
    ts detektierten Gruppierungen zu speichern
112.
113.
        vector<vector<Mat>> scalesOld; //Vektor um die einzelnen Bildausschnitte des v
    orherigen Frames zu speichern
114.
        vector<Point> pointTopLeftOld; //Vektor um die Position einzelner Bildausschni
    tte des vorherigen Frames zu speichern
115.
        vector<int> widthOld; //Vektor um die Breite einzelner Bildausschnitte des vor
    herigen Frames zu speichern
116.
        vector<int> heightOld; //Vektor um die Hoehe einzelner Bildausschnitte des vor
    herigen Frames zu speichern
117.
118.
        vector<Point2f> p0, p1; //In Vektor p0 werden die detektierten Merkmale des vo
    rherigen Frames und in Vektor p1 werden die durch Optischen Fluss berechneten Pu
    nkte im aktuellen Frame gespeichert
119.
        vector<Point2f> good_new; //Vektor um Punkte die durch optical flow berechnet
    wurden zu speichern
120.
        bool berechnungOF = false; //Variable zur Ueberpruefung ob der optische-
    Fluss berechnet werden konnte
121.
        vector<uchar> status; //Vektor um Berechnung einzelner Merkmale beim optischen
     Fluss zu ueberpruefen
122.
        vector<float> err; //Vektor um Fehler bei der Berechnung des optischen Flusses
     zu speichern
123.
        TermCriteria criteria = TermCriteria((TermCriteria::COUNT)+(TermCriteria::EPS)
    , 30, 0.01); //Austrittsbedingung bei der Berechnung des optischen Flusses
```

```
124.
      vector<double> distances; //Vektor fuer die Berechnung des Mittelwerts beim op
    tischen Fluss
125.
126.
      VideoCapture vc(0); //Kamera oeffnen, falls dies fehlschlaegt Programm beenden
127.                      if (!vc.open(0)) {
128.
      cout << "Kamera konnte nicht geoeffnet werden!" << std::endl;
129.                                  return 0;
130.                      }
131.
132.                      while (true) {
133.
      vc >> frame; //Aktuelles Kamerabild in frame speichern
134.
      if (frame.empty()) { //Beenden der Schleife, wenn keine Daten der Kamera empfa
    ngen werden
135.                                              break;
136.                                  }
137.
      resize(frame, frame, Size(320, 240)); //Aktuelles Eingangbild auf 320x240 Pixe
    l skalieren
138.
139.
      bgSubtractor(frame, BackSub, filteredFgMask); //Methode zur Entfernung des Hin
    tergrunds aufrufen
140.
141.
      featurePoints(frame, filteredFgMask, featurePunkte, frameFP, frameClustFP, clu
    ster); //Methode zur Detektion und Gruppierung von Merkmalen aufrufen
142.
143.                                  //Beginn Abschnitt Template Matching
144.
      vector<Mat> imageTemplates; //Vektor um Bildausschnitte des aktuellen Frames z
    u speichern
145.
      vector<int> width; //Vektor um die Breite der Bildausschnitte des aktuellen Fr
    ames zu speichern
146.
      vector<int> height; //Vektor um die Hoehe der Bildausschnitte des aktuellen Fr
    ames zu speichern
147.
      vector<Point> pointTopLeft; //Vektor um die Positionen einzelner Bildausschnit
    te des aktuellen Frames zu speichern
148.
149.
      frame.copyTo(frameTM); //aktuellen Frame in der Variable frameTM speichern
150.
151.
      if (!cluster.empty()) { //Ausschnitte um die Gruppierungen vom aktuellen Frame
     generieren
152.
      for (int i = 0; i < cluster.size(); i++) {
153.
      imageTemplates.push_back(frame(boundingRect(cluster[i])));
154.
      width.push_back(boundingRect(cluster[i]).width);
```

```
155.
      height.push_back(boundingRect(cluster[i]).height);
156.
      Point topLeft = Point(boundingRect(cluster[i]).x, boundingRect(cluster[i]).y);
157.
      pointTopLeft.push_back(topLeft);
158.                                         }
159.
160.
      vector<vector<Mat>> scales(imageTemplates.size()); //Vektor um die einzelnen B
    ildausschnitte des aktuellen Frames zu speichern
161.
      Mat scaledTemplate; //Variable um den skalierten Bildausschnitt zu speichern
162.
      double newHeight = 0; //Variable um Hoehe der skalierten Bildausschnitte zu be
    rechnen
163.
      double newWidth = 0; //Variable um Breite der skalierten Bildausschnitte zu be
    rechnen
164.
      for (int j = 0; j < imageTemplates.size(); j++) { //Für jeden Ausschnitt 9 Ska
    len generieren
165.                                                    for (int i = 0; i < 9; i++) {
166.
      if (width[j] < height[j]) {
167.
      newWidth = width[j] + i;
168.
      newHeight = round((newWidth / width[j]) * height[j]);
169.                                                                  }
170.                                                                  else {
171.
      newHeight = height[j] + i;
172.
      newWidth = round((newHeight / height[j]) * width[j]);
173.                                                                  }
174.
      resize(imageTemplates[j], scaledTemplate, Size(newWidth, newHeight));
175.
      scales.at(j).push_back(scaledTemplate);
176.                                                    }
177.                                         }
178.
179.
      double maxScale = 0; //Variable um die Skalierung mit der hoechsten Uebereinst
    immung zu speichern
180.
      double maxValOld = 0; //Variable um die Skalierung mit der hoechsten Uebereins
    timmung zu bestimmen
181.
      double minVal; double maxVal; Point minLoc; Point maxLoc; Mat result; //Variab
    len für die Berechnung des Template Matching
182.
      vector<int> everyMaxScale; //Vektor um die alle Skalierungen mit der hoechsten
     Uebereinstimmung zu speichern
183.
184.
      if (!firstRound) { //Ab dem zweiten Durchlauf des Programms
```

```
185.
186.
      for (int j = 0; j < scalesOld.size(); j++) { //Skalierte Ausschnitte des vorhe
    rigen Frames werden mit dem Ausschnitt des aktuellen Frames verglichen
187.                                                                  Rect b;
188.
      b.x = pointTopLeftOld.at(j).x; b.y = pointTopLeftOld.at(j).y; b.width = widthO
    ld.at(j); b.height = heightOld.at(j);
189.
      Mat curTem = frame(b); //Ausschnitt des aktuellen Frames generieren
190.
      for (int i = 0; i < scalesOld.at(j).size(); i++) {
191.
      if (!scalesOld.at(j).at(i).empty()) {
192.
      matchTemplate(scalesOld.at(j).at(i), curTem, result, TM_CCORR_NORMED);
193.
      minMaxLoc(result, &minVal, &maxVal, &minLoc, &maxLoc);
194.
      if (maxVal >= maxValOld) {
195.
      maxValOld = maxVal;
196.
      maxScale = i;
197.
      }
198.                                                                          }
199.                                                                  }
200.
      if ((maxValOld != 0)) {
201.
      everyMaxScale.push_back(maxScale);
202.                                                                  }
203.
      else { //Falls Vergleich der Ausschnitte fehlschlaegt, wird als eine "-
    1" gespeichert
204.
      everyMaxScale.push_back(-1);
205.                                                                  }
206.                                                                  maxValOld = 0;
207.                                                          }
208.
      for (int i = 0; i < scalesOld.size(); i++) { //Skalierung der Bildausschnitte
    farblich in frameTM markieren, blau = niedrige Skalierung, rot = hohe Skalierung
209.
      Point mitte = Point(pointTopLeftOld.at(i).x + widthOld.at(i) / 2, pointTopLeft
    Old.at(i).y + heightOld.at(i) / 2);
210.
      if ((everyMaxScale[i] != -1) && (everyMaxScale[i] < 5)) {
211.
      circle(frameTM, mitte, 4, CV_RGB(0, 0, 255), -1);
212.                                                                  }
213.
      if ((everyMaxScale[i] != -1) && (everyMaxScale[i] > 4)) {
214.
      circle(frameTM, mitte, 4, CV_RGB(255, 0, 0), -1);
215.                                                                  }
216.
```

```
217.                                                    }
218.                                            }
219.
220.                                            scalesOld = scales;
221.                                            pointTopLeftOld = pointTopLeft;
222.                                            widthOld = width;
223.                                            heightOld = height;
224.                                            pointTopLeft.clear();
225.                                            scales.clear();
226.                                            width.clear();
227.                                            height.clear();
228.                                            imageTemplates.clear();
229.                                            everyMaxScale.clear();
230.                                            cluster.clear();
231.                                    }
232.
233.                                    //Beginn Abschnitt Optischer Fluss
234.
      frame.copyTo(frameOF); //Aktuellen Frame in der Variable frameOF speichern
235.
236.
      if (firstRound) { //Im ersten Durchlauf des Programms Werte uebergeben
237.                                            firstRound = false;
238.                                            frame.copyTo(frameOld);
239.                                            p0 = featurePunkte;
240.                                    }
241.                                    else { //Ab dem zweiten Durchlauf des Programms
242.                                            if (p0.size() != 0) {
243.
244.
      calcOpticalFlowPyrLK(frameOld, frame, p0, p1, status, err, Size(21, 21), 3, cr
    iteria, OPTFLOW_LK_GET_MIN_EIGENVALS); //Positionen detektierter Merkmale des vo
    rherigen Frames im aktuellen Frame berechnen
245.
246.
      for (int i = 0; i < p0.size(); i++) { //Punkte, bei denen die Position bestimm
    t werden konnte, in good_new speichern
247.
      if (status[i] == 1) {
248.
      good_new.push_back(p1[i]);
249.
      distances.push_back(norm(p1[i] - p0[i])); //Distanz zwischen gleichen Merkmale
    n im vorherigen und aktuellen Frame berechnen
250.
      berechnungOF = true;
251.                                                                    }
252.                                                    }
253.                                                    if (berechnungOF) {
254.
255.                                                            double average = 0;
256.
      for (int j = 0; j < distances.size(); j++) { //Mittelwert aller Distanzwerte b
    ilden
257.
      average = average + distances[j];
258.
      }//Punkte mit Distanzen über Threshold werden rot, die anderen grün gezeichnet
```

```
259.
      average = average / distances.size();
260.
261.
      for (int j = 0; j < good_new.size(); j++) { //Berechnete Punkte mit einer Dist
    anz über dem Schwellwert werden rot und unter dem Schwellwert gruen in frameOF a
    bgebildet
262.
      if (distances[j] > (1.2 * average)) {
263.
      circle(frameOF, good_new[j], 2, CV_RGB(255, 0, 0), -1);
264.                                                                    }
265.                                                                    else {
266.
      circle(frameOF, good_new[j], 2, CV_RGB(0, 255, 0), -1);
267.                                                                    }
268.                                                                }
269.                                                                distances.clear();
270.                                                                good_new.clear();
271.
      berechnungOF = false;
272.                                                        }
273.                                                }
274.
275.
      frame.copyTo(frameOld); //Aktuellen Frame in frameOld speichern
276.
      p0 = featurePunkte; //Alle detetkierten Merkmale des aktuellen Frames in p0 sp
    eichern
277.                                }
278.
279.
      imshow("Eingangsbild", frame); //Gibt das Bild der Kamera wieder
280.
      imshow("Gefiltertes Eingangsbild", filteredFgMask); //Gibt Bild des gefilterte
    n Eingangsbildes aus
281.
      imshow("Gruppierte Merkmale", frameClustFP); //Gibt das Eingangsbild mit grupp
    ierten Merkmalen aus
282.
      imshow("Detektierte Merkmale", frameFP); //Gibt das Eingangsbild mit allen det
    ektierten Merkmalen aus
283.
      imshow("Ergebnis optischer Fluss", frameOF); //Gibt Ergebnis der Berechnung de
    s optischen Flusses aus
284.
      imshow("Ergebnis Template Matching", frameTM); //Gibt Ergebnis der Berechnung
    des Template Matchings aus
285.
286.
      int taste = waitKey(10); //Beenden der Schleife wenn "Esc" gedrueckt wird
287.                                if (taste == 27) {
288.                                        break;
289.                                }
290.
291.                                featurePunkte.clear();
292.                        }
293.            }
```

```
294.            catch (cv::Exception& e)
295.            {
296.                    cerr << e.msg << endl;
297.            }
298.            return 0;
299.    }
300.
```

Ansatz 2

```
1.  /*
2.  Shared Guide Dog - Hinderniserkennung / Erkennung von herabhängenden Objekten mi
    t einer monokularen Kamera
3.  Andrej Hofmann und Baljinder Singh
4.  15.08.2021
5.  Ansatz 2 nach Tomoyuki Mori und Sebastian Scherer: First Results in Detecting an
    d Avoiding Frontal Obstacles from a Monocular Camera for Micro Unmanned Aerial V
    ehicles
6.  Link: https://ieeexplore.ieee.org/document/6630807
7.  */
8.
9.  #include <iostream>
10. #include <windows.h>
11. #include <opencv2/opencv.hpp>
12. #include <opencv2/core.hpp>
13. #include "opencv2/highgui.hpp"
14. #include "opencv2/features2d.hpp"
15. #include "opencv2/xfeatures2d.hpp"
16.
17. using namespace cv;
18. using namespace std;
19. using namespace cv::xfeatures2d;
20.
21.
22. int main() {
23.   try {
24.             Mat frame, frameOld, outputFrame, matchFrame, frameFP;
25.
      bool firstRound = true; //Variable zur Ueberpruefung ob es sich um den ersten
    Durchlauf handelt
26.
27.             int minHessian = 800; //Threshold-
    Wert um zu entscheiden ob es sich um einen Feature-Punkt handelt
28.             Ptr<SURF> detector = SURF::create(minHessian); //SURF-
    Merkmalsdetektor initialisieren
29.
      Ptr<DescriptorMatcher> matcher = DescriptorMatcher::create(DescriptorMatcher::
    FLANNBASED); //Flann basierten Matcher initialisieren
30.
      vector<vector< DMatch >> matches; //Vektor um gefunde Matches zu speichern
31.
      vector<KeyPoint> keypoints, keypointsOld; //Vektoren um detektierte Merkmale z
    u speichern
32.
      Mat descriptors, descriptorsOld; //Variablen um Deskriptoren der detektierten
    Merkmale zu speichern
33.
34.
      VideoCapture vc(0); //Kamera oeffnen, falls dies fehlschlaegt Programm beenden
35.             if (!vc.open(0)) {
36.
      cout << "Kamera konnte nicht geoeffnet werden!" << std::endl;
37.                     return 0;
```

```
38.             }
39.             while (true) {
40.
41.                     vc >> frame; //Aktuelles Kamerabild in frame speichern
42.
      if (frame.empty()) { //Beenden der Schleife, wenn keine Daten der Kamera ankom
    men
43.                             break;
44.                     }
45.
      resize(frame, frame, Size(320, 240)); //Aktuelles Eingangbild auf 320x240 Pixe
    l skalieren
46.                     frame.copyTo(outputFrame);
47.
48.                     detector-
    >detectAndCompute(frame, noArray(), keypoints, descriptors); //Merkmale detektie
    ren und dazugehöhrige Deskriptoren generieren
49.
50.
      if (firstRound) { //Im ersten Durchlauf des Programms Werte uebergeben
51.                             keypointsOld = keypoints;
52.                             descriptorsOld = descriptors;
53.                             frameOld = frame;
54.                             firstRound = false;
55.                     }
56.                     else { //Ab dem zweiten Durchlauf des Programms
57.
      if ((!descriptors.empty()) && (!descriptorsOld.empty())) {
58.                                     matcher-
    >knnMatch(descriptors, descriptorsOld, matches, 2); //Detektierte Merkmale vom a
    ktuellen Frame mit Merkmalen aus vorherigen Frame vergleichen. Für jedes Merkmal
     werden die 2 besten Matches berechnet.
59.
60.
      vector<DMatch> matchesFiltered; //Vektor um gefilterte Matches zu speichern
61.
      for (int i = 0; i < matches.size(); ++i) //Matches filtern: den besten Match m
    it dem zweit besten Match vergleichen
62.                                     {
63.                                             const float ratio = 0.7f;
64.
      if (matches[i][0].distance < ratio * matches[i][1].distance)
65.                                             {
66.
      matchesFiltered.push_back(matches[i][0]);
67.                                             }
68.                                     }
69.
70.
      vector<DMatch> good_matches; //Vektor in dem nur die Matches gespeichert werde
    n, die groesser geworden sind
71.
      for (int i = 0; i < matchesFiltered.size(); i++) { //Matches filtern
72.
      if (keypoints[matchesFiltered[i].queryIdx].size > keypointsOld[matchesFiltered
    [i].trainIdx].size) {
73.
      good_matches.push_back(matchesFiltered[i]);
```

```
74.                                                 }
75.                                             }
76.
77.                                             if (!good_matches.empty()) {
78.
      vector<Point2f> obstacles; //Vektor um Position der Hindernisse zu speichern
79.
80.
      for (int i = 0; i < good_matches.size(); i++) { //Bildausschnitt vom vorherige
    n Frame um gefiltertes Merkmal generieren
81.
      Point2f center = keypointsOld[good_matches[i].trainIdx].pt; //Variable um Posi
    tion des gefilterten Merkmals des vorherigen Bildes zu speichern
82.
      double diameter = keypointsOld[good_matches[i].trainIdx].size; //Variable um D
    urchmesser des gefilterten Merkmals des vorherigen Bildes zu speichern
83.
      Rect a; //Variable um die Fläche des gefilterten Merkmals des vorherigen Bilde
    s zu speichern
84.
      a.x = center.x - diameter * 0.5; a.y = center.y - diameter * 0.5; a.width = di
    ameter; a.height = diameter;
85.
      Mat template1 = frameOld(a); //Variable um Bildausschnitt des vorherigen Bilde
    s zu speichern
86.
87.
      double maxValOld = 0; //Variable um die Skalierung mit der hoechsten Uebereins
    timmung zu bestimmen
88.
      double maxScale = 0; //Variable um die Skalierung mit der hoechsten Uebereinst
    immung zu speichern
89.
      Mat result; double minVal; double maxVal; Point minLoc; Point maxLoc; //Variab
    len für die Berechnung des Template Matching
90.
91.
      double widthNew = a.width; //Variable um Breite der skalierten Bildausschnitte
     zu berechnen
92.
      double heightNew = a.height; //Variable um Hoehe der skalierten Bildausschnitt
    e zu berechnen
93.
      for (int j = 1; j < 6; j++) {
94.
      resize(template1, template1, Size(widthNew, heightNew)); //Bildausschnitt des
    vorherigen Frames skalieren
95.
      widthNew = widthNew + 1; //Breite wird um einen Pixel erhoeht
96.
      heightNew = heightNew + 1; //Hoehe wird um einen Pixel erhoeht
97.
      Point2f center2 = keypoints[good_matches[i].queryIdx].pt; //Variable um Positi
    on des gefilterten Merkmals des aktuellen Bildes zu speichern
98.
      double diameter2 = keypoints[good_matches[i].queryIdx].size; //Variable um Dur
    chmesser des gefilterten Merkmals des aktuellen Bildes zu speichern
```

```
99.
      Rect b; //Variable um die Fläche des gefilterten Merkmals des aktuellen Bildes
     zu speichern
100.
      b.x = center2.x - diameter2 * 0.5; b.y = center2.y - diameter2 * 0.5; b.width
    = diameter2; b.height = diameter2;
101.
      Mat template2 = frame(b); //Variable um Bildausschnitt des aktuellen Bildes zu
     speichern
102.
103.
      if ((template1.size().height < template2.size().height) && (template1.size().w
    idth < template2.size().width)) { //Bildausschnitte miteinander vergleichen
104.
      matchTemplate(template2, template1, result, TM_CCORR_NORMED);
105.
      minMaxLoc(result, &minVal, &maxVal, &minLoc, &maxLoc);
106.
      if (maxVal >= maxValOld) {
107.
                maxValOld = maxVal;
108.
                maxScale = j;
109.
      }
110.
      }
111.                                                                          }
112.
      if (maxScale > 3) { //Positionen der Merkmale speichern, die sich um min. 2 Pi
    xel vergroessert haben
113.
      obstacles.push_back(keypoints[good_matches[i].queryIdx].pt);
114.                                                                          }
115.                                                                  }
116.
117.
      if (!obstacles.empty()) { //Positionen der Hindernisse werden in outputFrame r
    ot gekennzeichnet
118.
      for (int k = 0; k < obstacles.size(); k++) {
119.
      circle(outputFrame, obstacles[k], 5, CV_RGB(255, 0, 0), -1);
120.                                                                          }
121.                                                                  }
122.
      drawMatches(frame, keypoints, frameOld, keypointsOld, good_matches, matchFrame
    , Scalar::all(-1), Scalar::all(-
    1), std::vector<char>(), DrawMatchesFlags::NOT_DRAW_SINGLE_POINTS); //Matches de
    s vorherigen Frames zum aktuellen Frame werden dargestellt
123.                                                          }
124.
125.                                                          matches.clear();
126.                                                          matchesFiltered.clear();
127.                                                          good_matches.clear();
128.                                                  }
129.
130.                                                  keypointsOld = keypoints;
```

```
131.                                        descriptorsOld = descriptors;
132.                                        frameOld = frame;
133.                                }
134.
135.      imshow("Aktuelles Bild mit markierten Hindernissen", outputFrame); //Gibt Bild
     mit detektierten Hindernissen aus
136.      drawKeypoints(frame, keypoints, frameFP, CV_RGB(255, 0, 0));
137.      imshow("Aktuelles Bild mit detektierten Merkmalen", frameFP); //Gibt Bild mit
    detektierten Merkmalen aus
138.                                if (!matchFrame.empty()) {
139.      imshow("Wiedergefundene Merkmale", matchFrame); //Gibt Bild mit wiedergefunden
    en Merkmalen aus
140.                                }
141.
142.                                keypoints.clear();
143.                                descriptors.release();
144.
145.      int taste = waitKey(10); //Beenden der Schleife wenn "Esc" gedrueckt wurde
146.                                if (taste == 27) {
147.                                        break;
148.                                }
149.                        }
150.            }
151.            catch (cv::Exception& e)
152.            {
153.                        cerr << e.msg << endl;
154.            }
155.            return 0;
156.  }
157.
```

Ansatz 3 Variante 1

```
1.  /*
2.  Shared Guide Dog - Hinderniserkennung / Erkennung von herabhängenden Objekten mi
    t einer monokularen Kamera
3.  Andrej Hofmann und Baljinder Singh
4.  15.08.2021
5.  Ansatz 3 Version 1 mit Hilfe der Gleichung aus: Notes on Motion Estimation
6.  Link: https://www.cns.nyu.edu/~david/handouts/motion.pdf
7.  */
8.  #include <iostream>
9.  #include <windows.h>
10. #include <opencv2/opencv.hpp>
11. #include <opencv2/core.hpp>
12. #include "opencv2/features2d.hpp"
13. #include "opencv2/xfeatures2d.hpp"
14. #include <chrono>
15.
16. using namespace cv;
17. using namespace std;
18. using namespace cv::xfeatures2d;
19. using namespace std::chrono;
20.
21. struct distanceEuc //struct um Punkte in der Methode "partition" zu gruppieren.
    Aus: https://www.py4u.net/discuss/111223
22. {
23.   int _dist2;
24.   distanceEuc(int dist) : _dist2(dist + 1) {}
25.
26.   bool operator()(const Point& lhs, const Point& rhs) const
27.   {
28.
      return ((lhs.x - rhs.x) * (lhs.x - rhs.x) + (lhs.y - rhs.y) * (lhs.y - rhs.y))
     < _dist2;
29.   }
30. };
31.
32. int main() {
33.   try {
34.             Mat frame, frameOld, frameH, frameOF;
35.
      bool firstRound = true; //Variable zur Ueberpruefung ob es sich um den ersten
    Durchlauf handelt
36.
37.
      high_resolution_clock::time_point t1; //Variable um Zeit beim Aufnehmen des ak
    tuellen Bildes zu speichern
38.
      high_resolution_clock::time_point tOld; //Variable um Zeit beim Aufnehmen des
    vorherigen Bildes zu speichern
39.
40.
      bool region1 = false; //Variablen zur Ueberpruefung, ob sich Hindernisse in de
    n einzelnen Bereichen befinden
41.             bool region2 = false;
```

```
42.             bool region3 = false;
43.
44.
    VideoCapture vc(0); //Kamera oeffnen, falls dies fehlschlaegt Programm beenden
45.             if (!vc.open(0)) {
46.
    cout << "Kamera konnte nicht geoeffnet werden!" << std::endl;
47.                         return 0;
48.             }
49.
50.             Mat flow; //Variable um berechneten optischen Fluss zu speichern
51.             while (true) {
52.                         vc >> frame; //Aktuelles Kamerabild in frame speichern
53.                         vc >> frameOF; //Aktuelles Kamerabild in frameOF speichern
54.
    if (frame.empty()) { //Beenden der Schleife, wenn keine Daten der Kamera empfa
    ngen werden
55.                                     break;
56.                         }
57.
58.                         t1 = high_resolution_clock::now();
59.
60.
    cvtColor(frame, frame, COLOR_BGR2GRAY); //Aktuelles Kamerabild in Graubild kon
    vertieren
61.
    resize(frame, frame, Size(320, 240)); //frame auf 320x240 Pixel skalieren
62.
    resize(frameOF, frameOF, Size(frame.size().width, frame.size().height)); //fra
    meOF auf 320x240 Pixel skalieren
63.
64.
    frameOF.copyTo(frameH); //Aktuellen Frame in der Variable frameH speichern
65.
66.
    double fLength = 230.75753 * (frame.size().width / 320); //Variable fuer die B
    rennweite der verwendeten Kamera in Pixeln
67.
68.
    if (firstRound) { //Im ersten Durchlauf des Programms Werte uebergeben
69.                                     firstRound = false;
70.                                     frame.copyTo(frameOld);
71.                                     tOld = t1;
72.                         }
73.                         else { //Ab dem zweiten Durchlauf des Programms
74.
75.
    calcOpticalFlowFarneback(frameOld, frame, flow, 0.5, 3, 15, 3, 5, 1.2, 0); //B
    erechnung des optischen Flusses
76.
77.
    duration<double, std::milli> time_span = t1 - tOld; //Zeitspanne zwischen vorh
    erigem und jetzigem frame berechnen
78.
79.
    vector<Point2f> obstacles; //Vektor um die Positionen der Merkmale mit Tiefenw
    erten unter 1 m zu speichern
80.
```

```
81.
      if (time_span.count() != 0) { //Berechnung der Tiefenwerte für jeden 5. Pixel
    des Eingangsbildes
82.
      for (int y = 0; y < frame.size().height; y += 5) {
83.
      for (int x = 0; x < frame.size().width; x += 5) {
84.
85.
      Mat translation(3, 1, CV_64F); //Translationsvektor t
86.
      translation.at<double>(0, 0) = 0; //Negative Kamera-Geschwindigkeit in X-
    Richtung in cm/s
87.
      translation.at<double>(1, 0) = 0; //Negative Kamera-Geschwindigkeit in Y-
    Richtung in cm/s
88.
      translation.at<double>(2, 0) = -100; //Negative Kamera-Geschwindigkeit in Z-
    Richtung in cm/s
89.
90.
      Mat rotation(3, 1, CV_64F); //Rotationsvektor omega
91.
      rotation.at<double>(0, 0) = 0; //Negative Kamera-Rotation um die X-
    Achse in 1/s
92.
      rotation.at<double>(1, 0) = 0; //Negative Kamera-Rotation um die Y-
    Achse in 1/s
93.
      rotation.at<double>(2, 0) = 0; //Negative Kamera-Rotation um die Z-
    Achse in 1/s
94.
95.
      //Matrix A aufstellen
96.
      Mat a(2, 3, CV_64F); a.at<double>(0, 0) = -
    fLength; a.at<double>(0, 1) = 0; a.at<double>(0, 2) = -
    x + (frame.size().width / 2) - 1;
97.
      a.at<double>(1, 0) = 0; a.at<double>(1, 1) = -fLength; a.at<double>(1, 2) = -
    y + (frame.size().height / 2) - 1;
98.
99.
      //Matrix B aufstellen
100.
      Mat b(2, 3, CV_64F); b.at<double>(0, 0) = ((-
    x + (frame.size().width / 2) - 1) * (-
    y + (frame.size().height / 2) - 1)) / fLength; b.at<double>(0, 1) = -
    (fLength + (-x + (frame.size().width / 2) - 1) * (-
    x + (frame.size().width / 2) - 1) / fLength); b.at<double>(0, 2) = -
    y + (frame.size().height / 2) - 1;
101.
      b.at<double>(1, 0) = fLength + (-y + (frame.size().height / 2) - 1) * (-
    y + (frame.size().height / 2) - 1) / fLength; b.at<double>(1, 1) = -((-
    x + (frame.size().width / 2) - 1) * (-
    y + (frame.size().height / 2) - 1) / fLength); b.at<double>(1, 2) = -(-
    x + (frame.size().width / 2) - 1);
102.
```

```
103.
      //Matrix A * Vektor t = at
104.
      Mat at = a * translation;
105.
106.
      //Matrix B * Vektor omega = B omega
107.
      Mat bOmega = b * rotation;
108.
109.
      //Vektor um die Pixelgeschwindigkeiten u und v zu berechnen
110.
      Mat uv(2, 1, CV_64F); uv.at<double>(0, 0) = flow.at<Point2f>(y, x).x / (time_s
    pan.count() * 0.001); uv.at<double>(1, 0) = flow.at<Point2f>(y, x).y / (time_spa
    n.count() * 0.001);
111.
112.
      Mat w = uv - bOmega;
113.                                                                    Mat wt;
114.
      transpose(w, wt);
115.
      Mat temp1 = wt * w;
116.
      double wtw = temp1.at<double>(0, 0);
117.
      Mat temp2 = wt * at;
118.
      double wtAt = temp2.at<double>(0.0);
119.
      double z = wtAt / wtw; //Berechneter Tiefenwert des betrachteten Pixels in cm
120.
121.
      //Pixel abhaengig des Tiefenwerts farblich in frameOF markieren
122.
      if ((0 <= z) && (z < 100)) {
123.
      circle(frameOF, Point2f(x, y), 2, CV_RGB(255, 0, 0), -1);
124.
      obstacles.push_back(Point2f(x, y)); //Punkte die maximal 1 m von der Kamera en
    tfernt sind in obstacles speichern
125.                                                                    }
126.
      else if ((100 <= z) && (z < 200)) {
127.
      circle(frameOF, Point2f(x, y), 2, CV_RGB(255, 128, 0), -1);
128.                                                                    }
129.
      else if ((200 <= z) && (z < 300)) {
130.
      circle(frameOF, Point2f(x, y), 2, CV_RGB(255, 255, 0), -1);
131.                                                                    }
132.
      else if ((300 <= z) && (z < 400)) {
133.
      circle(frameOF, Point2f(x, y), 2, CV_RGB(128, 255, 0), -1);
134.                                                                    }
```

```
135.
      else if ((400 <= z) && (z < 500)) {
136.
      circle(frameOF, Point2f(x, y), 2, CV_RGB(0, 255, 0), -1);
137.                                                                    }
138.
      else if ((500 <= z) && (z < 600)) {
139.
      circle(frameOF, Point2f(x, y), 2, CV_RGB(0, 255, 128), -1);
140.                                                                    }
141.
      else if ((600 <= z) && (z < 700)) {
142.
      circle(frameOF, Point2f(x, y), 2, CV_RGB(0, 255, 255), -1);
143.                                                                    }
144.
      else if ((700 <= z) && (z < 800)) {
145.
      circle(frameOF, Point2f(x, y), 2, CV_RGB(0, 128, 255), -1);
146.                                                                    }
147.
      else if (800 <= z) {
148.
      circle(frameOF, Point2f(x, y), 2, CV_RGB(0, 0, 255), -1);
149.                                                                    }
150.
      else if (0 > z) {
151.
      circle(frameOF, Point2f(x, y), 2, CV_RGB(0, 0, 255), -1);
152.                                                                    }
153.                                                                    else {
154.
      circle(frameOF, Point2f(x, y), 2, CV_RGB(0, 0, 255), -1);
155.                                                                    }
156.                                                              }
157.                                                    }
158.                                          }
159.
160.                                          if (!obstacles.empty()) {
161.
      vector<int> labels; //Vektor um die Gruppen einzelner Hindernisse zu speichern
162.
      int cluster_count = cv::partition(obstacles, labels, distanceEuc(25)); //Hinde
    rnispunkte gruppieren
163.
      vector<vector<Point>> cluster(cluster_count); //Vektor um gruppierte Hindernis
    punkte zu speichern
164.
      vector <vector<Point>> hulls; //Vektor um Linien um Gruppen von Hindernissen z
    u speichern
165.
166.
      for (int i = 0; i < obstacles.size(); i++) {
167.
      cluster[labels[i]].push_back(obstacles[i]);
168.                                                    }
169.
```

```
170.
      for (int i = 0; i < cluster.size(); i++) { //Gruppierungen von Hindernissen na
    ch Anzahl filtern
171.
      if (cluster[i].size() > 9) {
172.
      vector<Point> hull;
173.
      convexHull(cluster[i], hull, false, true);
174.
      hulls.push_back(hull);
175.
176.
      //Berechnung der Richtungsanweisungen
177.
      for (int j = 0; j < cluster[i].size(); j++) {
178.
      if (cluster[i].at(j).x < (frame.size().width / 3)) {
179.
      region1 = true;
180.
      }
181.
      else if (cluster[i].at(j).x < ((frame.size().width / 3) * 2)) {
182.
      region2 = true;
183.
      }
184.
      else if (cluster[i].at(j).x < frame.size().width) {
185.
      region3 = true;
186.
      }
187.                                                                          }
188.                                                                  }
189.                                                          }
190.
      //Richtungsanweisungen im frameH darstellen
191.
      if (region1 && region2 && region3) {
192.                                                                  int baseline = 0;
193.
      Size textSize = getTextSize("STOP", FONT_HERSHEY_SIMPLEX, 1, 4, &baseline);
194.
      putText(frameH, "STOP", Point2f((frame.size().width / 2) - (textSize.width / 2
    ) - 1, (frame.size().height / 4) - 1), FONT_HERSHEY_SIMPLEX, 1, CV_RGB(255, 0, 0
    ), 4);
195.                                                          }
196.
      else if (region1 && region2 && !region3) {
197.
      arrowedLine(frameH, Point2f((frame.size().width / 2) - 1, (frame.size().height
     / 4) - 1), Point2f((frame.size().width / 2) + 79, (frame.size().height / 4) - 1
    ), CV_RGB(0, 255, 0), 3, 0, 0.9);
198.                                                          }
199.
      else if (region2 && region3 && !region1) {
```

```
200.
      arrowedLine(frameH, Point2f((frame.size().width / 2) - 1, (frame.size().height
     / 4) - 1), Point2f((frame.size().width / 2) - 81, (frame.size().height / 4) - 1
    ), CV_RGB(0, 255, 0), 3, 0, 0.9);
201.                                                    }
202.
      else if (!region1 && region2 && !region3) {
203.
      arrowedLine(frameH, Point2f((frame.size().width / 2) - 1, (frame.size().height
     / 4) - 1), Point2f((frame.size().width / 2) - 81, (frame.size().height / 4) - 1
    ), CV_RGB(0, 255, 0), 3, 0, 0.9);
204.
      arrowedLine(frameH, Point2f((frame.size().width / 2) - 1, (frame.size().height
     / 4) - 1), Point2f((frame.size().width / 2) + 79, (frame.size().height / 4) - 1
    ), CV_RGB(0, 255, 0), 3, 0, 0.9);
205.                                                    }
206.                                                    region1 = false;
207.                                                    region2 = false;
208.                                                    region3 = false;
209.
210.                                                    if (!hulls.empty()) {
211.
      drawContours(frameH, hulls, -
    1, CV_RGB(255, 0, 0)); //Linien um gruppierte Hindernisse in frameH darstellen
212.                                                    }
213.                                            }
214.
215.
      frame.copyTo(frameOld); //Aktuellen Frame in frameOld speichern
216.                                            tOld = t1;
217.                                    }
218.
219.
      imshow("Konvertiertes Eingangsbild", frame); //Gibt das Graubild der Kamera wi
    eder
220.
      imshow("Ergebnis optischer Fluss", frameOF);  //Gibt Bild mit Ergebnis der Ber
    echnung des optischen Flusses aus
221.
      imshow("Hindernisse", frameH); //Gibt das Eingangsbild mit Richtungsanweisung
    und markierten Hindernissen aus
222.
223.
      int taste = waitKey(10); //Beenden der Schleife wenn "Esc" gedrueckt wurde
224.                                    if (taste == 27) {
225.                                            break;
226.                                    }
227.                            }
228.            }
229.            catch (cv::Exception& e)
230.            {
231.                            cerr << e.msg << endl;
232.            }
233.            return 0;
234.    }
235.
```

Ansatz 3 Variante 2

```
1.  /*
2.  Shared Guide Dog - Hinderniserkennung / Erkennung von herabhängenden Objekten mi
    t einer monokularen Kamera
3.  Andrej Hofmann und Baljinder Singh
4.  15.08.2021
5.  Ansatz 3 Version 2 mit Hilfe der Gleichung aus: Notes on Motion Estimation
6.  Link: https://www.cns.nyu.edu/~david/handouts/motion.pdf
7.  */
8.  #include <iostream>
9.  #include <windows.h>
10. #include <opencv2/opencv.hpp>
11. #include <opencv2/core.hpp>
12. #include "opencv2/features2d.hpp"
13. #include "opencv2/xfeatures2d.hpp"
14. #include <chrono>
15.
16. using namespace cv;
17. using namespace std;
18. using namespace cv::xfeatures2d;
19. using namespace std::chrono;
20.
21. struct distanceEuc //struct um Punkte in der Methode "partition" zu gruppieren.
    Aus: https://www.py4u.net/discuss/111223
22. {
23.   int _dist2;
24.   distanceEuc(int dist) : _dist2(dist + 1) {}
25.
26.   bool operator()(const Point& lhs, const Point& rhs) const
27.   {
28.
      return sqrt((lhs.x - rhs.x) * (lhs.x - rhs.x) + (lhs.y - rhs.y) * (lhs.y - rhs
    .y)) < _dist2;
29.   }
30. };
31.
32. int main() {
33.   try {
34.             Mat frame, frameFP, frameOF, frameOld, frameH;
35.
      bool firstRound = true; //Variable zur Ueberpruefung ob es sich um den ersten
    Durchlauf handelt
36.
37.             int minHessian = 800; //Threshold-
    Wert um zu entscheiden ob es sich um einen Feature-Punkt handelt
38.             Ptr<SURF> detector = SURF::create(minHessian); //SURF-
    Merkmalsdetektor initialisieren
39.
      vector<KeyPoint> keypoints; //Vektor um detektierte Merkmale zu speichern
40.
      vector<Point2f> p0, p1; //In Vektor p0 werden die detektierten Merkmale des vo
    rherigen Frames und in Vektor p1 werden die durch Optischen Fluss berechneten Pu
    nkte im aktuellen Frame gespeichert
```

```
41.
      vector<Point2f> good_new; //Vektor um Punkte die durch optical flow berechnet
    wurden zu speichern
42.
      vector<uchar> status; //Vektor um Berechnung einzelner Merkmale beim optischen
     Fluss zu ueberpruefen
43.
      vector<float> err;  //Vektor um Fehler bei der Berechnung des optischen Flusse
    s zu speichern
44.
      TermCriteria criteria = TermCriteria((TermCriteria::COUNT)+(TermCriteria::EPS)
    , 30, 0.01); //Austrittsbedingung bei der Berechnung des optischen Flusses
45.
      vector<double> depthValues; //Vektor um berechnete Tiefenwerte zu speichern
46.
      high_resolution_clock::time_point t1; //Variable um Zeit beim Aufnehmen des ak
    tuellen Bildes zu speichern
47.
      high_resolution_clock::time_point tOld; //Variable um Zeit beim Aufnehmen des
    vorherigen Bildes zu speichern
48.
49.
      VideoCapture vc(0); //Kamera oeffnen, falls dies fehlschlaegt Programm beenden
50.           if (!vc.open(0)) {
51.
      cout << "Kamera konnte nicht geoeffnet werden!" << std::endl;
52.                     return 0;
53.           }
54.
55.           while (true) {
56.                     vc >> frame; //Aktuelles Kamerabild in frame speichern
57.
      if (frame.empty()) { //Beenden der Schleife, wenn keine Daten der Kamera empfa
    ngen werden
58.                               break;
59.                     }
60.
61.                     t1 = high_resolution_clock::now();
62.
63.
      resize(frame, frame, Size(320, 240)); //frame auf 320x240 Pixel skalieren
64.
      frame.copyTo(frameOF); //Aktuellen Frame in der Variable frameOF speichern
65.
      frame.copyTo(frameH); //Aktuellen Frame in der Variable frameH speichern
66.
      double fLength = 230.75753 * (frame.size().width / 320); //Variable fuer die B
    rennweite der verwendeten Kamera in Pixeln
67.
68.                     detector-
    >detect(frame, keypoints, noArray()); //Merkmale im Eingangsbild detektieren
69.
70.
      if (firstRound) { //Im ersten Durchlauf des Programms Werte uebergeben
71.                               firstRound = false;
72.                               frame.copyTo(frameOld);
73.                               tOld = t1;
```

```
74.
      for (int i = 0; i < keypoints.size(); i++) { //Detektierte Merkmale in p0 spei
    chern
75.                                                 p0.push_back(keypoints[i].pt);
76.                                 }
77.                         }
78.
79.                         else { //Ab dem zweiten Durchlauf des Programms
80.
      vector<Point2f> obstacles; //Vektor um die Positionen der Merkmale mit Tiefenw
    erten unter 1 m zu speichern
81.
82.                                 if (p0.size() != 0) {
83.
      calcOpticalFlowPyrLK(frameOld, frame, p0, p1, status, err, Size(21, 21), 4, cr
    iteria); //Positionen detektierter Merkmale des vorherigen Frames im aktuellen F
    rame berechnen
84.
      duration<double, std::milli> time_span = t1 - tOld; //Zeitspanne zwischen vorh
    erigem und jetzigem frame berechnen
85.
86.
      for (int i = 0; i < p0.size(); i++) { //Berechnung der Tiefenwerte für jedes d
    etektierte Merkmal
87.
      if (status[i] == 1) { //Punkte, bei denen die Position bestimmt werden konnte,
     in good_new speichern
88.
      good_new.push_back(p1[i]);
89.
90.
      if (time_span.count() != 0) {
91.
      Mat translation(3, 1, CV_64F); //Translationsvektor t
92.
      translation.at<double>(0, 0) = 0; //Negative Kamera-Geschwindigkeit in X-
    Richtung in cm/s
93.
      translation.at<double>(1, 0) = 0; //Negative Kamera-Geschwindigkeit in Y-
    Richtung in cm/s
94.
      translation.at<double>(2, 0) = -100; //Negative Kamera-Geschwindigkeit in Z-
    Richtung in cm/s
95.
96.
      Mat rotation(3, 1, CV_64F); //Rotationsvektor omega
97.
      rotation.at<double>(0, 0) = 0; //Negative Kamera-Rotation um die X-
    Achse in 1/s
98.
      rotation.at<double>(1, 0) = 0; //Negative Kamera-Rotation um die Y-
    Achse in 1/s
99.
      rotation.at<double>(2, 0) = 0; //Negative Kamera-Rotation um die Z-
    Achse in 1/s
100.
101.
      //Matrix A aufstellen
```

```
102.
      Mat a(2, 3, CV_64F); a.at<double>(0, 0) = -
    fLength; a.at<double>(0, 1) = 0; a.at<double>(0, 2) = -
    p1[i].x + (frame.size().width / 2) - 1;
103.
      a.at<double>(1, 0) = 0; a.at<double>(1, 1) = -fLength; a.at<double>(1, 2) = -
    p1[i].y + (frame.size().height / 2) - 1;
104.
105.
      //Matrix B aufstellen
106.
      Mat b(2, 3, CV_64F); b.at<double>(0, 0) = ((-
    p1[i].x + (frame.size().width / 2) - 1) * (-
    p1[i].y + (frame.size().height / 2) - 1)) / fLength; b.at<double>(0, 1) = -
    (fLength + (-p1[i].x + (frame.size().width / 2) - 1) * (-
    p1[i].x + (frame.size().width / 2) - 1) / fLength); b.at<double>(0, 2) = -
    p1[i].y + (frame.size().height / 2) - 1;
107.
      b.at<double>(1, 0) = fLength + (-p1[i].y + (frame.size().height / 2) - 1) * (-
    p1[i].y + (frame.size().height / 2) - 1) / fLength; b.at<double>(1, 1) = -((-
    p1[i].x + (frame.size().width / 2) - 1) * (-
    p1[i].y + (frame.size().height / 2) - 1) / fLength); b.at<double>(1, 2) = -(-
    p1[i].x + (frame.size().width / 2) - 1);
108.
109.
      //Matrix A * Vektor t = at
110.
      Mat at = a * translation;
111.
112.
      //Matrix B * Vektor omega = B omega
113.
      Mat bOmega = b * rotation;
114.
115.
      //Vektor um die Pixelgeschwindigkeiten u und v zu berechnen
116.
      Mat uv(2, 1, CV_64F); uv.at<double>(0, 0) = ((p1[i].x - p0[i].x) / (time_span.
    count() * 0.001)); uv.at<double>(1, 0) = ((p1[i].y - p0[i].y) / (time_span.count
    () * 0.001));
117.
118.
      Mat w = uv - bOmega;
119.
      Mat wt;
120.
      transpose(w, wt);
121.
      Mat zwischen = wt * w;
122.
      double wtw = zwischen.at<double>(0, 0);
123.
      Mat zwischen2 = wt * at;
124.
      double wtAt = zwischen2.at<double>(0.0);
125.
      double z = wtAt / wtw; //Berechneter Tiefenwert des betrachteten Pixels in cm
```

```
126.
      depthValues.push_back(z);
127.                                                                                  }
128.                                                                          }
129.                                                                  }
130.                                                          }
131.
      if (!good_new.empty()) { //Merkmale abhaengig des Tiefenwerts farblich in fram
    eOF markieren
132.
      for (int y = 0; y < good_new.size(); y++) {
133.
134.
      if ((0 <= depthValues[y]) && (depthValues[y] < 100)) {
135.
      circle(frameOF, good_new[y], 2, CV_RGB(255, 0, 0), -1);
136.
      obstacles.push_back(good_new[y]); //Punkte die maximal 1 m von der Kamera entf
    ernt sind in obstacles speichern
137.                                                                          }
138.
      else if ((100 <= depthValues[y]) && (depthValues[y] < 200)) {
139.
      circle(frameOF, good_new[y], 2, CV_RGB(255, 128, 0), -1);
140.                                                                          }
141.
      else if ((200 <= depthValues[y]) && (depthValues[y] < 300)) {
142.
      circle(frameOF, good_new[y], 2, CV_RGB(255, 255, 0), -1);
143.                                                                          }
144.
      else if ((300 <= depthValues[y]) && (depthValues[y] < 400)) {
145.
      circle(frameOF, good_new[y], 2, CV_RGB(128, 255, 0), -1);
146.                                                                          }
147.
      else if ((400 <= depthValues[y]) && (depthValues[y] < 500)) {
148.
      circle(frameOF, good_new[y], 2, CV_RGB(0, 255, 0), -1);
149.                                                                          }
150.
      else if ((500 <= depthValues[y]) && (depthValues[y] < 600)) {
151.
      circle(frameOF, good_new[y], 2, CV_RGB(0, 255, 128), -1);
152.                                                                          }
153.
      else if ((600 <= depthValues[y]) && (depthValues[y] < 700)) {
154.
      circle(frameOF, good_new[y], 2, CV_RGB(0, 255, 255), -1);
155.                                                                          }
156.
      else if ((700 <= depthValues[y]) && (depthValues[y] < 800)) {
157.
      circle(frameOF, good_new[y], 2, CV_RGB(0, 128, 255), -1);
158.                                                                          }
159.
      else if (800 <= depthValues[y]) {
```

```
160.
      circle(frameOF, good_new[y], 2, CV_RGB(0, 0, 255), -1);
161.                                                                 }
162.
      else if (0 > depthValues[y]) {
163.
      circle(frameOF, good_new[y], 2, CV_RGB(0, 0, 255), -1);
164.                                                                 }
165.                                                                 else {
166.
      circle(frameOF, good_new[y], 2, CV_RGB(0, 0, 255), -1);
167.                                                                 }
168.                                                         }
169.                                                 }
170.
171.                                                 if (!obstacles.empty()) {
172.
      vector<int> labels; //Vektor um die Gruppen einzelner Hindernisse zu speichern
173.
      int cluster_count = cv::partition(obstacles, labels, distanceEuc(20)); //Hinde
    rnispunkte gruppieren
174.
      vector<vector<Point>> cluster(cluster_count); //Vektor um gruppierte Hindernis
    punkte zu speichern
175.
      vector <vector<Point>> hulls; //Vektor um Linien um Gruppen von Hindernissen z
    u speichern
176.
177.
      for (int i = 0; i < obstacles.size(); i++) {
178.
      cluster[labels[i]].push_back(obstacles[i]);
179.                                                         }
180.
181.
      for (int i = 0; i < cluster.size(); i++) { //Gruppierungen von Hindernissen na
    ch Anzahl filtern
182.
      if (cluster[i].size() > 3) {
183.
      vector<Point> hull;
184.
      convexHull(cluster[i], hull, false, true);
185.
      hulls.push_back(hull);
186.                                                                 }
187.                                                         }
188.                                                         if (!hulls.empty()) {
189.
      drawContours(frameH, hulls, -
    1, CV_RGB(255, 0, 0)); //Linien um gruppierte Hindernisse in frameH darstellen
190.                                                         }
191.                                                 }
192.
193.
      frame.copyTo(frameOld); //Aktuellen Frame in frameOld speichern
194.                                                 p0.clear();
195.
```

```
196.
      for (int i = 0; i < keypoints.size(); i++) { //Detektierte Merkmale in p0 spei
    chern
197.
      p0.push_back(keypoints[i].pt);
198.                                                }
199.                                                tOld = t1;
200.                                    }
201.
202.
      drawKeypoints(frame, keypoints, frameFP, CV_RGB(255, 0, 0));
203.
      imshow("Aktuelles Bild mit detektierten Merkmalen", frameFP); //Gibt Bild mit
    detektierten Merkmalen aus
204.
      imshow("Ergebnis optischer Fluss", frameOF); //Gibt Bild mit dem Ergebnis der
    Berechnung des optischen Flusses aus
205.
      imshow("Hindernisse", frameH); //Gibt das Eingangsbild mit markierten Hinderni
    ssen aus
206.
207.
      int taste = waitKey(10); //Beenden der Schleife wenn "Esc" gedrueckt wurde
208.                                    if (taste == 27) {
209.                                                break;
210.                                    }
211.
212.                                    depthValues.clear();
213.                                    good_new.clear();
214.                          }
215.            }
216.            catch (cv::Exception& e)
217.            {
218.                          cerr << e.msg << endl;
219.            }
220.            return 0;
221.  }
222.
```

Ansatz 3 Variante 3

```
1.  /*
2.  Shared Guide Dog - Hinderniserkennung / Erkennung von herabhängenden Objekten mi
    t einer monokularen Kamera
3.  Andrej Hofmann und Baljinder Singh
4.  15.08.2021
5.  Ansatz 3 Version 3 mit Hilfe der Gleichung aus: Notes on Motion Estimation
6.  Link: https://www.cns.nyu.edu/~david/handouts/motion.pdf
7.  */
8.  #include <iostream>
9.  #include <windows.h>
10. #include <opencv2/opencv.hpp>
11. #include <opencv2/core.hpp>
12. #include "opencv2/features2d.hpp"
13. #include "opencv2/xfeatures2d.hpp"
14. #include <chrono>
15.
16. using namespace cv;
17. using namespace std;
18. using namespace cv::xfeatures2d;
19. using namespace std::chrono;
20.
21. struct distanceEuc //struct um Punkte in der Methode "partition" zu gruppieren.
    Aus: https://www.py4u.net/discuss/111223
22. {
23.   int _dist2;
24.   distanceEuc(int dist) : _dist2(dist + 1) {}
25.
26.   bool operator()(const Point& lhs, const Point& rhs) const
27.   {
28.
      return sqrt((lhs.x - rhs.x) * (lhs.x - rhs.x) + (lhs.y - rhs.y) * (lhs.y - rhs
    .y)) < _dist2;
29.   }
30. };
31.
32. int main() {
33.   try {
34.             Mat frame, frameFP, frameOF, frameOld, frameH;
35.
      bool firstRound = true; //Variable zur Ueberpruefung ob es sich um den ersten
    Durchlauf handelt
36.
37.             int minHessian = 800; //Threshold-
    Wert um zu entscheiden ob es sich um einen Feature-Punkt handelt
38.             Ptr<SURF> detector = SURF::create(minHessian); //SURF-
    Merkmalsdetektor initialisieren
39.
      vector<KeyPoint> keypoints; //Vektor um detektierte Merkmale zu speichern
40.
      vector<Point2f> p0, p1; //In Vektor p0 werden die detektierten Merkmale eines
    vorherigen Frames und in Vektor p1 werden die durch Optischen Fluss berechneten
    Punkte im aktuellen Frame gespeichert
```

```
41.
      vector<Point2f> good_new; //Vektor um Punkte die durch optical flow berechnet
    wurden zu speichern
42.
      vector<uchar> status; //Vektor um Berechnung einzelner Merkmale beim optischen
     Fluss zu ueberpruefen
43.
      vector<float> err; //Vektor um Fehler bei der Berechnung des optischen Flusses
     zu speichern
44.
      TermCriteria criteria = TermCriteria((TermCriteria::COUNT)+(TermCriteria::EPS)
    , 30, 0.01); //Austrittsbedingung bei der Berechnung des optischen Flusses
45.
      vector<double> depthValues; //Vektor um berechnete Tiefenwerte zu speichern
46.
      int numberOfFP = 0; //Variable um Anzahl der detektierten Merkmale zu speicher
    n
47.
      high_resolution_clock::time_point t1; //Variable um Zeit beim Aufnehmen des ak
    tuellen Bildes zu speichern
48.
      high_resolution_clock::time_point tOld; //Variable um Zeit beim Aufnehmen des
    vorherigen Bildes zu speichern
49.
50.
      VideoCapture vc(0); //Kamera oeffnen, falls dies fehlschlaegt Programm beenden
51.             if (!vc.open(0)) {
52.
      cout << "Kamera konnte nicht geoeffnet werden!" << std::endl;
53.                         return 0;
54.             }
55.
56.
57.             while (true) {
58.                         vc >> frame; //Aktuelles Kamerabild in frame speichern
59.
      if (frame.empty()) { //Beenden der Schleife, wenn keine Daten der Kamera empfa
    ngen werden
60.                                     break;
61.                         }
62.
63.                         t1 = high_resolution_clock::now();
64.
65.
      resize(frame, frame, Size(320, 240)); //frame auf 320x240 Pixel skalieren
66.
      frame.copyTo(frameOF); //Aktuellen Frame in der Variable frameOF speichern
67.
      frame.copyTo(frameH); //Aktuellen Frame in der Variable frameH speichern
68.
      double fLength = 230.75753 * (frame.size().width / 320); //Variable fuer die B
    rennweite der verwendeten Kamera in Pixeln
69.
70.                         detector-
    >detect(frame, keypoints, noArray()); //Merkmale im Eingangsbild detektieren
71.
72.
      if (firstRound) { //Im ersten Durchlauf des Programms Werte uebergeben
```

```
73.                                 firstRound = false;
74.                                 frame.copyTo(frameOld);
75.                                 tOld = t1;
76.
      for (int i = 0; i < keypoints.size(); i++) { //Detektierte Merkmale in p0 spei
    chern
77.                                         p0.push_back(keypoints[i].pt);
78.                                 }
79.                                 numberOfFP = p0.size();
80.                         }
81.                         else { //Ab dem zweiten Durchlauf des Programms
82.
      vector<Point2f> obstacles; //Vektor um die Positionen der Merkmale mit Tiefenw
    erten unter 1 m zu speichern
83.
84.                                 if (p0.size() != 0) {
85.
      calcOpticalFlowPyrLK(frameOld, frame, p0, p1, status, err, Size(21, 21), 4, cr
    iteria); //Positionen detektierter Merkmale eines vorherigen Frames im aktuellen
     Frame berechnen
86.
      duration<double, std::milli> time_span = t1 - tOld; //Zeitspanne zwischen eine
    s vorherigen und dem aktuellen frame berechnen
87.
88.
      for (int i = 0; i < p0.size(); i++) { //Berechnung der Tiefenwerte für jedes d
    etektierte Merkmal
89.
      if (status[i] == 1) { //Punkte, bei denen die Position bestimmt werden konnte,
     in good_new speichern
90.
      good_new.push_back(p1[i]);
91.
92.
      if (time_span.count() != 0) {
93.
      Mat translation(3, 1, CV_64F); //Translationsvektor t
94.
      translation.at<double>(0, 0) = 0; //Negative Kamera-Geschwindigkeit in X-
    Richtung in cm/s
95.
      translation.at<double>(1, 0) = 0; //Negative Kamera-Geschwindigkeit in Y-
    Richtung in cm/s
96.
      translation.at<double>(2, 0) = -100; //Negative Kamera-Geschwindigkeit in Z-
    Richtung in cm/s
97.
98.
      Mat rotation(3, 1, CV_64F); //Rotationsvektor omega
99.
      rotation.at<double>(0, 0) = 0; //Negative Kamera-Rotation um die X-
    Achse in 1/s
100.
      rotation.at<double>(1, 0) = 0; //Negative Kamera-Rotation um die Y-
    Achse in 1/s
101.
      rotation.at<double>(2, 0) = 0; //Negative Kamera-Rotation um die Z-
    Achse in 1/s
```

```
102.
103.
      //Matrix A aufstellen
104.
      Mat a(2, 3, CV_64F); a.at<double>(0, 0) = -
    fLength; a.at<double>(0, 1) = 0; a.at<double>(0, 2) = -
    p1[i].x + (frame.size().width / 2) - 1;
105.
      a.at<double>(1, 0) = 0; a.at<double>(1, 1) = -fLength; a.at<double>(1, 2) = -
    p1[i].y + (frame.size().height / 2) - 1;
106.
107.
      //Matrix B aufstellen
108.
      Mat b(2, 3, CV_64F); b.at<double>(0, 0) = ((-
    p1[i].x + (frame.size().width / 2) - 1) * (-
    p1[i].y + (frame.size().height / 2) - 1)) / fLength; b.at<double>(0, 1) = -
    (fLength + (-p1[i].x + (frame.size().width / 2) - 1) * (-
    p1[i].x + (frame.size().width / 2) - 1) / fLength); b.at<double>(0, 2) = -
    p1[i].y + (frame.size().height / 2) - 1;
109.
      b.at<double>(1, 0) = fLength + (-p1[i].y + (frame.size().height / 2) - 1) * (-
    p1[i].y + (frame.size().height / 2) - 1) / fLength; b.at<double>(1, 1) = -((-
    p1[i].x + (frame.size().width / 2) - 1) * (-
    p1[i].y + (frame.size().height / 2) - 1) / fLength); b.at<double>(1, 2) = -(-
    p1[i].x + (frame.size().width / 2) - 1);
110.
111.
      //Matrix A * Vektor t = at
112.
      Mat at = a * translation;
113.
114.
      //Matrix B * Vektor omega = B omega
115.
      Mat bOmega = b * rotation;
116.
117.
      //Vektor um die Pixelgeschwindigkeiten u und v zu berechnen
118.
      Mat uv(2, 1, CV_64F); uv.at<double>(0, 0) = ((p1[i].x - p0[i].x) / (time_span.
    count() * 0.001)); uv.at<double>(1, 0) = ((p1[i].y - p0[i].y) / (time_span.count
    () * 0.001));
119.
120.
      Mat w = uv - bOmega;
121.
      Mat wt;
122.
      transpose(w, wt);
123.
      Mat zwischen = wt * w;
124.
      double wtw = zwischen.at<double>(0, 0);
125.
      Mat zwischen2 = wt * at;
126.
      double wtAt = zwischen2.at<double>(0.0);
```

```
127.
      double z = wtAt / wtw; //Berechneter Tiefenwert des betrachteten Pixels in cm
128.
      depthValues.push_back(z);
129.                                                                                  }
130.
131.                                                                 }
132.                                                             }
133.                                                 }
134.
      if (!good_new.empty()) { //Merkmale abhaengig des Tiefenwerts farblich in fram
    eOF markieren
135.
      for (int y = 0; y < good_new.size(); y++) {
136.
137.
138.
      if ((0 <= depthValues[y]) && (depthValues[y] < 100)) {
139.
      circle(frameOF, good_new[y], 2, CV_RGB(255, 0, 0), -1);
140.
      obstacles.push_back(good_new[y]); //Punkte die maximal 1 m von der Kamera entf
    ernt sind in obstacles speichern
141.                                                                 }
142.
      else if ((100 <= depthValues[y]) && (depthValues[y] < 200)) {
143.
      circle(frameOF, good_new[y], 2, CV_RGB(255, 128, 0), -1);
144.                                                                 }
145.
      else if ((200 <= depthValues[y]) && (depthValues[y] < 300)) {
146.
      circle(frameOF, good_new[y], 2, CV_RGB(255, 255, 0), -1);
147.                                                                 }
148.
      else if ((300 <= depthValues[y]) && (depthValues[y] < 400)) {
149.
      circle(frameOF, good_new[y], 2, CV_RGB(128, 255, 0), -1);
150.                                                                 }
151.
      else if ((400 <= depthValues[y]) && (depthValues[y] < 500)) {
152.
      circle(frameOF, good_new[y], 2, CV_RGB(0, 255, 0), -1);
153.                                                                 }
154.
      else if ((500 <= depthValues[y]) && (depthValues[y] < 600)) {
155.
      circle(frameOF, good_new[y], 2, CV_RGB(0, 255, 128), -1);
156.                                                                 }
157.
      else if ((600 <= depthValues[y]) && (depthValues[y] < 700)) {
158.
      circle(frameOF, good_new[y], 2, CV_RGB(0, 255, 255), -1);
159.                                                                 }
160.
      else if ((700 <= depthValues[y]) && (depthValues[y] < 800)) {
161.
      circle(frameOF, good_new[y], 2, CV_RGB(0, 128, 255), -1);
```

```
162.                                                                    }
163.
      else if (800 <= depthValues[y]) {
164.
      circle(frameOF, good_new[y], 2, CV_RGB(0, 0, 255), -1);
165.                                                                    }
166.
      else if (0 > depthValues[y]) {
167.
      circle(frameOF, good_new[y], 2, CV_RGB(0, 0, 255), -1);
168.                                                                    }
169.                                                                    else {
170.
      circle(frameOF, good_new[y], 2, CV_RGB(0, 0, 255), -1);
171.                                                                    }
172.                                                            }
173.                                                    }
174.
175.                                                    if (!obstacles.empty()) {
176.
      vector<int> labels; //Vektor um die Gruppen einzelner Hindernisse zu speichern
177.
      int cluster_count = cv::partition(obstacles, labels, distanceEuc(20)); //Hinde
    rnispunkte gruppieren
178.
      vector<vector<Point>> cluster(cluster_count); //Vektor um gruppierte Hindernis
    punkte zu speichern
179.
      vector <vector<Point>> hulls; //Vektor um Linien um Gruppen von Hindernissen z
    u speichern
180.
181.
      for (int i = 0; i < obstacles.size(); i++) {
182.
      cluster[labels[i]].push_back(obstacles[i]);
183.                                                            }
184.
185.
      for (int i = 0; i < cluster.size(); i++) { //Gruppierungen von Hindernissen na
    ch Anzahl filtern
186.
      if (cluster[i].size() > 3) {
187.
      vector<Point> hull;
188.
      convexHull(cluster[i], hull, false, true);
189.
      hulls.push_back(hull);
190.                                                                    }
191.                                                            }
192.                                                            if (!hulls.empty()) {
193.
      drawContours(frameH, hulls, -
    1, CV_RGB(255, 0, 0)); //Linien um gruppierte Hindernisse in frameH darstellen
194.                                                            }
195.                                                    }
196.
```

```
197.
      if ((good_new.size() < (0.9 * numberOfFP)) || (good_new.size() == 0)) { //Uebe
    rpruefen ob nach der Berechnung des optischen Flusses genug Merkmale wiedergefun
    den werden konnten
198.                                                    p0.clear();
199.
      for (int i = 0; i < keypoints.size(); i++) {
200.
      p0.push_back(keypoints[i].pt);
201.                                                    }
202.                                                    numberOfFP = p0.size();
203.
      frame.copyTo(frameOld); //Aktuellen Frame in frameOld speichern
204.                                                    tOld = t1;
205.                                            }
206.
207.                                 }
208.
209.
      Mat frameOld2; //Variable zum speichern eines vorherigen Bildes
210.                                 frameOld.copyTo(frameOld2);
211.
      for (int i = 0; i < p0.size(); i++) { //Markieren der detektierten Merkmale au
    f einem vorherigen Bild
212.
      circle(frameOld2, p0[i], 2, CV_RGB(255, 255, 0), -1);
213.                                 }
214.
215.
      drawKeypoints(frame, keypoints, frameFP, CV_RGB(255, 0, 0));
216.
      imshow("Aktuelles Bild mit detektierten Merkmalen", frameFP); //Gibt Bild mit
    detektierten Merkmalen aus
217.
      imshow("Ergebnis optischer Fluss", frameOF); //Gibt Bild mit dem Ergebnis der
    Berechnung des optischen Flusses aus
218.
      imshow("Vorheriges Bild", frameOld2); //Gibt Bild eines vorherigen Frames mit
    detektierten Merkmalen aus
219.
      imshow("Hindernisse", frameH); //Gibt das Eingangsbild mit markierten Hinderni
    ssen aus
220.
221.
      int taste = waitKey(10); //Beenden der Schleife wenn "Esc" gedrueckt wurde
222.                                 if (taste == 27) {
223.                                          break;
224.                                 }
225.
226.                                 depthValues.clear();
227.                                 good_new.clear();
228.                         }
229.           }
230.           catch (cv::Exception& e)
231.           {
232.                          cerr << e.msg << endl;
233.           }
234.           return 0;
```

```
235.  }
```