HAW HAMBURG

BACHELOR THESIS
Hossein Banay

Ein Digitaler Zwilling in Unity für eine 1:87-Modellwelt mithilfe von maschinellen Lernverfahren

FAKULTÄT TECHNIK UND INFORMATIK
Department Informatik

Faculty of Engineering and Computer Science
Department Computer Science

HOCHSCHULE FÜR ANGEWANDTE WISSENSCHAFTEN HAMBURG
Hamburg University of Applied Sciences

Hossein Banay

Ein Digitaler Zwilling in Unity für eine 1:87-Modellwelt mithilfe von maschinellen Lernverfahren

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

Betreuender Prüfer: Prof. Dr. Stephan Pareigis
Zweitgutachter: Prof. Dr. Tim Tiedemann

Eingereicht am: 17.08.2023

Hossein Banay

Thema der Arbeit

Ein Digitaler Zwilling in Unity für eine 1:87-Modellwelt mithilfe von maschinellen Lernverfahren

Stichworte

Digitaler Zwilling, Unity, Überwachtes Lernen, neuronales Netz, Objekterkennung

Kurzzusammenfassung

Digitale Zwillinge sind virtuelle Repräsentationen realer Systeme, Objekte oder Umgebungen, die dazu dienen, eine Verbindung zwischen der physischen und der digitalen Welt herzustellen. Dadurch können Simulationen und Analysen durchgeführt werden, um das Verhalten des realen Gegenstücks vorherzusagen, zu optimieren oder zu verbessern, ohne die physischen Komponenten zu beeinträchtigen. Diese Arbeit befasst sich mit der Realisierung eines digitalen Zwillings einer Modellwelt in Unity, wobei Gebäude und Fahrzeuge dargestellt werden. Durch die Integration der Objekterkennung mit Hilfe von YOLOv8 können Gebäude und Fahrzeuge erkannt und im digitalen Zwilling dargestellt werden. Die Arbeit umfasst den Aufbau eines geeigneten Datensatzes, das Training und die Aufbereitung der Objekterkennung sowie die Implementierung in Unity. Die Ergebnisse zeigen, dass der digitale Zwilling in der Lage ist, Gebäude und Fahrzeuge mit der richtigen Farbe und Ausrichtung in der realen Welt korrekt zu erkennen und in der virtuellen Umgebung darzustellen.

Hossein Banay

Title of Thesis

A digital twin in Unity for a 1:87 model world using machine learning techniques

Keywords

Digital twin, unity, supervised learning, neural network, object detection

Abstract

Digital twins are virtual representations of real systems, objects or environments that serve to create a link between the physical and digital worlds. This allows simulations and analysis to be performed to predict, optimize, or improve the behavior of the real-world counterpart without affecting the physical components. This work addresses the realization of a digital twin of a model world in Unity, representing buildings and vehicles. By integrating object detection using YOLOv8, buildings and vehicles can be recognized and represented in the digital twin. The work includes the construction of a suitable dataset, the training and preparation of the object detection, and the implementation in Unity. The results show that the digital twin is able to correctly detect buildings and vehicles with the correct color and orientation in the real world and display them in the virtual environment.

Inhaltsverzeichnis

Abbildungsverzeichnis **vii**

Tabellenverzeichnis **ix**

1 Einleitung **1**

1.1 Aufgabenstellung 1

1.2 Verwandte Arbeiten 4

1.2.1 Digitaler Zwilling für einen Multirotor 4

1.2.2 Digitaler Zwilling für vernetzte und automatisierte Fahrzeuge . . . 4

1.2.3 Ein digitaler Zwilling eines Roboterarms 5

2 Grundlegende Technologien **6**

2.1 Evaluierungsmetriken 6

2.1.1 Intersecion over Union 6

2.1.2 Precision 6

2.1.3 Recall 7

2.1.4 Confusion Matrix 7

2.1.5 PR-Curve 7

2.1.6 Mean Average Precesion 8

2.1.7 Loss Funktionen 8

2.2 Frameworks 8

2.2.1 Unity 8

2.2.2 Unity Scripting-API 12

2.2.3 YOLOv8 14

3 Implementation **17**

3.1 Datensatz 17

3.1.1 Aufbau der Bilder 17

3.1.2 Data Augmentation 18

3.2 Image Labeling 20
3.2.1 YOLO Format 21
3.3 Training 23
3.3.1 Vergleich der YOLOv8 Modelle 24
3.3.2 Auswertung der YOLOv8 Modelle 26
3.4 Aufbereitung zur Verarbeitung in Unity 26
3.4.1 JSON Format 26
3.4.2 Farberkennung 28
3.4.3 Konturerkennung 30
3.4.4 Ablaufdiagramm 36
3.5 Unity 38
3.5.1 IPlaceableObject 39
3.5.2 Client 39
3.5.3 PlaceableManager 40
3.5.4 PlaceableObjectInfo 42
3.5.5 MappedIPlaceableObject 43

4 Evaluation **47**
4.1 Objekterkennung 47
4.2 Darstellung in Unity 51

5 Reinforcement Learning im digitalen Zwilling **54**
5.1 Ray Perception Sensor 3D 55
5.2 Behavior Parameters 57
5.3 Agent 58
5.4 Training 59

6 Fazit **61**
6.0.1 Ausblick 62

Literaturverzeichnis **64**

Glossar **67**
Selbstständigkeitserklärung 69

Abbildungsverzeichnis

1.1 Bild der Modellwelt der Forschungsgruppe Autosys 2

2.1 Eine Beispielszene in Unity 9
2.2 Ein GameObject in Unity 10
2.3 Die Transform-Komponente eines GameObjects 10
2.4 Die Mesh Renderer-Komponente eines GameObjects 11
2.5 Die visuelle Darstellung der Collider Komponente eines GameObjects . . . 12
2.6 Modelle der YOLOv8 14
2.7 Beispielbild von einer Objekterkennung 14
2.8 YOLOv8 Architektur 15

3.1 Beispielbilder von der Modellwelt 18
3.2 Beispielbild vom unteren Bereich der Modellwelt und dessen augmentierte Zwillinge 20
3.3 Illustriertes Beispiel für die Erkennung von Objekten in der Modellwelt . . 21
3.4 Beispielbild – YOLO Format 22
3.5 Ein Beispiel für ein Bild und seine Bounding Boxes 22
3.6 Die *.txt Datei für das obere Beispielbild 23
3.7 Liste der YOLOv8 Detect Modelle 23
3.8 mAP Vergleich der YOLOv8 Modelle 24
3.9 Vergleich der Anzahl der Parameter der YOLOv8 Modelle 25
3.10 Vergleich der Geschwindigkeit der YOLOv8 Modelle 25
3.11 JSON-Beispiele zur Verarbeitung in Unity 27
3.12 Beispielbild der Modellwelt und Objektausschnitt 28
3.13 Die Methode *get_dominant_colors()* 29
3.14 Das Ergebnis der k-means Algorithmus 30
3.15 Visuelle Darstellung der *MinAreaRect()* Methode 31
3.16 Drehung des Rechtecks nach rechts 32
3.17 Visuelle Darstellung der *get_color_mask_and_contour()* Methode 34

3.18 Ergebnis der Methode *get_ color_ mask_ and_ contour()* 35
3.19 Beispielbild von einem Fahrzeug und die Maske 35
3.20 Visuelle Darstellung der Auswirkung von *max_ contour* Parameter 36
3.21 Ablaufdiagramm für die in diesem Kapitel beschriebenen Schritte 37
3.22 Die Architektur des digitalen Zwillings in Unity als Klassendiagramm . . . 38
3.23 *PlaceableManager* Komponente 40
3.24 Diverse 3D-Modelle, die vor der Laufzeit in der Szene platziert werden . . 43
3.25 *PlaceableObjectInfo* vs *MappedPlaceableObject* 44
3.26 Darstellung des Beispielbilds im digitalen Zwilling 45
3.27 Beispielbild mit der Objekterkennung 46

4.1 Die Ergebnisse nach dem Training von YOLOv8s 48
4.2 Normalisierte Konfusionsmatrix des trainierten Modells 49
4.3 PR-Kurve des trainierten Modells 50
4.4 Anzahl der Instanzen im Datensatz 50
4.5 Beispiel für ein Fahrzeug mit 100 Punkte 51

5.1 Visuelle Darstellung der RL 54
5.2 *Ray Perception Sensor 3D* Komponente 55
5.3 Visuelle Darstellung der *Ray Perception Sensor 3D* Komponente 56
5.4 Die *Behavior Parameters* Komponente 57
5.5 Agenten beim Training im digitalen Zwilling 60

Tabellenverzeichnis

4.1 Ergebnisse für Videos aus dem **oberen Bereich** der Modellwelt 52
4.2 Ergebnisse für Videos aus dem **unteren Bereich** der Modellwelt 52

1 Einleitung

Heutzutage gewinnt der Einsatz von maschinellem Lernen in einer Vielzahl von Anwendungsbereichen zunehmend an Bedeutung. Diese fortschrittlichen Algorithmen ermöglichen es, spezifische Aufgaben zu bewältigen und die Arbeitseffizienz erheblich zu steigern [16] [17] [9] [1]. Große Teile der Gesellschaft sind bereits mit dieser Technologie vertraut und von ihren Möglichkeiten begeistert.

Der Begriff „Digitaler Zwilling“ ist für viele Menschen nicht auf den ersten Blick verständlich. Ein digitaler Zwilling ist im Wesentlichen ein virtuelles Abbild eines realen Systems, Objekts oder einer Umgebung. Er schlägt eine Brücke zwischen der physischen und der digitalen Welt. Digitale Zwillinge bieten eine Reihe von Vorteilen, darunter die Möglichkeit, das reale System in einer digitalen Umgebung zu analysieren und zu bewerten, ohne direkt mit dem physischen System interagieren zu müssen. Diese Technologie eröffnet neue Horizonte für die Forschung, Entwicklung und Optimierung komplexer Systeme. Digitale Zwillinge werden in verschiedenen Bereichen eingesetzt, von der Industrie und dem Transportwesen bis hin zur Medizin und Stadtplanung. Sie bieten eine effiziente Möglichkeit, Daten zu sammeln, zu analysieren und Handlungsstrategien zur Bewältigung komplexer Herausforderungen zu entwickeln.

1.1 Aufgabenstellung

Ziel dieses Projektes ist es, einen digitalen Zwilling einer realen Modellwelt zu erstellen. Die Modellwelt wird von der Forschungsgruppe Austosys der HAW Hamburg betrieben und umfasst eine Vielzahl von Elementen wie eine Rennstrecke, verschiedene Gebäudetypen, Fahrzeuge und weitere Objekte [18]. Um diese Modellwelt digital abzubilden, werden zwei Raspberry Pi’s über der realen Modellwelt positioniert, die jeweils mit integrierten Kameras ausgestattet sind. Mit diesen Kameras werden Videos der gesamten Modellwelt aufgenommen, die dann als Grundlage für den digitalen Zwilling dienen. In

diesem Projekt liegt der Schwerpunkt auf der Erfassung und Darstellung von Gebäuden und Fahrzeugen im digitalen Zwilling.

Abbildung 1.1: Bild der Modellwelt der Forschungsgruppe Autosys

Im ersten Schritt wird ein Objekterkennungsmodell trainiert, das in der Lage ist, Gebäude und Fahrzeuge in den Bildern oder Videos der Modellwelt zu erkennen. Dazu werden die Kameras der beiden Raspberry Pi's verwendet, um Daten zu sammeln, die für das Training verwendet werden. Im nächsten Schritt wird der digitale Zwilling in Unity implementiert. Die gewonnenen Daten aus der Objekterkennung dienen als Grundlage, um die erkannten Gebäude und Fahrzeuge im digitalen Zwilling zu platzieren. Der digitale Zwilling wird verschiedene Informationen zu den Objekten der Modellwelt darstellen:

1. **Position des Objekts**: Die erkannten Gebäude und Fahrzeuge werden entsprechend ihrer realen Position in der Modellwelt im digitalen Zwilling platziert. So entsteht ein räumlich genaues Abbild der Modellwelt.

2. **Farbe des Objekts**: Die Farbinformationen der erkannten Objekte werden im digitalen Zwilling berücksichtigt, um eine möglichst realitätsnahe Darstellung zu gewährleisten. So werden Gebäude und Fahrzeuge in ihren Originalfarben wiedergegeben.

3. **Ausrichtung des Objekts**: Die erkannten Orientierungen der Gebäude und Fahrzeuge werden in den digitalen Zwilling übernommen, um sicherzustellen, dass die Objekte in der richtigen Ausrichtung positioniert sind.

Darüber hinaus muss der digitale Zwilling über die Funktionalität verfügen, den Zustand der Objekte zu aktualisieren. Das heißt, wenn z.B. ein Gebäude aus dem Sichtfeld einer Kamera verschwindet und somit nicht mehr erkannt wird, wird dieses Gebäude auch nicht mehr im digitalen Zwilling angezeigt. Auf diese Weise bleibt der digitale Zwilling stets aktuell und repräsentiert die sich verändernde Szenerie der realen Modellwelt. Dieser Ansatz ermöglicht ein dynamisches und immersives Erleben des digitalen Zwillings, da er die Veränderungen der Modellwelt in der simulierten Szene widerspiegelt und somit eine lebendige und realistische Simulation erzeugt.

1.2 Verwandte Arbeiten

1.2.1 Digitaler Zwilling für einen Multirotor

Forscher der Guangdong University of Technology in China [21] haben einen digitalen Zwilling für ein Multirotor-UAV (Unmanned Aerial Vehicle) entwickelt. Diese virtuelle Plattform nutzt verschiedene Technologien wie Unity, ROS (Robot Operating System), Matlab und SimulIDE, um den gesamten Lebenszyklus eines Multirotor-UAVs zu verfolgen. Ein wichtiger Aspekt dieses digitalen Zwillings ist die mehrdimensionale Simulation der Umgebung und der physischen Objekte. Das bedeutet, dass nicht nur das UAV selbst simuliert wird, sondern auch die Umgebung, in der es operiert. Dazu gehören Aspekte wie Lichtverhältnisse, Geländebeschaffenheit und Wetterbedingungen, die in die digitale Umgebung integriert werden. Diese realitätsnahe Simulation ermöglicht es den Forschern, verschiedene Szenarien zu testen und das Verhalten des UAV unter unterschiedlichen Bedingungen zu analysieren. Insgesamt leistet die Arbeit der Guangdong University of Technology einen wichtigen Beitrag zur Anwendung von digitalen Zwillingen in der Drohnenindustrie. Die Entwicklung einer solchen Plattform eröffnet neue Möglichkeiten für zukünftige Forschung und Entwicklung in diesem Bereich.

1.2.2 Digitaler Zwilling für vernetzte und automatisierte Fahrzeuge

Ziran Wang, Kyungtae Han und Prashant Tiwari von Toyota-Motor North America [20] haben einen digitalen Zwilling für vernetzte und automatisierte Fahrzeuge (CAVs) in Unity entwickelt. Die Autoren schlagen eine Architektur für einen digitalen Zwilling vor, der aus der physischen und der digitalen Welt besteht. Insbesondere besteht die digitale Welt aus drei Schichten, in denen Unity GameObjects verwendet werden, um die Hardware zu simulieren, Unity Scripting API verwendet wird, um die Software zu simulieren und externe Tools wie SUMO, MATLAB, Python und AWS verwendet werden, um die Simulation zu verbessern. Die Effektivität des vorgeschlagenen digitalen Zwillings wird anhand einer Fallstudie zur adaptiven Geschwindigkeitsregelung (P-ACC) demonstriert. Dabei werden verschiedene Aspekte der Fahrzeugumgebung und der Fahrzeugsteuerung berücksichtigt, um die Leistungsfähigkeit des P-ACC-Systems zu evaluieren. In der digitalen Umgebung werden Straßennetzwerke einschließlich Straßengeometrie, Straßenneigung, Straßentyp, Straßenbegrenzungen, Fahrbahnmarkierungen und Verkehrszeichen (z.B. Stop, Geschwindigkeit) berücksichtigt. Bei den Fahrzeugen in der physischen Welt

werden vier verschiedene Fahrzeugtypen berücksichtigt: konventionelle Fahrzeuge (ohne Konnektivität oder Automatisierung), vernetzte Fahrzeuge (ohne Automatisierung), automatisierte Fahrzeuge (ohne Konnektivität) und CAVs. Insgesamt stellt die Entwicklung dieses digitalen Zwillings für vernetzte und automatisierte Fahrzeuge einen bedeutenden Fortschritt dar und bietet eine wertvolle Plattform für zukünftige Forschung und Entwicklung im Bereich der autonomen Fahrzeuge und des Straßenverkehrs.

1.2.3 Ein digitaler Zwilling eines Roboterarms

Forschern der Universität Birmingham in Großbritannien [15] ist es gelungen, einen digitalen Zwilling eines Roboterarms in Unity zu entwickeln, der als Trainingsplattform in einem virtuellen Raum dient. Dieser digitale Zwilling ermöglicht es, den Roboterarm in einer simulierten Umgebung zu trainieren und dabei wertvolle Erkenntnisse zu gewinnen. Die Ergebnisse des Trainings mit dem digitalen Zwilling werden dann auf den realen Roboterarm übertragen, um dessen Fähigkeiten und Leistung zu verbessern. Das Training des digitalen Roboterarms erfolgt mittels Reinforcement Learning. Dabei werden Sensordaten und physikalische Eigenschaften aus der realen Welt übertragen, um das Training in der Unity Umgebung durchzuführen. Die Kombination aus dem digitalen Zwilling und der Übertragung der Ergebnisse auf den realen Roboterarm zeigt das Potenzial der Forschung auf dem Gebiet der Robotik und der Nutzung virtueller Umgebungen für das Training und die Optimierung von Robotersystemen.

2 Grundlegende Technologien

2.1 Evaluierungsmetriken

In dieser Arbeit werden unterschiedliche Evaluierungsmetriken verwendet, um die Effizienz des Objekterkennungsmodells zu bewerten. Im Folgenden werden die verwendeten Evaluierungsmetriken im Detail beschrieben.

2.1.1 Intersecion over Union

Die Intersection over Union (IoU) misst das Verhältnis der Schnittfläche (Intersection) zwischen einer vorhergesagten Bounding Box und der tatsächlichen Bounding Box eines Objekts zur Vereinigungsfläche (Union) beider Flächen. Die IoU wird berechnet, indem die Überlappungsfläche der Bounding Box des vorhergesagten Objekts und des tatsächlichen Objekts durch die Vereinigung der beiden Flächen dividiert wird. Ein perfektes Ergebnis wäre ein IoU von 1, wenn die Bounding Box genau mit dem Objekt übereinstimmt. Ein geringerer Wert weist auf eine geringere Genauigkeit der Vorhersage hin.

2.1.2 Precision

Die Precision misst den Anteil der korrekt als positiv klassifizierten Instanzen an der Gesamtzahl der als positiv klassifizierten Instanzen. Die Formel zur Berechnung der Precision lautet:

$$Precision = \frac{\text{True Positives}}{\text{True Positives} + \text{False Positives}}$$

Die Precision gibt an, wie gut das Modell eine positive Instanz vorhersagt.

2.1.3 Recall

Der Recall misst das Verhältnis der korrekt als positiv klassifizierten Instanzen zur Gesamtzahl der tatsächlich positiven Instanzen. Die Formel zur Berechnung des Recalls lautet:

$$Recall = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}}$$

Der Recall gibt an, wie viele der tatsächlich positiven Instanzen vom Modell richtig erkannt wurden.

2.1.4 Confusion Matrix

Die Confusion Matrix ist eine Tabelle, die zur Bewertung der Leistung eines Klassifikationsmodells verwendet wird. Sie zeigt die Anzahl der richtigen und falschen Vorhersagen des Modells in Bezug auf die tatsächlichen Klassen der Daten [14].

Die Confusion Matrix enthält folgende Einträge:

1. **True Positives (TP)**: Die Anzahl der Instanzen, die vom Modell korrekt als positive Klasse vorhergesagt wurden.
2. **True Negatives (TN)**: Die Anzahl der Instanzen, die vom Modell korrekt als negative Klasse vorhergesagt wurden.
3. **False Positives (FP)**: Die Anzahl der Instanzen, die vom Modell fälschlicherweise als positive Klasse vorhergesagt wurden, obwohl sie eigentlich zur negativen Klasse gehören.
4. **False Negatives (FN)**: Die Anzahl der Instanzen, die vom Modell fälschlicherweise als negative Klasse vorhergesagt wurden, obwohl sie eigentlich zur positiven Klasse gehören.

2.1.5 PR-Curve

Die PR-Kurve (Precision-Recall-Curve) ist ein Diagramm, das die Beziehung zwischen Precision und Recall für verschiedene Schwellenwerte darstellt. Sie wird vor allem zur Bewertung von Klassifikationsmodellen verwendet, bei denen das Verhältnis zwischen

den Klassen im Datensatz stark unausgewogen ist. Je größer die Fläche unter der Kurve ist, desto wahrscheinlicher ist es, dass das Modell eine hohe Genauigkeit (niedrige FP-Rate) und einen hohen Recall (niedrige FN-Rate) aufweist [3].

2.1.6 Mean Average Precesion

Die Average Precision (AP) für eine Klasse wird berechnet, indem die PR-Kurve für diese Klasse erzeugt wird. Dabei wird die Fläche unter der Precision-Recall-Kurve berechnet, die die durchschnittliche Präzision für verschiedene Recall-Werte darstellt. Die AP ist von dem IoU-Schwellenwert abhängig. Je niedriger der Schwellenwert, desto höher ist die Precision.

Die Mean Average Precision (mAP) wird berechnet, indem die *average precesions* aller Klassen gemittelt werden. Sie gibt somit eine zusammenfassende Bewertung der Leistung des Modells über alle Klassen. Ein hoher mAP weist auf ein Modell hin, das sowohl eine hohe Genauigkeit als auch einen hohen Recall über die Klassen hinweg aufweist.

2.1.7 Loss Funktionen

Die Loss Funktion ist eine mathematische Funktion, die den Fehler zwischen den vorhergesagten Werten und den tatsächlichen Werten ermittelt. Sie wird zur Optimierung des Modells während des Trainings verwendet. Je größer der Loss Wert ist, desto größer sind die Gewichtsveränderungen der Neuronen in einem neuronalen Netz.

2.2 Frameworks

2.2.1 Unity

Unity ist eine Entwicklungsumgebung für die Entwicklung von 2D- und 3D-Spielen für PC und Spielkonsolen. Es bietet die Möglichkeit, 2D- und 3D-Modelle zu erstellen oder aus verschiedenen Quellen zu importieren, zu manipulieren und nach Belieben anzupassen. Die Modelle können verschiedene Texturen und Materialien sowie andere für die Modelle notwendige Komponenten enthalten. Auch die Simulation von Lichtverhältnissen wird unterstützt.

Szene

Eine Szene beschreibt in Unity, eine Umgebung, in der sich die unterschiedliche 2D/3D Objekte befinden. Beim Erstellen einer neuen Szene besteht die Möglichkeit zwischen einer 2D bzw. 3D Szene zu wählen. In einer 3D-Szene können mehrere 3D-Objekte platziert werden. Die Objekte innerhalb der Szene dürfen an unterschiedliche Koordinaten positioniert werden.

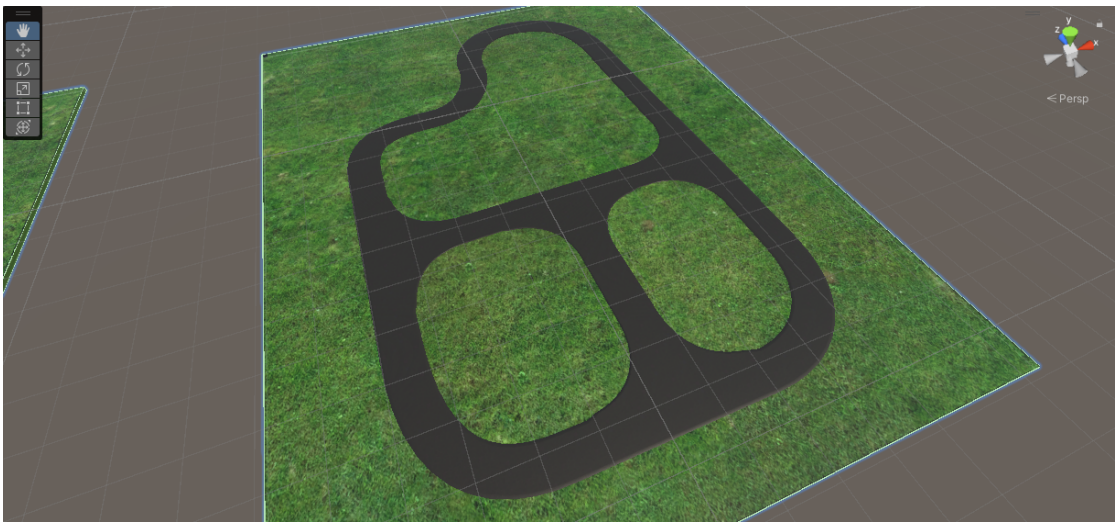


Abbildung 2.1: Eine Beispielszene in Unity

Die obige Abbildung 2.1 zeigt eine 3D-Szene in Unity, in der sich mehrere Objekte befinden. Die dargestellte Szene zeigt eine Strecke mit einem Untergrund aus Grasmaterial.

GameObject

Die Objekte innerhalb der Unity Umgebung werden als „GameObject“ bezeichnet. Diese können sowohl 3D- als auch 2D-Grafiken sein. Ein GameObject kann wiederum mehrere GameObjects enthalten. Mehrere GameObjects können zu einem GameObject zusammengeführt werden.

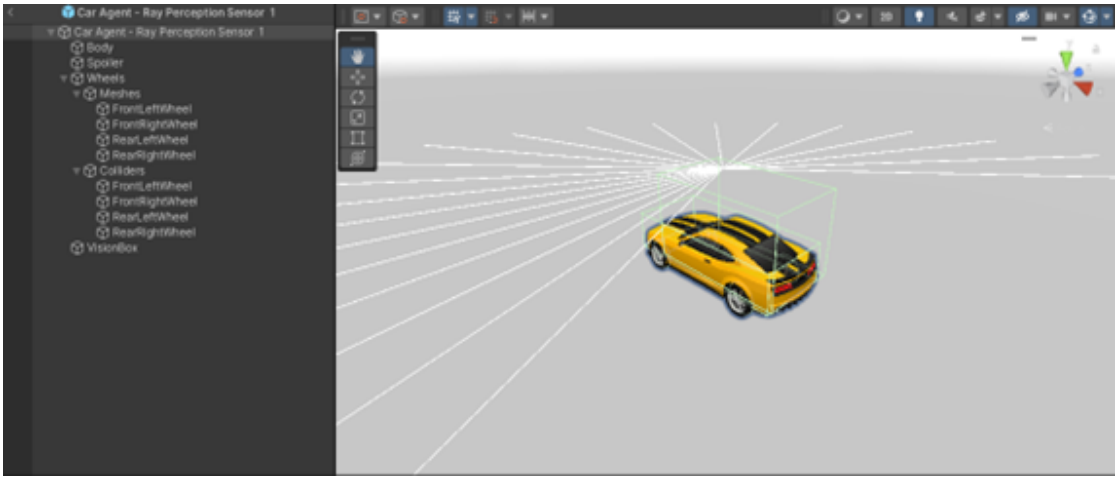


Abbildung 2.2: Ein GameObject in Unity

In der Abbildung 2.2 ist ein GameObject zu erkennen. Der Name des GameObjects steht links in der Hierarchie-Ansicht (Car Agent – Ray Perception Sensor 1). Dieses GameObject besteht aus mehreren GameObjects und besitzt jeweils einen *Body*, *Spoiler*, *Wheels*, *Colliders* sowie eine *Visionbox*.

Komponenten

Jedes GameObject kann mehrere Komponenten enthalten. Die Komponenten beschreiben den Zustand oder das Verhalten eines GameObjects. Einige Komponenten können ausgeblendet werden und verlieren somit ihrer Funktionalität. Außerdem können Komponenten eines GameObjects in einem Skript manipuliert werden.

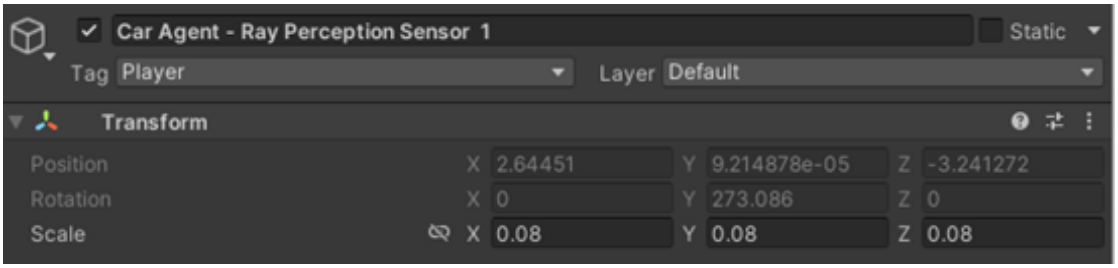


Abbildung 2.3: Die Transform-Komponente eines GameObjects

In der obigen Abbildung 2.3 ist die Transform-Komponente von *Car Agent – Ray Perception Sensor 1* dargestellt. Diese Komponente beschreibt den Zustand des GameObjects

innerhalb der Szene. Der Parameter *Position* beschreibt die Koordinaten des GameObjects in der Szene, der Parameter *Rotation* die Ausrichtung des GameObjects und der Parameter *Scale* die Größe des GameObjects im dreidimensionalen Raum. Diese Parameter können beliebig geändert werden.

Eine weitere Komponente ist der *Mesh Renderer*. Diese Komponente ist für die Darstellung eines 3D-Modells (Mesh) verantwortlich. Sie definiert, wie das Mesh gerendert wird, indem sie die Einstellungen für Materialien, Beleuchtung und Schatten festlegt. Die *Mesh Renderer* Komponente arbeitet eng mit der *Mesh Filter* Komponente zusammen, die das zugrunde liegende Mesh definiert, das gerendert werden soll.

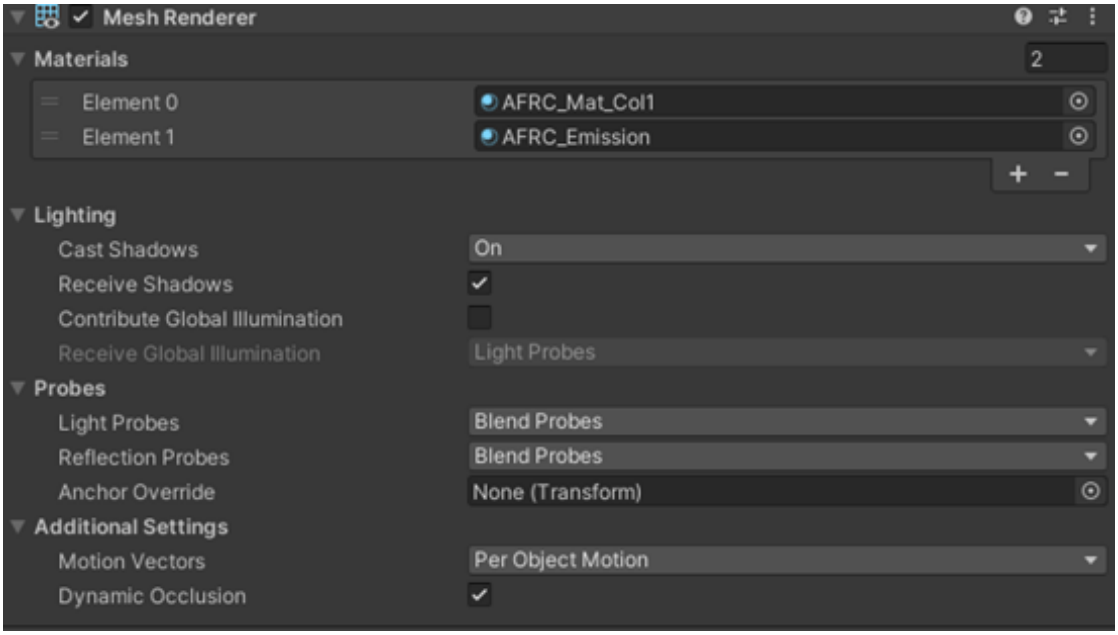
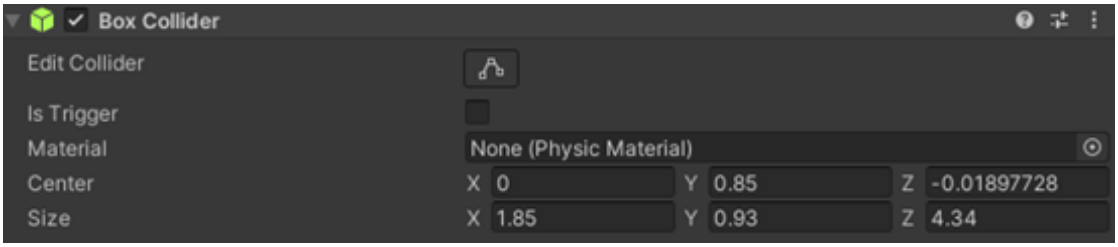


Abbildung 2.4: Die Mesh Renderer-Komponente eines GameObjects

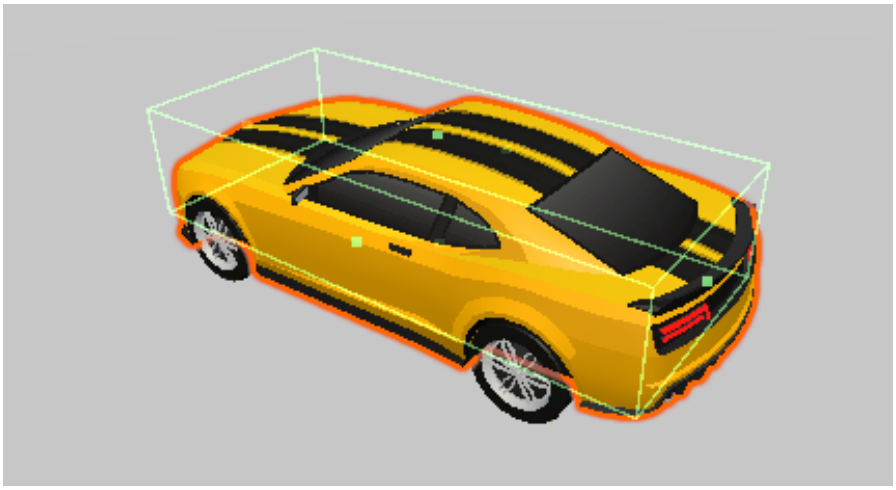
Die Abbildung 2.4 zeigt die *Mesh Renderer* Komponente von *Car Agent – Ray Perception Sensor 1*. Dieser besitzt zwei Materialien, die für die Visualisierung des GameObjects verantwortlich sind. Die restlichen Parameter sind Standardwerte.

Die dritte wichtige Komponente ist eine *Collider* Komponente. Diese Komponente ermöglicht eine kollisionsbasierte Interaktion mit anderen Objekten in der Szene. Der Collider definiert die Grenzen eines Objekts und kann feststellen, wenn ein anderes Objekt mit ihm kollidiert oder seine Grenzen überschreitet. Es gibt diverse Arten von *Collider* Komponenten, wie z. B. *Box Collider*, *Sphere Collider*, *Capsule Collider* oder *Mesh Collider*,

die jeweils unterschiedliche Formen und Kollisionseigenschaften haben. In der folgenden Abbildung 2.5 ist ein *Box Collider* zu erkennen. Die *Edit Collider* Funktion bietet die Möglichkeit, die Größe des Colliders beliebig anzupassen. Die *IsTrigger* Eigenschaft beschreibt die Funktionsweise des Colliders. Ist sie aktiviert, wird die Komponente als Auslöser (Trigger) behandelt und löst keine physische Kollision aus, sondern nur eine Abfrage, ob sich ein anderes Objekt in ihrem Bereich befindet. Die restlichen Parameter definieren die Größe und Position des Colliders.



Der Konfigurator der Collider Komponente



Die grünen Ränder definieren die Grenzen des Colliders

Abbildung 2.5: Die visuelle Darstellung der Collider Komponente eines GameObjects

2.2.2 Unity Scripting-API

Die Unity Scripting API ist eine Sammlung von Funktionen, Klassen und Methoden, die von der Unity Engine zur Verfügung gestellt werden. Die Unity Scripting API ermöglicht Entwicklern den Zugriff auf verschiedene Aspekte einer Szene und ihrer Objekte und unterstützt die Programmiersprache C#.

MonoBehaviour

MonoBehaviour ist eine Basisklasse der Unity Scripting API, von der alle selbstdefinierten Skripte in Unity erben müssen. Skripte, die von MonoBehaviour erben, werden in Unity als Komponenten betrachtet und müssen einem GameObject zugewiesen werden. Dadurch erhalten die Skripte Zugriff auf wichtige Funktionen und Eigenschaften von Unity, mit denen das Verhalten von GameObjects in einer Szene gesteuert und beeinflusst werden kann. Die wichtigsten Funktionen werden im Folgenden detailliert beschrieben:

- *Start()*: Diese Methode wird beim Starten einer Szene einmalig aufgerufen. Darin können beispielsweise Initialisierungen vorgenommen werden, die für das Skript notwendig sind.
- *Update()*: Diese Methode wird einmal pro Frame aufgerufen. Darin können beispielsweise die Position eines GameObjects regelmäßig aktualisiert werden.
- *LateUpdate()*: Diese Methode wird pro Frame erst aufgerufen, nachdem alle anderen *Update()* Methoden innerhalb der Szene aufgerufen wurden.
- *OnTriggerEnter(Collider)*: Diese Methode wird automatisch aufgerufen, wenn das mit diesem Skript verknüpfte GameObject mit einem anderen GameObject kollidiert.

Weitere Informationen zu anderen Funktionen können hier [8] aufgerufen werden.

2.2.3 YOLOv8

YOLOv8 [19] wurde von Ultralytics entwickelt und ist eine Plattform, die verschiedene ML-Modelle anbietet.

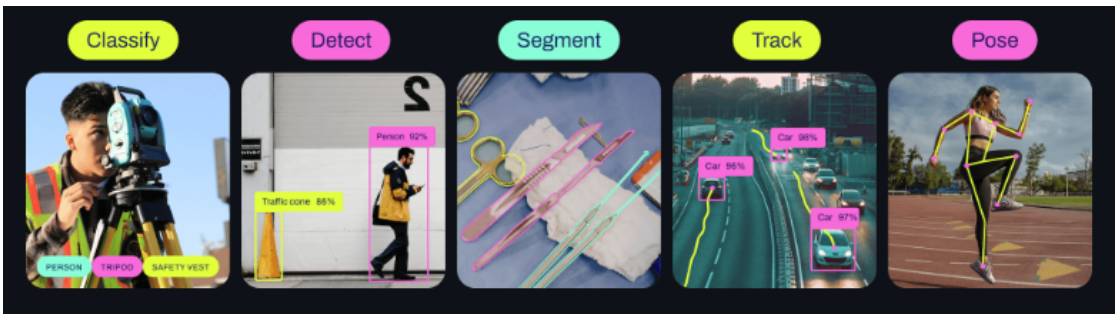


Abbildung 2.6: Modelle der YOLOv8

In dieser Arbeit wird das Detect Model von YOLOv8 verwendet. Beim Detect Model geht es um die Erkennung verschiedener Objekte in einem Bild/Video.

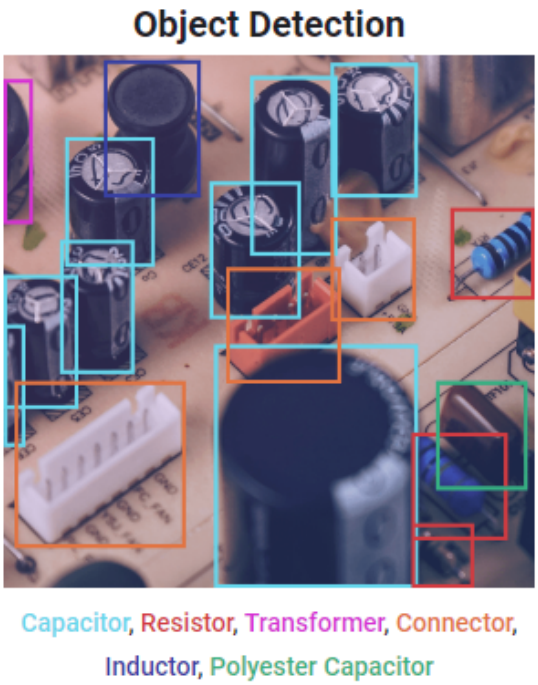


Abbildung 2.7: Beispielbild von einer Objekterkennung

Das Objekterkennungsmodell wird in diesem Projekt verwendet, um Gebäude und Fahrzeuge in der Modellwelt zu erkennen und im digitalen Zwilling darzustellen.

YOLOv8 Architektur

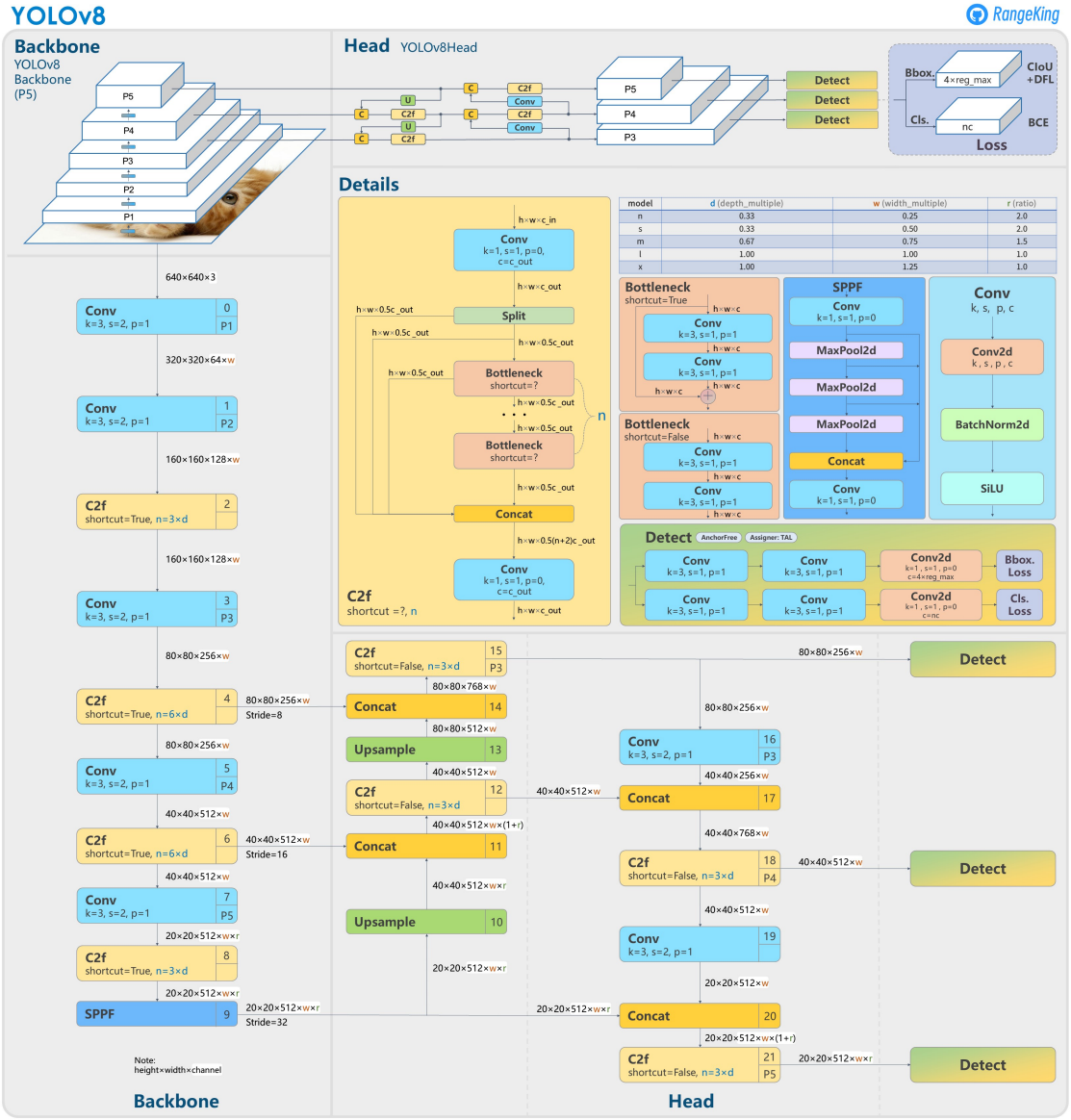


Abbildung 2.8: YOLOv8 Architektur

YOLOv8 besteht aus zwei Hauptkomponenten **Backbone** und **Head**

Die Abbildung 2.8 zeigt die Architektur von YOLOv8. Sie besteht aus zwei Hauptkomponenten, dem Backbone und dem Head. Der Backbone besteht aus einer modifizierten Version der früheren YOLO Modelle [2]. Die Struktur besteht aus einer Kombination verschiedener Convolutional layer, die in Schichten unterteilt sind, um Merkmale aus den Eingabedaten zu extrahieren. Der Head ist für die Erzeugung des Endergebnisses verantwortlich. Das Ergebnis des Heads ist eine Liste der erkannten Objekte, die für jedes erkannte Objekt die Position (Bounding Box Koordinaten) und die zugehörige Klasse enthält.

3 Implementation

In diesem Kapitel wird der verwendete Datensatz für das Training des neuronalen Netzes beschrieben, das für die Erkennung von Objekten in der Modellwelt verantwortlich ist. Es wird erläutert, wie der Datensatz aufbereitet wird und unter welchen Bedingungen das Training stattfindet. Anschließend werden die Schritte beschrieben, die notwendig sind, um die Daten in Unity zu verarbeiten. Im letzten Schritt erfolgt die Implementierung in Unity.

3.1 Datensatz

Zur Erkennung der Objekte in der Modellwelt [18] wird ein neuronales Netz trainiert. Dazu wird das vortrainierte YOLOv8 Detect Model verwendet, das speziell für diesen Anwendungsfall weiter trainiert wird, um die Gebäude und Fahrzeuge in der realen Modellwelt erkennen zu können. Für das Training werden Fotos mit einer Auflösung von 640×480 Pixeln von den beiden über der Modellwelt positionierten Kameras aufgenommen.

3.1.1 Aufbau der Bilder

Der Trainingsdatensatz für das Training besteht aus Bildern von zwei verschiedenen Raspberry Pi's mit integrierten Kamerakomponenten. Die Kameras sind so ausgerichtet, dass jeweils die obere Hälfte sowie die untere Hälfte der Modellwelt erfasst wird.
Insgesamt besteht der Trainingsdatensatz aus 270 Bildern. Der Datensatz wird in zwei Teile geteilt, so dass 80 % (216 Bilder) der Bilder als Trainingsdaten und 20 % (54 Bilder) als Testdaten verwendet werden. Die Anforderungen an den Datensatz sind wie folgt definiert:

1. Jedes Bild im Datensatz erfasst den oberen oder unteren Teil der Modellwelt, um die Vielfalt der darin enthaltenen Objekte und Strukturen darzustellen.
2. Eine Variation der Gebäudegrößen, z.B. kleine, mittlere und große Gebäude, ist im Bild vorhanden.
3. Die Gebäude befinden sich nicht nur in der Mitte des Bildes, sondern auch an den Rändern oder in anderen Positionen, um eine unterschiedliche räumliche Verteilung zu gewährleisten.
4. Auf den Bildern sind Fahrzeuge in verschiedenen Farben dargestellt.
5. Die Fahrzeuge erscheinen nicht nur an einer Stelle im Bild, sondern sind über den gesamten Bildbereich verteilt, um unterschiedliche Positionen und Perspektiven abzudecken.

(a) (b)

Abbildung 3.1: Beispielbilder von der Modellwelt

(a) Oberer Bereich der Modellwelt (b) Unterer Bereich der Modellwelt

3.1.2 Data Augmentation

Bei der Aufbereitung des Datensatzes wird Image Data Augmentation verwendet, um die Anzahl der Bilder und die Größe des Datensatzes zu erhöhen. Dabei werden verschiedene Techniken verwendet, um bestimmte Merkmale aus den Bildern zu entfernen oder hinzuzufügen.

Für die Generierung des augmentierten Datensatzes wird die *Albumentations* Bibliothek [4] verwendet, die eine Vielzahl von Funktionen zur Modifikation der Bilder bietet.

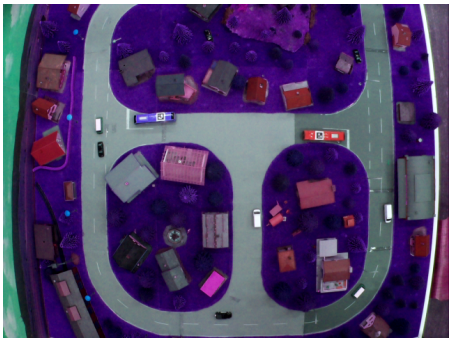
Der ursprüngliche Datensatz ohne die augmentierten Bilder besteht aus insgesamt 30 Bildern, 20 Bilder aus dem oberen Teil der Modellwelt und 10 Bilder aus dem unteren Teil der Modellwelt. Insgesamt wurden 240 neue Bilder durch Data Augmentation erzeugt.

Im Folgenden werden die einzelnen Techniken detaillierter beschrieben:

- Um die Variation und Vielfalt der Daten zu erhöhen, werden 30 Bilder horizontal gespiegelt. Dadurch entstehen neue Ansichten der Modellwelt und es werden unterschiedliche Perspektiven abgedeckt.
- Ähnlich wie beim horizontalen Spiegeln sind auch 30 Bilder vertikal gespiegelt. Dadurch entstehen weitere Variationen und Perspektiven der Modellwelt.
- Um die Robustheit des neuronalen Netzes gegenüber Rauschen zu verbessern, werden 30 Bilder mit zufälligem Rauschen versehen. Dies simuliert reale Störungen oder ungenaue Bildaufnahmen.
- Um die Datenvarianz weiter zu erhöhen, sind bei 30 Bilder eine zufällige Mischung der Farbkanäle durchgeführt worden.
- Weitere 30 Bilder sind um einen zufälligen Winkel (maximal 90 Grad) rotiert worden.
- Um die Leistungsfähigkeit des Modells zu erhöhen, wurden 60 Bilder mehrfach augmentiert. Bei der ersten Methode wurde zunächst ein Bewegungsunschärfe-Effekt mit einer Wahrscheinlichkeit von 70 % auf das Bild angewendet. Anschließend wurde mit einer Wahrscheinlichkeit von 60 % ein Schatten an einer zufälligen Stelle im Bild hinzugefügt. Schließlich wurde eine perspektivische Transformation mit einer Wahrscheinlichkeit von 50 % auf das Bild angewendet. Bei der zweiten Methode wurde zunächst die adaptive Histogramm-Equalisierung mit Kontrastbegrenzung (CLAHE) mit einer Wahrscheinlichkeit von 100 % auf das Bild angewendet. Anschließend wurde das Bild mit einer Wahrscheinlichkeit von 70 % unfokussiert.

(a) Originalbild

(b) Vertikal gespiegelt



(c) Farbkanalmischung

(d) Zufälliges Rauschen

Abbildung 3.2: Beispielbild vom unteren Bereich der Modellwelt und dessen augmentierte Zwillinge

3.2 Image Labeling

Um den Datensatz für das Training verwenden zu können, werden die Bilder im letzten Schritt der Aufbereitung mit dem Programm *LabelImg* manuell beschriftet. Insgesamt werden zwei Labels verwendet:

- ***Building*** als Namensbezeichnung für die Gebäude
- ***Car*** als Namensbezeichnung für die steuerbaren Fahrzeuge

Die untere Abbildung 3.3 zeigt, wie das Modell nach dem Training Objekte auf einem Bild erkennt.

Gebäude in Rot, Fahrzeuge in Rosa

Abbildung 3.3: Illustriertes Beispiel für die Erkennung von Objekten in der Modellwelt

3.2.1 YOLO Format

Da für das Training ein vortrainiertes YOLOv8-Modell verwendet wird, müssen die Labels für die jeweiligen Objekte im YOLO-Format gespeichert werden. Jedes annotierte Bild enthält daher eine *.txt Datei, in der für jedes annotierte Objekt folgende Informationen enthalten sind [5]:

- ***object-class*** – Ein numerischer Wert, der die Objektklasse repräsentiert (z. B. Car/Building).
- ***x-center*** – X-Koordinate des Mittelpunktes des Objektes.
- ***y-center*** – Y-Koordinate des Mittelpunktes des Objektes.
- ***width*** – Die Breite der Bounding Box, die das Objekt umschließt.
- ***height*** – Die Höhe der Bounding Box, die das Objekt umschließt.

```
<object-class> <x-center> <y-center> <width> <height>
```

Jedes Bounding Box enthält diese Informationen

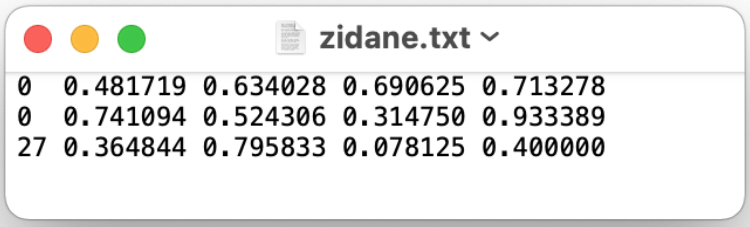
Abbildung 3.4: Beispielbild – YOLO Format

Wie es in der oberen Abbildung 3.4 zu sehen ist, enthält jeder Bounding Box 5 verschiedene Informationen, die für das Erkennen des jeweiligen Objekts relevant sind. Es ist wichtig zu beachten, dass die Werte für *x-center*, *y-center*, *width* und *height* normalisiert sind [11]. Das bedeutet, dass die Werte für *x-center* und *width* durch 640 und die Werte für *y-center* und *height* durch 480 geteilt werden müssen, wenn das Bild eine Auflösung von 640 × 480 Pixeln hat.

Die Abbildung 3.5 zeigt ein visuelles Beispiel der oben beschriebenen Informationen.

Klasse **Person** in Orange, **Krawatte** in Grün

Abbildung 3.5: Ein Beispiel für ein Bild und seine Bounding Boxes



object-class 0 für Person, 27 für die Krawatte

Abbildung 3.6: Die *.txt Datei für das obere Beispielbild

3.3 Training

Das YOLOv8 Detect Modell bietet eine Reihe von unterschiedliche Objekterkennungsmodelle [10].

Model	size (pixels)	mAP^{val} 50-95	Speed CPU ONNX (ms)	Speed A100 TensorRT (ms)	params (M)	FLOPs (B)
YOLOv8n	640	37.3	80.4	0.99	3.2	8.7
YOLOv8s	640	44.9	128.4	1.20	11.2	28.6
YOLOv8m	640	50.2	234.7	1.83	25.9	78.9
YOLOv8l	640	52.9	375.2	2.39	43.7	165.2
YOLOv8x	640	53.9	479.1	3.53	68.2	257.8

Abbildung 3.7: Liste der YOLOv8 Detect Modelle

Die Liste zeigt die Unterschiede der YOLOv8 Modelle in Bezug auf Performanz, Geschwindigkeit und Größe der Parameter.

3.3.1 Vergleich der YOLOv8 Modelle

Die Abbildung 3.7 zeigt alle YOLOv8 Modelle und deren Unterschiede. Um einen besseren Überblick über die Unterschiede zu bekommen, werden drei Diagramme aus der Abbildung erstellt.

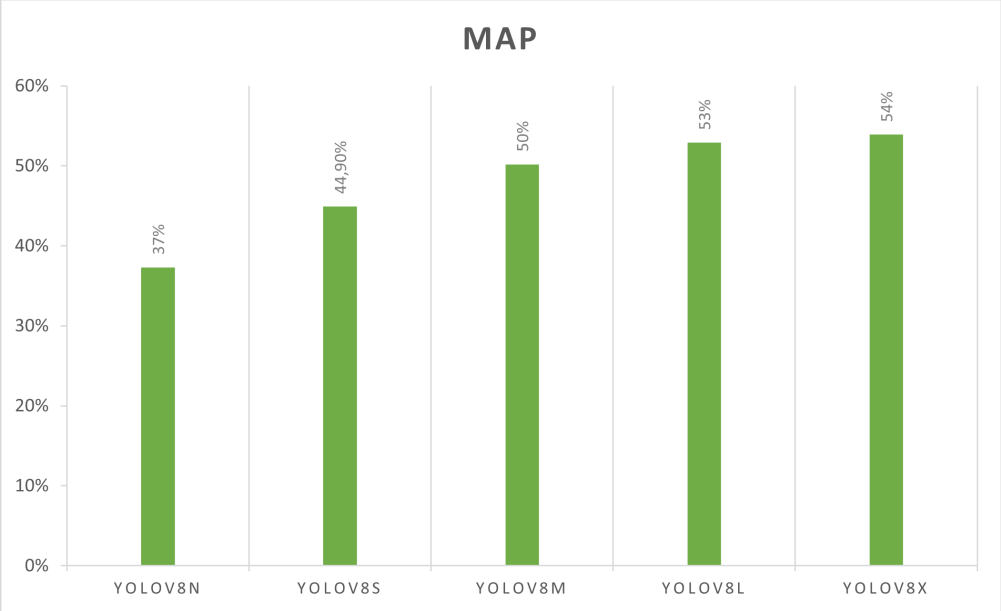


Abbildung 3.8: mAP Vergleich der YOLOv8 Modelle

Das Diagramm zeigt die YOLOv8 Modelle und deren mAP in Prozent.

Wie die Abbildung 3.8 zeigt, erreicht das Modell *YOLOv8s* eine mAP von 44,9 %. Verglichen mit dem *YOLOv8x*-Modell, das den höchsten mAP aufweist, bedeutet dies eine Genauigkeitseinbuße von etwa 16,7 %. Außerdem weist das *YOLOv8n*-Modell im Vergleich zum *YOLOv8x*-Modell eine um 31 % geringere Genauigkeit auf. Die Modelle *YOLOv8m* und *YOLOv8l* weisen im Vergleich zu *YOLOv8x* einen Genauigkeitsunterschied von weniger als 10 % auf.

In der Abbildung 3.9 wird deutlich, dass das *YOLOv8s* über **11,2 Millionen** Parameter verfügt, was etwa 71 % mehr sind als beim *YOLOv8n*. Dagegen hat *YOLOv8m* etwa 57 % mehr Parameter als *YOLOv8s*.

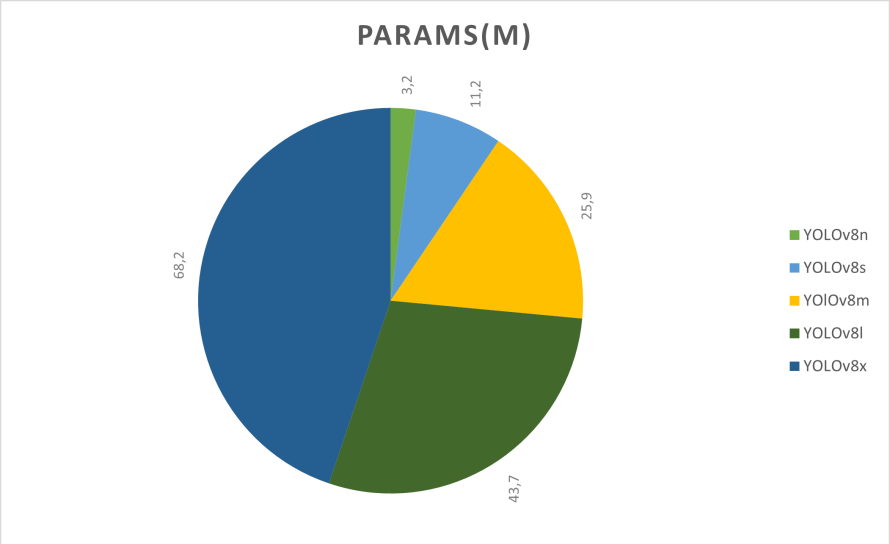


Abbildung 3.9: Vergleich der Anzahl der Parameter der YOLOv8 Modelle

YOLOv8n hat die geringste Anzahl an Parametern, während YOLOv8x die größte Anzahl aufweist.

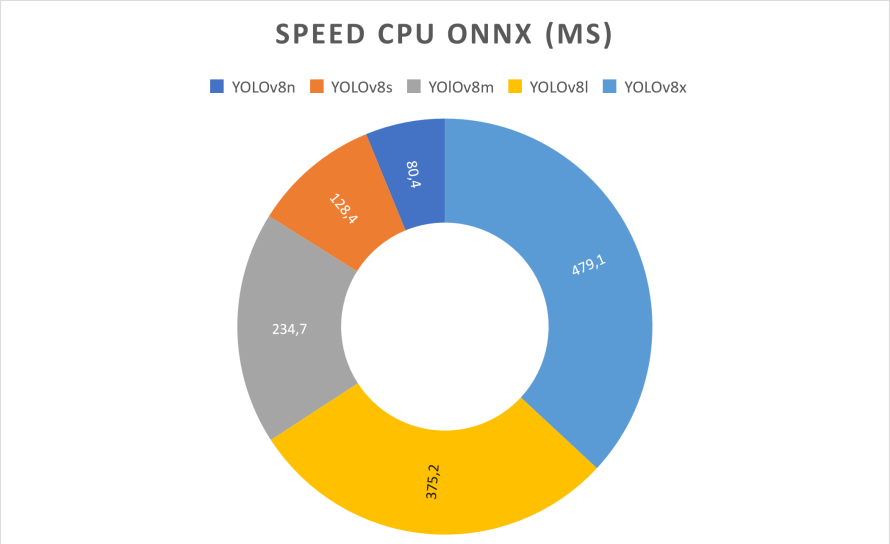


Abbildung 3.10: Vergleich der Geschwindigkeit der YOLOv8 Modelle

Bei der Ausführung benötigt YOLOv8n etwa 80,4 ms, welches im Vergleich zum nächstgrößeren Modell eine Zeitersparnis von 37 % bedeutet.

Der letzte Vergleich in der Abbildung 3.10 zeigt, dass das *YOLOv8n* mit einer Geschwindigkeit von 80,4 ms schneller ist als die anderen YOLOv8-Modelle. Das *YOLOv8s* Modell belegt den zweiten Platz mit einer Geschwindigkeit von 128,4 ms. Dieses Modell benötigt etwa 37 % mehr Zeit für die Ausführung.

3.3.2 Auswertung der YOLOv8 Modelle

Basierend auf den Vergleichen in den Abbildungen 3.8, 3.9 und 3.10 kann argumentiert werden, dass das Modell *YOLOv8s* die optimale Wahl für diesen Anwendungsfall ist. Es erreicht sowohl einen hohen mAP-Wert von 44,9 % als auch eine Ausführungszeit von 128,4 ms, was der zweithöchsten Geschwindigkeit entspricht. Die vergleichsweise geringe Anzahl an Parametern im *YOLOv8s* Modell deutet darauf hin, dass es weniger Ressourcen für die Erkennung von Objekten in der Modellwelt benötigt.

Das Modell *YOLOv8n* ist mit einer Ausführungszeit von 80,4 ms zwar schneller als das Modell *YOLOv8s*, hat aber einen mAP von nur 37 %. Die Modelle *YOLOv8m*, *YOLOv8l* und *YOLOv8x* erreichen einen höheren mAP von 40 %. Allerdings haben sie auch eine große Anzahl von Parametern, was zu einem höheren Ressourcenbedarf führt.

Bei der Auswahl eines Modells für diesen Anwendungsfall ist es wichtig, eine gute Balance zwischen hoher Genauigkeit und geringem Ressourcenverbrauch zu finden. Das *YOLOv8s* erfüllt diese Anforderungen und wird daher für die Objekterkennung trainiert und verwendet.

3.4 Aufbereitung zur Verarbeitung in Unity

Nachdem das YOLOv8s Modell mit dem Datensatz trainiert wird, müssen die Ergebnisse in ein geeignetes Format konvertiert werden, damit sie in Unity weiterverarbeitet werden können. In diesem Unterkapitel werden alle Schritte beschrieben, die notwendig für die Weiterverarbeitung in Unity sind.

3.4.1 JSON Format

Jedes Objekt, das in der Unity als Gebäude oder Fahrzeug erkannt werden soll, muss in ein JSON-Format umgewandelt werden, damit es später im digitalen Zwilling angezeigt

werden kann. Die folgende Abbildung 3.11 zeigt zwei Beispiele für den Aufbau der JSON-Datei.

```
{
    "type": "Building",
    "frame": 1,
    "xCenter": "0.6176142692565918",
    "yCenter": "0.21509212255477905",
    "width": "0.08986683189868927",
    "height": "0.09903907775878906",
    "color": "126-58-34",
    "angle": "90.00"
},
```

```
{
    "type": "Car",
    "frame": 386,
    "xCenter": "0.8667844533920288",
    "yCenter": "0.8068518042564392",
    "width": "0.03543052822351456",
    "height": "0.12801972031593323",
    "color": "229-220-74",
    "angle": "0.00"
},
```


(a) (b)

Abbildung 3.11: JSON-Beispiele zur Verarbeitung in Unity

(a) Beispiel JSON-Format für Gebäude (b) Beispiel JSON-Format für Fahrzeuge

Das JSON-Format besteht aus insgesamt 8 unterschiedlichen Attributen und ist wie folgt aufgebaut:

- ***type***: Typ des Objekts (z. B. Building/Car).
- ***frame***: Wenn es sich um die Erkennung eines Bildes handelt, ist *frame* immer 1, andernfalls enthält es die Frame-Nummer.
- ***xCenter***: X-Koordinate des Mittelpunkts des Objekts.
- ***yCenter***: Y-Koordinate des Mittelpunkts des Objekts.
- ***width***: Breite des Objekts.
- ***height***: Höhe des Objekts.
- ***color***: Farbe des Objekts im RGB-Format.
- ***angle***: Rotation des Objekts in Grad.

3.4.2 Farberkennung

Alle Gebäude und Fahrzeuge in der Modellwelt haben eine Farbe. Um die Farbe jedes einzelnen Objektes im digitalen Zwilling anzeigen zu können, muss die Farbe jedes Objektes bei der Objekterkennung erfasst werden. Dies bedeutet, dass die Farberkennung gleichzeitig mit der Objekterkennung erfolgen muss. Andernfalls ist ein zusätzlicher Verarbeitungsschritt erforderlich, um die Farben den Objekten zuzuordnen.

Aus diesem Grund wurde ein Python-Skript entwickelt, das die Farberkennung in den Objekterkennungsprozess integriert. Die einzelnen Schritte werden im Folgenden detailliert beschrieben.

YOLOv8 bietet die Methode *save_ one_ box()* [12], die es erlaubt, während der Erkennung einen Ausschnitt des erkannten Objektes aus dem ursprünglichen Bild/Frame auszuschneiden und abzuspeichern. Die jeweiligen Ausschnitte werden im nächsten Schritt verarbeitet, um die Farberkennung durchzuführen.

(a) (b)

Abbildung 3.12: Beispielbild der Modellwelt und Objektausschnitt

(a) Das Originalbild mit dem rot markierten Objekt
(b) Ausschnitt mit der Methode *save_ one_ box()*

k-Means-Algorithmus

Die Farben des jeweiligen Ausschnitts werden gruppiert und die dominanten Farben mithilfe des k-Means-Algorithmus [22] extrahiert.
Die einzelnen Schritte und deren Beschreibung sind in der nachfolgenden Abbildung 3.13 dargestellt.

```python
def get_dominant_colors(image, k=3, number=1):
    # Erstellen eines leeren Ergebnisdicts
    result = {"colors": [], "image": []}

    # Umwandlung des Bildes von BGR in RGB
    image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)

    # Umformung der Pixel in ein eindimensionales Array
    pixels = image.reshape(-1, 3)

    # Anwendung des k-means-Clustering-Algorithmus
    criteria = (cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER, 100, 0.2)
    _, labels, centers = cv2.kmeans(pixels.astype(np.float32), k, None, criteria, 10, cv2.KMEANS_RANDOM_CENTERS)

    # Konvertierung der Clusterzentren von Float zu uint8
    centers = np.uint8(centers)

    # Ermittlung der eindeutigen Labels und ihrer Häufigkeiten
    unique_labels, counts = np.unique(labels, return_counts=True)

    # Sortierung der Indizes in absteigender Reihenfolge
    sorted_indices = np.argsort(counts)[::-1]

    # Extraktion der dominanten Farben basierend auf den sortierten Indizes
    dominant_colors = centers[unique_labels[sorted_indices[:number]]]

    # Aktualisierung des Ergebnisdicts mit den dominanten Farben
    result["colors"] = dominant_colors

    # Erstellung eines neuen Bildes mit den geclusterten Farben
    new_image = centers[labels.flatten()]
    new_image = new_image.reshape(image.shape)

    # Aktualisierung des Ergebnisdicts mit dem neuen Bild
    result["image"] = new_image

    # Rückgabe des Ergebnisdicts
    return result
```

Abbildung 3.13: Die Methode *get_ dominant_ colors()*

Das Ergebnis der Methode ist eine Hashmap, die sowohl das veränderte Bild als auch die dominante Farbe im RGB-Format enthält.

(a) (b)

Abbildung 3.14: Das Ergebnis der k-means Algorithmus

(a) Eingabebild
(b) Die visuelle Darstellung nach der Ausführung get_dominant_colors()

Wie in der Abbildung 4.5 zu sehen ist, werden in diesem Beispiel die Pixel in drei verschiedene Farbcluster aufgeteilt. Die Methode get_dominant_colors() liefert als Ergebnis eine Hashmap, die sowohl das modifizierte Bild als auch die dominante Farbe im RGB-Format enthält. Die Methode erhält außerdem als ersten Parameter *Größe von k* und als zweiten Parameter *Anzahl der dominanten Farben*. Wird z.B. als zweiter Parameter der Wert 2 gewählt, gibt die Methode die zweitdominantesten Farben zurück. Die Methode verwendet intern die OpenCV-Bibliothek, um die relevanten Informationen zu berechnen.

3.4.3 Konturerkennung

Nach der Farberkennung ist der nächste Schritt die Konturerkennung. Ziel dieses Vorgangs ist es, anhand der Kontur des jeweiligen Objekts dessen Rotation zu bestimmen. Das heißt, es wird geprüft, ob das Objekt schräg in der Modellwelt liegt oder ob es gerade ausgerichtet ist. Dies ist besonders wichtig für Fahrzeuge, die sich in der Umgebung auf der Fahrbahn bewegen und dabei ständig ihre Position und Richtung ändern.

MinAreRect() Methode

Die Methode *MinAreaRect()* [6] der OpenCV-Bibliothek berechnet das minimale umschließende Rechteck mit der minimalen Fläche. Dabei wird auch die Rotation berück-

sichtigt und der Winkel berechnet, in dem das Objekt ausgerichtet ist. Zur Berechnung des minimal umschließenden Rechtecks benötigt die Methode 4 Eckpunkte des Rechtecks als Eingabe. Als Ausgabe liefert sie die x- und y-Koordinaten des Mittelpunktes, die Breite und Höhe des Rechtecks sowie den berechneten Rotationswinkel. Zur Berechnung des Winkels werden die 4 Eckpunkte im Uhrzeigersinn beginnend mit dem Punkt mit der höchsten y-Koordinate angeordnet. Wenn 2 Punkte die gleiche hohe y-Koordinate haben, wird der am weitesten rechts liegende Punkt als Startpunkt gewählt. Dann wird der Drehwinkel zwischen der Linie, die die Anfangs- und Endpunkte verbindet, und der Horizontalen berechnet. Die Abbildung 3.15 zeigt die Vorgehensweise.

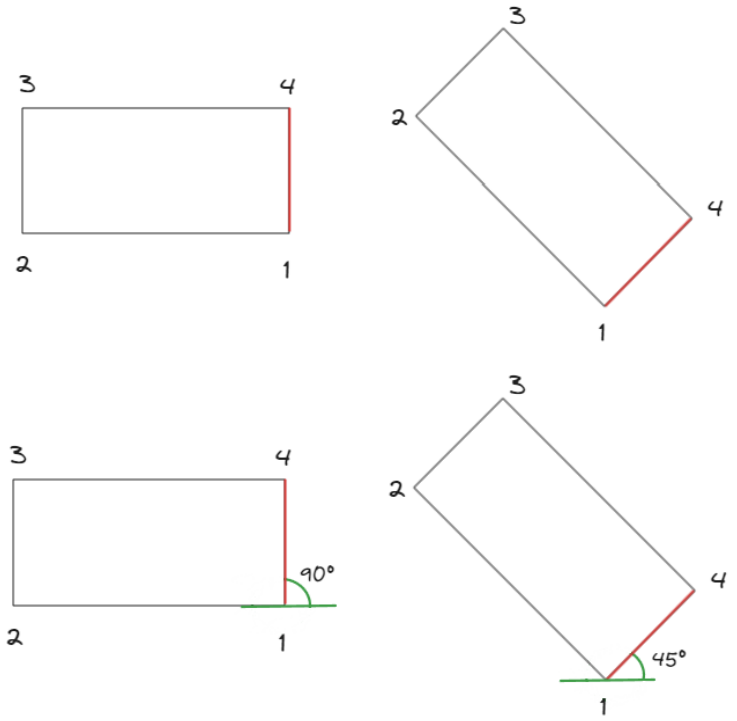
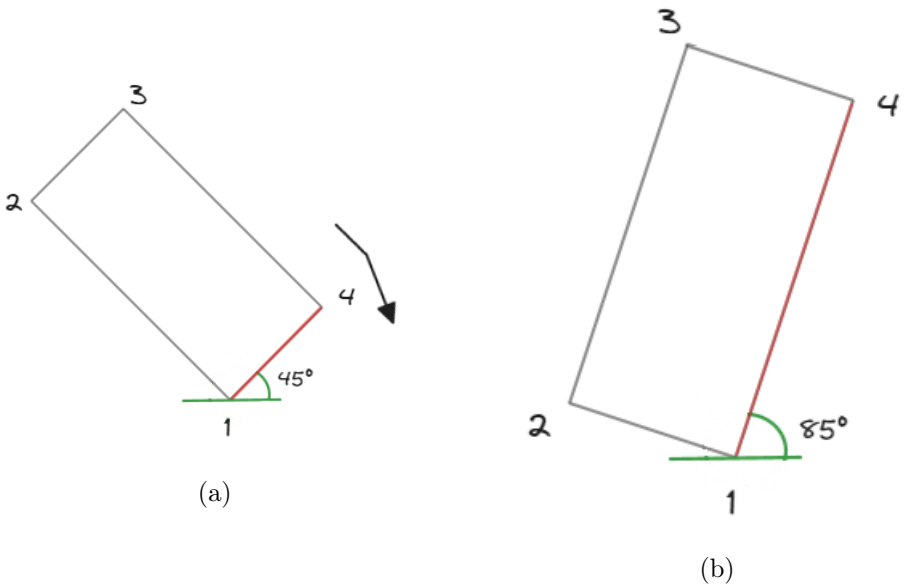


Abbildung 3.15: Visuelle Darstellung der *MinAreaRect()* Methode

Linie, von der der Winkel berechnet wird, in Rot
Berechneter Winkel in Grün

Wenn sich das Rechteck in der Abbildung 3.16 weiter nach rechts dreht, werden die Nummerierungen der Eckpunkte neu berechnet und dementsprechend wird die nächste

Linie zur Berechnung der Winkel verwendet. Aus diesem Grund bleibt der berechnete Winkel immer **zwischen 0 und 90 Grad**.



(a) (b)

Abbildung 3.16: Drehung des Rechtecks nach rechts

(a) Das Rechteck dreht sich weiter nach rechts → Eckpunkte werden neu nummeriert
(b) Neuer Linie zur Berechnung in Rot wird gewählt, berechneter Winkel in Grün

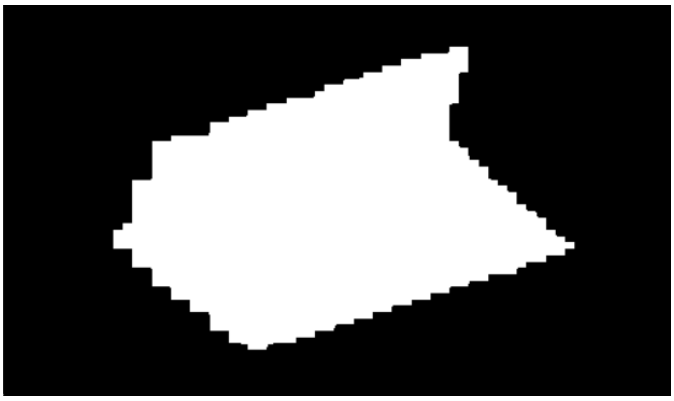
Bei der Farberkennung wird mithilfe des entwickelnden Skripts und die k-means Algorithmus die dominanten Farben in einem Bildausschnitt identifiziert. Die Methode *get_ dominant_ colors()* erzeugt als Ausgabe die Anzahl der dominanten Farben und den modifizierten Bildausschnitt, je nach Angabe des optionalen Parameters *number*. An dieser Stelle wird die erzeugte Ausgabe verwendet, um sie für eine weitere Methode namens *get_ color_ mask_ and_ contour()* zu verwenden, die als Eingabe den Bildausschnitt, eine Farbe im RGB-Format und einen optionalen Parameter *max_ contour* erhält. Die Methode verwendet intern die OpenCV-Bibliothek zur Berechnung des Rotationswinkels.

Im Folgenden wird die Vorgehensweise der Methode detailliert beschrieben:

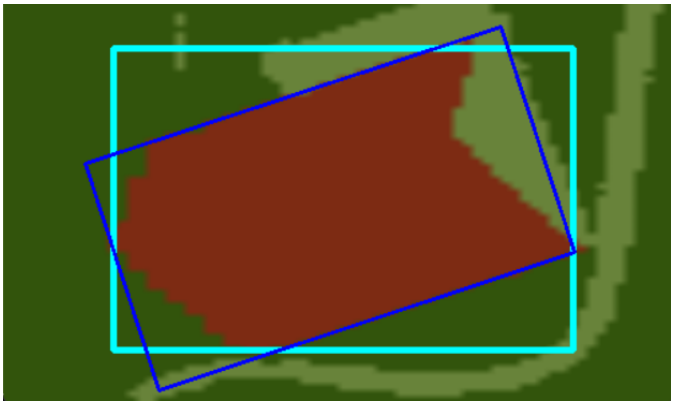
- Zuerst wird aus dem Bild eine Maske mit der eingegebenen Farbe erzeugt. Die Maske trennt die Bereiche des Bildes ab, die der angegebenen Farbe entsprechen.

- Im nächsten Schritt wird der prozentuale Anteil der Maske am Gesamtbild berechnet und auf zwei Dezimalstellen gerundet.
- Anschließend werden die Konturen der Maske mit der Funktion *cv2.findContours()* gesucht.
- Wenn mindestens eine Kontur gefunden wurde, wird der nächste Schritt anhand der optionalen Parameter bestimmt.
- Wenn *max_ contour* wahr ist, wird die größte gefundene Kontur aus allen Konturen ausgewählt, andernfalls werden alle gefundenen Konturen zusammengefasst und ausgewählt.
- Anschließend wird mithilfe der Funktion *CV2.MinAreaRect()* der Winkel des minimalen umschließenden Rechtecks der Kontur berechnet.
- Die Methode liefert als Ausgabe den prozentualen Anteil der Kontur an der Gesamtzahl der Pixel und den berechneten Winkel.

Originalbild als Eingabe



Die Maske aus der RGB Farbe: 125, 43, 19



Rechteck in Hellblau – Normaler Rechteck
Rechteck in Dunkelblau – *MinAreaRect* Rechteck mit der **Winkelberechnung**

Abbildung 3.17: Visuelle Darstellung der *get_ color_ mask_ and_ contour()* Methode

Wie in der Abbildung 3.17 zu sehen ist, erzeugt die Methode zunächst eine Maske aus dem Eingabebild und der Farbe. Anschließend sucht die Methode nach allen vorhandenen Konturen in der angegebenen Maske und zeichnet ein *MinAreaRect* (minimal umschließendes Rechteck) um die Kontur.

{'color': '125-43-19', 'percentage': '31.32', 'angle': '64.44'}

Abbildung 3.18: Ergebnis der Methode *get_ color_ mask_ and_ contour()*

Erster Wert – die RGB Farbe, mit der die Maske berechnet wird
Zweiter Wert – die Prozentzahl der Maske im Vergleich zu Gesamtpixeln
Dritter Wert – Berechneter Winkel der Maske

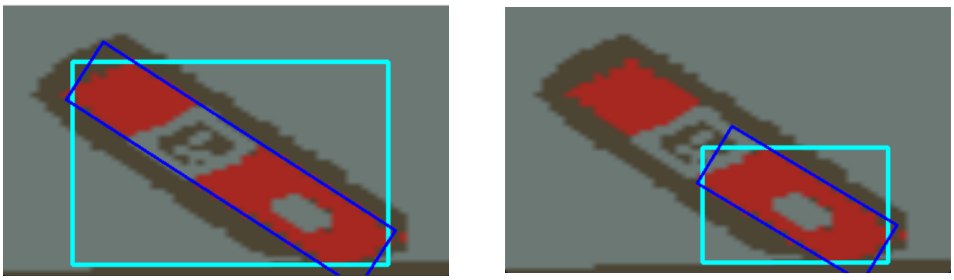
max_contour Parameter

Bei Verwendung der Methode *get_ color_ mask_ and_ contour()* kann ein optionaler Parameter ***max_ contour*** übergeben werden, der standardmäßig auf *True* gesetzt ist. Wenn der Wert dieser Variable *True* ist, wird bei der Ausführung der Methode die größte Kontur, der in der Maske gefunden wird, zurückgegeben, und anschließend wird das minimal umschließende Rechteck (*MinAreaRect*) für diese Kontur berechnet. Andernfalls werden alle gefundenen Konturen zusammengeführt und das ***MinAreaRect*** wird für diese kombinierten Konturen berechnet. Die Abbildung 3.20 visualisiert den Unterschied.

(a) (b)

Abbildung 3.19: Beispielbild von einem Fahrzeug und die Maske

(a) Bild von einem Fahrzeug als Eingabe
(b) Die Maske aus der Farbe Rot



(a) (b)

Abbildung 3.20: Visuelle Darstellung der Auswirkung von *max_ contour* Parameter

(a) *max_ contour* = False → Berechnung der **gesamten Kontur**
(b) *max_ contour* = True → Berechnung der **größten Kontur**

Das obige Beispiel zeigt, dass zuerst alle Konturen zusammengeführt werden und dann *MinAreRect()* berechnet wird, wenn *Max_ contour* auf *False* gesetzt ist.

3.4.4 Ablaufdiagramm

Abschließend zeigt die Abbildung 3.21 den Ablauf der Vorbereitung der Daten für die Verarbeitung in Unity als Flussdiagramm.

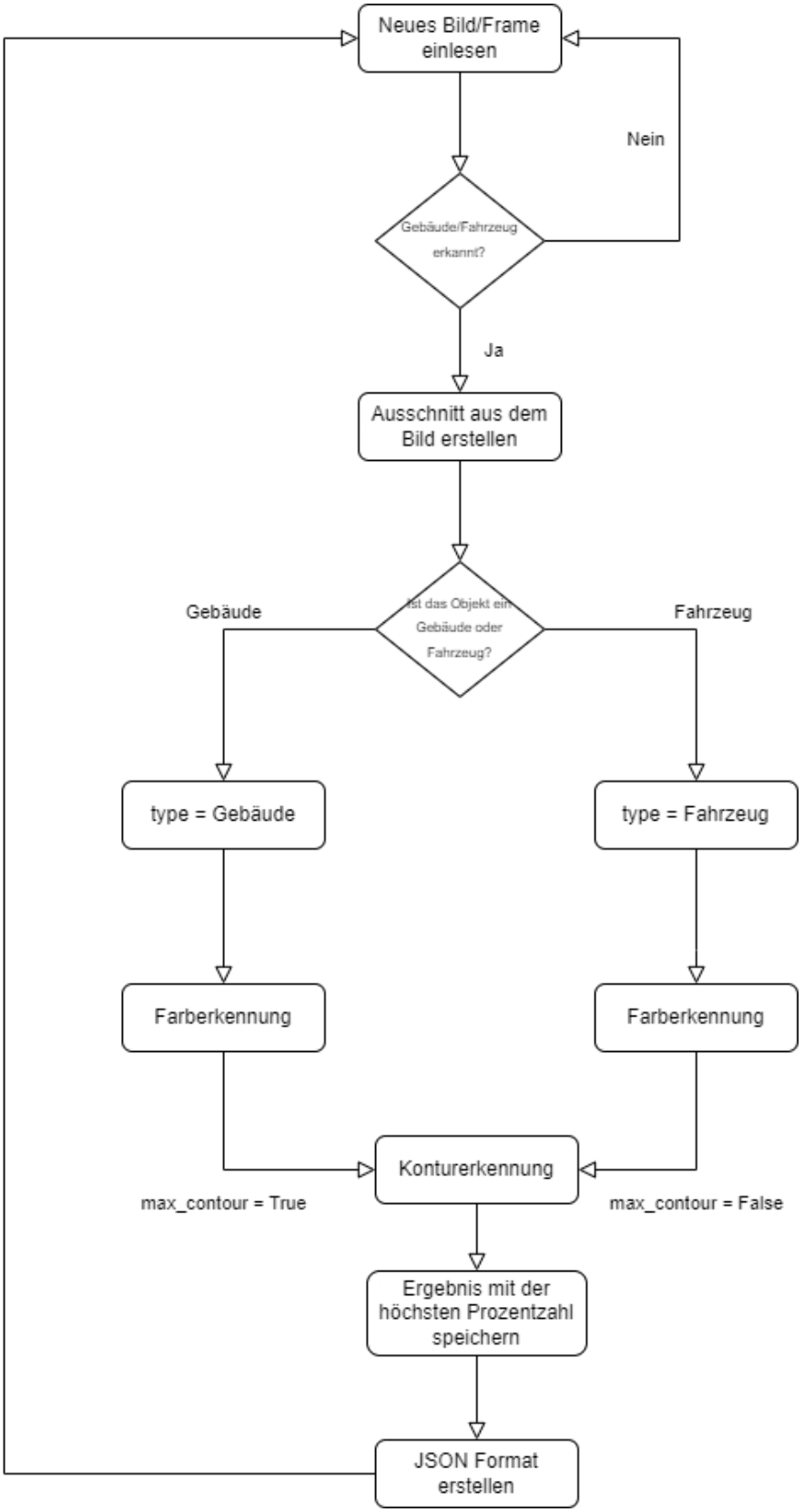


Abbildung 3.21: Ablaufdiagramm für die in diesem Kapitel beschriebenen Schritte

3.5 Unity

Um den digitalen Zwilling in Unity mit den in den vorherigen Schritten generierten Daten zu erstellen, werden einige Klassen benötigt. Diese Klassen sind im folgenden Klassendiagramm in Abbildung 3.22 dargestellt.

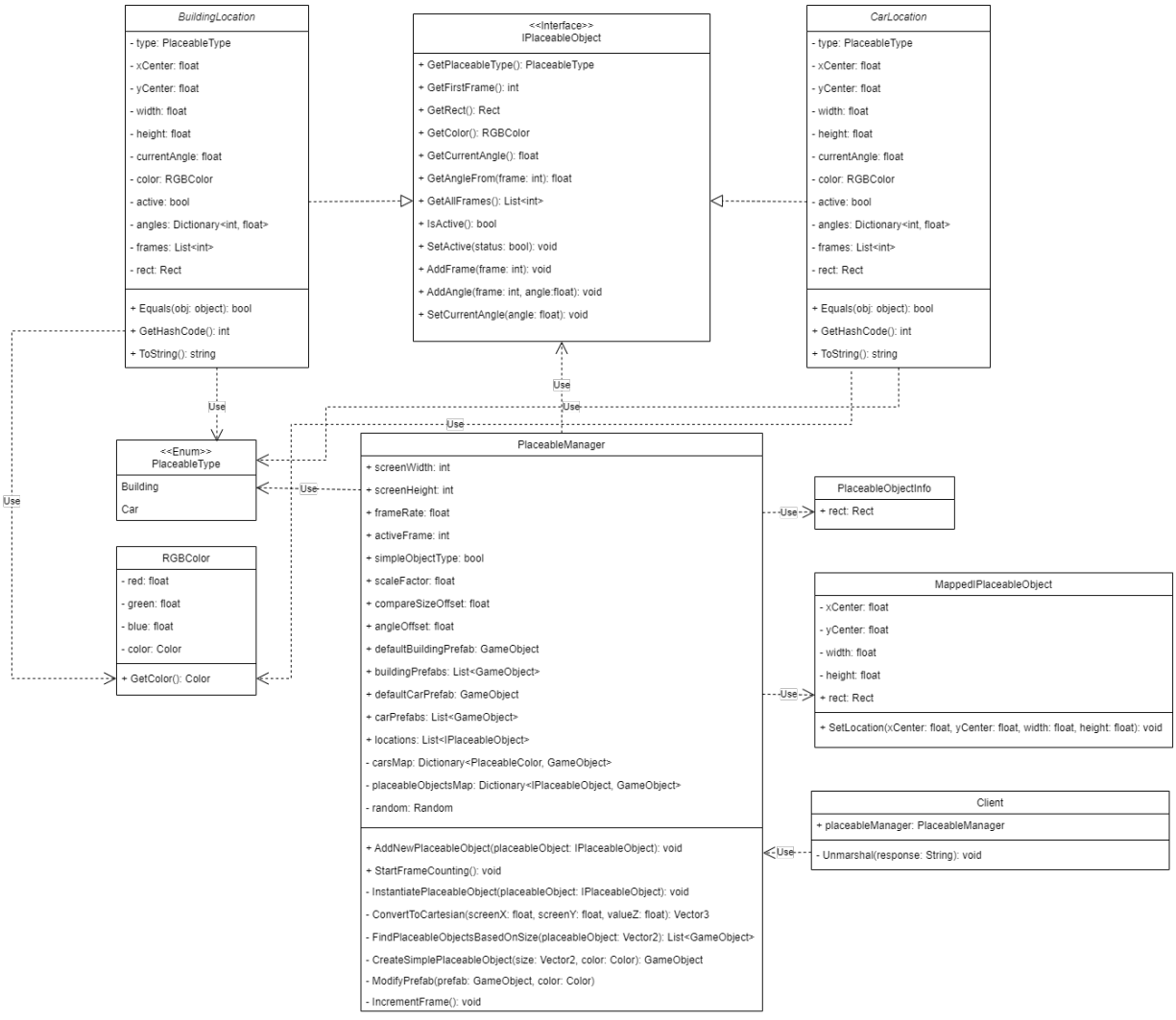


Abbildung 3.22: Die Architektur des digitalen Zwillings in Unity als Klassendiagramm

3.5.1 IPlaceableObject

Alle Objekte, die im digitalen Zwilling innerhalb einer Szene in Unity angezeigt werden sollen, müssen das Interface *IPlaceable* implementieren. Dieses Interface bietet eine Reihe von definierten Funktionen, die bei der Ausführung benötigt werden. Die wichtigsten Methoden und deren Funktionsweise wird wie folgt definiert:

1. *GetPlaceableType()*: Gibt den Typ des Objekts vom Typ *PlaceableType* zurück (Building/Car).
2. *IsActive()*: Boolescher Wert, der den Status des Objekts zurückgibt. Ist er auf *True* gesetzt, wird das entsprechende Objekt im aktuellen Frame angezeigt.
3. *GetRect()*: Liefert die Position des Objekts. Sie enthält die x- und y-Koordinaten sowie die Breite und Höhe des Objekts.
4. *GetColor()*: Gibt die Farbe des Objekts vom Typ RGBColor zurück.
5. *GetCurrentAngle()*: Liefert den aktuellen Winkel des Objektes.
6. *GetAllFrames()*: Gibt eine Liste aller Frames zurück, in denen das Objekt angezeigt werden soll.

Die Klassen **CarLocation** und **BuildingLocation**, die das Interface *IPlaceableObject* implementieren, enthalten relevante Informationen über die Position, Farbe, Winkel und den Zustand eines Objekts und werden bei der Instanziierung innerhalb des digitalen Zwillings benötigt.

3.5.2 Client

Um die erkannten Objekte im digitalen Zwilling anzeigen zu können, müssen die Ergebnisse der Objekterkennung in Unity verarbeitet werden. Die Klasse *Client* ruft beim Starten des Programms eine Ressource auf, die die gewünschten Ergebnisse im JSON-Format (siehe Abbildung 3.11) bereitstellt. Die Methode *Unmarshal()* innerhalb der Klasse analysiert den JSON-String und erzeugt basierend auf dem Objekttyp (Building/Car) ein *IPlaceableObject*. Anschließend wird die Methode *AddNewPlaceableObject()* der *PlaceableManager* Klasse mit dem erstellten *IPlaceableObject* aufgerufen.

3.5.3 PlaceableManager

Die Komponente *PlaceableManager* ist für die Erzeugung und Instanziierung der Objekte innerhalb des digitalen Zwillings verantwortlich. Sie wird beim Start der Szene aktiviert und verfügt über eine Vielzahl von Parametern, die das Verhalten der Objekte innerhalb der Szene beeinflussen. Diese Komponente wird an ein leeres GameObject *Placeable Settings* angehängt und kann somit über den Inspector in Unity gesteuert werden.

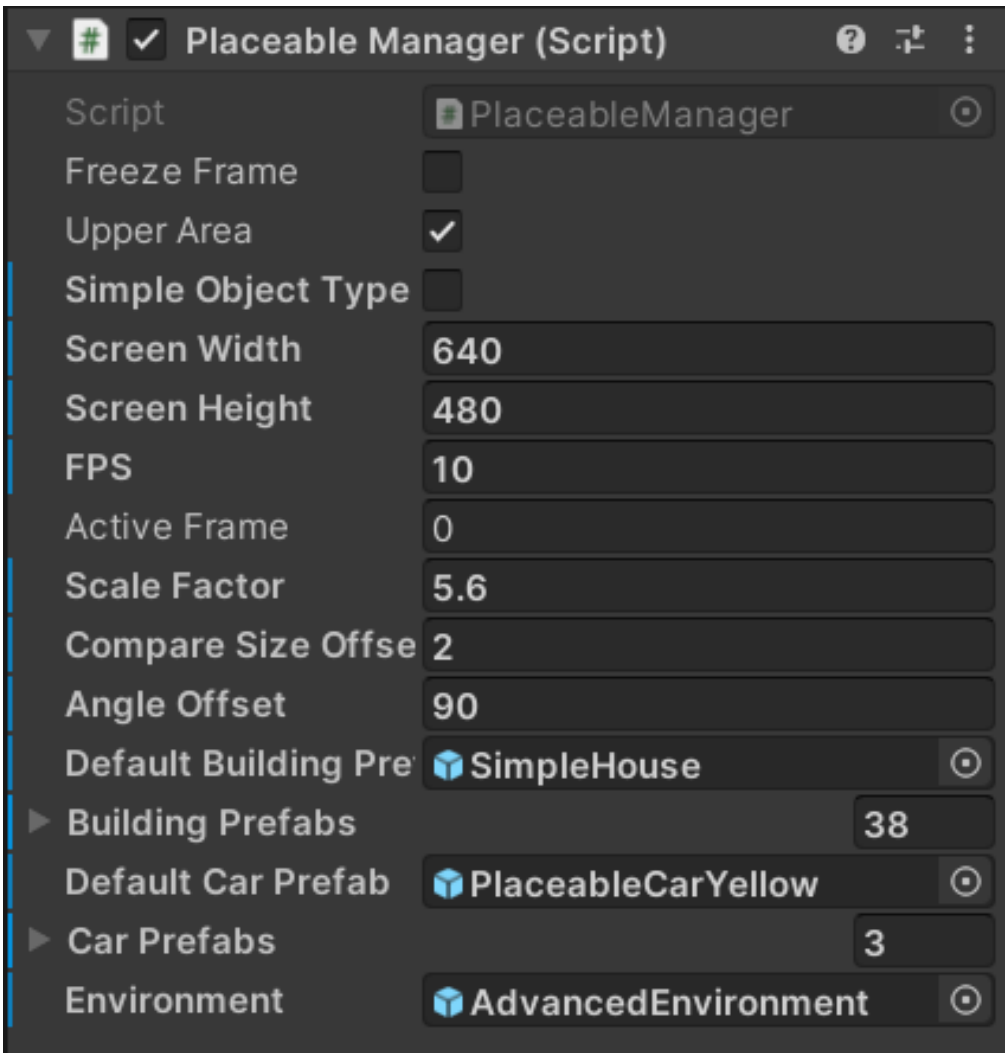


Abbildung 3.23: *PlaceableManager* Komponente

Die Abbildung 3.23 zeigt die *PlaceableManager* Komponente. Die Parameter werden wie folgt definiert:

- **FreezeFrame**: Wenn dieser Wert auf *True* gesetzt ist, wird der aktuelle Frame angehalten.
- **UpperArea**: Dieser Wert legt fest, ob die Eingabe ein Video aus dem oberen oder unteren Bereich der Modellwelt ist (entsprechend werden die Objekte an der richtigen Position positioniert).
- **SimpleObjectType**: Wenn dieser Wert gesetzt ist, werden anstelle der Gebäude und Fahrzeuge einfache Rechtecke angezeigt, die jedoch die Ausrichtung und Farbe der realen Objekte in der Modellwelt haben.
- **ScreenWidth** und ***ScreenHeight***: Die Breite und Höhe (in Pixel) der Bilder, die für die Objekterkennung verwendet werden. Diese Parameter sind wichtig für die Umrechnung der Bildschirmkoordinaten in kartesische Koordinaten.
- **FPS**: Die Bildrate in der Szene.
- **ActiveFrame**: Zeigt den aktuellen Frame an, in dem sich die Szene befindet.
- **ScaleFactor**: Skalierungsfaktor, mit dem die Objekte in der Szene skaliert werden. Dieser Wert wird verwendet, um die Objekte im digitalen Zwilling korrekt zu positionieren.
- **CompareSizeOffset**: Legt fest, wie weit die 3D-Modelle in der Szene von der tatsächlichen Breite und Höhe der Objekte in der Modellwelt abweichen dürfen (dieser Wert wird verwendet, um ein 3D-Modell mit der ungefähren Größe des Gebäudes zu finden).
- **AngleOffset**: Der Winkel eines Objekts wird um den **AngleOffset** subtrahiert.
- **DefaultBuildingPrefab**: Wird für ein Gebäude kein 3D-Modell in passender Größe gefunden, wird stattdessen dieses GameObject verwendet.
- **BuildingPrefabs**: 3D-Modelle der Gebäude, die in der Szene platziert werden sollen.
- **DefaultCarPrefab**: Wird für ein Fahrzeug kein 3D-Modell in der passenden Größe gefunden, wird stattdessen dieses GameObject verwendet.

- **CarPrefabs**: 3D-Modelle der Fahrzeuge, die in der Szene platziert werden sollen.
- **Environment**: Der digitale Zwilling GameObject wird hier hinzugefügt (wird für interne Berechnungen verwendet).

3.5.4 PlaceableObjectInfo

Um die Gebäude im digitalen Zwilling darstellen zu können, müssen zunächst 3D-Modelle in der Szene platziert werden. Während der Programmausführung werden dann Kopien dieser Modelle an den berechneten Positionen mit den entsprechenden Farben und Winkeln erzeugt. Um diesen Ansatz zu realisieren, wird die Klasse *PlaceableObjectInfo* benötigt, die zur Laufzeit die Breite und Höhe des jeweiligen 3D-Modells berechnet.

Bei der Ausführung des Programms werden die gesammelten Informationen verwendet, um ein passendes 3D-Modell mit einer ungefähren Größe auf der Grundlage des *CompareSizeOffset*-Wertes für jedes Gebäude zu finden und es an der richtigen Position zu platzieren. Dies geschieht durch Aufruf der entsprechenden Informationen. Die Größe des 3D-Modells wird dabei gemäß den Berechnungen der Klasse *PlaceableObjectInfo* bestimmt.

Wenn die Option *SimpleObjectType* in der PlaceableManager-Komponente aktiviert ist, sind diese Schritte nicht erforderlich. In diesem Fall wird für jedes Objekt ein einfaches 3D-Modell erzeugt, das genau die gleiche Größe hat. Dadurch entfällt die Berechnung der Größen der 3D-Modelle zur Laufzeit und es müssen keine 3D-Modelle vor der Ausführung des Programms in der Szene platziert werden.

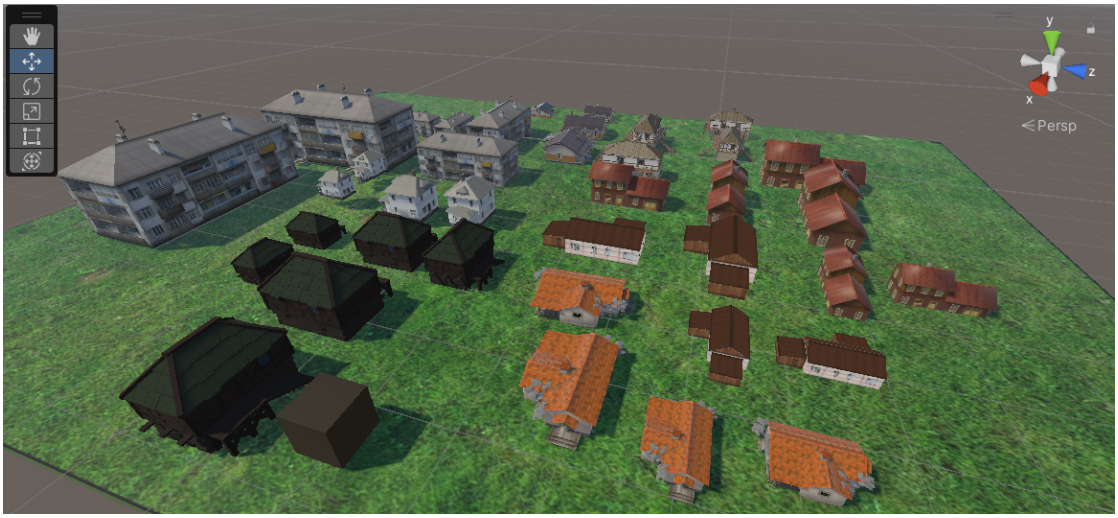


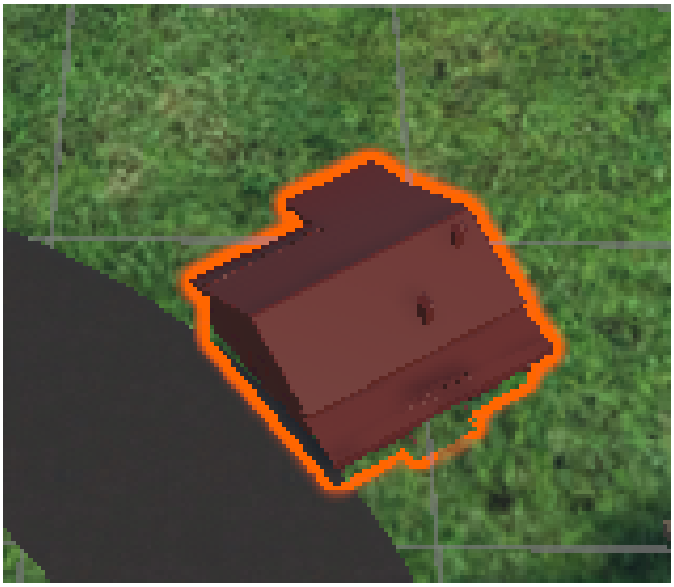
Abbildung 3.24: Diverse 3D-Modelle, die vor der Laufzeit in der Szene platziert werden

3D-Modelle in verschiedenen Größen, um eine Variation von Größen abzubilden

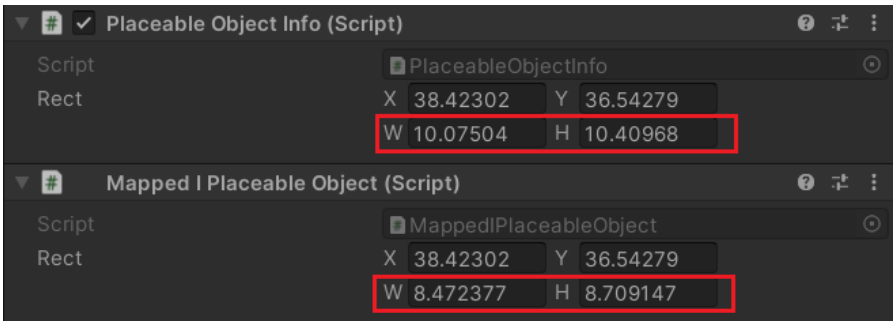
Die Abbildung 3.24 zeigt 38 verschiedene 3D-Modelle in unterschiedlichen Größen, die vor dem Programmstart in der Szene platziert werden. Jedes 3D-Modell muss in der *BuildingPrefabs*-Liste (siehe Abbildung 3.23) der Komponente *PlaceableManager* hinzugefügt werden, um bei der Berechnung berücksichtigt zu werden. Je mehr 3D-Modelle mit ähnlicher Größe wie die Gebäude in der Modellwelt bereits vor der Laufzeit in der Szene platziert sind, desto besser ist die Auswahlmöglichkeit für die entsprechenden Gebäude.

3.5.5 MappedIPlaceableObject

Nachdem für jedes Gebäude basierend auf seiner Größe ein geeignetes 3D-Modell ausgewählt wurde, wird eine Kopie des Modells erstellt und an der entsprechenden Position platziert. Anschließend wird jedem 3D-Modell eine *MappedIPlaceableObject* Komponente hinzugefügt, die die berechneten Informationen enthält. Diese Komponente speichert die X- und Y-Koordinaten sowie die Breite und Höhe der aktuellen Position in der Szene.



(a)



(b)

Abbildung 3.25: *PlaceableObjectInfo* vs *MappedPlaceableObject*

(a) Platziertes Gebäude im digitalen Zwilling
(b) Berechnete Größen markiert in Rot

Wie in Abbildung 3.25 gezeigt, unterscheiden sich die Breiten (W) und Höhen (H) in den jeweiligen Komponenten. Die berechnete Größe in der Komponente *PlaceableObjectInfo* entspricht der tatsächlichen Größe des 3D-Modells, während die Größe in der Komponente *MappedIPlaceableObject* die Größe des realen Gebäudes aus der Objekterkennung darstellt. Die Abweichung zwischen den Größen beträgt **maximal 2 floats**, wie in der Option *CompareSizeOffset* der Komponente *PlaceableManager* festgelegt (siehe Abbildung 3.23).

(a) Darstellung der Objekte mit 3D-Modelle

(b) Darstellung der Objekte ohne 3D-Modelle (*SimpleObjectType*)

Abbildung 3.26: Darstellung des Beispielbilds im digitalen Zwilling

(b) Gebäude werden als Rechtecke und Fahrzeuge als Kugeln dargestellt

Abbildung 3.27: Beispielbild mit der Objekterkennung

Erkannte Gebäude in Rot
Erkannte Fahrzeuge in Rosa

Die Abbildung 3.26 zeigt zwei Varianten der Darstellung von Objekten im digitalen Zwilling. Die erste Variante verwendet 3D-Modelle für Gebäude und Fahrzeuge, während die zweite Methode einfache Rechtecke und Kugeln anstelle der erkannten Objekte verwendet. Diese erkannten Objekte sind in Farbe und Position identisch mit den entsprechenden realen Modellen. Darüber hinaus wird an einigen Stellen der Winkel, in dem die Objekte in der realen Modellwelt ausgerichtet sind, beibehalten.

4 Evaluation

In diesem Kapitel wird eine Bewertung der angewandten Techniken vorgenommen. Die Bewertung ist in zwei Teile gegliedert. Der erste Teil beschäftigt sich mit der trainierten Objekterkennung. Im zweiten Teil wird die Implementierung in Unity bewertet.

4.1 Objekterkennung

Das *YOLOv8* Modell, das für die Objekterkennung in dieser Arbeit verwendet wurde, wurde mit den folgenden Parametern trainiert.

- **Model**: YOLOv8s
- **Epochs**: 500
- **Batch**: -1
- **Patience**: 100
- **Imgsz**: 640

Der Parameter *Model* legt fest, mit welchem Modell das Training durchgeführt werden soll, *epochs* ist die Anzahl der Epochen, die das Modell durchläuft. Ein *Batch* von -1 bedeutet, dass vor dem Start des Trainings automatisch eine Batch-Größe entsprechend der GPU-Hardware festgelegt wird (in diesem Anwendungsfall wurde eine Batch-Größe von 38 für die Ausführung gewählt) und ein *patience* von 100 bedeutet, dass das Modell das Training abbricht, wenn nach 100 Epochen keine Verbesserung zu erkennen ist. Schließlich ist *Imgsz* die Größe der Eingabebilder in Pixeln.

Nach 387 Epochen wurde das Training abgebrochen, da sich das Modell in den letzten 100 Epochen nicht verbessert hat. Die besten Ergebnisse wurden in Epoche 287 erzielt.

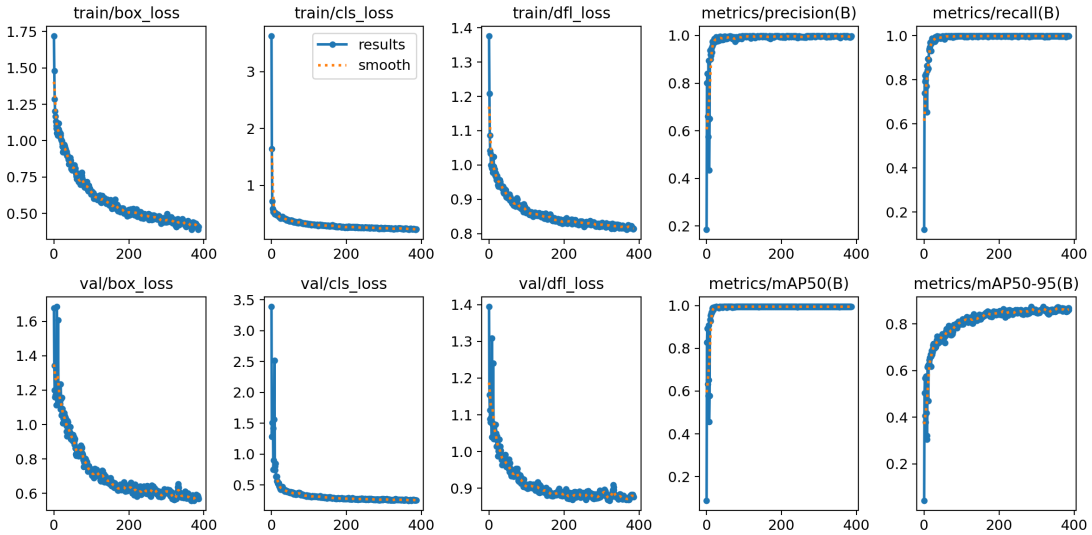


Abbildung 4.1: Die Ergebnisse nach dem Training von YOLOv8s

box_loss, *cls_loss* und *dfl_loss* sind während des Trainings gesunken, während die *mAP* Werte gestiegen sind.

Die Ergebnisse in Abbildung 4.1 zeigen, dass das Training sehr erfolgreich war und das Modell eine hohe Genauigkeit bei der Objekterkennung erreicht hat. Die *Box_loss* erreichte während des Trainings ihren niedrigsten Wert von ca. 0,4, während die *Cls_loss* bei 0,24 lag. Es ist zu erkennen, dass alle Verlustfunktionen sowohl während des Trainings als auch während der Validierung abnahmen. Daraus kann geschlossen werden, dass während des Trainings kein Overfitting stattgefunden hat. Gleichzeitig stiegen die Metriken für die Genauigkeit und die Erkennungsrate auf fast 1,0 an.

Die *mAP50* und *mAP50-95* zeigen ebenfalls einen deutlichen Anstieg nach etwa 50 Epochen. Dies deutet darauf hin, dass das Modell sowohl bei einem festen IoU-Schwellenwert von 0,5 als auch über einen breiteren Bereich von IoU-Schwellenwerten von 0,5 bis 0,95 (mAP50-95) eine immer bessere Objekterkennungsleistung erreicht hat.

Die Betrachtung der Konfusionsmatrix bestätigt die Aussage, dass das Modell eine sehr hohe Genauigkeit aufweist. In der Abbildung 4.2 ist deutlich zu erkennen, dass das Modell in allen Fällen eine Genauigkeit von 100 % bei der Klassifizierung der verschiedenen Klassen erreicht. Die normalisierte Konfusionsmatrix zeigt, dass alle Vorhersagen des Modells korrekt sind und es keine Fehlklassifikationen gibt. Dies unterstreicht die hohe

Leistungsfähigkeit des Modells und seine Fähigkeit, die verschiedenen Klassen genau zu unterscheiden und richtig zuzuordnen.

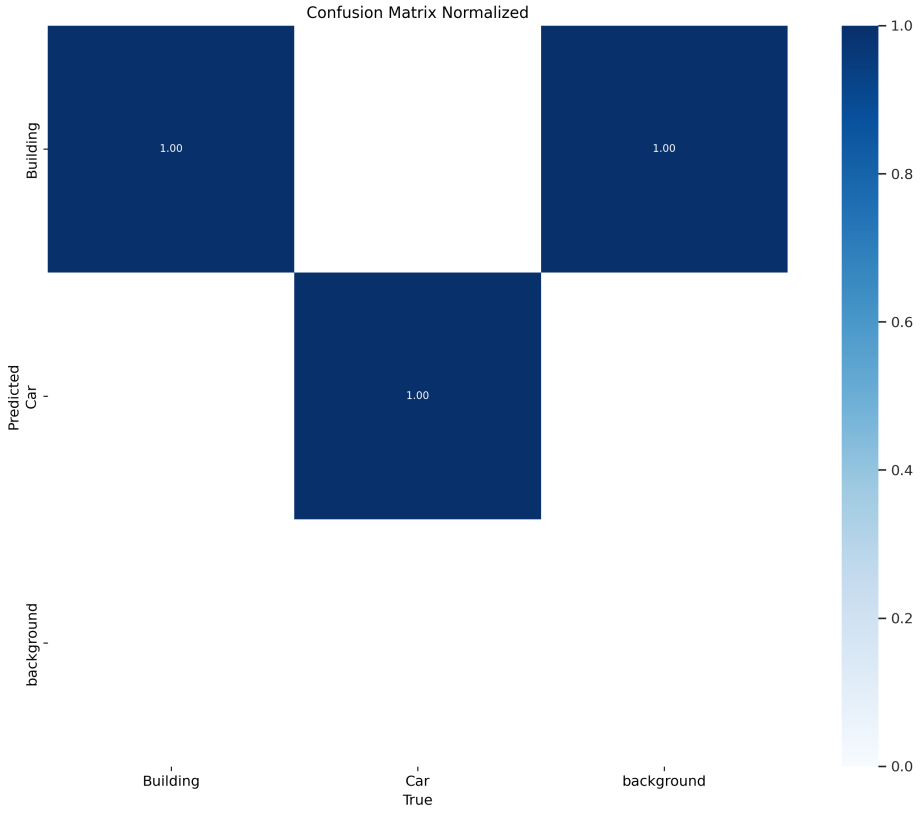


Abbildung 4.2: Normalisierte Konfusionsmatrix des trainierten Modells

Alle Klassen erreichen einen Wert von 1.0

Die PR-Kurve in Abbildung 4.3 zeigt eine gute Klassifikation zwischen Gebäuden und Fahrzeugen in der Modellwelt. Beide Klassen erreichen einen Wert von 0,995. Die Abbildung 4.4 zeigt die Anzahl der Instanzen in den jeweiligen Klassen. Die Klasse *Building* hat etwa 90 % mehr Instanzen als die Klasse *Car*. Obwohl der Datensatz unbalanciert ist, zeigt die PR-Kurve, dass beide Klassen einen hohen mAP erreichen und das Modell sehr gute Klassifikationsentscheidungen trifft, auch wenn wesentlich mehr Gebäudeinstanzen im Datensatz vorhanden sind.

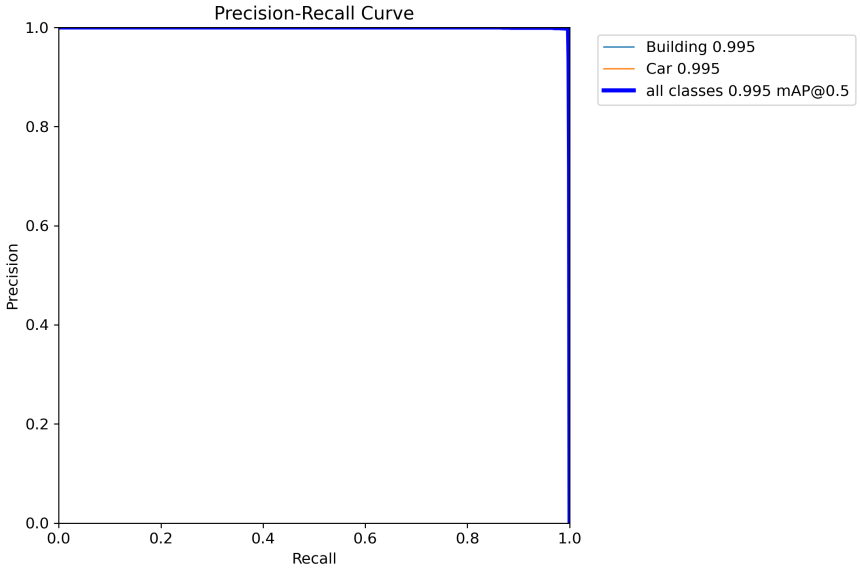


Abbildung 4.3: PR-Kurve des trainierten Modells

Alle Klassen erreichen einen mAP@0.5 von 95 %

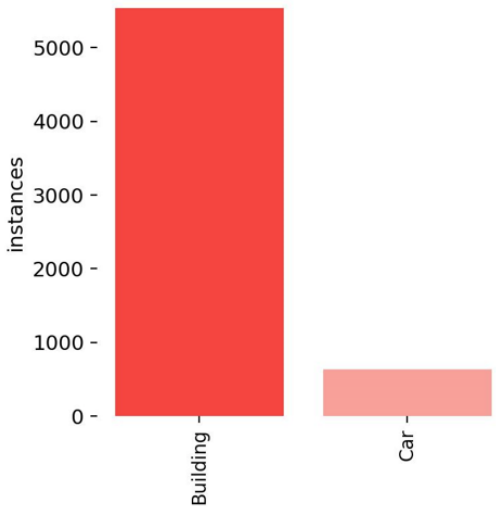


Abbildung 4.4: Anzahl der Instanzen im Datensatz

Die Klasse *Car* enthält im Vergleich zu *Building* nur ca. 10 % Instanzen

4.2 Darstellung in Unity

Um die Präzession von Unity bei der Darstellung von Objekten im digitalen Zwilling zu bewerten, werden verschiedene Videosequenzen im digitalen Zwilling abgespielt. Jedes in der Szene dargestellte Objekt wird pro Frame einzeln bewertet und kann eine maximale Punktzahl von 100 erreichen. Die Punkte werden wie folgt verteilt:

- **Relativer Position im digitalen Zwilling (Gesamt: 50 Punkte)**: Wenn das Objekt innerhalb der berechneten BoundingBox liegt, dann 50 Punkte, sonst 0 Punkte.
- **Die Farbe (Gesamt: 30 Punkte)**: Entspricht die Farbe im digitalen Zwilling der Originalfarbe in der Modellwelt, dann volle Punktzahl, sonst 0. Besteht ein Objekt aus mehreren Farben, dann erhält das Objekt die volle Punktzahl, wenn nur eine der Farben dargestellt wird.
- **Die Ausrichtung (Gesamt: 20 Punkte)**: Stimmt der Winkel im digitalen Zwilling mit dem ausgerichteten Winkel in der Modellwelt überein, dann volle Punktzahl, andernfalls ein Punktabzug pro Grad Abweichung. Beträgt die Abweichung mehr als 20 Grad, 0 Punkte.
- **Falsch abgebildete Objekte**: Für jedes falsch dargestellte Objekt werden 10 Punkte von der Gesamtpunktzahl des betreffenden Frames abgezogen.

Fahrzeug in der Modellwelt

Digitaler Zwilling in Unity

Abbildung 4.5: Beispiel für ein Fahrzeug mit 100 Punkte

+50 Punkte für die **Position**
+30 Punkte für die **Farbe**
+20 Punkte für die **Ausrichtung**

Nachdem alle Punkte in einem Frame addiert wurden, wird die Summe durch die Anzahl der Objekte in der Szene geteilt. Daraus ergibt sich die prozentuale Genauigkeit pro Frame. Nachfolgend sind die Ergebnisse von jeweils 5 Durchläufen für den oberen und unteren Bereich tabellarisch dargestellt. Die Gesamtpunkte pro Video sind der Mittelwert aus allen Frames.

Der höchste Wert ist in Rot und der niedrigste Wert in Blau markiert.

Tabelle 4.1: Ergebnisse für Videos aus dem **oberen Bereich** der Modellwelt

Video	**Frame-Anzahl**	**Gesamtpunkte**	**Gesamtpunkte (%)**
output_video	331	1980	82,5
output_video_2	331	2080	83,2
output_video_3	331	1970	**78,7**
output_video_4	331	2005	80,2
output_video_11	497	2170	**86,8**
Durchschnitt		**2041**	**82,16**

Tabelle 4.2: Ergebnisse für Videos aus dem **unteren Bereich** der Modellwelt

Video	**Frame-Anzahl**	**Gesamtpunkte**	**Gesamtpunkte (%)**
output_video_5	331	2830	85,75
output_video_6	331	2600	78,78
output_video_7	331	2480	**75,1**
output_video_8	331	2760	83,63
output_video_9	331	2940	**89,09**
Durchschnitt		**2722**	**82,47**

Die Ergebnisse aus den Tabellen 4.1 und 4.2 zeigen, dass die Videos aus dem oberen und unteren Bereich der Modellwelt eine durchschnittliche Gesamtgenauigkeit von ca. 82 % aufweisen, woraus geschlossen werden kann, dass der digitale Zwilling die Objekte der Modellwelt mit einer Genauigkeit von ca. **82 %** in Unity abbildet. Darüber hinaus zeigen die Beobachtungen, dass der digitale Zwilling eine höhere Genauigkeit bei der Darstellung eines Objektes erreicht, wenn dieses Objekt in der Modellwelt ständig seinen Zustand ändert. Diese Zustandsänderung ist insbesondere bei Fahrzeugen der Fall, wenn sie sich auf der Fahrbahn bewegen und ihre Position ständig aktualisiert wird. Aus

diesem Grund erhalten die Videos *output_video_11* und *output_video_9* eine höhere Punktzahl. Insgesamt liefert die Umsetzung in Unity mit einer Genauigkeit von ca. 82 % ein gutes Ergebnis. Darüber hinaus besteht die Möglichkeit, den Zustand der Objekte im digitalen Zwilling zu aktualisieren. Die Objekte können sowohl ihre Position als auch ihre Ausrichtung ändern. Wenn Objekte aus dem Sichtfeld der Kameras entfernt werden, werden sie auch nicht mehr in der Szene dargestellt.

5 Reinforcement Learning im digitalen Zwilling

In diesem Kapitel wird ein möglicher Einsatz des zuvor entwickelten digitalen Zwillings mit Hilfe von RL untersucht. ML-Agents [13] ist ein von Unity entwickeltes Framework, das für den Einsatz von RL in Unity verwendet wird. Mit Hilfe dieses Frameworks wird ein Agent im digitalen Zwilling trainiert, der in der Lage ist, autonom im digitalen Zwilling zu fahren.

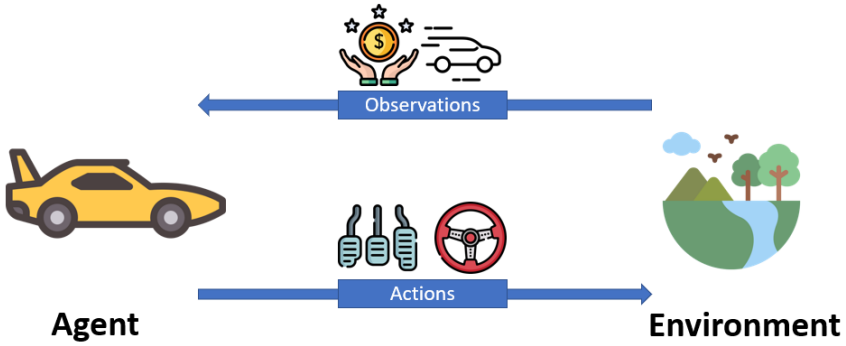


Abbildung 5.1: Visuelle Darstellung der RL

Beim Reinforcement Learning wird ein Agent in einer definierten Umgebung trainiert. Der Agent kann eine Reihe von Aktionen in der Umgebung ausführen und erhält nach jeder ausgeführten Aktion eine Beobachtung von der Umgebung zurück. Die Beobachtung enthält sowohl den aktuellen Zustand, in dem sich der Agent zu diesem Zeitpunkt befindet, als auch eine Belohnung für die ausgeführte Aktion. Anhand dieser Belohnung kann der Agent feststellen, inwieweit die gewählte Aktion seinen Zustand in der Umwelt

beeinflusst hat und ob diese Aktion zu diesem Zeitpunkt eine optimale oder nicht optimale Entscheidung war. Für optimale Handlungen erhält der Agent positive Belohnungen, für nicht-optimale Handlungen negative Belohnungen.

5.1 Ray Perception Sensor 3D

Der *Ray Perception Sensor 3D* ist eine der Möglichkeiten, mit denen der Agent seine Beobachtungen durchführen kann. Die Abbildung 5.3 zeigt die *Ray Perception Sensor 3D* Komponente, die jedem Agenten hinzugefügt werden muss, der eine Beobachtung mit einem Ray Perception Sensor durchführen soll.

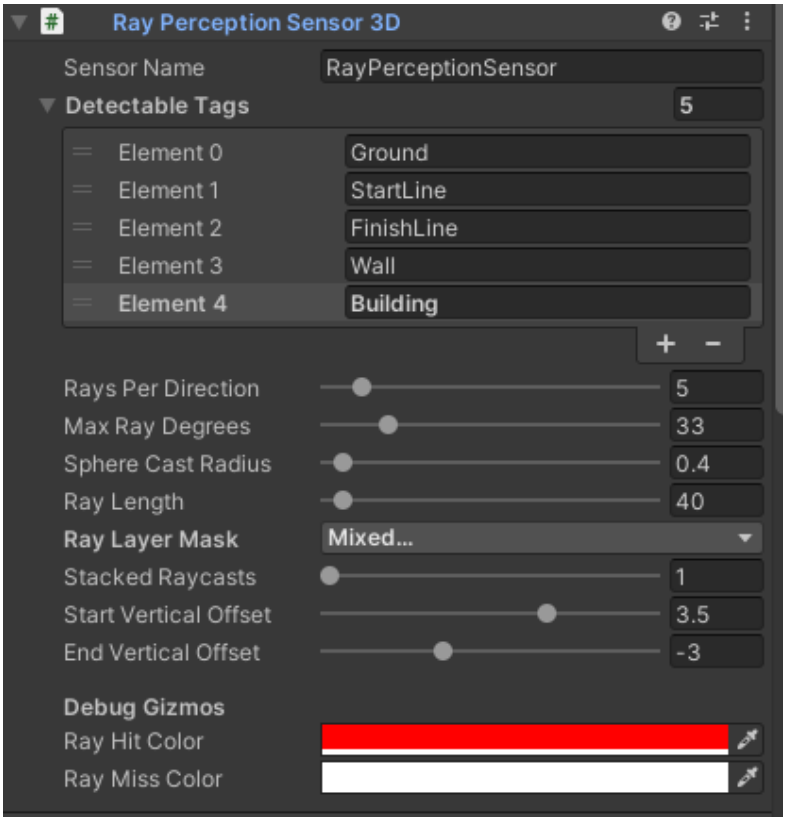


Abbildung 5.2: *Ray Perception Sensor 3D* Komponente

Die Parameter der *Ray Perception Sensor 3D* Komponente sind wie folgt definiert:

- **Sensor Name**: Optionaler Name der Komponente.

- **Detectable Tags**: Alle Tags, die bei der Beobachtung berücksichtigt werden sollen.
- **Rays Per Direction**: Anzahl der Strahlen (Rays) pro Richtung (eine hohe Anzahl von Strahlen erhöht gleichzeitig die Komplexität des Modells).
- **Max Ray Degress**: Winkel, unter dem die Strahlen verteilt werden sollen.
- **Sphere Cast Radius**: Radius der Kugeln am Ende jedes Strahls (ein großer Radius führt zu einer genaueren Objekterkennung).
- **Ray Length**: Die Länge der Strahlen. Je länger die Strahlen, desto mehr entfernte Objekte können erkannt werden.
- **Ray Layer Mask**: Legt fest, welche Layer von den Strahlen erfasst werden können.
- **Stacked Raycasts**: Anzahl der Strahlen, die gestapelt werden sollen, bevor sie an das neuronale Netz übergeben werden.
- **Start Vertical Offset**: Vertikaler Offset für den Beginn der Strahlen.
- **End Vertical Offset**: Vertikaler Offset am Ende der Strahlen.
- **Ray Hit Color**: Farbe des Strahls, wenn der Strahl Objekte identifiziert.
- **Ray Miss Color**: Standardfarbe der Strahlen ohne Erkennung.

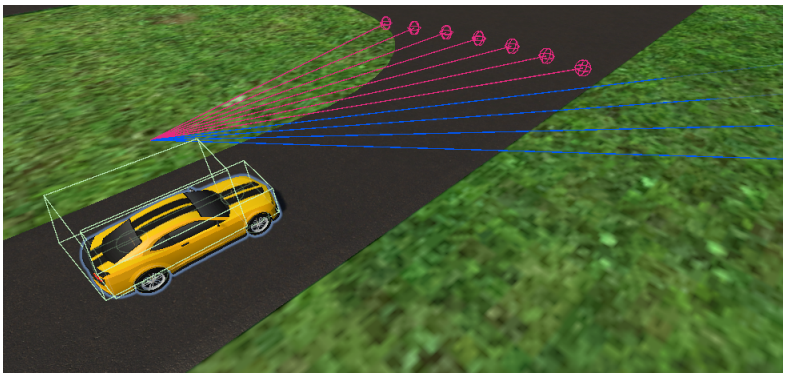


Abbildung 5.3: Visuelle Darstellung der *Ray Perception Sensor 3D* Komponente

Die Strahlen, die ein Objekt erfassen, in rot.
Strahlen, die kein Objekt erfassen, in Blau.

5.2 Behavior Parameters

Die *Behavior Parameters* Komponente ist für jeden Agenten erforderlich, damit das Training durchgeführt werden kann. Diese Komponente fungiert als Gehirn des Agenten und definiert die Strategien und Entscheidungsprozesse des Agenten, die es ihm ermöglichen, die gestellte Aufgabe zu erlernen und zu optimieren.

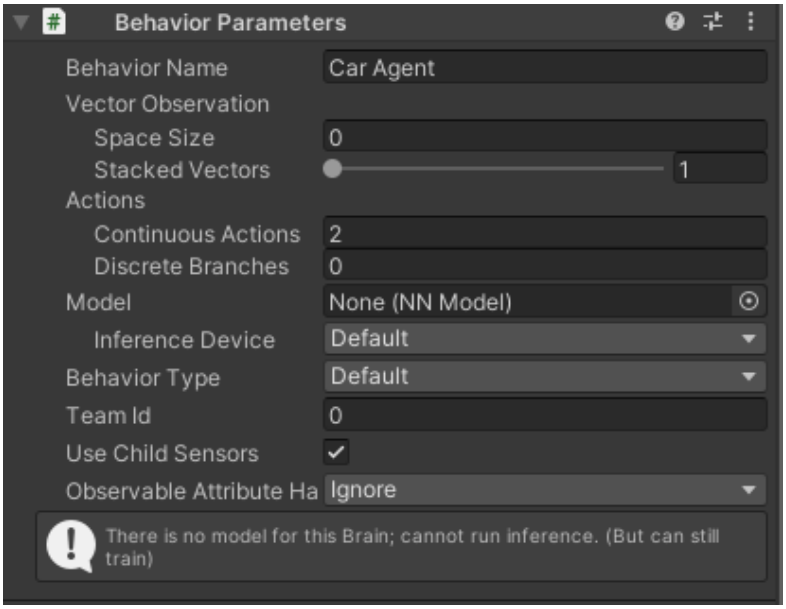


Abbildung 5.4: Die *Behavior Parameters* Komponente

Die Abbildung 5.4 stellt die *Bahvior Parameters* Komponente des Agenten dar. Die Optionen der Komponente sind wie folgt definiert:

- **Bahvior Name**: Name des Agenten.
- **Vector Observation**: Anzahl der Vektorbeobachtungen, falls Vektorbeobachtungen verwendet werden (in diesem Fall wird die Komponente *Ray Perception Sensor 3D* als Beobachtung verwendet, daher bleibt Vector Observation auf 0).
- **Actions**: Definiert den Aktionsraum des Agenten. Es kann zwischen *Continuous* und *Discrete* Werte gewählt werden.
  - *Continuous*: Fließkommazahlen zwischen 0 und 1.
  - *Discrete*: Definierter Zahlenbereich.

- **Model**: Hier kann ein vortrainiertes ONNX-Modell geladen werden oder während des Trainings leer bleiben.
- **Behavior Type**: Wie der Agent in der Umgebung handeln soll. Es gibt verschiedene Optionen wie *Default*, *Heuristic Only* (Agent handelt nach vordefinierten Regeln) oder *Inference Only* (Agent handelt nach trainierten Modellen ohne weitere Lernschritte).
- **Team Id**: Wenn es Agenten gibt, die eine Aufgabe konkurrierend ausführen, können sie einem Team zugeordnet werden.
- **Use Child Sensors**: Legt fest, ob der Agent die zu diesem Agenten hinzugefügten Sensorkomponenten verwenden soll.

In diesem Anwendungsfall hat der Agent 2 Werte (*Continuous Actions*) für den Aktionsraum. Diese Werte beeinflussen jeweils die Geschwindigkeit und die Lenkung des Fahrzeugs.

5.3 Agent

Jedes GameObject in der Szene, das als Agent identifiziert und trainiert werden soll, benötigt zusätzlich das Agent-Skript. Alternativ kann auch ein selbstdefiniertes Skript mit erweiterter Funktionalität erstellt werden, das von der Agentenklasse erbt. Die wichtigsten Methoden des Agenten sind wie folgt definiert:

- *OnEpisodeBegin()*: Alle Konfigurationen für einen Agenten vor dem Beginn einer Episode können hier vorgenommen werden.
- *AddReward()*: Erhöht die Belohnung um den angegebenen Wert.
- *SetReward()*: Setzt eine neue Belohnung für den Agenten.
- *RequestAction()*: Erfordert eine neue Aktion für den Agenten.
- *OnActionReceived()*: Hier wird das Verhalten des Agenten in jeder Episode implementiert.
- *EndEpisode()*: Beendet die aktuelle Episode.

Eine detaillierte Beschreibung aller Methoden der Agent-Klasse steht hier[7].

Die Belohnungen für den Agenten sind wie folgt definiert:

- **-5**: Die Startlinie wird überfahren (Zusätzlich wird die Episode beendet).
- **-2**: Kollision mit einem anderen Agenten.
- **-5**: Kollision mit einem Gebäude (Zusätzlich wird die Episode beendet).
- **+20**: Die Ziellinie wird überfahren.
- **+0 - 10**: Basierend auf die Geschwindigkeit wird eine Belohnung zwischen 0 (Agent nicht in Bewegung) und 10 (Agent mit maximaler Geschwindigkeit in Bewegung) vergeben.

5.4 Training

Nachdem alle für das Training notwendigen Komponenten konfiguriert wurden, kann das Training beginnen. Der Agent verwendet die bereitgestellten Informationen und Belohnungen, um über mehrere Episoden hinweg zu lernen, welche Aktionen zu einem bestimmten Zeitpunkt die optimale Entscheidung darstellen.

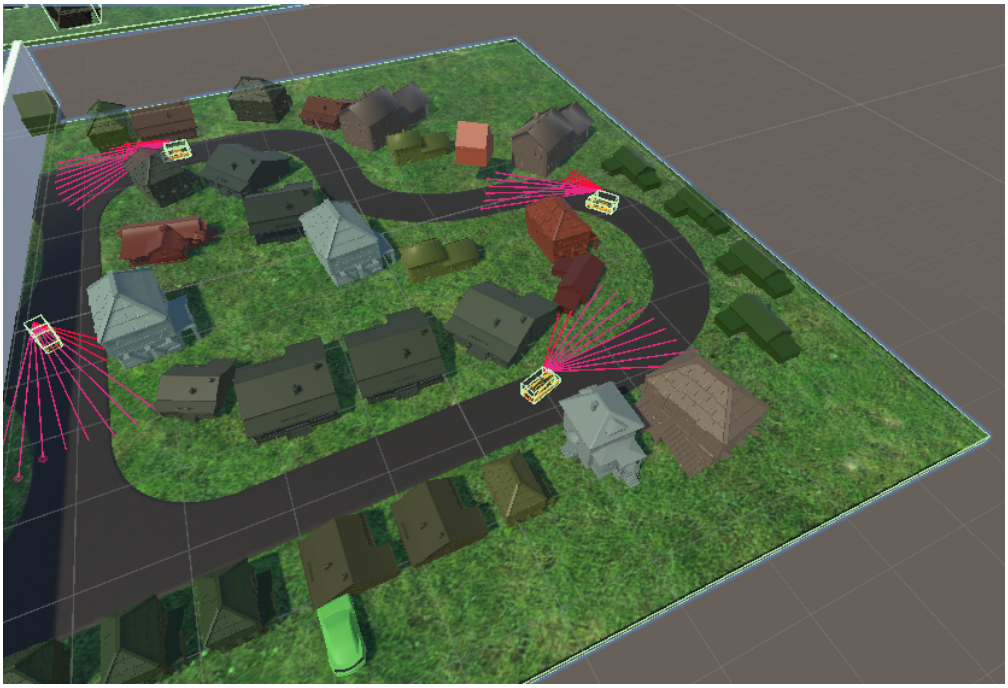


Abbildung 5.5: Agenten beim Training im digitalen Zwilling

Das Sichtfeld der Agenten wird mithilfe der *RayPerceptionLayer3D* in Rot dargestellt.

Die obere Abbildung 5.5 zeigt vier Agenten während des Trainings in der Szene. Die Anzahl der Agenten kann beliebig gewählt werden, um die Trainingszeit zu verkürzen. Die vergleichsweise geringe Anzahl an Konfigurationsschritten zeigt, wie einfach das Training in einer digitalen Umgebung durchgeführt werden kann. Ein großer Vorteil ist, dass die eingestellten Parameter jederzeit verändert werden können, um bessere Ergebnisse zu erzielen oder dem Agenten neue Funktionalitäten hinzuzufügen. Besonders bemerkenswert ist, dass es möglich ist, auf Basis eines bereits trainierten ONNX-Modells, die Agenten weiter zu trainieren und für spezielle Anwendungszwecke anzupassen. Dies ermöglicht eine hohe Flexibilität und Effizienz, da bereits erlerntes Wissen wiederverwendet und gezielt an spezifische Aufgaben oder Umgebungen angepasst werden kann. Die Kombination aus digitaler Umgebung und maschinellem Lernen eröffnet somit eine leistungsfähige und anpassungsfähige Trainingsplattform. Sie ermöglicht die Simulation komplexer Szenarien und Problemstellungen, ohne dass reale Objekte oder physische Experimente benötigt werden.

6 Fazit

Im Rahmen dieser Arbeit wurde ein digitaler Zwilling einer Modellwelt im Maßstab 1:87 in Unity entwickelt. Das Hauptziel des digitalen Zwillings war es, die Gebäude und Fahrzeuge der realen Modellwelt in Unity abzubilden. Dabei wurden die Objekte so erfasst und positioniert, dass sie in der richtigen Position, Farbe und Ausrichtung wiedergegeben werden konnten. Die Umsetzung des Projekts erfolgte in mehreren Schritten.

Zunächst wurde ein Objekterkennungsmodell trainiert, das in der Lage ist, Gebäude und Fahrzeuge aus den aufgenommenen Bildern und Videos zu erkennen. Diese Bilder und Videos wurden von zwei Rasbperry Pi's aufgenommen, die über der Modellwelt positioniert wurden. Durch Training des Modells war es möglich, die erkannten Objekte zu identifizieren und ihre Position zu bestimmen. Im nächsten Schritt konzentrierte sich die Arbeit darauf, die Farbe jedes erkannten Objekts zu bestimmen und die Ausrichtung des Objekts in einem bestimmten Winkel anzugeben. Dies ermöglichte eine genauere und realistischere Darstellung der Gebäude und Fahrzeuge im digitalen Zwilling. Im letzten Schritt wurde das entwickelte System in Unity implementiert. Dabei wurden die Ergebnisse des Objekterkennungsmodells und die Informationen über Farbe und Ausrichtung der Objekte in Unity integriert. Dadurch konnte der digitale Zwilling die realen Objekte in einer simulierten Umgebung darstellen und ein immersives Erleben der Modellwelt ermöglichen. Die Kombination aus Objekterkennung, Farberkennung, Berechnung der Rotationswinkel der Objekte und der Implementierung in Unity führte zu einem erfolgreichen digitalen Zwilling der Modellwelt im Maßstab 1:87, der eine realistische Darstellung der Gebäude und Fahrzeuge ermöglichte.

Im Kapitel Evaluation wurde die Umsetzung des digitalen Zwillings bewertet. Die Evaluation bestand aus zwei Teilen. Zuerst wurde das trainierte Objekterkennungsmodell analysiert. Die vorgestellten Ergebnisse zeigen, dass das Modell eine hohe Genauigkeit bei der Klassifizierung und Erkennung von Objekten erreicht. Die berechnete mittlere Genauigkeit (mAP50-95) lag bei ca. 90 %, was eine sehr gute Genauigkeit bei der

Lokalisierung der erkannten Objekte bedeutet. Darüber hinaus erreichte die mittlere Genauigkeit bei einem IoU-Schwellenwert von 0,5 (mAP50) einen hervorragenden Wert von 95 %, was auf eine nahezu perfekte Erkennung der Objekte hinweist. Diese Ergebnisse bestätigen die Effektivität und Effizienz des im digitalen Zwilling trainierten Objekterkennungsmodells.

Die Umsetzung des digitalen Zwillings in Unity erreichte unter Verwendung der definierten Regeln eine Genauigkeit von 82 %. In den meisten Fällen waren die Objekte korrekt positioniert und hatten die richtige Farbe. Es wurde jedoch festgestellt, dass die Ausrichtung der platzierten Objekte nicht immer genau war, was sich negativ auf die Gesamtgenauigkeit auswirkte.

Abschließend wurde im Kapitel „Reinforcement Learning im digitalen Zwilling“ ein Proof of Concept vorgestellt, der die Vorteile eines digitalen Zwillings veranschaulicht. Dabei wurde mit Hilfe von Reinforcement Learning ein Agent trainiert, der in der Lage ist, sich selbstständig innerhalb eines Parcours zu bewegen. Der vorgestellte Proof of Concept zeigt, dass der digitale Zwilling als Simulationsumgebung effektiv genutzt werden kann, um komplexe Szenarien zu trainieren und zu testen. Durch den Einsatz von Reinforcement Learning kann der Agent lernen, selbstständig Entscheidungen zu treffen und autonom in der simulierten Umgebung zu navigieren.

Der digitale Zwilling erzielt insgesamt ein gutes Ergebnis, da er sowohl bei der Objekterkennung als auch bei der Darstellung in Unity eine hohe Genauigkeit aufweist. Die erfolgreiche Objekterkennung ermöglicht eine präzise Lokalisierung und Darstellung von Gebäuden und Fahrzeugen aus der realen Modellwelt im digitalen Zwilling. Besonders bemerkenswert ist die dynamische Darstellung des digitalen Zwillings, der in der Lage ist, die Positionen der Objekte innerhalb der Szene im Laufe der Zeit zu aktualisieren. Dadurch kann der digitale Zwilling die Bewegungen der Objekte in eine Videosequenz verfolgen und ihre aktuellen Positionen und Orientierungen wiedergeben. Außerdem kann er Objekte, die aus dem Sichtfeld der Kameras in der Modellwelt verschwinden, aus der Szene entfernen, was zu einem realistischen und immersiven Erlebnis beiträgt.

6.0.1 Ausblick

Ein möglicher Ansatz zur Steigerung der Effizienz des digitalen Zwillings in Unity wäre die Verwendung von Objektsegmentierung anstelle von Objekterkennung. Durch die Objektsegmentierung könnten die Konturen der erkannten Objekte viel genauer erfasst

werden, was zu einer genaueren Berechnung des Rotationswinkels für die Ausrichtung führen würde. Außerdem wäre es möglich, für jedes segmentierte Objekt spezifische 3D-Modelle auf der Grundlage der erfassten Maske zu erstellen. Dieser Ansatz würde die Genauigkeit der Darstellung der Objekte im digitalen Zwilling erheblich verbessern. Allerdings ist zu beachten, dass die Segmentierung in der Regel ressourcenintensiver ist als die Objekterkennung. Dies liegt daran, dass die Bildsegmentierung zusätzliche Schritte erfordert, um die genauen Konturen der Objekte im Bild zu identifizieren und sie voneinander zu trennen. Dies kann die Rechenleistung und die Implementierungszeit erhöhen.

Ein weiterer Ansatz zur Verbesserung der Darstellung von Objekten im digitalen Zwilling wäre die Verwendung eines erweiterten Satzes von 3D-Modellen. Durch die Bereitstellung einer breiten Palette von 3D-Modellen in verschiedenen Größen und Variationen könnte sichergestellt werden, dass für die im digitalen Zwilling erkannten Gebäude realitätsnahe Entsprechungen in Unity gefunden werden können. Durch die Auswahl einer geeigneten 3D-Modellvariante für jedes erkannte Gebäude kann die Genauigkeit und Authentizität des digitalen Zwillings erheblich gesteigert werden. Die Vielfalt der verfügbaren 3D-Modelle ermöglicht es, Gebäude in unterschiedlichen Formen, Farben und Stilen darzustellen, die der realen Modellwelt entsprechen.

Um den digitalen Zwilling weiter zu verbessern, könnten zusätzliche Objekte aus der realen Modellwelt in den digitalen Zwilling integriert werden, wie z.B. Bäume, Straßenlaternen und Verkehrsschilder. Durch die Erfassung und Integration dieser zusätzlichen Objekte könnte der digitale Zwilling eine noch realistischere und lebendigere Darstellung der realen Modellwelt bieten. Dazu müsste das bestehende Objekterkennungsmodell weiter trainiert und erweitert werden, um die neuen Objekte zu erkennen und korrekt zu klassifizieren. Dies führt zu einem erweiterten Trainingsdatensatz, der Bilder und Videos mit den neuen Objekten enthält, um das Modell auf die neuen Klassen vorzubereiten.

Literaturverzeichnis

[1] Bertomeu, Jeremy ; Cheynel, Edwige ; Floyd, Eric ; Pan, Wenqiang: Using machine learning to detect misstatements. In: *Review of Accounting Studies* 26 (2021), Jun, Nr. 2, S. 468–519. – URL https://doi.org/10.1007/s11142-020-09563-8. – ISSN 1573-7136

[2] Bochkovskiy, Alexey ; Wang, Chien-Yao ; Liao, Hong-Yuan M.: *YOLOv4: Optimal Speed and Accuracy of Object Detection*. 2020

[3] Boyd, Kendrick ; Eng, Kevin H. ; Page, C. D.: Area under the Precision-Recall Curve: Point Estimates and Confidence Intervals. In: Blockeel, Hendrik (Hrsg.) ; Kersting, Kristian (Hrsg.) ; Nijssen, Siegfried (Hrsg.) ; Železný, Filip (Hrsg.): *Machine Learning and Knowledge Discovery in Databases*. Berlin, Heidelberg : Springer Berlin Heidelberg, 2013, S. 451–466. – ISBN 978-3-642-40994-3

[4] Documentation, Albumentations: *Albumentations Library*. albumentations.ai. – URL https://albumentations.ai/docs/getting_started/transforms_and_targets/

[5] Documentation, Cogniflow: *How to Create a Dataset for Object Detection using the YOLO Labeling Format*. cogniflow.ai. – URL https://docs.cogniflow.ai/en/article/how-to-create-a-dataset-for-object-detection-using-the-yolo-labeling-format-1tahk19/

[6] Documentation, OpenCV: *Creating Bounding rotated boxes and ellipses for contours*. opencv.org. – URL https://docs.opencv.org/3.4/de/d62/tutorial_bounding_rotated_ellipses.html

[7] Documentation, Unity: *ML-Agents Agent*. docs.unity3d.com. – URL https://docs.unity3d.com/Packages/com.unity.ml-agents@1.0/api/Unity.MLAgents.Agent.html#methods

[8] Documentation, Unity: *Unity Scripting API - MonoBehaviour*. docs.unity3d.com. – URL https://docs.unity3d.com/ScriptReference/MonoBehaviour.html

[9] Gavriluţ, Dragoş ; Cimpoeşu, Mihai ; Anton, Dan ; Ciortuz, Liviu: Malware detection using machine learning. In: *2009 International Multiconference on Computer Science and Information Technology*, 2009, S. 735–741

[10] Glenn Jocher, Ayush C.: *Yolov8 Detect*. ultralytics.com. – URL https://docs.ultralytics.com/tasks/detect/

[11] Glenn Jocher, Sergiu W.: *Object Detection Datasets Overview*. Ultralytics YOLOv8 Docs. – URL https://docs.ultralytics.com/datasets/detect/

[12] Glenn Jocher, Sergiu W.: *Yolov8 Detect - save_one_box()*. ultralytics.com. – URL https://docs.ultralytics.com/reference/yolo/utils/plotting/?h=save_one#save_one_box

[13] Juliani, Arthur ; Berges, Vincent-Pierre ; Teng, Ervin ; Cohen, Andrew ; Harper, Jonathan ; Elion, Chris ; Goy, Chris ; Gao, Yuan ; Henry, Hunter ; Mattar, Marwan ; Lange, Danny: Unity: A general platform for intelligent agents. In: *arXiv preprint arXiv:1809.02627* (2020). – URL https://arxiv.org/pdf/1809.02627.pdf

[14] Marom, Nadav D. ; Rokach, Lior ; Shmilovici, Armin: Using the confusion matrix for improving ensemble classifiers. In: *2010 IEEE 26-th Convention of Electrical and Electronics Engineers in Israel*, 2010, S. 000555–000559

[15] Matulis, Marius ; Harvey, Carlo: A robot arm digital twin utilising reinforcement learning. In: *Computers & Graphics* 95 (2021), S. 106–114. – URL https://www.sciencedirect.com/science/article/pii/S009784932100011X. – ISSN 0097-8493

[16] Ramesh, Shima ; Hebbar, Ramachandra ; M., Niveditha ; R., Pooja ; N., Prasad B. ; N., Shashank ; P.V., Vinod: Plant Disease Detection Using Machine Learning. In: *2018 International Conference on Design Innovations for 3Cs Compute Communicate Control (ICDI3C)*, 2018, S. 41–45

[17] Shah, Devansh ; Patel, Samir ; Bharti, Santosh K.: Heart Disease Prediction using Machine Learning Techniques. In: *SN Computer Science* 1 (2020), Oct, Nr. 6,

S. 345. – URL https://doi.org/10.1007/s42979-020-00365-y. – ISSN 2661-8907

[18] Tiedemann, Tim ; Schwalb, Luk ; Kasten, Markus ; Grotkasten, Robin ; Pareigis, Stephan: Miniature Autonomy as Means to Find New Approaches in Reliable Autonomous Driving AI Method Design. In: *Frontiers in Neurorobotics* 16 (2022). – URL https://www.frontiersin.org/articles/10.3389/fnbot.2022.846355. – ISSN 1662-5218

[19] Ultralytics: *YOLOv8*. ultralytics.com. – URL https://ultralytics.com/yolov8

[20] Wang, Ziran ; Han, Kyungtae ; Tiwari, Prashant: Digital Twin Simulation of Connected and Automated Vehicles with the Unity Game Engine. In: *2021 IEEE 1st International Conference on Digital Twins and Parallel Intelligence (DTPI)*, 2021, S. 1–4

[21] Yang, Yuanlin ; Meng, Wei ; Zhu, Shiquan: A Digital Twin Simulation Platform for Multi-rotor UAV. In: *2020 7th International Conference on Information, Cybernetics, and Computational Social Systems (ICCSS)*, 2020, S. 591–596

[22] Zheng, Xin ; Lei, Qinyi ; Yao, Run ; Gong, Yifei ; Yin, Qian: Image segmentation based on adaptive K-means algorithm. In: *EURASIP Journal on Image and Video Processing* 2018 (2018), Aug, Nr. 1, S. 68. – URL https://doi.org/10.1186/s13640-018-0309-3. – ISSN 1687-5281

Glossar

Bounding Box Ein rechteckiger Rahmen zur Kennzeichnung der Position und Ausdehnung eines bestimmten Objekts oder einer Region in einem Bild.

Box_loss Eine mathematische Funktion in der Objekterkennung, die den Unterschied zwischen der vorhergesagten und der tatsächlichen Position von Bounding Boxes für erkannte Objekte misst, um das Training eines Modells zu optimieren.

Cls_loss Eine mathematische Funktion in der Objekterkennung, die die Qualität der Klassifikation von Objekten misst, um das Training eines Modells zu optimieren.

Convolutional layer Eine Schicht in einem künstlichen neuronalen Netzwerk, die Faltungsoperationen auf Eingabedaten wie Bilder anwendet, um Merkmale zu extrahieren und die räumlichen Beziehungen zwischen Pixeln zu erfassen.

False Negative ein Element, das eigentlich zur Klasse "positiv" gehört, wird fälschlicherweise als "negativ" eingestuft.

False Positive ein Element, das eigentlich zur Klasse "negativ" gehört, wird fälschlicherweise als "positiv" eingestuft.

HAW Hamburg Die HAW Hamburg ist die vormalige Fachhochschule am Berliner Tor.

Hierarchie-Ansicht Fenster in der Unity-Entwicklungsumgebung, das eine strukturierte Liste aller Objekte in deiner aktuellen Szene anzeigt.

Inspector Fenster in der Unity-Entwicklungsumgebung zur Anzeige von Informationen und Einstellungen für ausgewählte Spielobjekte, Assets oder Komponenten.

Overfitting Wenn ein Modell so stark an die Trainingsdaten angepasst ist, dass es auf neuen, nicht gesehenen Daten schlechtere Leistungen zeigt.

Proof of Concept Ein experimenteller Nachweis, dass ein Konzept oder eine Technologie nützliche Vorteile haben kann.

Raspberry Pi Einplatinencomputer der Raspberry Pi Foundation.

True Negative ein Element, das zur Klasse "negativ" gehört, wird korrekt als "negativ" eingestuft.

True Positive ein Element, das zur Klasse "positiv" gehört, wird korrekt als "positiv" eingestuft.

Erklärung zur selbstständigen Bearbeitung

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen 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