UNIVERSITÄT KOBLENZ · LANDAU
Fachbereich 4: Informatik

Fraunhofer IDM@NTU

360-Grad-Tracking für 3D-Objekte

Masterarbeit

zur Erlangung des Grades einer Master of Science (M.Sc.)
im Studiengang Computervisualistik

vorgelegt von

Florian Kathe

Erstgutachter:	Prof. Dr.-Ing. Stefan Müller (Institut für Computervisualistik, AG Computergraphik)
Zweitgutachter:	Dr. Marius Erdt (Fraunhofer IDM@NTU)

Koblenz, im Januar 2015

Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

	Ja	Nein
Mit der Einstellung der Arbeit in die Bibliothek bin ich einverstanden.	☐	☐
Der Veröffentlichung dieser Arbeit im Internet stimme ich zu.	☐	☐

.....
(Ort, Datum) (Unterschrift)

Zusammenfassung

Augmented Reality gewinnt heutzutage immer mehr an Bedeutung in Gebieten wie der Industrie, der Medizin oder der Tourismus-Branche. Dieser Anstieg kann durch die Möglichkeit der Erweiterung der realen Welt mit weiteren Information durch Augmented Reality erklärt werden. Somit ist dieses Verfahren zu einer Methode geworden, den Informationsfluss wesentlich zu verbessern.
Um ein System zu erstellen, dass die reale Welt mit Zusatzinhalten erweitert, muss die Relation zwischen System und realer Welt bekannt sein. Die gängigste Methode zum Erstellen dieser Verbindung ist optisches Tracking. Das System berechnet die Relation zur realen Welt aus Kamerabildern. Dabei wird eine Referenz in der realen Welt als Orientierung genutzt. Zumeist sind dies 2D-Marker oder 2D-Texturen, die in der Szene der realen Welt platziert werden. Dies bedeutet allerdings auch einen Eingriff in die Szene. Deshalb ist es wünschenswert, dass das System ohne eine solche Hilfe arbeitet.
Ein Ansatz ohne Manipulation der Szene ist Objekt-Tracking. In diesem Ansatz kann ein beliebiges Objekt als Referenz genutzt werden. Da ein Objekt viel komplexer als ein Marker oder eine Textur ist, ist es für das System schwerer, daraus eine Relation zur realen Welt herzustellen. Deshalb reduzieren die meisten Ansätze für 3D-Objekt-Tracking das Objekt, indem nicht das gesamte als Referenz dient.
Der Fokus dieser Arbeit liegt auf der Untersuchung, wie ein ganzes Objekt als Referenz genutzt werden kann, sodass das System oder die Kamera sich 360 Grad um das Objekt herum bewegen kann, ohne dass das System die Relation zur realen Welt verliert. Als Basis dient das Augmented Reality-Framework „VisionLib“. Verschiedene Erweiterungen wurden im Rahmen dieser Arbeit für 360-Grad-Tracking in das System integriert und analysiert. Die unterschiedlichen Erweiterungen werden miteinander verglichen.
Durch das Verbessern des Reinitialisierungsprozesses konnten die besten Ergebnisse erzielt werden. Dabei werden dem System aktuelle Bilder der Szene übergeben, mit dem das System schneller eine neue Relation zur realen Welt herstellen kann, wenn diese verloren geht.

Abstract

Today, augmented reality is becoming more and more important in several areas like industrial sectors, medicine, or tourism. This gain of importance can easily be explained by its powerful extension of real world content. Therefore, augmented realty became a way to explain and enhance the real world information.
Yet, to create a system which can enhance a scene with additional information, the relation between the system and the real world must be known. In order to establish this relationship a commonly used method is optical tracking. The system calculates its relation to the real world from camera images. To do so, a reference which is known is needed in the scene to serve as an orientation. Today, this is mostly a 2D-marker or a 2D-texture. These are placed in the real world scenery to serve as a reference. But, this is an intrusion in the scene. That is why it is desirable that the system works without such an additional aid.
An strategy without manipulating the scene is object-tracking. In this approach, any object from the scene can be used as a reference for the system. As an object is far more complex than a marker, it is harder for the system to establish its relationship with the real world. That is why most methods for 3D-object-tracking reduce the object by not using the whole object as reference.
The focus of this thesis is to research how a whole object can be used as a reference in a way that either the system or the camera can be moved in any 360 degree angle around the object without loosing the relation to the real world. As a basis the augmented reality framework, the so called VisionLib, is used. Extensions to this system for 360 degree tracking are implemented in different ways and analyzed in the scope of this work. Also, the different extensions are compared.
The best results were achieved by improving the reinitialization process. With this extension, current camera images of the scene are given to the system. With the hek of these images, the system can calculate the relation to the real world faster in case the relation went missing.

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

Augmented Reality, also die Anreicherung der realen Welt mit Zusatzinformationen, erfreut sich immer steigender Beliebtheit und kommt in vielen Bereichen bereits zum Einsatz. So kommt Augmented Reality, kurz AR, nicht nur in der Industrie, wie z. B. beim Autobau, oder in der Medizin zum Einsatz, sondern auch in immer mehr Bereichen des alltäglichen Lebens. Es gibt Computer- oder Smartphone-Spiele, die mit Augmented Reality arbeiten oder auch Einrichtungen, wie Museen oder touristische Attraktionen, die Augmented Reality dazu nutzen, Informationen interaktiver zu gestalten, ein Beispiel dazu ist in Abbildung 1 zu sehen.

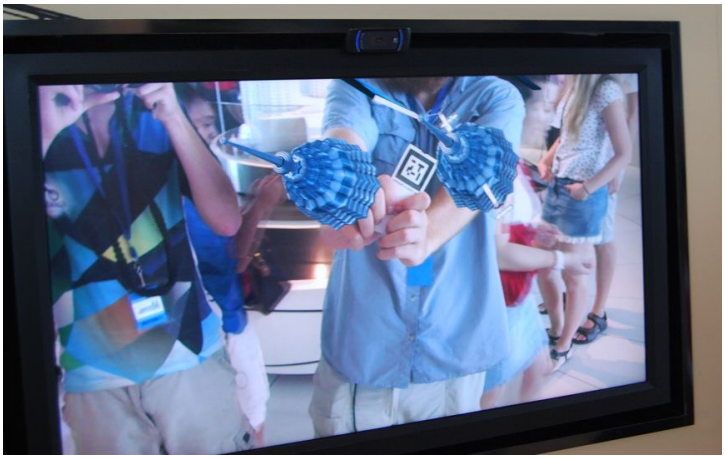


Abbildung 1: Augmented Reality in der Aussichtsplattform der Petronas Twin Towers in Kuala Lumpur, Malaysia [23]. Die Eintrittskarte kann dazu genutzt werden, sich auf dem Monitor die einzelnen Abschnitte der Türme anzeigen zu lassen. Durch das Drehen und Kippen der Karte kann man die Türme aus unterschiedlichen Blickwinkeln betrachten.

Um die reale Welt mit Informationen anzureichern, muss das Augmented Reality-System Wissen über die Welt haben. Dazu wird meist ein Referenzobjekt in der Welt benötigt, das dem System bekannt ist. Daraus kann es eine Relation zur realen Welt herstellen. In den meisten Fällen ist ein solches Referenzobjekt ein planarer Marker oder eine Textur. Das System ist darauf ausgelegt, diese Objekte wieder zu finden. Da ein in der Welt platzierter Marker optisch nicht ansprechend ist und eine Textur wenig Interaktionsmöglichkeiten bietet, wäre die Verwendung eines richtigen Objektes als Referenzobjekt von Vorteil.

1.2 Ziel der Arbeit

Die Verwendung eines dreidimensionalen Objekts als Referenzobjekt ist das Thema dieser Arbeit. Da ein Objekt im 3D-Raum aufgespannt wird, während ein Marker oder eine Textur nur im 2D-Raum aufgespannt wird, steigt damit auch die Komplexität für das Augmented Reality-System. Dadurch wird es schwieriger, das Objekt von allen Seiten wahr zu nehmen. Daher arbeiten die meisten für das Wahrnehmen von Objekten entwickelten Augmented Reality-System nicht mit dem ganzen Objekt. Sie können nur mit einem Teil des Objektes umgehen.
In dieser Arbeit soll untersucht werden, wie sich ein System realisieren lässt, mit dem man das ganze Objekt wahrnehmen kann. Es soll also möglich sein, einmal das System um 360 Grad um das Objekt herum zu bewegen, sodass das System das Objekt aus jedem Blickwinkel wahrnehmen kann.
Dazu wird bei dieser Arbeit ein bestehendes System des Fraunhofer-Institutes verwendet, welches analysiert wird und so angepasst werden soll, dass es die Zielsetzung erfüllt. Es werden Lösungsansätze für die Problemstellung entwickelt, diese evaluiert und miteinander verglichen.

1.3 Gliederung der Arbeit

Nachfolgend wird kurz der Aufbau dieser Arbeit vorgestellt. Abschnitt 2 befasst sich mit dem aktuellen Stand der Technik im Bereich des Augmented Reality und stellt gängige Verfahren vor. Außerdem wird darauf eingegangen, in wie weit schon Systeme entwickelt wurden, mit denen sich 360-Grad-Tracking realisieren lässt.
Abschnitt 3 stellt das für dieser Arbeit genutzte Framework vom Fraunhofer Institut vor und erklärt dessen Funktionsweise. Auch werden die vom Framework verwendeten Algorithmen präsentiert und genauer erläutert.
In Abschnitt 4 wird die Methode behandelt, wie die Perfomanz im Sinne des 360-Grad-Trackings bewertet und wie eine Evaluation umgesetzt werden kann.
Abschnitt 5 stellt die verschiedenen Ansätze vor, die in dieser Arbeit entwickelt wurden, um das bestehende Framework für das 360-Grad-Tracking anzupassen. Am Anfang des Abschnitts wird dargestellt, wie die Performanz des Frameworks im Bereich des Trackings zu Beginn der Arbeit war. Die Ansätze werden erklärt, evaluiert und miteinander verglichen.
Das Fazit dieser Arbeit wird in Abschnitt 6 gezogen. Darin wird die Arbeit zusammengefasst. Zusätzlich wird ein Ausblick gegeben, wie man die in dieser Arbeit entwickelten Ansätze weiter fortführen kann.

2 Stand der Technik

2.1 Augmented Reality

Augmented Reality, oder auch erweiterte Realität, ist ein Verfahren, um die reale Welt mit einer virtuellen Welt zu verknüpfen. Dabei wird mittels verschiedener Verfahren die virtuelle Welt in die reale integriert. Es wird also, im Gegensatz zur virtuellen Realität, bei der der Betrachter in die virtuelle Welt versetzt wird, die virtuelle Welt in der Umgebung der realen Welt angezeigt.
Die virtuelle Welt kann dabei entweder aus einer eigenen 3D-Welt bestehen, die perspektivisch korrekt in die reale Welt eingefügt wird, oder auch aus z. B. Texten oder Annotationen, die die reale Welt mit Zusatzinformationen ergänzen (Abbildung 2).
Das Zusammenführen dieser zwei Welten geschieht meistens optisch, z. B. am Computer oder an tragbaren Geräten wie einem Smartphone oder einem Tablet. Mittels einer Kamera wird eine Szene der realen Welt aufgezeichnet und auf dem Bildschirm dargestellt. Dieses Kamerabild wird von einer zweiten Ebene überlagert, auf der die virtuelle Welt projiziert wird. So entsteht für den Betrachter der Eindruck, dass diese zwei Welten miteinander verbunden sind.

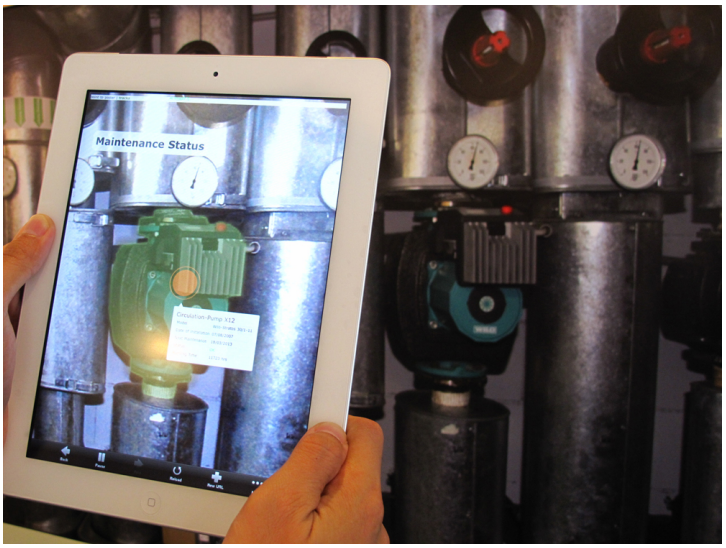


Abbildung 2: Augmented Reality im Bereich der Wartung. Auf dem Bildschirm werden dem Nutzer Informationen zum Zustand des betrachteten Systems gegeben.

Damit dieser Effekt für den Betrachter entsteht, muss sicher gestellt werden, dass die beiden Welten korrekt übereinander liegen. Dafür wird

die Szene der realen Welt analysiert und mittels eines Verfahrens die Relation von Kamera und realer Welt ermittelt. Mit Hilfe dieses Verfahrens kann anschließend die Kamera in der virtuellen Welt so gesetzt werden, dass sie der Kamera der realen Welt entspricht.

Um diese Relation zu ermitteln, gibt es verschiedene Möglichkeiten und Verfahren. Eine sehr einfache und rudimentäre Möglichkeit ist das Auslesen der in den heutigen Smartphones und Tablets verbauten Beschleunigungssensoren. Diese messen die Verschiebung des Geräts entlang der drei lokalen Raumachsen. Daraus lässt sich die Bewegung des Smartphones bzw. der Kamera in der realen Welt berechnen. Es gibt weitere ähnliche Verfahren, die durch eine zusätzliche Komponente die Verschiebung der Kamera in der realen Welt messen und dadurch die Relation der Kamera zur realen Welt berechnen.
Eine andere Möglichkeit besteht darin, auf eine zusätzliche Komponente zu verzichten und anhand des Kamerabildes Rückschlüsse darauf ziehen, wie sich die Kamera in Relation zur realen Welt verhält. Einige dieser Verfahren werden nachfolgend vorgestellt.

2.2 Optisches Augmented Reality und Tracking

Beim bildbasiertem Augmented Reality wird im akutellen Kamerabild nach Mustern oder Strukturen gesucht, die dabei helfen sollen, einen Rückschluss auf die Relation von Kamera zur realer Welt zu geben. Im Laufe der Zeit wurden mehrere Verfahren entwickelt, mit denen sich nur mittels Kamerabilder ein Augmented Reality-System realisieren lässt. Nachfolgend wird ein kurzer Überblick über die unterschiedlichen Verfahren gegeben und die Besonderheiten erläutert.

2.2.1 Markertracking

Bei Markertracking, erstmals vorgestellt von [28], wird in der realen Szene ein Referenzobjekt platziert. In diesem Fall ein zweidimensionales, quadartisches Objekt, genannt Marker (Abbildung 3), welches einen schwarzen Rand besitzt. Die Mitte eines solchen Markers besteht aus scharzen und weißen Flächen und dient der Unterscheidung, falls mehrere Marker im Bild sind.
Befindet sich ein solcher Marker im Kamerabild, kann er durch Bildverarbeitungsalgorithmen gefunden werden und somit kann die Relation von Kamera und realer Welt berechnet werden. Dies geschieht, indem man die Konturen des Markers im Kamerabild extrahiert. Hat man diese gefunden, kann man aus ihnen die vier Eckpunkte des Markers berechnen. Aus diesen wiederum lässt sich die 3D-Pose berechnen, die die Relation von Kamera

zur realen Welt beschreibt.

Abbildung 3: Markertracking in der Verwendung. Im Hintergrund ist ein Marker zu sehen, im Vordergrund wird das Kamerabild mit der virtuellen Welt überlagert. [6]

Dieses Verfahren hat den Vorteil, dass es einfach umzusetzen ist. Wird der Marker im Bild erkannt, kann davon ausgegangen werden, dass die berechnete Relation der wirklichen Relation entspricht. Der Nachteil, den das Verfahren mit sich bringt, ist, dass der Marker immer komplett im Bild sein muss, damit er vom Algorithmus erkannt wird. Dies schränkt die Bewegungsfreiheit der Kamera in der Welt ein. Außerdem kann es vorkommen, dass durch schlechte Lichtverhältnisse die Extraktion des Markers aus dem Bild nicht gelingt.

2.2.2 Kanten- und modellbasiertes Tracking

Eine Fortführung des Ansatzes des Markertrackings ist das Untersuchen aller Kanten im Kamerabild. Diese werden z. B. mit einem 3D-Linienmodel des gesuchten Objekts verglichen. Gibt es genügend Korrespondenzen, kann damit ein Rückschluss auf die Relation von Kamera zur realen Welt gezogen werden.
Es muss also ein Objekt in der realen Welt geben, das im Kamerabild zu sehen ist. Des Weiteren muss von diesem Objekt ein 3D-Modell existieren, aus dem ein Linienmodell, z. B. der Konturen extrahiert werden kann. Nun können aus dem Kamerabild die Kanten extrahiert werden und mittels einer Metrik mit dem Linienmodel verglichen werden. Liefert die Metrik genügend Korrespondenzen, kann daraus die 3D-Pose der Kamera in der realen Welt ermittelt werden.

So werden in [32] CAD-Modelle von Objekten verwendet, um diese im Kamerabild wieder zu finden und daraus die Relation von Kamera und realer Welt zu berechnen.

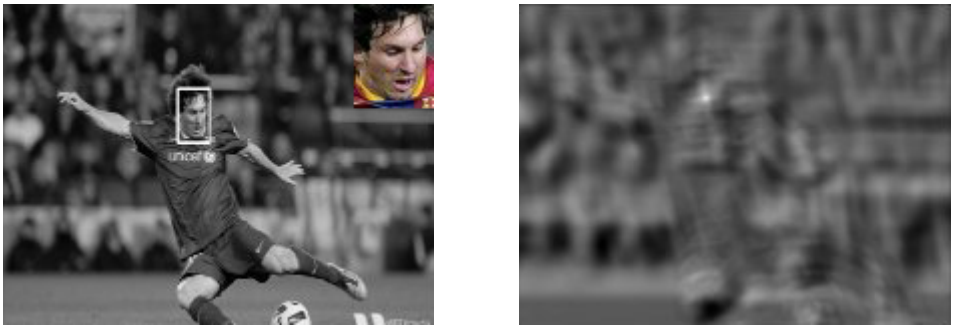
Abbildung 4: Kanten- und modellbasiertes Tracking. Das im Kamerabild zu sehende Fahrzeug wird mit seinem CAD-Modell perspektivisch korrekt überlagert. [32]

2.2.3 Bildvergleiche

Bei einem Bildvergleich wird versucht, ein Objekt in einem Bild wieder zu finden, indem ein Referenzbild und das Bild mit einer Metrik miteinander verglichen werden.
Dazu wird das Referenzbild über das Bild gelegt und der Unterschied berechnet. Wird nun das Referenzbild an verschiedenen Stellen über das Bild gelegt, erhält man für jede Stelle den Wert des Unterschiedes. Dort, wo der Unterschied am geringsten ist, befindet sich das Referenzbild bzw. das gesuchte Objekt. In Abbildung 5 ist ein solcher Vergleich und das Ergebnis zu sehen.
Da es sehr kostenintensiv ist, über jede Position im Bild das Referenzbild zu legen und den Unterschied zu berechnen, gibt es verschiedene komplexere Ansätze in dem Bereich. In Abschnitt 3.2 wird ein solcher Ansatz vorgestellt.

2.2.4 Merkmal-basiertes Tracking

Beim Merkmal-basierendem Tracking wird ein etwas anderer Ansatz verfolgt. Anstatt im Kamerabild nach Kanten zu suchen, werden hierbei aus



(a) (b)

Abbildung 5: Ein Bildvergleich zum Finden eines Bildausschnittes [24]. Vergleich eines Referenzbildes, zu sehen in der Abbildung 5(a) oben rechts, mit einem der Abbildung 5(a). Es wird nach der Position des Kopfes im Bild gesucht. Dazu wird das Referenzbild über jede Position im Bild gelegt und die Gemeinsamkeit berechnet. In Abbildung 5(b) ist für jede Position im Bild die Gemeinsamkeit als Helligkeit eingezeichnet. Man erkennt, dass an der Position, an der sich im eigentlichen Bild sich der gesuchte Kopf befindet, die Helligkeit der Pixel am höchsten ist und sich somit der Kopf dort befinden muss.

dem Kamerabild sogenannte *Merkmale* extrahiert. Dies sind spezielle Bildpunkte im Kamerabild, die sich gut durch ihre Umgebung beschreiben lassen. Es gibt ebenfalls daür unterschiedliche Metriken. Eine oft verwendete Metrik ist die der Eckpunkte. Hier werden im Kamerabild Eckpunkte, also Schnittpunkte von zwei Kanten, gesucht.
Auch ist ein Referenzobjekt nötig, um daraus die Relation von Kamera und realer Welt zu berechnen. Dies muss allerdings kein spezieller Marker sein, sondern ein Objekt mit beliebiger Textur. Aus diesem werden in einem ersten Schritt die Merkmale extrahiert und gespeichert. Anschließend können in jedem Kamerabild die Merkmale extrahiert werden und mittels einer Metrik verglichen und zugeordnet werden. Sind genügend Korrespondenzen vorhanden, lässt sich daraus die 3D-Pose der Kamera in der realen Welt ermitteln.
Ein Merkmal besteht somit aus einer Stelle oder Fläche im Bild und einem Maß, welches das Merkmal beschreibt und ihm einen Bezeichner gibt. Mit diesem können später andere Merkmale, z. B. aus einem anderem Kamerabild, diesem Merkmal wieder zugeordnet werden. Ein Merkmal-basiertes Tracking besteht damit aus einem Extraktor, der aus dem Kamerabild die Merkmale extrahiert und einem Deskriptor, der ihnen einen Bezeichner gibt, sowie aus einem Zuordner, der die Merkmale zwischen z. B. Kamerabild und Referenzobjekt einander zuordnet. Als Drittes wird ein Verfahren benötigt, welches aus diesen Zuordnungen eine 3D-Pose berechnen kann.

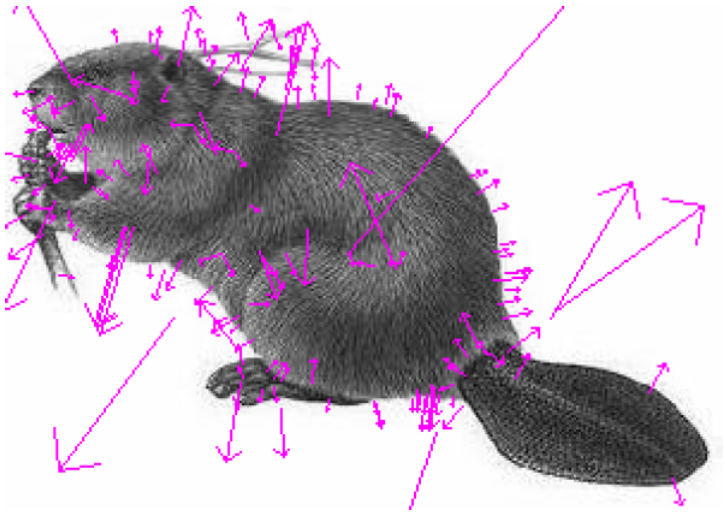


Abbildung 6: Extrahierte SIFT-Features einer Zeichnung eines Bibers. Dabei entspricht der Ursprung jedes Pfeils dem Merkmal und der Pfeil selbst veranschaulicht den Deskriptor. [12]

Einer der bekanntesten und robustesten Merkmals-Extraktoren ist der in [21] vorgestellte Ansatz *Scale Invariant Feature Transform* (SIFT). Es ist ein Deskriptor, der zunächst von dem Eingabebild eine Bildpyramide aufbaut. Dies geschieht, indem er das Bild in m unterschiedliche Größen skaliert. Danach wird jede dieser m Stufen in n Schritten mit *Gaussian Bluring* gefiltert. Durch die Subtraktion der auf einer Stufe liegenden Bilder erhält man mit *Difference of Gaussian* Ecken und Kanten. Die in der 26er Nachbarschaft liegenden Minima und Maxia ergeben die Merkmale des Kamerabildes.
Bestimmt man in einer 16×16 großen Nachbarschaft für jeden Bildpunkt den Gradientenwinkel und trägt diesen in ein Histogramm ein, welches den Vollwinkel in gleichgroße Abschnitte unterteilt, erhält man den Bezeichner für das Merkmal. Richtet man die Winkel am Gradientenwinkel des Merkmals aus, kann man rotationsinvariant Merkmale miteinander vergleichen und mittels Metrik einander zuordnen.

2.2.5 SLAM

SLAM, kurz für Simultaneous Localization And Mapping [19], nutzt ebenfalls Merkmale, allerdings vergleicht dieser Ansatz nicht das Kamerabild mit einem Referenzobjekt, sondern mit den zuvor aufgenommenen Kamerabildern. Dadurch kann die Transformation der Kamera in der Welt von Bild zu Bild berechnet werden. Gleichzeitig ist es möglich, aus mindestens zwei verschiedenen Kamerabildern die aufgenommene Szene der Welt in 3D zu rekonstruieren.

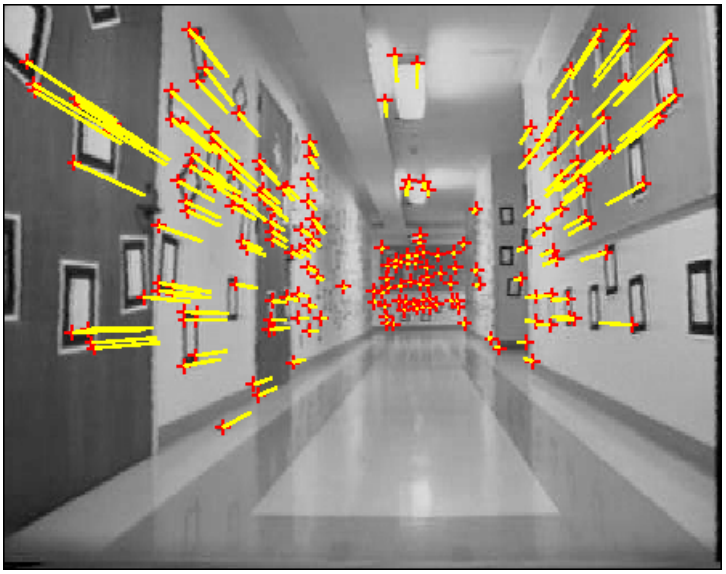


Abbildung 7: Visualisierung des SLAM-Algorithmus. Im Hintergrund ist das Kamerabild dargestellt. Im Vordergrund zeigen die roten Kreuze die aktuellen Merkmale im Bild an. Als gelbe Linie sind die Verbindungen zu den entsprechenden Merkmalen aus vorherigen Bildern visualisiert. [3]

2.3 360-Grad-Tracking

Nachfolgend werden Ansätze vorgestellt, die das Thema des 3D-Objekt-Tracking behandeln und fähig sind, Objekte von allen Blickwinkeln aus wiederzufinden.
In [10] wird ein System beschrieben, mit dem aus einer Sequenz von Bildern ein 3D-Modell von einem Objekt erstellt werden kann. Dieses kann später in einer anderen Sequenz von Bildern mit dem System wiedergefunden und dessen Pose im Bild berechnet werden. Dazu wird in einem ersten Schritt mittels des *SIFT*-Algorithmus aus den Referenzbildern die Merkmale extrahiert und zwischen den einzelnen Bildern Korrespondenzen gezogen. Um die Zuordnung der Korrespondenzen zu vereinfachen, werden diese durch den *Best-Bin-First*-Algorithmus angenähert. Aus diesen Korrespondenzen lässt sich mittels Triangulation das 3D-Modell des Objekts erstellen. Die Merkmale des erstellten Modells können mit Bildern z. B. einer Kamera verglichen werden. Mit einer Metrik werden ähnliche Merkmale einander zugeordnet. Aus dieser Zuordnung lässt sich die Pose des Objektes im Bild mit einer Kombination des *RANSAC*- und *Levenberg-Marquardt*-Algorithmus bestimmen.
In [20] wird ebenfalls ein System vorgestellt, dass ein Modell von einem Objekt erstellen und in anderen Bildern wiederfinden kann. Im Gegensatz zum vorher beschriebenen Ansatz wird hierbei allerdings ein 3D-CAD-

Modell (*Computer-Aided Design*) des Objekts benötigt, um ein Referenzmodell zu erstellen. Ebenfalls wird eine Sequenz von Bildern als Eingabe benötigt, die das Objekt aus verschiedenen Blickwinkeln zeigt. Durch die Zuordnung von Punkten aus den Bildern mit dem Modell des Objektes lassen sich die Kameraposen für jedes Bild bestimmen. Aus den Eingabebildern werden mit dem *Harris-Corner-Detector* Merkmale extrahiert, die wiederum mittels der Kamerapose auf das Modell projiziert werden können. Damit besteht ein Merkmal aus der Pixel-Nachbarschaft des Punktes und der Normalen des entsprechenden Punktes auf dem 3D-Modell. Außerdem wird die Kamerapose des Bildes gespeichert, aus dem das Merkmale extrahiert wurde. Um die Merkmale gegen die Veränderung des Blickwinkels stabil zu machen, wird die Pixel-Nachbarschaft mit unterschiedlichen Kameraposen neu aufgespannt und damit werden neue Referenzbilder erzeugt. Um die Kameraposen in einer neuen Sequenz von Bildern zu berechnen, werden die Bild mit den Referenzbildern verglichen. Um den Prozess zu beschleunigen, wird aber nur ein Referenzbild ausgewählt, das mit dem aktuellen Bild verglichen wird. Die Auswahl wird getroffen, indem bestimmt wird, welches Eingabebild am ähnlichsten zum vorherigen Bild der Sequenz war. Aus den Korrespondenzen zwischen aktuellem Bild und Referenzbild lassen sich auch die Korrespondenzen zu den 3D-Punkten des Modells ziehen. Daraus kann per *M-Schätzer* [17] die aktuelle Kamerapose berechnet werden. Als Eingabe für den M-Schätzer dient die berechnete Kamerapose des letzten Bildes. Die im Artikel vorgestellten Resultate zeigten allerdings nur Bildsequenzen mit 180-Grad-Tracking um ein Objekt.
Der in [26] vorgestellte Ansatz ist eine Kombination aus Kanten- und Textur-Tracking. Zunächst muss ein 3D-Modell eines Objekts erstellt werden, welches sowohl die Kanteninformationen des Objektes enthält, als auch die Texturen der planaren Flächen des Objekts. Während des Trackings versucht ein Kanten-Tracker die Korrespondenzen zwischen den extrahierten Kanten des aktuellen Kamerabildes und des Modells zu ziehen. Dies passiert, indem mit einer initialen Pose, z. B. die Kamerapose des letzten Bildes der Sequenz, das Kantenmodell auf die Bildebene projiziert wird und dann der Fehler zwischen den jeweiligen Kanten mit einem M-Schätzer minimiert wird. Ähnlich geschieht dies auch beim Textur-Tracking. Hier wird die Textur des Modells auf die Bildebene projiziert und dann der Unterschied zwischen den Intesitäten der Pixel minimiert. Die Kombination beider Verfahren erfolgt durch das Aufstellen einer gemeinsamen Matrix, die sowohl Kanten, als auch Textur-Informationen enthält. Werden beide Arten von Merkmalen auf eine einheitliche Skala gebracht, können die Fehler beider Arten von Merkmalen zusammen minimiert werden. Der Nachteil dieses Ansatzes besteht jedoch darin, dass er nur mit planaren Texturen arbeitet. Dies schränkt die Auswahl an Objekten für das Tracking stark ein.

3 Framework und Algorithmen

Nachfolgend wird das Framework beschrieben werden, mit dem gearbeitet wird. Zusätzlich wird ein Überblick über die bestehenden Algorithmen in diesem System und ein Ist-Zustand des Frameworks, wie es zu Beginn der Arbeit existierte, gegeben.

3.1 VisionLib

Die *VisionLib*-Architektur ist eine vom Fraunhofer IGD in Darmstadt entwickelte Architektur, die für die Entwicklung und Auswertung von bildbasierten Tracking-Algorithmen konzipiert wurde. [1]
Es gibt keinen allgemeinen Ansatz, der auf jedes Tracking-Problem angewendet werden kann. Deshalb wurde diese Architektur entwickelt, um ein flexibles und schnell re-konfigurierbares System zu besitzen. Dieses kann bei verschiedenen Aufgaben angewendet werden. Abbildung 8 zeigt die VisionLib-Architektur in ihrer Framework-Umgebung, genannt *InstantVision*.

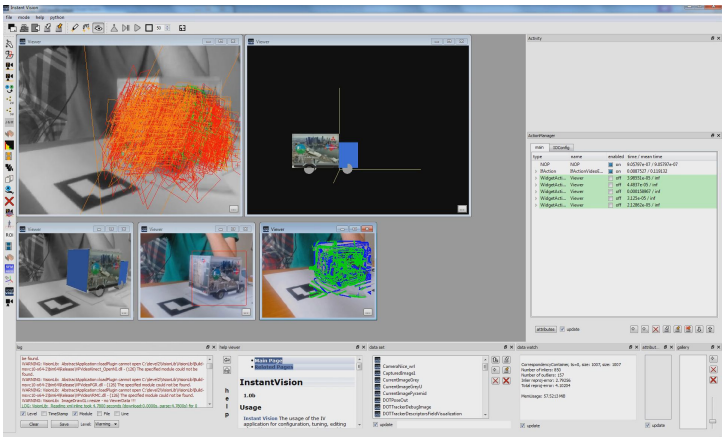


Abbildung 8: InstantVision in der Benutzung. Es wird ein Tracking-Algorithmus im Framework ausgeführt.

Es wurden viele Funktionen implementiert, die es ermöglichen, einfache, neue Programmabläufe zu entwickeln. So gibt es Werkzeuge, mit denen man u. a. eine Kamera kalibrieren oder Szenen virtuell rekonstruieren kann, aber auch Werkzeuge, um schnell kleine Anwendungen zu erstellen und einen Tracking-Ablauf auszuführen.
Außerdem wurde darauf geachtet, dass Tracking-Systeme schnell erstellt und auf einen hohen Abstraktionsgrad angepasst werden können. Zusätzlich können interne Daten visualisiert werden, was dabei helfen soll, Probleme schneller zu lokalisieren und Resultate zu bewerten. Deshalb sind

die Daten auch zur Laufzeit zugänglich und veränderbar.
Viele Tracking-Algorithmen gleichen sich in Teilen ihres Systemablaufes. Daher wurden diese Algorithmen in kleinere Blöcke aufgeteilt und diese können in mehreren Algorithmen wieder verwendet werden. Zusätzlich wurde der Datenaustausch zwischen diesen Blöcken vereinheitlicht, um einen schnellen Austausch einzelner Blöcke zu ermöglichen.
In Abbildung 9 werden die eben aufgeführten Funktionen der Architektur im Kontext des Frameworks beschrieben. Das Framework bietet die Möglichkeit, diese Funktionen zu nutzen und damit ein Tracking-System zu realisieren.

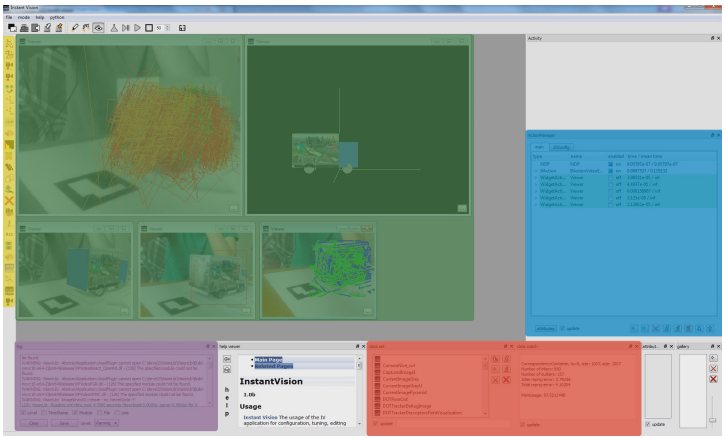


Abbildung 9: Elemente des InstantVision-Frameworks. Farblich hervorgehoben wurden die einzelnen Bestandteile, die im Framework der VisionLib zur Verfügung stehen. Gelb markiert sind die Menüpunkte, unter denen man u. a. eine Kamera kalibrieren und neue Tracking-Modelle erstellen kann. Die grüne Fläche beinhaltet Fenster, in denen verschiedenste Daten visualisiert werden können, wie z. B. die Merkmale eines Merkmal-basierten Trackers oder die Überlagerung von Kamerabild und virtueller Welt. Violett markiert ist das Log-Fenster, in dem Debug-Nachrichten und Ähnliches protokolliert werden. Im roten Bereich können alle vom System verwendeten Daten angezeigt und eingesehen werden. Blau hervorgehoben ist der Bereich, in dem die einzelnen Blöcke angezeigt werden, aus denen sich das Tracking-System zusammensetzt. Dort können auch deren Parameter verändert werden.

Das System, wie die einzelnen Blöcke eines Tracking-Systems Daten austauschen und sie organisiert werden, ähnelt dem *Robot Operating System*, kurz ROS [27]. Jeder einzelne Block ist ein eigener Knoten, hier genannt *Action*. Zwei Blöcke kommunizieren miteinander, indem sie auf die gleichen Daten zugreifen. Die Daten sind in der VisionLib in einem *DataSet* organisiert. Dabei können verschiedene Datentypen in eigenen Daten-Klassen gespeichert werden. Diese leiten sich von der Klasse *Data*.

Jede Action leitet sich von der selben abstrakten Klasse ab und bietet daher einige grundlegende Funktionen. So hat eine Action eine zuvor definierte Menge von Data-Klassen, auf die sie zur Laufzeit zugreifen kann. Außerdem hat sie Parameter, die auch zur Laufzeit geändert werden können. Es gibt eine `init`-Methode, die beim Initialisieren des Tracking-Systems aufgerufen wird und eine `apply`-Methode, die bei jedem Programmdurchlauf einmal aufgerufen wird.
Actions greifen per Pointer auf die Data-Klassen zu. Sie haben sowohl Lese- als auch Schreibrechte auf die Daten. Es kann aber definiert werden, ob die einzelnen Daten jeweils als Eingabe- oder Ausgabedaten gedacht sind. Jede Data-Klasse hat dabei einen eindeutigen Namen, mit dem die Actions die verschiedenen Data-Klassen identifizieren können. Im Gegensatz zu ROS werden die einzelnen Knoten bzw. Actions jedoch nicht gleichzeitig ausgeführt, sondern sequenziell, angeordnet in einer Pipeline. Abbildung 10 zeigt einen solchen Ablauf schematisch.

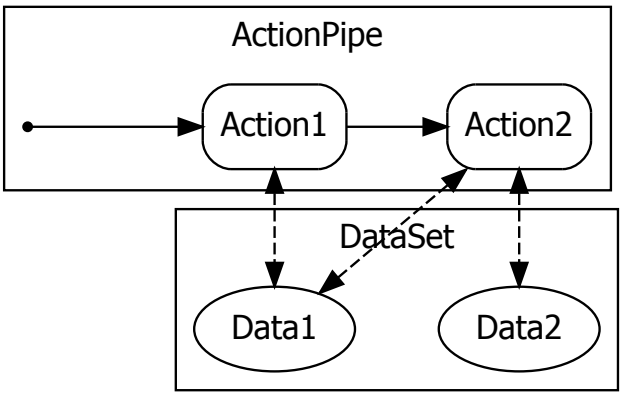


Abbildung 10: Beisiel einer Pipeline der VisionLib. Zuerst wird *Action1* ausgeführt, danach *Action2*. Beide haben dabei Zugriff auf die Daten in *Data1*, Action2 alleine auf die Daten in Data2.

Organisiert werden diese Pipelines in einer Datei mit XML-Struktur. Dort werden nacheinander die Actions aufgelistet, und zwar in der Reihenfolge, in der sie aufgerufen werden sollen. Zusätzlich wird definiert, auf welche Data-Klassen genau die einzelnen Actions zugreifen sollen. Außerdem können die Parameter der Actions definiert werden. Auflistung 1 zeigt eine in XML definierte Action und erklärt den Aufbau.

3.2 DOT-Tracking

Das in Abschnitt 3.3 beschriebene Merkmal-Tracking *KLT* benötigt als Initialisierungsschritt eine ungefähre Pose des Objektes im Kamerabild. Da-

```
<VideoSourceAction category="Action" name="
    VideoSourceAction"  enabled="1">
  <Keys size="2">
    <key val="InstantVideo"/>
    <key val="Intrinsics"/>
  </Keys>
  <ActionConfig source_url="ds://device=0;mode=;unit=0;"/>
</VideoSourceAction>
```

Auflistung 1: Die Action *VideoSourceAction* aus der VisionLib repräsentiert in XML. Das äußerste Element definiert die Action, die aufgerufen werden soll. Per Attribute kann man dieser in der Pipeline einen speziellen Namen geben, z. B. wenn die Action mehrfach vorkommt, und sie per `enable`-Attribut ein- und ausschalten. Das `Keys`-Element gibt an, dass darin die Data-Klassen definiert sind, die die Action verwenden kann. Das Attribut `size` gibt an, wie viele es sind. Jedes `key`-Element steht für eine Data-Klasse. Das Attribut `val` steht für den Bezeichner der entsprechenden Data-Klasse. Im `ActionConfig`-Element können per Attribute die Parameter der Action konfiguriert werden.
In diesem Beispiel handelt es sich um eine Action, die aus einer am Computer angeschlossenen Kamera das Bild ausliest und es in der Pipeline weiter gibt. In der `ActionConfig` ist im `source_url`-Attribut die Adresse zur Kamera enthalten. Das erste `key`-Element `InstantVideo` ist der Bezeichner, mit der das Bild als Data-Klasse im DataSet gespeichert werden soll. Entsprechend zeigt das Attribut `val` „`Intrinsics`“ des zweiten `key`-Elements auf die Data-Klasse der intrinsischen Parameter im DataSet.

für wird aktuell im InstantVision-Framework der in [13] beschriebene *DOT*-Tracker verwendet. DOT steht für Dominant Orientation Templates und ist ein *Template*-basiertes Tracking-Verfahren.
Template-basiertes Tracking ist ein Verfahren, bei dem Auschnitte im Bild gefunden werden sollen, indem sie mit einem Musterbild verglichen werden. Dies geschieht, indem das Musterbild über das gesamte Bild „geschoben“ wird und für jede Position mit einer bestimmten Metrik heraus gefunden wird, ob das Musterbild zum Bildausschnitt passt (Abbildung 11).

Für die Metrik, die bestimmt, ob der Bildausschnitt mit dem Musterbild übereinstimmt, gibt es verschiedenste Ansätze. Einer der ersten war der in [25] vorgestellte Ansatz, der die Konturen beider Bilder vergleicht. Einer der beliebtesten Ansätze heute ist *Hog* (Histogram of Oriented Gradients), [4]. Die Idee ist der Vergleich der Verteilung der Gradienten in den Bildern. Mit dieser Verteilung wird die Aussage getroffen, ob diese zueinander passen.
Der Nachteil dieses Verfahrens ist, dass es sehr lange Berechnungszeiten benötigt, da für jeden Bildauschnitt ein Vergleich mit dem Musterbild ge-

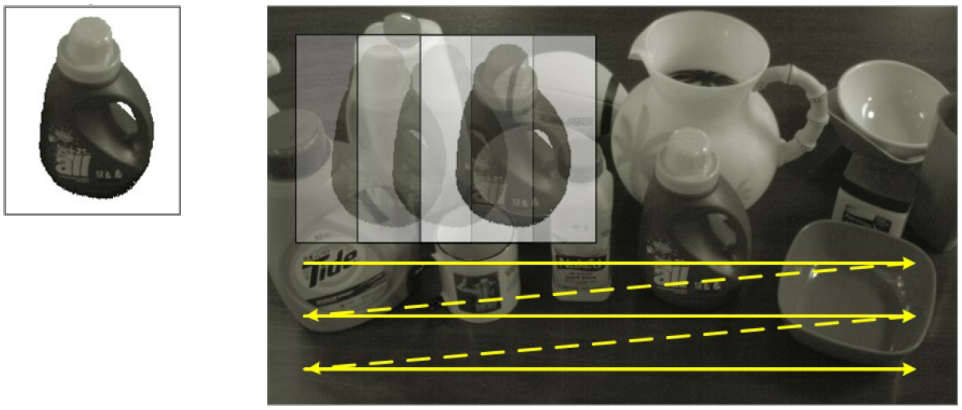


Abbildung 11: Durchführung von Template-Matching. Links ist das Musterbild zu sehen. Rechts das Bild, in dem das Musterbild gefunden werden soll. Dafür wird das Musterbild auf das Bild gelegt und für jede Position entlang der gelben Pfeile verglichen, ob das Musterbild hier mit dem Bildausschnitt übereinstimmt. [16]

macht wird. Dies kann bei wachsender Bildgröße sehr rechenintensiv werden. In der Initialisierungsphase ist es jedoch nötig, dass diese möglichst schnell fertig gestellt werden kann. DOT-Tracking vermeidet allerdings lange Berechnungszeiten, indem es mittels mehrerer Verfahren, die nachfolgend vorgestellt werden, den Prozess beschleunigt.

3.2.1 Algorithmus

Als Idee nutzt DOT-Tracking das Vergleichen von Bildern anhand der Orientierung ihrer Gradienten. Dies hat den Vorteil, dass Gradienten aussagekräftig und u. a. robust gegen Licht und Rauschen sind. Dabei wird jedoch nicht der absolute Wert des Gradienten gespeichert, sondern das Verhältnis zum stärksten Gradienten. Diese werden dann in eine Anzahl von *Bins* gespeichert, d. h. nach ihren Werten entsprechend eingeordnet.
Als Eingabe erwartet das DOT-Tracking ein Bild $\mathcal{I}$, ein Musterbild, nachfolgend Objekt genannt, $\mathcal{O}$ und einen Bildpunkt c in $\mathcal{I}$. Mit der Funktion

$$\mathcal{E}_1(\mathcal{I}, \mathcal{O}, c) = \sum_r \delta\big(ori(\mathcal{I}, c + r) = ori(\mathcal{O}, r)\big) \tag{1}$$

kann verglichen werden, wie gut das Objekt $\mathcal{O}$ zu der Stelle c im Bild $\mathcal{I}$ passt. $\delta(x)$ ist dabei eine binäre Funktion, die 1 zurück gibt, wenn x wahr und ansonsten 0 ist. $ori(\mathcal{O}, r)$ liefert die diskretisierte Gradientenorientierung an der Stelle r im Bild $\mathcal{O}$. Analog gilt dies für $ori(\mathcal{I}, c + r)$. Das Produkt $c + r$ beschreibt den Bildpunkt c, verschoben um r. Die Summe aller passenden Gradientenorientierungen ist ein absoluter Wert, der beschreibt,

wie gut das Objekt $\mathcal{O}$ zu dem Bildausschnitt bei c passt.
Da das Vergleichen jeder einzelnen Bildposition zu rechenintensiv wäre, wird sowohl Bild $\mathcal{I}$, als auch das Bild von Objekt $\mathcal{O}$ in kleine, quadratische Regionen $\mathcal{R}$ aufgeteilt. Dadurch können nur noch die dominanten Gradientenorientierungen miteinander verglichen werden. Die Funktion ändert sich dadurch wie folgt:

$$\mathcal{E}_2(\mathcal{I}, \mathcal{O}, c) = \sum_{\mathcal{R}\ in\ \mathcal{O}} \delta\big(do(\mathcal{I}, c + \mathcal{R}) \in DO(\mathcal{O}, \mathcal{R})\big) \tag{2}$$

Die Funktion $DO(\mathcal{O}, \mathcal{R})$ liefert eine Anzahl der stärksten Gradienten in der Region $\mathcal{R}$, wohingegen $do(\mathcal{I}, c + \mathcal{R})$ nur den stärksten Gradienten in $\mathcal{R}$, verschoben um c, des Bildes $\mathcal{I}$ zurück gibt.
Die Gradienten werden in beiden Fällen diskretisiert. Dies bedeutet, dass sie in einen von n_o vielen „Eimern" eingeordnet werden können. Im Kontext der Gradientenorientierung bedeutet dies, dass der Wertebereich der Orientierung $[0; 180]$ in n_o große Abschnitte aufgeteilt wird. Jede Gradientenorientierung gehört dann zu genau einem Abschnitt. Die Funktion $DO(.)$ gibt die k stärksten Gradienten zurück, eingeteilt in die jeweiligen Abschnitte. In der Praxis hat sich laut [13] $k = 7$ bewährt. Ist die Region $\mathcal{R}$ nicht aussagekräftig genug, wird $\{\perp\}$ zurück gegeben, um zu zeigen, dass sich dort keine starken Gradienten befinden. Die Funktion $do(.)$ gibt hingegen nur den stärksten Gradienten zurück. Dies geschieht, um Rechenzeit zu sparen. Abbildung 12 zeigt einen solchen Vergleich von $\mathcal{O}$ mit $\mathcal{I}$.

Um weitere Rechenzeit einzusparen, wird im Bild $\mathcal{I}$ nicht jede Bildposition c betrachtet, sondern nur eine bestimmte Anzahl an Positionen. Dafür muss das DOT-Tracking invariant gegen kleine Translationen gemacht werden. Dies geschieht, indem das Objekt $\mathcal{O}$ mit einer Transformation M verschoben wird. Die Formel ändert sich damit wie folgt:

$$\begin{aligned} \mathcal{E}_3(\mathcal{I}, \mathcal{O}, c) &= \max_{M \in \mathcal{M}} \mathcal{E}_2\big(\mathcal{I}, \mathrm{w}(\mathcal{O}, M), c\big) \\ &= \max_{M \in \mathcal{M}} \sum_{\mathcal{R}\ in\ \mathcal{O}} \delta\Big(do\big(\mathcal{I}, c + \mathcal{R}\big) \in DO\big(\mathrm{w}(\mathcal{O}, M), \mathcal{R}\big)\Big) \end{aligned} \tag{3}$$

Dabei ist $\mathrm{w}(\mathcal{O}, M)$, die Funktion, die $\mathcal{O}$ mit Hilfe von M transformiert. Es werden aber nur kleine Translationen der Größe $[-t; +t]^2$ verwendet. Es muss eine passende Größe für t gefunden werden, bei der zwischen Beschleunigung des Vorgangs und Verlust der Genauigkeit abgewogen werden muss. Hier empfielt [13] für ein 640×480 großes Bild $t = 7$.
Da die Regionen $\mathcal{R}$ unabhängig voneinander sind, können diese auch getrennt voneinander betrachtet werden. Außerdem können für jedes $\mathcal{R}$ in $\mathcal{O}$ die k stärksten Gradientenorietierungen vorher berechnet und gespeichert

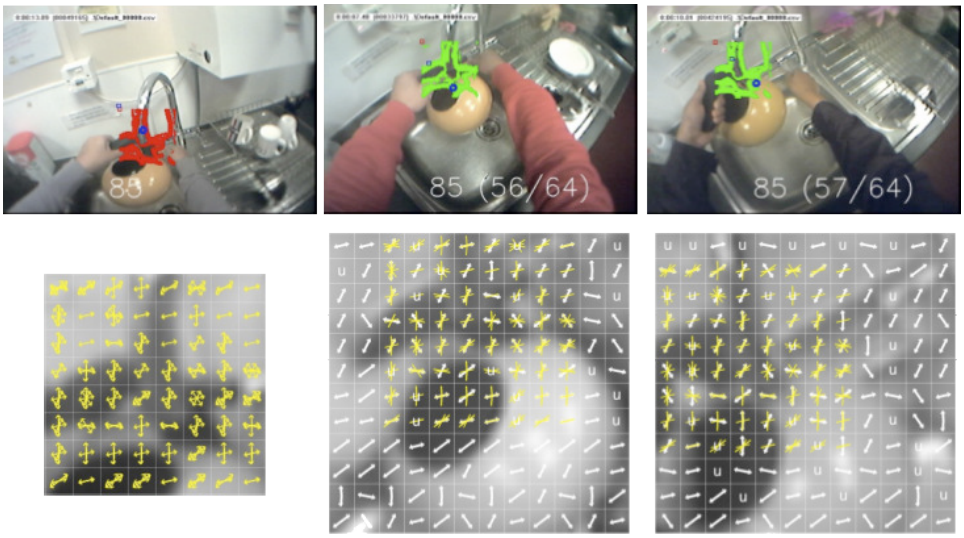


Abbildung 12: Beispiel für DOT-Tracking. Links ist das Objekt $\mathcal{O}$ zu sehen (rot markiert). Rechts zwei Kamerabilder $\mathcal{L}_1$ und $\mathcal{L}_2$, in denen die Stellen grün markiert sind, an denen $\mathcal{O}$ gefunden wurde. Darunter sind die jeweiligen Deskriptoren eingezeichnet. Jede Bündelung von Pfeilen ist eine Region $\mathcal{R}$, jeder Pfeil repräsentiert einen Gradientenorientierung(/-abschnitt). Gelb ist der Deskriptor von $\mathcal{O}$ und weiß jeweils von $\mathcal{L}_1$ und $\mathcal{L}_2$. [14]

werden mit der Funktion $\mathcal{L}(\mathcal{O}, \mathcal{R})$, um so weitere Rechenzeit während der Verarbeitung des Bildes $\mathcal{I}$ zu sparen. Damit vereinfacht sich $\mathcal{E}_3$ auf:

$$\begin{aligned} \mathcal{E}_4(\mathcal{I}, \mathcal{O}, c) &= \sum_{\mathcal{R}\ in\ \mathcal{O}} \max_{M \in \mathcal{M}} \delta\Big(do\big(\mathcal{I}, c+\mathcal{R}\big) \in DO\big(\mathrm{w}(\mathcal{O}, M), \mathcal{R}\big)\Big) \\ &= \sum_{\mathcal{R}\ in\ \mathcal{O}} \delta\big(do(\mathcal{I}, c+\mathcal{R}) \in \mathcal{L}(\mathcal{O}, \mathcal{R})\big) \end{aligned} \tag{4}$$

$\mathcal{L}(\mathcal{O}, \mathcal{R})$ speichert somit die k stärksten Gradientenorientierungen in ganz $\mathcal{M}$ über $\mathcal{R}$. Alle $\mathcal{L}(\mathcal{O}, \mathcal{R})$ der Regionen $\mathcal{R}$ in $\mathcal{O}$ bilden zusammen das Objekttemplate für $\mathcal{O}$.

Um das Vergleichen von Template und Bild zu beschleunigen, wird die Liste $\mathcal{L}(\mathcal{O}, \mathcal{R})$ und die dominante Orientierung $do(\mathcal{I}, c+\mathcal{R})$ binär gespeichert. Somit kann das Template und Bild mit ein paar binären Operationen verglichen werden. Nimmt man für die Gradientenorientierung 7 Bins, können damit $\mathcal{L}(\mathcal{O}, \mathcal{R})$ und $do(\mathcal{I}, c+\mathcal{R})$ in jeweils einer 8-Bit Integer Zahl gespeichert werden, da noch ein Bit für den Status $\{\perp\}$ benötigt wird, um anzeigen zu können, wenn kein starker Gradient gefunden wurde.
Ist also ein Bin in $\mathcal{L}(\mathcal{O}, \mathcal{R})$ einer der stärksten Gradienten, wird dessen Binärstelle auf 1 gesetzt. Bei $do(\mathcal{I}, c+\mathcal{R})$ wird die Binärstelle des Bins auf

1 gesetzt, dessen Gradient am stärksten ist. Somit lässt sich die Funktion $\delta\big(do(\mathcal{I}, c + \mathcal{R}) \in \mathcal{L}(\mathcal{O}, \mathcal{R})\big)$ wie folgt beschreiben:

$$\delta\big(do(\mathcal{I}, c + \mathcal{R}) \in \mathcal{L}(\mathcal{O}, \mathcal{R})\big) = 1 \; if \; L \otimes D \neq 0 \tag{5}$$

mit $\otimes$ als binären *AND* Operator.

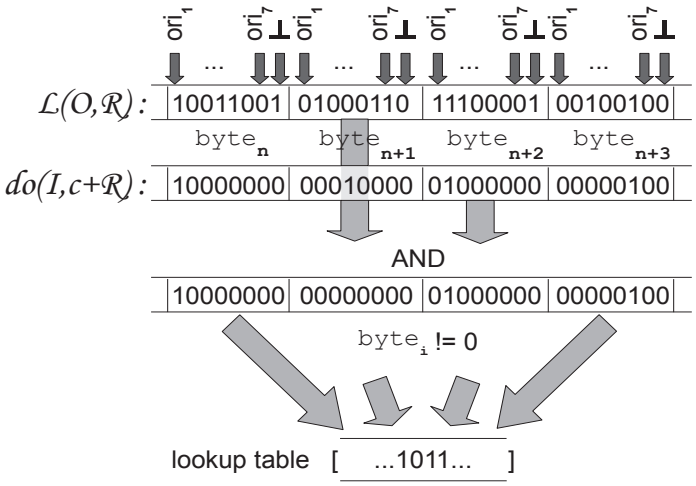


Abbildung 13: Vergleich von Template und Bild auf Binärebene. Es wird die Gleichung 4 gezeigt, wobei die Funktion $\delta()$ aus Gleichung 5 übernommen wurde. Jede Spalte steht dabei für ein $\mathcal{R}$ aus $\mathcal{O}$. [13]

SSE Technologie kann dieses binäre Verfahren nutzen und so 16 Regionen $\mathcal{R}_{1...16}$ parallel verarbeiten.
Gibt es mehrere Templates von verschiedenen Objekten oder aus unterschiedlichen Kameraperspektiven eines Objektes, können Vergleiche gespart werden, indem ähnliche Templates zu *Cluster* zusammengefasst werden. Wird nun eines der Templates aus dem Cluster mit dem Kamerabild verglichen und ist es nicht ausreichend ähnlich, kann von allen anderen Templates aus dem Cluster auch angenommen werden, dass diese nicht ähnlich genug dem Kamerabild sind. Um diese Cluster zu erzeugen, wird ein Template zufällig ausgewählt und nach anderen Templates gesucht, die die Bedingung

$$\underset{T \notin Cluster_i}{\operatorname{argmin}} \; \max\big(d_h(C \operatorname{or} T, T), d_h(C \operatorname{or} T, C)\big) \tag{6}$$

erfüllen. Dabei ist d_h die Hamming-Distanz und or der binäre or-Operator zwischen dem Cluster C und dem Template T. Ein Cluster wird dabei solange mit Templates gefüllt, bis entweder eine bestimmte Anzahl an Templates im Cluster sind oder alle Templates einem Cluster zugeordnet sind.

3.2.2 Effizienz

Der vorgestellte Algorithmus zeigt bei einem Vergleich mitdem *HoG*-Algorithmus (Histograms of Oriented Gradients, [5]) und anderen Algorithmen in [13], dass er genau so zuverlässig im Bereich des Trackings ist, allerdings einen deutlichen Vorteil in der Performance hat. So ist der DOT-Algorithmus bei der Verwendung von 1600 Templates 310 mal schneller als der HoG-Algorithmus und braucht rund $30ms$, um Templates in einem Bild zu finden. Dabei muss die Größe der Region $\mathcal{R}$ richtig gewählt werden, um einen guten Kompromiss zwischen Zuverlässigkeit und Geschwindigkeit zu finden. In den Versuchen hat sich gezeigt, dass eine Fenstergröße von 7×7 Pixeln ein guter Kompromiss ist. Der DOT-Algorithmus ist gut geeignet, sowohl um untexturierte Objekte zu tracken, als auch planare Bildausschnitte zu detektieren. Außerdem ist er robust im Tracken von 3D-Objekten und damit für die Verwendung im Kontext des 360-Grad-Trackings gut geeignet.

3.2.3 Umsetzung

Ziel beim DOT-Tracking eines Objektes ist es, das Template eines Referenzbild aus einer Menge an Referenzbildern zu finden, das dem aktuellen Kamerabild am ähnlichsten ist. Da für jedes Referenzbild die Kamerapose gespeichert ist, mit der die Kamera auf das Objekt schaut, kann diese Pose für das aktuelle Kamerabild übernommen werden. In Abbildung 14 werden DOT-Templates gezeigt.

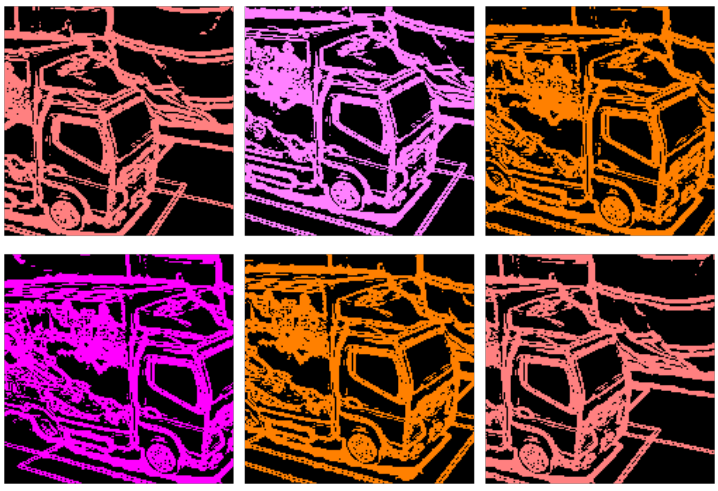


Abbildung 14: DOT-Templates der VisionLib visualisiert. Diese Referenzbilder werden während des Trackings mit dem aktuellen Kamerabild verglichen.

In VisionLib wurde der DOT-Algorithmus in mehreren Actions implementiert. Um ein Datenset von DOT-Templates zu erstellen, wird während der Erstellung eines Tracking-Modells für ein Objekt eine Action ausgeführt. Diese erstellt aus den Bildern, die für die Erstellung des Modells zu Verfügung stand, DOT-Templates welche in einer Data-Klasse gespeichert werden. Dieses Data-Klasse kann später während des Trackings verwendet werden. In Abbildung 15 sind die Kameraposen der in einem DOT-Template-Datensatz gespeicherten Templates visualisiert.

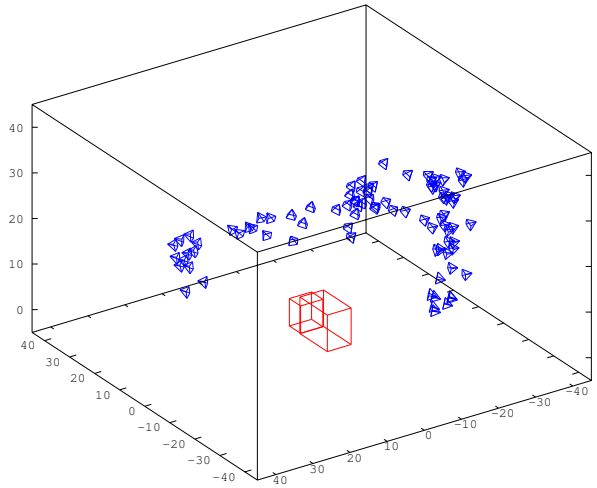


Abbildung 15: Visualisierung der DOT-Data-Klasse. Rot gezeichnet ist das Objekt, welches getrackt werden soll. Blau eingezeichnet sind die zugehörigen Kameraposen der DOT-Templates. Für jedes Template wurde eine entsprechende Kamerapose gespeichert.

Soll nun in einer Sequenz von Kamerabildern das Objekt getrackt werden, wird jedes Kamerabild mit den in der Data-Klasse gespeicherten Templates verglichen. Es wird die zum Template entsprechende Kamerapose übernommen, welche am Besten nach dem in Abschnitt 3.2.1 beschriebenen Algorithmus zum Kamerabild passt, und als Schätzung der aktuellen Kamera im Raum angegeben.
Dafür gibt es in der VisionLib eine Action *DOTTrackerAction*. Diese bekommt als Eingabe das aktuelle Kamerabild, die DOT-Templates mit den zugehörigen Kameraposen und die intrinsischen Parameter. Als Ausgabe liefert die Action die Pose des Templates, die am besten zu dem Eingabebild passt, in extrinsischen Kameraparameter. In Abbildung 16 ist der Vergleich von Kamerabild mit Template visualisiert. Wird kein passendes Template gefunden, werden extrinsische Kameraparameter ausgegeben, die nicht valide sind. Auflistung 2 zeigt diese Action in der XML-Struktur.

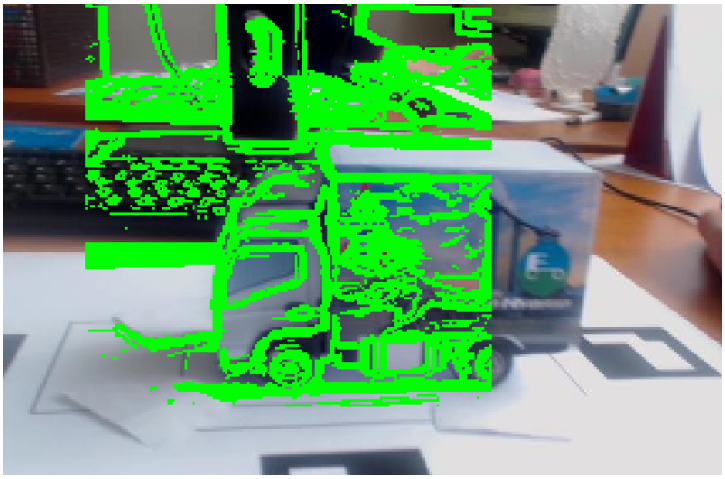


Abbildung 16: Überlagerung des Kamerabildes mit dem dazu am besten passenden DOT-Template. Das Template ist grün eingezeichnet an der Position, wo es dem Kamerabild am meisten gleicht.

3.3 KLT-Tracking

In InstantVision sind mehrere Merkmal-Tracker implementiert. Einer davon ist der *Kanade-Lucas-Tomasi Merkmal Tracker*, kurz *KLT*, er wurde vorgestellt in [22] und [29].
Die Grundidee von KLT ist, dass ein Merkmal „gut“ ist, wenn es sich gut verfolgen lässt. Dadurch muss der Extraktor des Algorithmus mit dem Deskriptor verbunden werden. Auch muss der Merkmal-Matcher, der aus zwei Bildern die jeweiligen Merkmale einander zuordnet, damit verknüpft sein.

3.3.1 Algorithmus

Ein gutes Merkmal ist für den KLT-Algorithmus ein Pixel p im Bild, das in der Nachbarschaft $W(p)$ sowohl in die x-Richtung, als auch in die y-Richtung einen starken Gradienten besitzt. Deshalb werden bei der Merkmal-Extraktion für jeden Pixel im Bild I die beiden Gradienten berechnet mit:

$$G = \sum_{x \in W(p)} \nabla I(x) \cdot \nabla I(x)^T \tag{7}$$

Dabei beschreibt $\nabla I(x)$ den Gradienten für den Pixel x. Berechnet wird dieser dabei z. B. mit dem Sobel-Operator, sodass $\nabla I = \frac{\partial I}{\partial (x,y)}$. $W(p)$ ist eine Region bestimmter Größe um den Punkt p. Daraus errechnet sich die Strukturmatrix G (Gleichung 7), die die Struktur der Umgebung des Pixels p beschreibt.

```
<DOTTrackerAction category="Action" name="DOTTrackerAction"
    >
  <Keys size="4">
    <key val="InstantVideo"/>
    <key val="World.Room.DOTDataModel1.DOTContainer"/>
    <key val="DOTPoseOut"/>
    <key val="Intrinsics"/>
  </Keys>
  <ActionConfig ComputeDebugImage="0"/>
</DOTTrackerAction>
```

Auflistung 2: Der DOT-Tracker repräsentiert in XML. `InstantVideo` enthält das aktuelle Kamerabild, `World.Room.DOTDataModel1.DOTContainer` ist die Data-Klasse, die alle DOT-Templates und die zugehörigen Kameraposen enthält und `Intrinsics` sind die intrinsischen Parameter. Diese drei Data-Klassen bekommt der DOT-Tracker als Eingabe. `DOTPoseOut` ist die Kamerapose des am besten passenden Templates als extrinsisches Parameter, dies liefert der DOT-Tracker als Ausgabe.

Mit den Eigenwerten λ_1 und λ_2 der Matrix G kann man eine Aussage darüber treffen, welche Struktur die Nachbarschaft W besitzt. Ist sie homogen, sind beide Eigenwerte λ_1 und $\lambda_2 = 0$. Ist der größere Eigenwert $\lambda_1 > 0$ und $\lambda_2 = 0$, zeigt dies an, dass eine Kante in der Nachbarschaft W vorhanden ist. Sind hingegen beide Eigenwerte $\lambda_1, \lambda_2 > 0$, deutet dies auf eine Ecke in W hin. Somit beschreibt der kleinere Eigenwert $\lambda = \min(\lambda_1, \lambda_2)$ die „Eckigkeit“ der Nachbarschaft W.
In Algorithmus 1 wird beschrieben, wie Merkmale aus einem Bild extrahiert und diese bewertet und ausgewählt werden. [11]

Algorithmus 1 Extraktion der Merkmale aus einem Bild I mittels des KLT-Algorithmus.

1: Berechne die Strukturmatrix G und die „Eckigkeit“ λ für jeden Pixel p im Bild I.
2: Ermittele das maximale λ_{max}, welches es im Bild gibt.
3: Verwerfe alle Pixel, die ein λ unter einem bestimmten Schwellwert, abhängig von λ_{max}, haben.
4: Finde die lokalen Maxima in einer 3×3 Nachbarschaft.
5: Wähle als entgültige Merkmale so viele verbleibende Merkmale wie nötig aus. Um Zentrierungen in bestimmten Regionen zu vermeiden, sollten alle Merkmale eine bestimmte Distanz, z. B. 10 Pixel, voneinander haben.

Um ein Merkmal über mehrere Bilder hinweg wiederzufinden und zu

verfolgen, wird für jedes Merkmal der Bewegungsvektor v berechnet, der angibt, wie sich das Merkmal von Bild zu Bild verhält. Wird dieses Merkmal nun z. B. im nächsten Kamerabild J gesucht, wird angenommen, dass der Bewegungsvektor v eines jeden Merkmals p zunächst $= 0$ ist. Damit wird dann der Unterschied E zwischen dem Bildauschnitt $J\big(W(p,v)\big)$ und dem Merkmal $I\big(W(p)\big)$ berechnet. Ebenfalls wird der Gradient ∇I an der Stelle $J\big(W(p,v)\big)$ berechnet und damit die Hesse-Matrix $H = \sum_{x \in W(p,v)} [\nabla I]^T [\nabla I]$ bestimmt (vergleiche Gleichung 7). Damit lässt sich die Translation Δv neu bestimmen durch

$$\Delta v = H^{-1} \sum_{x \in Wp,v} [\nabla I]^T \left[I\big(W(p)\big) - J\big(W(p,v)\big) \right] \tag{8}$$

und der Bewegungsvektor $v = v + \Delta v$ wird aktualisiert. Mit dem neuen Bewegungsvektor kann man den neuen Unterschied E zwischen J und I berechnen. Dies wird so lange wiederholt, bis v konvergiert. Ist dies der Fall, kann die neue Position des Merkmals $p = p + v$ im Bild J gesetzt werden. [30] Algorithmus 2 zeigt den eben beschriebenen Ablauf schematisch. Abbildung 17 zeigt den in Algorithmus 2 beschriebenen Ablauf mit Beispielbildern.

Algorithmus 2 Tracking eines Merkmale durch den KLT-Algorithmus.

$v = [0, 0]$
while $\|\Delta v\| > \epsilon$ **do**
  $E = J\big(W(p,v)\big) - I\big(W(p)\big)$ **(3.)**
  $\nabla J\big(W(p,v)\big)$ **(2.)**
  $H = \sum_{x \in W(p,v)} [\nabla J]^T [\nabla J]$ **(4.)**
  $\Delta v = H^{-1} \sum_{x \in Wp,v} [\nabla J]^T \left[I\big(W(p)\big) - J\big(W(p,v)\big) \right]$ **(5.)**
  $v = v + \Delta v$ **(6.)**
end while
$p = p + v$

Da der oben gezeigte KLT-Algorithmus nur bei kleinen Verschiebungen der Merkmale diese in zwei Bildern wiederfinden kann, kann man mittels einer Bildpyramide diesen Abstand erweitern. Es wird das Bild in verschiedenen Stufen skaliert und in der obersten, kleinsten Stufe wird die Verschiebung berechnet. Ist sie gefunden, wird diese auf die nächst tiefere Stufe umgerechnet und dort die Verschiebung erneut berechnet. Dies geschieht so lange, bis man an der untersten Stufe angelangt ist. Abbildung 18 zeigt eine solche Bildpyramide.

3.3.2 Umsetzung

Um ein Tracking-Modell eines Objektes mit VisionLib für das KLT-Tracking zu erstellen, gibt es im InstantVision-Framework einen eigenen Menüpunkt.

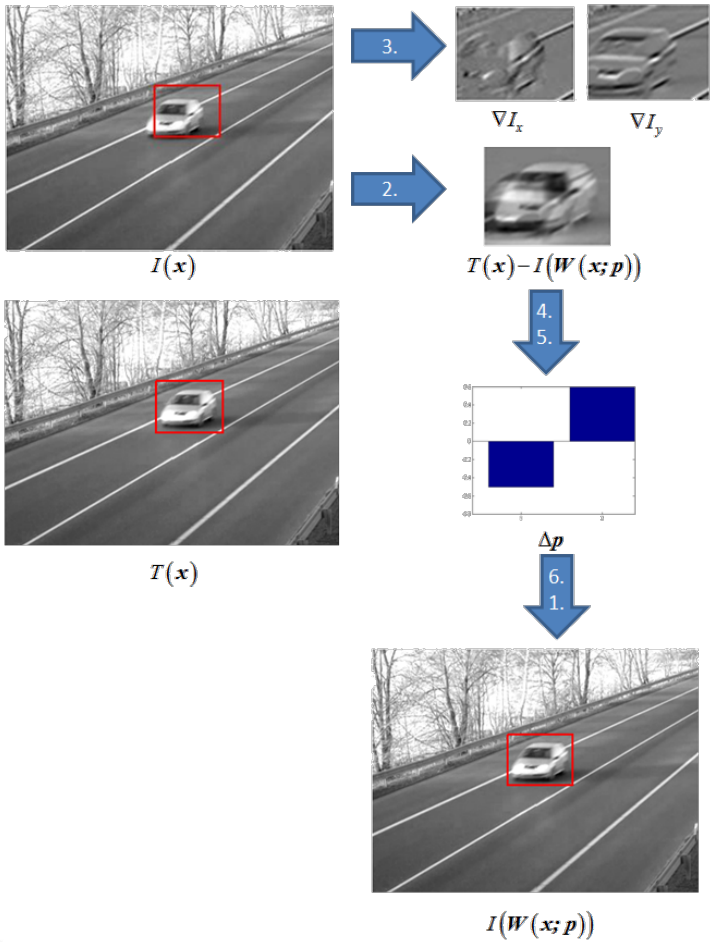


Abbildung 17: Tracking des KLT-Algorithmus. Die einzelnen Nummer entsprechen den Schritten von Algorithmus 2. [30]

Es wird eine Sequenz von Kamerabildern sowie intrinsische Kameraparameter benötigt. In jedem der Kamerabilder sucht KLT mit dem in Abschnitt 3.3.1 beschriebenen Algorithmus nach Merkmalen. Für jedes Bild werden die gefundenen Merkmale extrahiert. Danach werden die Merkmale zwischen den Bildern einander zugeordnet. Mittels Triangulation können die 2D-Merkmale in 3D-Merkmale umgerechnet werden. Alternativ können per Projektion der 2D-Merkmale auf ein gegebenes 3D-Modell des Objektes auch die entsprechenden 3D-Merkmale berechnet werden. Abbildung 19 zeigt die berechneten 3D-Merkmale von einem Objekt. Sind alle 3D-Merkmale berechnet, kann wiederum per linearer Pose-Schätzung und dem *RANSAC*-Algorithmus für jedes Bild der Sequenz die Pose der Kamera berechnet werden. Dies wird u. a. bei dem in Abschnitt 3.2.3 beschriebenen Verfahren zum Tracken von Objekten mittels DOT-Algorithmus benö-

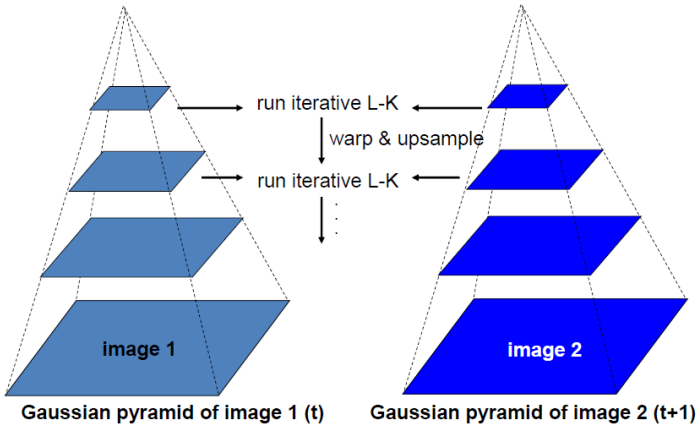
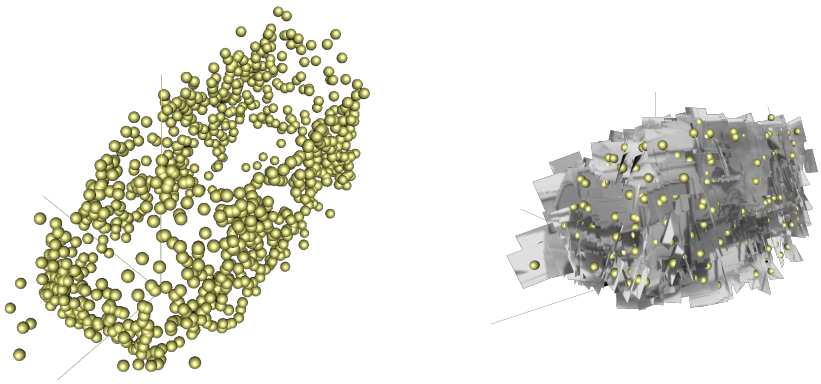


Abbildung 18: Bildpyramide. Beispiel für eine Bildpyramide, die mit dem KLT-Algorithmus verwendet werden kann. [15]

tigt.



(a) (b)

Abbildung 19: 3D-Merkmale eines Objekts, erstellt mit dem KLT-Algorithmus. In Bild 19(a) sind nur die 3D-Punkte zu sehen, bei Bild 19(b) wurden die 3D-Punkte mit den jeweiligen Deskriptoren überlagert.

Um mit diesem Modell in einem beliebigen Kamerabild nach dem Objekt zu suchen, wird als erstes eine initiale Kamerapose benötigt. Zusätzlich wird, da beim Erstellen der KLT-Merkmale eine Bildpyramide verwendet wurde, auch vom aktuellen Kamerabild eine Bildpyramide benötigt. Im ersten Schritt werden die 3D-Merkmale mit der aktuellen Kamerapose auf eine 2D-Ebene projiziert. Dort können die 2D-Merkmale angewendet werden und mit den 2D-Merkmalen des aktuellen Kamerabildes verglichen werden. Für den Vergleich von Tracking-Modell und aktuellem Kamerabild wird angenommen, dass der Verschiebungsvektor von Tracking-

Modell und Kamerabild berechnet werden kann. Dazu werden die auf eine 2D-Ebene projizierten Merkmale des Modells als Merkmale des Bildes I angesehen. Das aktuelle Kamerabild repräsentiert das neue Bild J. Nun wird nach Algorithmus 2 der Verschiebungsvektor zwischen den einzelnen Merkmalen der Bilder berechnet.

```
<BuildImagePyramidAction category="Action" enabled="1" name="BuildImagePyramidAction">
  <Keys size="2">
    <key val="InstantVideo"/>
    <key val="CurrentImagePyramid"/>
  </Keys>
  <ActionConfig ... />
</BuildImagePyramidAction>
<KLTProjectPointsAction category="Action" enabled="1" name="ProjectPoints">
  <Keys size="4">
    <key val="World.Room.KLTModel1"/>
    <key val="Intrinsics"/>
    <key val="DOTPoseOut"/>
    <key val="CurrentImagePyramid"/>
  </Keys>
  <ActionConfig ... />
</KLTProjectPointsAction>
<KLTTrackingAction category="Action" enabled="1" name="KLTTracking">
  <Keys size="3">
    <key val="CurrentImagePyramid"/>
    <key val="World.Room.KLTModel1"/>
    <key val="DOTPoseOut"/>
  </Keys>
  <ActionConfig ... />
</KLTTrackingAction>
<PoseRANSACAction category="Action" name="ransac" enabled="1">
  <Keys size="3">
    <key val="Intrinsics"/>
    <key val="World.Room.KLTModelFiltered.ValidCorrespondences"/>
    <key val="KLTPoseOut"/>
  </Keys>
  <ActionConfig ... />
</PoseRANSACAction>
<PoseNLLSAction category="Action" name="nlls" enabled="1">
  <Keys size="3">
    <key val="Intrinsics"/>
    <key val="World.Room.KLTModelFiltered.ValidCorrespondences"/>
    <key val="KLTPoseOut"/>
  </Keys>
  <ActionConfig ... />
</PoseNLLSAction>
```

Auflistung 3: Aufbau der Actions für KLT-Tracking im InstantVision-Framework in XML. Die Action `BuildImagePyramidAction` erzeugt aus einem Eingabebild eine Bildpyramide, `KLTProjectPointsAction` projiziert die 3D-Merkmale auf eine 2D-Ebene, `KLTTrackingAction` ordnet die Merkmale von Modell und Kamerabild einander zu und `PoseRANSACAction` und `PoseNLLSAction` berechnet daraus eine Pose für die Kamera. `World.Room.KLTModel1` ist der Bezeichner für die Data-Klasse, in der das KLT-Tracking-Modell gespeichert ist. Die Daten `InstantVideo`, `Intrinsics` und `DOTPoseOut` sind ebenfalls gegeben, siehe Auflistung 1 und 2.

Konnten genügend Merkmale zwischen den Bildern zugeordnet werden, können für die 2D-Merkmale im aktuellen Kamerabild die 3D-Punkte der korrespondierenden Merkmale aus dem Modell übernommen werden. Mittels RANSAC wird daraus und auf Basis der initialen Pose eine ers-

te genauerer Schätzung der Kamerapose berechnet. Durch ein *Non-Linear Least Squares*-Vefahren kann der Projektionsfehler dieser Pose nochmals minimiert und verbessert werden. So erhält man die entgültige Kamerapose für das aktuelle Kamerabild. Auflistung 3 zeigt den eben beschriebenen Ablauf in der XML-Struktur mit den dazugehörigen Actions aus der VisionLib.

3.4 Tracking mit VisionLib

In Abschnitt 3.2 und 3.3 wurde bereits beschrieben, wie die einzelnen Tracking-Algorithmen funktionieren. In diesem Abschnitt soll das Zusammenspiel zwischen den Komponenten genauer erklärt werden.
Da das KLT-Tracking zum Finden von Korrespondenzen eine initiale Pose benötigt, muss dafür gesorgt werden, dass zu jedem Zeitpunkt t_n der Bildsequenz eine Kamerapose existiert, die das KLT-Tracking als initiale Pose nutzen kann. Dabei sind drei Fälle zu unterscheiden:

a) Das KLT-Tracking konnte zum Zeitpunkt t_{n-1} für das Kamerabild *eine* Pose der Kamera berechnen.

b) Das KLT-Tracking konnte zum Zeitpunkt t_{n-1} für das Kamerabild *keine* Pose der Kamera berechnen.

c) Zum Zeitpunkt t_{n-1} existiert kein Kamerabild.

Für den ersten Fall wird angenommen, dass es sich bei der zu verarbeitenden Bildsequenz um eine kontinuierliche Reihe handelt, d. h., dass zwischen den einzelnen Bildern die Zeitsprünge klein sind. Dadurch kann auch angenommen werden, dass sich das Objekt zwischen zwei Bildern nur marginal ändert, also die Kamera sich nicht stark verschiebt bzw. rotiert. So kann als initiale Pose für das KLT-Tracking zum Zeitpunkt t_n die berechnete Pose des KLT-Trackings zu Zeitpunkt t_{n-1} genutzt werden.
Der zweite und dritte Fall werden zunächst gleich betrachtet. Existiert keine Pose vom KLT-Tracking zum Zeitpunkt t_{n-1}, wird zum Zeitpunkt t_1 das DOT-Tracking in der Pipeline aktiviert. Kann mittels DOT-Tracking eine Pose berechnet werden, kann diese als initiale Pose für das KLT-Tracking genutzt werden, andernfalls muss das aktuelle Bild der Sequenz übersprungen werden, da es nicht möglich ist, eine initiale Pose für das KLT-Tracking zu erzeugen. Abbildung 20 veranschaulicht diesen Ablauf schematisch.

3.5 Erstellung von 360-Grad-Modellen

Um ein Objekt aus allen Perspektiven, sprich 360 Grad, in einem Kamerabild wiederfinden zu können, braucht man ein Modell, welches das ganze Objekt repräsentiert.
Das Ziel dieser Arbeit ist es, ein Objekt um 360 Grad zu tracken, ohne dass

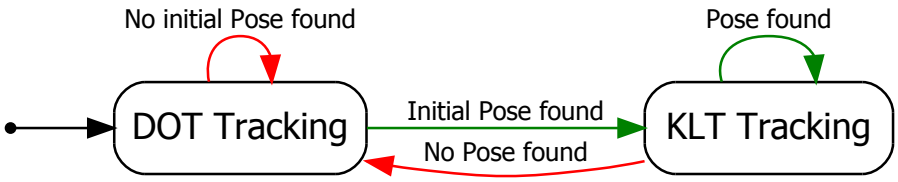
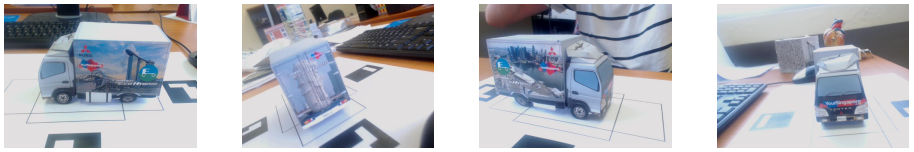


Abbildung 20: Tracking-Ablauf mit VisionLib. Das in Abschnitt 3.4 beschriebene Zusammenspiel von DOT- und KLT-Tracking. Beide Verfahren sind jeweils als ein Block dargestellt. Die Pfeile zeigen die Interaktionen an. Ein grüner Pfeil beteutet, dass der jeweilige Algorithmus eine Pose finden konnte und diese an die auf die gezeigte Action weitergibt. Ein roter Pfeil bedeutet, dass keine Pose gefunden wird und man in die Action wechselt, auf die der Pfeil zeigt.

für den Benutzer der Eindruck entsteht, dass das Programm sich neu initialisieren muss. Sprich, es soll möglich sein, mit der Kamera einmal komplett um ein Objekt zu gehen, ohne dass das Tracking das Objekt verliert.
Dafür wäre es von Vorteil, wenn das Modell, welches das Objekt repräsentiert, ein einziger Datensatz ist und sich nicht z. B. aus mehreren Datensätzen zusammensetzt.
Im InstantVision-Framework der VisionLib steht bereits die Funktionalität zur Verfügung, um aus einer Bildsequenz ein KLT- und DOT-Tracking-Modell zu erstellen. In Abbildung 19 ist zu sehen, dass es bei entsprechender Bildsequenz auch möglich ist, aus einem Objekt, wie z. B. in Abbildung 21, mit dem gegebenen InstantVision-Framework ein 360-Grad-Tracking-Modell zu erstellen.

Abbildung 21: Beispielobjekt zum Tracken. Ein Objekt, welches in dieser Arbeit als Beispiel verwendet wird.

Die benötigte Bildsequenz dafür sollte eine kontinuierliche Bildfolge sein, bei der - ohne hektische Bewegungen - die Kamera 360 Grad um das Objekt herum geführt wird. Abbildung 22 zeigt Ausschnitte einer solchen Bildsequenz.



(a) (b) (c) (d)

Abbildung 22: Bildsequenz eines Objektes zur Modellerstellung. Es wurde dabei mit der Kamera einmal um das Objekt herum geschwenkt, beginnend bei Abbildung 22(a) über 22(b), 22(c) und 22(d) und wieder endend in Abbildung 22(a).

Da der KLT-Algorithmus aus der Bildsequenz für die berechneten 2D-Merkmal per Triangulation auch 3D-Positionen der einzelnen Merkmale berechnen kann, ist es möglich, für jedes Bild der Sequenz die Kamerapose zu berechnen. Dies ist von Vorteil, da für eine Auswahl an Bildern der Sequenz die DOT-Templates bestimmt werden können. Durch die berechnete Kamerapose dieser Bilder mit dem KLT-Algorithmus, können den DOT-Templates diese Posen zugeordnet werden. Damit werden die Daten für das in Abschnitt 3.2.3 beschriebene DOT-Tracking der VisionLib generiert. Es ist also möglich, mit dem InstantVision-Framework ein 360 Grad KLT- und DOT-Tracking Modell eines Objektes zu erstellen.

4 Evaluations-Methode

In diesem Abschnitt wird beschrieben, wie die gemachten Änderungen an der Tracking-Pipline evaluiert wurden. Um zu bewerten, wie gut Augmented Reality funktioniert, muss gemessen werden, wie gut der Algorithmus das Objekt wiederfinden kann und wie genau er die Pose der Kamera berechnen kann.Die vom Algorithmus erzeugte Kamerapose muss mit der realen Pose der Kamera verglichen werden, um eine Aussage über ihre Genauigkeit treffen zu können.

Um die reale Kamerapose zu ermitteln, gibt es verschiedene Möglichkeiten. Man kann sie z. B. per Hand für jedes Kamerabild setzen oder sie mit anderen Hilfsmitteln, wie z. B. einem Gyroskop, messen und so ihre Pose berechnen. Für diese Arbeit wurde eine einfachere Methode gewählt. Es wurden um das zu trackende Objekt einfache Marker ausgelegt, die mit Marker-Tracking getrackt wurden. In Abbildung 23 ist ein Beispiel-Kamerabild zu sehen, in dem um ein Objekt, in dem Fall ein Papiermodell von einem Lastwagen, Marker verteilt wurden.

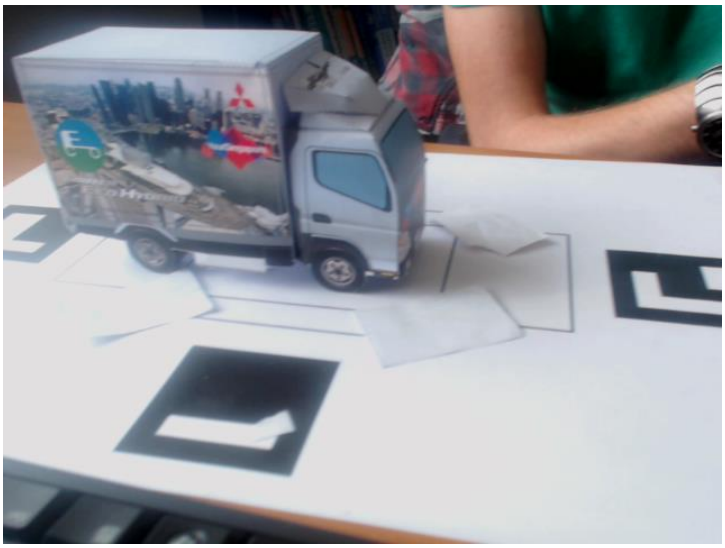


Abbildung 23: Ein Objekt, um das mehrere Marker verteilt wurden. Durch die Marker kann die reale Kamerapose berechnet werden. Von den drei Markern im Bild ist nur der Mittlere für das Markertracking sichtbar, da nur dieser vollständig im Bild liegt.

Marker-Tracking funktioniert nur unter bestimmten Bedingungen, z. B. muss der Marker immer komplett im Kamerabild sichtbar sein. Wurde der Marker vom Tracking gefunden, ist die berechnete Pose aber meist sehr genau. Durch das Verteilen von mehreren Markern wurde die Chance erhöht, dass das Marker-Tracking immer mindestens einen Marker im Kamerabild wie-

derfinden kann.
Im Framework der VisionLib ist bereits ein Marker-Tracking implementiert. Zusätzlich wurde eine Action entwickelt, die die berechneten Kameraposen des Marker-Trackings für jedes Kamerabild in einer Data-Klasse speichert. In einem nächsten Schritt kann diese Pose mit der berechneten Kamerapose eines anderen Tracking-Algorithmus verglichen und so die Abweichung dieses Algorithmus berechnet werden. In Abbildung 24 ist der Ablauf schematisch dargestellt.

Abbildung 24: Pipeline zur Evaluation eines Tracking-Algorithmus. Als Eingabe erhalten beide Tracking-Actions ein Kamerabild. Daraus berechnen beide die extrinsischen Kameraparameter, sprich die Position der Kamera in Relation zum Referenzobjekt. In der `Tracking Evaluation`-Action werden diese beiden Positionen miteinander verglichen und darauf die Abweichung des Merkmal-Trackings im Vergleich zum Marker-Tracking berechnet.

Auf die Sequenz von Kamerabildern wird zunächst das Marker-Tracking angewendet. Die extrahierte Pose wird gespeichert. In der `EvaluationDataPtr`-Klasse wurde für jedes Bild der Sequenz die extrinsischen Kameraparameter als Ground Truth gespeichert. Später ist es außerdem möglich, die Projektionsfehler dort zu speichern.
Danach wird das zu evaluierende Tracking-Verfahren, z. B. das in Abschnitt 3.4 beschriebene, auf die gleiche Bildsequenz angewandt. Nun können in der Action `TrackingEvaluationAction` die berechneten Kameraposen des Tracking-Verfahrens mit der Ground Truth verglichen und daraus der Projektionsfehler berechnet werden.
Der Projektionsfehler wird berechnet, indem ein Menge an definierten 3D-Punkten, am besten Punkte, die auf der Oberfläche des zu trackenden Objektes liegen, mit den extrinsischen Daten des Tracking-Verfahrens und der Ground Truth auf die 2D-Bildebene projiziert werden. Nun kann man zwischen den korrespondierenden 2D-Punkten den Abstand in Pixel messen und daraus den Mittelwert berechnen. Dieser gibt den Projektionsfehler des Tracking-Verfahrens an.

5 Ansätze für 360-Grad-Tracking

In diesem Abschnitt wird zunächst der Ist-Zustand des Tracking-System der VisionLib gezeigt. Dazu wird das Tracking-System mit dem in Abschnitt 4 beschriebenen Verfahren evaluiert und die Ergebnisse werden analysiert.
Anschließend werden verschiedene Verfahren präsentiert, die dabei helfen sollen, das aktuelle Tracking-System so anzupassen, dass es zu 360-Grad-Tracking fähig ist. Jedes der vorgestellten Verfahren wird evaluiert und die Ergebnisse werden vorgestellt.
Am Ende des Abschnitts folgt ein Vergleich der verschiedenen Verfahren sowie die Bilanz.

5.1 Ist-Zustand des VisionLib-Trackings

Hat man aus einer Bildsequenz, die ein Objekt aus allen Blickwinkeln zeigt, ein 360-Grad-Modell erstellt, kann man dieses auf andere Bildsequenzen anwenden. Nachfolgend soll kurz gezeigt werden, wie die Resultate dabei sind und wie sich diese im Bezug auf 360-Grad-Tracking auswirken. Bei der Anwendung des 360-Grad-Modells auf andere Bildsequenzen, in der sich die Kamera 360 Grad um das Objekt herum bewegt, zeigt sich, dass 360-Grad-Tracking allein nur mit einem Modell nicht möglich ist. So kann nicht zu jedem Bild eine Kamerapose berechnet werden, wie in Tabelle 1 zu sehen ist. Mit diesem Verfahren kann bei einer Bildsequenz nur bei weniger als 50% der Bilder eine Kamerapose berechnet werden. Bei den gefunden Kameraposen beträgt der mittlerer Projektionsfehler um 5 Pixel.

Sequenz	**Gefundene Kameraposen (in %)**	**Mittlerer Projektionsfehler (in Pixel, Auflösung 640×480)**
Würfel	35,2	5,59
Lastwagen	48,2	6,68

Tabelle 1: Anzahl der gefunden Kameraposen und mittlerer Projektionsfehler für verschiedene Bildsequenzen. Dabei wurde der Ist-Zustand des Tracking-Systems evaluiert, so wie er zu Beginn der Arbeit bestand.

Man kann erkennen, dass die Ergebnisse bezüglich der gefunden Kameraposen nicht den Erwartungen entsprechen. Die Kamerapose geht verloren und dadurch entstehen Abschnitte, in denen für längere Zeit keine Pose berechnet werden kann. Bei einem Tracking-System, welches für 360-Grad-Tracking ausgelegt ist, sollten die der Prozentsatz an gefunden Kameraposen in Bilder mindestens 80% bis 90% beantragen. Andernfalls entsteht für den Benutzer nicht der Eindruck, dass das System das Objekt von allen Seiten erkennen kann.

5.2 Filtern von Punkten

5.2.1 Idee

Der erste Versuch, die Genauigkeit des Trackings zu verbessern und damit das Tracking verässlicher für 360-Grad-Tracking zu machen, betrifft die 3D-Merkmale des KLT-Modells.
Da bei einem Blick einer Kamera auf ein Objekt nicht jede Seite des Objekts sichtbar sein kann, können von dessen Modell auch nicht alle Merkmale im Kamerabild sichtbar sein. Deshalb wird ein Ansatz entwickelt, der die 3D-Merkmale des KLT-Modells abhängig von der geschätzten Kamerapose filtert.
Die Idee dahinter ist, dass es durch das Verwenden aller Merkmale leichter zu Fehlzuordnungen kommen kann und damit die berechnete Pose verfälscht wird. Durch das Auswählen bestimmter Merkmale, die im Kamerabild der geschätzten Pose sichtbar sind, wurde die Hypothese aufgestellt, dass dies das Tracking stabilisieren kann.
In der VisionLib war bereits die Action `KLTSortByProbabilityAction` implementiert, die Merkmale je nach ihrer Sichtbarkeit gewichtet. In diesem Ansatz sollen die Merkmale jedoch nicht gewichtet werden, sondern gezielt nur die Merkmale ausgewählt und genutzt werden, die sichtbar im Kamerabild sind.
Umgesetzt wird der Ansatz im Framework, indem eine Action erstellt wird, die in der Pipeline den Actions der Auflistung 3 aus Abschnitt 3.3.2 voran gestellt wird. In Auflistung 4 wird die Action mit XML repräsentiert.
Erweitert man die Abbildung 20 mit der `KLTCorrespondencyFilter`-Action, sieht der Ablauf des Trackings wie in Abbildung 25 aus.

5.2.2 Algorithmen

Nachfolgend werden verschiedene Ansätze vorgestellt, die anhand einer gegebenen Kamerapose mittels eines Algorithmus diejenigen 3D-Merkmale auswählen, die für die Kamera sichtbar sind. Als gegebene Kamerapose wird dabei die initiale Pose des KLT-Algorithmus verwendet, dies kann entweder die vom KLT-Algorithmus berechnete Kamerapose vom vorherigem Bild sein oder die vom DOT-Algorithmus geschätze Kamerapose des aktuellen Bild.

Nach Winkel Der erste untersuchte Ansatz stellt eine Beziehung zwischen Kamera und 3D-Merkmalen über Winkel her. Dazu wird für die Position eines jeden 3D-Merkmals der Winkel zur X-Achse auf der X-Ebene bestimmt. Der Winkel berechnet sich aus dem Vektor vom Nullpunkt zum Merkmal der X-Achse. Dies wird auch für die Kameraposition berechnet.

```
<KLTCorrespondencyFilter category="Action" name="
    KLTCorrespondencyFilter" val="KLTCorrespondencyFilter">
        <Keys size="4">
                <key val="World.Room.ExtrinsicData"/>
                <key val="World.Room.KLTModel1"/>
                <key val="World.Room.KLTModel1.
                    AllCorrespondences"/>
                <key val="World.Room.KLTModelFiltered"/>
        </Keys>
        <ActionConfig maxAngle="60" Approach="1"
            zFilterRadius="25" ScalarCutoff="-0.1"/>
</KLTCorrespondencyFilter>
```

Auflistung 4: Action zum Filtern der 3D-KLT-Merkmale in XML. Als Eingabe erhält die Action die extrinsischen Kameraparameter `World.Room.ExtrinsicData` der letzten bekannten Kamera, in der die Position der Kamera gespeichert ist. Außerdem empfängt sie die 3D-Merkmale und ihre Positionen über die Data-Klasse `World.Room.KLTModel1`, in der alle KLT-Merkmale gespeichert sind. Durch Parameter können die einzelnen Filter konfiguriert werden, per Parameter `Approach` können verschiedene Ansätze ausgewählt werden, die ab Abschnitt 5.2.2 beschrieben werden. Das gefilterte KLT-Modell wird mit Hilfe von `World.Room.KLTModelFiltered` weitergegeben.

In Abbildung 26 wird dieses Vorgehen visualisiert.
Sind alle Winkel berechnet, kann für jedes 3D-Merkmal die Differenz zum Winkel der Kameraposition berechnet werden. Anschließend werden nur die Merkmale weitergegeben, die unter einem bestimmten Schwellwert liegen. In Algorithmus 3 wird dieses Vorgehen genauer beschrieben.
In Abbildung 27 wird das Resultat der Filterung visualisiert. Durch die Berechnung der Differenz der Winkel zwischen Kamera und jedem Merkmal wird eine Aussage über das Verhältnis der beiden getroffen. Da der Ursprung des Weltkoordinatensystems dem Koordinatensystem des Objektes entspricht und damit die lokale Position der 3D-Merkmale auch ihre globale Position ist, kann eine zu große Differenz zwischen Kamera und einem Merkmal auch bedeuteten, dass diese auf der Rückseite des Modells liegt und sich somit nicht im Sichtfeld der Kamera befindet.
Eine Alternative zu dem eben beschriebenen Ansatz ist die Hinzunahme des Blickvektors der Kamera. In den extrinsischen Kameraparametern wird nicht nur die Position der Kamera gespeichert, sondern auch die Vektor, in wessen Richtung die Kamera schaut.
Nun kann man zusätzlich noch den Vektor von Kamera zu jedem einzelnen Merkmal bestimmen. Berechnet man den Winkel zwischen den beiden Vektoren, kann man eine Aussage darüber treffen, wie weit jedes KLT-

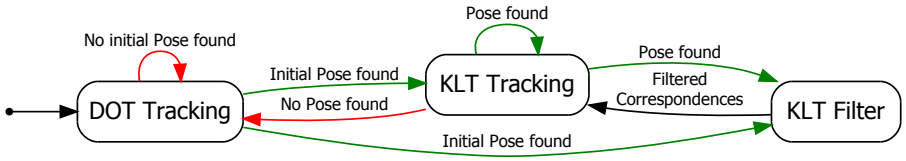


Abbildung 25: Abbildung 20 erweitert mit einer Action zum Filtern von KLT-Merkmalen. Wird eine Pose gefunden, erhält neben der KLT-Action auch die KLT-Filter-Action diese. Nach einem ausgewähltem Algorithmus werden die KLT-Merkmale gefiltert und die gefilterten Merkmale an die KLT-Action weitergegeben. Die KLT-Action arbeitet nur noch mit den gefilterten Merkmalen weiter.

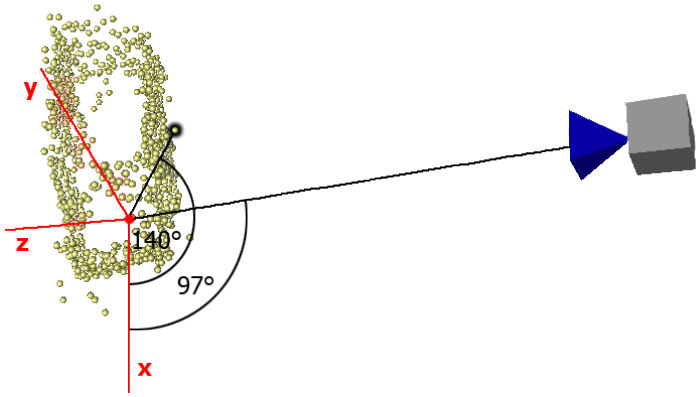


Abbildung 26: Berechnung des Winkels zwischen der Position eines Merkmals und der Kamera mit der X-Achse. Die gelben Punkte entsprechen dabei den 3D-Merkmalen, das Objekt auf der rechten Seite repräsentiert die Kamera und das Koordinatensystem ist rot eingezeichnet.

Merkmal außerhalb des Zentrums des Sichtfeldes der Kamera liegt. Merkmale, deren berechneter Winkel oberhalb eines Schwellwertes liegen, werden anschließend verworfen.
Dies ist genauer als die einfache Betrachtung der Positionen von Kamera und Merkmalen. Der Blickvektor gibt genau an, wo sich das Sichtfeld der Kamera befindet, wohingegen die Position nur angibt, von wo aus die Kamera auf die Szene blickt.

Algorithmus 3 Filtern der KLT-Merkmale anhand ihres Winkels zur X-Achse und der Position der Kamera.

$M \leftarrow KLTModelPtr_{in}$
$C \leftarrow CorrespondencyContainerPtr_{in}$
$E \leftarrow ExtrinsicDataPtr_{in}$
$0 \leftarrow zeropoint$
for all $c \in M$ **do**
  if $angle(c, 0) - angle(E, 0) \leq$ `maxAngle` **then**
    $M_{out} \leftarrow c$
    $C_{out} \leftarrow c$
  end if
end for
$M_{out} \rightarrow KLTModelPtr_{out}$
$C_{out} \rightarrow CorrespondencyContainerPtr_{out}$

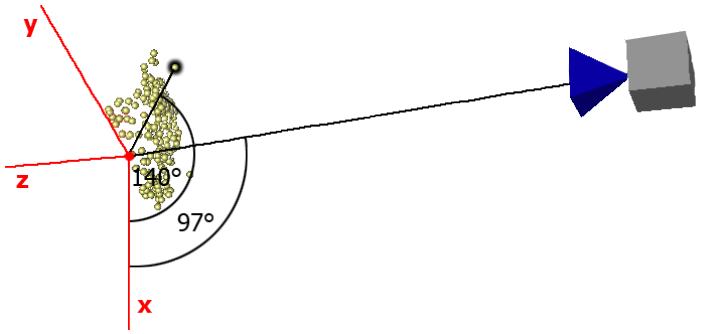


Abbildung 27: Filterung der Merkmale mit einem Schwellwert von $60°$. Ist die Differenz der beiden Winkel größer als der Schwellwert, werden die Merkmale aussortiert. In dieser Abbildung werden nur die Merkmale angezeigt, dessen Winkel unter $60°$ betragen.

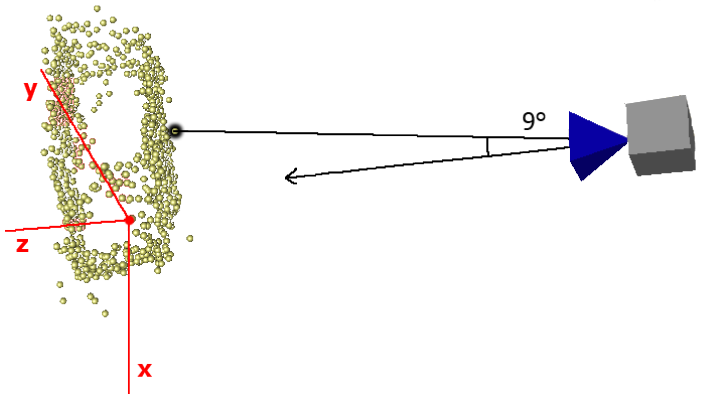
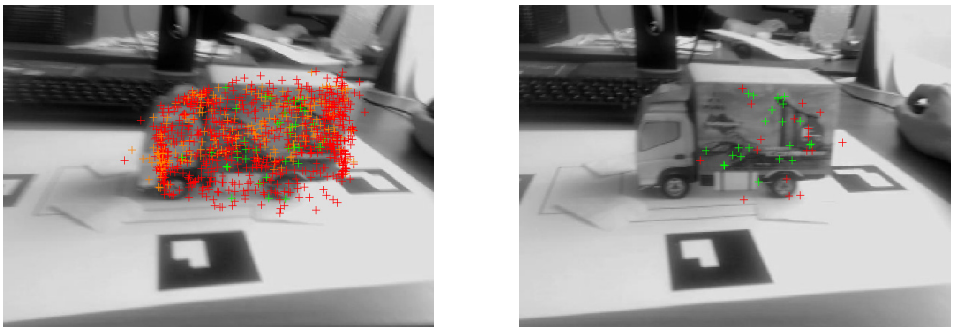


Abbildung 28: Filterung der Merkmale anhand des Blickvektors der Kamera. Es wird der Winkel zwischen dem Blickvektor und dem Vektor, der sich von Kamera zum entsprechenden Merkmal aufspannt, berechnet. Ist dieser größer als ein Schwellwert, wird das Merkmal verworfen.

Nach Status Da jedes KLT-Merkmal einen Status besitzt, der beschreibt, ob das Merkmal im letzten Kamerabild wiedergefunden wurde, wird versucht, dies beim Filtern zu nutzen. Die Idee dahinter ist, dass sich das Kamerabild in einer Sequenz von Bild zu Bild nicht stark verändert und damit auch die Position der Merkmale im Kamerabild sich nicht stark verschieben. Wurde ein KLT-Merkmal im vorherigem Bild gefunden, hat es damit eine hohe Wahrscheinlichkeit, auch in dem jetzigen Kamerabild sichtbar zu sein. Somit werden alle Merkmale gefiltert, die im letzten Bild nicht gefunden wurden. Es wird nur mit den gefunden Merkmalen aus dem letzten Bild weitergearbeitet, wie es in Abbildung 29 dargestellt wird.



(a) (b)

Abbildung 29: Filterung der KLT-Merkmale nach ihrem Status. Die 3D-Merkmale, symbolisiert durch Kreuze, werden mit der vom KLT-Tracking berechneten Projektionsmatrix auf die 2D-Bildebene projiziert. Grüne Kreuze zeigen an, dass eine Übereinstimmung zwischen Modell-Merkmal und Bild-Merkmal gefunden wurde, orange und rote Kreuze bedeuten, dass diese Modell-Merkmale nicht im Kamerabild gefunden wurden. In Abbildung 29(a) sind alle Merkmale des Modells zu sehen, da in diesem Bild die Kamerapose neu initialisiert wurde. Abbildung 29(b) ist das darauf folgende Bild, in dem nur noch die KLT-Merkmale verwendet wurden, die auch im vorherigen Kamerabild (Abbildung 29(a)) gefunden wurden.

Dies soll so lange geschehen, bis nicht mehr ausreichend Merkmale vorhanden sind. Dann sollen in einem Durchlauf, sprich bei einem Kamerabild wieder alle Merkmale dem KLT-Algorithmus zu Verfügung gestellt werden. Findet dieser eine ausreichende Menge an Merkmalen, soll daraufhin nur noch mit dieser Menge gearbeitet werden. Algorithmus 4 zeigt diesen Ablauf schematisch.

Nach Tiefe Im Gegensatz zu dem in Abschnitt 5.2.2 beschriebenen Ansatz mit Winkeln als Maß für die Sichtbarkeit eines Merkmale, kann auch die Entfernung des Merkmale zur Kamera als Maß genommen werden.

Algorithmus 4 Ein Filter, der nur die Merkmale passieren lässt, die im letzten Durchlauf valide waren.

$M \leftarrow KLTModelPtr_{in}$
$C \leftarrow CorrespondencyContainerPtr_{in}$
if $size(C) > Minimum$ **then**
  for all $c \in M$ **do**
    if $state(c) = VALID$ **then**
      $M_{out} \leftarrow c$
      $C_{out} \leftarrow c$
    end if
  end for
else
  $C_{out} \leftarrow C$
end if
$M_{out} \rightarrow KLTModelPtr_{out}$
$C_{out} \rightarrow CorrespondencyContainerPtr_{out}$

Damit soll verhindert werden, dass Merkmale, die auf der Rückseite des Modells liegen, dem Vergleich von KLT-Modell und Kamerabild hinzugezogen werden. In Abbildung 30 sind die gefilterten KLT-3D-Merkmale zu sehen, vergleiche dazu das vollständige KLT-Modell in Abbildung 19(a).
Um die Filterung nach Tiefe zu realisieren, muss zunächst für jedes Merkmal die Entfernung zur Kamera berechnet werden. Ist dies gesehen, werden alle Merkmale mit den extrinsischen Kameraparameter der initialen Pose auf die 2D-Bildebene projiziert. Dann können entweder alle Merkmale mit einer Distanz zur Kamera über einem Schwellwert aussortiert werden oder es kann für jedes Merkmal ermittelt werden, ob sich in der 2D-Nachbarschaft ein weiteres Merkmal befindet, welches eine geringere Distanz zur Kamera hat. Ist dies so, wird das weiter entfernte Merkmal verworfen, weil davon ausgegangen wird, dass es sich auf der Rückseite des Modells befindet und nicht im Sichtfeld der Kamera ist.
Algorithmus 5 zeigt den Ablauf schematisch.

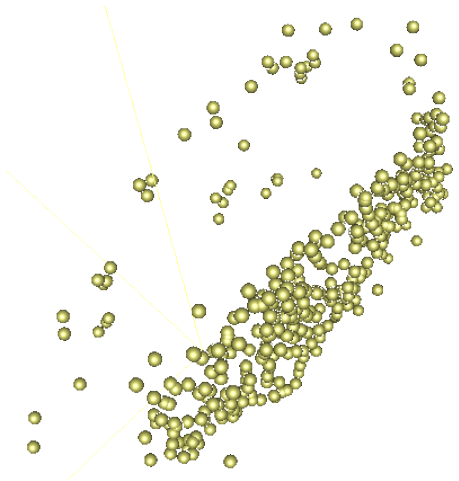


Abbildung 30: KLT-Merkmale, die nach ihrer Tiefe gefiltert wurden. Die Kamera befindet sich außerhalb des Bildes unten rechts. Deshalb sind die Merkmale auf der zur Kamera gerichteten Seite des Modells vorhanden, während die meisten Merkmale auf der Rückseite heraus gefiltert wurden. In Abbildung 19(a) ist das vollständige KLT-Modell zu sehen.

Algorithmus 5 Filtern von KLT-Merkmale nach der Distanz zur Kamera in 3D.

$M \leftarrow KLTModelPtr_{in}$
$C \leftarrow CorrespondencyContainerPtr_{in}$
$E \leftarrow ExtrinsicDataPtr_{in}$
for all $c \in M$ **do**
  $distance3D(c, E)$
  for all $c_{other} \in M : distance2d(c, c_{other}) <$ `zFilterRadius` **do**
    if $zValue(c) > zValue(c_{other})$ **then**
      $skip\ c$
    end if
  end for
  $M_{out} \leftarrow c$
  $C_{out} \leftarrow c$
end for
$M_{out} \rightarrow KLTModelPtr_{out}$
$C_{out} \rightarrow CorrespondencyContainerPtr_{out}$

Nach Blickvektor der Merkmale Ein Bild-Merkmal kann je nach Blickwinkel der Kamera für den Algorithmus verschieden aussehen. Da bei der in Abschnitt 3.3.2 beschriebenen Erstellung des Kameramodells anschließend für jedes Eingabebild die Kamerapose berechnet werden kann, kann für jedes Merkmal gespeichert werden, wie die Kamera zum Modell stand, als das Bild aufgenommen wurde.In Abbildung 19(b) sind die Deskriptoren so eingezeichnet, dass sie orthogonal zu der Kamerapose stehen, die zum Bild gehört, aus dem das Merkmal extrahiert wurde.
Will man diese Information zum Filtern nutzen, ist ein Ansatz, die geschätzte Kamerapose als Vergleich zu nehmen. Es werden nur die Merkmale als gültig angesehen, deren bei der Erstellung des Modells gespeicherte Kamerapose der geschätzten Kamerapose nach einem bestimmten Maß gleicht.
Algorithmus 6 stellt den eben beschriebenen Filtervorgang dar.

Algorithmus 6 Das Filtern der KLT-Merkmale anhand ihrer gespeicherten Blickvektoren. Wird die Methode `lookAt` mit einem Merkmal als Eingabe aufgerufen, wird der Blickvektor der Kamera zurückgegeben, mit der das Merkmal aufgenommen wurde.

$M \leftarrow KLTModelPtr_{in}$
$C \leftarrow CorrespondencyContainerPtr_{in}$
$E \leftarrow ExtrinsicDataPtr_{in}$
for all $c \in M$ **do**
  if $lookAt(c) * lookAt(E) \leq$ `ScalarCutoff` **then**
    $M_{out} \leftarrow c$
    $C_{out} \leftarrow c$
  end if
end for
$M_{out} \rightarrow KLTModelPtr_{out}$
$C_{out} \rightarrow CorrespondencyContainerPtr_{out}$

Über eine Metrik kann definiert werden, wann sich Kamerapose des Merkmals und die des aktuellen Bildes nicht mehr gleichen. In Algorithmus 6 beispielsweise wurde das Skalarprodukt aus den Blickvektoren der Kamera genommen. Ist der resultierende Winkel über einem bestimmten Schwellwert, wird das Merkmal nicht übernommen. In Abbildung 31 ist ein so gefiltertes KLT-Modell zu sehen.
In Algorithmus 6 fällt auf, dass nur der reine Blickvektor verglichen wird, ohne die Position der Kamera zu beachten. Dies ist valide, da angenommen werden kann, dass man sich beim 3D-Objekt-Tracking für Augmented Reality auf einer Halbkugel-förmigen Bahn/Ebene bewegt. Dadurch ist die Wahrscheinlichkeit groß, dass, wenn der Blickvektor ungefähr gleich ist, auch die Position ähnlich sein muss.

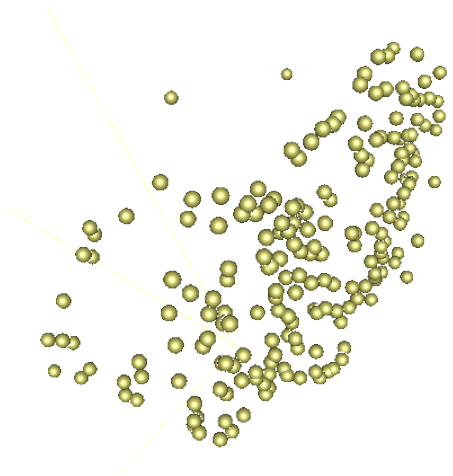


Abbildung 31: KLT-Merkmale nach ihrem Blickvektor gefiltert. Es wird der Blickvektor der Schätzung der Kamerapose mit dem des Merkmals aus der Erstellung des Modells verglichen. Sind diese beiden Vektoren nicht ähnlich genug, fällt das Merkmal für die Poseberechnung heraus.

Im Vergleich zur Filterung nach Tiefe kann man sehen, dass bei der Filterung nach dem Blickvektor mehr Merkmale herausfallen (siehe dazu Abbildung 30 und 31). Dies geschieht, da nun mehr gewichtet wird, dass die Merkmale aus Modell und aktuellem Kamerabild aus dem ungefähr gleichem Blickwinkel aufgenommen werden. Dies ist ein stärkeres Ausschlusskriterium als die Tiefe, da hier mit absoluten Werten gearbeitet wird und nicht mit einer Relation von Werten, wie bei der Tiefe.

5.2.3 Evaluation

Nachfolgend werden Ergebnisse der in Abschnitt 5.2.2 beschriebenen Algorithmen präsentiert und besprochen.
Dafür wurden verschiedene Bildsequenzen aufgenommen, in denen ein vorher definiertes Objekt zu sehen ist. Die in Abschnitt 3.4 beschriebene Tracking-Pipeline wurde auf diese Bildsequenzen angewendet mit Einbeziehung der Filter-Algorithmen. Die Resultate wurden mit einer Ground Truth verglichen, um eine Aussage über die Genauigkeit der Ergebnisse zu treffen. Das Vorgehen zur Erstellung der Ground Truth ist in Abschnitt 4 beschrieben.
In Abbildung 32 sind die Ergebnisse der verschiedenen Filter zu sehen. Sie wurden angewendet auf die Bildsequenz „Lastwagen“ , dieähnlich zu der in Abbildung 22 gezeigten Sequenz ist. Es galt, das Objekt aus Abbildung 21 in den Bildern zu finden. Die Ergebnisse zu der Sequenz „Würfel“ sind im Anhang zu finden. Zu sehen ist in der Abbildung 32 der Projektionsfeh-

ler für jeden Filter pro Bild der Sequenz. Zur Verbesserung der Übersichtlichkeit werden nur die Projektionsfehler zu jedem fünften Bild angezeigt. Ist für ein Bild keine Markierung des entsprechenden Filters vorhanden, konnte das Tracking mit diesem Filter nicht das Objekt im Bild wiederfinden und keine valide Kamerapose berechnen. Die in der Legende angegebenen Filter-Namen entsprechen folgenden Filter-Ansätzen:

Kein Filter Es wurde kein Filter auf die KLT-Merkmale angewendet.

Winkel 1 Filter Der zuerst beschriebene Filter aus Abschnitt „Nach Winkel“ wurde angewendet, mit dem die Merkmale nach ihrer Position zur Kamera gefiltert werden.

Winkel 2 Filter Der zweite beschriebene Filter aus Abschnitt „Nach Winkel“ wurde angewendet.

Status Filter Der in Abschnitt „Nach Status“ beschriebene Filter wurde angewendet, der Merkmale nach ihrem Status auswählt.

Tiefen Filter Es wurde der Filter angewendet, der die Merkmale nach ihrer Tiefe filtert, beschrieben in Abschnitt „Nach Tiefe“.

Blickvektor Filter Die Merkmale wurden mit dem Filter aus Abschnitt „Nach Blickvektor der Merkmale“ gefiltert, bei dem der Blickvektor eine Rolle spielt.

Man kann in Abbildung 32 erkennen, dass es zwei größere Lücken in der Bildsequenz gibt, bei der das Tracking mit und ohne Filter keine Kamerapose berechnen konnte. Diese entsprechen den Abschnitten, in denen der Lastwagen von der Vorder- und Rückseite gefilmt wurde. Dort sind nicht genug Details für das KLT-Tracking vorhanden, sodass nicht genügend Korrespondenzen zwischen den Merkmale gezogen werden können, um eine Kamerapose zu berechnen. Die Unterschiede, wenn bei allen eine Kamerapose gefunden wurde, sind außerdem nicht sonderlich groß.
In Tabelle 2 ist die Anzahl der gefunden Kameraposen in der Sequenz „Lastwagen“ und der durchschnittliche Projektionsfehler für jeden Filter aufgelistet. Man kann sehen, dass der Mittelwert des Projektionsfehlers nicht merklich mit einem Filter verbessert werden kann. Er fällt sogar um bis zu einem Pixel pro Filter. Beim Status-Filter werden sogar noch schlechtere Werte erzielt. Es ist also nicht möglich, die Stabilität des KLT-Algorithmus mit einer Filterung der Merkmale zu verbessern. Einzig der Ansatz des Filterns der Merkmale nach ihrem Winkel zur Kamera und ihrer Ausrichtung zum Blickvektors verbessern leicht das Tracking bei der Sequenz Würfel, die Tabelle 6 dazu ist im Anhang zu finden.
Die Anzahl der gefunden Kameraposen in den Bildern der Sequenz lässt sich durch die Filter auch nicht deutlich steigern. Bei der Sequenz „Lastwagen“ ist bei keinem Filter ein ein Anstieg des Prozentsatz zu sehen, in

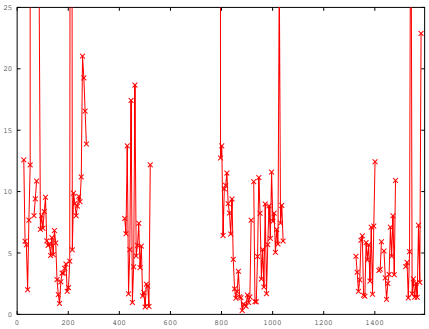
	Gefundene Kameraposen (in %)	**Mittlerer Projektionsfehler (in Pixel, Auflösung 640×480)**
Kein Filter	50,3	5,85
Winkel 1 Filter	46,5	6,69
Winkel 2 Filter	45,6	5,96
Status Filter	28,8	9,48
Tiefen Filter	45,2	6,14
Blickvektor Filter	47,5	6,47

Tabelle 2: Anzahl der gefunden Kameraposen und mittlerer Projektionsfehler bei der Bildsequenz „Lastwagen“ für die verschiedenen Filteransätze.

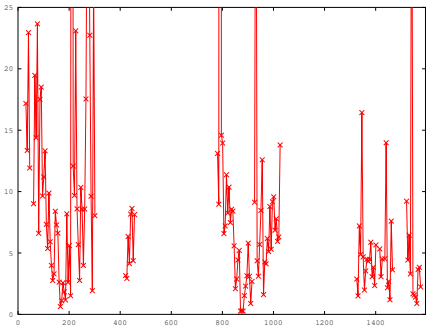
der Sequenz „Würfel“ hebt sich sie Anzahl an gefunden Kameraposen um 10% bei den Filtern, die mit Winkeln zur Kamera und dem Blickvektor arbeiten an. Es ist eine Steigerung von ungefähr 30% auf 40% und damit nicht ausreichend hoch.
In Abbildung 33 sind die Histogramme zu sehen, die die Verteilung der Projektionsfehler für jeden Filtern anzeigen. Bei Vergleich der Verteilung der Filter zeigen sich keine großen Unterschiede zur Verteilungen ohne Filter. Einzig der Status Filter hat einen deutlichen Abfall der Werte. Alle anderen Verteilungen gleichen sich, was man auch daran sieht, dass der durchschnittliche Projektionsfehler sich nicht stark unterscheidet. Das Filtern der Merkmale hat folglich keinen großen Einfluss auf die Stabilität des Trackings. Im Anhang sind die hier präsentierten Diagramme nochmals in einer höheren zu finden.

5.2.4 Fazit

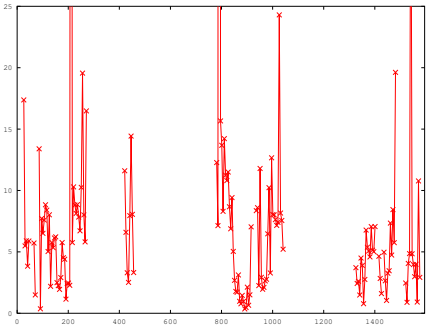
Zusammenfassend lässt sich sagen, dass die Stabilität des KLT-Algorithmus durch das Filtern der Merkmale nicht verbessert werden kann. Der KLT-Algorithmus nutzt zum Berechnen der Kamerapose bereits ein ausreichendes Bewertungssystem, welches ein vorheriges Filtern der Merkmale überflüssig macht. Jedoch konnte mit dem Selektieren von Merkmalen die Zeit verkürzt werden, die der KLT-Algorithmus zur Berechnung der Pose benötigt. Der Leistungsgewinn beträgt dabei um die 5%.
Durch die kleinen Verbesserungen in der Berechnung der Kamerapose konnte gezeigt werden, dass der Ansatz des Filterns von Merkmalen Potential bietet, z. B. bei einem Algorithmus, der kein oder kein gutes Verfahren zum Bewerten der Qualität der Merkmale besitzt.



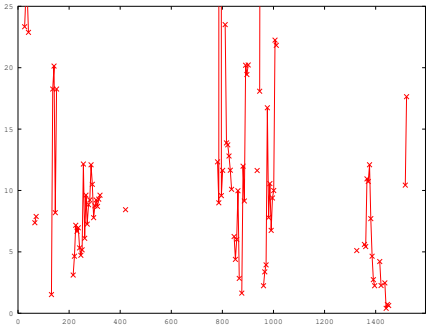
(a) Kein Filter



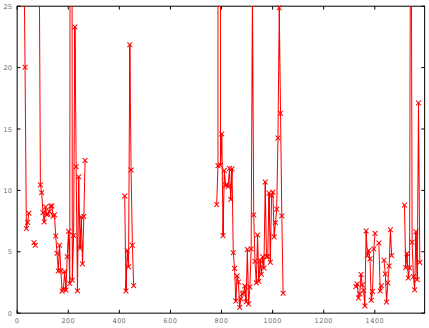
(b) Winkel 1 Filter



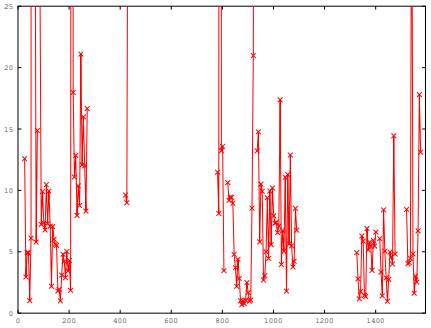
(c) Winkel 2 Filter



(d) Status Filter

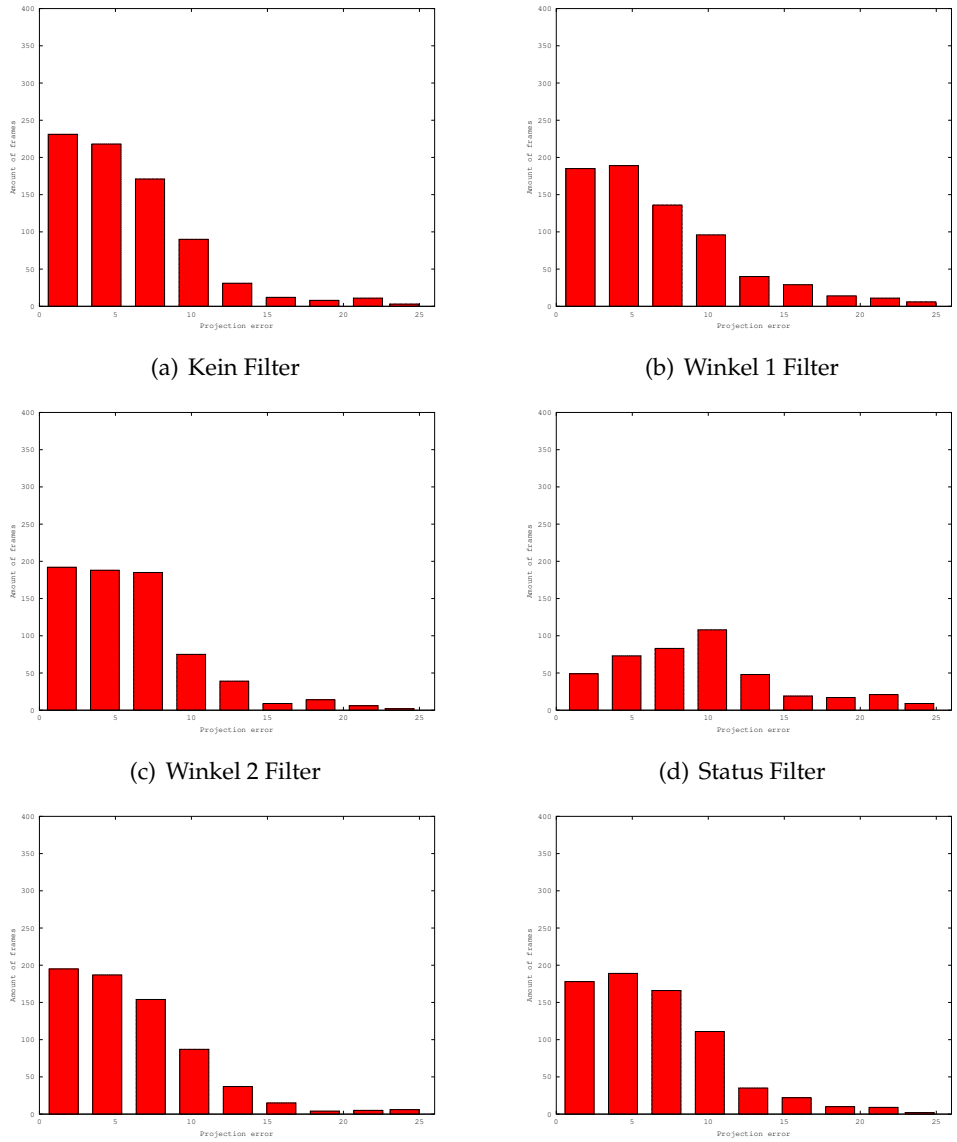


(e) Tiefen Filter



(f) Blickvektor Filter

Abbildung 32: Projektionsfehler für die Bildsequenz „Lastwagen“. Es wird der Projektionsfehler angezeigt für jedes Bild der Sequenz. Dabei entspricht die X-Achse dem Verlauf der Sequenz und die Y-Achse der Höhe des Projektionsfehlers. Jedes Diagramm entspricht dabei einem Filter-Ansatz. In Abschnitt 5.2.3 ist aufgelistet, welcher Name zu welchem Filter-Ansatz gehört.



(a) Kein Filter

(b) Winkel 1 Filter

(c) Winkel 2 Filter

(d) Status Filter

(e) Tiefen Filter

(f) Blickvektor Filter

Abbildung 33: Histogramm des Projektionsfehlers. Auf der X-Achse wird der Projektionsfehler in Pixel angezeigt, auf der Y-Achse wie viele Bilder einen Projektionsfehler in diesem Bereich haben. Dabei wurde der Wertebereich der X-Achse in Klassen eingeteilt. Die Größe jeder Klasse ist drei Pixel. Die letzte Klasse, sprich die rechte, zeigt an, wie viele Bilder eine berechnete Kamerapose besaßen, die größer gleich als der Wert 25 waren. Jedes Diagramm entspricht dabei einem Filter-Ansatz. In Abschnitt 5.2.3 ist aufgelistet, welcher Name zu welchem Filter-Ansatz gehört.

5.3 Vorhersage der Pose

5.3.1 Idee

Nachdem das Filtern der Merkmale für das KLT-Tracking keine deutliche Verbesserung brachte, wurde als Nächstes die Vorhersagen der Kamerapose untersucht. Hierbei betrachtet man die Situation, wenn der KLT-Algorithmus aus dem aktuellen Kamerabild keine Kamerapose berechnen kann. In diesem Fall soll ein anderes Verfahren aus den Kameraposen der letzten Bilder eine Vorhersage der aktuellen Pose treffen können. Dies soll dabei helfen, Bilder zu überbrücken, bei denen KLT keine Kamerapose berechnen kann, z. B. wenn ein zu starkes Rauschen im Bild ist. Dadurch wird vermieden, dass für den Betrachter der Eindruck entsteht, dass das Tracking ausgefallen sei.
In Abbildung 34 ist zu sehen, wie sich eine Action zur Pose-Schätzung, in diesem Fall der Kalman-Filter, in die Pipeline des Trackings integriert. Dabei ist zu beachten, dass die Pose-Schätzung nur dann funktionieren kann, wenn genügend Kamerapose zuvor berechnet werden konnten. Auch muss sicher gestellt sein, dass, wenn das KLT-Tracking trotz Pose-Schätzung nicht das Objekt im Kamerabild wiederfinden kann, zum DOT-Tracking zurück gekehrt wird, um den Initialisierungsprozess neu zu starten.

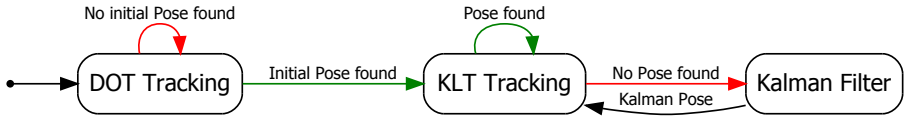


Abbildung 34: Aufbau der Tracking-Pipeline mit einem Kalman-Filter. Hier wird das Tracking aus Abbildung 20 mit einem Kalman-Filter erweitert. Der Anfang entspricht dem ursprünglichem Ablauf. Bis eine Pose gefunden werden kann, kommt das DOT-Tracking zum Einsatz. Die gefundene Pose wird dem KLT-Tracking übergeben. Solange das KLT-Tracking eine Pose berechnen kann, wird diese im nächsten Bild wieder als initiale Pose verwendet. Wird allerdings keine Pose im Bild gefunden, kommt nicht wieder das DOT-Tracking zum Einsatz, sondern der Kalman-Filter übernimmt und versucht aus den zuletzt gefundenen Posen eine Vorhersage zu treffen, wie die Kamerapose im aktuellen Kamerabild sein könnte. Diese Schätzung wird dem KLT-Tracking als initale Pose übergeben und das KLT-Tracking versucht, damit weiter zu arbeiten.

5.3.2 Ansätze

Nachfolgend werden verschiedene Ansätze vorgestellt, mit denen die Kamerapose geschätzt werden soll. Die Idee wurde zunächst in Octave um-

gesetzt, um evaluieren zu können, welche Ansätze das größte Potenzial besitzen. Dazu wurde die Positionen einer Kamera bei der Bewegung um ein Objekt herum aufgezeichnet und die Daten auf eine 2D-Ebene projiziert. Auf diese Daten wurden anschließend die verschiedenen Ansätze angewandt.

Kalman-Filter Der Kalman-Filter ist ein nach Rudolf Emil Kalman benanntes Verfahren zum Filtern von Messrauschen [18]. Das Resultat des Verfahrens ist eine optimale Schätzung eines Zustandes von einem beobachtetem System. Da das Verfahren iterativ arbeitet, müssen nicht alle Messungen gespeichert werden, sondern können direkt verarbeitet werden. Siehe dazu Abbildung 35. Um den Kalman-Filter im Tracking-Kontext zu

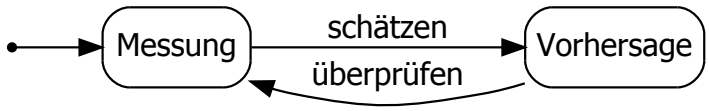


Abbildung 35: Vereinfachte Darstellung des Kalman-Filters. Zunächst wird aus dem aktuellen Zustand eine Vorhersage berechnet, wie sich das System verändern wird. Danach wird eine Messung durchgeführt und damit die Vorhersage überprüft.

nutzen, ist die Idee, dass wenn das Tracking keine Kamerapose findet, die Schätzung vom Kalman-Filter verwendet werden kann. Nachfolgend wird kurz die Funktionsweise des Kalman-Filters vorgestellt.
Als Erstes wird eine Matrix benötigt, die den Zustand des Systems beschreibt. Für den 2D-Raum ist dies

$$x = \begin{bmatrix} x \\ y \\ \dot{x} \\ \dot{y} \end{bmatrix}, \tag{9}$$

bei der x und y für die Position im Raum steht und $\dot{x}$ und $\dot{y}$ für die jeweilige Geschwindigkeit anzeigt. Zusätzlich muss die Unsicherheit des Zustandes repräsentiert werden. Dies geschieht mit der Matrix

$$P = \begin{bmatrix} t & 0 & 0 & 0 \\ 0 & t & 0 & 0 \\ 0 & 0 & t & 0 \\ 0 & 0 & 0 & t \end{bmatrix}. \tag{10}$$

Der Wert t gibt die Unsicherheit des initialen Zustandes an und die Matrix P verändert sich zur Laufzeit. Die Matrix A gibt an, wie sich die Matrix x

beim Zustandsübergang dt verhält mit

$$x = A \cdot x = \begin{bmatrix} 1 & 0 & dt & 0 \\ 0 & 1 & 0 & dt \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ \dot{x} \\ \dot{y} \end{bmatrix}. \tag{11}$$

Die Messung wird mit der Matrix $H = \left[\begin{smallmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{smallmatrix}\right]$ angegeben. Da wir vom Tracking die absolute Position im Raum erhalten und nicht z. B. den Geschwindigkeitsvektor, wird mit der Matrix H festgelegt, dass es sich bei einer Messung um die Position handelt. Zusätzlich wird eine Matrix R benötigt, die das Rauschen der Messung beschreibt und die Prozess-Rausch-Kovarianzmatrix Q. Diese gibt an, wie das System durch äußere Einflüsse gestört werden kann.
Hat der Kalman-Filter einen Zustand, wird im ersten Schritt mit der Gleichung 11 eine Vorhersage getroffen, wo sich das System im nächsten Schritt befinden könnte. Danach wird die Unsicherheit des Zustandes neu berechnet mit $P = A \cdot P \cdot A' + Q$. Die Unsicherheit wird größer, da nur eine Vorhersage getroffen wird.
Ist für den nächsten Schritt die Messung Z eingetroffen, kann die Vorhersage überprüft werden. Dies geschieht, indem die Messwertdifferenz $w = Z - (H \cdot x)$ zwischen Vorhersage und Messung berechnet wird. Danach wird berechnet, wie hoch die Kovarianz $S = (H \cdot P \cdot H' + R)$ ist. Diese gibt an, wie stark die Messunsicherheiten im System sind. Damit lässt sich das Kalman-Gain $K = \frac{P \cdot H'}{S}$ bestimmen, welches angibt, ob dem Messwert oder Vorhersage mehr vertraut werden soll.Wird K kleiner, entspricht die Messung der Vorhersage, wird K größer, unterscheiden sich Vorhersage und Messung. Mit K lassen sich anschließend der Systemzustand $x = x + (K \cdot w)$ und seine Kovarianz $P = \big(I - (K \cdot H)\big)$ berechnen. x ist damit der neue Zustand, an dem sich das System befindet. P gibt an, wie genau dieser Zustand ist.
In VisionLib ist der Kalman-Filter bereits implementiert. Als Eingabe erhält die Action die extrinsischen Kameraparameter und liefert als Ergebnisse extrinsischen Kameraparameter zuürck, wobei diese die vorhergesagte Kamerapose des Kalman-Filters enthält. In Abbildung 36 ist der Kalman-Filter auf ein Beispiel-Kamerapfad in 2D angewendet.
Man kann sehen, dass die Vorhersagen des Kalman-Filters z. T. schon sehr stark von dem richtigen Kamerapfad abweichen. Dies kommt durch die Linearität des Kalman-Filters. Dieser geht bei seinen Berechnungen von einem linearen Model aus. Die Bewegung einer Kamera um ein Objekt herum ist aber keine lineare Bewegung. Daher hat der Kalman-Filter besonders dann Probleme, wenn die Kamera abrupt ihre Richtung ändert. So sind die Vorhersagen am Anfang des Pfades noch recht genau, sobald sich die Kamera aber nicht mehr gerade aus bewegt, sondern sich in eine andere Richtung bewegt, weicht die Vorhersage stark von der richtigen Position

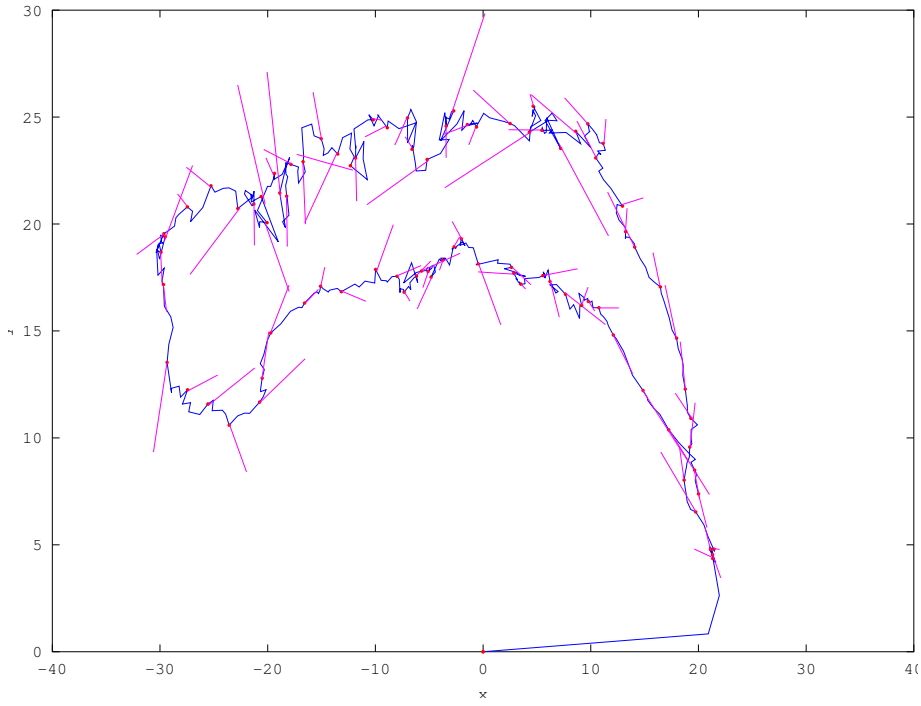


Abbildung 36: Anwendung des Kalman-Filters auf einen Kamerapfad. Die blaue Linie repräsentiert einen Pfad, den die Kamera entlang bewegt wurde. Die Kamera startet im Nullpunkt. An mehreren Punkten wurde der Kalman-Filter auf diese Messungen angewendet. Mit einem roten Punkt sind die Stellen markiert, an denen dem Kalman-Filter nicht mehr die Messdaten zu Verfügung standen und er nur noch mit der Vorhersage arbeiten konnte. Magenta eingezeichnet ist der Pfad, den der Kalman-Filter für die Kamera berechnet hat.

ab, weil der Kalman-Filter das System weiter in die gleiche Richtung bewegt.
Da bei der Bewegung um ein Objekt die Kamera oft ihre Richtung wechselt, stellt sich heraus, dass der Kalman-Filter nicht für die Vorhersage einer Position der Kamera zur Initialisierung des KLT-Trackings geeignet ist.

Ellipsen-Fitting Ein weiterer Ansatz der im Rahmen dieser Arbeit untersucht wurde, ist das Ellipsen-Fitting. Wie man in Abbildung 36 sehen kann, verlaufen die Bewegungen der Kamera um das Objekt, welches sich im Punkt $(0, 0)$ befindet, herum kreisförmig. Daher ist die Idee, in eine bestimmten Anzahl von Kamerapositionen eine Ellipse einzupassen, sodass der Pfad der Kamera durch diese Ellipse vorher gesagt werden kann. Die Ellipse wird aus einer Menge an Punkten erzeugt, indem die Summe des

quadratischen Abstandes der Punkte zur Ellipse minimiert wird. Auf der erzeugten Ellipse wird die Stelle gesucht, die am dichtesten zur letzten validen Kameraposition ist. Von dieser Position aus kann man die nächste Kameraposition schätzen, indem man sich mit einer bestimmten Geschwindigkeit auf der Ellipse weiter bewegt. In Abbildung 37 wird dieser Algorithmus auf den Kamerapfad angewendet.

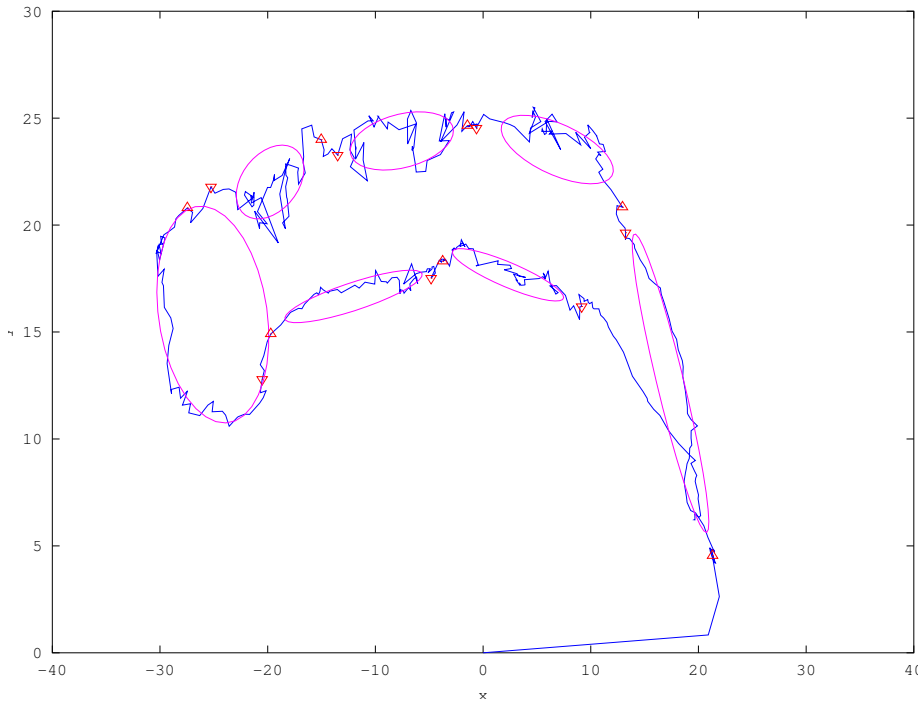


Abbildung 37: Aus einer Menge von Kamerapositionen werden Ellipsen erzeugt. Blau eingezeichnet ist der Pfad der Kamera. Jeweils zwei rote Dreiecke grenzen die Menge an Kamerapositionen ein, die für die jeweilige Berechnung für eine Ellipse verwendet wurden. Magenta eingezeichnet ist die resultierende Ellipse.

Wie in Abbildung 37 zu sehen ist, funktioniert das Ellipsen-Fiiting nicht wie gewünscht. Die Ellipsen werden zu klein. Durch das starke Zittern der Kameraposition wird der Bewegungsfluss der Kamera gestört und kleinere Ellipsen erfüllen eher die Bedingung des Algorithmus als große Ellipsen, mit der man die Bewegung der Kamera fortführen könnte.

Linien-Schätzung Um die starken Schwankungen in der Kamerabewegung zu kompensieren und eine ungefähre Ahnung zu haben, in welche Richtung sich die Kamera bewegt, ist die Idee entstanden, aus einer Anzahl von Punkten eine Linie zu erstellen. Auch hier wird versucht, den Fehler $e = \sum_i (y_i - \dot{y}_i)$ zwischen der Menge an beobachteten Punkten

y_i und der geschätzten Linie $\dot{y}_i$ zu minimieren. Daraus lässt sich die Geradengleichung $\dot{y} = mx + c$ aufstellen, bei der die Steigung m und die Konstante c berechnet werden muss. Mit der *Methode der kleinsten Quadrate* $\begin{bmatrix} m \\ c \end{bmatrix} = (X^T X)^{-1} X^T Y$ lässt sich die Gleichung nach m und c auflösen, wobei Y der Menge der gegebenen Punkte entspricht und X eine Menge an Koeffizienten ist.

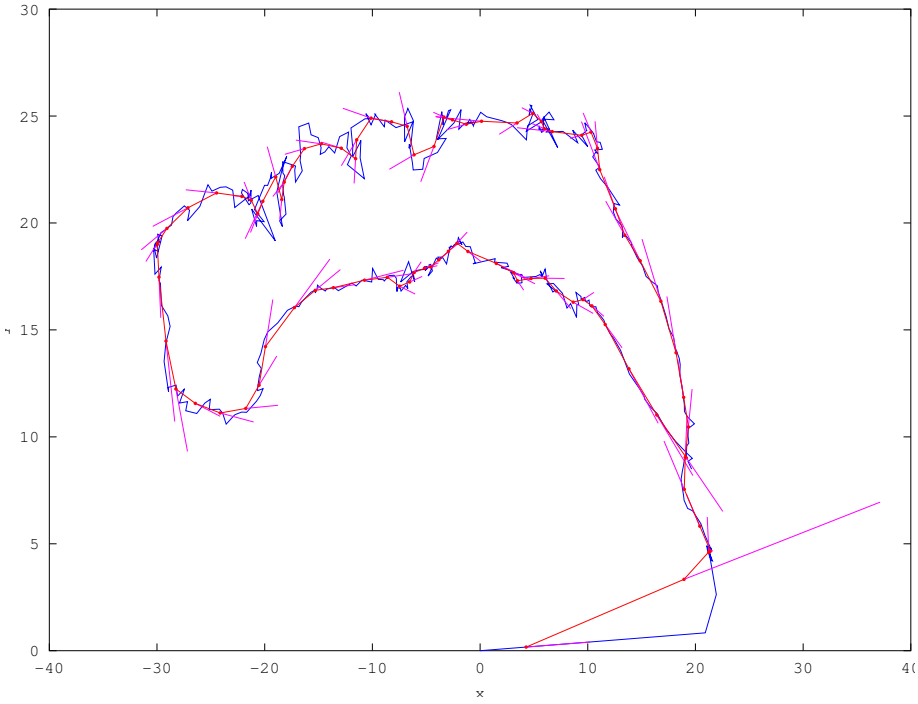


Abbildung 38: Vorhersage der Kamerabewegung mittels Linien-Schätzung. Die blaue Linie entspricht der tatsächlichen Kamerabewegung. Der Start der Bewegung ist im Ursprung. Die roten Punkte markieren den Punkt, an dem die Schätzung der Bewegung beginnt. Dafür wird die Menge an Kamerapostionen zwischen den zwei letzten roten Punkten verwendet und damit mittels des in Abschnitt 5.3.2 vorgestelltem Verfahren eine Linie erzeugt. Magenta eingezeichnet ist diese Linie fortgeführt, also die Vorhersage nach dem Verfahren.

Wie man in Abbildung 38 sehen kann, kommen die Vorhersagen mit dieser Methode näher an der tatsächlichen Kamerabewegung heran. Da dieses Verfahren mit Geraden arbeitet, besteht hier wie beim Kalman-Filter das Problem der Linearität. So lange sich die Kamera ungefähr in eine Richtung bewegt, funktioniert die Vorhersage, schlägt die Kamera allerdings eine andere Richtung ein, kann das Verfahren damit nicht umgehen.

Kurven/Funktions-Fitting Um die Linearität aus der Vorhersage zu entfernen, war es ein weiterer Ansatz, anstatt Linien in eine Menge von Punkte einzupassen, Kurven zu verwenden. Dazu wurde die Methode `polyfit` [7] in Octave verwendet, die Polynome eines bestimmten Ranges in eine Menge an Punkte einpasst, sodass die Summe der Distanz von den Punkten zur Kurve minimal ist.

Wie in Abbildung 39 zu sehen ist, funktioniert das Vorhersagen der Kame-

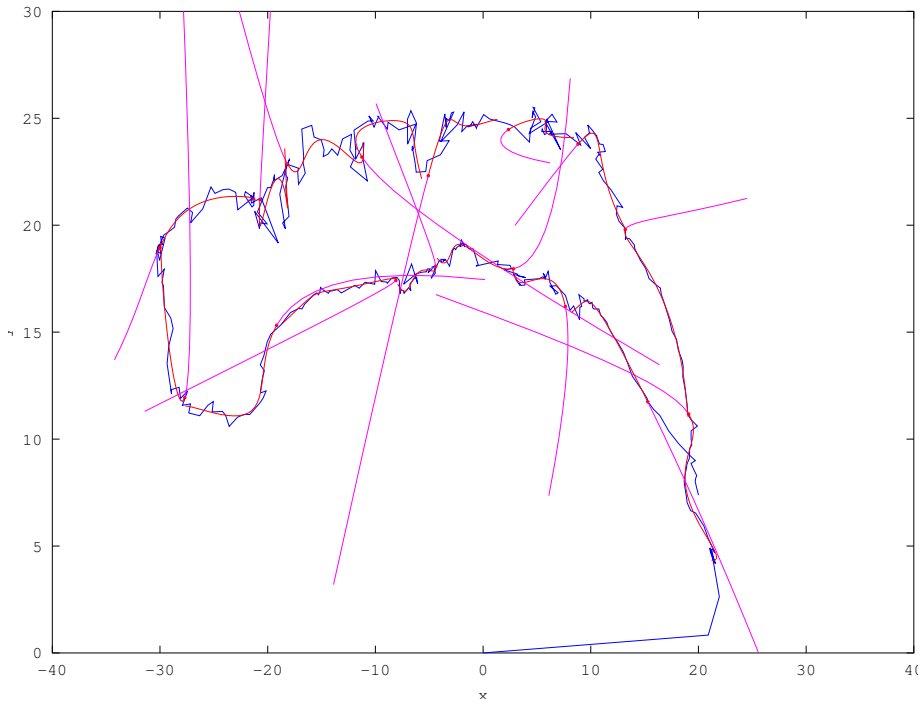


Abbildung 39: Vorhersage der Kamerabewegung mit Polynomen. Bei dieser Berechnung wurde mit Polynomen des Rangs 4 gerechnet. Blau markiert ist der tatsächliche Pfad der Kamera. Die Kamera startet im Punkt $(0, 0)$. Die roten Pfade zeigen die Kurve an, die aus den jeweilige Kamerapositionen mittels Kurven-Fitting errechnet wurden. Der rote Punkt markiert jeweils den Beginn der Vorhersage. Die Magenta-farbigen Kurven sind die Vorhersagen basierend auf den berechneten Polynomen.

rabewegung mit dem Einpassen von Kurven nicht wie gewünscht. Durch den hohen Rang der Polynome steigen die Kurven exponentiell an und geraten damit „außer Kontrolle“. Die Kurven gehen z. T. in komplett andere Richtungen, als die Kamera sich eigentlich bewegt. Damit kann man diesen Ansatz auch nicht effektiv zur Vorhersage der Bewegung der Kamera einsetzen.

5.3.3 Fazit

Wie man in den vorherigen Abbildungen erkennen kann, war es nicht möglich, mit einem Ansatz das gewünschte Ergebnis zu erzielen. Durch das starke Zittern in der Kamerabewegung und das nicht-lineare Verhalten ist es schwer, mathematisch die Bewegung der Kamera vorher zu sagen, sodass die vorhergesagte Pose als initiale Pose für das KLT-Tracking verwendet werden kann.

5.4 Online Template-Erstellung

5.4.1 Idee

Da das Vorhersagen der Kamerabewegung bei Verlust des KLT-Tracking-Objektes nicht die gewünschten Resultate erbringen konnte, wurde ein anderer Ansatz entwickelt, um das KLT-Tracking gegenüber dem 360-Grad-Tracking stabiler zu machen. Die Idee dahinter ist, es zu ermöglichen, dass das KLT-Tracking schneller das gesuchte Objekt wiederfinden kann. Dazu muss das DOT-Tracking verbessert werden, indem die initiale Pose noch schneller gefunden werden kann.
Das DOT-Tracking gehört zur Gruppe der Tracking-Algorithmen, die mit Bildvergleichen arbeiten. Wie in Abschnitt 3.2.3 hat das DOT-Tracking ein Set aus Bildern mit dazugehörigen Kamerapositionen. Das aktuelle Kamerabild wird mit dem Set verglichen und die Kamerapose als aktuelle Pose ausgewählt, deren Bild am besten dem Kamerabild gleicht. Durch die Effizienz des Algorithmus kann das DOT-Tracking viele Bilder schnell verarbeiten. Allerdings ist das DOT-Tracking auf das Set an Bildern angewiesen. Enthält das Set des DOT-Trackings z. B. keine Bilder von der Rückseite eines Modells, kann es von der Rückseite des Objekts auch keine Kamerapose berechnen. Auch kann der Hintergrund des Bildes Fehler bei den Bildvergleichen erzeugen. Gleicht sich der Hintergrund beider Bilder zu stark, aber das eigentliche Objekt ist gar nicht im Bild vorhanden, kann das DOT-Tracking trotzdem dieses Bild als besten Vergleich ansehen, da die Hintergründe zu stark übereinstimmen. Genau so kann der Hintergrund den Bildvergleich aber auch stören, wenn es starke Unterschiede gibt. Deshalb ist es von Vorteil, wenn man so aktuelle Bilder wie möglich vom Objekt hätte.
Genau hier setzt dieser Ansatz an. Es sollen dem DOT-Tracker zur Laufzeit neue Bilder des Objektes hinzugefügt werden, damit er das Objekt schneller und besser wiederfinden kann, wenn das KLT-Tracking dazu nicht mehr in der Lage ist. So ist ein schnelleres Reinitialisieren des KLT-Trackings möglich. Um neue Bilder dem Set von Bildern des DOT-Trackings zuzuführen, werden die Bilder verwendet, bei der das KLT-Tracking eine Kamerapose berechnen konnte, siehe dazu Abbildung 40.

5.4.2 Umsetzung

Die Erstellung eines neuen Vergleichsbilds für das DOT-Tracking übernimmt die Action `DOTTrainerAction` aus der VisionLib, die auch schon bei der Erstellung eines 3D-Modells verwendet wird. Sie erhält als Eingabe das Set an Vergleichsbildern des DOT-Trackings, das aktuelle Kamerabild, eine *Region of Interest*, kurz ROI, und sowohl die intrinsischen als auch die extrinsischen Parameter. Mit der ROI wird das Kamerabild auf die Maße des

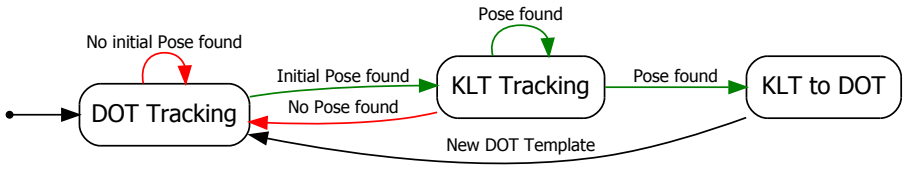
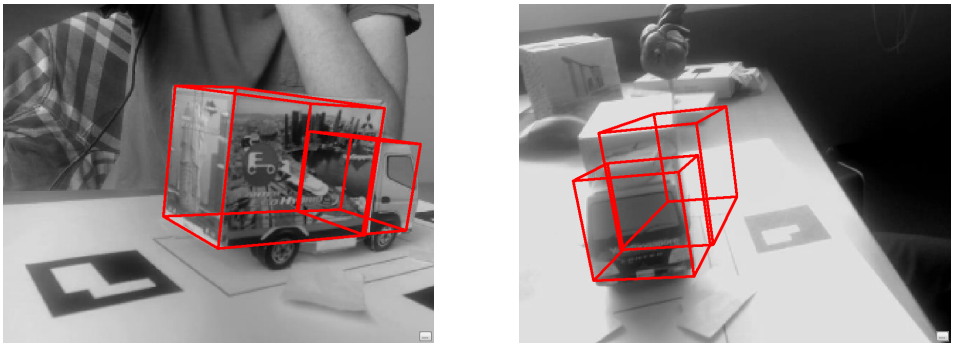


Abbildung 40: Die Tracking-Pipeline aus Abbildung 20, ergänzt mit einer Action zum Erstellen von Vergleichsbildern für das DOT-Tracking. Konnte das KLT-Tracking eine Kamerapose bestimmen, wird diese und das aktuelle Kamerabild dem DOT-Tracking übergeben. Dies wird als neues Vergleichsbild verwendet, wenn das DOT-Tracking das nächste Mal eine initiale Kamerapose für das KLT-Tracking berechnet.

Vergleichbildes zugeschnitten.Aus den extrinsischen Parametern wird die Kamerapose entnommen, die das DOT-Tracking ebenfalls benötigt. Diese Daten werden dann in das Set an Vergleichsbildern eingefügt, sodass das DOT-Tracking das neue Bild verwenden kann, sobald es das nächste Mal aufgerufen wird.
Würde jedes Kamerabild, aus dem das KLT-Tracking eine Kamerapose berechnen kann, in ein Vergleichsbild für das DOT-Tracking umgewandelt, wäre der Speicher schnell voll und die Zeit, die das DOT-Tracking pro Bild zum Vergleichen benötigt, würde auch stark ansteigen. Deshalb sollten nur ausgewählte Bilder vom DOT-Tracking übernommen werden, nach Möglichkeit die Bilder mit der stärksten Aussagekraft bzw. der am genausten berechneten Kamerapose. Dafür wurde eine Action vor die eigentliche Action zum Umwandeln der Kamerabilder in Templates geschaltet, die entscheidet, ob das aktuelle Kamerabild auch als Templates verwendet werden soll.
Als Auswahlkriterien wurden verschiedene Parameter definiert. Da davon ausgegangen wird, dass die Eingabebilder ein kontinuierlicher Strom an Bildern einer Kamera ohne Unterbrechungen sind, kann sich das Objekt im Bild von Bild zu Bild nicht stark verändern. Deshalb ist ein erster Auswahlpunkt die Zeit. Wurde innerhalb einer vorher definierten Zeitspanne bereits ein Bild zu einem Vergleichsbild konvertiert, wird davon ausgegangen, dass sich das Objekt im aktuellen Kamerabild nicht wesentlich von dem des zuvor aufgenommen Bild unterscheidet und deshalb kein neues Vergleichsbild erstellt werden muss. Natürlich kann es vorkommen, dass die Kamera über eine länger Zeit an der gleichen Position verharrt. Zusätzlich wird deshalb geprüft, wie sich die Position der Kamera seit dem letzten erstellten Vergleichsbild verändert hat. Sind hier keine signifikanten Unterschiede festzustellen, besteht keine Notwendigkeit, ein neues Vergleichsbild zu erzeugen, weil in der Umgebung der aktuellen Kamerapose bereits ein Vergleichsbild erstellt wurde, welches das DOT-Tracking verwenden

kann.
Sind beide zuvor genannten Bedingungen erfüllt, also dass genügend Zeit seit dem letzten Vergleichsbild vergangen ist und sich die Position der Kamera ausreichend verändert hat, kann ein neues Vergleichsbild erstellt werden. Dabei ist jedoch zu beachten, dass es vorkommen kann, dass die berechnete Kamerapose des KLT-Trackings nicht mit der tatsächlichen übereinstimmt. Dies sollte man ausschließen, da sonst das DOT-Tracking später falsche Ergebnisse liefert und somit die Reinitialisierung des KLT-Trackings stark stört, anstatt sie zu beschleunigen.
Das erste Maß, um zu bestimmen, wie gut die berechnete Kamerapose ist, ist die Anzahl der gefunden Übereinstimmungen zwischen den Merkmalen im Kamerabild und den Merkmalen aus dem KLT-Modell. Sind nur wenige Übereinstimmungen gefunden, ist die Wahrscheinlichkeit höher, dass dies Fehlzuordnungen sind. Sind hingegen viele Übereinstimmungen gefunden, steigt damit die Wahrscheinlichkeit, dass auch wirklich das Objekt im Bild gefunden wurde. Dies bedeutet ebenfalls, dass das Objekt gut im Bild sichtbar ist und damit die Kamerapose daraus gut berechnet werden kann. Wenige Übereinstimmungen bedeuten zwar nicht, dass das Objekt gar nicht im Bild gefunden wurde, weisen aber darauf hin, dass es z. B. schlecht sichtbar ist, siehe dazu Abbildung 41.

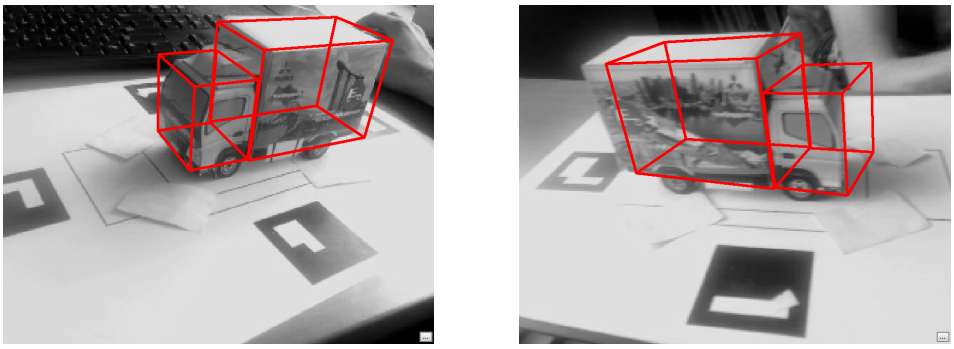


(a) (b)

Abbildung 41: Vergleich zweier berechneter Kameraposen mit einer unterschiedlichen Anzahl an übereinstimmenden Merkmalen zwischen Kamerabild und KLT-Modell. Im Hintergrund ist das Kamerabild zu sehen. Im Vordergrund rot eingezeichnet das Linienmodell des Objektes, projiziert mit der Kamerapose auf die Bildebene. In Abbildung 41(a) wurden 97 Korrespondenzen gefunden. Die Überlagerung vom Linienmodell des Objektes mit dem Kamerabild zeigt, dass die berechnete Kamerapose stimmt. In Abbildung 41(b) wurden nur sieben Korrespondenzen gefunden. Diese Überlagerung zeigt, dass die berechnete Kamerapose zu weit rechts von der richtigen Position der Kamera liegt.

Ein weiteres Maß zum Bewerten der Kamerapose ist die Kovarianz der be-

rechneten Kamerapose. Diese ergibt sich aus der Rückprojektion der 3D-Punkte der gefunden Korrespondenzen mit der vom KLT-Tracking berechneten Kamerapose auf die 2D-Bildebene. Diese 2D-Punkte und die aus dem Kamerabild extrahierten Merkmale werden einander zugeordnet und die Differenz berechnet. Diese Differenz ergibt den jeweiligen Projektionsfehler. Aus der Kovarianz-Matrix der Projektionsfehler lässt sich eine Aussage über die Genauigkeit der berechneten Pose treffen. Berechnet man für diese Matrix die Frobeniusnorm [31], erhält man einen absoluten Wert, den man mit einem Schwellwert vergleichen kann. Liegt der Wert unter dem Schwellwert, kann die berechnete Kamerapose als gut angesehen werden.



(a) (b)

Abbildung 42: Vergleich der berechneten Kameraposen durch das KLT-Tracking mit unterschiedlicher Kovarianz. Im Hintergrund wird das Kamerabild angezeigt, darüber ist mit Rot das Linienmodell des Objektes eingezeichnet. Dabei wurde es mit der berechneten Kamerapose auf die Bildebene projiziert. In Abbildung 42(a) hat die berechnete Pose eine Kovarianz von $0,146$. Die berechnete Pose stimmt fast perfekt mit der tatsächlichen überein. In Abbildung 42(b) hat die berechnete Pose eine Kovarianz von $5,667$, man sieht deutlich, dass es bei der Überlagerung zu Unstimmigkeiten kommt.

Sind alle eben beschriebenen Bedingungen erfüllt, kann aus dem aktuellen Kamerabild ein neues Vergleichsbild für das DOT-Tracking erstellt werden. Dies übernimmt die am Anfang des Abschnitts vorgestellte Action `DOTTrainerAction`.
Bei Versuchen hat sich dabei gezeigt, dass gute Werte für den zeitlichen Abstand, nachdem erst wieder ein neues Template erstellt wird, 20 bis 25 Bilder sind und der räumliche Abstand zwischen den Kamerapositionen der Templates 5 Einheiten betragen sollte. Ein guter Wert für die Kovarianz ist $0,5$, den die vom KLT-Tracking berechnete Pose nicht übersteigen sollte. Auch sollten mindestens, je nach Anzahl der Merkmale im Modell, zwische 5% und 10% aller Merkmale im Kamerabild wiedergefunden sein. Um zu verhindern, dass bei längeren Bildsequenzen der benötigte Speicher

für die DOT-Templates zu stark ansteigt, wird zusätzlich darauf geachtet, „alte“ Templates aus dem System zu entfernen. Alt ist ein Template dann, wenn es in seiner Umgebung, sprich es eine ähnliche Kamerapose gibt, die zu einem aktuellerem Zeitpunkt aufgenommen wurde. Dazu wurde zu jedem DOT-Template zusätzlich der Aufnahmezeitpunkt gespeichert, um bewerten zu können, wie lange das Template schon existiert.
Die Ähnlichkeit zwischen zwei Kameraposen wurde bestimmt, indem von der jeweiligen Kameraposition aus entlang des Blickvektors ein Zylinder aufgespannt wurde. Zwischen den zwei Zylindern wurde ermittelt, wie stark sich die beiden überschneiden. Ist die Überschneidung groß genug, wurden die beiden Kameraposen als ähnlich angesehen.
Ist ein Template aus dem Datensatz ähnlich zu dem aktuellen Template, welches gespeichert werden soll, kann auf die Aufnahmezeit verglichen werden. Gibt es das Template schon länger als ein Schwellwert, wird das Template aus dem Datensatz entfernt und durch das neue Template ersetzt. Damit soll dem DOT-Tracking garantiert werden, dass ihm aktuelle Templates zum Vergleich mit dem Kamerabild zu Verfügung stehen und die Menge an Templates eine bestimmte Größe nicht überschreitet.
Genau so kann man die DOT-Tracking-Action, um die Zahl der Vergleiche zu minimieren. Das DOT-Tracking kann viele Bildvergleiche in kurzer Zeit ausführen, ab einer bestimmten Anzahl von Bildern wird jedoch auch diese Menge zu groß. Daher ist die Idee, falls eine alte Kamerapose, z. B. die letzte valide Pose des KLT-Trackings, dem DOT-Tracking zu Verfügung steht, kann die Menge an Bildvergleichen reduziert werden. Dazu werden die Posen der Templates mit denen der letzten validen Kamerapose verglichen. Sind diese Pose ähnlich wie in dem im Absatz zuvor beschriebenen Verfahren, zu weit auseinander, wird das Template übersprungen. Dies kann gemacht werden, da von einer kontinuierlichen Sequenz ausgegangen wird und von Bild zu Bild sich die Position nicht stark ändert. So kann man davon ausgehen, dass die neue initiale Pose für das KLT-Tracking nahe an der alten Liegen muss und daher nur die Templates mit dem Kamerabild verglichen werden müssen, die sich nahe an der letzten Kamerapose befinden. In der Praxis hat sich jedoch gezeigt, dass dies Verfahren dafür sorgt, dass der mittlere Projektionsfehler kleiner wird, aber im Vergleich dazu die Anzahl der gefunden Kameraposen zu stark abnimmt. Dies kommt daher, da in der Praxis doch größere Verschiebungen der Kameraposition von Bild zu Bild auftreten können. Ist dies der Fall, sucht das DOT-Tracking zuerst dennoch erst in der Umgebung und weitet erst nach einer gewissen Zeit seinen Suchraum aus. Dadurch wird eine initiale Kamerapose für das DOT-Tracking erst später gefunden.

5.4.3 Bildpyramide

Der DOT-Tracker ist nicht skalierinvariant, da er mit Bildvergleichen arbeitet. Deshalb muss das Template sehr genau zum aktuellen Kamerabild passen, damit dessen Kamerapose als initiale Pose übernommen wird. Durch eine Bildpyramide kann erreicht werden, dass das Template mit verschiedenen Auflösungsstufen dem DOT-Tracker zu Verfügung steht. Dazu wird zusätzlich ein Verfahren benötigt, dass bei der Skalierung des Templates auch die dazugehörige Kamerapose anpasst. Damit kann der DOT-Tracker mit weniger Templates besser eine initiale Kamerapose für das KLT-Tracking finden.
Um eine solche Bildpyramide zu erstellen, muss als erstes das Kamerabild verkleinert werden. Außerdem muss für das verkleinerte Kamerabild die von der `DOTTrainerAction` benötigte ROI auf das kleinere Bild angepasst werden und die extrinsischen Kameraparameter verändert werden. Dies übernimmt eine implementierte Action `KLTtoDOTPyramidAction`, die als Eingabe die ursprüngliche ROI und extrinsischen Kameraparameter erhält und die angepassten Daten für zwei Skalierungsstufen der Bildpyramide ausgibt. Zusätzlich können zwei Faktoren definiert werden, mit denen die Eingabedaten skaliert werden sollen. Die Action skaliert die ROI, indem ihren Mittelpunkt mit dem Skalierungsfaktor multipliziert wird. So entsteht die angepasste ROI für die skalierten Kamerabilder. Um die extrinsischen Kameraparameter anzupassen, muss die Kameraposition verändert werden. Mit dem Verkleinern oder Vergrößern des Bildes ändert man nicht den Blickvektor der Kamera, aber deren Position. Wie in Abbildung 43 zu sehen ist, besteht eine Abbhängigkeit zwischen der Größe des Objektes im Bild und der Distanz von Kamera zu Objekt. Diese verhält sich anti-proportional. Deshalb wird die Kameraposition mit dem invertierten Skalierungsfaktor multipliziert.
Sind ROI und extrinsische Kameraparamter angepasst, können diese zusammen mit den verkleinerten Kamerabildern an die `DOTTrainerAction` weitergeben werden, die darauß neue DOT-Templates erstellt. Mit diesem Verfahren erhält man aus einem Kamerabild drei DOT-Templates, das des original Kamerabildes und die der zwei Verkleinerungsstufen. In Abbildung 44 sind diese drei Stufen abgebildet. So stehen dem DOT-Tracking mehr Templates zu Verfügung, die unterschiedliche Abstände von Kamera zu Objekt repäsentieren. Das DOT-Tracking erhält damit eine Methode, die die Skalierinvarianz des DOT-Trackings ausgleicht.

5.4.4 Evaluation

Nachfolgend werden die Ergebnisse dieses Ansatzes vorgestellt. Angewendet auf den gleiche Bildsequenz wie in Abschnitt 5.2.3. Die berechneten

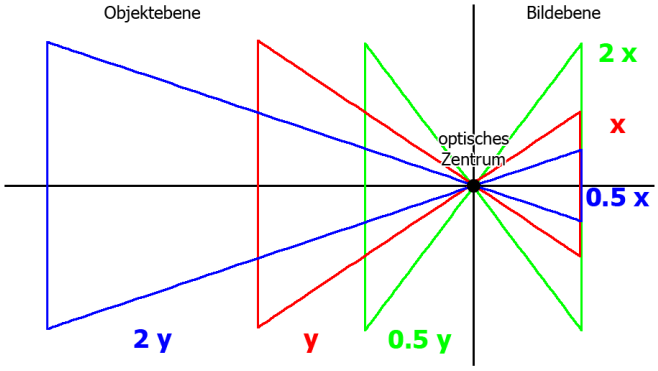


Abbildung 43: Relation von Objektgröße auf Bildebene zu Distanz auf Objektebene im Lochkameramodell. Je größer das Bild auf der Bildebene wird, desto weiter entfernt vom optischem Zentrum muss das Objekt auf der Objektebene sein.

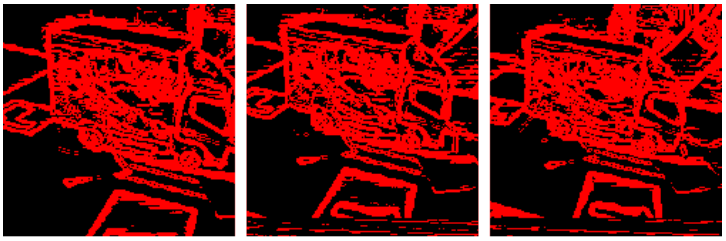


Abbildung 44: Bildpyramide des DOT-Trackings. Das linke Bild ist das DOT-Template des original Kamerabilds. Das mittlere Template wurde mit einem Faktor von $0,75$ skaliert, das rechte mit einem Faktor von $0,66$.

Kameraposen wurden mit einer Ground Truth verglichen, um so den Projektionsfehler zu ermitteln. Die Ergebnisse werden in Tabelle 3 gezeigt.
Man kann erkennen, dass die Anzahl der gefundenen Kameraposen signifikant höher ist, als mit dem ursprünglichen Zustand des Trackings. Während ohne die Erstellung von Online-Templates bei der Sequenz „Würfel“ nur rund 35% der Kameraposen im Bild gefunden wurden, sind es mit den Online-Templates fast 60%. Bei der Sequenz „Lastwagen“ konnte die Erfolgsrate von gefunden Kameraposen von rund 50% auf 86% gesteigert werden. Es ist also eine deutliche Verbesserung zu erkennen. Der mittlere Projektionsfehler verbessert sich auch leicht. So verbessert er sich bei der Sequenz „Würfel“ von $5,59$ Pixel auf $5,33$ und bei der Sequenz „Lastwagen“ von $6,68$ auf $6,36$.
Schaut man sich Abbildung 45 an, sieht man, dass bei mehr Einzelbildern eine Kamerapose gefunden wurde und so das Diagramm dichter gefüllt ist.

Sequenz	Gefundene Kameraposen (in %)	Mittlerer Projektionsfehler (in Pixel, Auflösung 640×480)
Würfel	57,6	5,33
Lastwagen	86,2	6,36

Tabelle 3: Anzahl der gefunden Kameraposen und mittlerer Projektionsfehler einer Bildsequenz für das Tracking-Verfahren mit Erstellung neuer DOT-Templates zur Laufzeit.

Einzig zwischen Bild 300 bis 400 und 600 bis 800 sind noch Lücken im Diagramm. Diese resultieren daraus, dass bei diesen Bildern auf die schmale Seite des Lastwagens geschaut wurde, so wie in Abbildung 41(b). Dort sind für den KLT-Algorithmus zu wenig Merkmale vorhanden, sodass er dort eine schlechte bis keine Kamerapose berechnet werden kann.
Schaut man sich in Abbildung 46 das Histogramm der Projektionsfehler der Sequenz an, erkennt man, dass sich das Maxima des Histogramms leicht nach links verschiebt. Dies zeigt, dass durch eine bessere Initialisierung auch der durchschnittliche Projektionsfehler, wenn auch nur leicht, verbessert wird. Außerdem wird der Anteil an Bildern, bei denen eine Kamerapose mit sehr hohem (> 25) Projektionsfehler berechnet wurde, kleiner im Vergleich zum ursprünglichem Tracking.
Fügt man der Pipeline bei der Erstellung von neuen DOT-Templates zusätzlich noch das Verfahren zur Erstellung einer Bildpyramide aus Abschnitt 5.4.3 hinzu, kann man einen weiteren leichten Anstieg in der Zuverlässigkeit des Trackings beobachten. Wie in Tabelle 4 zu sehen, steigt bei der Sequenz „Würfel“ der Prozentsatz der Anzahl an gefunden Kameraposen noch weiter auf $60,6\%$ mit der Bildpyramide. Mit der Sequenz „Lastwagen“ geht der Prozentsatz an gefunden Kameraposen leicht zurück auf $84,3\%$. Dies ist damit zu erklären, dass in der Sequenz keine starke Bewegung der Kamera in der Tiefe passiert. Dadurch bringt dort eine solche Bildpyramide keinen Vorteil. Allgemein gesehen zeigt sich jedoch, dass mit dieser Bildpyramide das Zusammenspiel aus DOT- und KLT-Tracking noch weiter verbessert werden kann.

5.4.5 Fazit

Die Ergebnisse in Abschnitt 5.4.4 zeigen, dass es mit einer Verbesserung der Initialisierung möglich ist, das Tracking robuster für 360-Grad-Tracking zu machen.
Durch die Erstellung neuer DOT-Templates stehen dem DOT-Tracking mehr und genauere Templates zu Verfügung, mit der sich besser eine initiale Pose ermitteln lässt. So kann das DOT-Tracking z. B. wenn durch eine Ruckeln

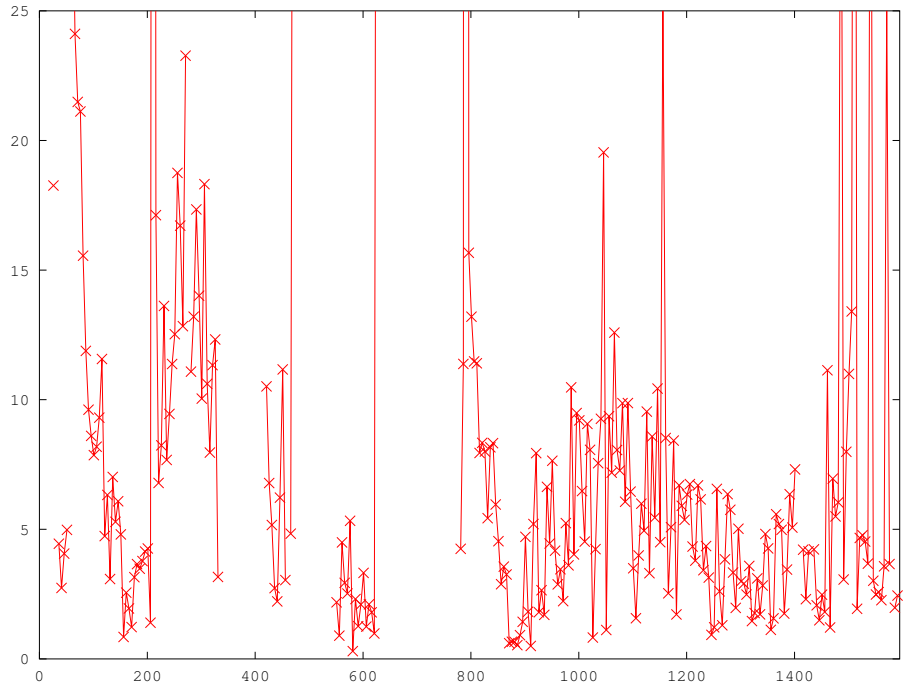


Abbildung 45: Projektionsfehler beim Tracking der Bildsequenz „Lastwagen“, auf die das in Abschnitt 5.4 beschriebene Verfahren angewandt wurde. Das Diagramm zeigt den Projektionsfehler pro Bild der Sequenz. Entlang der X-Achse ist der Verlauf der Bilder dargestellt, die Y-Achse zeigt die Höhe des Projektionsfehlers pro Bild an. Ist für ein Bild kein Kreuz eingezeichnet, bedeutet dies, dass der Algorithmus für dieses Bild keine Kamerapose berechnen konnte.

an der Kamera das KLT-Tracking versagt, das Objekt direkt im Bild wiederfinden, da zuvor ein neues Template aus einem ähnlichen Blickwinkel erstellt wurde. Auch kann mit diesem Ansatz das DOT-Modell des Objekts zur Laufzeit hin erweitert werden. Ist z. B. nur ein DOT-Modell von der Vorderseite des Objektes vorhanden, kann bei erfolgreichem KLT-Tracking Templates von anderen Blickwinkeln auf das Objekt erstellt werden und so nach und nach das DOT-Modell auf 360 Grad erweitern.

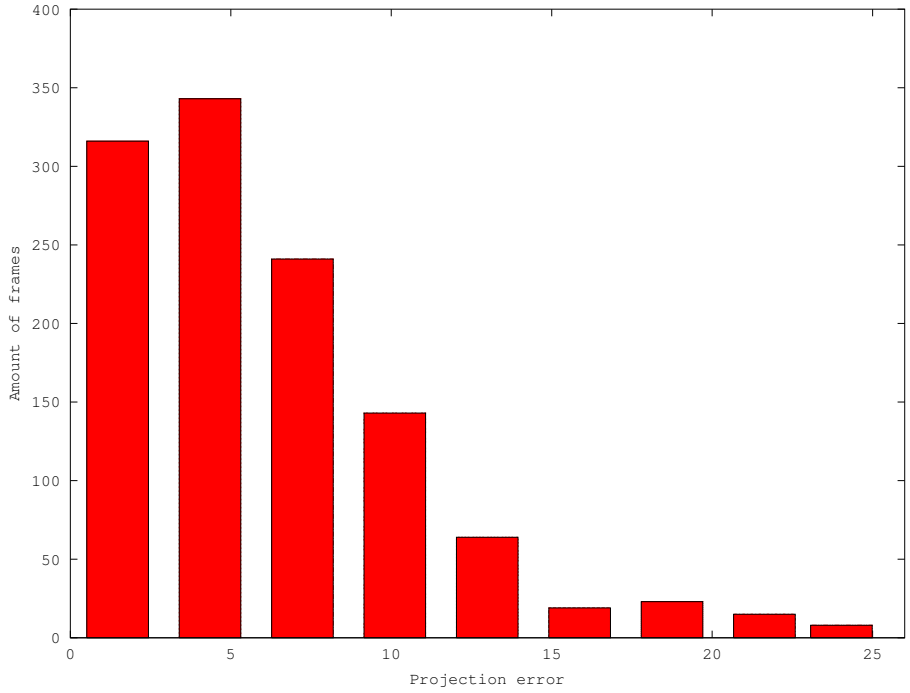


Abbildung 46: Histogramm der Projektionsfehler aus dem Datensatz zu Abbildung 45. Auf der X-Achse ist die Höhe des Projektionsfehlers eingezeichnet, auf der Y-Achse die Anzahl der Bilder, die einen Projektionsfehler in dieser Höhe haben. Dabei werden zur Übersichtlichkeit die Projektionsfehler in Klassen eingeteilt. Jede Klasse ist mit einem roten Balken markiert und hat die Länge von 3 Einheiten. Der erste rote Balken zeigt also die Anzahl der Bilder an, die einen Projektionsfehler zwischen 0 und 3 besitzen.

Sequenz	**Gefundene Kameraposen (in %)**	**Mittlerer Projektionsfehler (in Pixel, Auflösung 640×480)**
Würfel	60,6	4,91
Lastwagen	84,3	6,41

Tabelle 4: Anzahl der gefunden Kameraposen und mittlerer Projektionsfehler einer Bildsequenz für das Tracking-Verfahren mit Erstellung neuer DOT-Templates zur Laufzeit und zusätzlicher Erstellung einer Bildpyramide für die DOT-Templates.

5.5 Vergleich der Ansätze

Im Vergleich der drei in diesem Abschnitt vorgestellten Ansätze zeigt sich, dass der dritte Ansatz das meiste Potenzial bietet.
Das Filtern der Merkmale nach ihrer Lage zur Kamera hat auf das KLT-Tracking keine Auswirkung gehabt, da der Algorithmus schon im Ausgangszustand aus den gegebenen Korrespondenzen eine Kamerapose berechnen konnte. Die Filterung hatte also keinen Einfluss auf das KLT-Tracking, um das System im Sinne der Stabilität zu verbessern.
Mit der Vorhersage der Pose sollte die Reinitialisierung des KLT-Trackings verbessert werden, wenn dieses ausfällt. Damit sollten Störungen in den Bildsequenzen überbrückt werden. Durch das komplexe Bewegungsmodell der Kamera war es jedoch nicht möglich, einen Ansatz zu finden, mit dem das KLT-Tracking zuverlässig neu initialisiert werden konnte.
Der dritte Ansatz diente ebenfalls dazu, das Reinitialisieren des KLT-Trackings zu verbessern, jedoch auf eine andere Art. Durch die Möglichkeit, mit dem DOT-Tracking viele Bildvergleiche in kürzester Zeit auszuführen, werden neue Bilder zur Template-Datenbank hinzugefügt. Die neuen Bilder entstehen aus den aktuellen Kamerabildern. Eine Metrik wählt passende Bilder aus, bei denen das KLT-Tracking eine Kamerapose berechnen konnte. Durch die neuen Templates hat das DOT-Tracking eine größere und bessere Auswahl an Referenzbildern, mit denen es eine initiale Kamerapose für das KLT-Tracking berechnen kann. Durch die Geschwindigkeit des DOT-Trackings ist es so möglich, innerhalb weniger Bilder das KLT-Tracking wieder neu zu initialisieren.
Deshalb ist einzeln betrachtet der dritte Ansatz am Besten. Eine Alternativ ist noch die Kombination aus verschiedenen Ansätzen. Nimmt man z. B. zuerst eine Filterung der Merkmale nach ihrer Position (Abschnitt 5.2.2) vor und wendet anschließend noch das Erstellen neuer Templates aus den aktuellen Kamerabildern an (Abschnitt 5.4), kann man die Stabilität den Projektionsfehler senken, ohne dabei zu starke Verluste bei der Anzahl der gefunden Kameraposen zu erzeugen. Siehe dazu Tabelle 5 im Vergleich zu Tabelle 3.

Sequenz	**Gefundene Kameraposen (in %)**	**Mittlerer Projektionsfehler (in Pixel, Auflösung 640×480)**
Würfel	54,1	4,19
Lastwagen	85,7	5,98

Tabelle 5: Anzahl der gefunden Kameraposen und mittlerer Projektionsfehler einer Bildsequenz für das Tracking-Verfahren mit der Filterung „Winkel 2“ und der Erstellung neuer DOT-Templates zur Laufzeit.

6 Fazit

6.1 Zusammenfassung

Es gibt viele verschiedene Arten, wie sich bildbasiertes Tracking realisieren lässt. In Abschnitt 2.2 werden verschiedene Möglichkeiten vorgestellt. Diese Tracking-Verfahren werden angepasst, damit diese auch mit 3D-Objekten umgehen können bzw. 360-Grad-Tracking ermöglichen. Dabei werden meist unterschiedliche Verfahren miteinander kombiniert, um die Nachteile des einen Verfahrens mit den Vorteilen des anderen Verfahrens auszugleichen. In Abschnitt 5.5 werden exemplarisch solche Ansätze vorgestellt. Durch die große Anzahl an unterschiedlichen Tracking-Verfahren entstehen viele verschiedene Möglichkeiten, die noch nicht alle untersucht wurden.
Die VisionLib-Bibliothek für diesen Kontext ein Framework zu Verfügung. Durch dessen Größe und Komplexität , sind viele Dinge bereits vorhanden. Es sind verschiedene Tracking-Ansätze implementiert und Komponenten, wie z. B. die Ein- und Ausgabe von Kamerabildern vorhanden. Durch die modulare Pipeline-Struktur lassen sich so schnell neue Ideen umsetzen und testen.
Da bereits ein 3D-Tracking-Ansatz im Framework vorhanden war, musste dieser zunächst evaluiert werden und auf seine Fähigkeiten bzgl. 360-Grad-Tracking untersucht werden. Die Analyse hat ergeben, dass 360-Grad-Tracking mit diesem Ansatz unter Laborbedingungen bereits möglich ist, in der Praxis jedoch noch nicht stabil funktioniert. Deshalb wurden neue Ansätze entwickelt, die das 3D-Tracking in Bezug auf 360-Grad-Tracking unterstützen und es robuster machen sollen. Es sind drei unterschiedliche Verfahren entstanden.
Das Filtern der Punkte brachte im Zusammenspiel mit dem KLT-Tracking keine deutliche Verbesserung für das 360-Grad-Tracking, jedoch haben selbst kleine Verbesserungen gezeigt, dass hier Potenzial besteht für andere Tracking-Verfahren. Hat das verwendete Verfahren zur Korrespondenzenbildung zwischen Modell und Bild keine gute Methode für die Zuordnung, kann ein vorheriges Filtern der Punkte die Stabilität des Verfahrens verbessern.
Die Vorhersage der Bewegung der Kamera entsprach nicht den Erwartungen. Durch das starke Zittern in der Positionsveränderung war es sehr schwierig, eine zuverlässige Vorhersage zu treffen. Hier konnte kein Ansatz gefunden werden, der das gewünschte Ergebnis lieferte.
Die besten Resultate lieferte das Erstellen und HinzufÃ$\frac{1}{4}$gen neuer Templates durch das DOT-Tracking. Durch die aktuellen Templates konnte das DOT-Tracking besser eine initiale Pose für das KLT-Tracking finden und so das KLT-Tracking schneller wieder reinitialisieren. Damit konnte gezeigt werden, dass durch die Anpassung eines bestehenden Tracking-Verfahrens an das 360-Grad-Tracking das Ziel erreicht werden konnte.

6.2 Ausblick

Um die Stabilität des Trackings noch weiter zu verbessern, gibt es verschiedene Möglichkeiten. Eine Idee, die Erstellung neuer Templates zu verbessern, ist die Überprüfung des Kamerabildes, das als neues Template ausgewählt wurde. Dies könnte man z. B. dadurch realisieren, indem man mit der aktuellen Kamerapose ein realistisches 3D-Modell oder Linienmodell auf die Bildebene projiziert. Durch einen Bildvergleich kann dann festgestellt werden, wie gut das projizierte Modell zu dem Bild passt. Ist die Übereinstimmung nicht groß, kann angenommen werden, dass die vom KLT-Tracking berechnete Kamerapose nicht gut genug ist. Somit würden nur neue Templates erstellt, bei denen die berechnete Kamerapose stimmt. Diese Idee kann man weiterführen, um die Template-Generierung zu verbessern. Hat man z. B. das Linienmodell auf die Bildebene projiziert, kennt man damit auch die Umrisse des Objekts auf der Bildfläche. Mit diesem Wissen kann man dann den Hintergrund vom eigentlichen Objekt im Bild trennen und zur Erstellung des DOT-Templates das Bild des Objekts ohne Hintergrund nutzen. Damit wird verhindert, dass z. B. durch einen zu stark strukturierten Hintergrund die Erstellung der initialen Pose für das DOT-Tracking durch Fehlzuordnungen gestört wird.
Ein Ansatz, um die Vorhersage der Bewegung besser schätzen zu können, ist darüber hinaus die Vorhersage der Bewegung der Merkmale auf Bildebene. Die Merkmale bewegen sich von Bild zu Bild nur in zwei Dimensionen und können so leichter verfolgt werden. Wenn man weiß, wie sich die Merkmale verhalten, kann man versuchen, eine Vorhersage zu treffen, wo sie als nächstes sein werden. Nutzt man die Vorhersagen mehrerer Merkmale, kann man daraus eine Kamerapose berechnen, die eine Schätzung ist, wie sich die Kamera fortbewegen könnte.
Allgemein kann es auf Grund der vielen verschiedenen bildbasierten Trackern auch versucht werden, einen komplett neuen Ansatz zu entwickeln. So benötigen die meisten Tracking-Verfahren, die auf 360-Grad-Tracking ausgelegt sind, immer eine initiale Pose, bevor sie mit dem Tracking beginnen können. Dies hat zur Folge, dass ein zweites Verfahren benötigt wird, welches eine initiale Pose für das eigentliche Tracking generiert. Ein Ansatz für 360-Grad-Tracking, der allein mit einem einzigen Tracking-Verfahren auskommt, wäre sicher wünschenswert.

Literatur

[1] Mario Becker, Harald Wuest, Folker Wientapper, and Timo Engelke. A prototyping architecture for augmented reality. In *2nd Workshop on Software Engineering and Architectures for Realtime Interactive Systems (SEARIS@VR 2009)*, 2009.

[2] Gabriele Bleser, Harald Wuest, and Didier Stricker. Online camera pose estimation in partially known and dynamic scenes. In *Proceedings of the 5th IEEE and ACM International Symposium on Mixed and Augmented Reality*, ISMAR '06, pages 56–65, Washington, DC, USA, 2006. IEEE Computer Society.

[3] Jean-Yves Bouguet and Pietro Peronan. Visual navigation. URL: `http://www.vision.caltech.edu/bouguetj/Motion/navigation.html`, 1998. Abgerufen am 13.11.14.

[4] N. Dalal and B. Triggs. Histograms of oriented gradients for human detection. In *Computer Vision and Pattern Recognition, 2005. CVPR 2005. IEEE Computer Society Conference on*, volume 1, pages 886–893 vol. 1, June 2005.

[5] Navneet Dalal and Bill Triggs. Histograms of oriented gradients for human detection. In Cordelia Schmid, Stefano Soatto, and Carlo Tomasi, editors, *International Conference on Computer Vision & Pattern Recognition*, volume 2, pages 886–893, INRIA Rhône-Alpes, ZIRST-655, av. de l'Europe, Montbonnot-38334, June 2005.

[6] Jose Dolz. Markerless augmented reality. URL: `http://www.arlab.com/blog/markerless-augmented-reality/`, Mai 2012. Abgerufen am 10.11.14.

[7] John W. Eaton. Polynomial interpolation. URL: `https://www.gnu.org/software/octave/doc/interpreter/Polynomial-Interpolation.html`, 2013. Abgerufen am 15.01.15.

[8] Timo Engelke, Mario Becker, Harald Wuest, Jens Keil, and Arjan Kuijper. Mobilear browser - a generic architecture for rapid ar-multi-level development. *Expert Syst. Appl.*, 40(7):2704–2714, June 2013.

[9] A. Fitzgibbon, M. Pilu, and R.B. Fisher. Direct least square fitting of ellipses. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 21(5):476–480, May 1999.

[10] Iryna Gordon and DavidG. Lowe. What and where: 3d object recognition with accurate pose. In Jean Ponce, Martial Hebert, Cordelia

Schmid, and Andrew Zisserman, editors, *Toward Category-Level Object Recognition*, volume 4170 of *Lecture Notes in Computer Science*, pages 67–82. Springer Berlin Heidelberg, 2006.

[11] Peter Schallauer Werner Bailer Hannes Fassold, Rosner J. Realtime klt feature point tracking for high definition video. In *GraVisMa 2009 Workshop Proceedings*, 2009.

[12] Rob Hess. An open-source siftlibrary. In *Proceedings of the International Conference on Multimedia*, MM '10, pages 1493–1496, New York, NY, USA, 2010. ACM.

[13] S. Hinterstoisser, V. Lepetit, S. Ilic, P. Fua, and N. Navab. Dominant orientation templates for real-time detection of texture-less objects. In *Computer Vision and Pattern Recognition (CVPR), 2010 IEEE Conference*, pages 2257–2264, June 2010.

[14] Irwandi Hipiny and Walterio Mayol-Cuevas. Recognising egocentric activities from gaze regions with multiple-voting bag of words. *Technical Report*, (CSTR-12-003), 2012.

[15] Derek Hoiem. Feature tracking and optical flow. University of Illinois, Presentation, 2012.

[16] Stefan Holzer. Object detection using dot. Willow Garage, Intern Presentation, June 2010.

[17] PeterJ. Huber. Robust statistics. In Miodrag Lovric, editor, *International Encyclopedia of Statistical Science*, pages 1248–1251. Springer Berlin Heidelberg, 2014.

[18] Rudolph Emil Kalman. A new approach to linear filtering and prediction problems. *Transactions of the ASME–Journal of Basic Engineering*, 82(Series D):35–45, 1960.

[19] J.J. Leonard and H.F. Durrant-Whyte. Simultaneous map building and localization for an autonomous mobile robot. In *Intelligent Robots and Systems '91. 'Intelligence for Mechanical Systems, Proceedings IROS '91. IEEE/RSJ International Workshop on*, pages 1442–1447 vol.3, Nov 1991.

[20] Vincent Lepetit, Luca Vacchetti, Daniel Thalmann, and Pascal Fua. Fully automated and stable registration for augmented reality applications. In *Proceedings of the 2Nd IEEE/ACM International Symposium on Mixed and Augmented Reality*, ISMAR '03, pages 93–, Washington, DC, USA, 2003. IEEE Computer Society.

[21] David G. Lowe. Distinctive image features from scale-invariant keypoints. *Int. J. Comput. Vision*, 60(2):91–110, November 2004.

[22] Bruce D. Lucas and Takeo Kanade. An iterative image registration technique with an application to stereo vision. In *Proceedings of the 7th International Joint Conference on Artificial Intelligence - Volume 2*, IJCAI'81, pages 674–679, San Francisco, CA, USA, 1981. Morgan Kaufmann Publishers Inc.

[23] Like It Matters. Malaysia: Kuala lumpur. URL: `http://likeitmatters.com/malaysia-kuala-lumpur/`, February 2014. Abgerufen am 22.01.15.

[24] A. Mordvintsev and K. Abid. Template matching. URL: `http://opencv-python-tutroals.readthedocs.org/en/latest/py_tutorials/py_imgproc/py_template_matching/py_template_matching.html`, October 2013. Abgerufen am 20.01.15.

[25] C.F. Olson and D.P. Huttenlocher. Automatic target recognition by matching oriented edge pixels. *Image Processing, IEEE Transactions on*, 6(1):103–113, Jan 1997.

[26] M. Pressigout and E. Marchand. Real-time 3d model-based tracking: combining edge and texture information. In *Robotics and Automation, 2006. ICRA 2006. Proceedings 2006 IEEE International Conference on*, pages 2726–2731, May 2006.

[27] Morgan Quigley, Ken Conley, Brian P. Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y. Ng. Ros: an open-source robot operating system. In *ICRA Workshop on Open Source Software*, 2009.

[28] J. Rekimoto. Matrix: a realtime object identification and registration method for augmented reality. In *Computer Human Interaction, 1998. Proceedings. 3rd Asia Pacific*, pages 63–68, Jul 1998.

[29] Jianbo Shi and Carlo Tomasi. Good features to track. Technical report, Ithaca, NY, USA, 1993.

[30] vojirtom. 4_tracking. URL: `https://cw.felk.cvut.cz/wiki/courses/ae4m33mpv/labs/4_tracking/start`, Mai 2014. Abgerufen am 10.12.14.

[31] Eric W. Weisstein. Frobenius norm. URL: `http://mathworld.wolfram.com/FrobeniusNorm.html`, 2015. Abgerufen am 25.01.15.

[32] Harald Wuest and Didier Stricker. Tracking of industrial objects by using cad models. *JVRB - Journal of Virtual Reality and Broadcasting*, 4(2007)(1), 2007.

A Sequenz „Würfel“

(a) (b) (c) (d)

Abbildung 47: Bildsequenz des Objekts „Würfel“ zur Modellerstellung. Es wurde dabei mit der Kamera einmal um das Objekt herum geschwenkt.

	Gefundene Kameraposen (in %)	**Mittlerer Projektionsfehler (in Pixel, Auflösung 640×480)**
Ist-Zustand	28,5	4,86
Winkel 1 Filter	37,9	6,44
Winkel 2 Filter	40,3	6,49
Status Filter	12,6	9,83
Tiefen Filter	28,9	5,53
Blickvektor Filter	40,3	6,49
Online Template	28,5	4,86
Online Template & Winkel 2 Filter	54,1	4,19
Online Template & Bildpyramide	60,6	4,91

Tabelle 6: Anzahl der gefunden Kameraposen und mittlerer Projektionsfehler bei der Bildsequenz „Würfel“.

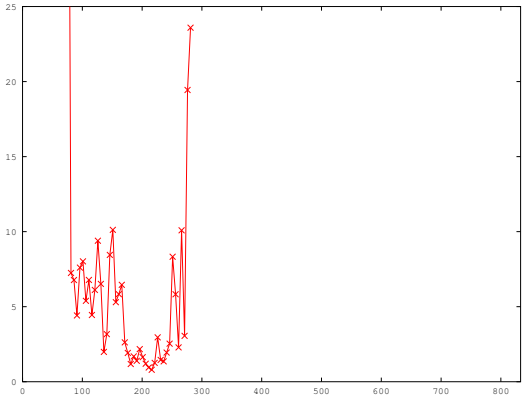


Abbildung 48: Ist-Zustand

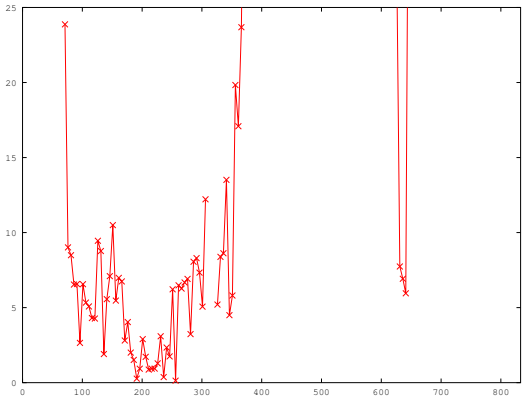


Abbildung 49: Winkel 1 Filter

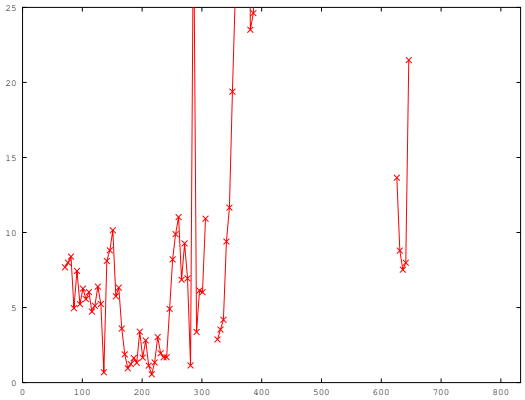


Abbildung 50: Winkel 2 Filter

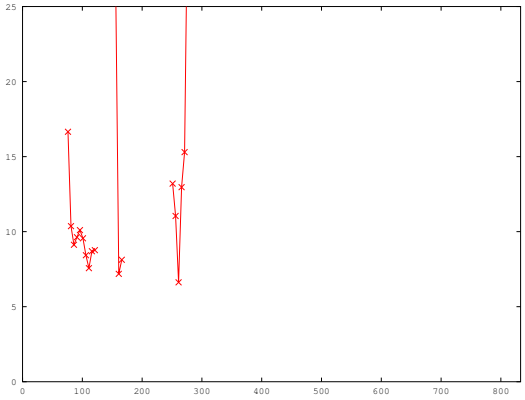


Abbildung 51: Status Filter

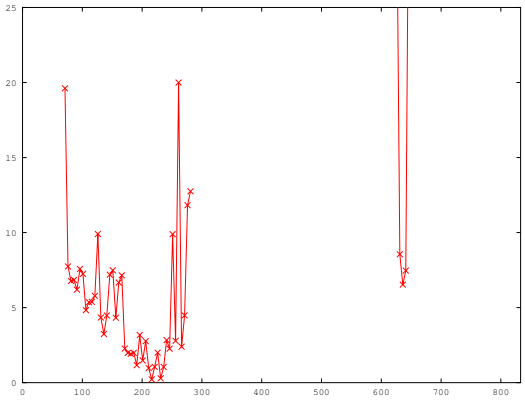


Abbildung 52: Tiefen Filter

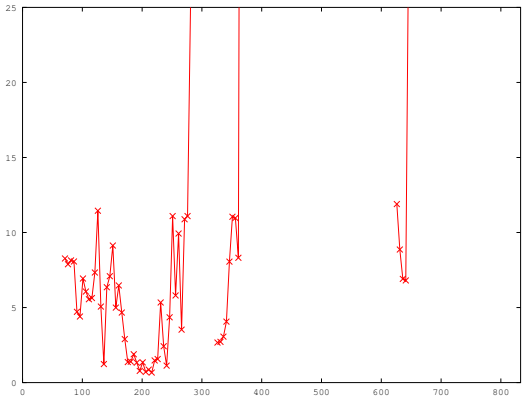


Abbildung 53: Blickvektor Filter

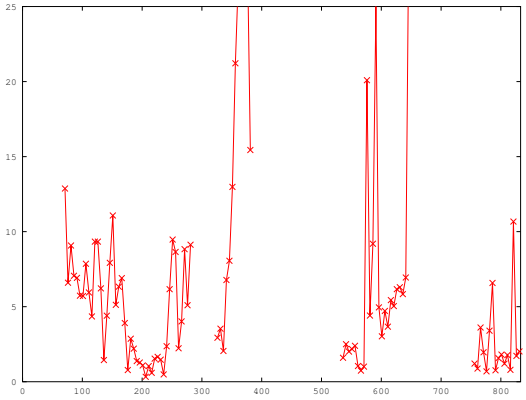


Abbildung 54: Online Templates

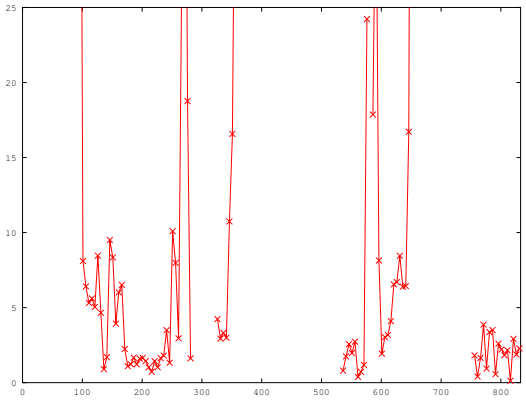


Abbildung 55: Online Templates & Winkel 2 Filter

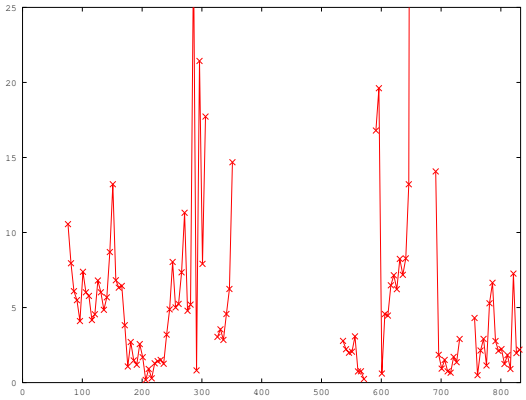


Abbildung 56: Online Templates & Bildpyramide

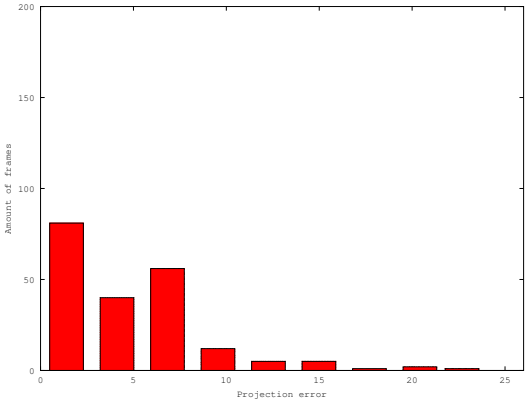


Abbildung 57: Ist-Zustand

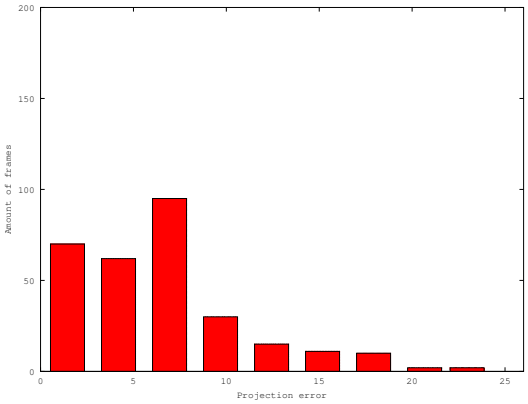


Abbildung 58: Winkel 1 Filter

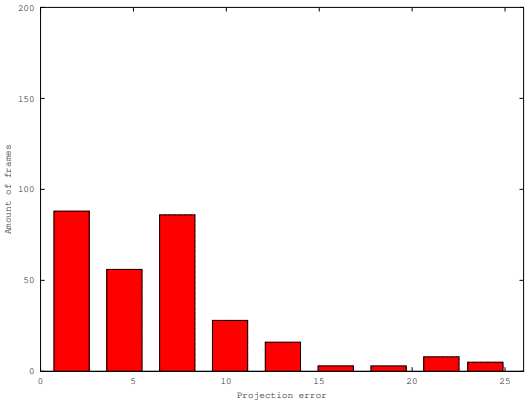


Abbildung 59: Winkel 2 Filter

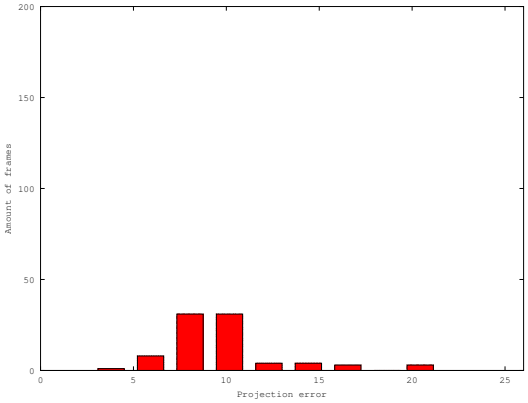


Abbildung 60: Status Filter

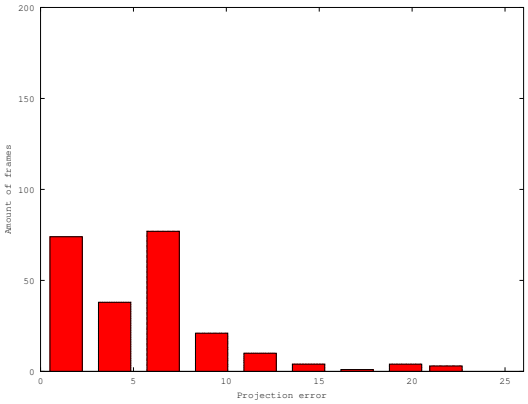


Abbildung 61: Tiefen Filter

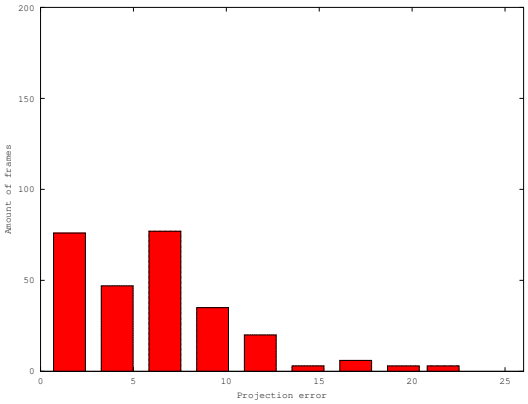


Abbildung 62: Blickvektor Filter

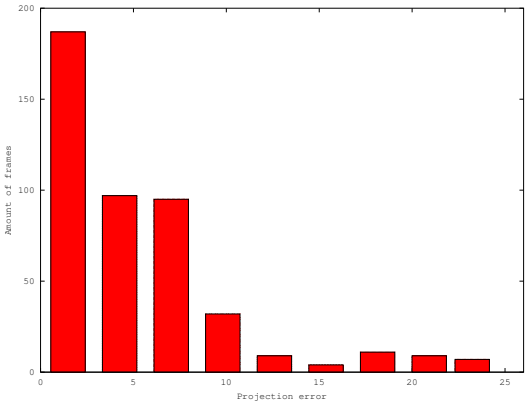


Abbildung 63: Online Templates

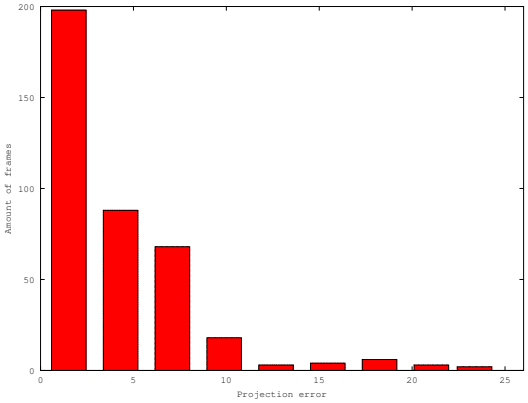


Abbildung 64: Online Templates & Winkel 2 Filter

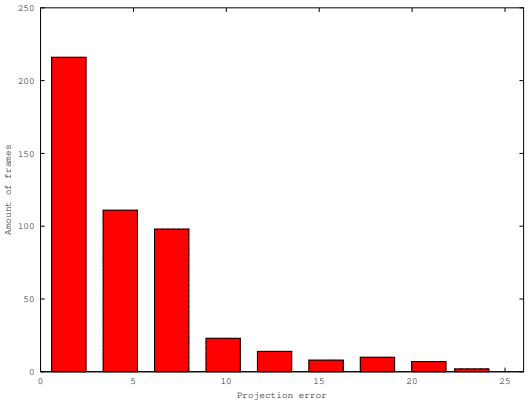


Abbildung 65: Online Templates & Bildpyramide, man beachte die Veränderte Skala der Y-Achse von einem Maximum von 200 auf 250.

B Sequenz „Lastwagen“

(a) (b) (c) (d)

Abbildung 66: Bildsequenz des Objekts „Lastwagen“ zur Modellerstellung. Es wurde dabei mit der Kamera einmal um das Objekt herum geschwenkt.

	Gefundene Kameraposen (in %)	**Mittlerer Projektionsfehler (in Pixel, Auflösung 640×480)**
Ist-Zustand	50,5	5,85
Winkel 1 Filter	46,5	6,69
Winkel 2 Filter	45,6	5,96
Status Filter	28,8	9,48
Tiefen Filter	45,2	6,14
Blickvektor Filter	47,5	6,47
Online Template	86,2	6,36
Online Template & Winkel 2 Filter	85,7	5,98
Online Template & Bildpyramide	84,3	6,41

Tabelle 7: Anzahl der gefunden Kameraposen und mittlerer Projektionsfehler bei der Bildsequenz „Lastwagen“.

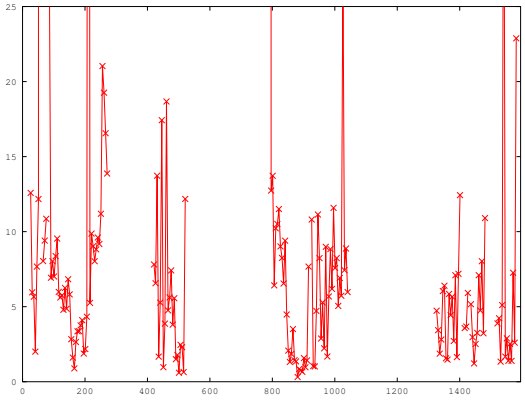


Abbildung 67: Ist-Zustand

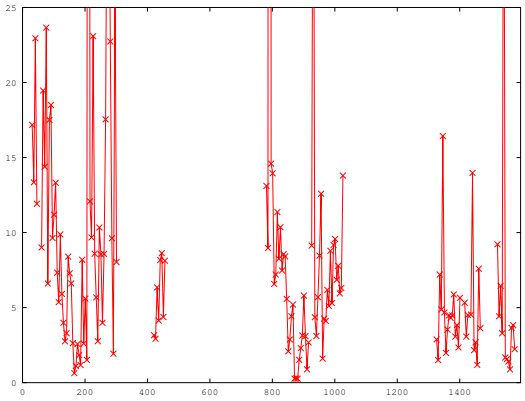


Abbildung 68: Winkel 1 Filter

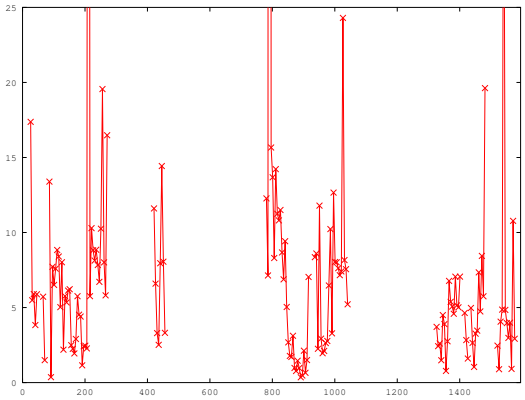


Abbildung 69: Winkel 2 Filter

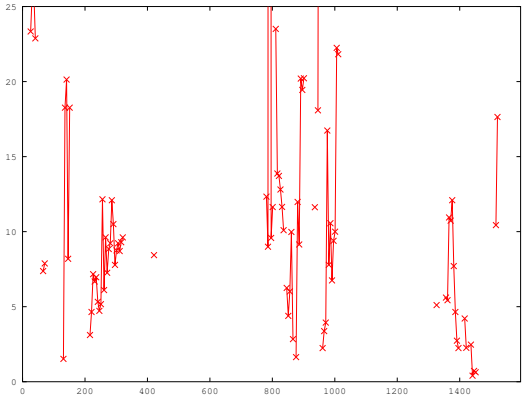


Abbildung 70: Status Filter

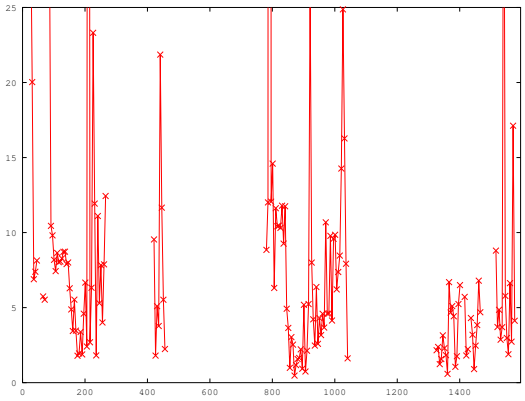


Abbildung 71: Tiefen Filter

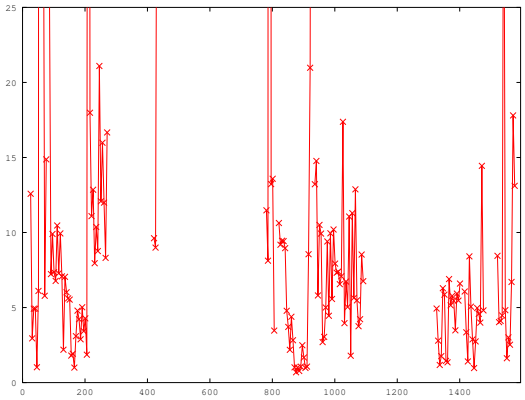


Abbildung 72: Blickvektor Filter

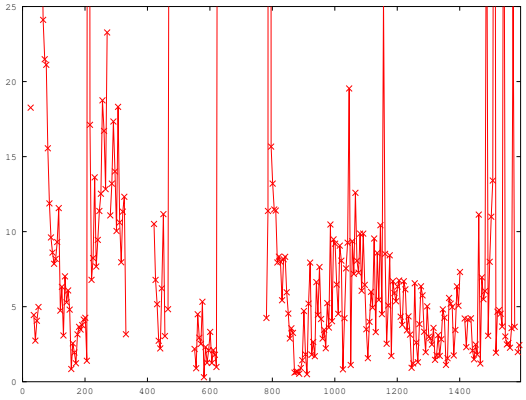


Abbildung 73: Online Templates

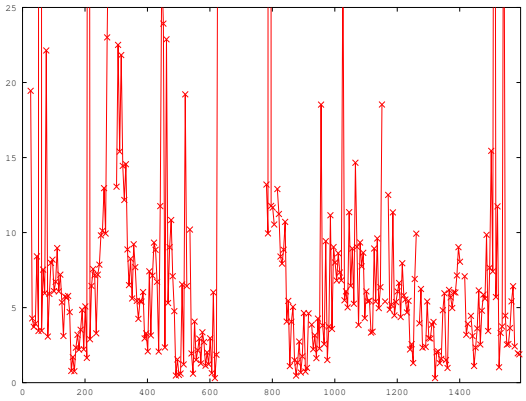


Abbildung 74: Online Templates & Winkel 2 Filter

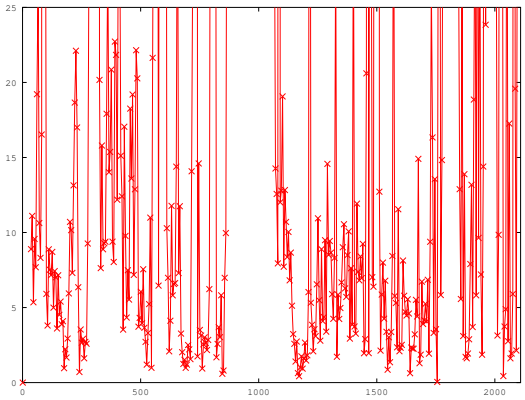


Abbildung 75: Online Templates & Bildpyramide

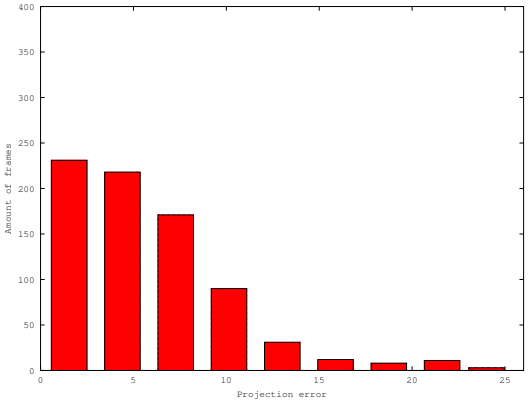


Abbildung 76: Ist-Zustand

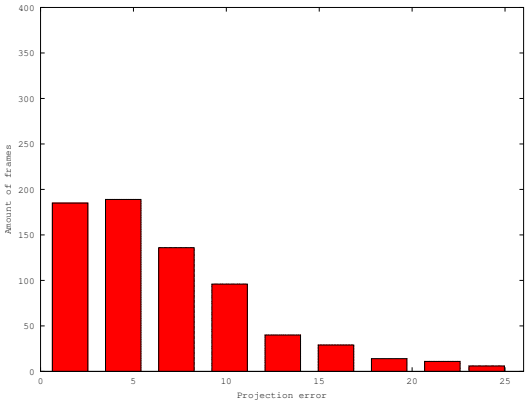


Abbildung 77: Winkel 1 Filter

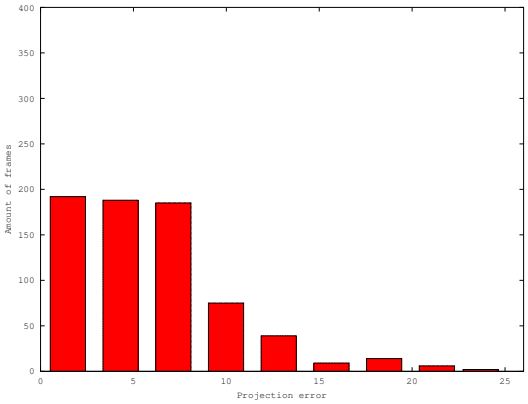


Abbildung 78: Winkel 2 Filter

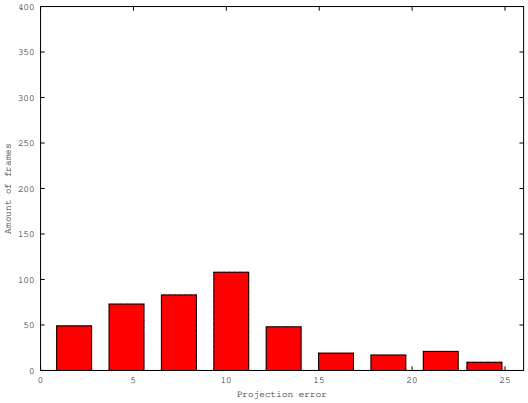


Abbildung 79: Status Filter

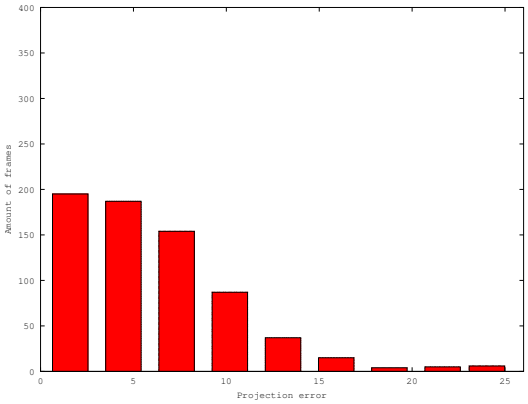


Abbildung 80: Tiefen Filter

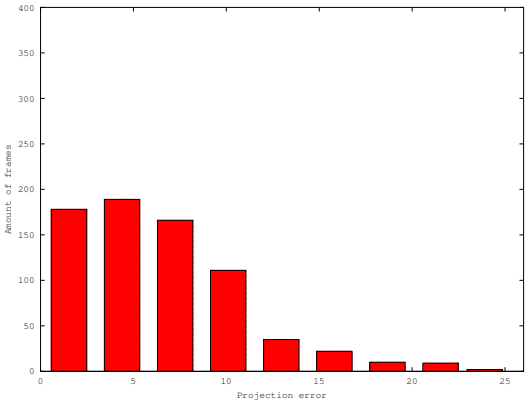


Abbildung 81: Blickvektor Filter

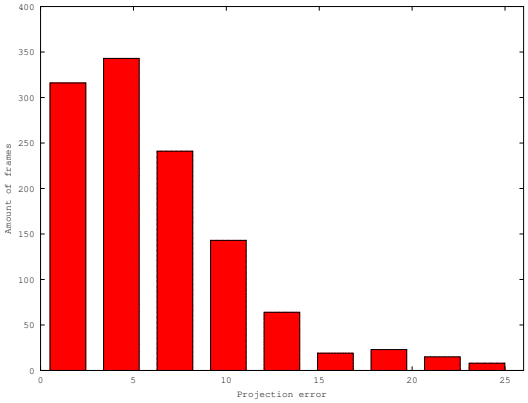


Abbildung 82: Online Templates

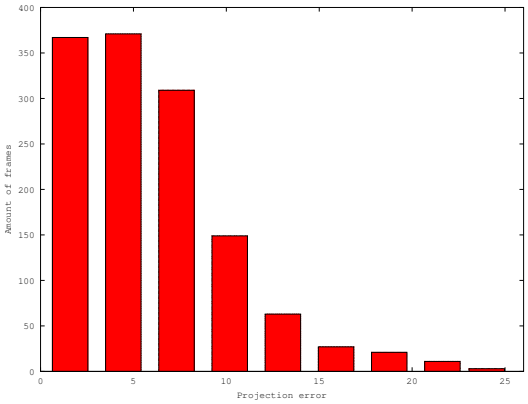


Abbildung 83: Online Templates & Winkel 2 Filter

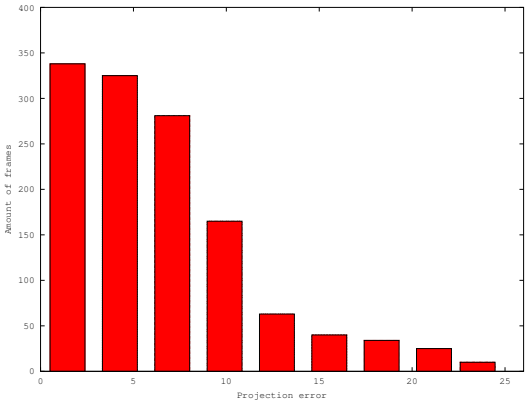


Abbildung 84: Online Templates & Bildpyramide