HAW HAMBURG

BACHELORTHESIS
Ruben Horn

Outdoor Objekt-Detektion und -Tracking von Verkehrsteilnehmern für Augmented Reality

FAKULTÄT TECHNIK UND INFORMATIK
Department Informatik

Faculty of Computer Science and Engineering
Department Computer Science

HOCHSCHULE FÜR ANGEWANDTE WISSENSCHAFTEN HAMBURG
Hamburg University of Applied Sciences

Ruben Horn

Outdoor Objekt-Detektion und -Tracking von Verkehrsteilnehmern für Augmented Reality

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang Bachelor of Science Technische Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Philipp Jenke
Zweitgutachter: Prof. Dr.-Ing Andreas Meisel

Eingereicht am: 28.08.2020

Ruben Horn

Thema der Arbeit

Outdoor Objekt-Detektion und -Tracking von Verkehrsteilnehmern für Augmented Reality

Stichworte

Augmented Reality, Objekt-Detektion, Objekt-Tracking

Kurzzusammenfassung

In den letzten Jahren ist eine Entwicklung im Bereich Augmented Reality von reinen technischen Demonstrationen zu großflächiger Verbreitung durch Smartphone-Anwendungen zu beobachten. Auch bei fortschreitender Innovation im Hardware-Bereich ähneln sich diese Anwendungen stark in ihrem Funktionsumfang. Eine Revolution des Umgangs mit Computern durch Augmented Reality ist noch nicht eingetreten. Mit gleichzeitigem Fortschritt im Bereich Computer Vision und speziell Objekt-Detektion, äußert sich hier eine Chance für dynamischere Anwendungen unter Verwendung beider Technologien. Im Rahmen dieser Thesis wird ein Prototyp entwickelt um zu ermitteln, ob solche Anwendungen mit bestehenden Hardware- und Software-Komponenten möglich sind. Dabei werden zwei Lösungen für Objekt-Tracking umgesetzt und evaluiert. Als Szenario stellt dabei eine ungesehene Verkehrssituation eine ambitionierte Herausforderung dar. Bei der Evaluierung stellt sich heraus, dass dies im Prinzip möglich ist. Für die Umsetzung einer robusten Anwendung muss jedoch die Konsistenz der Objekterkennung verbessert werden.

Ruben Horn

Title of Thesis

Outdoor object detection and tracking of road users for Augmented Reality

Keywords

Augmented Reality, Object Detection, Object Tracking

Abstract

Over the last couple of years, developments in the area of Augmented Reality have seen a shift from mere tech demos to widespread adoption in smartphone applications. Despite continuous hardware innovation, these applications are very similar in terms of functionality. However, a revolution of how we operate computers with Augmented Reality has yet to occur. Advancements in the area of Computer Vision and specifically Object Detection present a chance for more dynamic applications by combining both technologies. This thesis aims to answer the question, whether such an application is possible using existing hardware and software components. For that, a prototype, including two solutions for Object Tracking, will be realized and evaluated. In this context, an unseen traffic situation presents an ambitious challenge. The evaluation shows that such applications are technically possible. However, the realization of a robust application requires more consistent Object Detection.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

Augmented Reality (AR) hat in den letzten Jahren zunehmend an Aufmerksamkeit gewonnen. Dies geht einher mit deutlichen Fortschritten bei Head-up-Displays wie zuletzt der Microsoft Hololens 2[1], aber vor allem mit dem stetig wachsende Angebot an Smartphone-basierten AR-Inhalten. Das AR für Smartphones mittlerweile eine wichtige Rolle spielt wurde vor allem 2017 bestätigt, als die führenden Betriebssysteme iOS und Android eigene Plattformen (ARKit und ARCore) für solche Apps mit jeweils ähnlichem Funktionsumfang einführten.[2]

Besonders hervorzuheben ist auch das 2016 veröffentlichte Smartphone-Spiel Pokemon Go[3]. Wie in Abbildung 1.1 zu sehen, werden dabei Spielfiguren in der Umgebung des Spielers dargestellt, wodurch die Spielwelt mit der Realität verschmilzt. Darüber hinaus bieten weit verbreitete Apps von soziale Netzwerke wie Snapchat oder Instagram AR-Bildeffekte und proprietäre Werkzeuge zur Erstellung solcher Inhalte an.[4]

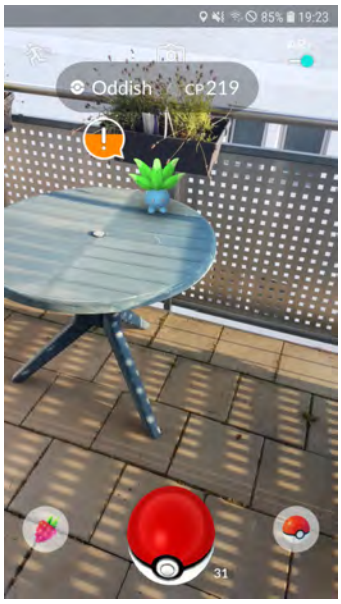


Abbildung 1.1: Screenshot aus der Android-Version von Pokemon Go

[1] https://www.microsoft.com/en-us/hololens/ (Abgerufen am 15.07.2020)
[2] ARKit wurde auf der Sitzung 602 der Apple WWDC 2017 vorgestellt (https://developer.apple.com/videos/play/wwdc2017/602/ Abgerufen am 30.07.2020), ARCore auf den GDD Europe 2017 (https://youtu.be/rFbcOGuDMPk Abgerufen am 30.07.2020)
[3] https://www.pokemongo.com/en-us/ (Abgerufen am 15.07.2020)
[4] https://www.indestry.com/blog/2019/10/11/the-brief-history-of-social-media-ar-filters (Abgerufen am 15.07.2020)

1.1 Motivation

Aktuell ist die Funktionalität dieser und anderer AR-Anwendungen jedoch noch eingeschränkt. Die Überlagerung mit digitalen 3D-Inhalten findet bisher nur für einzelne, vorher bekannte Objekte statt. Diese dürfen dabei nicht von den in der App hinterlegten Eigenschaften abweichen. Eine Ausnahme stellt dabei die Erkennung vom Gesicht des Benutzers in den erwähnten AR-Bildeffekten dar. Die Plattformen ARKit und ARCore ermöglichen zudem die Erkennung der unmittelbaren Topografie und der relativen Position des Geräts. Bewegliche oder bewegte Objekten können dabei von dieser nicht unterschieden werden. Dadurch sind solche Anwendungen auf die freie Platzierung digitaler 3D-Inhalte durch den Benutzer oder die digitale Erweiterung weniger Objekte wie z.B. Printmedien beschränkt.
Heterogene Objekte können noch nicht mit AR-Inhalten angereichert werden. Zudem kann eine solche Anwendung dynamische räumliche Veränderungen wie beispielsweise das Verschieben eines Bürostuhls oder gar eine Verkehrssituation nicht erkennen.

Diese Thesis ist in Anlehnung an den Antrag „Visualisierung und Augmented Reality“ im Projekt „Mikromobilität“ der Fakultät Technik und Informatik an der HAW Hamburg entstanden. In diesem Rahmen soll prototypisch eine Smartphone-Anwendung umgesetzt werden, mit welcher Verkehrsteilnehmer wie Autos, Fußgänger, Radfahrer und ähnliche in AR eingebunden werden können. Ziel ist es festzustellen, wie gut sich mit bestehenden Werkzeugen für die Entwicklung von AR-Anwendungen und Verfahren aus dem Bereich der Computer Vision dynamischere Anwendungsfälle umsetzen lassen können. Speziell geht es darum, ob solche Anwendung im Freien, in urbanen Szenarien möglich sind. Dabei geht es nicht um die Umsetzung eines konkreten Anwendungsfalls. Als Zielplattform wird sich dabei auf handelsübliche Smartphones ohne spezielle Hardware-Komponenten konzentriert.

2 Grundlagen

Im diesem Kapitel werden die wichtigsten Grundlagen zu Augmented Reality und Computer Vision beschrieben. Dabei wird auf die Aufgaben Erkennung und Verfolgung von Objekten (Objekt-Detektion und -Tracking) auf diesem Gebiet eingegangen, welche für diese Thesis relevant sind.

2.1 Computer Vision

Computer Vision (CV) beschreibt ein Teilgebiet der Informatik, das sich mit der Gewinnung von Information aus Bilddaten beschäftigt. Dabei kann es sich etwa um die Position und Bewegung abgebildeter Objekte oder der Kamera in einer Bildsequenz handeln oder um die semantische Bedeutung eines Fotos.[1] Dabei darf CV nicht mit Mustererkennung oder Image Processing, der digitalen Verarbeitung von Bildmaterial zur Umwandlung, Restauration, Komprimierung und Ähnlichem, verwechselt werden. Einige Techniken aus Letzterem finden jedoch auch in CV Anwendung [40, Kapitel 1].

2.2 Augmented Reality

Unter dem Begriff Augmented Reality (AR) werden alle Software-Anwendungen zusammengefasst, welche die Wahrnehmung der physikalischen Umwelt des Benutzers mit digitalen Inhalten anreichern. Dies geschieht meist durch die Einblendung von virtuellen Objekten im Sichtfeld des Benutzers.

[1] Semantic image understanding : from the web, in large scale, with real-world challenging data `https://purl.stanford.edu/qk372kq7966` (Abgerufen am 15.06.2020)

Im Vergleich zu Augmented Virtuality (AV) und Virtual Reality (rein virtueller Umwelt) macht die physikalische Umwelt also bei AR den Hauptteil der sichtbaren Umgebung aus [1, 27, 8]. (vgl. Abbildung 2.1) Dabei können sowohl die physikalische Umwelt als auch der Benutzer durch Interaktion mit der Software Einfluss auf die Anwendung nehmen. Bei den meisten AR-Anwendungen im Endverbraucherbereich handelt es sich um Smartphone-Anwendungen, welche sich als „video see-through“ klassifizieren lassen. Im Gegensatz zu Systemen mit „optical see-through“ wie zum Beispiel der Microsoft Hololens 2 werden die digitalen Inhalte dabei nicht direkt in das Sichtfeld des Benutzers projiziert, sondern über das Kameravorschaubild des Smartphones gelegt [1]. Somit nutzen AR-Anwendungen auch Techniken aus dem Bereich CV. Innerhalb der Kategorie Smartphone-AR kann aus Sicht eines Benutzers zwischen Marker-basierten und Marker-losen Anwendungen unterschieden werden.

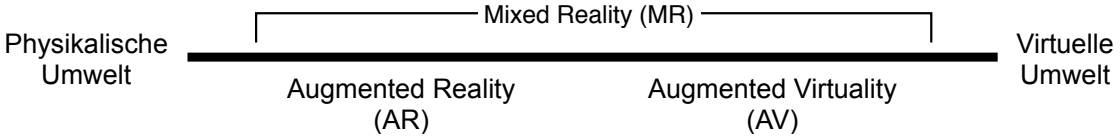


Abbildung 2.1: Realität-Virtualität-Kontinuum nach Milgram et al. [27]

2.2.1 Marker-basierte Anwendungen

Ein bewährtes Verfahren für AR ist die Verwendung von Markern zur Bestimmung der Position, Größe und Ausrichtung virtueller 3D-Objekte relativ zum Betrachter. Als Marker werden entweder Flächen mit einem markanten Muster wie Logos, Plakate u.Ä. oder räumliche Objekte wie z.B. Spielzeugmodelle verwendet. Aus markanten Eckpunkten in den Graustufenbildern dieser Marker können wiedererkennbare Merkmale extrahiert werden. Mit diesen kann die Position des virtuellen Objekts so bestimmt werden, dass es realistisch auf dem Marker platziert scheint. Dabei muss es sich immer um den gleichen Marker handeln, damit die selben Merkmale erkannt und somit die virtuellen Objekte korrekt eingefügt werden. Neben einer Kamera wird keine spezielle Hardware benötigt. Allerdings muss der Marker dauerhaft im Sichtfeld der Kamera sein, was die Bewegungsfreiheit des Benutzers einschränkt. Zudem erlauben einige Anwendungsfälle die Verwendung vorgefertigter Marker nicht. Bibliotheken zur Entwicklung Marker-basierter Anwendungen sind unter anderem Vuforia[2], ARToolKit[3] und Wikitude[4].

[2] `https://developer.vuforia.com/` (Abgerufen am 15.07.2020)
[3] `http://www.artoolkitx.org/` (Abgerufen am 15.07.2020)
[4] `https://www.wikitude.com/` (Abgerufen am 15.07.2020)

2.2.2 Marker-lose Anwendungen

Um virtuelle Objekte ohne die Verwendung von Markern in der physikalischen Umwelt zu verankern, müssen die Bewegungen des Beobachters (des Smartphones) anhand von Sensordaten erfasst und ausgeglichen werden. In den eingangs erwähnten AR-Bibliotheken ARKit und ARCore für iOS- und Android-Smartphones kommt dafür Visual-Inertial Odometry (VIO)[5] beziehungsweise Simultaneous Localization and Mapping (SLAM)[6] zum Einsatz. Bei VIO wird auf Basis einer Bildsequenz und Bewegungsdaten eines Beschleunigungssensors die Veränderung der Position und Orientierung des Geräts ermittelt. Aus dieser Bewegung kann die Transformation zum Ursprung berechnet werden. Durch die Anwendung der Inversen dieser Transformation auf die virtuellen Objekte werden diese immer an derselben Position in der Umwelt dargestellt. Bei SLAM wird zusätzlich laufend eine Karte aus der erfassten Umgebung erzeugt, in welcher das Gerät verortet wird [44]. Es stellen beide Bibliotheken rudimentäre Informationen über die physikalischen Umgebung in Form von erkannten Ebenen bereit. Im Bereich der Tourismus-Anwendungen kann zudem, auf die mit GPS bereitgestellte Position des Smartphones zugegriffen werden, um zu ermitteln, welche virtuellen Inhalte angezeigt werden sollen, wie Wedyan et al. [42] demonstrieren.

[5] Understanding World Tracking | Apple Developer Documentation `https://developer.apple.com/documentation/arkit/world_tracking/understanding_world_tracking` (Abgerufen 18.06.2020)

[6] Fundamental concepts | ARCore | Google Developers `https://developers.google.com/ar/discover/concepts` (Abgerufen 18.06.2020)

2.3 Objekt-Detektion

Objekt-Detektion ist eine Aufgabe aus dem Gebiet der CV. Dabei sollen markante Objekte oder solche, die zu einer vorab definierten Klasse wie Person, Fahrzeug, Katze, etc. gehören, in einem Bild erkannt und lokalisiert werden. Die Objekte können dabei innerhalb der jeweiligen Klasse unterschiedliche Ausprägungen besitzen, geometrisch transformiert, teilweise verdeckt oder durch die Beleuchtung beeinflusst sein. Mit einer naiven Suche nach Vorlagen kann dieses Problem also nicht gelöst werden. In Szenarien, in denen die Umgebung sowie die Kamera statisch sind, kann das Teilproblem der Lokalisierung dynamischer Objekte mittels Hintergrundsubtraktion gelöst werden. In einer dynamischen oder unbekannten Umwelt oder bei Anwendungen mit einer bewegten Kamera, wie im Kontext dieser Thesis, sind komplexere Verfahren erforderlich.

2.3.1 Konventionelle Verfahren

Für die Erkennung von Objekten in digitalen Bildern wurden verschiedene Ansätze entwickelt. Eine etablierte Technik ist die Verwendung von Merkmal-Extraktion auf Basis allgemeiner Algorithmen in Verbindung mit einem kaskadierenden Klassifizierer, der auf die Erkennung verschiedener Merkmale trainiert wird. Sobald eine Stufe des Klassifizierers ein Fenster ablehnt, wird die Suche darin abgebrochen. Implementiert wird dieser oft mit der Stützvektormethode. Zur Erkennung gleitet ein Fenster über das Bild, auf welches der trainierte Klassifizierer angewendet wird. Um Objekte verschiedener Größen zu erkennen wird die Klassifizierung auf Merkmals-Karten mit absteigender Auflösung (Merkmal-Pyramiden) angewendet [45].

2.3.2 Convolutional Neural Networks

Seit einigen Jahren werden verstärkt künstliche, neuronale Netzwerke verwendet, um dieses Problem zu lösen. Im Bereich CV werden dabei vor allem neuronale Netzwerke eingesetzt, welche zur Extraktion von Bildmerkmalen Faltung verwenden. Bei Faltung werden die Pixel-Werte in einem Fenster von der Größe des Faltungskerns (w, h) mit diesem elementweise multipliziert und die Ergebnisse aufsummiert. Der Wert an der Stelle (x, y) im Fenster wird mit dem Wert an der Stelle $(w-x, h-y)$ des Faltungskerns multipliziert. Das Fenster wird schrittweise über das Bild bewegt und der Vorgang wiederholt. So entsteht als Ergebnis wieder ein Bild.

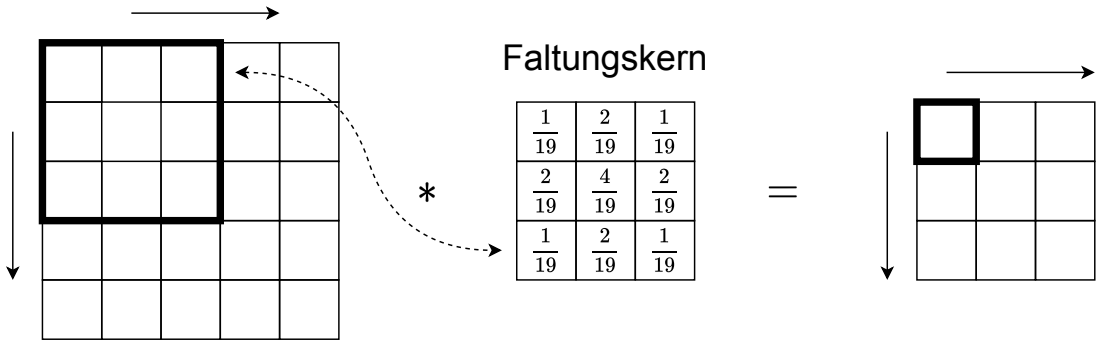


Abbildung 2.2: Weichzeichnen durch Anwendung eines entsprechenden Faltungskerns

Die Anwendung dieser Operation auf einem Bild ist in Abbildung 2.2 exemplarisch dargestellt.
Die bei dieser Operation angewendeten Filter werden beim Trainieren des Netzwerks ermittelt. Durch weitere Faltungs-Schichten lassen sich zunehmend komplexere Merkmale erkennen. Ein solches neuronales Netz wird als Convolutional Neural Network (CNN) bezeichnet [11, Kapitel 9]. Neben Faltungsebenen kann auch eine Vielzahl anderer Operationen in einem solchen Netz zum Einsatz kommen. Als Auslöser für das gesteigerte Interesse an CNNs kann das überragende Ergebnis des AlexNet bei der ImageNet Large Scale Visual Recognition Challenge (ILSVRC) im Jahr 2012 [37] angesehen werden. Durch die Verwendung von fünf Faltungs-Schichten und drei vollständig vernetzten Schichten konnte eine Fehlerrate von nur 15,3% erreicht werden [19], während Andere Fehlerraten von nicht unter 26,2% erreichten.[7] Mithilfe solcher Klassifizierungs-Netze lässt sich auch Objekt-Detektion implementieren. In diesem Kontext werden sie als das „Backbone“ bezeichnet und stehen vor dem eigentlichen Netzwerk für die Erkennung.

R-CNN

Eine Anwendung davon ist die Familie der Region-based CNN (R-CNN) Algorithmen. Dazu wird zunächst ermittelt, welche Ausschnitte eines Bildes wahrscheinlich ein Objekt enthalten. Diese heißen Region of Interest (ROI). Aus ihnen werden mit Hilfe des „Backbone“-Netzes Merkmals-Karten extrahiert, auf Basis derer die Klasse und der exakte Rahmen des Objekts bestimmt werden können.

[7] `http://image-net.org/challenges/LSVRC/2012/results.html` (Abgerufen am 15.08.2020)

Da dieses Verfahren also in die Ermittlung von ROIs und die Objekterkennung in diesen aufgeteilt ist, wird es als „Two-Stage Object Detection“ bezeichnet. Für Echtzeitanwendungen sind solche Algorithmen jedoch zu langsam, weshalb hier in der Regel ein „One-Stage Object Detection“-Algorithmus verwendet wir, der beides in einem Schritt vereint [45].

YOLO

Eine Vertreter davon ist You Only Look Once (YOLO) [33]. Die Ausgangs-Schicht des Netztes hat eine Ausgabe der Größe $S \times S \times (B * 5 + C)$. Jede Zelle in der Ausgangs-Ebene hängt mit einem Ausschnitt aus dem Eingabe-Bild an der selben relativem Position zusammen. In diesem Raster könnte jede Zelle den Mittelpunkt eines Objekts enthalten. Da die Ausgabe mehrere Ebenen besitzt, kann mehr über eine Zelle ausgesagt werden als nur, ob diese ein Objekt beinhaltet. Neben einer Wahrscheinlichkeit für ein Objekt in der jeweiligen Zelle p_{obj} wird für jede Klasse $c \in C$ eine bedingte Wahrscheinlichkeit $P(c|p_{obj})$ sowie ein das Objekt umfassender Rahmen vorhergesagt. In der ersten Iteration dieser Architektur werden die Erkennungen, wie in Abbildung 2.3 zu sehen, noch direkt durch eine vollständig vernetzte Schicht auf Basis von vieler Merkmals-Karten mit geringer Auflösung vorhergesagt. Weiterentwicklungen der ursprünglichen Version bestehen aus einem vollständigen Faltungsnetz und bestimmen den Rahmen von Objekten relativ zu einem aus fünf Anker-Rahmen der jeweiligen Zelle. Als „Backbone“ kommt hier nicht mehr eine Abwandlung der Googlenet-Architektur, sondern das von Redmon und Farhadi [34] entwickelte Darknet-19 und später Darknet-53 [35] zum Einsatz.

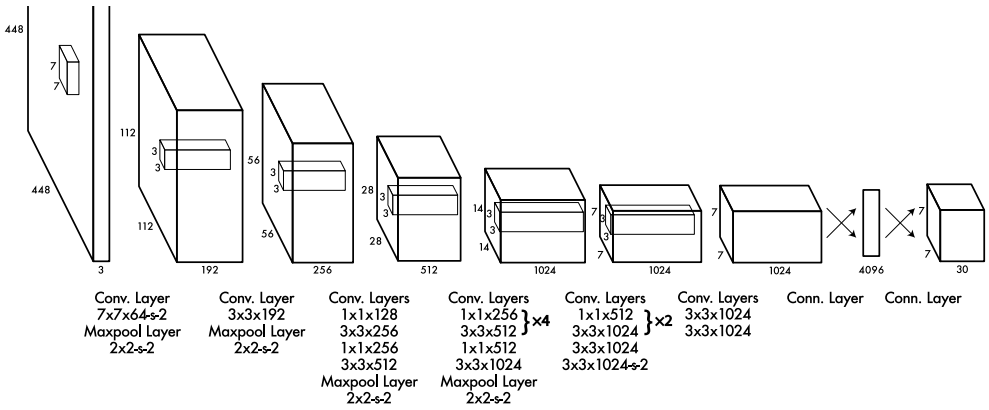


Abbildung 2.3: Ursprüngliche Architektur des YOLO-Netzes (Quelle: Redmon et al. [33])

SSD

Ein weiteres CNN-basiertes Verfahren für „One-Stage Object Detection“ ist der Single Shot MultiBox Detector (SSD) [25]. Dieser verwendet eine feste Anzahl von Vorlagen für Objektrahmen pro Zelle. Relativ zu diesen Vorlagen wird der eines erkannten Objekts bestimmt. Die Wahrscheinlichkeiten für die jeweiligen Objekt-Klassen sind unbedingt, p_{obj} wird nicht bestimmt. Für den Fall, dass in einer Zelle kein erkennbares Objekt existiert, wird eine Platzhalter-Klasse verwendet. Um Objekte verschiedener Größen zu erkennen, wird die Erkennung auf Basis von Merkmals-Karten mit unterschiedlicher Auflösung durchgeführt, wie in Abbildung 2.4 dargestellt.

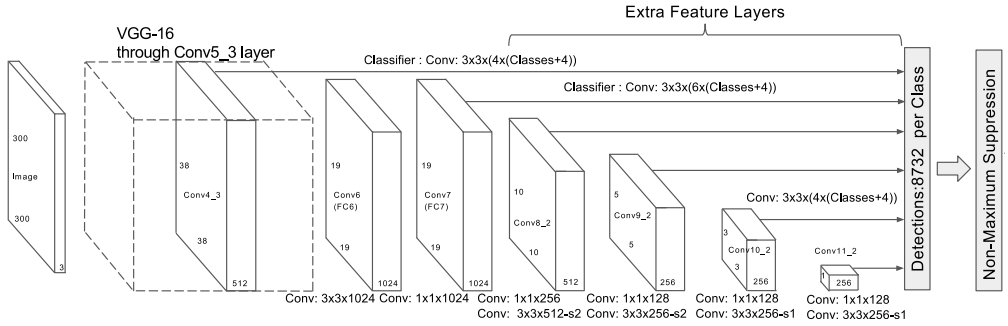


Abbildung 2.4: SSD-Architektur mit VGG-16 als „Backend“ (Quelle: Liu et al. [25])

Optimierung für Low-End-Geräte

Um Objekt-Detektion in Echtzeit Geräten mit geringerer Rechenleistung oder ohne Hardwarebeschleunigung für CNNs zu nutzen, existieren Varianten der beschriebenen Detektoren mit reduzierter Komplexität.
Für YOLO existiert die vereinfachte „tiny“ Architektur, welche anstelle von fünfundsiebzig Faltungsoperationen nur dreizehn besitzt.[8] Die reduzierte Komplexität des CNNs geht allerdings auch mit einer deutlich geringeren Präzision bei der Detektion einher.[9]

[8] Visualisiert in Abbildung A.1
[9] Nach den Angaben auf `https://pjreddie.com/darknet/yolo/` (Abgerufen am 30.07.2020)

Für den SSD kann als „Backbone“ MobileNet [13] verwendet werden. Dabei handelt es sich um eine CNN-Architektur, die primär für Anwendungen auf mobilen Geräten entwickelt wurde. Durch die Aufspaltung der Faltungsoperationen entlang der Bild-Ebenen („pointwise convolution“ folgt „depthwise separable convolution“) wird die Anzahl der Fließkommazahl-Multiplikationen verringert. Howard et al. [13] zeigen zudem, dass die Genauigkeit von Objekt-Detektion mit SSD und MobileNet nur geringfügig niedriger ist als mit anderen „Backbone“-Netzen.
Mit solchen Optimierungen ist es möglich, Objekterkennung auf Basis von CNNs auf Einplatinencomputern[10] oder Smartphones[11] einzusetzen.

2.3.3 3D-Objekt-Detektion

Die bisher erwähnten Verfahren bestimmen die Position und Größe von Objekten in einem Bild als Rechteck, also ohne Translation entlang der Tiefenachse und der Rotation im Raum. Die Bestimmung von Objektrahmen als Quader mit Ausrichtung entlang drei Achsen wird als 3D-Objekt-Detektion bezeichnet und ist vor allem im Bezug auf selbstfahrende Autos relevant. In diesem Kontext können gegebenenfalls zusätzliche Informationen durch Verwendung einer zweiten Kamera oder von Tiefensensoren verarbeitet werden, einige Algorithmen lösen dieses Problem jedoch sogar für ein einfaches Kamerabild. Diese sind allerdings noch zu langsam oder ungenau [43]. Hou et al. [12] zeigten Anfang 2020 das dies sogar auf Smartphones möglich ist. Der beschriebene Detektor erkennt allerdings nur Objekte einer Klasse. Im Rahmen dieser Thesis wird sich somit auf 2D-Objekt-Detektion konzentriert.

[10] object detection, tiny yolov3 on raspberry Pi using NCS2 OpenVINO IR, deep learning `https://youtu.be/Nl6KppDqQnE` (Abgerufen am 18.06.2020)
[11] Realtime Object Detection With SSD MobileNet v3 on Android `https://youtu.be/2vV2SKBhthM` (Abgerufen am 18.06.2020)

2.3.4 Jaccard-Koeffizient

Der Jaccard-Koeffizient, auch IOU, beschreibt die Ähnlichkeit von zwei Mengen [16]. Als Maß für die Überlappung von Objekten ist IOU die am häufigsten verwendete Metrik zur Bewertung von Objekt-Detektion. Für zwei Objekte A und B berechnet sie sich durch $\frac{A \cap B}{A \cup B}$ wie in Abbildung 2.5 dargestellt und hat somit einen Wertebereich von 0 bis 1 [17].

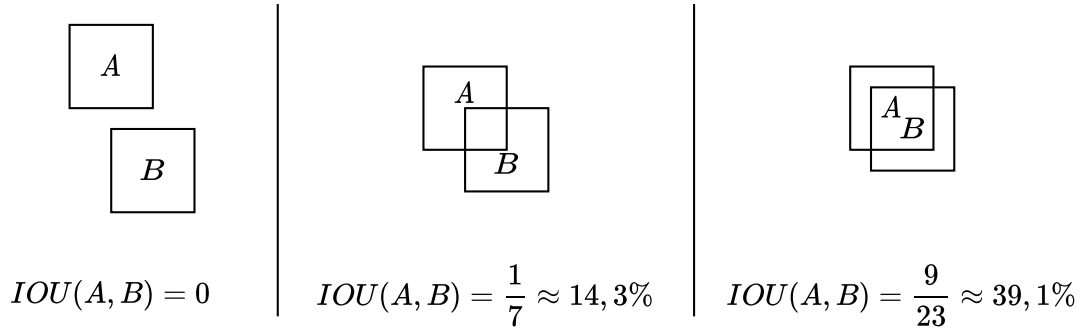


Abbildung 2.5: Beispiele für IOU für Objektrahmen

2.4 Objekt-Tracking

Tracking beschreibt in diesem Kontext die Verfolgung eines Objekts über eine Sequenz von Bildern. Um die Position des Objektes in Echtzeit bestimmen zu können, erfordert der verwendete Tracking-Algorithmus eine ausreichende Verarbeitungsgeschwindigkeit aufweisen. Je schneller sich Objekte bewegen, desto kürzer muss diese also sein.

2.4.1 Visual Object Tracking

Bei VOT geht es um die Bestimmung der Position eines Objekts mit bekannter, initialer Position in allen folgenden Bildern einer Sequenz. Da die Klasse des Objekts nicht bestimmt werden muss und die visuelle Ausprägung des Objekts aus dem ersten beziehungsweise vorangegangenen Bild bekannt ist, gibt es diverse Ansätze um es in folgenden Bildern zu finden. Wang et al. [41] bezeichnen Tracking daher als Such-Aufgabe.

2.4.2 Tracking durch Detektion

Wenn Objekt-Detektion periodisch durchgeführt wird, kann auf Basis der in aufeinander folgenden Bildern erkannten Objekten eine Zuordnung erfolgen. Dieses Verfahren kann eingesetzt werden, um mehrere Objekte zu verfolgen beziehungsweise zuzuordnen, da die genauen Positionen durch die Detektion bereits bekannt sind. Um die Objekte eindeutig zuordnen zu können, muss die Bewegung dieser zwischen den einzelnen Bildern möglichst gering sein [3]. Für Echtzeit-Anwendungen bedeutet das, dass die Laufzeit des Detektors ausschlaggebend ist.

2.4.3 Herausforderungen bei Objekt-Tracking

Abhängig vom gewählten Lösungsansatz wird das Tracking durch verschiedene Faktoren beeinflusst, was zu einer Abweichung der bestimmten von der tatsächlichen Position bis zum Scheitern des Trackings führen kann. Wang et al. (2011) beschreiben folgende Probleme, welche sich auf verschiedene Tracking-Algorithmen unterschiedlich stark auswirken [41]:

- Veränderungen in der Beleuchtung des Objekts und der unmittelbaren Umgebung, wie z.B. harte Schatten
- Verformung des Objekts, z.B. durch sich bewegende Elemente
- Rotation des Objekts
- Verdeckung des verfolgten Objekts durch Objekte im Vordergrund
- Schnelle Bewegungen des Objekts, wodurch dieses den Suchbereich verlässt
- Bewegungen im Hintergrund des Objekts

2.4.4 Kalman-Filter

Bei der Erkennung von Objekten sowie der Bestimmung ihrer Bewegung ergibt sich also eine Ungenauigkeit bei der Messung in einem konkreten Zeitpunkt. Möglicherweise wird das Objekt kurzzeitig auch nicht erkannt. Somit muss ein Verfahren verwendet werden, mit dem der Zustand des Objekts über die Zeit optimal abgeschätzt werden kann. Dafür kann der Kalman-Filter verwendet werden [40, Kapitel 8.4.2].

Dieser rekursive Algorithmus besteht aus zwei Schritten, der Vorhersage (vgl. Gleichung 2.1) und Korrektur (vgl. Gleichung 2.2). Eine vollständige Herleitung des Kalman-Filters ist nicht Bestandteil dieser Thesis.

$$\begin{aligned} \hat{x}'_n &= \Theta_{n-1}\hat{x}_{n-1} \\ P'_n &= \Theta_{n-1}P_{n-1}\Theta^\top_{n-1} + Q_{n-1} \end{aligned} \tag{2.1}$$

$$\begin{aligned} K_n &= P'_n H^\top_n (H_n P'_n H^\top_n + R_n)^{-1} \\ \hat{x}_n &= \hat{x}'_n + K_n(z_n - H_n\Theta_{n-1}\hat{x}_{n-1}) \\ P_n &= (\mathbb{1} - K_n)P'_n(\mathbb{1} - K_n)^\top + K_n R_n K^\top_n \end{aligned} \tag{2.2}$$

Dabei ist der Vektor $\hat{x}$ die Observation des zeitabhängigen Zustands x, P die zugehörige Kovarianzmatrix und Θ die zeitabhängige Übergangsmatrix zum jeweils folgenden Zustand. Die Kalman-Matrix K steuert den Einfluss des Innovationsterms auf die Vorhersage des Zustands. z_n ist das Messergebnis im Zeitpunkt n und H die zugehörige Abbildung des Zustands auf die Messung. Q und R sind die Kovarianzmatrizen des Rauschens in der Zustandsveränderung und in der Messung. Bei der Anwendung des Kalman-Filters für das Tracking von Objektpositionen auf der Bildebene ist der Zustand ein vierdimensionaler Vektor aus Position auf und Geschwindigkeit entlang der Bildebene. Dabei ist zu beachten, dass die Komponenten an unterschiedlichen Achsen keinen Einfluss aufeinander haben. Somit enthält die Matrix dort nur die Varianzen.

2.4.5 Munkres-Algorithmus

Bei Tracking durch Detektion muss unterschieden werden, ob es sich bei einem erkannten Objekt um eine neue Instanz oder um ein bereits verfolgtes Objekt handelt. Im zweiten Fall muss zudem ermittelt werden, um welche Instanz es sich dabei handelt. Je ähnlicher sich die Objekte in den aufeinander folgenden Bildern der Sequenz sind, desto wahrscheinlicher ist es, dass es sich um das selbe Objekt handelt. Dieses Problem ist nicht trivial, da ein primitiver Algorithmus, welcher nacheinander jedes Objekt optimal zuordnet in vielen Fällen nicht die insgesamt beste Zuordnung produziert. Die Auswahl der besten Zuordnung durch Ausprobieren ist auch kein brauchbarer Ansatz, da er eine faktorielle Komplexität hat. Der Algorithmus von Munkres [28], auch ungarischer Algorithmus genannt, löst das Zuweisungsproblem effizienter.

Algorithmus 1 Algorithmus von James Munkres für das allgemeine Zuweisungsproblem

Vorbedingung: K ist eine Matrix mit n Zeile und n Spalten. K_{ij} sind die Kosten der Zuordnung $i \to j$.

Liefert: Eine optimale Zuordnung

$K \leftarrow K - \min K$

solange keine optimale Zuweisung gefunden wurde **wiederhole**

Abdecken der Zellen mit dem Wert 0 durch die minimale Anzahl an Linien durch streichen der jeweiligen Zeile oder Spalte

wenn die Anzahl der Linien n entspricht **dann**

Es wurde eine optimale Zuweisung gefunden

sonst

$x \leftarrow$ min der nicht durchgestrichenen Zellen von K

Aktualisierung der Matrix mit $K_{ij} \leftarrow \begin{cases} K_{ij} - x, & \text{wenn } K_{ij} \text{ nicht durchgestrichen ist} \\ K_{ij} + x, & \text{wenn } K_{ij} \text{ von zwei Linien bedeckt ist} \\ K_{ij}, & \text{sonst} \end{cases}$

Entfernen aller Linien

beende wenn

beende solange

Wahl einer optimalen Zuweisung aus Zellen mit dem Wert 0, wobei jede Zeile und Spalte nur einmal verwendet werden darf und die Zelle kein Schnittpunkt von zwei Linien ist.

Die Suche nach der minimalen Zuweisung ist beendet, sobald mindestens n Linien benötigt werden. Das Durchstreichen und Prüfen der Matrix wird für jede Zelle sequentiell ausgeführt und hat somit quadratische Zeitkomplexität. Dieser Schritt wird solange wiederholt, bis n Linien notwendig sind. Somit kann der Algorithmus mit kubische Zeitkomplexität umgesetzt werden. (s. Algorithmus 1)

2.5 Themenverwandte Arbeiten

Die Anreicherung von AR-Anwendungen mit Objekt-Detektion wurde bereits in den folgenden Veröffentlichungen behandelt. Dabei werden jedoch unterschiedliche Schwerpunkte gesetzt:

Liu et al. [22] beschreiben eine Architektur zur Erkennung von ungesehenen Objekten in einer mobilen AR-Anwendung, bei welcher die eigentliche Erkennung der Objekte in die Cloud ausgelagert wird, wodurch hierfür mehr Rechenleistung zur Verfügung steht. Die Erkennung von ROIs für optimale Komprimierung der übertragenen Bilder wird auf dem mobilen Gerät durchgeführt. Durch die parallele Verarbeitung durch ein CNN und Übertragung der Bilder wird die Verzögerung minimiert.

Rao et al. [31] kombinieren Daten verschiedener Sensoren mit dem Ergebnis eines CNN, um Geoinformationen in einer mobilen AR-Anwendung zu visualisieren. Die Inferenz durch das CNN wird dabei auf dem Smartphone durchgeführt. Die Erkennung beschränkt sich somit auf markante, stationäre Objekte wie große Gebäude.

Wedyan et al. [42] zeigen einen Algorithmus für 3D-Objekt-Detektion, welcher für mobile AR-Anwendungen genutzt werden könnte. Das Modell wird dabei sowohl anhand von realen als auch synthetischen Daten trainiert. Es wird die Erkennung mehrerer Objekte der selben Klasse demonstriert.

3 Anforderungen

In diesem Kapitel werden die Anforderungen an den Prototypen beschrieben. Allgemeine Anforderungen an ein Softwareprojekt beziehungsweise eine Smartphone-Anwendung werden dabei nicht explizit erwähnt, sondern nur solche, die für diese Anwendung speziell relevant sind.

3.1 Objekterkennung

Der Prototyp soll relevante Verkehrsteilnehmer erkennen können. Dazu gehören vor allem Fußgänger, Fahrradfahrer, Autos, Busse und Hunde. Eine Anwendungsfall-spezifische Klassifizierung ist nicht Teil dieser Thesis. Es sollen Objekte im Abstand von einem bis zu einhundert Metern erkannt werden. Dazu soll die Entfernung des erkannten Objekts auf bis zu einen Meter genau abgeschätzt werden, um so die Position des Objekts im Raum zu ermitteln. Diese Objekte sollen dabei so lange verfolgt werden, bis sie den Sichtbereich der Kamera verlassen. Ein Objekt, dass den Sichtbereich der Kamera verlässt und dann wieder in diesen zurückkehrt, wird als neues Objekt erkannt. Abgesehen von Erkennung und Verfolgen dieser Objekte ist keine Anwendungsfall-spezifische Interaktion über den Prototypen oder weitere Informationsgewinnung im Rahmen dieser Thesis vorgesehen.

3.2 Umgebung

Da der Prototyp im Rahmen von urbaner Mikromobilität erstellt wird, ist die Anwendung für den Einsatz in einer städtischen Umgebung vorgesehen. Es sollen aber auch andere Verkehrsteilnehmer erfasst werden. Der Prototyp wird bei Tag, das heißt bei guten Lichtverhältnissen, eingesetzt.

3.3 Performance

Um dem Benutzer den Eindruck einer flüssigen Bewegung zu vermitteln muss eine Bildwiederholrate von mindestens zehn Bildern pro Sekunde erreicht werden [32, Kapitel 4.7.1]. Dementsprechend darf die Aktualisierung der AR-Ansicht maximal einhundert Millisekunden benötigen. Eine höhere Bildrate ist erstrebenswert, um eine ergonomische Benutzung der Anwendung zu ermöglichen. Da es sich also um eine Echtzeit-Anwendung handelt, stellt die gleichzeitige Verwendung von AR- und CV-Softwarekomponenten somit auch eine zentrale Herausforderung dar, da beiden vergleichsweise rechenaufwändig sind. Die Visualisierung mit AR wird minimal gehalten, da diese nicht Gegenstand dieser Thesis ist. Allerdings muss dieser Aspekt im Bezug auf die Performance für einen möglichen Ausbau des Prototypen mit berücksichtigt werden. Der Prototyp soll als App für ein Mittelklasse-Smartphone umgesetzt werden. Alle Komponenten der Anwendung sollen auf diesem Gerät laufen.

4 Planung der Umsetzung

In diesem Kapitel wird das grundlegende Konzept für den Prototypen erläutert. Entscheidungsmöglichkeiten werden präsentiert und die Begründung für den jeweils gewählten Ansatz dargelegt.

4.1 Datenfluss

In Abbildung 4.1 ist dargestellt, wie sich die einzelnen Funktionen des Prototypen bezüglich des Datenflusses aufteilen lassen. Jede Funktion ist dabei unabhängig von den anderen, weshalb diese nebenläufig ausgeführt werden können. Bei der Übergabe von Daten müssen diese allerdings synchronisiert werden. Somit kann das Kamerabild der AR-Ansicht unabhängig von der Visualisierung der Objekte aktualisiert werden. Dadurch erscheint das Bild flüssiger und führt beim Benutzer nicht zu Unwohlsein aufgrund zu niedriger Bildrate. (vgl. Abschnitt 3.3) Da im Rahmen des Prototypen keine Interaktion mit Datenquellen oder -senken vorgesehen ist, kann die Architektur des Prototypen an die einer Pipeline angelehnt werden. Lediglich das Objekt-Tracking ist zustandsbehaftet, da bereits erkannte Objekte re-identifiziert werden müssen. (vgl. Abbildung 4.1)

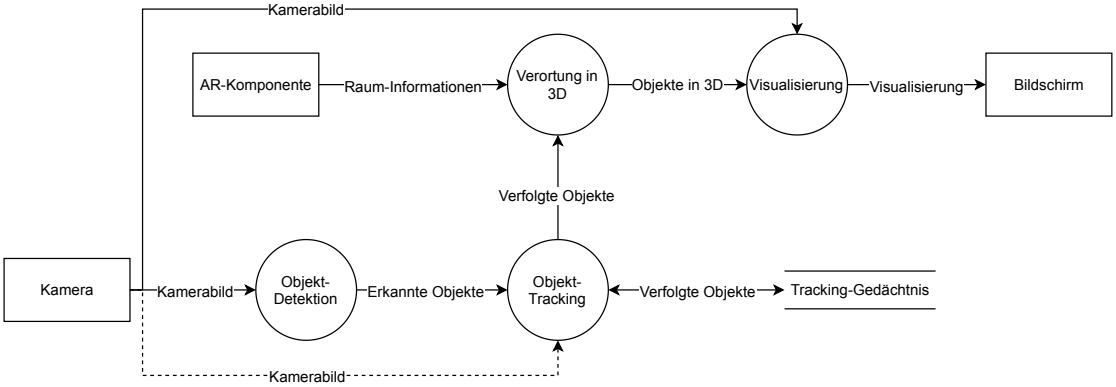


Abbildung 4.1: Datenflussdiagramm des Prototypen

4.2 Hardware

Als Testgerät für den Prototypen wurde ein Samsung Galaxy S7 edge[1] mit Android 8.0.0 ausgewählt. Dieses Modell verfügt weder über spezielle Sensoren noch Hardware-Beschleuniger neben der CPU und GPU. Zum Zeitpunkt der Verfassung dieser Thesis ist es also repräsentativ für den Software-Stand und die Hardware-Leistung eines Mittelklasse-Smartphones.[2]

4.3 Bibliothek für Computer Vision

Für die Implementierung von Objekt-Tracking sowie weiterer Bild-Operationen wird die Bibliothek OpenCV [7] verwendet. Die Software-Bibliothek beinhaltet Implementierungen für viele CV-Algorithmen und es existieren Schnittstellen für diverse Programmiersprachen und Platformen, weshalb es sich als De-facto-Standart-Bibliothek für bildverarbeitende Anwendungen etabliert hat.

4.4 Objekt-Detektion

Für die Erkennung von Objekten wird, wie in Unterabschnitt 2.3.2 beschrieben, ein CNN verwendet. Da die zu erkennenden Objekte zu sehr allgemeinen Klassen gehören, kann auf bereits trainierte Modelle zurückgegriffen werden. Die Entwicklung eines neuen Algorithmus für Objekt-Detektion und die Zusammenstellung eines eigenen Datensatzes ist nicht Bestandteil dieser Thesis.

[1] Modelnummer: SM-G935F `https://www.gsmarena.com/samsung_galaxy_s7_edge-7945.php` (Abgerufen am 20.06.2020)
[2] Nachvollziehbar anhand Abbildung A.2

Modell	Framework	Laufzeit (ms)	mAP	Datentyp	Auflösung
Tiny YOLO v2	OpenCV DNN	355,79	23,7	float32	416x416
Tiny YOLO v3	OpenCV DNN	397,66	33,1	float32	416x416
SSD MobileNet v1	TensorFlow Lite	103,75	18	int8	300x300
SSD MobileNet v3 (small)	TensorFlow Lite	54,92	15,4	int8	320x320

Tabelle 4.1: Auswertung verschiedener Modelle für Objekt-Detektion

Für den Einsatz im Prototypen wurden vier unterschiedliche Modelle auf dem Testgerät evaluiert. (s. Tabelle 4.1 und Abschnitt A.2) Die Modelle wurden auf dem COCO-Datensatz [21] trainiert. Dieses enthält über zweihunderttausend annotierte Bilder mit Objekten aus 80 unterschiedlichen Klassen und wird daher oft für das Training und den Vergleich von Algorithmen für Objekt-Detektion verwendet. Da dieser Datensatz jedoch auch Klassen beinhaltet, welche für den Prototypen nicht relevante sind, werden die Ergebnisse der Erkennung nach relevanten Klassen gefiltert. Das Framework ist nicht Bestandteil des Modells, wird aber verwendet um dieses auszuführen. Das Format in welchem das Modell serialisiert wurde, gibt dabei vor, welches Framework verwendet werden kann. Dementsprechend wird kein Framework-spezifisches Messwerkzeug wie das „TFLite Model Benchmark Tool“[3] verwendet, sondern eine minimale Android-Anwendung entwickelt, welche das jeweilige Modell auf das aktuelle Kamerabild anwendet, um unterschiedliche Modelle vergleichen zu können. Der Grafikprozessor kann nicht zur Beschleunigung des neuronalen Netzwerks verwendet werden, da hierfür mindestens Android 8.1 erforderlich ist[4], das vorab definierte Testgerät unterstützt jedoch nur die Android-Version 8.0.0. Dabei muss außerdem erwähnt werden, dass die Zuverlässigkeit der Objekterkennung mit den „SSD MobileNet“-Modellen trotz geringerer Bild-Auflösung und Quantisierung merklich besser war als mit den „Tiny YOLO“-Modellen. Die angegebene[5] mittlere durchschnittliche Präzision (mAP in Tabelle 4.1) ist jedoch geringer. Obwohl die vierte Version des YOLO-Algorithmus eine deutlich höhere Präzision verspricht, hat sie bei der Erkennung eine vergleichbare Laufzeit zu dem Vorgänger [4] und wird daher nicht für den Prototypen in Erwägung gezogen. Mit Blick auf Abschnitt 3.3 wird das schnellste Modell, also „SSD MobileNet v3 (small)“ verwendet.

[3] `https://github.com/tensorflow/tensorflow/tree/master/tensorflow/lite/tools/benchmark` (Abgerufen am 15.07.2020)

[4] `https://developer.android.com/ndk/guides/neuralnetworks` (Abgerufen am 15.07.2020)

[5] `https://pjreddie.com/darknet/yolo/` (Abgerufen am 15.07.2020) und `https://github.com/tensorflow/models/blob/master/research/object_detection/g3doc/tf1_detection_zoo.md` (Abgerufen am 15.07.2020)

4.5 Objekt-Tracking

Bestenfalls soll der Prototyp alle erkannten Objekte verfolgen. Im Rahmen des Prototypen werden VOT (s. Unterabschnitt 2.4.1) und Tracking durch Detektion (s. Unterabschnitt 2.4.2) unabhängig voneinander verwendet, um erkannte Objekte zu verfolgen, da diese unterschiedliche Vor- und Nachteile haben.

4.5.1 Tracking durch Detektion

Bei Tracking durch Detektion müssen in einer Sequenz von Bildern die Ergebnisse des Detektors zu denen aus dem jeweils vorangegangenen Bild zugeordnet werden.

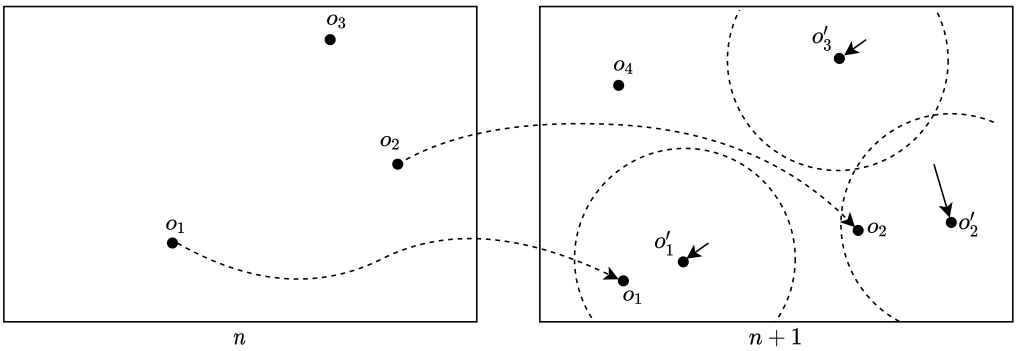


Abbildung 4.2: Illustration des Trackings durch Detektion über zwei Bilder

In Abbildung 4.2 wird dies im Bezug auf den Prototypen dargestellt. Im vorausgegangen Bild wurden dabei drei Objekte erkannt. Zunächst wird für diese die jeweilige Position o'_i für das Objekt o_i im darauf folgenden Bild vorhergesagt. Dafür wird der in Unterabschnitt 2.4.4 beschriebene Kalman-Filter verwendet. Der Zustand eines Objekts besteht dabei aus dessen Position im Bild und Geschwindigkeit entlang beider Achsen. Die Objekt-Nummern der neuen Observationen werden mithilfe des Algorithmus für die optimale Zuweisung aus Unterabschnitt 2.4.5 und den Vorhersagen bestimmt (vgl. o_1 und o_2 in Abbildung 4.2).

Da es sich bei dem Zuweisungsproblem um eine allgemeine Optimierungsaufgabe handelt, kann auf eine quelloffene Implementierung zurückgegriffen werden.[6] Der Abstand, den ein Objekt zwischen zwei Bildern zurücklegen kann, ist begrenzt. Somit handelt es sich bei Objekten, die nicht zugewiesen werden, entweder um neue Objekte (vgl. o_4 in Abbildung 4.2) oder um solche, die nicht mehr verfolgt werden (vgl. o_3' in Abbildung 4.2). Damit das Tracking nicht durch kurze Ausfälle der Detektion eines Objekts scheitert, wird es erst beendet, wenn mehrere Bilder ohne erfolgreiche Erkennung des Objekts aufeinander folgenden.

4.5.2 OpenCV Tracking-API

Da Objekt-Detektion, wie in Abschnitt 4.4 gezeigt, verhältnismäßig viel Zeit in Anspruch nimmt, kann dies bei einer Bildrate von 30 Hertz nicht für alle Bilder der Sequenz durchgeführt werden. Für alle anderen Bilder wird VOT mit OpenCV verwendet, um die Position der erkannten Objekte in diesen zu bestimmen. In der OpenCV-Bibliothek sind mehrere Tracking-Algorithmen implementiert. Die gemeinsame Schnittstelle der Tracker ist dabei aber nur für die Verfolgung eines einzelnen Objekts ausgelegt. Somit muss für jedes verfolgte Objekt eine neue Instanz eines Trackers erstellt werden. Bei der Auswahl eines Tracking-Algorithmus für den Prototypen ist dessen Laufzeit das Hauptkriterium. Lehtola et al. [20] zeigen, dass Median Flow [18] der schnellste aber auch ungenauste Algorithmus für den Einsatz auf Low-End-Geräten ist. Sie empfehlen den KCF-Algorithmus, allerdings ist der MOSSE-Algorithmus [5], welcher sich auch der Korrelation bedient sogar schneller als Median Flow, allerdings ist die Initialisierung vergleichsweise zeitaufwendig.

Algorithmus	Initialisierung (ms)	Aktualisierung (ms)
MOSSE	54,09	8,76
TLD	26,81	198,29
Median Flow	1,11	14,66
KCF	6,56	34,17

Tabelle 4.2: Auswertung einiger OpenCV-Implementierungen von Tracking-Algorithmen auf dem Testgerät

[6] `https://github.com/KevinStern/software-and-algorithms/blob/master/src/main/java/blogspot/software_and_algorithms/stern_library/optimization/HungarianAlgorithm.java` (Abgerufen am 15.07.2020)

Mit Hinblick auf die experimentell ermittelte Laufzeit dieser Algorithmen (s. Tabelle 4.2) auf dem Testgerät ist der Einsatz dieser Implementierungen für die Verfolgung mehrerer Objekte nicht sinnvoll. Somit muss der Benutzer ein erkanntes Objekt auswählen, welches anschließend durch den Tracker verfolgt wird. Von den untersuchten Algorithmen eignen sich nur MOSSE und Median Flow. MOSSE ist bei diesen Experimenten robuster gegenüber Verlust des verfolgten Objekts und das Tracking wird in diesem Fall zuverlässig abgebrochen. Bei Median Flow kommt es hingegen bei unruhiger Bewegung des verfolgten Objekts teilweise dazu, dass ein ehemaliger Aufenthaltsort des Objekts weiter verfolgt wird.

Allerdings dauert die Initialisierung des Trackers bei MOSSE deutlich länger als bei Median Flow. Da diese vergleichsweise häufig durchgeführt wird, ist die Laufzeit von Median Flow insgesamt kürzer, weshalb dieser Algorithmus für den Prototypen verwendet wird.

4.5.3 Weitere Ansätze

Des weiteren wurde eine eigene Implementierung auf Basis von optischem Fluss in Erwägung gezogen. Optischer Fluss bezeichnet die wahrgenommene Bewegung von Punkten zwischen zwei aufeinander folgenden Bildern. Bei dichtem optischen Fluss wird dabei die Bewegung für jeden Pixel bestimmt. Diese Operation ist sehr rechenaufwändig. Die OpenCV-Implementierung des Algorithmus von Farnebäck [10] zur Bestimmung des dichten Optischen Flusses läuft zwar auf dem Testgerät in etwas kürzerer Zeit als die Objekt-Detektion, allerdings hat das Ergebnis starke Artefakte und Fehler wie in Abbildung 4.3 zu sehen ist. Teilweise wird für große Bildbereiche eine abweichende Bewegungsrichtung ermittelt und die Umgebung bewegter Objekte wird durch diese fälschlicherweise beeinflusst. Deshalb würde dieses Verfahren keine zuverlässige Information über die Bewegung von Objekten zwischen zwei Bildern in der Sequenz liefern.

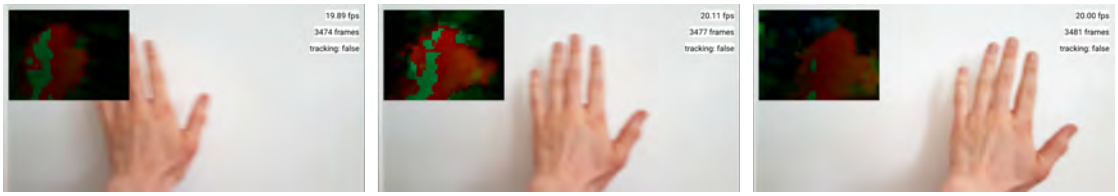


Abbildung 4.3: Bildsequenz mit dichtem Optischem Fluss dargestellt durch Verfärbung der bewegten Pixel

Mit der Lucas-Kanade-Methode [6] kann spärlicher Optischer Fluss, also nur für einzelne Bildmerkmale anstelle von jedem Pixel, bestimmt werden. Die Intuition hinter diesem Algorithmus ist, dass die unmittelbare Umgebung eines Pixels den selben optischen Fluss erfährt, womit dieser als Gleichungssystem mit nur zwei unbekannten Variablen ausgedrückt werden kann. Dieses Verfahren ist schneller als solche zur Bestimmung des dichten optischen Flusses. Da sich Eckpunkte am besten mit diesem Verfahren verfolgen lassen, wird der Algorithmus von Shi und Tomasi [38] zur Bestimmung der n besten Punkte verwendet. Dabei wird die Bewegung der Punkte auch bei mehrfachem Glätten und Downsampling (Bildpyramide mit absteigender Auflösung) bestimmt, um so auch größere Bewegungen zu erkennen. Wie Abbildung 4.4 allerdings zeigt, sind die durch diesen Algorithmus ausgewählten Merkmale nicht immer gleichmäßig im Bild verteilt. Um ein beliebiges Objekt verfolgen zu können, müsste allerdings mindestens ein Merkmal in dem selben Bereich liegen. Eine ausgewogenere Verteilung kann zwar durch mehrfache Anwendung auf maskierte Teilbereiche des Bildes erreicht werden, dies erhöht die Laufzeit der Merkmal-Suche allerdings zu stark, weshalb dieser Ansatz nicht weiter verfolgt wird.

Abbildung 4.4: Extrahierte Merkmale für spärlichen optischen Fluss

4.6 Verortung in 3D

Wie bereits in Abschnitt 3.1 erwähnt, sollen die erkannten Objekte im Raum verortet werden. Da die Objekterkennung nur die Position der Objekte auf der Bildebene bestimmt, muss die Position im Raum durch die Bestimmung des Abstands zur Kamera in einem weiteren Schritt ermittelt werden. Dieser wird in Abbildung 4.1 mit „Verortung in 3D“ dargestellt. Wie in Unterabschnitt 2.3.3 beschrieben, existieren zwar bereits Algorithmen zur Bestimmung des dreidimensionalen Objektrahmens („Bounding Volume“) auf Basis eines einzelnen Kamerabildes, allerdings sind diese nicht für den Einsatz im Prototypen geeignet. Somit muss eine andere Strategie genutzt werden, um die Position der Objekte im Raum zu bestimmen.

Im Folgenden werden drei Ansätze zur Bestimmung der Entfernung von Objekten im Rahmen des Prototypen erläutert. Dabei wird nur der Abstand und nicht die Ausrichtung oder das Volumen der Objekte bestimmt.

4.6.1 Entfernungsbestimmung von Objekten mit bekannter Größe

Je weiter ein Objekt von einem Beobachter entfernt ist, desto kleiner erscheint es. Die Größe des Objekts im Bild s' ist somit umgekehrt proportional zum Abstand zur Kamera d und direkt proportional zur tatsächlichen Größe s. Wenn diese bekannt ist, kann d mit der Gleichung 4.1 unter der Voraussetzung, dass Verzerrungen durch die Kameralinse ignoriert werden können, bestimmt werden [14, Beschreibung 0021]. Unter der Voraussetzung, dass ein Objektiv mit festem Fokus verwendet wird, ist f eine Konstante. Sie kann experimentell ermittelt werden.

$$d = f * \frac{s}{s'} \tag{4.1}$$

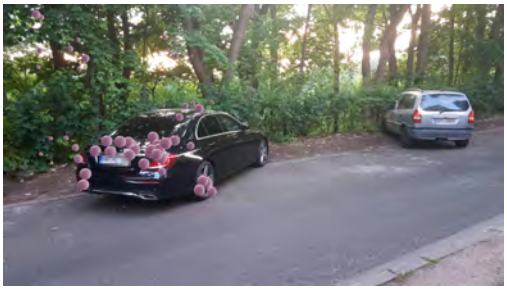
Wenn alle Objekte einer Klasse ungefähr gleich groß sind, kann so deren Entfernung grob abgeschätzt werden.
Allerdings ist aufgrund der immer noch sehr hohen Varianz der Größe innerhalb der Klassen die Genauigkeit dieser Methode hier nicht ausreichend. Alleine für Passanten muss bei einer Durchschnittsgröße von 170 Zentimetern von einer möglichen Abweichung um mindestens zehn Prozent ausgegangen werden, da nicht zwischen Kindern und Erwachsenen sowie unterschiedlichen Körperhaltungen unterschieden wird. Durch die Gruppierung von Kleintransportern bis Sportwagen als Autos besteht hier das gleiche Problem. Für partiell sichtbare Objekte wird außerdem gegebenenfalls eine stark abweichende Entfernung bestimmt, sobald ein kleinerer Objekt-Rahmen erkannt wird.

4.6.2 Entfernungsbestimmung mit spärlicher Punktewolke

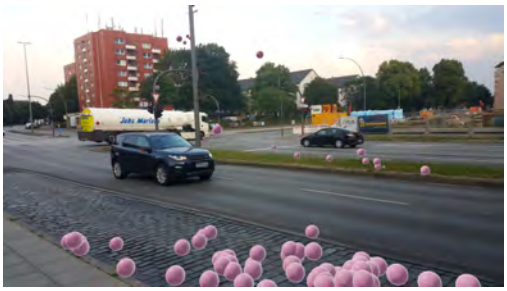
Das ARCore-Framework stellt eine durch SLAM ermittelte, spärliche Punktewolke über die API bereit. Diese besteht aus Punkten im dreidimensionalen Raum, welche mit Merkmalen in der Umwelt übereinstimmen. Durch Projektion auf die Bildebene kann ermittelt werden, welche dieser Punkte in den Rahmen eines erkannten Objekts fallen. Über die Auswertung der Abstände zu diesen Punkten kann etwa durch Bilden des Mittelwerts der Abstand zu dem jeweiligen Objekt abgeschätzt werden.

Die Praxis zeigt jedoch eine Häufung der Punkte auf statischen Objekten in geringem Abstand zur Kamera. Somit fallen diese nur gelegentlich in denselben Bereich wie die zu erkennenden Objekte mit einem Abstand von mehr als zwei Metern. Zudem gibt es keine Möglichkeit über die API auf die Auswahl der verfolgten Punkte Einfluss zu nehmen.

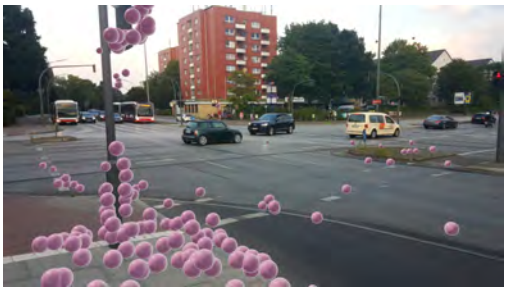
(a) Positivbeispiel auf kurze Distanz



(b) Negativbeispiel auf kurze Distanz



(c) Negativbeispiel auf mittlere Distanz



(d) Negativbeispiel auf größere Distanz

Abbildung 4.5: Visualisierung der von ARCore bereitgestellten Punktewolke für Entfernungsbestimmung

Dieses Problem ist in Abbildung 4.5 dargestellt. Die Punkte sind hier durch Kugeln mit einem Radius von zehn Zentimetern visualisiert. Dabei wird der Zuversichtlichkeitswert der erkannten Punkte ignoriert. Während in Abbildung 4.5a ausreichend Punkte auf beiden Fahrzeugen platziert sind um die Entfernung zu beiden zu bestimmen, befinden sich nach einem Kameraschwenk zu Abbildung 4.5b keine Punkte auf dem zweiten Fahrzeug. Abbildung 4.5c und Abbildung 4.5d zeigen die Häufung der Punkte in Nähe der Kamera. Außerdem befinden sich keine Punkte auf den sich bewegenden Fahrzeugen in den Bildern. Somit ist diese Methode ungeeignet zur Bestimmung der Entfernung von erkannten Objekten.

4.6.3 Entfernungsbestimmung durch Strahl-Ebenen-Schnitt

Unter der Annahme, dass sich der Boden der Umgebung durch eine Ebene im dreidimensionalen Raum darstellen lässt, kann die Bestimmung der Distanz zu Objekten durch den Schnitt dieser Ebene und einem Strahl approximiert werden. Dabei wird angenommen, dass die niedrigste durch ARCore erkannte Ebene den Boden des physikalischen Raums abbildet. Bei der Anwendung der Inversen der Bild-Projektion auf den Rahmen des erkannten Objekts liegt dessen untere Kante auf dieser Ebene. Dafür muss sich das Objekt auch in der Realität auf diesem Boden befinden.

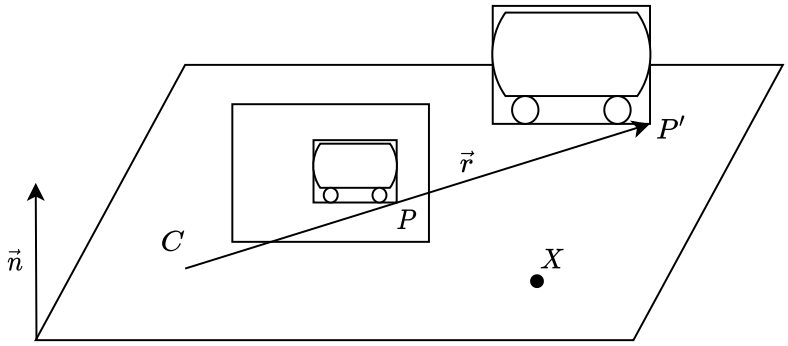


Abbildung 4.6: Bestimmung der Entfernung eines Bildpunktes P auf dem Boden zum Beobachter C durch Schnitt der Boden-Ebene mit dem Strahl $\overrightarrow{CP}$ im Punkt P'

Für einen Punkt P auf der Bildebene kann der zugehörige Punkt auf dem Boden im Raum bestimmt werden. Dazu sei C die Position der Kamera und $\vec{r}$ ein Strahl mit dem Segment $\overrightarrow{CP}$. Des weiteren sei X ein beliebiger Punkt auf der Ebene, welche den Boden darstellt und $\vec{n}$ die Normale dieser Ebene. (vgl. Abbildung 4.6)

$$\begin{aligned} \vec{d} &= \frac{P - C}{|P - C|} \\ l &= \frac{(X - C) \cdot \vec{n}}{\vec{d} \cdot \vec{n}} \\ P' &= C + l * \vec{d} \end{aligned} \tag{4.2}$$

Von der Position der Kamera aus lässt sich durch die Richtung $\vec{d}$ des Strahls mit der Entfernung l die Position des Schnittpunktes P' mit Gleichung 4.2 bestimmen [36, Kapitel 7.3.1].

Durch die Eckpunkte des Objektrahmens auf der Bildebene kann das Seitenverhältnis des Objekts berechnet werden. Somit wird die Position eines erkannten Objekts bestimmt unter Verwendung von P und $\vec{n}$, X, C, welche von der AR-Komponente des Prototypen bereitgestellt werden.
Die Tiefe des Objekts ist unbekannt, weshalb nur die Entfernung der zur Kamera zugewandten Seite des Objekts grob abgeschätzt werden kann, nicht aber der Mittelpunkt des Körpers in 3D. Außerdem wird, wie bereits erwähnt, vereinfacht davon ausgegangen, dass sich der Boden der Umgebung durch eine einzige Ebene approximieren lässt. Somit führen nicht lineare Steigungen, Gräben oder Brücken zur Berechnung falscher Werte. Eine Berechnung der Entfernung durch das Schneiden mit unterschiedlichen Ebenen kann dieses Problem jedoch potentiell lösen.

Im Gegensatz zu den beschriebenen alternativen Lösungsansätzen ist dieses Verfahren präzise genug und Laufzeit-unkritisch. Die Erkennung von Ebenen durch ARCore weist bei Experimenten eine Genauigkeit von über 94% [29] auf und ist damit hinreichend zuverlässig. Daher wurde sich im Rahmen des Prototypen für diese Methode zur Distanzbestimmung entschieden.

4.7 Benutzeroberfläche

Die Benutzeroberfläche des Prototypen wird minimal gehalten. Die AR-Ansicht spannt über den gesamten Bildschirm. Über dem Mittelpunkt der erkannten Objekte wird deren Name und der zuvor ermittelte Abstand zum Objekt eingeblendet als beispielhafte Visualisierung der erkannten Objekte angezeigt. Die Textboxen werden als dreidimensionale Schilder an der Position der zugehörigen Objekte und mit dem selben Abstand platziert. Um die Genauigkeit der Erkennung besser beurteilen zu können, wird zusätzlich eine Ansicht für Objekt-Detektion beziehungsweise -Trackings am linken Bildschirmrand angezeigt. Darin werden die Ergebnisse (Objektrahmen) direkt visualisiert. (vgl. Abbildung 4.7)

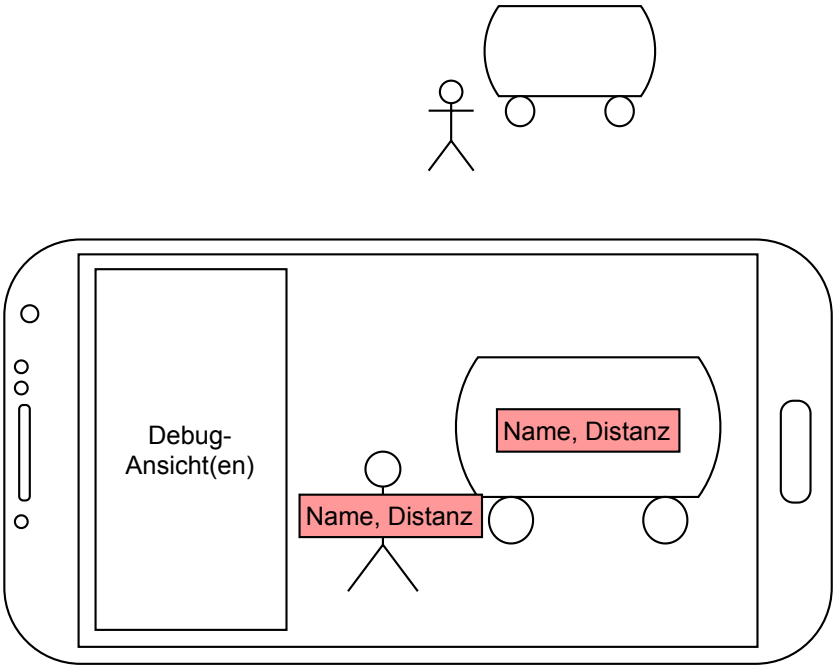


Abbildung 4.7: Konzept der Benutzeroberfläche des Prototypen

4.8 Entwicklungsumgebung

Der Prototyp wird als Android-Anwendung umgesetzt. Dementsprechend wird das Android SDK mit dem Build-System Gradle[7] verwendet. Diese Werkzeuge sind in der IDE AndroidStudio[8] enthalten und stellen die offizielle Entwicklungsumgebung für Android-Anwendungen dar. Für die Umsetzung der Visualisierung und Benutzeroberfläche stehen diverse Werkzeuge und Bibliotheken zur Verfügung. Da die Visualisierung jedoch für die Entwicklung des Prototypen nicht im Vordergrund steht, wird sich bei der Auswahl auf solche Optionen beschränkt, welche durch ausführliche Dokumentation und hohe Abstraktion eine möglichst schnelle Entwicklung ermöglichen.
Somit kann entweder die Unity Game Engine[9] oder das Sceneform SDK[10] verwendet werden. Erstere bietet einen Editor zur einfachen Komposition visueller Inhalte und bietet mit AR Foundation[11] eine Programmierschnittstelle für die Entwicklung von AR-Anwendungen. Dazu muss Code, der auf Android-Ressourcen zugreift, als Plug-in extern kompiliert und eingebunden werden. Da jedoch die Benutzeroberfläche im Rahmen des Prototypen nur einen kleinen Teil der Anwendung ausmacht, ist es sinnvoller hierfür die Sceneform-Bibliothek zu verwenden.

[7] `https://gradle.org/` (Abgerufen am 25.06.2020)
[8] `https://developer.android.com/studio` (Abgerufen am 25.06.2020)
[9] Middleware zur Entwicklung multimedialer Anwendungssoftware und von Videospielen `https://unity.com/products/core-platform` (Abgerufen am 25.06.2020)
[10] `https://developers.google.com/sceneform` (Abgerufen am 25.06.2020)
[11] `https://docs.unity3d.com/Packages/com.unity.xr.arfoundation@2.2/manual/index.html` (Abgerufen am 25.06.2020)

5 Implementierung

Im Folgenden wird die Umsetzung des Prototypen als Android-Anwendung beschrieben. Zudem wird auf die verwendeten Bibliotheken eingegangen, sowie Details, die bei der Implementierung berücksichtigt werden müssen.

5.1 AndroidStudio-Projekt

Der Aufbau des Projekts (vgl. Abbildung 5.1) ist teilweise durch AndroidStudio vorgegeben. „LongDistanceAR“ ist der Name des App-Moduls. Die Modelle für neuronale Netzwerke werden als Assets (LongDistanceAR/src/main/assets) in die App eingebunden und zur Laufzeit in das Dateisystem kopiert. Dies ist notwendig, da das verwendete Framework nicht direkt auf die Assets der App zugreifen kann.

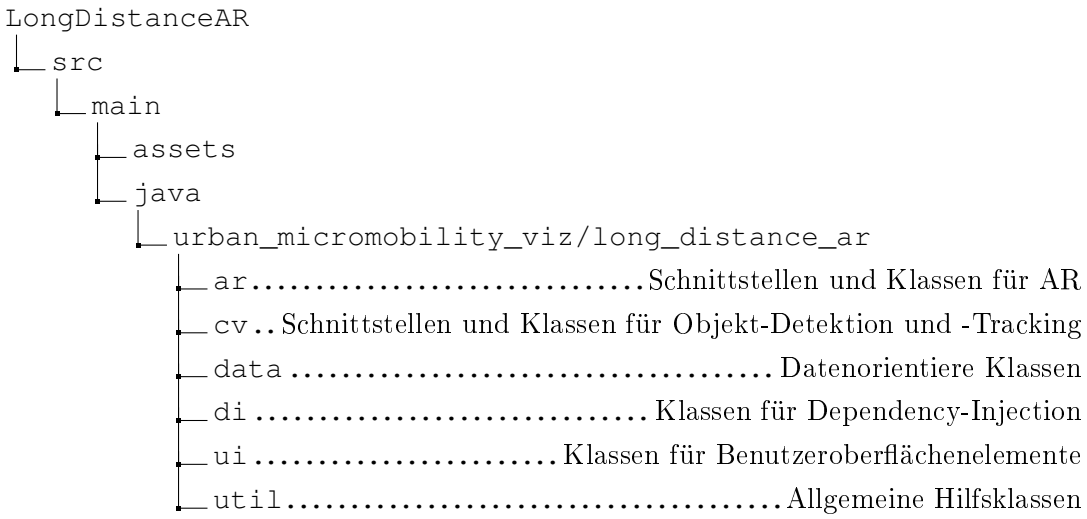


Abbildung 5.1: Vereinfachte Struktur des AndroidStudio-Projekts

Innerhalb des Pakets werden die Klassen in einzelne Unterpakete aufgeteilt. Soweit möglich, werden Klassen der AR- und CV-Bibliotheken nur aus den zugehörigen Unterpaketen referenziert, um die eventuelle Verwendung alternativer Bibliotheken zu einem späteren Zeitpunkt zu erleichtern.

5.2 Kotlin

Der Prototyp wird mit der Programmiersprache Kotlin[1] implementiert. Die von Jetbrains entwickelte Sprache kann zu Bytecode für die JVM kompiliert werden und hat Java als die Standard-Programmiersprache für Android-Anwendungen abgelöst. Dadurch ist zudem unkomplizierte Interoperabilität zwischen den beiden Programmiersprachen möglich, das heißt, Java-Klassen können direkt von Kotlin-Klassen verwendet werden und umgekehrt.

5.3 Externe Module

Verwendete Bibliotheken wurden über die „build.gradle“-Datei des Moduls eingebunden. Diese Abhängigkeiten werden als externe Module in Form von Artefakten über die Google[2]- und JCenter[3]-Repositories bezogen. Die verwendeten Versionen sind zum Zeitpunkt der Implementierung die jeweils aktuellsten und werden explizit spezifiziert, um Inkompatibilität durch Aktualisierungen zu vermeiden.[4]

[1] `https://kotlinlang.org/` (Abgerufen am 05.08.2020)
[2] `https://maven.google.com/` (Abgerufen am 30.06.2020)
[3] `https://bintray.com/bintray/jcenter` (Abgerufen am 30.06.2020)
[4] Abschnitt A.3 enthält die Liste der verwendeten Bibliotheken und Versionen

5.4 Reaktive Programmierung

Der Prototyp weist, wie in Abschnitt 4.1 beschrieben, eine Pipeline-ähnliche Struktur auf. Daten werden sequentiell und asynchron über mehrere Stufen verarbeitet, weshalb sich reaktive Programmierung für die Umsetzung des Prototypen anbietet. Dabei handelt es sich um ein Event-orientiertes Programmierparadigma [2]. Eine weit verbreitete Bibliothek für reaktive Programmierung ist ReactiveX[5]. Sie bietet für diverse Sprachen ein einheitliches Konzept und eine funktionale Programmierschnittstelle, wodurch die Anwendungslogik mit verhältnismäßig wenig Aufwand auf andere Plattformen portiert werden kann. Für den Prototypen wird RxJava[6] eingesetzt, da dieses aktuell weiter verbreitet und besser dokumentiert ist als die Kotlin-Version.

5.4.1 ReactiveX-Klassen

Im Rahmen des Prototyps werden folgende Klassen verwendet:

`Observable` Basisklasse einer Event-Quelle, auf welche Operationen wie `map` angewendet werden können. Sie implementiert die Schnittstelle `ObservableSource`.

`Observer` Schnittstelle für die Behandlung von Events.

`Flowable` Eine Implementierung der `Publisher`-Schnittstelle, welche im Gegensatz zum `Observable` verschiedene Strategien für die Behandlung von Gegendruck unterstützt.

`Disposable` Schnittstelle für Ressourcen innerhalb ReactiveX, welche geschlossen werden müssen. Die Beobachtung einer Event-Produzenten durch `Observable#subscribe` erstellt einen `Disposable`. Dieses wird an den Lebenszyklus der Android-Anwendung geknüpft und somit automatisch geschlossen.

`Subject` Eine Klasse, welche sowohl `Observable` und `Observer` vereinigt. Sie wird im Prototyp verwendet um Callbacks und Listener zu behandeln.

[5] `http://reactivex.io/` (Abgerufen am 30.06.2020)
[6] `http://reactivex.io/RxJava/javadoc/` (Abgerufen am 05.07.2020)

5.4.2 Scheduling

In RxJava kann über einen `Scheduler` eine Operation auf einem anderen Thread oder mit einer anderen Strategie ausgeführt werden. Der `Scheduler` für das `Observable` sowie dessen Beobachter können dabei unabhängig gewählt werden.[7] Die Verarbeitungsstufen des Prototypen laufen in Arbeiter-Threads des RxJava-Pools. In Android darf die Benutzeroberfläche nur durch den Haupt-Thread der Anwendung verändert werden. Dementsprechend wird vor der terminalen Verarbeitungsstufe auf diesen gewechselt.

5.4.3 Gegendruck-Strategie

Wenn ein Konsument länger für die Verarbeitung eines Events benötigt als der Produzent, kommt es zu Gegendruck. Standardmäßig werden die Elemente in einem Puffer zwischengespeichert. Läuft dieser voll kommt es zu einem Fehler. Unter der Annahme, dass der Prototyp dreißig Bilder pro Sekunde aufnimmt, stellt der Objekt-Detektor mit den in Tabelle 4.1 gemessenen Zeiten einen Flaschenhals dar. Pufferung der Bilder ist hier nicht möglich, da dieser kontinuierlich wachsen würde. Da es sich bei den zu puffernden Objekten um unkomprimierte Bilddaten handelt, führt selbst eine geringe Anzahl von Elementen im Puffer zu Speicherknappheit. Des weiteren wird die starke Bildverzögerung durch den Benutzer der Anwendung wahrgenommen und sorgt für Verwirrung oder Unwohlsein.
Daher wird nur ein Element bis zur Weiterverarbeitung gepuffert und gegebenenfalls durch neuere Objekte überschrieben. Somit werden zwar möglicherweise einzelne Bilder in der Sequenz übersprungen, aber es wird immer das aktuellste Bild verarbeitet.

5.4.4 Zusammenführen von Event-Quellen

Zwei Event-Quellen können mit den Operationen `Observable#merge` beziehungsweise `Flowable#zip` elementweise zusammengeführt werden. Dies wird zum Beispiel verwendet, um den Zustand des Trackers zu aktualisieren. Dazu wird der letzte Zustand des Trackers mit dem nächsten Ergebnis der Objekt-Detektion bzw. dem nächste Kamerabild zu einer neuen Event-Quelle zusammengeführt.

[7] `http://reactivex.io/documentation/scheduler.html` (Abgerufen am 30.06.2020)

5.5 Model-View-Presenter

Model-View-Presenter (MVP) ist ein Architekturmuster zur Trennung von Businesslogik, Daten und Benutzeroberfläche. Es ähnelt stark dem Architekturmuster Model-View-Controller (MVC) und ist aus diesem hervorgegangen [39]. Es kann unter anderem auch auf Android-Anwendungen angewendet werden[8] und wird für die Entwicklung des Prototypen gewählt, denn es ist flexibler als alternative Architekturmuster und bei Projekten mit kleinem Umfang sehr übersichtlich. Da es von Seiten des Benutzers nur wenige Eingabemöglichkeiten gibt, ist die View größtenteils passiv. In Android stellt die Activity das zentrale Lebenszyklus-Objekt einer Benutzeroberfläche da und wird vom System erstellt. Dementsprechend implementiert es die View. Da die Implementierungen nur Schnittstellen (Interfaces) referenzieren, sind sie nicht aneinander gekoppelt und können mit wenig Aufwand durch Stubs ersetzt werden. Ein Teil des Models ist die `FrameDataSource`-Klasse, welche die Kamerabilder bereitstellt. Da hierfür auf Systemressourcen zugegriffen werden muss, wofür eine Referenz auf den Android-Context notwendig ist, wird dieses Objekt direkt durch die View, also die Activity für den Presenter bereitgestellt. Die View benutzt dieses Objekt allerdings nicht direkt. (vgl. Abbildung 5.2)

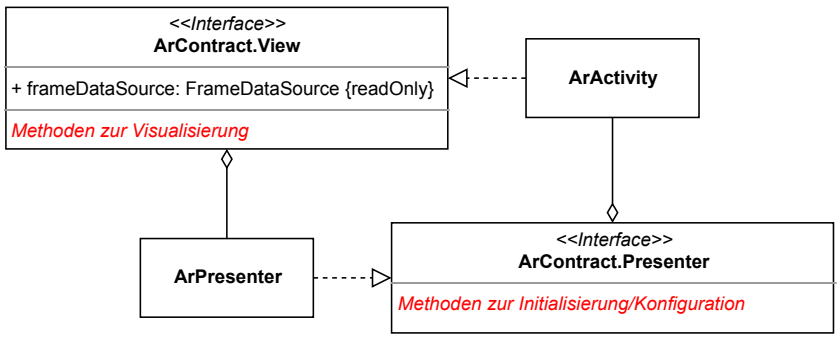
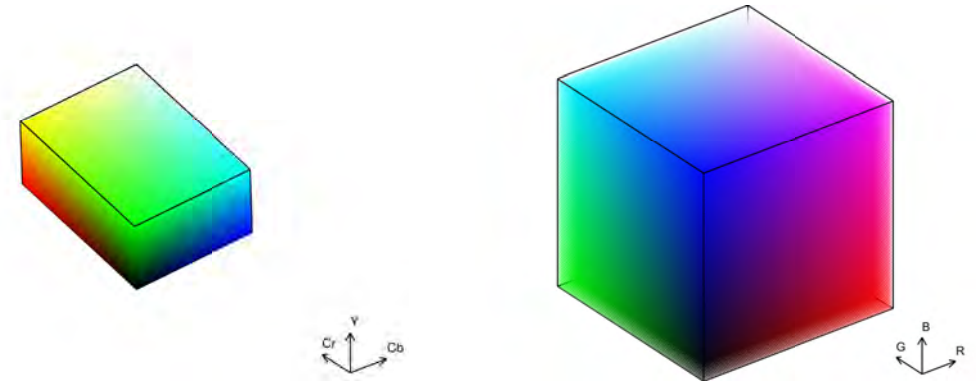


Abbildung 5.2: Vereinfachtes UML-Klassendiagramm zur MVP-Architektur des Prototypen

[8] `https://medium.com/@jintin/mvp-architecture-in-android-1f98a74e7e1b` (Abgerufen 30.06.2020)

5.6 Farbmodelle

Da die Bilder in verschiedenen Formaten benötigt werden und die zugreifenden Verarbeitungsstufen nebenläufig sind, wird bei der Umwandlung zwischen den Farbmodellen und -formaten Speicherplatz für ein neues Bild alloziert. Die Konvertierung dazwischen ist in OpenCV implementiert.



(a) YCbCr-Farbmodell (b) RGB-Farbmodell

Abbildung 5.3: Farbmodelle (100^3 Samples)

5.6.1 YCbCr

Die von der Kamera aufgezeichneten Bilddaten werden im Format YUV_420_888 bereitgestellt. Dabei handelt es sich um ein YCbCr-Format (vgl. Abbildung 5.3a). Der erste Kanal enthält die Luminanz, also die Lichtstärke. Die beiden weiteren Kanäle bestimmen die Chrominanz, also den Farbton ausgedrückt durch den Blau-Unterschied und Rot-Unterschied [15]. Dabei wird jedes Sample aus den Chrominanz-Kanälen immer für vier Pixel verwendet. Ein Sample in jedem Kanal belegt in diesem Format jeweils ein Byte[9]. Somit werden für jeden Pixel zwölf Bit Speicher benötigt.

[9] `https://developer.android.com/reference/android/graphics/ImageFormat#YUV_420_888` (Abgerufen am 30.06.2020)

5.6.2 RGB

Das Modell des Objekt-Detektors verwendet das RGB-Farbmodell. (vgl. Abbildung 5.3b) Dabei speichert jeder Kanal den Farbwert einer der drei Primärfarben. Die Farbtiefe pro Kanal beträgt ein Byte. Für die Darstellung der Debug-Ansicht wird eine `SurfaceView` verwendet. Dazu muss jedes Bild zunächst in ein `Bitmap`-Objekt umgewandelt werden. Hierbei wird als einziges Format ohne Alpha-Kanal nur RGB565 unterstützt. Dabei ist die Auflösung der Farbkanäle nicht mehr gleich. Mit sechs Bit ist die Auflösung des Grünanteils doppelt so hoch wie die des Rot- und Blauanteils.[10] Die Reduktion der Farbgenauigkeit von 2^{24} zu 2^{16} Farben ist für diese Anwendung nicht kritisch, da die Daten nach der Umwandlung nicht weiter verarbeitet werden.

5.7 Augmented Reality mit ARCore

Die Zentrale Klasse für die Visualisierung der AR-Inhalte stellt das `ArFragment` aus der Sceneform-Bibliothek dar. Es initialisiert die `Session` der AR-Anwendung, welche an den Lebenszyklus des Fragments gebunden ist. Das Fragment enthält zudem eine Instanz der `ArScene`-Klasse, welche das Kamerabild und die AR-Inhalte in einem Benutzeroberflächen-Objekt zusammengefügt darstellt. Für die Szene lässt sich ein Callback einrichten, welches für jedes Update der Szene aufgerufen wird. Aus dem Parameter dieser Methode entnimmt die Implementierung der Schnittstelle `FrameDataSource` das zugehörige Kamerabild. Somit muss die Anwendung nicht direkt auf die Kamera zugreifen.

5.8 Visualisierung mit Sceneform

Objekte werden in Sceneform durch `Node`-Objekte im dreidimensionalen Raum in der Szene platziert und durch ein `Renderable` dargestellt. Da die Visualisierung der erkannten Objekte mit AR nicht im Vordergrund steht, wird sie minimal gehalten, weshalb ein `ViewRenderable` verwendet wird.

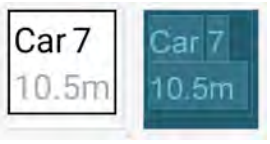


Abbildung 5.4: Layout für erkanntes Objekt

[10] `https://developer.android.com/reference/android/graphics/Bitmap.Config` (Abgerufen am 30.06.2020)

Diese stellen ein Layout für Android-Benutzeroberflächenenelemente auf einer Ebene in der dreidimensionalen Szene dar, wobei diese sich weiterhin wie in einer zweidimensionalen Anwendung verhalten. (s. Abbildung 5.4) Die Schilder werden am bestimmten Mittelpunkt des erkannten Objekts positioniert. Da sich diese Visualisierung der Inhalte nur inhaltlich und nicht strukturell unterscheiden, werden die zugehörigen `Node`-Objekte beim Start der Szene angelegt und bei jedem Tracking-Ergebnis inhaltlich angepasst und gegebenenfalls ausgeblendet. Dieser Mechanismus ist angelehnt an das Objektpool-Muster [26]. Dadurch wird vermieden, dass wiederholt `Node`-Objekte erstellt und zerstört werden müssen. Bei Entfernungen vom mehr als ein paar Metern werden diese Schilder jedoch so klein, dass sie kaum noch auf dem Smartphone-Bildschirm erkannt werden. Daher werden diese mit einem festen Abstand von drei Metern dargestellt. Die tatsächliche Entfernung des Objekts wird auf dem Schild dargestellt. (s. Abbildung 5.5)

Abbildung 5.5: Platzierung des Schildes mit der selben Entfernung des etwa sechseinhalb Meter entfernten Objekts (links) und mit festem Abstand von drei Metern (rechts)

5.9 Dependency-Injection

Für Dependency-Injection (DI) in Android-Anwendungen wird das Framework Dagger 2[11] verwendet. Es wird eingesetzt, um die Wartbarkeit von Softwareprojekten zu erleichtern, indem Konstruktion und Verwendung von Objekten getrennt werden. Zudem lässt sich darüber die Wiederverwendung langlebiger Objekte oder das Binden von Schnittstellen an konkrete Implementierungen umsetzten. Da der Abhängigkeitsbaum beim Kompilieren aufgelöst wird, hat es keinen Einfluss auf das Laufzeitverhalten der Anwendung.

[11] `https://dagger.dev/` (Abgerufen am 30.06.2020)

5.10 Objekt-Detektion mit TensorFlow Lite

TensorFlow Lite (TFLite) ist ein Framework für die Anwendung von trainierten Machine-Learning-Modellen auf leistungsschwächeren Geräten sowie das dazugehörige Dateiformat. Letzteres bündelt die Netzwerkdefinition und Werte für die Gewichte in eine Binärdatei. Die zentrale Komponente des Objekt-Detektors ist das `Interpreter`-Objekt von TFLite, das mit der Modell-Datei initialisiert wird. Um die Detektion auf einem Bild durchzuführen, muss dieses zunächst auf die Eingabegröße des Modells skaliert und anschließend in einem `ByteBuffer` gespeichert werden. Die Ausgabe des Modells wird in einem Array aus Arrays gespeichert. Bei dem verwendeten Netzwerk werden nicht die Zuversichtlichkeitswerte für die einzelnen Klassen direkt ausgegeben, sondern zur direkten Verwendung weiterverarbeitet. Dabei werden auf der ersten Position die Rahmen der erkannten Objekte, auf der zweiten die ermittelten Klassen der Objekte, auf der dritten die Zuversichtlichkeitswerte und auf der letzten die Anzahl der erkannten Objekte ausgegeben. (vgl. Abbildung 5.6) Dieses Feld wird jedoch ignoriert, da für den Zuversichtlichkeitswert ein Schwellwert verwendet wird.

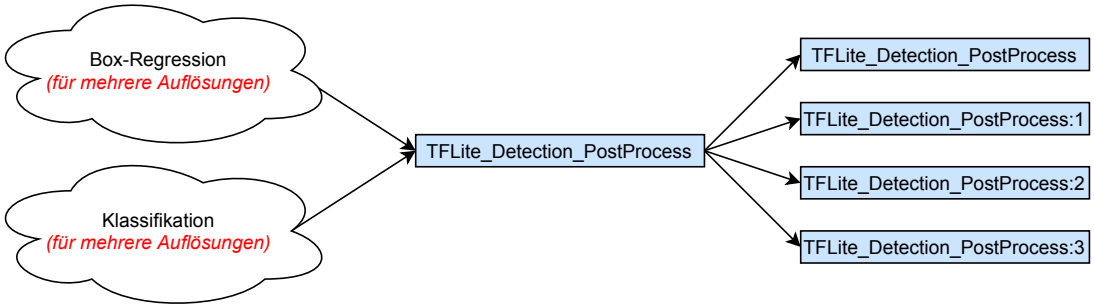


Abbildung 5.6: Nachverarbeitung (eingefärbt) im verwendeten Modell nach der letzten Ebene des eigentlichen Detektor-Netzes

Wie in Abschnitt 4.4 erwähnt, werden die erkannten Objekte gefiltert, sodass nur solche weiterverarbeitet werden, die zu relevanten Klassen gehören. Diese Klassen sind `PERSON`, `BICYCLE`, `CAR`, `MOTORCYCLE`, `BUS`, `TRAIN`, `TRUCK`, `CAT`, `DOG` und `SKATEBOARD`.

Die erkannten Objekte werden als Sammlung über ein ReactiveX-`Subject` an mögliche `Observer` geschickt, sodass die Weiterverarbeitung unabhängig vom Detektor und nebenläufig geschehen kann.

5.11 Objekt-Tracking

Wie bereits erwähnt, sind für das Tracking mehrerer Objekte mittels Tracking durch Detektion und das eines einzelnen Objekts mithilfe der Tracking-API von OpenCV unterschiedliche Vor- und Nachteile zu erwarten. Dementsprechend werden beide umgesetzt. Zur Laufzeit kann zum Vergleich zwischen beiden Verfahren umgeschaltet werden. Da der Tracker für Tracking durch Detektion lediglich auf dem Ergebnis des Detektors basiert, beim Tracking mit der OpenCV-API hingegen auch die Kamerabilder direkt verarbeitet werden, unterscheiden sich die Schnittstellen beider Tracker. (vgl. Abbildung 5.7)

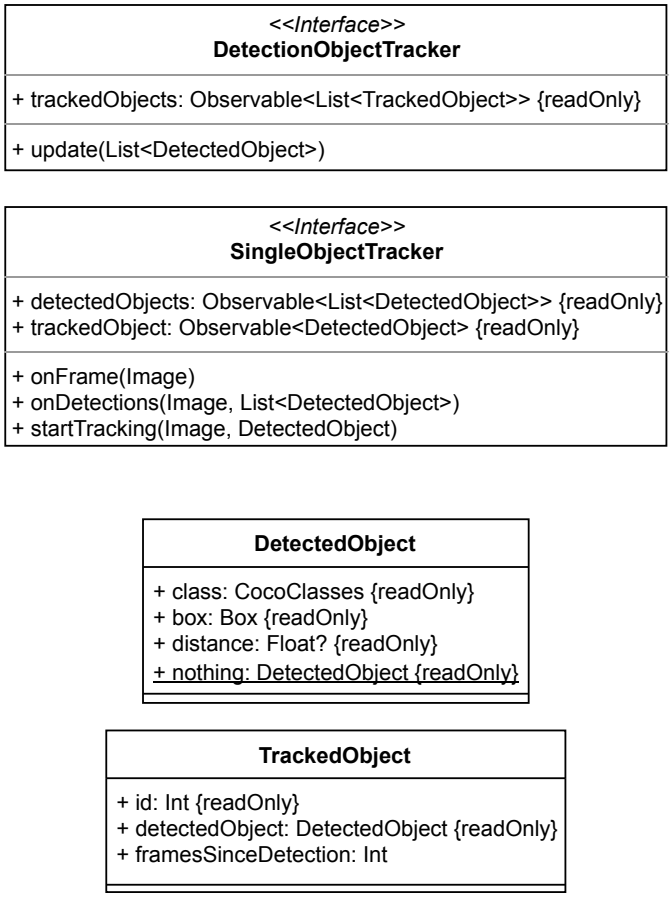


Abbildung 5.7: UML-Klassendiagramm der Schnittstellen (`Interface`) für die Implementierung des Objekt-Trackings und zugehöriger Daten-Klassen

5.11.1 Tracking durch Detektion

Algorithmus 2 beschreibt das Tracking in der `update`-Methode der Implementierung für die `DetectionObjectTracker`-Schnittstelle aus Abbildung 5.7. Der Detektor liefert erkannte Objekte in einer Liste von `DetectedObject`-Instanzen. Da zwei durch den Tracker verfolgte Objekte allerdings zu derselben Objektklasse gehören können, muss zur Unterscheidung beider Objekte in folgenden Bildern der Sequenz eine eindeutige Nummer (`id`) vergeben werden. Hierzu wird die Klasse `TrackedObject` verwendet. (vgl. Abbildung 5.7) Die Zuordnung A wird mit dem Munkres-Algorithmus aus Unterabschnitt 2.4.5 bestimmt.

Algorithmus 2 Tracking mehrerer Objekte durch Detektion

Vorbedingung: T_{n-1} ist Vektor der zuletzt verfolgten Objekte und K_{n-1} Vektor der zugehörigen Kalman-Filter, wobei $|T_{n-1}| = |K_{n-1}|$. T_n und K_n sind leere Vektoren. D ist ein Vektor der erkannten Objekte.

$T'_n \leftarrow \{\text{Vorhersage für } t \text{ durch } k | (t,k) \in (T_{n-1} \circ K_{n-1})\}$
$A \leftarrow$ Zuordnung von T'_n zu D
für $i = 1$ bis $|T'_n|$ **wiederhole**
  wenn A keine Zuweisung für T'_{ni} enthält **dann**
    wenn das Objekt T'_{ni} nicht abgelaufen ist **dann**
      $T_n \leftarrow T_n \oplus T'_{ni}$
      $K_n \leftarrow K_n \oplus K_{n-1i}$
    beende wenn
  sonst wenn $IOU(T_{n-1i}, A(T'_{ni}))$ signifikant ist **dann**
    $T_n \leftarrow T_n \oplus A(T'_{ni})$
    $K_n \leftarrow K_n \oplus K_{n-1i}$
  sonst {Es handelt sich um ein neues Objekt}
    $T_n \leftarrow T_n \oplus A(T'_{ni})$ mit fortlaufender Nummer
    $K_n \leftarrow K_n \oplus$ neuer Kalman-Filter für $A(T'_{ni})$
  beende wenn
beende für
für alle nicht zugewiesenen Objekte d aus D **wiederhole**
  $T_n \leftarrow T_n \oplus$ verfolgtes Objekt mit fortlaufender Nummer aus d
  $K_n \leftarrow K_n \oplus$ neuer Kalman-Filter für d
beende für
Korrektur von k mit t, wenn das Objekt in n erkannt wurde, für alle $(t,k) \in (T_n \circ K_n)$

5.11.2 OpenCV Tracking-API

Wie in Unterabschnitt 4.5.2 beschrieben, wird durch VOT mit der OpenCV Tracking-API nur ein Objekt verfolgt. Somit muss keine Nummer zur Identifizierung vergeben werden, weshalb die Klasse `TrackedObject` nicht verwendet wird. (vgl. Abbildung 5.7 und Abbildung 5.7) Die anderen Objekte werden zur Visualisierung auch weitergegeben. (vgl. `SingleObjectTracker` in Abbildung 5.7) Der OpenCV-Tracker wird mit jedem Kamerabild in der Sequenz aktualisiert. Dabei werden aufeinander folgende Bilder gezählt, für die das Tracking nicht erfolgreich war, um das Scheitern des Trackings zu erkennen. Der Tracking-Algorithmus wird mit dem vom Detektor erkannten Objektrahmen initialisiert. Da die Größe des Rahmens für das verfolgte Objekt jedoch nicht angepasst wird und der Fehler des Trackers über die Zeit möglicherweise anwächst, wird dieser in regelmäßigen Abständen mit dem Ergebnis des Detektors neu initialisiert. Da die Methode `onFrame` und die Methode `onDetection` von zwei unterschiedlichen Threads aufgerufen werden, stellt die Initialisierung des Trackers einen kritischen Abschnitt dar und wird über einen Monitor geschützt. Überlappt ein durch den Objekt-Detektor erkanntes Objekt stark mit dem aktuell durch den Tracker verfolgten Objekt, wird dieser mit `startTracking` neu initialisiert.

5.12 Einstellungen

Da die optimalen Einstellungen zuvor nicht bekannt sind, können diese über die Benutzeroberfläche angepasst werden. Nach dem Öffnen der Anwendung werden diese angezeigt (s. Abbildung 5.8) und beim Starten AR-Ansicht geladen.

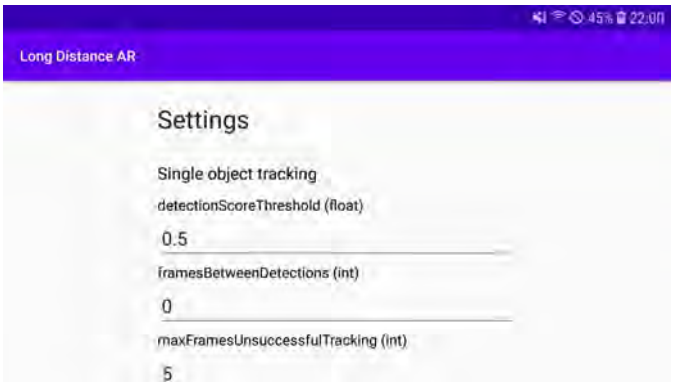


Abbildung 5.8: Benutzeroberfläche für die Einstellungen des Prototypen

5.12.1 Einstellungen für das Tracking mehrerer Objekte

Folgende Parameter für das Tracking mehrerer Objekte mit Tracking durch Detektion können durch den Benutzer eingestellt werden:

trackerLostTTL Es kann sein, dass ein Objekt durch Verdeckung oder andere Faktoren nicht in jedem Bild erkannt wird. Damit kurzzeitige Nicht-Erkennung nicht dazu führt, dass das Tracking des Objekts scheitert, kann das Objekt auch erst in einem nicht unmittelbar folgenden Bild in der Sequenz wieder erkannt werden. Wie viele Bilder dabei übersprungen werden dürfen, wird über diesen Parameter eingestellt.

sameInstanceMaxDist Dabei handelt es sich um den maximalen Abstand der Mittelpunkte zweier Objekte in aufeinander folgenden Bildern, damit diese derselben Instanz zugeordnet werden können.

classChangeCost Es ist theoretisch möglich, dass für ein Objekt in zwei aufeinander folgenden Bildern eine unterschiedliche Klasse bestimmt wird. So könnte beispielsweise ein Kleintransporter entweder als `TRUCK` oder `CAR` erkannt werden. In der Regel handelt es sich in so einem Fall allerdings nicht um dasselbe Objekt in beiden Bildern. Somit wird für diese Zuordnung der Wert dieses Parameters als zusätzliche Kosten veranschlagt. (vgl. Unterabschnitt 2.4.5)

measurementNoiseVariance Dieser Parameter wird zur Berechnung für die Varianz der Messung bei der Korrektur des Kalmanfilters verwendet. Da die einzelnen Messungen voneinander unabhängig sind und für alle die gleiche Varianz angenommen wird, ergibt sich die Kovarianz-Matrix aus der Multiplikation dieses Parameters mit der Einheitsmatrix. (vgl. R in Unterabschnitt 2.4.4)

meanAcceleration Hierbei handelt es sich um die erwartete Beschleunigung eines Objekts. Dieser Parameter wird zur Berechnung des Prozessrauschens in der Vorhersage durch den Kalmanfilter verwendet. (vgl. Q in Unterabschnitt 2.4.4)

detectionScoreThreshold Die Erkennungsschwelle für Objekte durch den Detektor in Prozent lässt sich über diesen Parameter einstellen. Er wird auch verwendet, wenn nur ein Objekt verfolgt wird.

5.12.2 Einstellungen für das Tracking eines einzelnen Objekts

Folgende Parameter für das Tracking eines einzelnen Objekts mit der Tracking-API von OpenCV können durch den Benutzer eingestellt werden:

framesBetweenDetection Dabei handelt es sich um die Anzahl an Bildern in der Sequenz, die der Detektor nach jedem Ausführen überspringt. Mit der Reduktion der durch den Detektor verarbeiteten Bilder soll die Auslastung durch diesen zugunsten des Trackings reduziert werden.

maxFramesUnsuccessfulTracking Dieser Parameter legt die maximale Anzahl an aufeinander folgenden Bildern fest, in denen das Objekt nicht erfolgreich verfolgt werden konnte, bevor das Tracking abgebrochen wird. Durch Verdeckungen, schnelle Bewegungen oder plötzliche Veränderung des Lichtverhältnisse ist davon auszugehen, dass das Tracking in solchen Fällen scheitert. Über diesen Parameter wird die Toleranz dafür eingestellt, sodass kurzzeitiges Scheitern des Trackings überbrückt werden kann.

detetionsBetweenTrackerReinitialization Objekt-Detektion ist genauer als Tracking. Über die Zeit steigt das Risiko, dass aufgrund von optischer Veränderung des verfolgten Objekts durch Drehung oder Bewegung das Tracking scheitert. Daher wird der Tracker neu initialisiert, wenn das verfolgte Objekt im aktuellen Bild durch den Detektor erkannt wurde. Mit diesem Parameter lässt sich einstellen, wie viele Ergebnisse des Detektors davor abgewartet werden.

sameInstanceIOU Dieser Parameter legt die Erkennungsschwelle für die Zuordnung von einem Objektrahmen im aktuellen Bild zu einem Objektrahmen aus dem vorherigen Bild in der Sequenz bei der erneuten Initialisierung des Trackers fest. Als Metrik dafür wird IOU zwischen den beiden Objektrahmen verwendet, da davon ausgegangen wird, dass zwischen zwei aufeinander folgenden Bildern die verfolgten Objekte signifikant überlappen, da die Veränderung zwischen den Bildern minimal ist.

6 Evaluation

Im folgenden werden die Ergebnisse der Evaluierung des entwickelten Prototypen beschrieben. Dabei wird vor allem auf Herausforderungen eingegangen, welche während der Konzeptionsphase nicht absehbar waren und deren Einfluss auf die Anwendung insgesamt untersucht.

6.1 Performance

Für die Evaluierungsphase wurde die Anwendung mit der „debug"-Variante erstellt. Somit werden keine Optimierungen durch den Compiler vorgenommen, was eventuell einen geringen Einfluss auf die Performance des Prototypen hat.

Da die rechenintensiven Verarbeitungsstufen (Detektor und Tracker) auf jeweils eigenen Threads ausgeführt werden, wird der Thread der Benutzeroberfläche dadurch nicht unmittelbar[1] blockiert. Lediglich Aktualisierungen der Benutzeroberfläche - der Debug-Ansicht und der AR-Inhalte - werden auf diesem Thread ausgeführt, weshalb die Bildrate der AR-Ansicht zwischen zwanzig und dreißig Hertz schwankt. Das Kamerabild erscheint trotzdem flüssig. Da die Visualisierung der erkannten Objekte jedoch nicht zeitgleich mit der Aktualisierung des Kamerabilds und langsamer als diese geschieht, fügen sich die überlagerten Inhalte nicht perfekt flüssig in die AR-Ansicht ein. Wie in Abschnitt 4.1 beschrieben, ist dies allerdings ein notwendiger Kompromiss, um die sonst niedrigere Bildwiederholrate des Kamerabilds zu vermeiden.

[1] Da es sich bei Android nach [30] nicht um ein Echtzeit-fähiges Betriebssystem handelt, ist die Kontrolle über das Scheduling beschränkt.

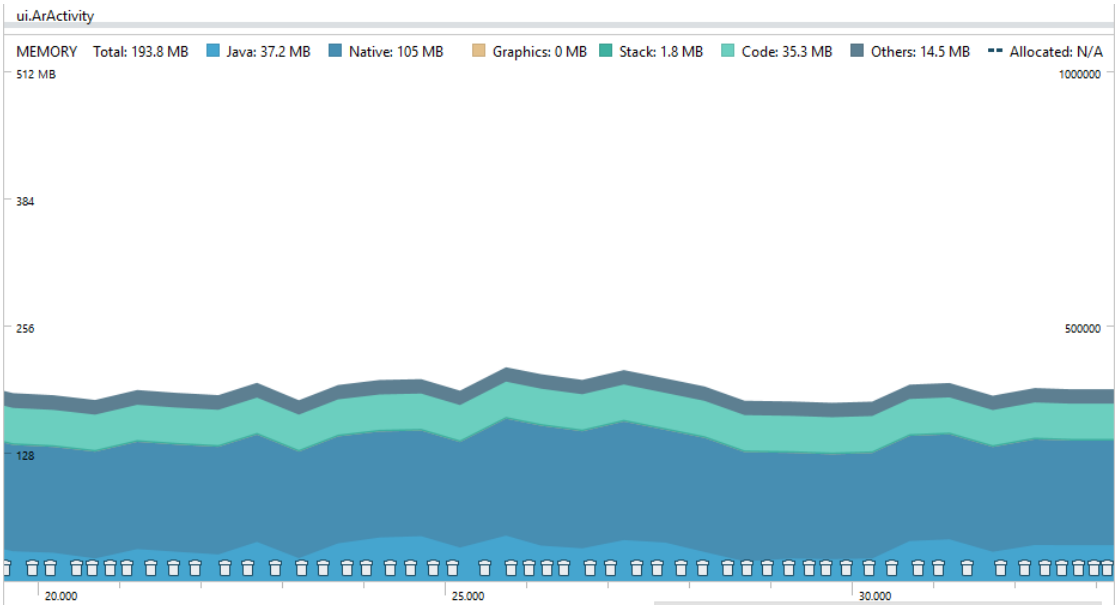


Abbildung 6.1: Speicherverbrauch der Anwendung über zehn Sekunden im AndroidStudio-Profiler

Abbildung 6.1 zeigt, dass der Speicherverbrauch der Anwendung dauerhaft bei ungefähr 200 Megabyte liegt. Da zur Speicherung eines Kamerabildes immer ein neuer `ByteBuffer` erstellt wird, wird in regelmäßigen und kurzen Abständen „garbage collection“ durchgeführt. (Zu sehen in Abbildung 6.1 an den Symbolen direkt über der Zeitachse) Diese beträgt allerdings nur wenige Millisekunden und ist somit im Vergleich zur Laufzeit des Detektors und des Trackings nicht ausschlaggebend. Durch Wiederverwendung der `ByteBuffer`-Objekte mithilfe eines Objektpools kann dies reduziert werden. Allerdings würde das auch die Komplexität der Anwendung deutlich erhöhen. Dazu müsste genau bekannt sein, wie viele Objekte gleichzeitig benötigt werden, wobei nur eine geringe Anzahl von unkomprimierten Bildern zeitgleich im Speicher gehalten werden kann, und dabei die Interaktion mit dem Objektpool Thread-sicher sein muss. Die Verarbeitungsstufen für Objekt-Detektion und -Tracking werden jeweils nur von einem Thread und an einer Stelle in der Anwendung benutzt. Somit können die zur Verarbeitung benötigten OpenCV-Bild-Matrizen und das TFLite-Modell nur einmal alloziert und danach wieder verwendet werden. (vgl. „Native“ in Abbildung 6.1) Der Speicherbedarf dafür bleibt also nahezu konstant, ist aber im Vergleich zu den meisten Android-Anwendungen relativ hoch. Dies ist jedoch auch der Tatsache geschuldet, dass zwei Tracking-Lösungen sowie eine Debug-Visualisierung implementiert ist, welche zueinander asynchron sind und somit Kopien von Ressourcen verwenden.

6.2 Einstellungen

Bei der Evaluierung des Prototypen wurden die Werte aus Tabelle 6.1 für die in Abschnitt 5.12 beschriebenen Einstellungen gewählt. Mit diesen wurden die besten Ergebnisse während des Testens erreicht. Dabei ist aufgefallen, dass ein höherer Wert für „framesBetweenDetections“ keine positive Auswirkung auf die Performanz der Anwendung hat, ein höherer Wert für „detectionsBetweenTrackerReinitialization“ verhindert jedoch Verzögerungen durch die verhältnismäßig langsame Initialisierung des VOT von OpenCV.

Einstellung	Wert
trackerLostTTL	6
sameInstanceMaxDist	150,00
classChangeCost	1,00
measurementNoiseVariance	0,10
meanAcceleration	20,00
detectionScoreThreshold	0,60
framesBetweenDetection	0
maxFramesUnsuccessfulTracking	15
detetionsBetweenTrackerReinitialization	10
sameInstanceIOU	0,30

Tabelle 6.1: Einstellungen für Objekt-Detektion und -Tracking

6.3 Objekt-Detektion

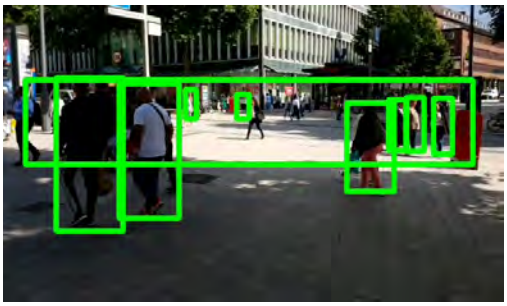
In Kombination mit Objekt-Tracking ist die Laufzeit des Detektors merklich länger als die in der Tabelle 4.1 für das Modell gemessene Zeit.

Die Objektrahmen umfassen das Objekt meist passend und sind nur gelegentlich deutlich zu groß oder klein. Bei einem Schwellwert von 0,6 für die Zuversichtlichkeit von erkannten Objekten enthält das Ergebnis des Detektors fast nur richtig positive Erkennungen. Es werden allerdings möglicherweise nicht alle zu erkennenden Objekte tatsächlich erkannt, die Falsch-negativ-Rate ist also vergleichsweise hoch.

Bei deutlich niedrigeren Schwellwerten von zum Beispiel 0,5 oder 0,4 ist die Falsch-positiv-Rate des Detektors jedoch sehr hoch. Neben nicht vorhandenen Objekten werden einige Objekte doppelt erkannt. In Abbildung 6.2a wird beispielsweise das prominente Fahrzeug auf der Kreuzung nicht erkannt, während in Abbildung 6.2b ein nicht existierendes Objekt erkannt wird, das fast die Breite des Bildes besitzt.

Das verwendete Modell verfügt über keine Informationen aus vorherigen Bildern der Sequenz. Es kommt zudem nicht selten vor, dass erkannte Objekte in folgenden Bildern der Sequenz nicht wiedererkannt werden. Die Einschränkung des verwendeten Modells von maximal zehn gleichzeitig erkannten Objekten scheint dabei nicht die Ursache zu sein, da dieses Phänomen auch auftreten kann, wenn weniger Objekte erkannt wurden. Vielmehr ist davon auszugehen, dass Zuversichtlichkeitswerte für diese Erkennungen sehr nah am Schwellwert liegen. Somit kommt es häufig dazu, dass in zwei direkt aufeinander folgenden Bildern unterschiedliche Objekte erkannt werden, obwohl sich diese sehr ähnlich sind. (s. Abbildung 6.2c)

(a) falsch negative Erkennung



(b) falsch positive Erkennung

(c) Sequenz aus ähnlichen Bildern mit unterschiedlichen Ergebnissen für Objekt-Detektion

Abbildung 6.2: Scheitern der Objekt-Detektion

Das Modell scheint außerdem unterschiedlich robust in der Erkennung von Objekten unterschiedlicher Klassen oder Blickwinkel zu sein. So werden zum Beispiel Fahrrad- oder Motorradfahrer meist nur als `PERSON` erkannt und während Autos in fast allen Fällen bei seitlicher Ansicht erkannt werden, hat die Erkennung von schräg vor oder hinter dem Fahrzeug teilweise deutlich niedrigere Zuverlässigkeitswerte.

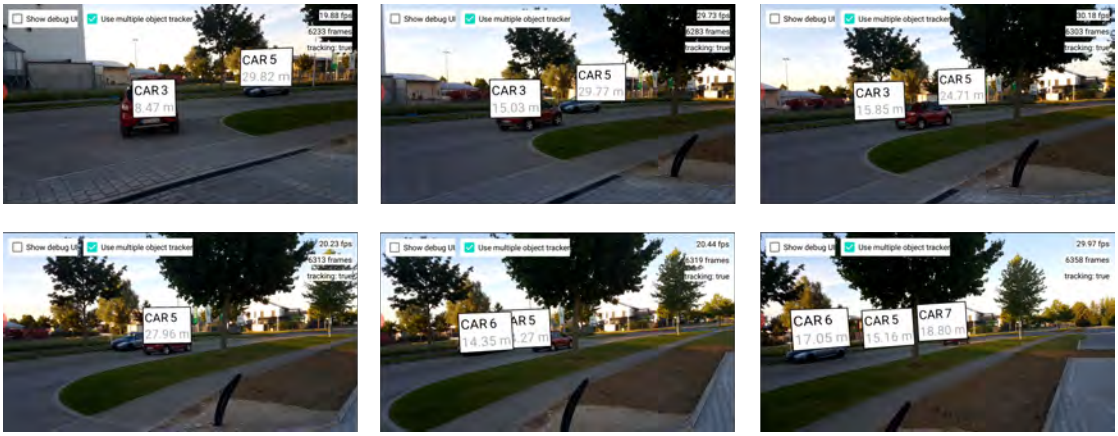
6.4 Tracking

Im Rahmen des Prototypen wurden sowohl das Tracking mehrerer Objekte mit Tracking durch Detektion als auch das Tracking eines ausgewählten Objekts mit VOT implementiert. Im Rahmen der Evaluierung zeigen sich dabei unterschiedliche Vor- und Nachteile für beide Ansätze.

6.4.1 Tracking mehrerer Objekte

Bei großzügigen Abständen zwischen den zu erkennenden Objekten auf dem Bildschirm und zuverlässiger Erkennung funktioniert Tracking durch Detektion dieser Objekte gut. Da es nicht auf visuellen Merkmalen beruht, ist es zudem robust gegenüber Rotationen und Verformungen.

(a) Vorhergesagte Position (blau) und Korrektur durch Detektion (grün) mithilfe des Kalman-Filters (Zugeschnittenes Bild)



(b) Seqzenz von Screenshots des Prototypen mit teilweise korrektem Tracking

Abbildung 6.3: Ergebnisse des Tracking durch Detektion

Abbildung 6.3a zeigt die Positionsbestimmung unter Verwendung des Kalman-Filters in drei aufeinander folgenden Bildern. Im ersten Bild liegt die vermutete Position noch weit hinter der Erkannten. Im weiteren Verlauf verbessert sich die Genauigkeit der Positionsvorhersage. Somit steigt die Wahrscheinlichkeit, dass das Objekt passend zugeordnet wird. Allerdings liegen oft nur wenige, unregelmäßig verteilte Erkennungen des Objekts durch den Detektor vor, welche nicht ausreichend sind, um mithilfe des Kalman-Filters die Position passend genug vorherzusagen, wenn das Objekt zwischenzeitlich nicht erkannt wurde.

Für den Fall, dass Objekte zeitweise nicht erkannt werden ist über die Einstellung „trackerLostTTL“ festgelegt wie viele Tracking-Ergebnisse rein auf der Vorhersage durch den Kalman-Filter basieren dürfen. Dies birgt jedoch das Risiko für Objektvertauschung. Über „sameInstanceMaxDist“ kann das Risiko dafür zwar verhindert werden, allerdings sorgt dies bei einem zu geringen Wert für „Geisterobjekte“, da bei erneuter Erkennung das Objekt möglicherweise zu weit von der vorhergesagten Position entfernt ist und somit als neues Objekt erkannt wird. Ähnlich verhält es sich mit „classChangeCost“. Der Fall, dass beispielsweise ein Motorradfahrer teilweise als `PERSON` oder `MOTORCYCLE` erkannt wird, tritt nicht häufig genug auf, um das Risiko einer Vertauschung von Objekten sehr unterschiedlicher Klassen in Kauf zu nehmen. Es gibt keine universell optimalen Einstellungen, da diese stark von der Anzahl und Geschwindigkeit der erkannten Objekte sowie deren Position im Bild abhängen.

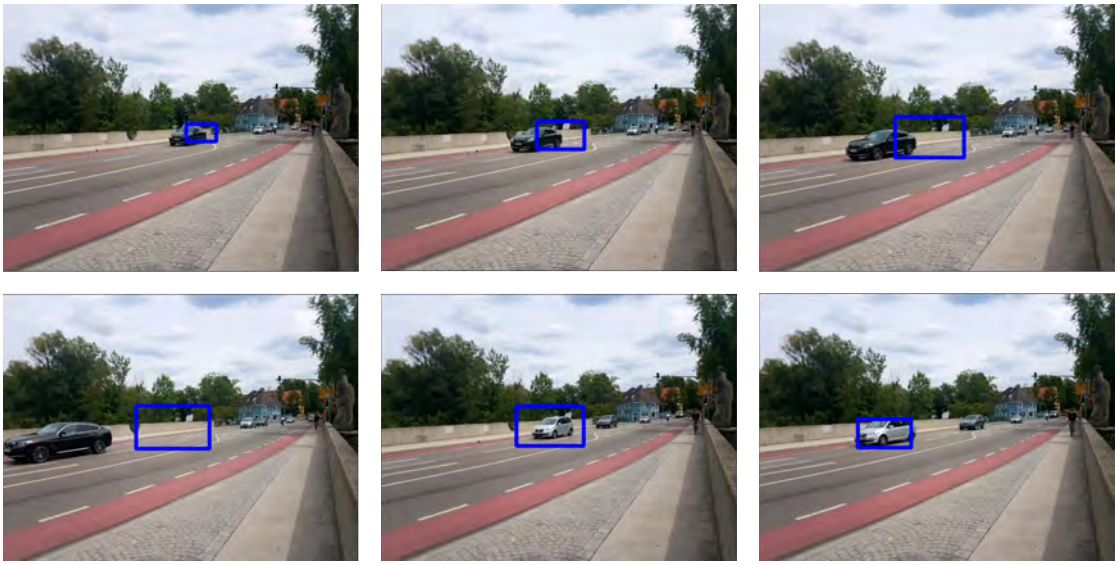
Aufgrund der verhältnismäßig langen Laufzeit des Detektors folgt die AR-Visualisierung meist etwas versetzt hinter dem Objekt und liegt nicht direkt auf dem Objekt selbst. Durch die Vorhersage der Position mit dem Kalman-Filter zwischen den Ergebnissen des Detektors könnte die Position für jedes Bild aktualisiert werden. Da die Vorhersage des Kalman-Filters allerdings zu oft stark von der tatsächlichen Position des Objekts abweicht, stellt dies keine hilfreiche Verbesserung dar.

Abbildung 6.3 zeigt diese Probleme. In der oberen Reihe ist das Tracking zunächst erfolgreich. Im dritten Bild wurden keine Objekte durch den Detektor erkannt. Daraufhin wird das abbiegende Fahrzeug mit dem geparkten verwechselt, da die vorhergesagte Position für „CAR 3“ nun zu stark von der eigentlichen aufgrund der ursprünglichen Bewegungsrichtung abweicht. Im folgenden wird das Fahrzeug aufgrund partieller Verdeckung durch einen Baum wieder nicht erkannt. Somit ist die vermutete Bewegung von „CAR 5“ sehr gering, das Fahrzeug wird als neues Objekt erkannt und „CAR 5“ wird zu einem „Geisterobjekt“.

6.4.2 Tracking eines einzelnen Objekts

Durch Verwendung von VOT kann ein insgesamt flüssigeres Ergebnis für das ausgewählte Objekt erzielt werden. Abbildung 6.4a zeigt, dass ein bewegtes Objekt auch bei partieller Verdeckung verfolgt werden kann.

(a) Screenshots des Prototypen bei erfolgreichem Tracking



(b) Objektverwechslung aufgrund eines Tracking-Fehlers. Der verwendete Algorithmus ist nicht robust im Bezug auf die Erkennung von Tracking-Fehlern.

Abbildung 6.4: Ergebnisse des VOT

Leider ist der gewählte Algorithmus nicht robust gegenüber visuellen Veränderungen des Tracking-Ziels durch Drehung. Wird beispielsweise ein vorbeifahrendes Fahrzeug verfolgt, bricht das Tracking oft ab, sobald es sich wieder entfernt. Scheitern des Trackings durch zu starke Abweichung von der Objektposition werden allerdings oft nicht erkannt. Wie in Abbildung 6.4b zu sehen, kann dies dazu führen, dass nicht existierende oder andere Objekte weiter verfolgt werden. Manchmal wird durch den Algorithmus ein anderes Objekt damit „eingefangen“.

6.5 Entfernungsbestimmung

Abbildung 6.5 zeigt die Ungenauigkeit der Entfernungsbestimmung eines erkannten Objekts direkt vor dem Benutzer. Dabei zeigt sich, dass diese auf geringe Distanzen verhältnismäßig präzise ist, der Fehler jedoch schon bei mittlerer Distanz von etwa 15 Metern sehr groß wird. Das verwendete Verfahren zur Entfernungsbestimmung der Objekte (vgl. Unterabschnitt 4.6.3) ist in der Theorie sehr präzise. Allerdings schließt die untere Kante des ermittelten Rahmens nicht immer passend an das Objekts an und auch wenn der Boden der Umgebung eine perfekte Ebene darstellt, gibt es einen geringen Fehler zu der ermittelten Ebene. Mit zunehmender Entfernung wird dieser Fehler signifikant.

In Anbetracht der vorgestellten Optionen liefert der angewendete Ansatz jedoch das beste Ergebnis und der Fehler in der Distanzbestimmung ist gering genug, um beispielsweise eine glaubwürdige Skalierung und Sortierung für realistische Verdeckung digitaler Inhalte für eine AR-Anwendung damit umzusetzen. Erfordert eine Anwendung genaue Abstandsermittlung, wie zum Beispiel eine Fahrerassistenz, kann sie mit dieser Methode nicht realisiert werden.

(a) $-0,25$ Meter auf 5 Meter

(b) $0,61$ Meter auf 10 Meter

(c) $3,51$ Meter auf 15 Meter

Abbildung 6.5: Fehler bei der Entfernungsbestimmung

7 Fazit

Der Prototyp hat in der Evaluierung gezeigt, dass AR-Anwendungen für handelsübliche Smartphones in dynamischen Umgebungen prinzipiell technisch möglich sind und mithilfe von Objekt-Detektion auf Basis von CNNs diese verstehen können. Für Tracking auf Basis der Ergebnisse des Detektors sind diese jedoch zu ungenau und die Erkennung unzuverlässig. Auf Geräten mit höherer Rechenleistung können hier gegebenenfalls bessere Ergebnisse erzielt werden. Aufgrund der niedrigen Frequenz der Ergebnisse kommt es bei der Zuordnung von nahegelegenen Objekten derselben Klasse teilweise zu Verwechslungen. Durch Zuordnung der Objekte im dreidimensionalen Raum unter Verwendung der ermittelten Entfernung könnte dies möglicherweise abgemildert werden.
Die Anwendung eines OpenCV-Algorithmus für VOT kann zwar bei der Verfolgung eines Objekts unterstützen und somit für einen flüssigeren Gesamteindruck sorgen, jedoch ist dieser selbst relativ fehleranfällig. Zuverlässigere Verfahren haben hingegen eine deutlich längere Laufzeit.

Bei der Evaluierung des Prototypen ist auch aufgefallen, dass eine solche Anwendung weitere Herausforderungen im Bereich der Benutzerfreundlichkeit hat. Der verhältnismäßig kleine Bildschirm des Testgeräts zusammen mit der Vielzahl und durchschnittlichen Bewegungsgeschwindigkeit der Verkehrsteilnehmer führt dazu, dass der Benutzer die Informationen nur selektiv verarbeiten kann. Es entsteht das Gefühl eines „Tunnelblicks“. Dies ist gegebenenfalls auch relevant für Szenarien, in denen der Benutzer aus Sicherheitsgründen nicht zu stark von seiner Umgebung abgelenkt werden darf. Da sich diese Thesis rein auf die technische Umsetzung konzentriert, wurde keine Benutzerstudie durchgeführt, um dies weiter zu untersuchen.

7.1 Anwendung

Die zuverlässige Erkennung und Verfolgung aller Verkehrsteilnehmer in einem beliebigen urbanen Szenario ist mit dem für den Prototypen gewählten Ansatz somit noch nicht möglich. Allerdings ermöglicht die in diesem Rahmen umgesetzte Erkennung und grobe Lokalisierung neue Anwendungen im Bereich AR unter Verwendung bestehender Hardware. In Szenarien, bei denen die Anzahl der relevanten Objekte geringer und möglicherweise bekannt ist, kann eine Anwendung auf eine variable oder sich verändernde Umgebung reagieren. Mögliche Anwendungsfälle sind zum Beispiel optische Navigation und Information auf Messen und Ausstellungen, interaktive Unterhaltungsangebote auf saisonalen Freizeitveranstaltungen oder digitaler Wartungsanleitungen für die Industrie.

7.2 Weiterentwicklung

Ende 2019 kündigte Google eine Erweiterung der API für ARCore in einem Blogpost an, mit der Pixel-genau die Bildtiefe des Kamerabildes bestimmt werden kann.[1] Mit Hilfe dieser Information kann gegebenenfalls das zur Bestimmung der Entfernung verwendete Verfahren aus Unterabschnitt 4.6.3 abgelöst werden, um auch Objekte in komplexem Terrain korrekt zu verorten. Du et al. [9] erwähnen jedoch eine maximale Reichweite von acht Metern, in welchem Fall beide Verfahren angewendet werden sollten. Das verwendete Testgerät unterstützt diese neue Funktion jedoch leider nicht mehr.[2]

Wie in Abschnitt 6.3 und Unterabschnitt 6.4.1 erwähnt, ist das Ergebnis der Objekt-Detektion in aufeinander folgenden Bildern zu inkonsistent, um Tracking durch Detektion effektiv umzusetzen. Dieses Problem könnte möglicherweise durch die Verwendung eines rückgekoppelten Netzes für Objekt-Detektion abgemildert werden. Bei einer ausreichend hohen Samplingrate ist für aufeinander folgende Kamerabilder anzunehmen, dass zuvor erkannte Objekte im nächsten Bild in der unmittelbaren Umgebung wiedererkannt werden. Durch Verstärkung der Zuversichtlichkeitswerte von Objekt-Kandidaten und der zugehörigen Klasse in dieser Region kann gegebenenfalls die Erkennungsschwelle im folgenden Bild überschritten werden. Die Entwicklung einer solchen Architektur ist nicht trivial, allerdings gibt es bereits vielversprechende Vorarbeit zu diesem Ansatz [23, 24].

[1] `https://developers.googleblog.com/2019/12/blending-realities-with-arcore-depth-api.html` (Abgerufen am 25.06.2020)

[2] `https://developers.google.com/ar/discover/supported-devices` (Abgerunfen am 25.06.2020)

7.3 Ergebnis

Im Rahmen dieser Thesis wurde gezeigt, dass dynamische AR-Apps für Außenbereiche möglich sind. Wie erwähnt, sind mit dem umgesetzten Lösungsansatz bereits Anwendungsfälle denkbar. Allerdings wurde auch festgestellt, dass die Benutzerinteraktion mit einer solchen Anwendung weiter untersucht werden muss. Darüber hinaus kommt die verwendete Hardware mit existierenden Modellen für Objekterkennung an ihre Grenzen. Wie beschrieben, handelt es sich dabei aber um kein reines Hardware-Problem und ein neuer Algorithmus ist notwendig. Im Hinblick auf jüngste Entwicklungen auf diesem Gebiet ist die Zukunft jedoch aussichtsreich.

Literaturverzeichnis

[1] Steve Aukstakalnis. 2017. *Practical augmented reality : a guide to the technologies, applications, and human factors for AR and VR.* Addison-Wesley, Boston.

[2] Engineer Bainomugisha, Andoni Lombide Carreton, Tom van Cutsem, Stijn Mostinckx, and Wolfgang de Meuter. 2013. A Survey on Reactive Programming. *ACM Comput. Surv.* 45, 4, Article 52 (Aug. 2013), 34 pages. https://doi.org/10.1145/2501654.2501666

[3] Erik Bochinski, Volker Eiselein, and Thomas Sikora. 2017. High-Speed Tracking-by-Detection Without Using Image Information. In *International Workshop on Traffic and Street Surveillance for Safety and Security at IEEE AVSS 2017.* Lecce, Italy. http://elvera.nue.tu-berlin.de/files/1517Bochinski2017.pdf

[4] Alexey Bochkovskiy, Chien-Yao Wang, and Hong-Yuan Mark Liao. 2020. YOLOv4: Optimal Speed and Accuracy of Object Detection. arXiv:2004.10934 [cs.CV]

[5] David Bolme, J. Beveridge, Bruce Draper, and Yui Lui. 2010. Visual object tracking using adaptive correlation filters. *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 2544–2550. https://doi.org/10.1109/CVPR.2010.5539960

[6] J.-Y. Bouguet. 1999. Pyramidal implementation of the lucas kanade feature tracker.

[7] G. Bradski. 2000. The OpenCV Library. *Dr. Dobb's Journal of Software Tools* (2000).

[8] D. Chatzopoulos, C. Bermejo, Z. Huang, and P. Hui. 2017. Mobile Augmented Reality Survey: From Where We Are to Where We Go. *IEEE Access* 5 (2017), 6917–6950.

[9] Ruofei Du, Eric Turner, Max Dzitsiuk, Luca Prasso, Ivo Duarte, Jason Dourgarian, Joao Afonso, Jose Pascoal, Josh Gladstone, Nuno Cruces, Shahram Izadi, Adarsh

Kowdle, Konstantine Tsotsos, and David Kim. 2020. DepthLab: Real-time 3D Interaction with Depth Maps for Mobile Augmented Reality. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology (UIST)*. ACM, 14.

[10] Gunnar Farnebäck. 2003. Two-Frame Motion Estimation Based on Polynomial Expansion. In *Image Analysis*, Josef Bigun and Tomas Gustavsson (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 363–370.

[11] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. 2016. *Deep Learning*. MIT Press. `http://www.deeplearningbook.org`.

[12] Tingbo Hou, Adel Ahmadyan, Liangkai Zhang, Jianing Wei, and Matthias Grundmann. 2020. MobilePose: Real-Time Pose Estimation for Unseen Objects with Weak Shape Supervision. arXiv:2003.03522 [cs.CV]

[13] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. arXiv:1704.04861 [cs.CV]

[14] Hsiu-O Hsu. 2008. Method for calculating distance and actual size of shot object. `https://patents.google.com/patent/US20080101784A1/en` US Patent App. 11/716,466.

[15] Noor A Ibraheem, Mokhtar M Hasan, Rafiqul Z Khan, and Pramod K Mishra. 2012. Understanding color models: a review. *ARPN Journal of science and technology* 2, 3 (2012), 265–275.

[16] Paul Jaccard. 1902. Lois de distribution florale dans la zone alpine. *Bulletin de la Société Vaudoise des Sciences Naturelles* 38, 144 (1902), 72. `https://doi.org/10.5169/seals-266762`

[17] Licheng Jiao, Fan Zhang, Fang Liu, Shuyuan Yang, Lingling Li, Zhixi Feng, and Rong Qu. 2019. A Survey of Deep Learning-Based Object Detection. *IEEE Access* 7 (2019), 128837–128868. `https://doi.org/10.1109/access.2019.2939201`

[18] Z. Kalal, K. Mikolajczyk, and J. Matas. 2010. Forward-Backward Error: Automatic Detection of Tracking Failures. In *2010 20th International Conference on Pattern Recognition*. 2756–2759.

[19] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In *Advances in Neural Information Processing Systems 25*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger (Eds.). Curran Associates, Inc., 1097–1105. http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf

[20] Ville Lehtola, Heikki Huttunen, François Christophe, and Tommi Mikkonen. 2017. Evaluation of Visual Tracking Algorithms for Embedded Devices. 88–97. https://doi.org/10.1007/978-3-319-59126-1_8

[21] Tsung-Yi Lin, Michael Maire, Serge Belongie, Lubomir Bourdev, Ross Girshick, James Hays, Pietro Perona, Deva Ramanan, C. Lawrence Zitnick, and Piotr Dollár. 2014. Microsoft COCO: Common Objects in Context. arXiv:1405.0312 [cs.CV]

[22] Luyang Liu, Hongyu Li, and Marco Gruteser. 2019. Edge Assisted Real-time Object Detection for Mobile Augmented Reality. 1–16. https://doi.org/10.1145/3300061.3300116

[23] Mason Liu and Menglong Zhu. 2017. Mobile Video Object Detection with Temporally-Aware Feature Maps. arXiv:1711.06368 [cs.CV]

[24] Mason Liu, Menglong Zhu, Marie White, Yinxiao Li, and Dmitry Kalenichenko. 2019. Looking Fast and Slow: Memory-Guided Mobile Video Object Detection. https://arxiv.org/pdf/1903.10172.pdf

[25] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C. Berg. 2016. SSD: Single Shot MultiBox Detector. *Lecture Notes in Computer Science* (2016), 21–37. https://doi.org/10.1007/978-3-319-46448-0_2

[26] Kircher Michael and J Prashant. 2002. Pooling pattern. *7th European Confer ence on Pattern Languages of Programs, Irsee, Germany* (2002). http://www.kircher-schwanninger.de/michael/publications/Pooling.pdf

[27] Paul Milgram, Haruo Takemura, Akira Utsumi, and Fumio Kishino. 1994. Augmented reality: A class of displays on the reality-virtuality continuum. *Telemanipulator and Telepresence Technologies* 2351 (01 1994). https://doi.org/10.1117/12.197321

[28] James R. Munkres. 1957. Algorithms for the Assignment and Transportation Problems. *Journal of The Society for Industrial and Applied Mathematics* 10 (1957), 196–210. https://doi.org/10.1137/0105003

[29] Paweł Nowacki and Marek Woda. 2020. *Capabilities of ARCore and ARKit Platforms for AR/VR Applications*. 358–370. https://doi.org/10.1007/978-3-030-19501-4_36

[30] L. Perneel, H. Fayyad-Kazan, and M. Timmerman. 2012. Can Android be used for real-time purposes?. In *2012 International Conference on Computer Systems and Industrial Informatics*. 1–6.

[31] Jinmeng Rao, Yanjun Qiao, Fu Ren, Junxing Wang, and Qingyun Du. 2017. A Mobile Outdoor Augmented Reality Method Combining Deep Learning Object Detection and Spatial Relationships for Geovisualization. *Sensors* 17, 9 (Aug. 2017), 1951. https://doi.org/10.3390/s17091951

[32] Paul Read. 2000. *Restoration of motion picture film*. Butterworth-Heinemann, Oxford Boston. https://books.google.com/books?id=jzbUUL0xJAEC&pg=PA24

[33] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. 2015. You Only Look Once: Unified, Real-Time Object Detection. arXiv:1506.02640 [cs.CV]

[34] Joseph Redmon and Ali Farhadi. 2016. YOLO9000: Better, Faster, Stronger. arXiv:1612.08242 [cs.CV]

[35] Joseph Redmon and Ali Farhadi. 2018. YOLOv3: An Incremental Improvement. arXiv:1804.02767 [cs.CV]

[36] Joseph Rourke. 1998. *Computational geometry in C*. Cambridge University Press, Cambridge, UK New York, NY, USA.

[37] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. 2015. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)* 115, 3 (2015), 211–252. https://doi.org/10.1007/s11263-015-0816-y

[38] Jianbo Shi and Carlo Tomasi. 1994. Good Features to Track. 593–600. https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.36.2669

[39] Artem Syromiatnikov and Danny Weyns. 2014. A Journey through the Land of Model-View-Design Patterns. *Proceedings - Working IEEE/IFIP Conference on Software Architecture 2014, WICSA 2014*, 21–30. https://doi.org/10.1109/WICSA.2014.13

[40] Emanuele Trucco and Alessandro Verri. 1998. *Introductory techniques for 3-D computer vision*. Vol. 201. Prentice Hall Englewood Cliffs.

[41] Qing Wang, Feng Chen, Wenli Xu, and Ming-Hsuan Yang. 2011. An Experimental Comparison of Online Object Tracking Algorithms. *Proceedings of SPIE - The International Society for Optical Engineering* (09 2011). https://doi.org/10.1117/12.895965

[42] Fadi Wedyan, Reema Freihat, Ibrahim Aloqily, and Suzan Wedyan. 2016. JoGuide: A Mobile Augmented Reality Application for Locating and Describing Surrounding Sites.

[43] Junhui Wu, Dong Yin, Jie Chen, Yusheng Wu, Huiping Si, and Kaiyan Lin. 2020. A Survey on Monocular 3D Object Detection Algorithms Based on Deep Learning. *Journal of Physics: Conference Series* 1518 (apr 2020), 012049. https://doi.org/10.1088/1742-6596/1518/1/012049

[44] Khalid Yousif, Alireza Bab-Hadiashar, and Reza Hoseinnezhad. 2015. An Overview to Visual Odometry and Visual SLAM: Applications to Mobile Robotics. *Intelligent Industrial Systems* 1, 4 (Nov. 2015), 289–311. https://doi.org/10.1007/s40903-015-0032-7

[45] Zhengxia Zou, Zhenwei Shi, Yuhong Guo, and Jieping Ye. 2019. Object Detection in 20 Years: A Survey. arXiv:1905.05055 [cs.CV]

A Anhang

A.1 Zusätzliche Abbildungen

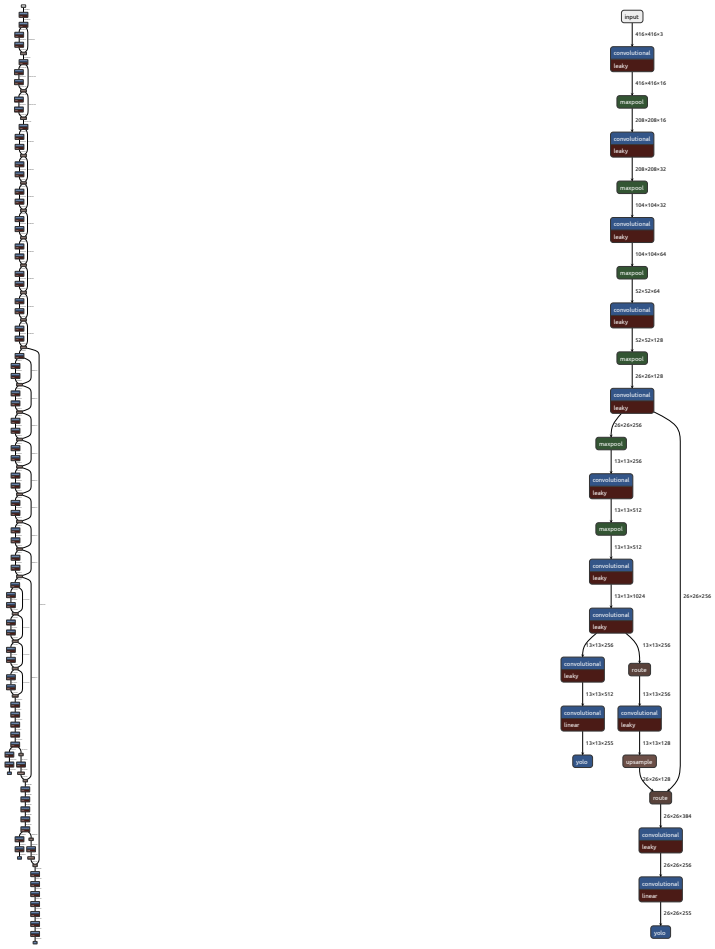


Abbildung A.1: YOLOv3 und Tiny YOLOv3 mit reduzierter Komplexität für deutlich kürzere Laufzeit auf leistungsschwächeren Geräten[1]

[1] Konfigurationen von `https://pjreddie.com/darknet/yolo` (Abgerufen 18.06.2020) und visualisiert mit `https://lutzroeder.github.io/netron/` (Abgerufen 18.06.2020)

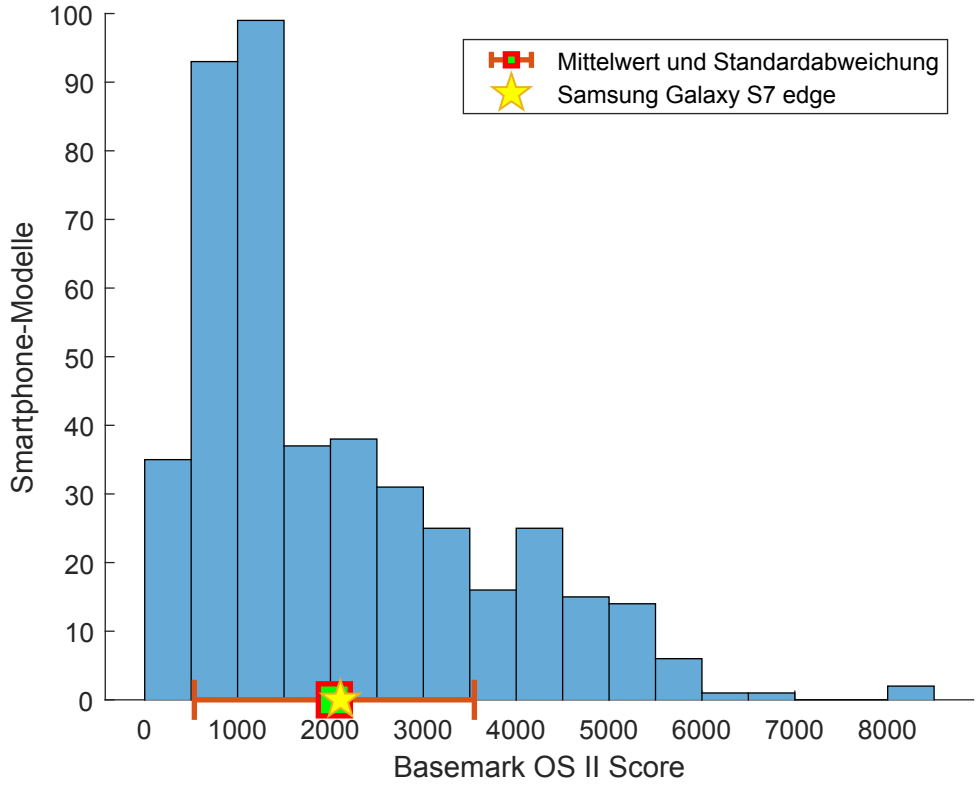


Abbildung A.2: Rechenleistung verschiedener Smartphone-Modelle[2]

[2]Ergebnisse verschiedener Smartphones mit dem Benchmark Basemark OS II https://www.basemark.com/benchmarks/legacy-products/#basemarkOSII (Abgerufen am 20.06.2020) von https://www.gsmarena.com/benchmark-test.php3 (Abgerufen am 20.06.2020)

A.2 Trainierte Modelle für Objekt-Detektion

- Tiny YOLO v2 Konfiguration `https://github.com/pjreddie/darknet/blob/master/cfg/yolov2-tiny.cfg` (Abgerufen am 25.06.2020)
- Tiny YOLO v2 Gewichte `https://pjreddie.com/media/files/yolov2-tiny.weights` (Abgerufen am 25.06.2020)
- Tiny YOLO v3 Konfiguration `https://github.com/pjreddie/darknet/blob/master/cfg/yolov3-tiny.cfg` (Abgerufen am 25.06.2020)
- Tiny YOLO v3 Gewichte `https://pjreddie.com/media/files/yolov3-tiny.weights` (Abgerufen am 25.06.2020)
- SSD MobileNet v1 Modell `https://storage.googleapis.com/download.tensorflow.org/models/tflite/coco_ssd_mobilenet_v1_1.0_quant_2018_06_29.zip` (Abgerufen am 25.06.2020)
- SSD MobileNet v3 (small) Modell `https://github.com/tensorflow/models/files/3819458/ssd_mobilenet_v3_coco.zip` (Abgerufen am 25.06.2020)

A.3 Verwendete Bibliotheken

- ARCore (`com.google.ar:core:1.15.0`)
- Sceneform (`com.google.ar.sceneform.ux:sceneform-ux:1.15.0`)
- OpenCV (`com.quickbirdstudios:opencv:4.1.0-contrib`)
- TensorFlow Lite (`org.tensorflow:tensorflow-lite:2.1.0`)
- RxJava (`io.reactivex.rxjava3:rxjava:3.0.0`)
- RxAndroid (`io.reactivex.rxjava3:rxandroid:3.0.0`)
- Dagger2 (`com.google.dagger:dagger:2.27`, `com.google.dagger:dagger-compiler:2.27`)
- Munkres-Algorithmus (`com.github.kevinstern:software-and-algorithms:1.0`)

Glossar

Activity Basisobjekt der Benutzeroberfläche eines einzelnen Anwendungsfalls in Android.

API (Application Programming Interface) Programmierschnittstelle.

App (Application) Umgangssprachliche Abkürzung Anwendungssoftware, meist im Bezug auf Smartphones.

Callback Funktion, welche als Argument übergeben und nach einem bestimmten Ereignis aufgerufen wird.

Cloud Transparente, verteilte Recheninfrastruktur.

CPU (Central Processing Unit) Hauptprozessor eines Computers.

Fragment Objekt welches einen eigenständigen Teil der Benutzeroberfläche einer Android-Anwendung enthält.

Framework Bibliotheken und Werkzeuge die den Rahmen für die Entwicklung von Anwendungssoftware darstellen.

Gegendruck Phänomen der Anstauung von Objekten bei einem Konsumenten mit längerer Verarbeitungszeit als der zugehörige Produzent.

GPS (Global Positioning System) Satelliten-basiertes, globales System zur Positionsbestimmung.

GPU (Graphics Processing Unit) Grafikprozessor, der vor allem für hoch-parallele Fließkommaoperationen optimiert ist.

IDE (Integrated Development Environment) Anwendung zur Unterstützung bei der Entwicklung. Sie besteht meist aus einem Editor sowie Werkzeugen zur Erstellung der entwickelten Software und Fehlerbehebung.

JVM (Java Virtual Machine) Java-Laufzeitumgebung zur Ausführung von Java-Anwendungen in Bytecode-Form.

Listener Schnittstelle zur Verarbeitung von Ereignissen.

Mikromobilität Kategorie kleiner, vergleichsweise langsamer Verkehrsteilnehmer mit häufig begrenzter Reichweite wie beispielsweise E-Scooter oder Fahrräder.

Sample Abgetasteter Wert eines analogen Signals.

SDK (Software Development Kit) Sammlung von Werkzeugen für die Entwicklung von Software für eine bestimmte Plattform. Sie enthält meist einen Compiler sowie Standard-Bibliotheken.

Stub Platzhalter-Implementierung ohne Funktionalität, welche allerdings Konform zur jeweiligen Schnittstelle ist.

UML (Unified Modeling Language) Standardisierte (ISO/IEC 19501:2005) Modellierungssprache zur Dokumentation, Visualisierung und Spezifizierung von Software.

Erklärung zur selbstständigen Bearbeitung einer Abschlussarbeit

Gemäß der Allgemeinen Prüfungs- und Studienordnung ist zusammen mit der Abschlussarbeit eine schriftliche Erklärung abzugeben, in der der Studierende bestätigt, dass die Abschlussarbeit „— bei einer Gruppenarbeit die entsprechend gekennzeichneten Teile der Arbeit [(§ 18 Abs. 1 APSO-TI-BM bzw. § 21 Abs. 1 APSO-INGI)] — ohne fremde Hilfe selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel benutzt wurden. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich zu machen."

Quelle: § 16 Abs. 5 APSO-TI-BM bzw. § 15 Abs. 6 APSO-INGI

Erklärung zur selbstständigen Bearbeitung der Arbeit

Hiermit versichere ich,

Name: _____

Vorname: _____

dass ich die vorliegende Bachelorarbeit – bzw. bei einer Gruppenarbeit die entsprechend gekennzeichneten Teile der Arbeit – mit dem Thema:

Outdoor Objekt-Detektion und -Tracking von Verkehrsteilnehmern für Augmented Reality

ohne fremde Hilfe selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

_____	_____	_____
Ort	Datum	Unterschrift im Original