Besondere Lernleistung

Martin-Andersen-Nexö-Gymnasium Dresden

Automatisierte infrastrukturgestützte Klassifizierung und 3D-Lokalisierung nicht-motorisierter Verkehrsteilnehmer mittels Computer Stereo Vision

Peter Haiduk

Abgabetermin der Arbeit: 20.12.2019

Betreuer: Dipl.-Ing. Vincent Latzko

Technische Universität Dresden
Fakultät Elektrotechnik und Informationstechnik
Institut für Nachrichtentechnik
Deutsche Telekom Professur für Kommunikationsnetze

Kurzfassung

Neben dem automatisierten Fahren stellt vor allem das vernetzte Fahren einen großen Trend in der heutigen Verkehrsforschung dar. Hierbei wird besonders Wert auf den Informationsaustausch zwischen Verkehrsteilnehmern untereinander sowie zwischen Verkehrsteilnehmern und der Verkehrsinfrastruktur gelegt, denn durch eine breitere Informationsgrundlage können alle Beteiligten grundsätzlich qualifiziertere Entscheidungen treffen, wodurch Verkehrsfluss und -sicherheit positiv beeinflusst werden.

Das Ziel der vorliegenden Arbeit war es, einen Prototyp für die automatisierte optische Erfassung und dreidimensionale Ortung von nicht-motorisierten Verkehrsteilnehmern mithilfe zweier Verkehrskameras zu entwickeln. Darauf folgend wurde ein Versuch in einer realen Verkehrssituation vorbereitet, durchgeführt und ausgewertet, wofür der Programmcode in der Programmiersprache Python implementiert werden sollte.

Zur Erkennung der nicht-motorisierten Verkehrsteilnehmer wird ein quelloffener, auf Deep Learning basierender Object Detection Algorithmus fortlaufend auf die Bilder der beiden Verkehrskameras angewendet. Durch die bestimmte Pixel-Position eines solchen interessanten Objekts der Verkehrsszene auf den Bildern dieser beiden Kameras und unter Einbeziehung aller durch Kalibrierung bekannten Merkmale der Kameras wird auf die genaue Position des Verkehrsteilnehmers im dreidimensionalen Raum zurückgeschlossen. Diese Information kann dann im Rahmen der Car2Infrastructure Communication als Gefahrennachricht an sich unmittelbar in der Nähe befindende Kraftfahrzeuge gesendet und dort weiter verarbeitet werden.

Im Mittelpunkt der Arbeit stand das Erarbeiten der notwendigen mathematischen Grundlagen für die beschriebene dreidimensionale Ortung eines Objekts auf der Grundlage von zwei überlappenden Kamerabildern. Bei der Entwicklung des Prototypenkonzepts und der Durchführung des Versuches spielten allerdings noch viele weitere theoretische und praktische Aspekte eine Rolle. Dazu zählten unter anderem die Kamerakalibrierung, das Ordnen der gefundenen interessanten Objekte zu korrespondierenden Paaren zwischen den Kamerabildern, das Rechnen mit Kugelkoordinaten, der Einsatz von externer Karten- und Visualisierungssoftware, sowie die Programmierung in Python.

Der erarbeitete Prototyp erwies sich als wertvoller Beitrag zur Forschung am vernetzten Fahren, zum einen im Sinne des Versuches als Demonstrator und zum anderen als Grundlage für real einsetzbare Systeme. Gegenstand weiterer Forschung könnte die Vereinfachung des Kalibrierungsvorgangs, die Erhöhung der Genauigkeit und Zuverlässigkeit der 3D-Ortung, das Erfassen von Geschwindigkeitsinformationen und die Voraussage von Positionen der Verkehrsteilnehmer sein.

Abstract

Besides automated driving, particularly studies on connected cars are a major trend in today's traffic research. The focus of this principle lies on the exchange of information between the vehicles as well as between the vehicles and the road infrastructure. This is because a broader information basis will enable all parties involved to make more qualified decisions, which will have a positive impact on traffic flow and safety.

The present scholarly work aimed to develop a prototype for the automated optical detection and three-dimensional localization of non-motorized vehicles using two traffic cameras. Subsequently, a test in a real traffic situation was prepared, carried out and evaluated, for which the program code was to be implemented in the Python programming language.

To detect the non-motorized vehicles, an open source object detection algorithm based on deep learning is continuously applied to the images of the two traffic cameras. With the determined pixel position of an interesting object of the scene on the pictures of these two cameras, and by incorporating all features of the cameras obtained by calibration, the exact position of the road user in the three-dimensional space is concluded. Within the framework of Car2Infrastructure Communication, this information could then be sent as an alert message to motorized vehicles in the immediate vicinity and processed further there.

The key aspect of the research was the elaboration of the necessary mathematical foundations for the described three-dimensional localization process of an object based on two overlapping camera images. However, many other theoretical and practical aspects played a role in the development of the prototype concept and the conduct of the experiment. Among other things, this included camera calibration, arranging the found interesting objects into corresponding pairs between the camera images, operating with spherical coordinates, the use of external cartography and visualization software, as well as programming in Python.

The developed prototype proved oneself to be a valuable contribution to research on connected traffic, on the one hand as a demonstrator in the sense of the experiment and on the other hand as a basis for systems that can be used in real applications. Further research could focus on simplifying the calibration process, increasing the accuracy and reliability of 3D positioning, gathering speed information and predicting road users' positions.

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Allgemeines

1.1 Motivation

Ein großer Forschungsbereich bei der Anwendung von Computertechnologien im Alltag ist der Verkehr. Fast jeder hat schon von der Idee der selbst fahrenden Autos gehört und zahlreiche Automobilhersteller und Wissenschaftler konnten bisher erstaunliche Ergebnisse vorstellen. Assistenzsysteme werden heute schon in fast allen neuen Autos verbaut.

Doch viel weniger Aufmerksamkeit als das automatisierte Fahren [1] bekommt das Gebiet des vernetzten Fahrens [1]. Hier wird berücksichtigt, dass Straßenverkehr viel effizienter stattfinden kann, wenn Kraftfahrzeuge nicht nur auf sich bedacht (autonom [1]) fahren, sondern mit anderen Verkehrsteilnehmern und der Verkehrsinfrastruktur (z.B. Ampeln, digitale Verkehrstafeln, Parkautomaten oder Verkehrskameras) kommunizieren (*Car2Car Communication* und *Car2Infrastructure Communication* [2, 3]). Durch den Informationsaustausch wird ein sowohl sichererer als auch flüssigerer Verkehr ermöglicht und vorrangig gilt: geteilte Informationen sind besser als verschlossene Informationen. Denn je mehr Informationen ein Verkehrsteilnehmer oder z.B. eine Ampelanlage über die aktuelle Verkehrslage zur Verfügung hat, desto qualifizierter kann eine Entscheidung getroffen werden [2, 3].

Wegen seinem großen Potenzial wächst das Interesse am vernetzten Fahren in der Forschung und in der Wirtschaft stetig. Beispielhaft zu nennen ist hier das Forschungs-Projekt *Ko-HAF*[1], bei dem vier deutsche Univeritäten mit zahlreichen Industriepartnern unter staatlicher Förderung eine führende Kompetenz im Bereich automatisiertes Fahren und vernetztes Fahren entwickeln [4, 5].

Auch diese Arbeit soll dabei helfen, die Rolle des vernetzten Fahrens, besonders der *Car2Infrastructure Communication*, im Verkehr der Zukunft zu erörtern.
Besonders wichtig ist weiterhin, dass die Ergebnisse der Forschung in erster Linie den nicht-motorisierten Verkehrsteilnehmern zu Gute kommen, da sich der Großteil der Verkehrsforschung um Verbesserungen für den Automobilverkehr dreht.

[1] "kooperatives, hochautomatisiertes Fahren"

1.2 Zielstellung

Das Ziel der Arbeit ist es, einen Prototypen für die automatisierte optische Erfassung und dreidimensionale Ortung von nicht-motorisierten Verkehrsteilnehmern mithilfe von zwei Verkehrskameras zu entwickeln.

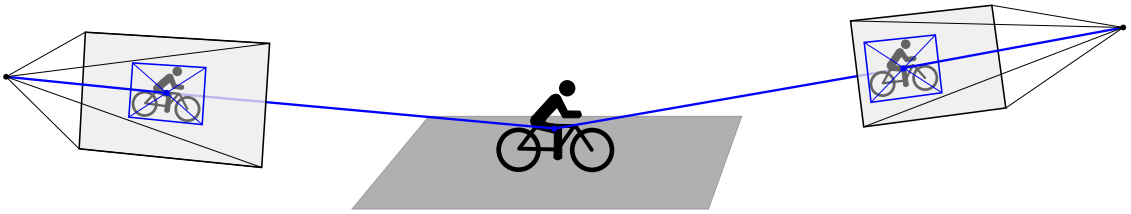


Abbildung 1.1: Grundkonzept der dreidimensionalen Ortung eines Objekts aus der Grundlage von zwei Kamerabildern (*Stereo Vision*)

Die nicht-motorisierten Verkehrsteilnehmer sollen mithilfe eines offen, auf *Deep Learning* basierenden *Object Detection Algorithmus'* erkannt, das heißt detektiert und klassifiziert, werden. Es werden zwei Kameras eingesetzt, in dessen beiden Bildern diese Erkennung fortlaufend stattfindet. Durch die Kenntnis über alle Eigenschaften der beiden Kameras (innere Parameter, genaue Position und Orientierung) lässt sich dann die räumliche Position des gesehenen Verkehrsteilnehmers rückermitteln (en.: *Stereo Vision*). Dazu werden zwei Kameras benötigt, weil unter Nutzung einer einzelnen Kamera noch keine Tiefeninformationen über das gesehene Objekt der Szene ermittelt werden kann, sondern nur ein gerichteter Strahl auf dem dieses sich befinden muss.

Im Mittelpunkt der Arbeit steht das Erarbeiten der für den Prototyp notwendigen mathematischen Grundlagen[2]. Dabei wird in erster Linie die beschriebene dreidimensionale Ortung eines Objekts auf der Grundlage von zwei überlappenden Kamerabildern. Essentiell sind aber auch die Betrachtungen zur Kamerakalibrierung mithilfe von ausgemessenen Kalibrierungspunkten und zum Ordnen der in den Kamerabildern gefundenen Verkehrsteilnehmer zu korrespondierenden Paaren zwischen den Kamerabildern (en.: *Correspondence Matching*).

Aus den Grundlagen ableitend soll im Anschluss das Prototypenkonzept erarbeitet werden, welches die allgemeine Funktionalität des Systems in theoretischer Weise beschreibt.

Weiterhin wird als Anwendungstest des entwickelten Prototyps ein Versuch in einer konkreten Verkehrssituation geplant, durchgeführt und ausgewertet. Hierfür wird der Programmcode in Python implementiert.

Der entwickelte Prototyp soll zukünftig im Rahmen der *Car2Infrastructure Communication* Anwendung finden können, indem motorisierte Verkehrsteilneh-

[2] vgl. Kapitel 2 (*Theoretische Grundlagen*)

mer, wie beispielsweise das Auto, Warninformationen von dem System über nichtmotorisierte Verkehrsteilnehmer in unmittelbarer Nähe erhalten, die sonst leicht übersehen werden können. Durch die genau ermittelten dreidimensionalen Koordinaten könnten die gefährdeten Verkehrsteilnehmer in Autos zum Beispiel mit einem auf die Windschutzscheibe projizierten Overlay markiert werden.

2 Theoretische Grundlagen

2.1 Projektive Geometrie

Die Projektive Geometrie beschäftigt sich als Teilgebiet der Geometrie mit der projektiven Transformation, die eine dreidimensionale Szene auf eine zweidimensionale Ebene abbildet. Die hier geltenden mathematischen Zusammenhänge lassen sich unter der Annahme des Lochkamera-Modells (vgl. 2.1.1 (*Kameramodell*)) auf die Bildentstehung in einer Kamera anwenden.

Unter Einsatz der linearen Algebra und mit dem Zusammenspiel von euklidischen Koordinaten und homogenen Koordinaten (vgl. 2.1.2 (*Homogene Koordinaten*)) lässt sich dieser Prozess der Bildentstehung anschaulich mit mathematischen Gleichungen beschreiben, welche in 2.1.3 (*Bezugssysteme und Bildentstehung*) hergeleitet werden.

Im Unterabschnitt 2.1.4 (*Kamerakalibrierung*) wird dann auf das Verfahren eingegangen, mit dem die unbekannten Parameter des Bildentstehungsprozesses ermittelt werden.

Schließlich wird in 2.1.5 (*Triangulation*) mithilfe der erlangten Grundlagen und Gleichungen der Bildentstehungsprozess umgekehrt betrachtet. Aus einem Bildpunkt wird also der zugehörige Lichtstrahl rekonstruirt. Durch Triangulation mit den zwei verschiedenen Lichtstrahlen der beiden Kameras kann letztendlich auch die Position des gesehenen Objektes bestimmt werden.

Grundlage für alle Betrachtungen dieses Abschnitts stellen [6, 7, 8] dar.

2.1.1 Kameramodell

Als Kameramodell für die Bildentstehung wird in der projektiven Geometrie häufig die Lochkamera betrachtet. Eine Lochkamera besteht aus einer unendlich kleinen Blende und einem Schirm. Es ist eine Vereinfachung, deren Hauptmerkmal es ist, dass alle einfallenden Lichtstrahlen sich in einem zentralen Punkt, dem Projektionszentrum, schneiden und auf dem gesamten Weg vom Gegenstandspunkt zum Bildpunkt gerade verlaufen.

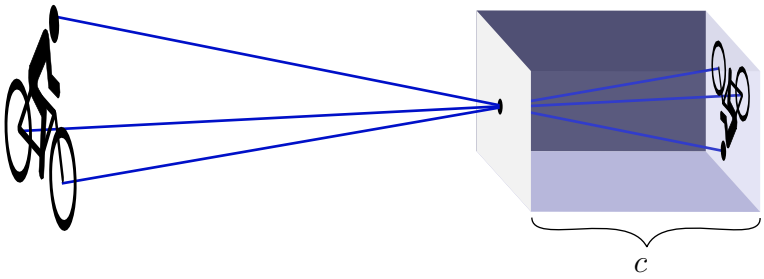


Abbildung 2.1: Lochkamera-Modell;
die Kamerakonstante c stellt den Abstand zwischen Projektionszentrum und Schirm dar

Bei vielen heutigen Kameras ist diese Vereinfachung nur schwach fehlerbehaftet und damit als Modell anwendbar, weil diese über eine kleine Blende, wenn auch nicht unendlich klein, wie die Lochkamera verfügen. Und sogar unter Vernachlässigung der kleinen Blende, sind die Eigenschaften des Lochkameramodells immer noch erfüllt, wenn man das Linsensystem der Kamera zu einer einzigen dünnen Linse vereinfacht und annimmt, dass alle beobachteten Objekten in der scharfgestellten Gegenstands-Ebene liegen. Das Modell weist erst dann grobe Fehler auf, wenn es auf Kameras mit stark Bild-verzerrenden Kameraobjektiven, wie zum Beispiel dem Fischaugenobjektiv, angewendet wird.

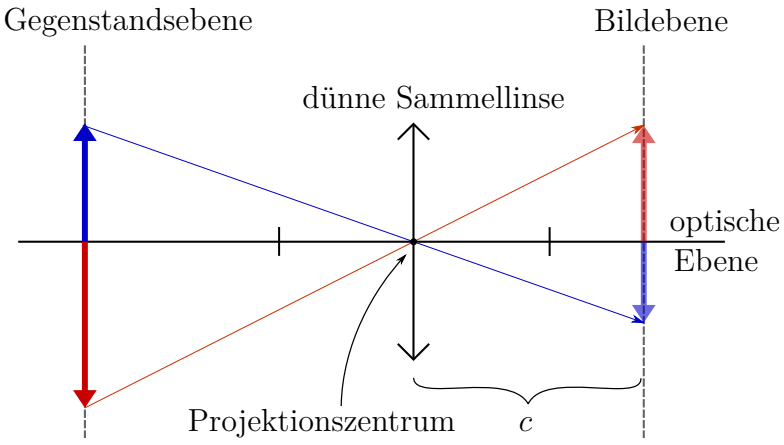


Abbildung 2.2: Anwendbarkeit des Lochkamera-Modells auf eine einfache Linsenkamera, nachvollziehbar über den jeweiligen Mittelpunktstrahl

Daher darf im Weiteren das Modell der Lochkamera genutzt werden und für den Prototyp wird darauf geachtet, diesem Modell möglichst getreue Exemplare zu wählen (vgl. Kapitel 3.2.2 (*Installation und Kalibrierung*)).

Bei einer Lochkamera entsteht das Bild auf dem Schirm, der sich hinter der Blende befindet. Es stellt allerdings eine große Vereinfachung für die Vorstellung

und die Gleichungen dar, wenn man sich den Schirm in gleichem Abstand zur Blende, aber vor dieser, vorstellt. Dann wäre das Bild auf diesem Schirm nämlich aufrecht und nicht seitenverkehrt.

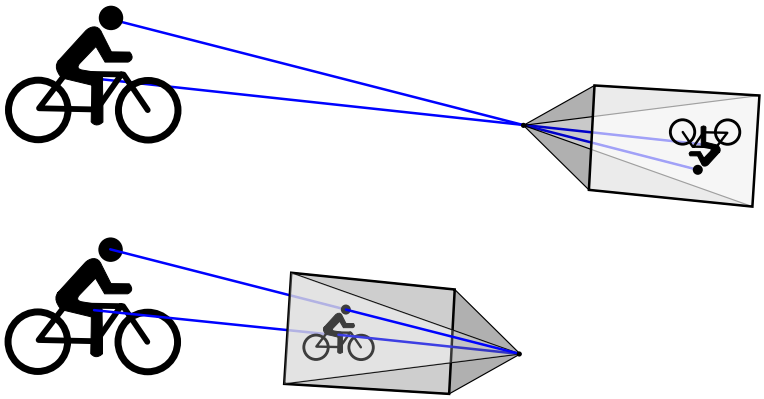


Abbildung 2.3: Normales Lochkameramodell (oben) und vereinfachtes Lochkameramodell mit umgeklappter Bildebene (unten)

2.1.2 Homogene Koordinaten

Im Rahmen dieser Arbeit wird grundsätzlich zwischen zwei Typen von Koordinaten unterschieden, den *euklidischen Koordinaten* und den *homogenen Koordinaten*.

Schreibweise:	$\mathbb{R}^2$ (2D)	$\mathbb{R}^3$ (3D)
Punkt in euklidischen Koordinaten	$\mathbf{x}$	$\mathbf{X}$
Punkt in homogenen Koordinaten	$\mathbf{x}$	$\mathbf{X}$

Euklidische Koordinaten sind die allgemein geläufigen, oft auch als "normal" bezeichneten, Koordinaten, die einen Punkt im n-dimensionalen Raum eindeutig durch n Koordinaten identifizieren.

$$\mathbf{x} = \begin{pmatrix} x \\ y \end{pmatrix} \qquad \mathbf{X} = \begin{pmatrix} X \\ Y \\ Z \end{pmatrix} \tag{2.1}$$

Bei den homogenen Koordinaten handelt es sich um eine alternative Darstellungsweise, bei dem im n-dimensionalen Raum nun $n+1$ Koordinaten verwendet werden, stets eine mehr als bei den euklidischen Koordinaten.

$$\mathbf{x} = \begin{pmatrix} u \\ v \\ w \end{pmatrix} \qquad \mathbf{X} = \begin{pmatrix} U \\ V \\ W \\ T \end{pmatrix} \tag{2.2}$$

Die ersten n davon stellen ein beliebiges Vielfaches der n euklidischen Koordinaten dar, während die spezielle n+1-te homogene Koordinate den Faktor dieses Vielfachen angibt.

Eine Umformung der euklidischen Koordinaten eines Punktes in seine homogenen Koordinaten ist damit in jedem Fall möglich.

$$\begin{aligned}
\boldsymbol{x}: \quad \mathbf{x} = \begin{pmatrix} x \\ y \end{pmatrix} \quad &\Longrightarrow \quad \mathbf{x} = \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} \overset{\text{bzw.}}{=} \begin{pmatrix} \lambda x \\ \lambda y \\ \lambda \end{pmatrix} \\
\boldsymbol{\mathcal{X}}: \quad \mathbf{X} = \begin{pmatrix} X \\ Y \\ Z \end{pmatrix} \quad &\Longrightarrow \quad \mathbf{X} = \begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix} \overset{\text{bzw.}}{=} \begin{pmatrix} \lambda X \\ \lambda Y \\ \lambda Z \\ \lambda \end{pmatrix} \quad \begin{pmatrix} \lambda \in \mathbb{R}, \\ \lambda \neq 0 \end{pmatrix}
\end{aligned} \tag{2.3}$$

Im allgemeinen Fall kann ein und derselbe Punkt also durch unendlich viele verschiedene Zahlenwerte von homogenen Koordinaten beschrieben werden, die jeweils einem der beliebigen Vielfachen entsprechen. Alle verschiedenen homogenen Koordinaten-Tupel, die den gleichen Punkt beschreiben, werden als äquivalent betrachtet. Damit gilt allgemein bei homogenen Koordinaten:

$$\mathbf{x} = \lambda\mathbf{x} \quad \text{bzw.} \quad \mathbf{X} = \lambda\mathbf{X} \qquad \big(\lambda \in \mathbb{R},\ \lambda \neq 0\big) \tag{2.4}$$

Dies ist ein grundlegender Unterschied zu den euklidischen Koordinaten bei denen ein Vielfaches im allgemeinen Fall nicht mehr den selben Punkt beschreibt.

Die Umwandlung der homogenen Koordinaten eines Punktes in seine euklidischen Koordinaten ergibt sich, indem die homogenen Koordinaten in die einfache Form gebracht werden und letzte Koordinate entfällt.

$$\begin{aligned}
\boldsymbol{x}: \quad \mathbf{x} = \begin{pmatrix} u \\ v \\ w \end{pmatrix} = {}^{1}\!/_{w} \begin{pmatrix} u \\ v \\ w \end{pmatrix} = \begin{pmatrix} {}^{u}\!/_{w} \\ {}^{v}\!/_{w} \\ 1 \end{pmatrix} \quad &\Longrightarrow \quad \mathbf{x} = \begin{pmatrix} {}^{u}\!/_{w} \\ {}^{v}\!/_{w} \end{pmatrix} \\
\boldsymbol{\mathcal{X}}: \quad \mathbf{X} = \begin{pmatrix} U \\ V \\ W \\ T \end{pmatrix} = {}^{1}\!/_{T} \begin{pmatrix} U \\ V \\ W \\ T \end{pmatrix} = \begin{pmatrix} {}^{U}\!/_{T} \\ {}^{V}\!/_{T} \\ {}^{W}\!/_{T} \\ 1 \end{pmatrix} \quad &\Longrightarrow \quad \mathbf{X} = \begin{pmatrix} {}^{U}\!/_{T} \\ {}^{V}\!/_{T} \\ {}^{W}\!/_{T} \end{pmatrix}
\end{aligned} \tag{2.5}$$

Der größte Vorteil[1] der Punkt-Darstellung durch homogene Koordinaten liegt in der Vereinfachung mathematischer Gleichungen in der projektiven Geometrie. Für diese Arbeit bedeutsam ist hier vor allem die erweiterte Funktionalität von Abbildungsmatrizen. Es lässt sich zum Beispiel die Translation eines Punktes im zweidimensionalen Raum mit einer solchen Abbildungsmatrix durchführen, die bei euklidischen Koordinaten nur mit der Addition eines Verschiebungsvektors möglich

[1] auf diese Forschungsarbeit bezogen; weitere, in der Projektiven Geometrie sehr wichtigen, Vorteile der Verwendung homogener Koordinaten werden im Rahmen dieser Arbeit nicht betrachtet

wäre.

$$\text{euklidische Koordinaten:} \quad \mathbf{x}' = \mathbf{x} + \begin{pmatrix} \Delta x \\ \Delta y \end{pmatrix} \tag{2.6}$$

$$\text{homogene Koordinaten:} \quad \mathbf{x}' = \begin{pmatrix} 1 & 0 & \Delta x \\ 0 & 1 & \Delta y \\ 0 & 0 & 1 \end{pmatrix} \cdot \mathbf{x} \tag{2.7}$$

Bei homogenen Koordinaten kann man sogar mit einer einzigen Abbildungsmatrix eine ganze projektive Transformation durchführen[2]. Für den Rahmen dieser Arbeit reicht es aber zu wissen, wie der allgemeine Aufbau einer Abbildungsmatrix für eine Translation, Rotation und Achsenstreckung aussieht.

$$\boldsymbol{H} = \begin{pmatrix} \boldsymbol{AR} & \boldsymbol{t} \\ 0^T & 1 \end{pmatrix} \tag{2.8}$$

$$\text{Mit:} \qquad \boldsymbol{A}_{2\times 2} = \begin{pmatrix} a_1 & 0 \\ 0 & a_2 \end{pmatrix}, \quad \boldsymbol{R}_{2\times 2} = \begin{pmatrix} r_1 & r_2 \\ r_3 & r_4 \end{pmatrix}, \quad \boldsymbol{t}_{2\times 1} = \begin{pmatrix} t_1 \\ t_2 \end{pmatrix}$$

$$\text{bzw:} \qquad \boldsymbol{A}_{3\times 3} = \begin{pmatrix} a_1 & 0 & 0 \\ 0 & a_2 & 0 \\ 0 & 0 & a_3 \end{pmatrix}, \quad \boldsymbol{R}_{3\times 3} = \begin{pmatrix} r_1 & r_2 & r_3 \\ r_4 & r_5 & r_6 \\ r_7 & r_8 & r_9 \end{pmatrix}, \quad \boldsymbol{t}_{3\times 1} = \begin{pmatrix} t_1 \\ t_2 \\ t_3 \end{pmatrix}$$

$$a_i, r_i, t_i \in \mathbb{R}$$

Die Abbildungsmatrix H entspricht hierbei einer Rotation des Punktes um den Koordinatenursprung gemäß der Drehmatrix R mit anschließender Achsenstreckung nach A und abschließender Verschiebung entsprechend des euklidischen Vektors t.

Dadurch, dass sich nun alle benötigten Transformationen mit Abbildungsmatrizen beschreiben lassen, werden Transformationen simpel verkettbar (durch linksseitige Matrixmultiplikation: $H' = H_2 H_1$) und in manchen Fällen umkehrbar (durch die Bildung der Inverse: $H' = H^{-1}$).

Weil die Multiplikation von homogenen Koordinaten mit λ ($\lambda \in \mathbb{R}$, $\lambda \neq 0$) eine Äquivalenzumformung darstellt (vgl. Gleichung 2.4), gilt auch für die Abbildungsmatrizen homogener Koordinaten:

$$\boldsymbol{H} = \lambda \boldsymbol{H} \tag{2.9}$$

2.1.3 Bezugssysteme und Bildentstehung

Der Prozess der Bildentstehung wird durch mathematische Gleichungen nachvollzogen, indem der beobachtete Punkt p aus verschiedenen Perspektiven bzw. Bezugssystemen betrachtet wird. Mit Abbildungsmatrizen lässt sich zwischen den Bezugssystemen wechseln.

[2] all diese Operationen sind auch mit euklidischen Koordinaten durchführbar, jedoch stets mit erheblich höherem Aufwand verbunden

Zunächst betrachtet man das dreidimensionale *Welt-Bezugssystem* (Index w) in dem sich p und die Kamera an bestimmter Position und mit bestimmter Orientierung befinden.
Durch eine Translation und eine Drehung lässt sich der gleiche Punkt p nun aus dem dreidimensionalen *Kamera-Bezugssystem* (Index k) betrachten, dessen Koordinatenursprung auf dem Projektionszentrum der Kamera liegt. Die Z-Achse zeigt senkrecht nach vorne aus der Kamera, die x-y-Ebene liegt parallel zur Kamera-Bildebene.
Als nächstes wird in das *Bildebenen-Bezugssystem* (Index b) gewechselt, welches nur noch zweidimensional ist. Die x-y-Ebene liegt genau auf der Bildebene der Kamera mit dem Koordinatenursprung zentral auf dem Sensor, als auf der Z-Achse des Kamera-Bezugssystems. Die Koordinaten geben jetzt an, wo p auf dieser Bildebene zusehen ist.
Im letzten Schritt soll die Position des Punktes p aus dem *Sensor-Bezugssystem* (Index s) angegeben werden. Die x-y-Ebene des zwei-dimensionalen Bezugssystems liegt immer noch auf der Bildebene der Kamera, allerdings sollen die Koordinaten jetzt den letztlich relevanten Bildpunkt-Einheiten (Pixel) entsprechen. Dazu muss, ausgehend von den Koordinaten des Bildebenen-Bezugssystems, eine Verschiebung des Koordinatenursprungs und eine Achsenstreckung stattfinden.

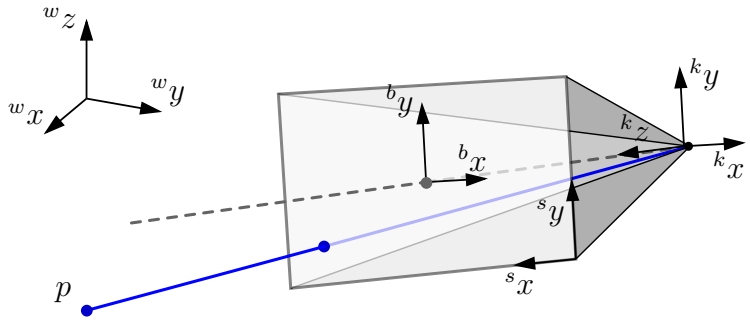


Abbildung 2.4: Übersicht zum Welt-Bezugssystem (w), Kamera-Bezugssystem (k), Bildebenen-Bezugssystem (b), Sensor-Bezugssystem (s)

Die oben beschriebenen Transformationen werden nun mit Abbildungsmatrizen für homogene Koordinaten beschrieben. Die Notation ${}^yH_x$ meint dabei den Wechsel vom Bezugsystem x zum Bezugssystem y. ${}^y\mathrm{X}_p$ meint die homogenen Koordinaten des Punktes p im Bezugssystem y.
Ausgegangen wird vom Punkt p in homogenen Koordinaten: ${}^w\mathrm{X}_p = (X, Y, Z, 1)^T$.

${}^k\boldsymbol{H}_w$: Zur Verschiebung des Koordinatenursprungs auf das Kameraprojektionszentrum werden die euklidischen Weltkoordinten X_0 des Kameraprojektionszentrums von den Weltkoordinaten von p subtrahiert. Erst anschließend findet eine Drehung um den neuen Koordinatenursprung mit der noch nicht weiter spezifizier-

ten Drehmatrix $R_{3\times3}$ statt.

$$ {}^{k}\mathbf{X}_p = \begin{pmatrix} \boldsymbol{R} & 0 \\ 0^T & 1 \end{pmatrix} \begin{pmatrix} E_3 & -\mathbf{X}_0 \\ 0^T & 1 \end{pmatrix} \cdot {}^{w}\mathbf{X}_p \tag{2.10} $$

$$ {}^{k}\boldsymbol{H}_w = \begin{pmatrix} \boldsymbol{R} & 0 \\ 0^T & 1 \end{pmatrix} \begin{pmatrix} E_3 & -\mathbf{X}_0 \\ 0^T & 1 \end{pmatrix} = \begin{pmatrix} \boldsymbol{R} & -\boldsymbol{R}\mathbf{X}_0 \\ 0^T & 1 \end{pmatrix} \tag{2.11} $$

${}^{b}\boldsymbol{H}_k$: Diese Abbildung ist besonders, da von einem dreidimensionalem Bezugssystem in ein zwei-dimensionales gewechselt wird. Gesucht ist daher eine 3×4 -Matrix. Zur Umsetzung wird der Strahlensatz herangezogen und man nehme sich zur Herleitung wiederum die euklidischen Koordinaten von p vor: ${}^{k}\mathsf{X}_p = (X,\, Y,\, Z)^T$. Aus Sicht der Kamera[3] handelt es sich nämlich um eine zentrische Streckung, bei der p auf der Z-Achse von Z auf c [4] gestaucht wird. Im gleichen Verhältnis wie c zu Z stehen auch die neuen x- und y-Bildebenen-Koordinaten zu ihren entsprechenden X- und Y-Koordinaten im Kamerabezugssystem. X und Y müssen daher mit dem Faktor $\frac{c}{Z}$ multipliziert werden, um auf die Bildebene projeziert zu werden. Z entfällt, da das Bildebenen-Bezugssystem nur noch zwei Dimensionen hat. Genau das wird, wieder in homogenen Koordinaten ausgedrückt, durch die folgende Abbildungsmatrix erreicht:

$$ {}^{b}\boldsymbol{H}_k = \begin{pmatrix} c & 0 & 0 & 0 \\ 0 & c & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} \tag{2.12} $$

Denn:

$$ \begin{pmatrix} c & 0 & 0 & 0 \\ 0 & c & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix} = \begin{pmatrix} cX \\ cY \\ Z \end{pmatrix} = \begin{pmatrix} \frac{c}{Z}\cdot X \\ \frac{c}{Z}\cdot Y \\ 1 \end{pmatrix} $$

${}^{s}\boldsymbol{H}_b$: Der Koordinatenursprung wird auf den Ursprung des Pixel-Koordinatensystems verlegt, indem der euklidische Ortsvektor des neuen Ursprungs $(x_s,\, y_s)^T$ von ${}^{b}\mathbf{x}_p$ subtrahiert wird. Dann findet eine Achsenstreckung mit den beiden Faktoren a_1 und a_2 statt, die von der Pixelgröße auf dem Sensor abhängen. Bei quadratischen Pixeln ist a_1 gleich a_2.

$$ {}^{s}\boldsymbol{H}_b = \begin{pmatrix} a_1 & 0 & 0 \\ 0 & a_2 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & -x_s \\ 0 & 1 & -y_s \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} a_1 & 0 & -a_1 x_s \\ 0 & a_2 & -a_2 y_s \\ 0 & 0 & 1 \end{pmatrix} \tag{2.13} $$

Die erhaltenen Abbildungsmatrizen der einzelnen Transformationen lassen sich nun noch sinnvoll vereinen. Dabei werden alle Abbildungsmatrizen aneinandergereiht, um schließlich die gesamte Projektion vom Weltpunkt ${}^{w}\mathrm{X}_p$ zum Bildpunkt

[3] gemeint ist das Kamera-Bezugssystem

[4] c ist die Kamerakonstante, siehe 2.1.1 (*Kameramodell*)

${}^s\mathrm{x}_p$ darzustellen. Das Produkt aller drei Abbildungsmatrizen wird mit P (*Projektionsmatrix*) bezeichnet.

$$\begin{aligned} {}^s\mathbf{x}_p &= \boldsymbol{P} \cdot {}^w\mathbf{X}_p && (2.14)\\ &= {}^s\boldsymbol{H}_b\, {}^b\boldsymbol{H}_k\, {}^k\boldsymbol{H}_w \cdot {}^w\mathbf{X}_p && (2.15) \end{aligned}$$

Bei dieser Zusammenfassung der Transformtionen können Vereinfachung vorgenommen werden, zunächst bei dem Produkt aus ${}^bH_k$ und ${}^kH_w$.

$$\begin{aligned} {}^b\boldsymbol{H}_k\, {}^k\boldsymbol{H}_w &= \begin{pmatrix} c & 0 & 0 & \not{0} \\ 0 & c & 0 & \not{0} \\ 0 & 0 & 1 & \not{0} \end{pmatrix} \begin{pmatrix} \boldsymbol{R} & -\boldsymbol{R}\mathbf{X}_0 \\ \not{0}^T & \not{1} \end{pmatrix} \\ &= \begin{pmatrix} c & 0 & 0 \\ 0 & c & 0 \\ 0 & 0 & 1 \end{pmatrix} \left(\boldsymbol{R} \mid -\boldsymbol{R}\mathbf{X}_0 \right) && (2.16) \end{aligned}$$

Es ergeben sich die beiden neuen Abbildungsmatrizen ${}^bH'_k$ und ${}^kH'_w$.

$${}^b\boldsymbol{H}'_k = \begin{pmatrix} c & 0 & 0 \\ 0 & c & 0 \\ 0 & 0 & 1 \end{pmatrix} \qquad {}^k\boldsymbol{H}'_w = \left(\boldsymbol{R} \mid -\boldsymbol{R}\mathbf{X}_0 \right) \tag{2.17}$$

Weiterhin wird noch das Produkt von ${}^sH_b$ und ${}^bH'_k$ als K zusammengefasst.

$$\begin{aligned} \boldsymbol{K} = {}^s\boldsymbol{H}_b {}^b\boldsymbol{H}'_k &= \begin{pmatrix} a_1 & 0 & -a_1 x_s \\ 0 & a_2 & -a_2 y_s \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} c & 0 & 0 \\ 0 & c & 0 \\ 0 & 0 & 1 \end{pmatrix} \\ \boldsymbol{K} &= \begin{pmatrix} ca_1 & 0 & -a_1 x_s \\ 0 & ca_2 & -a_2 y_s \\ 0 & 0 & 1 \end{pmatrix} && (2.18) \end{aligned}$$

K wird *Kalibrierungsmatrix* genannt, weil sie allein nun alle inneren Kameraparameter[5] beinhaltet (c, a_1, a_2, x_s, y_s). Der restliche Teil von P enthält alle sogenannten äußeren Kameraparameter[6] ($\mathsf{X_0}$, R).

$$\begin{aligned} \boldsymbol{P} &= {}^s\boldsymbol{H}_b\, {}^b\boldsymbol{H}'_k\, {}^k\boldsymbol{H}'_w \\ &= \boldsymbol{K}\, {}^k\boldsymbol{H}'_w \\ &= \boldsymbol{K} \left(\boldsymbol{R} \mid -\boldsymbol{R}\mathbf{X}_0 \right) \\ \boldsymbol{P} &= \boldsymbol{K}\, \boldsymbol{R} \left(E_3 \mid -\mathbf{X}_0 \right) && (2.19) \end{aligned}$$

[5] die für eine Kamera spezifischen Parameter, die nicht von der Orientierung der Kamera im Raum abhängig sind

[6] die Parameter einer Kamera, die ihre Orientierung im Raum beschreiben bzw. davon abhängig sind

2.1.4 Kamerakalibrierung

Bevor man in 2.1.5 (*Triangulation*) schließlich den Bildentstehungsprozess umkehren kann, müssen alle inneren (K) und äußeren (X_0, R) Parameter beider Kameras bekannt sein. Sie werden durch eine Kalibrierung erlangt. Dazu soll der sogenannte Algorithmus *Direct Linear Transform* (*DLT*) verwendet werden. Er bezieht sich direkt auf die schon hergeleiteten Gleichungen aus 2.1.3 (*Bezugssysteme und Bildentstehung*) und eignet sich deswegen sehr gut, um nicht den Rahmen dieser Arbeit auszureizen. Die Kalibrierung wird für jede Kamera individuell durchgeführt.

Als Grundlage für die Kalibrierung benötigt der *DLT*-Algorithmus ein Bild der zu kalibrierenden Kamera, sowie die Position einiger (allgemein: I Stück) spezieller Kalibrierungs-Punkte auf diesem Bild und die entsprechenden Welt-Koordinaten dieser. Es gilt dann zunächst die gesamte Projektionsmatrix $P_{3\times4}$ näherungsweise zu ermitteln, im Anschluss lassen durch das Wissen über ihre Zusammensetzung alle benötigten inneren und äußeren Parameter extrahieren.

Die Projektionsmatrix P wird zunächst allgemein notiert.

$$\boldsymbol{P} = \begin{pmatrix} p_1 & p_2 & p_3 & p_4 \\ p_5 & p_6 & p_7 & p_8 \\ p_9 & p_{10} & p_{11} & p_{12} \end{pmatrix} \tag{2.20}$$

X und x seien die homogenen Koordinaten eines Kalibrierungs-Punktes in der Welt und auf dem Sensor.

$$\begin{aligned} \mathbf{x} &= \boldsymbol{P} \cdot \mathbf{X} \\ \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} &= \begin{pmatrix} \begin{pmatrix} p_1 & p_2 & p_3 & p_4 \end{pmatrix} \cdot \mathbf{X} \\ \begin{pmatrix} p_5 & p_6 & p_7 & p_8 \end{pmatrix} \cdot \mathbf{X} \\ \begin{pmatrix} p_9 & p_{10} & p_{11} & p_{12} \end{pmatrix} \cdot \mathbf{X} \end{pmatrix} \end{aligned} \tag{2.21}$$

Mithilfe der Umformung von $\mathbf{x}$ zu den euklidischen Koordinaten entsprechend Gleichung 2.5 ergeben sich für x und y des Kalibrierungspunkts auf dem Sensor:

$$x = \frac{\begin{pmatrix} p_1 & p_2 & p_3 & p_4 \end{pmatrix} \cdot \mathbf{X}}{\begin{pmatrix} p_9 & p_{10} & p_{11} & p_{12} \end{pmatrix} \cdot \mathbf{X}} \qquad y = \frac{\begin{pmatrix} p_5 & p_6 & p_7 & p_8 \end{pmatrix} \cdot \mathbf{X}}{\begin{pmatrix} p_9 & p_{10} & p_{11} & p_{12} \end{pmatrix} \cdot \mathbf{X}} \tag{2.22}$$

Durch jeden Kalibrierungspunkt erhält man also zwei Gleichungen. Aus diesen wird im Folgenden ein Gleichungssystem zusammengesetzt, in dem nur $p_1, \ldots, p_{12}$ unbekannt sind, diese sollen ja ermittelt werden. Zunächst müssen die beiden Gleichungen aus (2.22) umgeformt werden.

$$\begin{aligned} -\mathbf{X}^T \begin{pmatrix} p_1 \\ p_2 \\ p_3 \\ p_4 \end{pmatrix} \qquad\qquad + x \cdot \mathbf{X}^T \begin{pmatrix} p_9 \\ p_{10} \\ p_{11} \\ p_{12} \end{pmatrix} &= 0 \\ -\mathbf{X}^T \begin{pmatrix} p_5 \\ p_6 \\ p_7 \\ p_8 \end{pmatrix} + y \cdot \mathbf{X}^T \begin{pmatrix} p_9 \\ p_{10} \\ p_{11} \\ p_{12} \end{pmatrix} &= 0 \end{aligned} \tag{2.22a}$$

Das Gleichungsystem soll als Matrizengleichung formuliert werden. Aus diesem Grund werden alle $p_1, \ldots, p_{12}$ zu einem Lösungvektor $p_{12\times 1}$ zusammengefasst.

$$\boldsymbol{p} = \begin{pmatrix} p_1 & p_2 & p_3 & \ldots & p_{12} \end{pmatrix}^T \tag{2.23}$$

Die Gleichungen von (2.22a) werden mit p umgeschrieben zu:

$$\begin{array}{rcl} \left(\quad -\mathbf{X}^T, \quad 0_4^T, \quad x\mathbf{X}^T \quad \right) \cdot \boldsymbol{p} & = & 0 \\ \left(\quad 0_4^T, \quad -\mathbf{X}^T, \quad y\mathbf{X}^T \quad \right) \cdot \boldsymbol{p} & = & 0 \end{array} \tag{2.22b}$$

Das Gleichungssystem in Matrixschreibweise ergibt sich auf triviale Weise dann folgendermaßen:

$$\begin{pmatrix} & \vdots & \\ -\mathbf{X}_i^T & 0_4^T & x_i\mathbf{X}_i^T \\ 0_4^T & -\mathbf{X}_i^T & y_i\mathbf{X}_i^T \\ & \vdots & \end{pmatrix} \cdot \boldsymbol{p} = 0 \qquad (0 \leq i \leq I) \tag{2.22c}$$

$$\boldsymbol{M}_{2I\times 12} \cdot \boldsymbol{p} = 0$$

Für eine eindeutige Lösung von P würden schon 6 Kalibrierungspunkte ausreichen, weil dann die 12 Unbekannten von $2I = 12$ Gleichungen gedeckt wären[7]. Allerdings werden für eine höhere Genauigkeit am besten mehr als 6 Kalibrierungspunkte verwendet und das überbestimmte Gleichungssystem dann mit der *Methode der kleinsten Quadrate*[8] gelöst.

Wenn die Koordinaten aller Kalibrierungspunkte mathematisch exakt wären, würde nun auch das überbestimmte Gleichungssystem eine eindeutige Lösung haben. Weil aber diese Annahme beim Kalibrierungsvorgang in der Regel wegen zufälligen, minimalen Messungenauigkeiten nicht zutrifft, hat das Gleichungssystem durch die Überbestimmtheit eigentlich gar keine Lösung. In jedem Fall verbleibt ein Ergebnisvektor ω, der leicht von 0 abweicht.

$$\boldsymbol{M} \cdot \boldsymbol{p} = \boldsymbol{\omega} \neq 0 \tag{2.24}$$

$$\boldsymbol{\omega}^T\boldsymbol{\omega} = \boldsymbol{\Omega} \tag{2.25}$$

Mit der schon genannten Methode der kleinsten Quadrate lässt sich diese Fehler-Quadratsumme Ω aber minimieren. Ohne auf die mathematischen Hintergründe genauer einzugehen, lässt sich zusammenfassen, dass hierbei eine Singulärwertzerlegung (vgl. Gleichung 2.26) auf die Matrix M angewendet wird und anschließend der Rechts-Singulärvektor von M, der dem kleinsten Singulärwert zugehörig ist, die beste Näherungslösung darstellt.

Es muss darauf geachtet werden, dass die genauen Elementwerte einer Abbildungsmatrix für homogene Koordinaten wegen der beliebigen Skalierbarkeit (vgl. Gleichung 2.9) uneindeutig sind. Das gilt auch für die gesuchte Matrix P. Deswegen

[7] genaugenommen würden sogar noch weniger Gleichungen genügen, was nicht ausführlicher betrachtet wird

[8] eine Näherungslösung eines überbestimmten Gleichungssystem wird durch Minimierung der Fehlerquadrate erhalten

wird ohne Beschränkung der Allgemeinheit willkürlich festgelegt, dass $|p| = 1$ gilt. Dadurch wird auch gleich die triviale, aber nicht gesuchte, Lösung $p = 0$ formal ausgeschlossen.
Jedoch sind diese beiden Bedingungen nur in der Theorie von Bedeutung, damit es aus rein mathematischer Sicht überhaupt eine eindeutige Lösung geben kann. In der praktischen Umsetzung mit der gerade beleuchteten Singulärwertzerlegung entfällt die Lösung $p = 0$ ohnehin und es wird trotz der theoretisch unumgänglichen Uneindeutigkeit von P eine beste Näherungslösung für p gefunden, weil die Singulärwertzerlegung die skalare Uneindeutigkeit von P nicht einbezieht. Die gefundene Näherungslösung erfüllt dann im allgemeinen Fall eben nicht genau $|p| = 1$, aber stellt einfach ein Vielfaches dieser, theoretisch auf die Norm 1 festgelegten, Zahlenwerte dar und ist damit als Lösung äquivalent.

Singulärwertzerlegung:
$$\underset{2I\times 12}{\boldsymbol{M}} = \underset{2I\times 12}{\boldsymbol{U}} \cdot \underset{12\times 12}{\boldsymbol{S}} \cdot \underset{12\times 12}{\boldsymbol{V}^T} \tag{2.26}$$

$\boldsymbol{U}$:	orthogonal,	beinhaltet Links-Singulärvektoren als Spalten
$\boldsymbol{S}$:	Diagonalmatrix,	Hauptdiagonale besteht aus positiven Singulärwerten in absteigender Reihenfolge
$\boldsymbol{V^T}$:	orthogonal,	beinhaltet Rechts-Singulärvektoren als Zeilen

Der kleinste Singulärwert ist damit s_{12}, der letzte Wert auf der Hauptdiagonalen von S. Und der zugehörige Rechts-Singulärvektor ist v_{12}, die letzte Zeile von V^T bzw. die letzte Spalte von V. Nach der Methode der kleinsten Quadrate stellt v_{12} damit, wie schon erläutert, die beste Näherungslösung für p dar.

$$\widehat{\boldsymbol{p}} = \boldsymbol{v_{12}} = \begin{pmatrix} \widehat{p}_1 & \widehat{p}_2 & \widehat{p}_3 & \dots & \widehat{p}_{12} \end{pmatrix}^T \tag{2.27}$$

$$\widehat{\boldsymbol{P}} = \begin{pmatrix} \widehat{p}_1 & \widehat{p}_2 & \widehat{p}_3 & \widehat{p}_4 \\ \widehat{p}_5 & \widehat{p}_6 & p_7 & \widehat{p}_8 \\ \widehat{p}_9 & \widehat{p}_{10} & \widehat{p}_{11} & \widehat{p}_{12} \end{pmatrix} \tag{2.28}$$

Aus der erlangten Projektionsmatrix $\widehat{P}$ lassen sich nun ausgehend von dem allgemeinen Aufbau einer Projketionsmatrix P entsprechend Gleichung 2.19 die benötigten Parameter $\widehat{K}$, $\widehat{\mathsf{X}}_0$ und $\widehat{R}$ extrahieren.

$$\widehat{\boldsymbol{P}} = \widehat{\boldsymbol{K}}\,\widehat{\boldsymbol{R}}\left(E_3 \mid -\hat{\mathbf{X}}_0\right) \tag{2.29}$$
$$= \left(\widehat{\boldsymbol{K}}\widehat{\boldsymbol{R}} \mid -\widehat{\boldsymbol{K}}\widehat{\boldsymbol{R}}\cdot\hat{\mathbf{X}}_0\right)$$

mit:
$$\boldsymbol{H}_{3\times3} = \widehat{\boldsymbol{K}}\widehat{\boldsymbol{R}} \tag{2.30}$$
$$\boldsymbol{h}_{3\times1} = -\widehat{\boldsymbol{K}}\widehat{\boldsymbol{R}}\cdot\hat{\mathbf{X}}_0 \tag{2.31}$$

Aus den Bekannten H und h lässt sich als erstes $\hat{\mathsf{X}}_0$ ermitteln.

aus (2.30) und (2.31) folgt:
$$\boldsymbol{h} = -\boldsymbol{H}\cdot\hat{\mathbf{X}}_0$$
$$\hat{\mathbf{X}}_0 = (-\boldsymbol{H})^{-1}\cdot\boldsymbol{h}$$

Um $\widehat{K}$ und $\hat{R}$ zu extrahieren zieht man die QR-Zerlegung an, die eine beliebige Matrix $\widetilde{A}_{3\times3}$ vom Rang 3 in das Produkt einer orthogonalen Matrix $\widetilde{Q}_{3\times3}$ und einer oberen Dreiecksmatrix $\widetilde{R}_{3\times3}$ aufspalten kann.

QR-Zerlegung:
$$\widetilde{A}_{3\times3} = \widetilde{Q}_{3\times3}\cdot\widetilde{R}_{3\times3} \tag{2.32}$$

Weil $\hat{R}$ eine Drehmatrix, und damit orthogonal, und $\widehat{K}$ eine obere Dreiecksmatrix ist, lässt sich die QR-Zerlegung geschickt ausnutzen. Allerdings beeinhaltet H die beiden Matrizen in verkehrter Reihenfolge, um die QR-Zerlegung direkt auf H anzuwenden. Es wird daher zuerst die Inverse von H gebildet.

$$\boldsymbol{H}^{-1} = (\widehat{\boldsymbol{K}}\widehat{\boldsymbol{R}})^{-1} = \widehat{\boldsymbol{R}}^{-1}\widehat{\boldsymbol{K}}^{-1} = \widehat{\boldsymbol{R}}^{T}\widehat{\boldsymbol{K}}^{-1}$$

Auf H^{-1} kann dann die QR-Zerlegung angewendet werden, da $\hat{R}^T$ immernoch orthogonal ist, $\widehat{K}^{-1}$ immer noch eine obere Dreiecksmatrix ist und beide jetzt in der richtigen Reihenfolge vorliegen. Man erhält $\hat{R}$ sowie $\widehat{K}$ durch Bildung der transponierten Matrix bzw. der Inverse.

QR-Zerlegung:
$$\boldsymbol{H}^{-1} \Longrightarrow \widehat{\boldsymbol{R}}^{T},\quad \widehat{\boldsymbol{K}}^{-1}$$
$$()^T\downarrow \qquad \downarrow()^{-1}$$
$$\widehat{\boldsymbol{R}} \qquad \widehat{\boldsymbol{K}}$$

Zur Kamerakalibrierung muss abschließend noch betrachtet werden, dass es bestimmte kritische Konfigurationen gibt. Das sind räumliche Lagebeziehungen zwischen den Kalibrierungspunkten, die vermieden werden müssen, weil eine Kalibrierung sonst nicht durchführbar wäre. Nach [9, S. 537] lassen sich diese kritischen Konfigurationen zu sechs allgemeinen Fällen zusammenfassen.

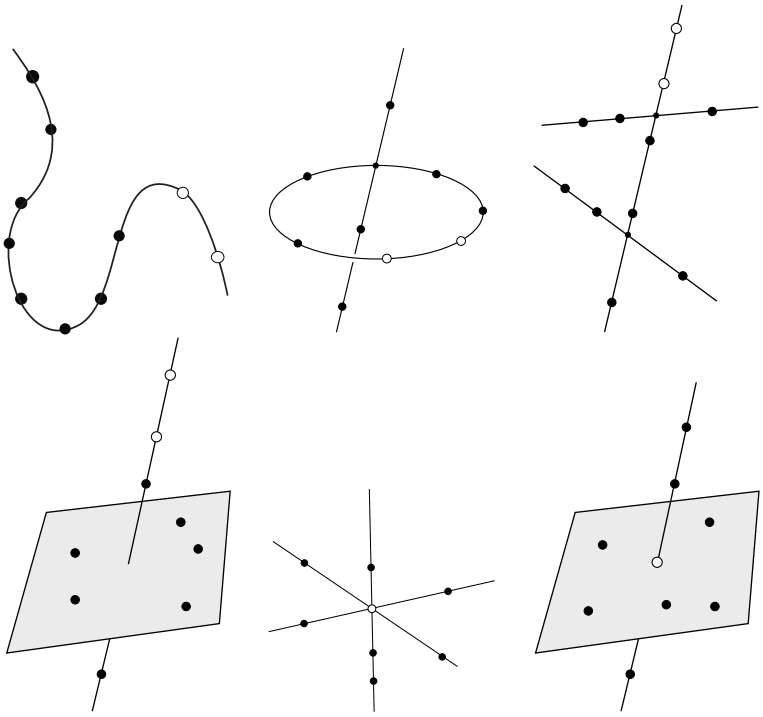


Abbildung 2.5: Sechs kritische Konfigurationen von Kalibrierungspunkten nach [9, S. 537], Quelle: [9, S. 538]

In diesen Fällen ist die Kameraperspektive rein mathematisch uneindeutig. Wenn die Koordinaten der Kalibrierungspunkte diese Beziehungen exakt erfüllen, ist gar keine Lösung für P möglich. Auch hier gilt wieder, dass durch Messungenauigkeiten diese kritischen Lagebeziehungen in der Regel nie genau vorliegen werden. Sie sind trotzdem strengstens zu meiden, denn auch bei einer annähernd kritischen Konfiguration tritt die Uneindeutigkeit in Kraft, indem sie die Messungenauigkeiten enorm amplifiziert. Die erhaltene Projektionsmatrix P wäre dann nahezu willkürlich oder zumindest stark fehlerbehaftet und damit in der Praxis nicht einsetzbar.
Der häufigste Fehler wäre, alle Kalibrierungspunkte auf der Erdoberfläche zu wählen. Sie würden dann alle annähernd in einer Ebene liegen und soeben beschriebene Effekte träten in Kraft. Darauf muss bei der Entwicklung des Prototyps[9] geachtet werden.

2.1.5 Triangulation

Bei der sogenannten *Triangulation* wird ein Obejekt durch die Kenntnis seiner Sensorkoordinaten zweier Kameras räumlich geortet (engl.: *Stereo Vision*). Denn, wie allgemein bekannt, kann eine einzelne Kamera nur Richtungsinformationen und keine Tiefeninformationen über einen gesehenen Gegenstand liefern. Durch zwei in ihrer Position verschiedene Kameras wird die Tiefeninformation aber berechenbar.

[9] Referenz

Als erstes muss der Bildentstehungsprozess umgekehrt werden. Man erhält die Richtungs-Halbgerade, auf der der räumliche Punkt aus Sicht einer der Kameras liegen muss.
Dazu wird der Bildpunkt in Sensorkoordinaten $\mathrm{x} = (x, y, 1)^T$ zunächst mit der Inversen der Kalibrierungsmatrix K multipliziert. Man gelangt formal vom Sensor-Koordinatensystem zum Kamerakoordinatensystem.

$$\boldsymbol{K}^{-1} \cdot {}^{s}\mathbf{x} \tag{2.33}$$

Wichtig ist hier, dass man aufgrund der 3×3-Dimension von K nicht die vier homogenen Koordinaten für die dreidimensionale Position des Punktes erhält[10], sondern nur die ersten drei dieser eigentlich vier homogenen Koordinaten. Ohne die vierte Koordinate kann man die ersten drei nicht zur einfachen Form normalisieren[11] und bleibt mit dem bloßen Verhältnis von X, Y und Z des Punktes aus Sicht der Kamera zurück.

$$\boldsymbol{K}^{-1} \cdot {}^{s}\mathbf{x} \;=\; \lambda\, {}^{k}\mathbf{X} \tag{2.33'}$$

$\lambda\, {}^{k}\mathsf{X}$ entspricht nun der Richtung, in der der Punkt aus Sicht der Kamera liegt, bzw. der angesprochenen Halbgeraden.
Um den Richtungsvektor aus dem Kamera-Koordinatensystem nun noch in das Welt-Koordinatensystem zu übertragen, muss nur die Inverse R^T der entsprechenden Drehmatrix R linksseitig anmultipliziert werden.

$$\boldsymbol{a} = \boldsymbol{R}^T \boldsymbol{K}^{-1} \mathbf{x} \tag{2.34}$$

a stellt jetzt im Welt-Koordinatensystem dar, in welche Richtung vom Kameraprojektionszentrum aus der gesehene Punkt liegt. Die Koordinaten Punktes im Raum F würden sich dann mithilfe der Position des Kameraprojektionszentrums X_0 so angeben lassen:

$$\mathsf{F} = \mathsf{X}_0 + \lambda\, \boldsymbol{a} \tag{2.35}$$

Jetzt werden beide Kameras zusammen betrachtet und es wird berücksichtigt, dass sich die Halbgeraden der beiden Kameras wegen Messungenauigkeiten bei der Kalibrierung und endlich genauer Erkennung durch den *Object Detection Algorithmus* nicht genau schneiden. Es ist daher der Punkt X gesucht, der zu beiden Halbgeraden den kürzesten und gleichen Abstand hat. Er lässt sich als Mittelpunkt der Abstandsstrecke zwischen beiden Halbgeraden konstruieren. Die Abstandstrecke schneidet die die Halbgeraden in den Punkten F und G und steht auf beiden senkrecht.

[10] dafür müsste K^{-1} an dieser Stelle eine 4×3 -Matrix sein
[11] vgl. Gleichung 2.5

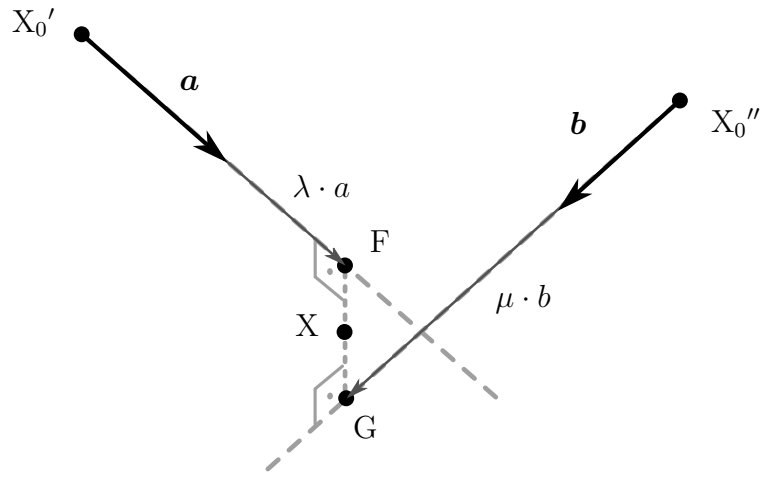


Abbildung 2.6: Geometrie der Triangulation

Die beschriebenen Zusammenhänge werden im Folgenden durch Gleichungen ausgedrückt und schließlich zur Lösung gebracht. Parameter der ersten Kamera werden mit einem Strich und die der zweiten mit zwei Strichen notiert.

Die Richtungsvektoren a, b der beiden Halbgeraden im Welt-Koordinatensystem ergeben sich analog zu (2.34) als:

$$\boldsymbol{a} = \boldsymbol{R}'^{T}\boldsymbol{K}'^{-1}\mathbf{x}' \qquad \boldsymbol{b} = \boldsymbol{R}''^{T}\boldsymbol{K}''^{-1}\mathbf{x}'' \tag{2.36}$$

F und G liegen jeweils auf einer der beiden Halbgeraden und ihre Koordinaten ergeben sich analog zu (2.35) daher als:

$$\mathbf{F} = \mathbf{X}'_0 + \lambda\,\boldsymbol{a} \qquad \mathbf{G} = \mathbf{X}''_0 + \mu\,\boldsymbol{b} \tag{2.37}$$

Die Orthogonalität der Abstandstrecke zu den beiden Halbgeraden lässt sich jeweils über das Skalarprodukt beider Richtungsvektoren ausdrücken, welches nach Definition 0 sein muss[12].

Es lässt sich demnach folgendes Gleichungssystem aufstellen, dessen Lösung die unbekannten Variablen λ und μ sind.

$$\left|\begin{array}{lcl} (\mathbf{F}-\mathbf{G})\cdot\boldsymbol{a} &=& 0 \\ (\mathbf{F}-\mathbf{G})\cdot\boldsymbol{b} &=& 0 \end{array}\right|$$

$$\left|\begin{array}{lcl} (\mathbf{X}'_0+\lambda\,\boldsymbol{a}-\mathbf{X}''_0-\mu\,\boldsymbol{b})\cdot\boldsymbol{a} &=& 0 \\ (\mathbf{X}'_0+\lambda\,\boldsymbol{a}-\mathbf{X}''_0-\mu\,\boldsymbol{b})\cdot\boldsymbol{b} &=& 0 \end{array}\right|$$

$$\left|\begin{array}{llll} (\mathbf{X}'_0-\mathbf{X}''_0)\cdot\boldsymbol{a} & +\lambda\,\boldsymbol{a}^2 & -\mu\,\boldsymbol{a}\cdot\boldsymbol{b} &= 0 \\ (\mathbf{X}'_0-\mathbf{X}''_0)\cdot\boldsymbol{b} & +\lambda\,\boldsymbol{a}\cdot\boldsymbol{b} & -\mu\,\boldsymbol{b}^2 &= 0 \end{array}\right|$$

$$\left|\begin{array}{llll} \boldsymbol{a}^2\cdot\lambda & -\boldsymbol{a}\cdot\boldsymbol{b}\cdot\mu & +(\mathbf{X}'_0-\mathbf{X}''_0)\cdot\boldsymbol{a} &= 0 \\ \boldsymbol{a}\cdot\boldsymbol{b}\cdot\lambda & -\boldsymbol{b}^2\cdot\mu & +(\mathbf{X}'_0-\mathbf{X}''_0)\cdot\boldsymbol{b} &= 0 \end{array}\right|$$

[12] bei einem Schnittwinkel von 90° zweier Vektoren u und v wird deren Skalarprodukt $u \cdot v = |u||v|\cos(90°) = 0$

$$\begin{pmatrix} \boldsymbol{a}^2 & -\boldsymbol{a}\cdot\boldsymbol{b} \\ \boldsymbol{a}\cdot\boldsymbol{b} & -\boldsymbol{b}^2 \end{pmatrix} \begin{pmatrix} \lambda \\ \mu \end{pmatrix} = \begin{pmatrix} (\mathsf{X}_0'' - \mathsf{X}_0')\cdot\boldsymbol{a} \\ (\mathsf{X}_0'' - \mathsf{X}_0')\cdot\boldsymbol{b} \end{pmatrix} \tag{2.38}$$

Die gesuchten euklidischen Koordinaten des Punktes im Raum X ergeben sich dann, wie erläutert, als arithmetisches Mittel von F und G.

$$\mathsf{X} = {}^1\!/_2(\mathsf{X}_0' + \lambda\boldsymbol{a} + \mathsf{X}_0'' + \mu\boldsymbol{b}) \tag{2.39}$$

2.2 Epipolargeometrie

Die Epipolargeometrie betrachtet den Aufbau zweier Kameras, die eine bestimmte, gleiche Szene beobachten, und trifft Aussagen über die geometrischen Beziehungen zwischen ihren überlappenden Bildern.
Für diese Arbeit soll sie beim Abgleichen der Detektionen (en.: *Correspondence Matching*) zum Einsatz kommen, das heißt zum Ordnen der in den beiden Kamerabildern jeweils erkannten Verkehrsteilnehmer zu korrespondierenden Paaren, wofür eine Möglichkeit der Überprüfung für die Zusammengehörigkeit zweier Objekt-Entdeckungen zwischen den Kamerabildern benötigt wird[13].
Dafür lässt sich die sogenannte Komplanaritätsbedingung nutzen, welche die zentrale Gesetzmäßigkeit der Epipolargeometrie darstellt.

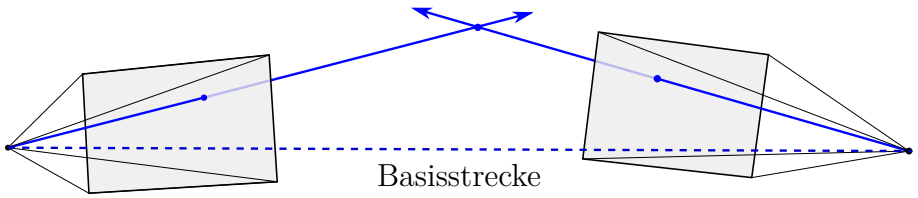


Abbildung 2.7: Übersicht der Epipolargeometrie

2.2.1 Komplanaritätsbedingung

Die Komplanaritätsbedingung sagt, dem Namen entsprechend, aus, dass die Verbindungsstrecke der beiden Kameraprojektionszentren ("Basisstrecke") mit den beiden Halbgeraden[14] der Kameras in einer Ebene liegt. Diese Bedingung ergibt sich aus dem Umstand, dass die beiden Kameraprojektionszentren mit dem beobachteten Punkt ein Dreieck darstellen, welches immer eben ist (vgl. Abbildung 2.7). Die Verbindungsvektoren zwischen diesen drei Punkten, beziehungsweise Vielfache davon, müssen daher in einer Ebenen liegen.
Als Gleichung kann dieser Zusammenhang mithilfe des Kreuzprodukts und des Skalarprodukts ausgedrückt werden. Die euklidischen Koordinaten des Kameraprojektionszentrums X_0', X_0'' sowie die Richtungsvektoren der Halbgeraden a, b sind aus

[13] vgl. 3.2.4 (*Abgleichen der Entdeckungen*)
[14] es ist von den Halbgeraden die Rede, die jeweils von einem Kameraprojektionszentrum aus in die Richtung verlaufen, in der das gesehene Objekt liegt

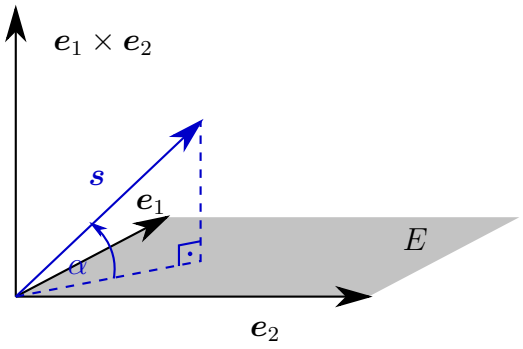
dem Unterkapitel 2.1.4 (*Kamerakalibrierung*) beziehungsweise 2.1.5 (*Triangulation*) bekannt.

$$\boldsymbol{a} \times (\mathbf{X}_0' - \mathbf{X}_0'') \cdot \boldsymbol{b} \;=\; 0 \tag{2.40}$$

Zum *Correspondence Matching* kann die Komplanaritätsbedingung dann insofern eingesetzt werden, als dass simpel geprüft wird, ob sie zutrifft ist. Ein nicht korrespondierendes Paar kann auf diese Weise in vielen Fällen einfach ausgeschlossen werden.

Allerdings muss auch hier wieder beachtet werden, dass sich die beiden Halbgeraden wegen zufälligen, minimalen Messungenauigkeiten bei der Kalibrierung und beschränkt genauer Lokalisierung durch den *Object Detection Algorithmus* in der Regel immer windschief zueinander verhalten, also gerade so nicht mit der Basisstrecke zusammen in einer Ebene liegen[15].
Statt die Komplanaritätsbedingung wie in (2.40) gesetzmäßig, exakt zu prüfen, sollte daher viel eher ein Maß dafür gefunden werden, wie gut, beziehungsweise schlecht, die Komplanaritätsbedingung erfüllt ist. Es bietet sich an, den Winkel zu ermitteln, mit dem die eine Halbgerade die Ebene schneidet, in der die andere Halbgerade und die Basisstrecke liegen. Später kann dann geprüft werden, ob sich dieser Winkel in einem definierten Toleranzbereich befindet.

Bei der Berechnung des Schnittwinkels eines divergenten Richtungsvektors s zu einer Ebene $E(e1, e2)$ wird der geometrische Zusammenhang ausgenutzt, dass sich der Winkel zwischen s und dem zur E senkrechten Richtungsvektor $e_1 \times e_2$ mit dem gesuchten Schnittwinkel *alpha* genau zu 90° ergänzen.



Ausgehend von dem noch vorzeichenbehafteten Schnittwinkel $\tilde{\alpha}$ und seinem Komplementwinkel[a] gilt ausgehend von der Definition des Skalarprodukts:

$$\sin\tilde{\alpha} \;=\; \cos\left(90° - \tilde{\alpha}\right) \;=\; \frac{\boldsymbol{e_1} \times \boldsymbol{e_2} \cdot \boldsymbol{s}}{|\boldsymbol{e_1} \times \boldsymbol{e_2}| \cdot |\boldsymbol{s}|} \tag{2.41}$$

[15] vgl. Abbildung 2.6

Denn:

$$\boldsymbol{e_1} \times \boldsymbol{e_2} \cdot \boldsymbol{s} = |\boldsymbol{e_1} \times \boldsymbol{e_2}| \cdot |\boldsymbol{s}| \cdot \cos\left(90^\circ - \tilde{\alpha}\right) \tag{2.42}$$

Der Schnittwinkel ergibt sich nun durch Anwendung der Umkehrfunktion Arkussinus und abschließend der Betragsbildung, da Schnittwinkel allgemein positiv angegeben werden.

$$\alpha \;=\; \left| \arcsin \frac{\boldsymbol{e_1} \times \boldsymbol{e_2} \cdot \boldsymbol{s}}{|\boldsymbol{e_1} \times \boldsymbol{e_2}||\boldsymbol{s}|} \right| \tag{2.43}$$

[a] zwei Komplementwinkel ergänzen sich zu 90°

Angewendet auf die vorliegende Epipolargeometrie (X_0', X_0'', a, b)[16] ergibt sich damit der Schnittwinkel α nach:

$$\alpha \;=\; \left| \arcsin \frac{\boldsymbol{a} \times (\mathbf{X}_0'' - \mathbf{X}_0') \cdot \boldsymbol{b}}{|\boldsymbol{a} \times (\mathbf{X}_0'' - \mathbf{X}_0')| \cdot |\boldsymbol{b}|} \right| \tag{2.44}$$

2.3 Kugelkoordinaten und ihre Transformationen

Im Abschnitt 2.1 (*Projektive Geometrie*) und 2.2 (*Epipolargeometrie*) findet jeweils ein Welt-Bezugssystem Verwendung, um Positionen und Richtungen im Raum, z.B. mit euklidischen Koordinaten, anzugeben. Es wird im Folgenden lokales Bezugssystem genannt.
Es stellt sich die Frage nach der Erlangung bzw. Konstruktion dieses lokalen Bezugssystems, wodurch wird es definiert. Hier bietet sich die Verwendung von geographischen Koordinaten[17] an, weil diese sowohl eine einfache Vermessung der Kalibrierungspunkte (eben mit geographischen Koordinaten) ermöglichen, als auch praktisch für die Ausgabe der letztendlich ermittelten Positionen der Verkehrsteilnehmer zu verwenden sind.

Dafür ist nun die Hin- und Rück-Transformation zwischen den geographischen Koordinaten im globalen Bezugssystem und den entsprechenden euklidischen Koordinaten im lokalen Bezugssystem gesucht. Im Folgenden werden diese in einzelnen Teilschritten hergeleitet.

2.3.1 Kugelkoordinaten und geographische Koordinaten

Als erster Schritt werden die geographischen Koordinaten als Kugelkoordinaten verstanden, wodurch sie einfacher mathematisch handhabbar werden.

Es ist anzumerken, dass das Modell einer Kugel eine Vereinfachung gegenüber den bei der Erdvermessung üblichen Referenz-Ellipsoiden darstellt. Die Form der Erde weicht nämlich nicht unerheblich von der einer Kugel ab. Die Kugelkoordinaten sind an dieser Stelle dennoch hilfreich in der Anwendung, da ihr Ziel hier nur die

[16] vgl. Abbildung 2.7

[17] das heißt die Positionsangabe im globalen Bezugssystem mit Längengrad, Breitengrad und der Höhe über dem Nullniveau

Konstruktion eines lokalen Koordinatensystems ist, aus dem auch wieder zu geographischen Koordinaten zurückgerechnet werden kann. Durch die kleine verwendete Größe des konstruierten lokalen Koordinatensystems hält sich der systematische Fehler hierbei in Grenzen und ist, besonders in Hinsicht auf den gewünschten Umfang dieser Arbeit, vernünftig einzugehen.

Kugelkoordinaten, oder auch sphärische Koordinaten, sind räumliche Polarkoordinaten. Sie werden demnach mit zwei Winkeln, die die Richtung des Punktes vom Kugelmittelpunkt aus beschreiben, und dem Abstand vom Kugelmittelpunkt angegeben. Dabei handelt es sich um den vertikalen Polarwinkel θ, den horizontalen Azimutwinkel φ und den Radius r.

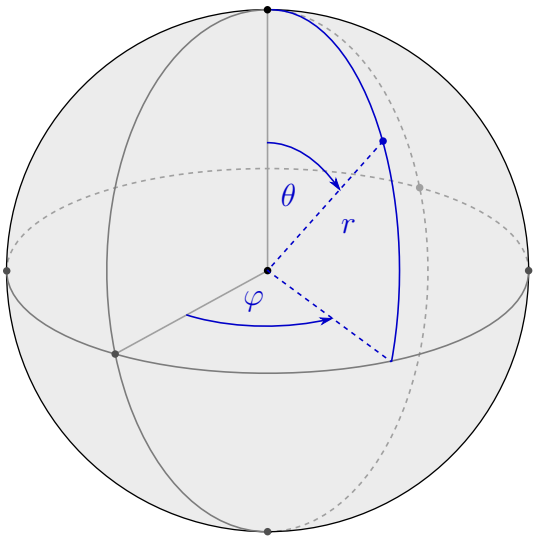


Abbildung 2.8: Kugelkoordinatensystem

Nun soll die Kugel des Kugelkoordinatensystems die Erde darstellen und das System der Kugelkoordinaten auf die geographischen Koordinaten angewendet werden. Die Äquatorebene des Kugelkoordinatensystems sei dazu identisch zur Äquatorebene der Erde und der Nord- und Südpol der Erde liege auf der Polachse des Kugelkoordinatensystems. Weiterhin sei das Kugelkoordinatensystem so definiert, dass der Azimutwinkel, wie der Längengrad bei den geographischen Koordinaten, vom Nullmeridian der Erde aus gemessen wird.

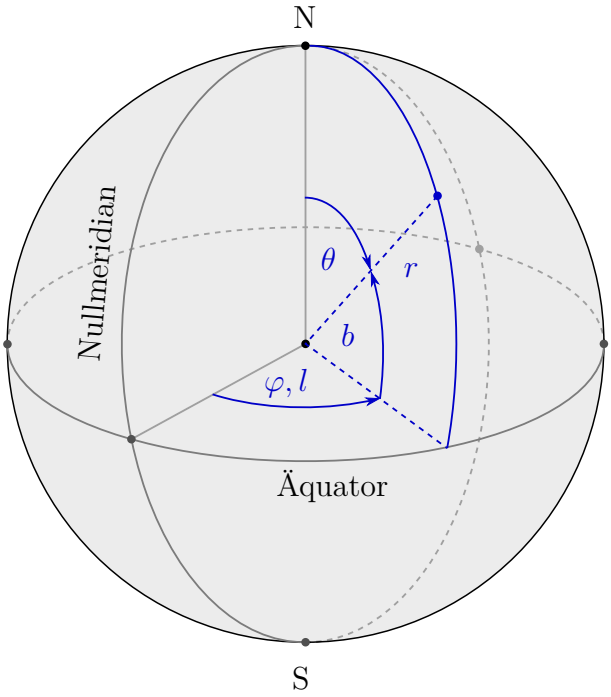


Abbildung 2.9: Globales Kugelkoordinatensystem

Die Kugelkoordinaten sind den geographischen Koordinaten nun sehr ähnlich, aber noch nicht identisch. Es muss beachtet werden, dass der Breitengrad der geographischen Koordinaten vom Äquator aus gemessen wird, während der Polarwinkel der Kugelkoordinaten den vertikalen Winkel vom Nordpol aus meint, und dass der Radius bei den Kugelkoordinaten nicht identisch mit der Höhe über dem Nullniveau bei den geographischen Koordinaten ist. Hier muss noch der Abstand dieses Nullniveaus zum Erdmittelpunkt r_0 addiert werden.
Für die Nordhalbkugel der Erde ergeben sich die Kugelkoordinaten aus den geographischen Koordinaten damit wie folgt:

geographische Koordinaten		**Kugelkoordinaten**
Höhe über Nullniveau: h	$\Longrightarrow$	Radius: $r = r_0 + h$
Breitengrad: b	$\Longrightarrow$	Polarwinkel: $\theta = 90° - b$
Längengrad: l	$\Longrightarrow$	Azimutwinkel: $\varphi = l$

Umgekehrt gilt für die Rückumformung:

Kugelkoordinaten		**geographische Koordinaten**
Radius: r	$\Longrightarrow$	Höhe über Nullniveau: $h = r - r_0$
Polarwinkel: θ	$\Longrightarrow$	Breitengrad: $b = 90° - \theta$
Azimutwinkel: φ	$\Longrightarrow$	Längengrad: $l = l$

2.3.2 Kugelkoordinaten und euklidische Koordinaten

Als nächster Schritt, um dann im anschließenden Unterabschnitt letztendlich zu den euklidischen Koordinaten im lokalen Bezugssystem zu gelangen, muss zunächst die

Umwandlungen zwischen Kugelkoordinaten und euklidischen Koordinaten innerhalb des globalen Bezugssystems betrachtet werden.
Dazu werden einfach die kartesischen Koordinatenachsen so gelegt, dass der Ursprung des euklidischen Koordinatensystems im Ursprung des Kugelkoordinatensystems liegt, die X-Achse in der Äquatorebene auf den Nullmeridian zeigt, sich die Y-Achse dazu senkrecht ebenfalls in der Äquatorebene befindet und die Z-Achse in Richtung des Nordpols deutet.

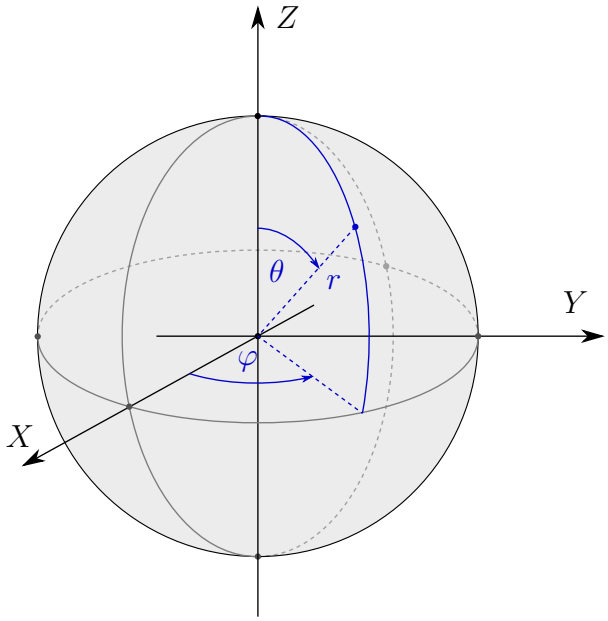


Abbildung 2.10: Globales Kugelkoordinatensystem mit globalem euklidischen Koordinatensystem

Nach grundlegenden Sätzen der Trigonometrie ergibt sich die Umrechnung zwischen beiden Koordinatensystemen:

Kugelkoordinaten		**euklidische Koordinaten**
Radius: r		$X = r \sin\theta \cos\varphi$
Polarwinkel: θ	$\Longrightarrow$	$Y = r \sin\theta \sin\varphi$
Azimutwinkel: φ		$Z = r \cos\theta$

euklidische Koordinaten		**Kugelkoordinaten**
X		Radius: $r = \sqrt{X^2 + Y^2 + Z^2}$
Y	$\Longrightarrow$	Polarwinkel: $\theta = \arccos \frac{Z}{r}$
Z		Azimutwinkel: $\varphi = \arctan \frac{Y}{X}$

2.3.3 Globales und Lokales Bezugssystem

Im letzten Schritt wird jetzt die Transformation vom euklidischen Koordinatensystem des globalen Bezugssystems (im Weiteren verkürzt “globales Koordinatensystem” genannt) zum euklidischen Koordinatensystem des lokalen Bezugssystems (im

Weiteren verkürzt “lokales Koordinatensystem” genannt) erarbeitet. Dazu muss zunächst die Lage dieses lokalen Bezugssystems definiert werden.
Das lokale Koordinatensystem soll mit seinen drei kartesischen Achsen so gelegen sein, dass der Ursprung auf der Erdoberfläche an einem festgelegten Ursprungspunkt O_L lokalisiert ist und dann die X-Achse in Richtung Osten, die Y-Achse in Richtung Norden und die Z-Achse senkrecht zum Zenit zeigt.

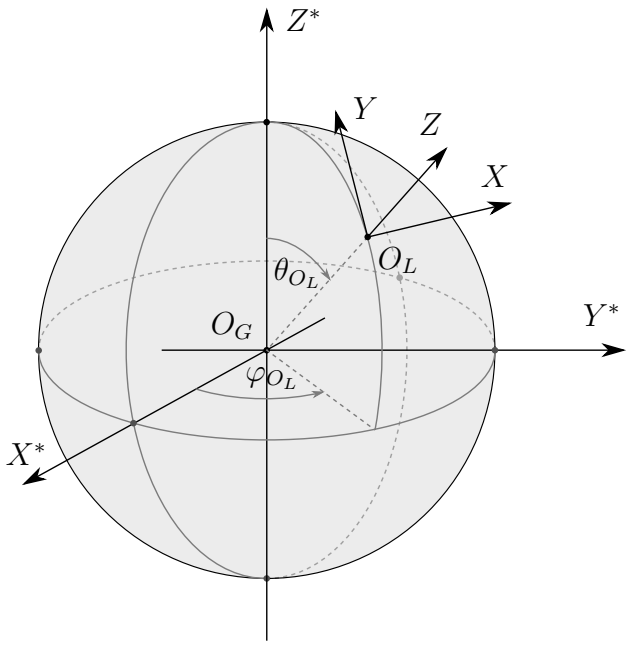


Abbildung 2.11: Globales Koordinatensystem (Index G) und lokales Koordinatensystem (Index L);
Vektoren und Koordinaten im globalen Koordinatensystem mit * notiert, die im lokalen Koordinatensystem ohne Zusatz

Da beide Koordinatensysteme kartesisch und rechtshändig sind, sowie den gleichen Größenmaßstab haben, lässt sich die Transformation zwischen ihnen auf eine Translation, die Verschiebung des einen Koordinatenursprungs auf den anderen, und anschließende Rotation, formal gesehen ein Basiswechsel, vereinfachen.
Die relative Beziehung zwischen beiden Koordinatenursprüngen für die Translation ist durch die Festlegung von O_L gegeben. Es muss demnach nur noch der Basiswechsel nachvollzogen werden, welcher übersichtlich als Übergangsmatrix dargestellt wird. Dadurch muss auch nur der Basiswechsel in eine Richtung betrachtet werden, da die Inverse der Übergangsmatrix den Basiswechsel in die umgekehrte Richtung beschreibt.
Am einfachsten ist der Basiswechsel vom lokalen zum globalen Koordinatensystem nachzuvollziehen. Dazu müssen die Basisvektoren des lokalen Koordinatensystems aus Sicht des globalen Koordinatensystems e_X^*, e_Y^* und e_Z^* ermittelt werden. Nach grundlegendem geometrischen Verständnis über Winkelfunktionen ergeben

sich diese in Abhängigkeit vom Polarwinkel θ_{O_L} und Azimutwinkel φ_{O_L} des lokalen Koordinatenursprungs O_L im globalem Bezugssystem.

$$\begin{gathered} \boldsymbol{e}_X^* = \begin{pmatrix} -\sin\varphi_{O_L} \\ \cos\varphi_{O_L} \\ 0 \end{pmatrix} \qquad \boldsymbol{e}_Y^* = \begin{pmatrix} -\cos\varphi_{O_L} \ \cos\theta_{O_L} \\ -\sin\varphi_{O_L} \ \cos\theta_{O_L} \\ \sin\theta_{O_L} \end{pmatrix} \\ \boldsymbol{e}_Z^* = \begin{pmatrix} \cos\varphi_{O_L} \ \sin\theta_{O_L} \\ \sin\varphi_{O_L} \ \sin\theta_{O_L} \\ \cos\theta_{O_L} \end{pmatrix} \end{gathered} \tag{2.45}$$

Der Basiswechsel erfolgt nun, in dem einfach die einzelnen Basisvektoren des lokalen Koordinatensystems mit ihrer zugehörigen lokalen Koordinate multipliziert werden, dann wird aufsummiert. Es wird dabei von einem beliebigen Punkt $X(X|Y|Z)$ [18] ausgegangen.

$$\overrightarrow{O_L X}^{\,*} = X \cdot \boldsymbol{e}_X^* + Y \cdot \boldsymbol{e}_Y^* + Z \cdot \boldsymbol{e}_Z^* \tag{2.46}$$

Genau dies wird durch die Übergangsmatrix D beschrieben:

$$\boldsymbol{D} = \left(\boldsymbol{e}_X^* \mid \boldsymbol{e}_Y^* \mid \boldsymbol{e}_Z^* \right) \tag{2.47}$$

$$\overrightarrow{O_L X}^{\,*} = \boldsymbol{D} \cdot \overrightarrow{O_L X} \tag{2.48}$$

Die Transformation der lokalen Koordinaten eines Punktes X zu seinen globalen ergibt sich unter Nutzung von D als:

$$\overrightarrow{O_G X}^{\,*} = \boldsymbol{D} \cdot \overrightarrow{O_L X} + \overrightarrow{O_G O_L}^{\,*} \tag{2.49}$$

Wie schon angedeutet, ergibt sich die umgekehrte Transformation, vom globalen Koordinatensystem zum lokalen, dann mit der Inversen von D.

$$\overrightarrow{O_L X} = \boldsymbol{D}^{-1} \cdot \left(\overrightarrow{O_G X}^{\,*} - \overrightarrow{O_G O_L}^{\,*} \right) \tag{2.50}$$

2.3.4 Zusammenfassung

Die Gesamttransformation von den geographischen Koordinaten eines Punktes zu seinen euklidischen Koordinaten in einem bestimmten lokalen Bezugssystem oder umgekehrt findet zusammengefasst über drei Zwischenschritte statt:

geographische Koordinaten (global)
⇕
Kugelkoordinaten (global)
⇕
euklidische Koordinaten (global)
(“globales Koordinatensystem”)
⇕
euklidische Koordinaten (lokal)
(“lokales Koordinatensystem”)

[18] X, Y, Z sind entsprechend Abbildung 2.11 die lokalen Koordinaten von X, weil sie ohne * notiert sind

3 Der Prototyp als Konzept

Nach der Erarbeitung aller nötigen theoretischen Grundlagen folgt in diesem Kapitel die Zusammenführung dieser mit weiteren kleinen Überlegungen zum gesuchten Prototyp. In diesem Kapitel soll dabei zunächst nur dessen Konzept ausgeführt werden. Es entsteht ein allgemeine Anleitung, nach der man das System in verschiedensten Anwendungsfällen zum konkreten Einsatz bringen kann.
Erst im nächsten Kapitel 4 (*Der Prototyp in der Praxis, Versuch*) soll dann ein solcher konkreter Anwendungsfall simuliert werden.

3.1 Beschreibung des Gesamtsystems

Das Konzept sieht, entsprechend der allgemeinen Abbildung 1.1 der Zielstellung, ein System aus zwei fest installierten Verkehrskameras an einer bestimmten Verkehrssituation gepaart mit einem Computer vor, auf dem ein im Weiteren einfach "Programm" genannter Programmcode für die gewünschte Funktionalität sorgt.

Damit das Aussenden von Warnnachrichten an z.B. Autos grundsätzlich möglich ist, muss eine Verarbeitung der Kameradaten in Echtzeit erfolgen. Hierfür wird grundlegend zwischen zwei Ansätzen unterschieden. Zum einen kann die Rechenlogik in Form einer kleinen Computer-Lösung direkt mit den Kameras vor Ort eingesetzt werden. Es ist aber auch der Einsatz von *Cloud Computing* denkbar, bei dem die Kameradaten zum Auswerten von leistungsstarken Rechennetzwerken über das Internet vermittelt werden. Dies scheint besonders in Hinsicht auf den neuen Mobilfunkstandard 5G attraktiv, der besonders niedrige Latenzen und hohen Datendurchsatz bei der Datenübertragung verspricht [10].

Für das System an sich ist letztendlich auch noch wichtig, wie die Kommunikation zu den Informations-Empfängern stattfindet. Dies soll jedoch im Sinne der aktuellen Forschung nicht ausführlich konzeptioniert oder entwickelt werden, da an dieser Stelle schon ein anderes Forschungsgebiet beginnt.

3.2 Teilaspekte

3.2.1 Voraussetzungen

Um den Prototyp nun detaillierter zu entwickeln sind zunächst die nötigen Voraussetzungen zu betrachten.

In erster Linie wird eine geeignete Verkehrssituation benötigt. Diese sollte sinngebend für den Einsatzes des Systems besondere Gefahrenstellen für nicht-motorisierte Verkehrsteilnehmer aufweisen. Das heißt über Bereiche verfügen, die nicht aus jeder Verkehrslage einsehbar sind. In der Regel handelt es sich dabei um Kreuzungen mit eng parkenden Autos.
Günstig wäre auch, dass die Verkehrsinfrastruktur Installationsmöglichkeiten für die Kameras und eventuell den Verarbeitungscomputer, wie zum Beispiel eine Ampelanlage, zu Verfügung stellen kann. Auf die Installation der Kameras wird weiter im nächsten Abschnitt eingegangen.

Neben den offensichtlichen Voraussetzungen, wie die rechtliche Absicherung für die automatisierte, anonyme Verkehrsüberwachung und dem Vorhandensein eines Kommunikationsweges zu den vorgesehenen Empfängern der Warninformationen, stellt der Prototyp keine weiteren einschränkenden Bedingungen für den Einsatz. Er ist grundsätzlich sehr allgemein und vielseitig einsetzbar. Sogar die Installation von mehreren solchen Systemen an einer einzigen Kreuzung ist denkbar.

3.2.2 Installation und Kalibrierung

Bei der Positionierung der Kameras ist Einiges zu beachten.

Zu aller erst sollten die Kameras hauptsächlich den gleichen Sichtbereich abdecken, da die nicht-motorisierten Verkehrsteilnehmer nur geortet werden können, wenn sie in beiden Kamerabildern sichtbar sind.

Auf keinen Fall dürfen beide Kameras sehr dicht nebeneinander positioniert sein. Dadurch würden große Ungenauigkeiten in der Ortung entstehen, weil bei der Triangulation als Methode der *Computer Stereo Vision* der Schnittpunkt der beiden jeweils zu den Bildpunkten gehörenden Kamerastrahlen berechnet wird[1]. Wenn die Strahlen aufgrund dieser schlechten Kamerapositionierung annähernd parallel wären, würden kleine zufällige Fehler zu einer enormen Abweichung von der realen Position des zu ortenden Objektes führen.

Die Entfernung von der zu beobachtenden Verkehrssituation muss abhängig von dem einzusehendem Bereich gewählt werden, ist aber nahezu beliebig, wenn für besondere Fälle einfach spezielle Kameratypen eingesetzt werden. So bieten sich für sehr kurze Distanzen Weitwinkelobjektive und für weiter entfernte Kamerastandorte Teleobjektive an.

Für den Standort der Kameras sollte stets eine erhöhte Lage präferiert werden, weil daraus eine bessere Übersicht über die Verkehrssituation resultiert. Seltener werden Verkehrsteilnehmer von anderen verdeckt, oder Ähnliches. Diese Eigenschaft ist aber ebenso essentiell für das *Correspondence Matching*. Die Kameras dürfen keinesfalls auf Höhe der Verkehrsteilnehmer angebracht werden, weil die Komplanaritätsbedingung sonst immer erfüllt ist und damit kein Kriterium für die Entscheidung einer Übereinstimmung zwischen zwei Detektionen mehr darstellt.

[1] vgl. 2.1.5 (*Triangulation*)

Wie in 2.1.1 (*Kameramodell*) erläutert, muss weiterhin auf die Vermeidung von stark Bild-verzerrenden Kameraobjektiven geachtet werden, da das verwendete Lochkameramodell sonst nicht mehr anwendbar wäre. Diese Eigenschaft lässt sich mit der Beobachtung von in der Welt geraden Linien auf dem Bild der Kamera feststellen. Wenn diese geraden Linien auch auf dem Bild der Kamera gerade Linien bleiben[2], dann lässt sich das Lochkameramodell anwenden. Wenn diese allerdings als Kurven abgebildet werden, liegt eine Verzerrung vor.

Wenn final installiert und justiert, dürfen die Kameras bestmöglich nicht mehr verstellt werden, weil nur solange keine Erneuerung der vorgenommenen Kalibrierung notwendig ist, wie Position, Orientierung und innere Parameter der Kamera identisch bleiben.
Für die Kalibrierung werden die Grundlagen aus 2.1.4 (*Kamerakalibrierung*) aufgegriffen. Es werden mindestens sechs, besser mehr, Kalibrierungspunkte benötigt, für die händisch die dreidimensionale Position im lokalen Welt-Bezugssystem und die Pixel-Position auf jeweils einem Testbild bestimmt wird. Kritische Konfigurationen der Kalibrierungspunkte im Raum, wie zum Beispiel die Beziehung, in einer Ebene zu liegen, müssen entsprechend der Hinweise vermieden werden.
Um die gerade erwähnten und für die Kalibrierung notwendigen dreidimensionalen Positionen der Kalibrierungspunkte im Raum zu erhalten, sind grundsätzlich verschiedene Wege denkbar. Eine in der Anwendung einfache und deshalb für diesen Prototypen gewählte Methode wurde in den theoretischen Grundlagen in 2.3 (*Kugelkoordinaten und ihre Transformationen*) vorgestellt. Hierbei werden ledeglich die geographischen Koordinaten und Höhen über dem Nullniveau der Kalibrierungspunkte gemessen und daraus ein lokales Koordinatensystem konstruiert.
Gleichzeitig könnten auch andere manuelle Verfahren zur Vermessung der Kalibrierungspunkte angewandt werden, die sich der Wissenschaft der Geodäsie zuordnen lassen. Hierbei kommen in der Regel spezielle Laser-Messgeräte, sogenannte *Tachymeter*, zum Einsatz, um die räumliche Szene mit den Kalibrierungspunkten vor Ort auszumessen. Ein Vorteil davon wäre, dass sich nicht auf Kalibierungspunkte beschränkt werden muss, die auf den Satellitenbildern sichtbar sind, und eventuell eine ungünstige Konstellation aufweisen. Als Kalibierungspunkte könnten auch beliebig platzierbare Objekte, wie z.B. ein Tachymeter-Reflektorstab, verwendet werden.
Beide Kameras durchlaufen nun einzeln den hergeleiteten Kalibrierungsalgorithmus aus 2.1.4 (*Kamerakalibrierung*) und man erhält die äußeren und inneren Parameter jener.

3.2.3 Erkennung

Im laufenden Betrieb wendet das Programm nun fortlaufend jeweils für jedes neue Paar von Kamerabildern ein Erkennungsalgorithmus[3] an. Die Auswahl an diesen ist sehr groß, die Entscheidung sollte von verschiedenen Faktoren abhängig gemacht werden.

[2] en.: *straight line preserving mapping*
[3] en.: *Object Detection* Algorithmus

Es muss zum einen die geforderte Zuverlässigkeit und Präzision gegen die dafür notwendige Rechenleistung abgewogen werden. Weiterhin gibt es sehr viele verschiedene Grundtypen von Erkennungsalgorithmen, die jeweils einsatzspezifische Vor- und Nachteile haben. Entscheidend ist auch die Frage, ob auf einen vortrainierten[4] Algorithmus zurückgegriffen werden soll, oder ob nicht ein eigenes Training mit angepassten Datensätzen, vielleicht sogar eine ganz eigene Implementation eines *Object Detection* Algorithmus, für höhere Präzision und Zuverlässigkeit zu präferieren ist. Das muss gegen den potenziell sehr hohen Arbeitsaufwand abgewogen werden.
Es ist schließlich auch noch zu betonen, dass in diesem Gebiet der Forschung sehr schnell Fortschritte gemacht werden und deshalb kein Erkennungsalgorithmus für immer der beste Kandidat sein wird.

3.2.4 Abgleichen der Entdeckungen

Nachdem der implementierte Erkennungsalgorithmus für alle erkannten Objekte je die Position und Größe eines das Objekt umschließenden Rechtecks und die Objektklasse[5] der Detektion ausgegeben hat, folgt im Programm das Abgleichen der Entdeckungen[6] zwischen den beiden Bildern. Zur nachfolgenden Ortung muss nämlich klar sein, welche beiden Entdeckungen zusammengehören.
Das Problem wird deutlich, wenn man sich vorstellt in der Verkehrssituation befänden sich mehrere Fahrradfahrer. Dem Programm muss eine Möglichkeit gegeben werden, zu entscheiden, welche Entdeckung eines Fahrradfahrers im einen Bild zu welcher im zweiten Bild geordnet werden soll. Ohne klare Bildpositionspaare könnte letztendlich keiner der Fahrradfahrer geortet werden.

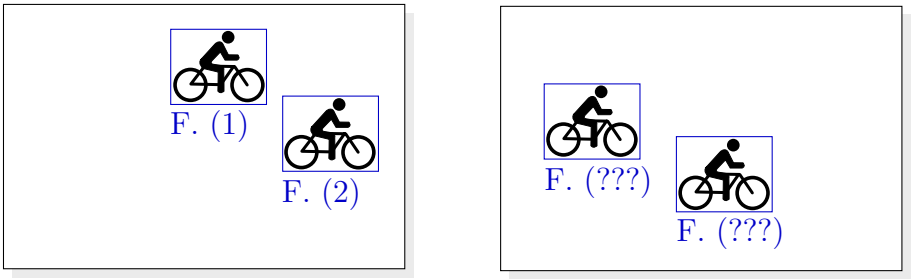


Abbildung 3.1: Zuordnungsproblem

Deswegen wird der in 2.2.1 (*Komplanaritätsbedingung*) beschriebene Schnittwinkel als Maß für die Erfüllung der Komplanaritätsbedingung auf alle potenziellen Entdeckungs-Paare angewendet. Für eine Detektion des einen Kamerabildes wird die entsprechende Partner-Detektion nun entsprechend des kleinsten Schnittwin-

[4] beim *Deep Learning* oder allgemein *Machine Learning* wird die Funktionalität des Erkennens von Objekten durch Training auf bzw. mit bestimmten Datensätzen erlernt
[5] abhängig von der Implementation und dem Training des Algorithmus' stehen meist Objektklassen wie "Fahrradfahrer", "Person", "Auto", "Katze" etc. zur Erkennung bereit
[6] en.: *Correspondence Matching*

kels gewählt, solange dieser noch in einem frei definierbaren Toleranzbereich[7] liegt. Durch den Toleranzbereich wird gewährleistet, dass das Programm nicht irrtümlich einen falschen Partner wählt, nur weil der richtige Partner auf dem anderen Bild zufällig gar nicht erkannt wurde.

Sowohl für die Berechnung des Schnittwinkels zu einem Detektions-Paar, als auch für die in Kürze folgende Ortung eines Verkehrsteilnehmers aus einem Detektions-Paar werden die Richtungsvektoren gebraucht, die vom jeweiligen Kameraprojektionszentrum hin zu dem gesehenen Objekt zeigen.
Für die Ermittlung dieser sind eindeutige Pixel-Positionen der Entdeckungen auf den Bildern notwendig. Diese sind, wie in Abbildung 1.1 zu erkennen, näherungsweise durch die Mittelpunkte der vom Erkennungsalgorithmus ausgegebenen Erkennungsrechtecke gegeben.

3.2.5 Verortung

Die dreidimensionale Verortung der Verkehrsteilnehmer auf Grundlage der Detektionspaare zählt zu der Kernfunktionalität des Programms.
Wie auch bei der Berechnung des Schnittwinkels werden die Pixel-Positionen der Objekte auf den Bildern durch die Mittelpunkte der Entdeckungsrechtecke ermittelt und schließlich kann das Programm unter Einsatz des in 2.1.5 (*Triangulation*) ausführlich beschriebenen *Stereo Vision* Algorithmus' die räumliche Position des Verkehrsteilnehmers ausmachen. Alle benötigten inneren und äußeren Parameter der Kameras sind durch die Kalibrierung gegeben.
Anschließend können die erhaltenen lokalen euklidischen Koordinaten, wenn erforderlich, auch in praktisch nützliche geographische Koordinaten zurückgerechnet werden, wie in 2.3 (*Kugelkoordinaten und ihre Transformationen*) beschrieben wurde.

3.2.6 Weitere Verarbeitung

Mit den erhaltenen Positionen der Verkehrsteilnehmer in potenziell gefährlicher Situation kann letztendlich je nach Anwendungsfall beliebig verfahren werden.

Für den Haupteinsatzzweck, der Sendung von Warnnachrichten an sich in der Nähe befindende Kraftfahrzeuge, muss das Programm an dieser Stelle eine Risikobewertung durchführen und eine Schnittstelle für die Kommunikation zu den genannten Kraftfahrzeugen implementiert haben. Der Endpunkt der Datenverarbeitung ist dann beim Informationsempfänger. In Autos können die Positionsinformationen zum Beispiel auf internen Navigationskarten und in modernen Head-up-Displays visualisiert werden.
Ausführlichere Betrachtungen dieser Art sollen allerdings, wegen der schon in 3.1 skizzierten Gründe nicht Teil dieser Forschungsarbeit sein. Das Prototypenkonzept

[7] der Toleranzbereich muss abhängig von der Genauigkeit der Kalibrierung und des Erkennungsalgorithmus' gewählt werden

betrachtet nur den Weg zum Erhalten der dreidimensionalen Ortung beliebiger Verkehrsteilnehmer.

4 Der Prototyp in der Praxis, Versuch

Grundsätzliches Ziel des Versuches soll es sein, die Funktionstüchtigkeit des Prototyps im Sinne eines *Proof of Concept* an einem konkreten Einsatz zu belegen. Die Leistungsbewertung und -optimierung des Systems sei nur zweitrangig von Bedeutung. Ursächlich hierfür ist, dass bisher kein direkt vergleichbares Erkennungssystem bekannt ist und es daher in erste Linie um die Potenzbeurteilung des Konzepts geht.

Demnach reicht es für den Versuch, den Programmcode zur Verarbeitung der Kameradaten im Nachhinein auszuführen, und nicht wie bei einem realen Anwendungsfall in Echtzeit[1]. Dadurch wird viel Aufwand bei der Versuchsdurchführung gespart, und die Funktionstüchtigkeit des Systems kann nichtsdestotrotz gleichartig beurteilt werden.

In den einzelnen Unterabschnitten dieses Kapitels wird folgend auf die für den Versuch zu erfüllenden Aufgaben eingegangen.

4.1 Programmierung

Für die Implementation des für den Versuch benötigten Programmcodes wurde die Programmiersprache Python gewählt. Diese zeichnet sich durch eine einfache Syntax und eine breite Unterstützung von Programmbibliotheken aus, was ein effizientes Programmieren unterstützt und im Allgemeinen besser lesbaren Code produziert.
Zur einfachen Handhabung von Vektoren und Matrizen sowie deren Operationen und Zerlegungen wurde die Programmbibliothek NumPy [11] verwendet, eine der meist verwendeten Python Erweiterungen für wissenschaftliches Rechnen [11]. Dadurch wird auf eine zuverlässige und im Vergleich zur eigenen Implementierung effiziente und zeitsparende Lösung zurückgegriffen.

Ein weiterer Aspekt der Software-Implementation ist, dass der gesamte Programmcode in englischer Sprache verfasst wurde, um neben dem besseren Zusammenspiel mit der Programmiersprache hauptsächlich die Zugänglichkeit des Codes für die internationale Gemeinschaft zu ermöglichen. Außerdem verbessert sich die Lesbarkeit allgemein.

[1] vgl. 3.1 (*Beschreibung des Gesamtsystems*)

Als Erkennungsalgorithmus[2] wurde eine quelloffene und vortrainierte Implementierung [12] des *Single Shot Detector* Algorithmus [13] in Form eines *Mobile Net* [14] eingesetzt. Diese hält ein für die Anwendung in dieser Forschung perfektes Mittelmaß zwischen Effizienz und Präzision beziehungsweise Zuverlässigkeit. Bei der Programmierung zum Einsatz des genannten Erkennungsalgorithmus' galt [15] als Vorlage.

Als weiteres Hilfsmittel der Programmierung wurde auf die verbreitete Versionierungs-Software Git [16] zurückgegriffen, welche weitläufigen Einsatz in fast allen heutigen Software-Entwicklungen aufweist. Sie eignet sich zum dezentralen, verteilten und nicht-linearen Entwickeln und stellt unter Nutzung eines *Online Remote Repositories*[3] einen von überall und jedem zugänglichen Speicherort dar.

Der gesamte beim Versuch eingesetzte Programmcode, inklusive Annotationen, findet sich im Anhang der Dokumentation wieder.

4.2 Vorbereitung und Kalibrierung

Für den Versuch mussten zunächst einige Vorkehrungen getroffen werden, speziell galt es die erläuterten Hinweise des vorangegangenen Kapitels 3 (*Der Prototyp als Konzept*) einzuhalten.

Das bedeutete erstens, dass als Verkehrssituation eine Kreuzung gewählt wurde, an der nicht-motorisierte Verkehrsteilnehmer schlecht von vorbeifahrenden Autos gesehen werden können. Die Kameras sollten zudem erhöht und beide auf die Szene gerichtet positioniert werden. Dies wurde durch den Zugang zu einem angrenzenden Gebäude realisiert. Die Kameras wurden auf festen Stativen installiert, um sicherzustellen, dass sich die Kameraparameter sich nach der Installation nicht mehr ändern würden, und aus zwei verschiedenen Fenstern gerichtet, wodurch der geforderte Abstand der beiden Kameras voneinander ausreichend gegeben war.

Als Kamera-Hardware haben sich Smartphones, im konkreten Fall zwei Exemplare des Modells *Google Pixel*, besonders angeboten, weil sie aufgrund ihrer kleinen Optik allgemein sehr wenig optische Verzerrungen erzeugen. Dies ließ sich auch bei den beiden verwendeten Exemplaren feststellen.

[2] wie in 3.2.3 (*Erkennung*) beschrieben

[3] ein *Git Repository* ist, vereinfacht gesagt, ein Depot für alle Projektdateien;
ein *Remote Repository* ist eine dynamische Kopie dieses Depots bei einem Cloud-Anbieter

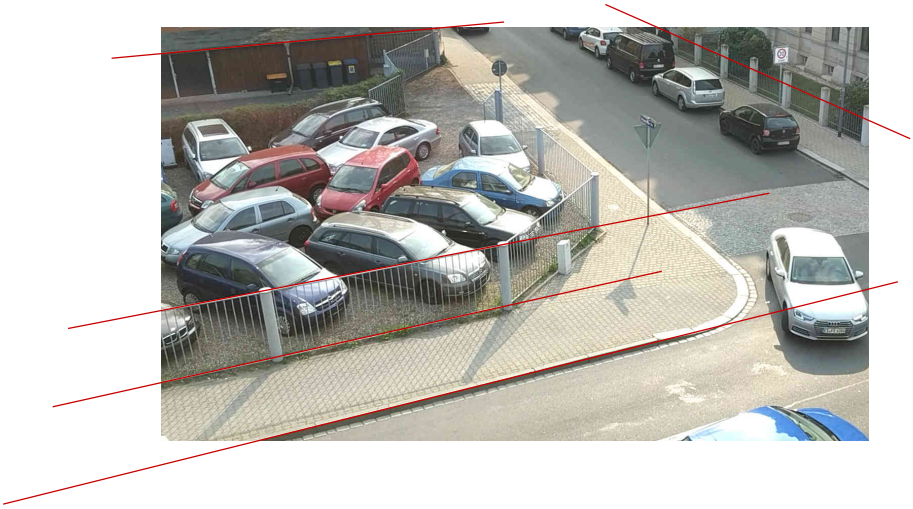


Abbildung 4.1: Geraden im Raum werden auch als Geraden abgebildet

Bei dem Erwerb von Kalibrierungsdaten wurden zunächst elf markante Kalibrierungspunkte, in der Anzahl ein gutes Mittelmaß zwischen Genauigkeit und Aufwand, gewählt und unter Nutzung des offen zugänglichen Satellitenbild-Datensatzes *Luft Sachsen aktuell* [17] in dem Programm *Java OpenStreetMap Editor (JOSM)* [18] markiert[4]. Dadurch konnten die genauen Längengrade und Breitengrade aller Kalibrierungspunkte bestimmt werden. Um nun die Höhe der Kalibrierungspunkte exakt zu bestimmen, wurden die Punkte anschließend in das Programm *Magic Maps Tour Explorer 25 Deutschland* [19] importiert, dass mit amtlichen topographischen Daten arbeitet und über ein fein aufgelöstet Höhenmodell verfügt. Abschließend mussten nur noch händisch die Pixel-Positionen der Kalibrierungspunkte auf jeweils einem Testbild ermittelt werden. Dazu lässt sich ein Bildbetrachtungs- oder Bildbearbeitungsprogramm nutzen.

[4] vgl. Abbildung 4.2

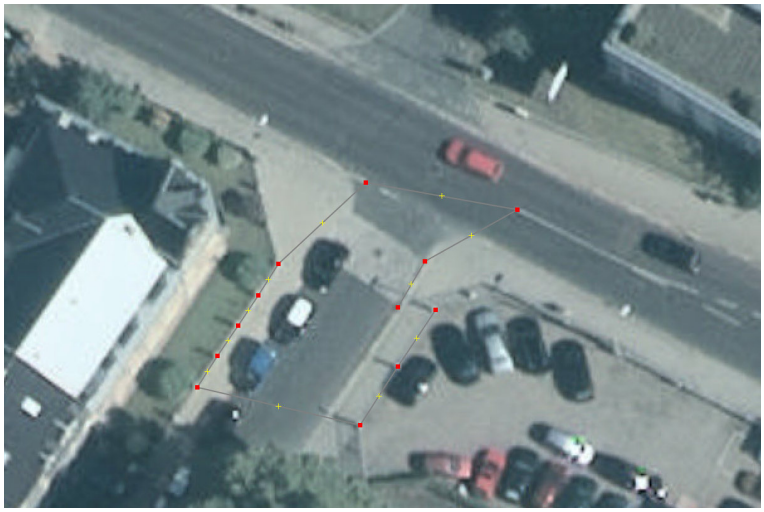


Abbildung 4.2: Lokalisierung der Kalibrierungspunkte auf dem Satellitenbild

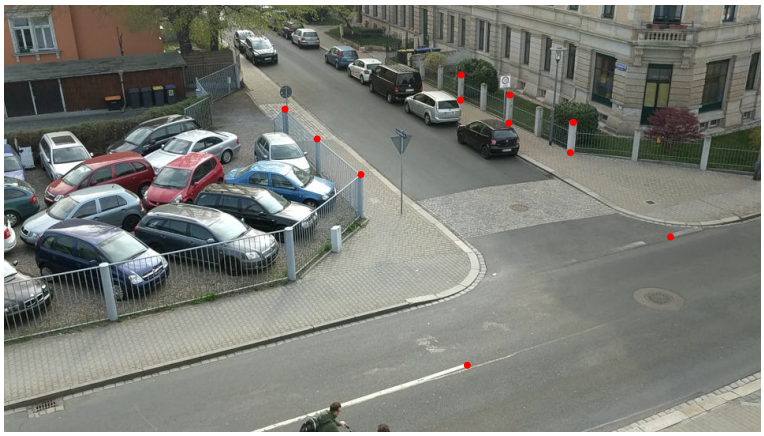


Abbildung 4.3: Veranschaulichung der Lokalisierung der Kalibrierungspunkte in einem Testbild

4.3 Durchführung

Zur eigentlichen Durchführung des Versuchs wurden im ersten Schritt beide Smartphone-Kameras konfiguriert und im Video-Modus gestartet, mit einem Klatschen eine audio-visuelle Markierung für die spätere Synchronisierung beider Videos gesetzt und einige Minuten Datenmaterial aufgenommen.

Im zweiten Schritt konnten die Videos in einem Videoschnittprogramm synchronisiert werden und es galt anschließend eine für die Analyse interessante Stelle zu wählen.

Abschließend wurde im dritten Schritt der Programmcode[5] mit den Kalibrierungsdaten und den Videoaufnahmen als Eingabe gestartet.

[5] vgl. 4.1 (*Programmierung*)

4.4 Auswertung und Ergebnisse

Um die Funktionsfähigkeit des Prototyps anhand des konkreten Versuches beurteilen zu können, erwies es sich als sinnvoll, die vom Programmcode ausgegebenen Objekt-Detektionen und die dreidimensionalen Positionen der Verkehrsteilnehmer zunächst zu visualisieren.
Der Programmcode wurde daher dazu erweitert, die einzelnen Kamerabilder mit den darauf gezeichneten Detetektionen wieder als Video-Datei auszugeben. Dies geschah für beide Kameras.
Für die Visualisierung der Ortungs-Informationen hat sich hingegen die Anwendung *Blender* [20] angeboten, weil sie enorm funktionsreich und leistungsstark für 3D-Modellierung und -Rendering einsetzbar und gleichzeitig kostenlos und quelloffen ist. Mit ihr sollte eine 3D-Animation der Szene mit ihren bekannten Objekten, speziell den dynamisch erkannten und georteten Verkehrsteilnehmern, aber auch den statischen Kameras und Kalibrierungspunkten zur Orientierungshilfe, erstellt werden. Zum Darstellen der Objekte der Szene wurde ein in *Blender* ausführbares Python-Skript geschrieben, dass diese an ihrem entsprechendem Zeitpunkt des Auftretens unter Nutzung von *Keyframes* in eine dreidimensionale Render-Szene einfügt und mit farbigen Geraden beziehungsweise Kugeln sichtbar macht. Die entstehende Animation ist wiederum als Video exportierbar.

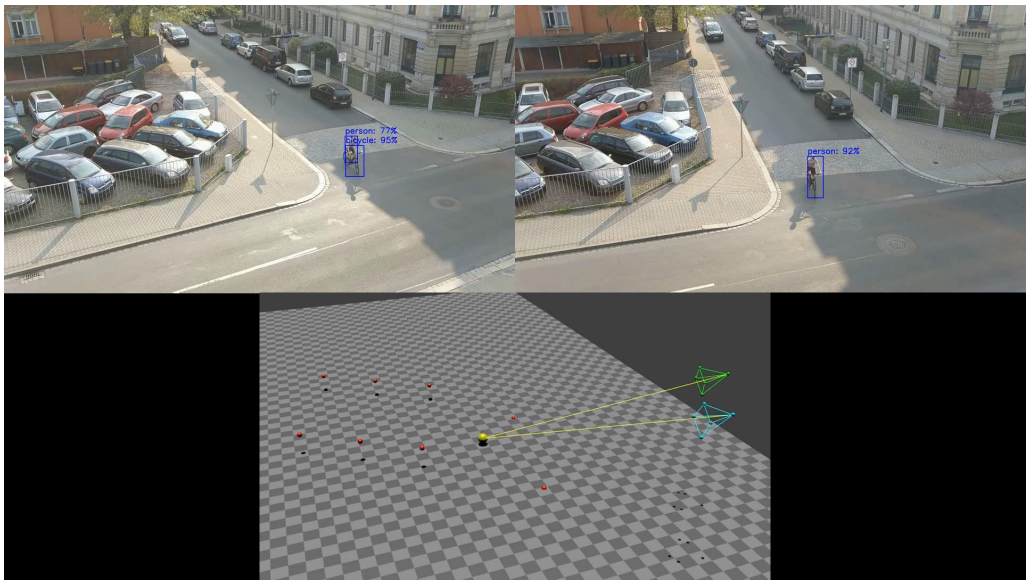


Abbildung 4.4: Standbild der erhaltenen Visualisierung

Mit der vorliegenden visuellen Auswertung der Ergebnisse des Versuches lässt sich die Potenz des Prototyps schließlich gut beurteilen.

Da ist als erstes hervorzuheben, dass es dem System im konkreten Anwendungsfall sehr gut gelingt, den zu sehenden Fahrradfahrer kontinuierlich in den Kamerabildern zu erkennen, und auch das dreidimensionale Verorten funktioniert einwandfrei. Dies ist daher abzuleiten, dass bei der gleichzeitigen Betrachtung der drei

Auswertungs-Videos das Heranrollen des Fahrradfahrers genau mit der Bewegung des in der 3D-Animation visualisierten georteten Objekts übereinstimmt. Dabei tritt auch wenig zufällige Bewegung der lokalisierten Position des Verkehrsteilnehmers aufgrund etwaiger Ungenauigkeiten bei der Objekterkennung oder in der Kalibrierung auf, welche durch ein starkes Zittern der Visualisierung in der Animation erkennbar wäre.

Es zeigten sich allerdings auch Problemstellen auf.
So ist es dem Erkennungsalgorithmus zum einen nicht durchweg möglich, den in der Szene zentralen Fahrradfahrer zuverlässig zu erkennen. Dies ist dem Fakt geschuldet, dass auf künstlicher Intelligenz beruhende Objekterkennungsalgorithmen immer eine gewisse Ungenauigkeit in ihrer Erkennung aufweisen. Im konkreten Fall spielte sicherlich auch eine große Rolle, dass der von den Kameras abgedeckte Sichtbereich sehr groß war, und demnach der vom Fahrradfahrer eingenommene Bildbereich sehr klein und weniger scharf aufgelöst. Dies vermindert die Informationsgrundlage für den Erkennungsalgorithmus. Zuletzt ist auch die Blickperspektive der rechten Kamera auf den Fahrradfahrer, die im konkreten Fall ungünstiger Weise sehr frontal ausfällt, für die Lücken in der Erkennung verantwortlich zu machen.
Zum anderen ist auch die Laufzeit des Programmcodes kritisch zu betrachten. Für die Ermittlung der Detektionen und die 3D-Ortung benötigte das Programm durschnittlich $4,2s$ [6]. Jener Umstand ergibt sich daraus, dass der Erkennungsalgorithmus enorm hohen Ressourcenbedarf bei der Verarbeitung von großen Bildern aufweist, wie sie wegen der benannten relativ gesehen kleinen Größe der Verkehrsteilnehmer auf den Kamerabildern eingesetzt wurden mussten.

[6] bei der Ausführung auf einem Laptop aus dem privaten Haushalt

5 Zusammenfassung und Ausblick

Nachdem im ersten großen Teil der Dokumentation[1] mit der detaillierten Betrachtung theoretischer Grundlagen wichtige Funktionsalgorithmen und technische Hilfsmittel erarbeitet wurden, konnte aus diesen im zweiten Hauptteil[2] ein allgemein einsetzbares Prototypenkonzept zur automatisierten optische Erfassung und dreidimensionale Ortung von nicht-motorisierten Verkehrsteilnehmern mithilfe von zwei Verkehrskameras zusammengesetzt werden, welches sich schließlich im dritten großen Abschnitt[3] in Python implementiert und in einem konkreten Versuch getestet als funktionstüchtiges und vielversprechendes System erwies.

Der erarbeitete Prototyp stellt sich damit als wertvoller Beitrag zur Forschung am vernetzten Fahren, zum einen als Demonstrator und zum anderen als Grundlage für real einsetzbare Systeme, heraus.
Auf der Basis dieser Forschungsarbeit könnte aus dem Prototypen, aus der Idee, eine hocheffiziente Lösung zur Erhöhung der Verkehrssicherheit im Rahmen der *Car2Infrastructure Communication* reifen. Dazu wären Verbesserungen an der grundsätzlichen Funktion Prototyps in verschiedenen Aspekten interesssant beziehungsweise notwendig zu erörtern:

- die Vereinfachung des Kalibrierungsalgorithmus, sodass zum Beispiel weniger Kalibrierungspunkte benötigt werden oder die inneren Kameraparameter vollautomatisch unter Nutzung von Abgleichsalgorithmen wie dem *RANSAC*-Algorithmus [21] erhalten werden können

- die Erhöhung der Zuverlässigkeit der Objekterkennung; hier wäre unter Anderem der Einsatz von mehreren Kameras gleichzeitig denkbar

- die Einbeziehung weiterer Informationen zum Abgleichen der Detektionen, wie die Seitenverhältnisse der Detektionsrechtecke oder die Bildinformationen der Detektionen selber[4]

[1] Kapitel 2 (*Theoretische Grundlagen*)
[2] Kapitel 3 (*Der Prototyp als Konzept*)
[3] Kapitel 4 (*Der Prototyp in der Praxis, Versuch*)
[4] zwei Bilder können mit sogenannten *Content-based image retrieval*-Algorithmen (*CBIR*) [22], die üblicherweise für Bilder-Suchmaschinen eingesetzt werden, auf Übereinstimmung geprüft werden

- die Weiterverarbeitung der Analysedaten, zum Beispiel zur Ermittlung von Geschwindigkeitsinformationen oder zum Voraussage des Standorts mithilfe der bisherig bekannten Bewegungsrichtung

Literaturverzeichnis

[1] Andreas Kuhn. *Was ist der Unterschied zwischen autonomen, automatisierten, vernetzten, kooperativen Fahren?* ANDATA, 2018. URL: https://www.andata.at/de/antwort/was-ist-der-unterschied-zwischen-autonomen-automatisierten-vernetzten-kooperativen-fahren.html.

[2] Barbara Lange. *Sichtweite erhöhen.* heise online, 2009. URL: https://www.heise.de/ix/artikel/Sichtweite-erhoehen-820516.html.

[3] Prof. Dr. Dieter Zöbel. *"Car 2 Car/Car 2 X" Kommunikation.* Universität Koblenz-Landau, 2009. URL: https://www.uni-koblenz-landau.de/de/koblenz/fb4/ist/AGZoebel/Lehre/ss09/Seminar09/graf.

[4] *Ko-HAF Projektpartner.* Ko-HAF, 2018. URL: https://www.ko-haf.de/konsortium/.

[5] *Ko-HAF Ergebnisse.* Ko-HAF, 2018. URL: https://www.ko-haf.de/ergebnisse/.

[6] Cyrill Stachniss. *Photogrammetry I - 14 - Homogeneous Coordinates.* YouTube, 2015. URL: https://www.youtube.com/watch?v=ZNB6SpEBnBQ.

[7] Cyrill Stachniss. *Photogrammetry I - 15 - Camera Extrinsics and Intrinsics.* YouTube, 2015. URL: https://www.youtube.com/watch?v=DX2GooBIESs.

[8] Cyrill Stachniss. *Photogrammetry I - 16 - DLT & Camera Calibration.* YouTube, 2015. URL: https://www.youtube.com/watch?v=ywternCEqSU.

[9] Richard Hartley und Andrew Zisserman. *Multiple View Geometry in Computer Vision Edition 2.* Cambridge University Press, 2003. URL: http://cvrs.whu.edu.cn/downloads/ebooks/Multiple%5C%20View%5C%20Geometry%5C%20in%5C%20Computer%5C%20Vision%5C%20(Second%5C%20Edition).pdf.

[10] Satbir Singh, Hemant Singh und Tanvir Singh. *An Overview of Fifth Generation (5G) Technology for Advanced Mobile and Wireless Communication Services.* ResearchGate, 2016. URL: https://www.researchgate.net/publication/305774984_An_Overview_of_Fifth_Generation_5G_Technology_for_Advanced_Mobile_and_Wireless_Communication_Services.

[11] NumPy developers. *NumPy.* NumPy, 2019. URL: https://www.numpy.org/.

[12] chuanqi305. *MobileNet-SSD.* Github, 2018. URL: https://github.com/chuanqi305/MobileNet-SSD.

[13] Wei Liu u. a. *SSD: Single Shot MultiBox Detector*. CoRR, 2015. URL: http://arxiv.org/abs/1512.02325.

[14] Andrew G. Howard u. a. *MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications*. CoRR, 2017. URL: https://arxiv.org/abs/1704.04861.

[15] Adrian Rosebrock. *Object detection with deep learning and OpenCV*. pyimagesearch, 2017. URL: https://www.pyimagesearch.com/2017/09/11/object-detection-with-deep-learning-and-opencv/.

[16] *Git is a free and open source distributed version control system*. Git, 2019. URL: https://git-scm.com/.

[17] *Webdienste für Luftbilder (Digitale Orthophotos)*. Staatsbetrieb Geobasisinformation und Vermessung Sachsen (GeoSN), 2016. URL: http://www.landesvermessung.sachsen.de/inhalt/info/archiv/2015/150120/150120.html.

[18] *JOSM is an extensible editor for OpenStreetMap (OSM) for Java 8+*. OpenStreetMap, 2019. URL: https://josm.openstreetmap.de/.

[19] *Tour Explorer 25 Deutschland*. MagicMaps GmbH, 2019. URL: https://www.magicmaps.de/produkte/tourenplanung-am-pc/tour-explorer-25.html.

[20] *Blender is the free and open source 3D creation suite*. blender, 2019. URL: https://www.blender.org/.

[21] Martin A. Fischler und Robert C. Bolles. *Random Sample Consensus: A Paradigm for Model Fitting with Applications to Image Analysis and Automated Cartography*. Communications of the ACM, 1981.

[22] Jagpal Singh, Jashanbir Singh Kaleka und Reecha Sharma. *Different Approaches of CBIR Techniques*. ResearchGate, 2019. URL: https://www.researchgate.net/publication/266169223_Different_Approaches_of_CBIR_Techniques.

Selbständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst und keinen anderen als die im Literatur- und Quellenverzeichnis angegebenen Hilfsmittel verwendet habe.

Insbesondere versichere ich, dass ich alle wörtlichen und sinngemäßen Übernahmen aus anderen Werken als solche kenntlich gemacht habe.

Dresden, 08.12.2019
Ort, Datum

Peter Haiduk
Unterschrift

Anhang

Programmcode

calibration_points.py

```
# --TASK-- manages calibration point data

import numpy as np
from helper_functions import *

# calibration points
measured_calibration_points = [
  {'lat': 51.032952522, 'lon': 13.709240147, 'ele': 131+1.68, 'x_A': (903,428), 'x_B': (725,457)},
  {'lat': 51.032914426, 'lon': 13.709200348, 'ele': 131+1.68, 'x_A': (793,336), 'x_B': (689,346)},
  {'lat': 51.032875435, 'lon': 13.709159880, 'ele': 131+1.68, 'x_A': (711,262), 'x_B': (666,260)},
  {'lat': 51.032900885, 'lon': 13.708986442, 'ele': 131+1.51, 'x_A': (1158,174), 'x_B': (1202,186)},
  {'lat': 51.032900885, 'lon': 13.708986442, 'ele': 131+0.00, 'x_A': (1157,237), 'x_B': (1200,254)},
  {'lat': 51.032941878, 'lon': 13.709030178, 'ele': 131+1.51, 'x_A': (1282,226), 'x_B': (1300,250)},
  {'lat': 51.032941878, 'lon': 13.709030178, 'ele': 131+0.00, 'x_A': (1278,296), 'x_B': (1294,328)},
  {'lat': 51.032983504, 'lon': 13.709072909, 'ele': 131+1.51, 'x_A': (1443,294), 'x_B': (1429,338)},
  {'lat': 51.032983504, 'lon': 13.709072909, 'ele': 131+0.00, 'x_A': (1436,371), 'x_B': (1420,424)},
  {'lat': 51.033037829, 'lon': 13.709166331, 'ele': 131+0.00, 'x_A': (1689,583), 'x_B': (1584,700)},
  {'lat': 51.033019650, 'lon': 13.709327787, 'ele': 131+0.00, 'x_A': (1173,909), 'x_B': (748,1048)}
]

# randomly selected origin point
measured_origin = {'lat': 51.03302354862048, 'lon': 13.709257663085804, 'ele': 131+0.00}
```

```
23
24  # gives access to calibration points
25  def get_calibration_points():
26    calibration_points = []
27    Oa_spheric = (6371000+measured_origin['ele'],
      ↪  np.deg2rad(90-measured_origin['lat']),
      ↪  np.deg2rad(measured_origin['lon']))
28    for mcp in measured_calibration_points:
29      X_spheric = (6371000+mcp['ele'], np.deg2rad(90-mcp['lat']),
        ↪  np.deg2rad(mcp['lon']))
30      OaX = global_to_local(X_spheric, Oa_spheric)
31      calibration_points.append({
32        'x_A': mcp['x_A'],
33        'x_B': mcp['x_B'],
34        'X': OaX
35      })
36    return calibration_points
```

calibration.py

```
1   # --TASK-- carries out the calibration procedure
2
3   import numpy as np
4   from calibration_points import get_calibration_points
5   from helper_functions import *
6
7   # gives access to calibration data
8   def get_calibration():
9
10    # load calibration points
11    calibration_points = get_calibration_points()
12
13    # create empty M matrices
14    M_A = M_B = np.empty((0,12))
15
16    # fill M matrices
17    for cp in calibration_points:
18      # camera A
19      line = np.concatenate([ -to_homogeneous(cp['X']), [0,0,0,0],
        ↪  cp['x_A'][0]*to_homogeneous(cp['X']) ])
20      M_A = np.append(M_A, [line], axis=0)
21      line = np.concatenate([ [0,0,0,0], -to_homogeneous(cp['X']),
        ↪  cp['x_A'][1]*to_homogeneous(cp['X']) ])
22      M_A = np.append(M_A, [line], axis=0)
23
24      # camera B
```

```
    line = np.concatenate([ -to_homogeneous(cp['X']), [0,0,0,0],
    ↪  cp['x_B'][0]*to_homogeneous(cp['X']) ])
    M_B = np.append(M_B, [line], axis=0)
    line = np.concatenate([ [0,0,0,0], -to_homogeneous(cp['X']),
    ↪  cp['x_B'][1]*to_homogeneous(cp['X']) ])
    M_B = np.append(M_B, [line], axis=0)

  # singular value decomposition
  U_A, s_A, VT_A = np.linalg.svd(M_A, full_matrices=False)
  U_B, s_B, VT_B = np.linalg.svd(M_B, full_matrices=False)

  # assemble P matrices
  P_A = np.reshape(VT_A[-1], (3,4))
  P_B = np.reshape(VT_B[-1], (3,4))

  # decompose P matrices
  XO_A = np.reshape( np.dot( np.linalg.inv(-P_A[:,:3]) , P_A[:,3:] ), 3)
  XO_B = np.reshape( np.dot( np.linalg.inv(-P_B[:,:3]) , P_B[:,3:] ), 3)
  RT_A, KI_A = np.linalg.qr(np.linalg.inv(P_A[:,:3]))
  RT_B, KI_B = np.linalg.qr(np.linalg.inv(P_B[:,:3]))

  return {
    'XO_A': XO_A,
    'RT_A': RT_A,
    'KI_A': KI_A,

    'XO_B': XO_B,
    'RT_B': RT_B,
    'KI_B': KI_B
  }
```

detection_matching_triangulation.py

```
# --TASK-- applies object detection
# --TASK-- matches detections
# --TASK-- locates objects by triangulation

# command line arguments:
#     <source_video_A_path>
#     <source_video_B_path>
#     <target_video_A_path>
#     <target_video_B_path>
#     <target_datafile_path>


import sys, time, json, cv2, numpy as np
```

```
14 from calibration import get_calibration
15 from helper_functions import *
16
17 # load recorded videos
18 vcap_A = cv2.VideoCapture(sys.argv[1])
19 vcap_B = cv2.VideoCapture(sys.argv[2])
20
21 # open video write stream for detection output
22 vwriter_A = cv2.VideoWriter(sys.argv[3], cv2.VideoWriter_fourcc(*'mp4v'), 30.0,
   ↪  (960,540))
23 vwriter_B = cv2.VideoWriter(sys.argv[4], cv2.VideoWriter_fourcc(*'mp4v'), 30.0,
   ↪  (960,540))
24
25 # load MobileNetSSD object detection algorithm
26 mobile_net_ssd = cv2.dnn.readNetFromCaffe('MobileNetSSD.prototxt.txt',
   ↪  'MobileNetSSD.caffemodel')
27 object_classes = ('background', 'aeroplane', 'bicycle', 'bird', 'boat',
   ↪  'bottle', 'bus', 'car', 'cat', 'chair', 'cow', 'diningtable', 'dog',
   ↪  'horse', 'motorbike', 'person', 'pottedplant', 'sheep', 'sofa', 'train',
   ↪  'tvmonitor')
28 object_classes_wanted = {'bicycle','cat','cow','dog','horse','person','sheep'}
29
30 # global data storage (saved to file at the end)
31 data = {
32   # <frame_number>: {objects: [{object_class, X, intersection_angle}, ..],
   ↪  unpaired_detections, processing_time}
33   # ..
34 }
35
36 # iterate video frames
37 frame_count = 0
38 frame_count_end = int(vcap_A.get(cv2.CAP_PROP_FRAME_COUNT))
39 while vcap_A.isOpened() and vcap_B.isOpened() and frame_count<frame_count_end:
40
41   # start time measure
42   timestamp_start = time.time()
43
44   # read next frame
45   vcap_A.set(cv2.CAP_PROP_POS_FRAMES, frame_count)
46   vcap_B.set(cv2.CAP_PROP_POS_FRAMES, frame_count)
47   _, frame_A = vcap_A.read()
48   _, frame_B = vcap_B.read()
49
50   # apply MobileNetSSD object detection
51   blob_A = cv2.dnn.blobFromImage(frame_A, 0.007843, (1920, 1080), 127.5)
52   mobile_net_ssd.setInput(blob_A)
53   detections_A = mobile_net_ssd.forward()
54   detections_A = [ {
55     'object_class': object_classes[int(detection[1])],
```

```
    'confidence': detection[2],
    'startX_percentage': detection[3],
    'startY_percentage': detection[4],
    'endX_percentage': detection[5],
    'endY_percentage': detection[6]
  } for detection in detections_A[0][0] ]
  detections_A = list_filter(detections_A, lambda detection:
  ↪  detection['confidence']>0.65 and detection['object_class'] in
  ↪  object_classes_wanted)

  blob_B = cv2.dnn.blobFromImage(frame_B, 0.007843, (1920, 1080), 127.5)
  mobile_net_ssd.setInput(blob_B)
  detections_B = mobile_net_ssd.forward()
  detections_B = [ {
    'object_class': object_classes[int(detection[1])],
    'confidence': detection[2],
    'startX_percentage': detection[3],
    'startY_percentage': detection[4],
    'endX_percentage': detection[5],
    'endY_percentage': detection[6]
  } for detection in detections_B[0][0] ]
  detections_B = list_filter(detections_B, lambda detection:
  ↪  detection['confidence']>0.65 and detection['object_class'] in
  ↪  object_classes_wanted)

  # show detections, write to file stream
  cv2.imshow('frame A', cv2.resize(draw_detections(frame_A, detections_A),
  ↪  (480,270)))
  cv2.imshow('frame B', cv2.resize(draw_detections(frame_B, detections_B),
  ↪  (480,270)))
  vwriter_A.write(cv2.resize(draw_detections(frame_A, detections_A),
  ↪  (960,540)))
  vwriter_B.write(cv2.resize(draw_detections(frame_B, detections_B),
  ↪  (960,540)))

  # load calibration
  calibration = get_calibration()
  locals().update(calibration)

  # add ray directions to the detections
  for detection_A in detections_A:
    detection_A['a'] = np.reshape(double_dot( RT_A, KI_A ,
    ↪  to_homogeneous(center_point(detection_A,frame_A)) ),3)
  for detection_B in detections_B:
    detection_B['b'] = np.reshape(double_dot( RT_B, KI_B ,
    ↪  to_homogeneous(center_point(detection_B,frame_B)) ),3)

  # correspondence matching (by image class and coplanarity check (gives
  ↪  intersection angle alpha))
```

```
94    # and triangulation
95    data[frame_count] = {"objects": [], "unpaired_detections": 0}
96    for detection_A in detections_A:
97      detections_B_copy = detections_B[:]
98      detections_B_copy = list_filter(detections_B_copy, lambda detection_B:
        ↪  detection_B['object_class']==detection_A['object_class'])
99      for detection_B in detections_B_copy:
100       detection_B['intersection_angle'] = intersection_angle(X0_A, X0_B,
          ↪  detection_A['a'], detection_B['b'])
101     detections_B_copy = list_filter(detections_B_copy, lambda detection_B:
        ↪  np.rad2deg(detection_B['intersection_angle']) < 1.0)
102     if len(detections_B_copy) == 0:
103       data[frame_count]['unpaired_detections'] += 1
104     else:
105       detections_B_copy.sort(key = lambda detection_B:
          ↪  detection_B['intersection_angle'])
106       detection_B = detections_B_copy[0]
107       X = triangulate(X0_A, X0_B, detection_A['a'], detection_B['b'])
108       data[frame_count]['objects'].append({"object_class":
          ↪  detection_A['object_class'], "X": tuple(X), "intersection_angle":
          ↪  detection_B['intersection_angle']})
109       detections_B.remove(detection_B)
110   data[frame_count]['unpaired_detections'] += len(detections_B)
111
112   # end time measure
113   timestamp_end = time.time()
114   data[frame_count]['processing_time'] = timestamp_end-timestamp_start
115
116   # log
117   print("done frame "+str(frame_count))
118
119   # next frame
120   frame_count += 1
121   cv2.waitKey(1)
122
123
124 # write data to json file
125 data_file = open(sys.argv[5], 'w+')
126 data_file.write(json.dumps(data, indent=4))
127 data_file.close()
128
129 # clean up
130 vcap_A.release()
131 vcap_B.release()
132 vwriter_A.release()
133 vwriter_B.release()
134 cv2.destroyAllWindows()
```

helper_functions.py

```python
# --TASK-- defines functions that may be used across all scripts

import cv2, numpy as np
from functools import lru_cache

# --- Coordinate Type Conversion ----------
def to_homogeneous(x):
    return np.append(x, [1])

def to_euclidian(x):
    return np.array([x[i]/x[-1] for i in range(len(x)-1)])

def spherical_to_euclidian(X_spheric):
    r, theta, phi = X_spheric
    return np.array([
        r * np.sin(theta) * np.cos(phi),
        r * np.sin(theta) * np.sin(phi),
        r * np.cos(theta)
    ])

def euclidian_to_spherical(X):
    r = np.linalg.norm(X)
    theta = np.arccos(X[2]/r)
    phi = np.arctan(X[1]/X[0])
    return r, theta, phi


# --- Coordinate System Conversion ----------
@lru_cache()
def get_L(theta, phi):
    return np.array([
        [-np.sin(phi),   -np.cos(phi)*np.cos(theta),
        ↪  np.cos(phi)*np.sin(theta) ],
        [ np.cos(phi),   -np.sin(phi)*np.cos(theta),
        ↪  np.sin(phi)*np.sin(theta) ],
        [ 0,              np.sin(theta),               np.cos(theta)
        ↪  ]
    ])

def local_to_global(OaX, Oa_spheric):
    ObOa_ = spherical_to_euclidian(Oa_spheric)
    L = get_L(Oa_spheric[1], Oa_spheric[2])
    ObX_ = np.dot(L, OaX) + ObOa_
    return euclidian_to_spherical(ObX_)
def global_to_local(X_spheric, Oa_spheric):
    ObX_ = spherical_to_euclidian(X_spheric)
```

```
    ObOa_ = spherical_to_euclidian(Oa_spheric)
    L = get_L(Oa_spheric[1], Oa_spheric[2])
    OaX = np.dot(L.T, ObX_ - ObOa_)
    return OaX


# --- Object Detection ----------
def detection_coordinates(detection, image):
  image_height, image_width = image.shape[:2]
  startX = int(detection['startX_percentage']*image_width)
  startY = int(detection['startY_percentage']*image_height)
  endX = int(detection['endX_percentage']*image_width)
  endY = int(detection['endY_percentage']*image_height)
  return startX, startY, endX, endY

def center_point(detection, image):
  startX, startY, endX, endY = detection_coordinates(detection, image)
  return (startX+endX)/2, (startY+endY)/2

def draw_detections(image, detections):
  for detection in detections:
    label = '{}: {}%'.format(detection['object_class'],
    ↪  int(detection['confidence']*100))
    startX, startY, endX, endY = detection_coordinates(detection, image)
    cv2.rectangle(image, (startX, startY), (endX, endY), color=(255,0,0),
    ↪  thickness=2)
    cv2.putText(image, label, (startX, startY-10),
    ↪  fontFace=cv2.FONT_HERSHEY_SIMPLEX, fontScale=1.0, color=(255,0,0),
    ↪  thickness=2)
  return image


# --- Ray Geometry ----------
def ray_endpoint(X0, KI, RT, x, length=1):
  return X0 + vector_to_length(-double_dot( RT , KI , to_homogeneous(x) ),
  ↪  length)

def triangulate(X0_A, X0_B, a, b):
  la, mu = np.linalg.solve(np.array([
    [np.dot(a,a), -np.dot(a,b)],
    [np.dot(a,b), -np.dot(b,b)]
  ]), np.array([
    np.dot(X0_B-X0_A, a),
    np.dot(X0_B-X0_A, b)
  ]))
  return 0.5*(X0_A + la*a + X0_B + mu*b)

def intersection_angle(X0_A, X0_B, a, b):
  alpha = np.arcsin(
```

```
    np.dot( np.cross( a , X0_B-X0_A) , b )
    / np.linalg.norm( np.cross( a , X0_B-X0_A) )
    / np.linalg.norm( b )
  )
  return abs(alpha)


# --- Linear Algebra ----------
def double_dot(a,b,c):
  return np.dot(a,np.dot(b,c))

def vector_to_length(vector, length):
  return vector / np.linalg.norm(vector) * length


# --- Array Functions ----------
def list_filter(the_list, filter_function):
  return [ item for item in the_list if filter_function(item) ]
```

average_processing_time.py

```
# --TASK-- determines the average processing time of frame pair based on the
↪  datafile

# command line arguments:
#     <source_datafile_path>

import sys, json

# counters
duration = 0
number = 0

# load data
data_file = open(sys.argv[1], 'r')
data = json.loads(data_file.read())
data_file.close()

# iterate frames
for framecount, frame_data in data.items():
  duration += frame_data['processing_time']
  number += 1

# output
print('average processing time:', duration/number)
```

Python-Skripte für Blender

generate_mesh.py

```
# --TASK-- generates a blender mesh of vertices and edges

import bpy

def generate_mesh(vertices, edges, color, render_vertices=True,
↪  vertex_size_factor=1.0, single_frame=False):
  # argument:  vertices = [ (X,Y,Z,label), ... ]
  # argument:  edges = [ (label1,label2), ... ]

  # generate mesh object
  mesh = bpy.data.meshes.new('mesh')
  mesh_obj = bpy.data.objects.new("MeshObject", mesh)

  # link object into scene
  scene = bpy.context.scene
  scene.objects.link(mesh_obj)

  # add vertices
  for vertex in vertices:
    mesh.vertices.add(1)
    mesh.vertices[-1].co = vertex[:3]

  # add edges
  for edge in edges:
    for vertex in vertices:
      if len(vertex)>3 and vertex[3]==edge[0]:
        index_point_1 = vertices.index(vertex)
      if len(vertex)>3 and vertex[3]==edge[1]:
        index_point_2 = vertices.index(vertex)
    mesh.edges.add(1)
    mesh.edges[-1].vertices = (index_point_1, index_point_2)

  # generate particle system for vertices
  if render_vertices:
    bpy.ops.mesh.primitive_uv_sphere_add(size=1.0, location=(0.0, 0.0, 0.0))
    uv_sphere = bpy.context.active_object
    material = bpy.data.materials.new(name="UVSphereParticleMaterial")
    material.diffuse_color = color
    uv_sphere.data.materials.append(material)
    uv_sphere.layers[1] = True
    uv_sphere.layers[0] = False
```

```
    particle_system_modifier = mesh_obj.modifiers.new(name="VertexParticles",
    ↪  type="PARTICLE_SYSTEM")
    particle_system_modifier.particle_system.settings.type = 'HAIR'
    particle_system_modifier.particle_system.settings.use_advanced_hair = True
    particle_system_modifier.particle_system.settings.count = len(vertices)
    particle_system_modifier.particle_system.settings.emit_from = 'VERT'
    particle_system_modifier.particle_system.settings.use_emit_random = False
    particle_system_modifier.particle_system.settings.render_type = 'OBJECT'
    particle_system_modifier.particle_system.settings.dupli_object = uv_sphere
    particle_system_modifier.particle_system.settings.particle_size =
    ↪  0.025*vertex_size_factor


  # generate wire material for edges
  if len(edges)>0:
    material = bpy.data.materials.new(name='WireMaterial')
    material.type = 'WIRE' #'SURFACE'
    material.diffuse_color = color
    material.use_shadeless = True
    mesh.materials.append(material)

  # update
  mesh.update()

  # single frame
  if single_frame:
    scene.frame_set(scene.frame_current-1)
    mesh_obj.hide = True
    mesh_obj.keyframe_insert(data_path='hide')
    mesh_obj.hide_render = True
    mesh_obj.keyframe_insert(data_path='hide_render')

    scene.frame_set(scene.frame_current+2)
    mesh_obj.hide = True
    mesh_obj.keyframe_insert(data_path='hide')
    mesh_obj.hide_render = True
    mesh_obj.keyframe_insert(data_path='hide_render')

    scene.frame_set(scene.frame_current-1)
    mesh_obj.hide = False
    mesh_obj.keyframe_insert(data_path='hide')
    mesh_obj.hide_render = False
    mesh_obj.keyframe_insert(data_path='hide_render')

```

visualize_calibration_points.py

```python
# --TASK-- visualize calibration points in blender scene

from calibration_points import get_calibration_points
from generate_mesh import generate_mesh

# load calibration points
calibration_points = get_calibration_points()

# generate mesh
vertices = []
edges = []
for cp in calibration_points:
    vertices.append(cp['X'])
generate_mesh(vertices, edges, color=(1.0, 0.0, 0.0), vertex_size_factor=2.0)
```

visualize_cameras.py

```python
# --TASK-- visualize cameras in blender scene

from calibration import get_calibration
from helper_functions import ray_endpoint
from generate_mesh import generate_mesh

# load calibration
calibration = get_calibration()
locals().update(calibration)

# generate mesh for camera A
vertices = []
edges = []
vertices.append( tuple(calibration['X0_A']) + ('X0_A',) )
vertices.append( tuple(ray_endpoint(X0_A, KI_A, RT_A, x=(0,0), length=2)) + ('topleft_ray_A',) )
vertices.append( tuple(ray_endpoint(X0_A, KI_A, RT_A, x=(0,1080), length=2)) + ('bottomleft_ray_A',) )
vertices.append( tuple(ray_endpoint(X0_A, KI_A, RT_A, x=(1920,1080), length=2)) + ('bottomright_ray_A',) )
vertices.append( tuple(ray_endpoint(X0_A, KI_A, RT_A, x=(1920,0), length=2)) + ('topright_ray_A',) )
edges.append(('X0_A','topleft_ray_A'))
edges.append(('X0_A','bottomleft_ray_A'))
edges.append(('X0_A','bottomright_ray_A'))
edges.append(('X0_A','topright_ray_A'))
```

```
edges.append(('topleft_ray_A','bottomleft_ray_A'))
edges.append(('bottomleft_ray_A','bottomright_ray_A'))
edges.append(('bottomright_ray_A','topright_ray_A'))
edges.append(('topright_ray_A','topleft_ray_A'))
generate_mesh(vertices, edges, color=(0.0, 1.00, 1.00))

# generate mesh for camera B
vertices = []
edges = []
vertices.append( tuple(calibration['X0_B']) + ('X0_B',) )
vertices.append( tuple(ray_endpoint(X0_B, KI_B, RT_B, x=(960,540),
    length=10000)) + ('center_ray_B',) )
vertices.append( tuple(ray_endpoint(X0_B, KI_B, RT_B, x=(0,0), length=2)) +
    ('topleft_ray_B',) )
vertices.append( tuple(ray_endpoint(X0_B, KI_B, RT_B, x=(0,1080), length=2)) +
    ('bottomleft_ray_B',) )
vertices.append( tuple(ray_endpoint(X0_B, KI_B, RT_B, x=(1920,1080), length=2))
    + ('bottomright_ray_B',) )
vertices.append( tuple(ray_endpoint(X0_B, KI_B, RT_B, x=(1920,0), length=2)) +
    ('topright_ray_B',) )
edges.append(('X0_B','center_ray_B'))
edges.append(('X0_B','topleft_ray_B'))
edges.append(('X0_B','bottomleft_ray_B'))
edges.append(('X0_B','bottomright_ray_B'))
edges.append(('X0_B','topright_ray_B'))
edges.append(('topleft_ray_B','bottomleft_ray_B'))
edges.append(('bottomleft_ray_B','bottomright_ray_B'))
edges.append(('bottomright_ray_B','topright_ray_B'))
edges.append(('topright_ray_B','topleft_ray_B'))
generate_mesh(vertices, edges, color=(0.0, 1.0, 0.0))
```

visualize_data.py

```
# --TASK-- visualize located objects based on datafile

import bpy, json
from calibration import get_calibration
from generate_mesh import generate_mesh

# load data
data_file = open('<source_datafile_path>', 'r')
data = json.loads(data_file.read())
data_file.close()

# iterate frames in data
for frame_count, frame_data in data.items():
```

```
14
15    # generate mesh for located objects
16    vertices = []
17    edges = []
18    for obj in frame_data['objects']:
19      vertices.append(tuple(obj['X']))
20    bpy.context.scene.frame_set(int(frame_count))
21    generate_mesh(vertices, edges, color=(0.9, 1.0, 0.0), vertex_size_factor=4.0,
      ↪  single_frame=True)
22
23    # generate mesh for camera rays
24    calibration = get_calibration()
25    for obj in frame_data['objects']:
26      vertices = [
27        tuple(calibration['X0_A']) + ('X0_A',),
28        tuple(calibration['X0_B']) + ('X0_B',),
29        tuple(obj['X']) + ('X',)
30      ]
31      edges = [
32        ('X0_A','X'),
33        ('X0_B','X'),
34      ]
35      generate_mesh(vertices, edges, color=(0.9, 1.0, 0.0),
        ↪  render_vertices=False, single_frame=True)
36
37  # set keyframe range of animation
38  bpy.context.scene.frame_start = 0
39  bpy.context.scene.frame_end = len(data.keys())-1
```