Universität der Bundeswehr München
Fakultät für Luft- und Raumfahrttechnik
Institut für Autonomes Fahren

Erstellung und Evaluation virtueller Fahrszenarien in einer Simulationsumgebung zur Datengenerierung für KI-basierte autonome Fahrsysteme

Vincent Mann

Bachelorarbeit

Betreuer:	Leon Pohl, M.Sc.
Prüfer:	Univ.Prof. Dr.-Ing. Mirko Mählisch Dr.-Ing. Thorsten Lüttel
Abgabedatum:	28.04.2025

Hiermit erkläre ich, Vincent Mann, an Eides statt, dass ich die vorliegende Bachelorarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Neubiberg, den 28.04.2025

Aufgabenstellung

Bachelorarbeit für Herrn Vincent Mann, Matrikelnummer 1220857

Titel der Arbeit: **Erstellung und Evaluation virtueller Fahrszenarien in einer Simulationsumgebung zur Datengenerierung für KI-basierte autonome Fahrsysteme**
Abgabedatum: **28.04.2025**

Hintergrund

In den vergangenen Jahren hat sich die Entwicklung von autonomen Fahrsystemen deutlich weiterentwickelt. Waren früher klassische, modular aufgebaute Software-Stacks üblich – also klar getrennte Verarbeitungsketten für Wahrnehmung, Planung und Regelung – so setzen sich inzwischen zunehmend Methoden durch, die stark auf Künstlicher Intelligenz (KI) basieren. Diese neuen Ansätze reichen von hybriden Systemen, bei denen KI-Module einzelne Schritte des Prozesses übernehmen, bis hin zu End-to-End-Methoden, bei denen ein tiefes neuronales Netzwerk ohne explizite Zwischenstufen direkt vom Sensoreingang zum Steuerbefehl gelangt.

Um solche KI-basierten oder End-to-End-Modelle erfolgreich zu entwickeln und zu trainieren, werden jedoch große Mengen qualitativ hochwertiger Daten benötigt. Traditionell stammen diese Daten aus realen Fahrten unter Einsatz spezialisierter Sensorik. Dies ist jedoch kostspielig, zeitaufwendig und mit Risiken behaftet. Hier setzen virtuelle Simulationsumgebungen an, die es ermöglichen, realitätsnahe Szenarien synthetisch nachzubilden.

Eine vielversprechende Plattform für diese Art von Datengenerierung ist NVIDIA Drive Sim. Diese physikalisch basierte Simulationsumgebung erlaubt die konfigurierbare, hochrealistische Darstellung sowohl urbaner Fahrumgebungen (etwa ein Universitätscampus) als auch Offroad-Szenarien. Durch das Einbinden virtueller Versuchsträger (Ego-Fahrzeuge) und die Nutzung verschiedenster Sensorsimulationen (Kamera, LiDAR, Radar etc.) lassen sich große Mengen synthetischer Daten generieren. Auf Basis dieser Daten können KI-Modelle, etwa für Objekterkennung oder Szeneninterpretation, trainiert und evaluiert werden. Darüber hinaus ist es möglich, eine geschlossene Regelungsschleife aufzubauen, bei der das KI-Modell direkt Einfluss auf die Steuerung des Ego-Fahrzeugs in der Simulation nimmt, um Rückwirkungen des Modells im Fahrszenario zu untersuchen.

Aufgaben und Ausarbeitung

1. Literaturrecherche: Sichtung aktueller Forschung und industrieller Entwicklungen zu KI-basierten autonomen Fahrsystemen, End-to-End-Trainingsmethoden sowie virtuellen Fahrsimulationen.
2. Einarbeitung in NVIDIA Drive Sim: Untersuchung von Kernfunktionen, verfügbaren Sensor-Modellen, Simulationsparametern und vorhandenen Schnittstellen zur Integration externer Algorithmen.
3. Erstellung von realitätsnahen Szenarien im urbanen (z. B. Universitätscampus) und im Offroad-Bereich.
4. Generierung umfangreicher synthetischer Sensordaten für das Training von KI-Modellen.
5. Aufbau einer geschlossenen Regelungsschleife: Anbindung des KI-Outputs an die Steuerung des Ego-Fahrzeugs in Drive Sim.
6. Optional: Anwendung und Evaluation eines KI-Ansatzes (z. B. Objekt-Detektion) auf die synthetischen Daten.

Betreuer

- Leon Pohl, M.Sc., UniBw München, Institut für Autonomes Fahren, Professur Maschinelle Wahrnehmung für autonomes Fahren,
  `leon.pohl@unibw.de`, Tel. +49 89 6004-3286

Prüfer

- Univ.-Prof. Dr.-Ing. Mirko Mählisch
- Dr.-Ing. Thorsten Lüttel

Kurzfassung

Die vorliegende Arbeit untersucht die Simulationsumgebung NVIDIA DriveSim im Hinblick auf ihre Eignung zur Generierung synthetischer Bilddaten für KI-basierte autonome Fahrsysteme. Der Schwerpunkt liegt auf der realitätsnahen Darstellung von Verkehrsszenarien sowie der Verwendbarkeit der erzeugten Daten zum Training von Objekterkennungsnetzwerken. Hierzu wurden Szenarien in E2E erstellt, mit Replicator synthetische Bilddaten erzeugt und ein neuronales Netz (YOLOv11) ausschließlich auf diesen Daten trainiert. Der anschließende Test auf realen Bildern diente zur Bewertung einer möglichen Sim-to-Real-Gap. Zusätzlich wurde das Verhalten der simulierten Verkehrsteilnehmer qualitativ analysiert.

Die Ergebnisse zeigen, dass DriveSim in Bezug auf die erzeugte Datenrealität und die Handhabung innerhalb der Simulationsumgebung klare Schwächen aufweist. Dazu zählen unter anderem instabile Writer bei der Datenausgabe, eine fehlende Unterstützung für Regen oder Schnee sowie eine unzureichende Dokumentation zentraler Funktionen (z. B. ROS-Anbindung, Karteneinbindung). Diese Faktoren erschweren die verlässliche Nutzung erheblich.

Zwar bietet DriveSim durch seine physikbasierte Modellierung, sein realistisches Sensormodell und die Einbindung in Omniverse eine leistungsfähige Grundlage für industrielle Anwendungen, insbesondere in Großunternehmen. Für kleinere universitäre Forschungsteams stellt die Plattform jedoch eine Herausforderung dar, da diese typischerweise nicht über die personellen und infrastrukturellen Ressourcen verfügen, um technische Schwächen eigenständig zu beheben oder fehlende Funktionalitäten zu ersetzen.

Abstract

The present thesis investigates the simulation environment NVIDIA DriveSim with regard to its suitability for generating synthetic image data for AI-based autonomous driving systems. The focus lies on the realistic representation of traffic scenarios and the usability of the generated data for training object detection networks. For this purpose, scenarios were created in E2E, synthetic image data was generated using Replicator, and a neural network (YOLOv11) was trained exclusively on this data. The subsequent test on real images served to evaluate a possible Sim-to-Real gap. Additionally, the behavior of the simulated traffic participants was analyzed qualitatively.

The results show that DriveSim exhibits clear weaknesses in terms of the realism of the generated data and its handling within the simulation environment. These include, among others, unstable writers during data export, lack of support for rain or snow, and insufficient documentation of key functions (e.g., ROS integration, map import). These factors significantly hinder reliable use.

While DriveSim offers a powerful foundation for industrial applications due to its physics-based modeling, realistic sensor framework, and integration with Omniverse—especially for large corporations—the platform presents a challenge for smaller university research teams, as they typically lack the personnel and infrastructure to independently resolve technical shortcomings or compensate for missing functionalities.

Inhaltsverzeichnis

Aufgabenstellung **iii**
Abbreviations / Abkürzungen 2

1 Einleitung **3**

2 Stand der Technik und Grundlagen **5**
2.1 Entwicklung autonomer Fahrsysteme 5
2.2 Simulationsplattformen 5
2.3 World Models – Beispiel GAIA-2 7
2.4 Werkzeuge und Infrastruktur 8

3 Methodik **9**
3.1 Ziele und Motivation 9
3.2 Evaluationsansatz 9
3.3 Datengenerierung und Netzwerkauswertung 9
3.4 Bewertungsfaktoren in der Simulation 10

4 Implementierung **11**
4.1 E2E 11
4.2 Replicator 15
4.3 Postprocessing – Voxel51 16
4.4 Training eines neuronalen Netzwerks zur Objekterkennung 17

5 Evaluierung **19**
5.1 Realismus des Verhaltens der Verkehrsteilnehmer 19
5.2 Realismus der von Kameras aufgenommenen Bilddaten 22
5.2.1 Optische Eigenschaften 22
5.2.2 Vielfalt & Komplexität der Szenen 24
5.2.3 Realwelt-Evaluation eines simulativ trainierten YOLO-Netzwerks 24

6 Zusammenfassung **29**

A Anhang **31**
A.1 Python Skripte 31
A.2 Tabellen 41
A.3 Abbildungen 45

Literaturverzeichnis **51**

Abbreviations / Abkürzungen

AI Artificial Intelligence

AV Autonomous Vehicle

CARLA Car learning to act

E2E End-to-End

FOV Field of View

GAIA Generative Autonomous Imitation Agent

GPU Graphics Processing Unit

LiDAR Light Detection and Ranging

NGC NVIDIA GPU Cloud

RL Reinforcement Learning

ROS Robot Operating System

SDK Software Development Kit

USD Universal Scene Description

YOLO You Only Look Once

1 Einleitung

Der Fortschritt im Bereich des autonomen Fahrens hängt maßgeblich von der Verfügbarkeit geeigneter Trainings- und Testdaten ab. Insbesondere datengetriebene Systeme wie neuronale Netze benötigen große Mengen annotierter Bilddaten, um zuverlässig in komplexen Verkehrssituationen agieren zu können. Die manuelle Erfassung dieser Daten aus realen Fahrumgebungen ist jedoch mit erheblichem Aufwand, hohen Kosten und sicherheitskritischen Risiken verbunden.

Virtuelle Simulationsumgebungen versprechen hier eine effiziente und skalierbare Alternative. Plattformen wie NVIDIA DriveSim ermöglichen die Generierung synthetischer Sensordaten unter kontrollierten Bedingungen, bei gleichzeitiger Ausgabe detaillierter Annotationen (z.B. Bounding Boxes oder Klassenzugehörigkeit). Damit könnten sie einen wesentlichen Beitrag zur kostengünstigen und flexiblen Entwicklung autonomer Fahrsysteme leisten. Zugleich stellen sich aber Fragen hinsichtlich der Realitätsnähe und Übertragbarkeit solcher synthetischer Daten auf reale Einsatzszenarien.

Ziel der vorliegenden Untersuchung ist die Bewertung synthetischer Kameradaten sowie des Verkehrsverhaltens innerhalb der Simulation, um die Eignung der Plattform für die Entwicklung autonomer Fahrfunktionen einzuordnen. Der Schwerpunkt liegt dabei auf einem objektiven Vergleich zwischen simulierten und realen Daten durch den Einsatz eines auf Simulatorbildern trainierten Objekterkennungsnetzwerks. Darüber hinaus wird das Verhalten der Verkehrsteilnehmer in der Simulation qualitativ analysiert, um Rückschlüsse auf die Plausibilität der Umgebungsmodellierung zu ziehen.

Die Relevanz dieser Fragestellung ergibt sich insbesondere aus der Problematik realweltlicher Datenbeschaffung: Nicht nur sind hochaufgelöste Kameradaten teuer zu erfassen und zu annotieren, sondern sie unterliegen auch strengen Datenschutzauflagen, besonders im öffentlichen Straßenraum. Simulationen wie DriveSim ermöglichen es hingegen, beliebige Verkehrssituationen unter Einhaltung vollständiger Kontrolle zu erzeugen – mitsamt automatisch generierter Labelinformationen. Damit eröffnen sich neue Möglichkeiten für Training, Validierung und schnelles Prototyping im Bereich KI-gestützter Fahrfunktionen.

Die Arbeit gliedert sich wie folgt: Kapitel 2 stellt den Stand der Technik sowie die eingesetzten Werkzeuge vor, insbesondere die Simulationsplattform DriveSim, den Open-Source-Simulator CARLA sowie das generative Modell GAIA-2. In Kapitel 3 wird das methodische Vorgehen zur Bewertung der Simulation beschrieben. Die Umsetzung der entwickelten Methodik in DriveSim sowie die durchgeführten Testschritte sind Inhalt von Kapitel 4. Die Ergebnisse der Evaluation und die Einordnung im Hinblick auf die Sim-to-Real-Übertragbarkeit folgen in Kapitel 5. Kapitel 6 fasst abschließend die gewonnenen Erkenntnisse zusammen und gibt einen Ausblick auf weiterführende Fragestellungen.

2 Stand der Technik und Grundlagen

Die Entwicklung autonomer Fahrsysteme ist in den letzten Jahren stark von Fortschritten im Bereich künstlicher Intelligenz und realistischer Simulationssysteme geprägt worden. Der Übergang von regelbasierten zu datengetriebenen Modellen verlangt nach neuen Methoden zur Datengenerierung, -analyse und -validierung. Dieses Kapitel gibt einen Überblick über den technologischen Wandel, relevante Simulationsplattformen sowie die in dieser Arbeit eingesetzten Software-Tools.

2.1 Entwicklung autonomer Fahrsysteme

Frühere autonome Fahrsysteme basierten auf klassischen Regelungstechniken und fest kodierten Entscheidungslogiken. Sensordaten wurden nach festgelegten Regeln verarbeitet. Diese Systeme waren jedoch wenig flexibel und skalierten schlecht mit wachsender Umweltkomplexität.

Mit dem Aufkommen leistungsfähiger Hardware und wachsender Datenverfügbarkeit konnten neuronale Netzwerke für Aufgaben wie Objekterkennung, Spurverfolgung und Planung eingesetzt werden. Moderne Self-Driving-Stacks bestehen häufig aus modularen KI-Komponenten und lernen aus großen Mengen annotierter Daten – insbesondere aus Bilddaten.

2.2 Simulationsplattformen

Simulatoren bieten eine kontrollierte, skalierbare und sichere Umgebung für das Training, die Validierung und das Testen autonomer Systeme. Neben dem Erzeugen realitätsnaher Fahrszenarien ermöglichen sie auch die Ausgabe synthetischer Sensordaten mit exakten Labels. Zwei besonders relevante Plattformen sind NVIDIA DriveSim und CARLA.

NVIDIA DriveSim

NVIDIA DriveSim ist ein High-Fidelity-Simulator für autonomes Fahren, der auf der Omniverse-Plattform basiert. Die Plattform bietet eine umfassende Umgebung für die Erstellung, Simulation und Analyse von Verkehrsszenarien und KI-Trainingsdaten [1]. Drei zentrale Komponenten sind:

E2E

Die Komponente E2E dient der interaktiven Erstellung von Fahrszenarien sowie Ego-Fahrzeugen [2]. Verkehrsverhalten wird über Verhaltenbäume (ScenarioGraphs) gesteuert. Karten basieren auf dem OpenDRIVE-Standard (.xodr) und beinhalten vollständige Spurinformationen und Verkehrsregeln.

Replicator

Mit Replicator lassen sich aus erstellten Szenarien großflächig synthetische Bilddaten mit Annotationsinformationen (z.B. Bounding Boxes, Segmentation Maps) generieren [3]. Die Ausgaben eignen sich für das Training neuronaler Netze. Der typische Workflow ist in Abbildung 2.1 dargestellt.

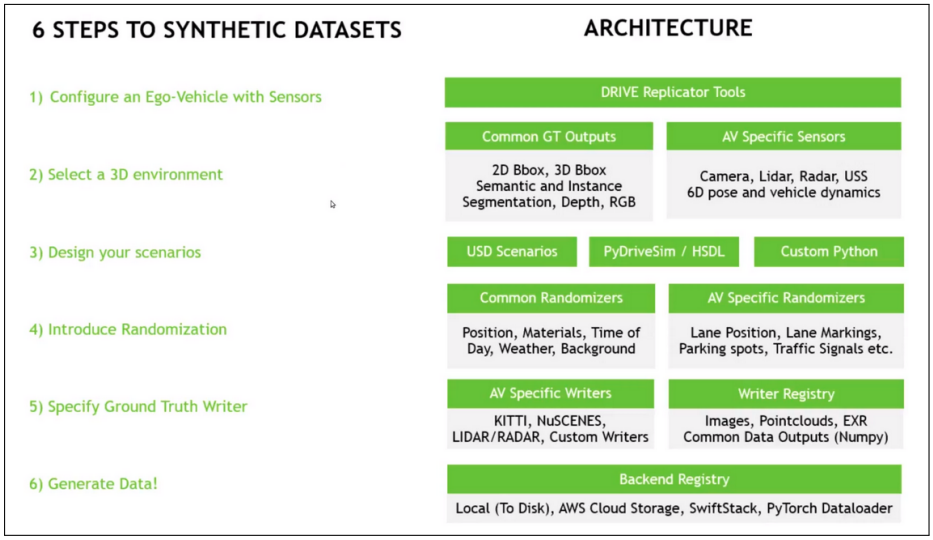


Abbildung 2.1: *Übersicht des Replicator-Workflows: Schritt 1–3 (Szenarioerstellung) erfolgt in E2E, Schritt 4–6 (Randomisierung, Datenausgabe) über Replicator.*

SDK

Das SDK erlaubt erweiterte Eingriffe in die Simulation, etwa durch benutzerdefinierte Sensoren, Netzwerkanbindungen oder neue Fahrzeugdynamiken [4].

CARLA

CARLA ist ein frei verfügbarer Open-Source-Simulator für autonomes Fahren, entwickelt vom Computer Vision Center (CVC) in Barcelona [**?**]. Die Plattform ist vollständig ROS-kompatibel (ROS1/ROS2) und unterstützt komplexe Sensor-Setups, darunter Kameras, LiDAR und Radar.

Architektur: CARLA basiert auf der Unreal Engine (derzeit in Version UE5) und lässt sich über eine Python-API vollständig steuern. Szenarien werden in Form von OpenDrive-basierten Karten (z.B. RoadRunner) definiert und mit Agenten, Verkehrsregeln und Umwelteinflüssen ergänzt.

Funktionen:

- Echtzeitsteuerung von Agenten über Python oder ROS
- Realistische Sensoren: Kamera, LiDAR, Radar, GNSS
- ROS-Bridge für Integration in bestehende AV-Stacks wie Autoware
- Tools wie `scenario_runner` und `driving-benchmarks` für Tests und Evaluation

CARLA eignet sich besonders für Forschung, Prototyping und das Benchmarking von Modellen. Die aktive Community, umfangreiche Dokumentation und freie Lizenzierung machen es zu einer attraktiven Alternative für Forschungseinrichtungen.

2.3 World Models – Beispiel GAIA-2

World Models sind generative KI-Systeme, die durch Lernverfahren eine interne Repräsentation von Umgebungen und Dynamik erzeugen. Anstelle klassischer Simulationspipelines erzeugen sie Videosequenzen basierend auf strukturierten Eingaben wie Fahrdaten, Umgebungsbedingungen und Agentenverhalten.

GAIA-2 von Wayve stellt ein solches multimodales World Model dar, basierend auf Latent Diffusion [5]. Es kann Videos aus fünf Perspektiven gleichzeitig erzeugen und verarbeitet komplexe Steuerungs- und Umgebungsparameter wie:

- Fahrverhalten: Geschwindigkeit, Lenkung, Fahrzeugtrajektorie
- Szeneneigenschaften: Wetter, Tageszeit, Ort, Kreuzungen
- Agentenzustände: Position, Geschwindigkeit, Bounding Boxes
- Kameraparameter: Blickwinkel, Synchronisierung, Multiview-Kohärenz

GAIA-2 wurde mit mehr als 25 Millionen realen Fahrvideos aus den UK, USA und Deutschland trainiert. Die Modellarchitektur erlaubt Szenenmodifikation und Steuerung über Textprompts und ist damit flexibler als klassische Simulatoren.[6]

Ein limitierender Faktor ist aktuell, dass keine annotierten Sensordaten erzeugt werden können – etwa Bounding Boxes oder Segmentationen. Für das Training von Erkennungsnetzwerken ist GAIA-2 daher (noch) nicht direkt einsetzbar, zeigt jedoch das Potenzial generativer Ansätze zur Realitätsdarstellung in der AV-Forschung.

2.4 Werkzeuge und Infrastruktur

Omniverse

Omniverse ist die zugrunde liegende Entwicklungsplattform von NVIDIA. Das USD-Format (Universal Scene Description) erlaubt modulare Szenendefinition und Asset-Sharing. DriveSim verwendet Omniverse zum Zugriff auf Szenen, Asset-Pakete und Server-Streaming.

OpenDRIVE

OpenDRIVE definiert im XML-Format Straßenverläufe, Verkehrsinfrastruktur und Metainformationen wie Steigung oder Kurvigkeit [7]. Die meisten Simulatoren – darunter DriveSim und CARLA – nutzen OpenDRIVE für ihre Szenengeometrie.

Docker

Docker ermöglicht die isolierte und reproduzierbare Ausführung von Software in sogenannten Containern [8]. DriveSim wird vollständig in Containerform ausgeliefert. Änderungen an der Umgebung wurden über Visual Studio Code mit den Extensions „Docker“ und „Dev Containers“ vorgenommen.

Voxel51

Voxel51 (FiftyOne) dient als Postprocessing-Tool zur Analyse synthetischer Datensätze. Es bietet interaktive Visualisierung und Filterfunktionen sowie die Möglichkeit, Datensätze zu exportieren und für Trainingspipelines vorzubereiten [9].

Ultralytics – YOLO

YOLO (You Only Look Once) ist ein Framework zur Objekterkennung in Echtzeit. Ultralytics stellt eine moderne, gut dokumentierte Python-Implementierung mit vortrainierten Gewichten bereit [10]. YOLO wurde in dieser Arbeit verwendet, um die Qualität der DriveSim-Daten im Vergleich zu realen Bildern zu bewerten.

3 Methodik

In diesem Kapitel wird das methodische Vorgehen zur strukturierten Bewertung von NVIDIA DriveSim erläutert. Der Fokus liegt auf der Definition der Ziele, des Evaluationsrahmens sowie der eingesetzten Werkzeuge zur Datengenerierung und Ergebnisanalyse.

3.1 Ziele und Motivation

Diese Arbeit verfolgt das Ziel, die Simulationsumgebung NVIDIA DriveSim daraufhin zu evaluieren, ob sie als Datengenerator für Trainings- und Testdaten KI-basierter Fahrfunktionen geeignet ist. Hintergrund ist der Bedarf an flexiblenund realitätsnahen Simulationsplattformen

Zentrales Untersuchungsziel ist es, zu erkennen, ob die mit DriveSim erzeugten Daten qualitativ und strukturell geeignet sind, ein neuronales Netz zur Objekterkennung zu trainieren, das anschließend zuverlässig auf reale Bilddaten angewendet werden kann. Zusätzlich soll analysiert werden, ob das Verhalten der Verkehrsteilnehmer in der Simulation plausibel und realitätsnah erscheint.

3.2 Evaluationsansatz

Die Evaluation erfolgt zweiseitig:

- **Verhaltensbewertung:** Analyse des simulierten Verkehrsverhaltens
- **Bilddatenbewertung:** Prüfung der Realitätsnähe der erzeugten Kamerabilder über den Umweg der Netzauswertung (Sim2Real-Vergleich).

Beide Aspekte sollen Bewerten, ob DriveSim in der Lage ist, sowohl die Umgebung als auch den Verkehr realitätsgetreu abzubilden.

3.3 Datengenerierung und Netzwerkauswertung

Die Bilddaten wurden mithilfe von Replicator erzeugt. Dabei wurden vordefinierte sowie manuell angepasste Szenarien genutzt. Die Ausgabe erfolgte in Form annotierter 2D-Bilder mit Bounding Boxes.

Zur objektiven Bewertung wurde ein neuronales Netz (YOLOv11) gewählt und ausschließlich auf diesen Simulationsdaten trainiert. Die Wahl fiel auf YOLO aufgrund seiner seiner Zugänglichkeit und der einfachen Integration.

Das trainierte Modell wurde anschließend mit realen Bildern getestet. Durch diesen Vergleich lassen sich Rückschlüsse auf die sogenannte *Sim-to-Real-Gap* ziehen – also die Generalisierungsfähigkeit eines auf synthetischen Daten trainierten Modells.

3.4 Bewertungsfaktoren in der Simulation

Die Teilbereichen der Evaluation wurden auf folgende Merkmale analysiert:

- **Verhalten:** Einhaltung von Verkehrsregeln, Interaktion mit anderen Agenten, Umfahrung von Hindernissen, Reaktion auf Ampeln und Schilder.
- **Kamera-Output:** Bildqualität, Lichtverhalten, Reflexionen, Wettereffekte, Variabilität von Szenarien.
- **Datennutzen:** Netzwerkgüte, Sichtbarkeit des Sim2Real-Gaps, Generalisierbarkeit.

Die Auswertung der Ergebnisse erfolgt im späteren Kapitel 5. Dort wird geprüft, inwiefern die methodisch gesetzten Kriterien im praktischen Einsatz erfüllt wurden.

4 Implementierung

In diesem Kapitel wird auf Basis der in Kapitel 3 beschriebenen Methodik die technische Umsetzung in der Simulationsumgebung NVIDIA DriveSim durchgeführt. Der Workflow umfasst die manuelle Erstellung von Fahrszenarien in E2E, die automatisierte Datengenerierung mit Replicator, das Postprocessing mit Voxel51 sowie das Training eines neuronalen Netzes (YOLOv11). Die Umsetzung erfolgte innerhalb eines vordefinierten Docker-Containers unter Verwendung bereitgestellter NVIDIA-Tools und -Skripte.

Die folgenden Abschnitte beschreiben die technische Durchführung chronologisch gegliedert nach den Hauptkomponenten des Systems.

4.1 E2E

Nach erfolgreicher Installation kann E2E über folgenden Befehl im Terminal gestartet werden:

```
bash
nvdsl run -m e2e  # Startet DriveSim mit dem zuletzt verwendeten Container.
                  # F r weitere Optionen: "nvdsl run -h"
```

Nach dem erfolgreichen Start lässt sich im neuen Fenster über „Create Scenario“ ein neues Szenario erstellen bzw. über „Open/Import“ ein bereits vorhandenes importieren oder öffnen. Über „Create new Ego Config“ können neue Ego-Fahrzeuge erstellt werden.

Erstellen eines Szenarios

Nach dem Klick auf „Create Scenario“ öffnet sich ein neues Fenster. Dort kann eine der vorhandenen Karten ausgewählt werden (eigene Karten siehe Abschnitt 4.4). Danach lässt sich auswählen, welche Teile bzw. Payloads der Karte mitgeladen werden sollen.

Nach dem Laden der Karte (siehe Abbildung 4.1) kann man sich in der Szene folgendermaßen bewegen: Verschieben mit gedrückter mittlerer Maustaste, Zoomen mit mittlerer Maustaste, Rotieren mit rechter Maustaste. Objekten kann mit „F“ gefolgt werden [11].

Abbildung 4.1: *Die linke Werkzeugleiste enthält die wichtigsten Werkzeuge. Mit „Select (Q)“, „Move (W)“, „Rotate (E)“ und „Scale (R)“ können platzierte Objekte verschoben, rotiert und skaliert werden. Über Rechtsklick auf „Move (W)“ kann zwischen dem straßengebundenen Modus („Roadware“) und freien Koordinatensystemen („Local/Global“) umgeschaltet werden.*

Neue Objekte werden über die „Creation“-Funktion in der linken Leiste hinzugefügt. Unter dem Reiter „Placer“ sind alle verfügbaren Modelle wie Fahrzeuge, Fußgänger, statische Objekte sowie Ego-Fahrzeuge zu finden (siehe Abbildung 4.2) [12].

Abbildung 4.2: *Objekte können per Drag-and-Drop platziert werden. Die Positionierung kann erneut über das Menü oberhalb der Suchleiste zwischen Straßenbezug und Koordinatensystem umgeschaltet werden.*

Mit der Funktion „Waypoint Graphs“ lassen sich Wegpunkt-Grafen erzeugen, falls Verkehrsteilnehmer bestimmte Routen nehmen sollen. Zusätzlich können über „Sun“ und „Weather“ Sonnenstand, Wolkenbedeckung und Nebel konfiguriert werden. Die Menüs „Driver Interactions“, „Intersections“ und „Data Studio“ wurden aufgrund fehlender Funktionalität in dieser Arbeit nicht berücksichtigt.

Ein platziertes Fahrzeug erscheint im rechten Menü „DriveSim Scenario Outliner“ unter /Scenarios/Entity/Vehicle (siehe Abbildung 4.3) und kann dort selektiert werden. Im darunterliegenden „Property“-Menü lassen sich Eigenschaften wie Farbe, Startgeschwindigkeit oder Aggressivität anpassen (siehe Abbildung 4.4).

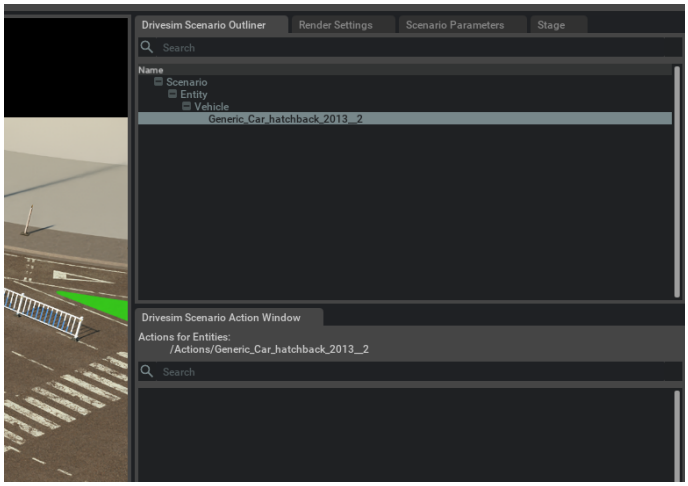


Abbildung 4.3: *Alternativ kann die Auswahl über die Leiste „Select (Q)“ erfolgen. Die zusätzlichen Menüs „Render Settings“, „Scenario Parameters“ und „Stage“ erlauben globale Anpassungen wie Raytracing, Lichtverhältnisse usw. [13]*

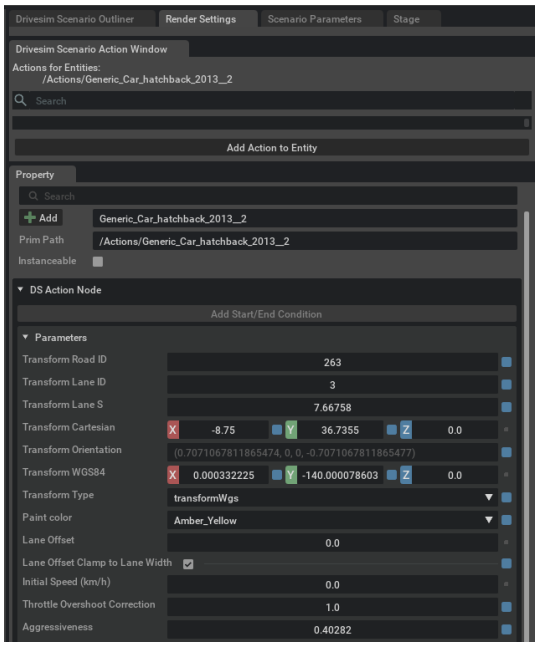


Abbildung 4.4: *Das übergeordnete „DriveSim Scenario Action Window“ erlaubte früher globale oder lokale Aktionen wie das Anbringen von Sensoren. Diese Funktion wurde inzwischen durch „ScenarioGraphs“ ersetzt.*

Die Steuerung eines gesamten Szenarios erfolgt über sogenannte „ScenarioGraphs“. Diese können per Rechtsklick auf „Scenario“ im Outliner und Auswahl von „Create Scenario Graph“ erstellt werden. Ein Doppelklick öffnet das zugehörige Interface (siehe Abbildung 4.5).[14]

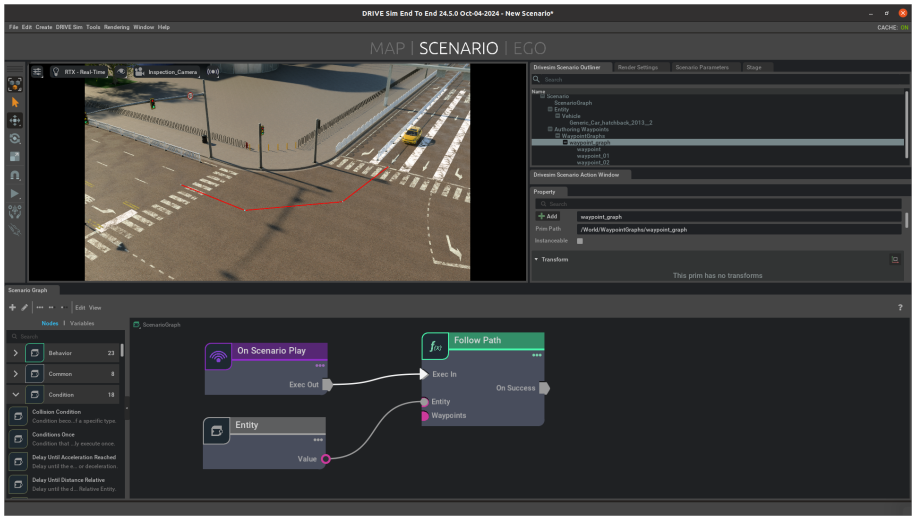


Abbildung 4.5: *Links finden sich Drag-and-Drop-Bausteine für Aktionen, Entscheidungen, Logik etc., die über Ein-/Ausgänge verknüpft werden können. Die Eigenschaften einzelner Bausteine lassen sich im „Property“-Menü anpassen. Gezeigt wird hier ein Auto, das bei Szenariostart einem Wegpunkt-Grafen folgt.*

Szenarien werden über „File“ → „Save As“ lokal oder auf dem Omniverse-Server gespeichert. Der Szenariotest erfolgt über den „DriveSim“-Reiter im oberen Menü.

Erstellen eines Ego-Fahrzeugs

Die Auswahl von „Create a new Ego Config“ verlangt zunächst die Auswahl eines Fahrzeugmodells in Form der Datei main_with_physx.usda (zu finden unter /Projects/dsready_content/nv_content/common_assets/vehicles/). Danach wird die Konfiguration unter einem neuen Namen im /Projects-Ordner gespeichert [15].

Nach dem Neustart erscheint das Fahrzeug unter „Placer → Ego Config Vehicle“ („Show Experimental“ aktivieren). Nach dem Platzieren kann im Menü „Ego“ das Fahrzeug ausgewählt werden. Im Menü „Ego Outliner“ kann per Rechtsklick auf „Sensor Rig“ über „Add Sensor“ ein Sensor hinzugefügt werden.

Alle Basissensoren befinden sich im Container unter /drivesim-ov/_build/linux-x86_64/release/assets/rig_lib/sensors/ (siehe Tabelle A.4 und [16]).

Sensorpositionen können über das „EgoProperty Panel“ angepasst werden (siehe Abbildung 4.6).

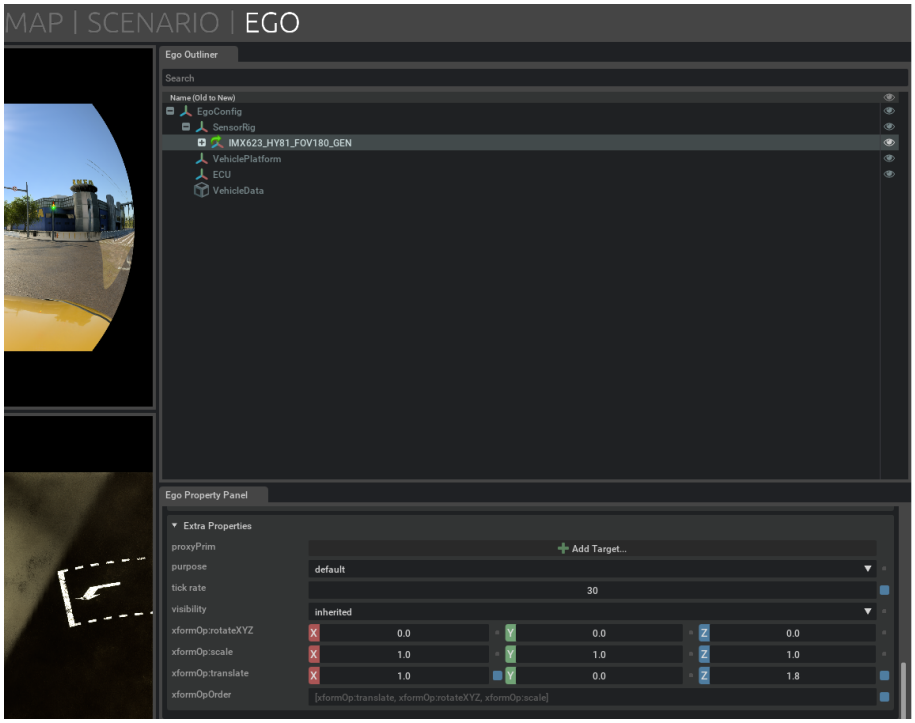


Abbildung 4.6: *Im Property Panel lassen sich Sichtfeld (FOV), Auflösung, Kameraart usw. anpassen.*

Zum Speichern stehen zwei Möglichkeiten zur Verfügung: „Save“ speichert das komplette Fahrzeug mit aktueller Konfiguration. Alternativ kann das SensorRig separat gespeichert werden („Save as Sensor Rig“). Bereits gespeicherte Rigs lassen sich über „Add Sensor Rig“ wieder zuweisen.

4.2 Replicator

Replicator wird über das Skript DRIVESim_launch.sh gestartet (siehe Abschnitt A.3). Nach dem erfolgreichen Start sollte das Terminal mit „drivesim@<user>“ beginnen. Replicator arbeitet headless, also ohne grafische Oberfläche. Gestartet wird der Prozess über generate_SDG_headless.sh. Die eigentliche Grundlage bilden jedoch die Python-Skripte generate_SDG_headless.py und generate_SDG_headless_args.py, welche sich alle im Container unter /drivesim-ov/ befinden.

Das Skript generate_SDG_headless.py ist der Hauptprozess zur Datengenerierung. Es greift auf vordefinierte Basisszenarien zurück (siehe Tabelle A.2). Die Parameter werden über generate_SDG_headless_args.py übergeben. Ein typischer Startbefehl sieht wie folgt aus:

Listing 4.1: *Startbefehl für das Basisszenario „sign“*

```
./generate_SDG_headless.sh --scenario=sign --num-frames=50 -w=BasicDSWriterFull
    -q=Ultra --nucleus-path=omniverse://137.193.76.216
```

Dieser Befehl startet Replicator mit dem Szenario „sign“. Es werden 50 Bilder generiert, mit dem Writer „BasicDSWriterFull“, in der Qualitätsstufe „Ultra“, und die Daten werden auf den angegebenen Omniverse-Server geschrieben.

Ein Beispiel mit vollem Pfad zu einem eigenen Szenario und zusätzlichen Parametern:

Listing 4.2: *Startbefehl mit voller Konfiguration*

```
./generate_SDG_headless.sh --scenario=/app/drivesim/defaultNucleusRoot/Projects
    /ds2_scenarios/sdg/preset_scenes/Example.usda --num-frames=50 -w=
    BasicDSWriterFull --rand-config=ExampleConfig.toml --ran-config-dir=/app/
    drivesim/defaultNucleusRoot/Projects/ds2_scenarios/sdg/preset_scenes/
    ExampleConfigFolder/ --ran-assetdir=/app/drivesim/defaultNucleusRoot/
    Projects/ds2_scenarios/sdg/preset_scenes/ExampleAssets/ -q=ultra --nucleus-
    path=omniverse://137.193.76.216
```

Die wichtigsten zusätzlichen Argumente sind:

1. `-rand-config`: Verweist auf eine Konfigurationsdatei zur Szenenrandomisierung.
2. `-ran-config-dir`: Definiert den Pfad zur Konfigurationsdatei.
3. `-ran-assetdir`: Pfad zu den darin referenzierten Assets.

Diese Parameter sind zentral für die synthetische Datengenerierung (siehe A.4, A.5 und [17]).

Die erzeugten Bilder werden im Container unter /tmp/replicator/out/<Writer-Name> gespeichert. Der gesamte Prozess dauert je nach Szenario und Hardware einige Minuten.

Der Pointcloudwriter führe zu Abstürzen. Es scheint hier Probleme mit dem richtigen Dartsellen der Daten zu geben. Zusätzlich kommt es wiedeholt zu Dependancy Problemen mit der Radar- und Lidar-Sensorik. Derzeit können nur Kameradaten sicher verarbeitet werden. Für alle Ergebnisse wurde der Writer BasicDSWriterFull verwendet.

4.3 Postprocessing – Voxel51

Die erzeugten Daten liegen in Formaten vor, die nicht direkt weiterverwendet werden können. Für die Umwandlung wurde das Script run_post_process.sh verwendet, das sich bereits im Container befindet [18]. Es konvertiert u.a. die Ausgaben von BasicDSWriterFull in das Voxel51-Format.

Das zugehörige Modul fiftyone_core.py befindet sich unter /drivesim-ov/source/python/scripts/sdg_po (siehe Anhang A.7). Für die korrekte Verarbeitung von Bounding Boxes musste zudem das Skript basic_ds_parser.py ersetzt werden (siehe A.6).

Zur Umwandlung in das YOLO-Format wurde ein separates Python-Skript eingesetzt (siehe Anhang A.8), welches die Daten in .jpg-Bilder und .txt-Annotationen exportiert (siehe auch A.9).

Die Konvertierung erfolgt auf einem externen Rechner mit installiertem Voxel51-Environment [19]. Eine direkte Umwandlung ist ebenfalls möglich (siehe Übersicht unterstützter Formate in [20]).

4.4 Training eines neuronalen Netzwerks zur Objekterkennung

Zur Überprüfung der Trainingsfähigkeit der Daten wurden vortrainierte YOLO-Netze erneut auf dem erzeugten Datensatz trainiert. Das verhätniss zwischen Trainingsbilder und validierungsbildern beträgt 80% zu 20%. Ziel ist es, wie angesprochen die Kernfunktionalität von Replicator als synthetische Trainingsdatenquelle zu evaluieren und das Bewerten der Sim-Real Gap durch anschließende Konfrontation mit Realbilder.

Zunächst wurde eine virtuelle Programmierumgebung mit Python 3.11 oder 3.12 erstellt:

Listing 4.3: *Einrichtung der Python-Umgebung*

```
python -m venv yolo-env          # Erstellt das Environment im Ordner 'yolo-env'
source yolo-env/bin/activate     # Linux/macOS
yolo-env\Scripts\activate        # Windows
```

Dann wurden alle Abhängigkeiten installiert:

Listing 4.4: *Installation von Ultralytics*

```
pip install ultralytics
```

Das Training erfolgte mit folgendem Skript:

Listing 4.5: *YOLO-Trainingsskript*

```
from ultralytics import YOLO

# Initialisiere YOLO mit dem vortrainierten 'Medium'-Modell
model = YOLO('yolo11m.pt')

# Trainiere das Modell mit dem bereitgestellten Datensatz
model.train(
    data='/home/tas/Desktop/Yolo_Enviroment/MergedDataset/dataset.yaml',  #
        Pfad zur dataset.yaml-Datei
    epochs=100,          # Anzahl der Trainingsdurchl ufe
    imgsz=640,           # Zielbildgr  e (Skalierung auf 640x640)
    batch=32,            # Batch-Gr   e (Bilder pro Trainingsschritt)
    device=0,            # GPU-Index (0 = erste GPU)
    workers=8,           # Anzahl paralleler Prozesse zum Datenladen
    classes=[0],         # Nur Klasse 0 trainieren (z. B .     Automobile    )
    project='runs/detect',  # Ausgabeordner
    name='<Name>',       # Name des Versuchs (Ordnerbezeichnung)
    verbose=True         # Ausgabe detaillierter Trainingslogs
)
)
```

Um Rechenzeit zu verkürzen und die Bilder zu entschlacken, wurden nur die 2D-Bounding-Boxes von `automobile` und `road_sign` übernommen. Das besttrainierte Modell wird automatisch in runs/detect/<Name>/weights/best.pt gespeichert. Dieses kann mit realen Testbildern evaluiert werden:

Listing 4.6: *Test des trainierten Modells*

```
# Laden des trainierten Modells
model = YOLO('runs/detect/<Name>/weights/best.pt')

# Durchf hrung einer Vorhersage auf einem Testdatensatz
results = model.predict(source='<Pfad_zum_Testbilderordner>', save=True)

Die Ergebnisse werden mit Bounding Boxes im Ordner \textnormal{/runs/predict}
    abgespeichert.
```

Anmerkungen

ROS-Anbindung

Eine direkte Anbindung von ROS an den Simulator war im Rahmen dieser Arbeit weder im End-to-End- (E2E) noch im SDK-Kontext realisierbar. Die offizielle Dokumentation erweist sich in diesem Bereich als unzureichend und enthält keine verwertbaren Implementierungshinweise. Ob eine funktionale Integration derzeit grundsätzlich vorgesehen ist, bleibt offen.

Custom Maps

Das einfache Einbinden eigener OpenDRIVE-Karten im .xodr-Format ist nicht direkt möglich. Obwohl OpenDRIVE die Grundlage für das Straßennetz in DriveSim bildet, erwartet das Programm zur Laufzeit eine konvertierte .usd-Datei. Eine direkte Umwandlung kann bisher nicht erfolgreich durchgeführt werden. Auch hier fehlt eine zuverlässige Dokumentation oder ein offizielles Tool zur Konvertierung.

5 Evaluierung

Die folgende Evaluation dient dazu, ein allgemeines Bild über die Fähigkeiten von DriveSim zu erhalten. Im Mittelpunkt steht der Realismus der Simulation. Dieser bezieht sich einerseits auf das Verhalten der Verkehrsteilnehmer, andererseits auf die erzeugten Sensordaten. Als objektives Benchmark dient wie angesprochen ein neuronales Netzwerk, das ausschließlich auf Simulationsdaten trainiert und anschließend mit realen Daten konfrontiert wird.

5.1 Realismus des Verhaltens der Verkehrsteilnehmer

Verkehrsteilnehmer in DriveSim lassen sich in zwei Hauptgruppen unterteilen: Fahrzeuge (z.B. Autos, Motorräder, Fahrräder) und Fußgänger.

Fahrzeugverhalten

Die Grundlage für das Fahrzeugverhalten bildet das OpenDrive-Format, insbesondere die .xodr-Straßendefinition (siehe Kapitel 1). Fahrten außerhalb dieses definierten Straßennetzes sind kaum möglich und führen häufig zu Abstürzen der Simulation. Hinsichtlich der Animationen ist wenig zu beanstanden: Bremsverhalten, Raddrehung, Lenkeinschläge, Blinker etc. sind realistisch umgesetzt.

Pathfinding

Fahrspuren werden, wie bereits erwähnt, vom .xodr-Straßensystemen vorgegeben und von allen Fahrzeugtypen korrekt eingehalten. Ein Auto, das sich in einer Abbiegespur befindet, wird – sofern nicht anders definiert – abbiegen. Brems- und Beschleunigungsverhalten wirken realistisch. Statische Objekte innerhalb des Straßensystems können eigenständig umfahren werden. Größere Umwege, etwa durch Zurücksetzen oder große Lenkwinkel, sind jedoch nicht möglich (siehe Abbildung 5.1).

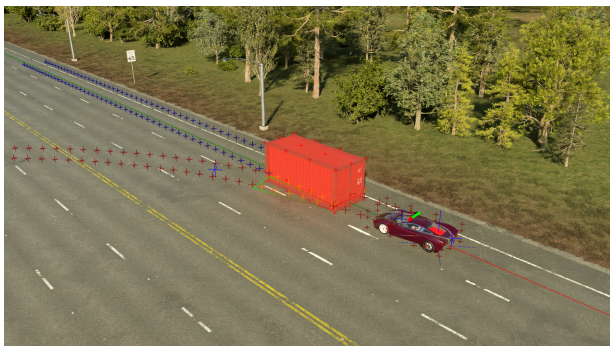


Abbildung 5.1: *Das Auto bleibt an einem größeren Objekt stecken. Rote Kreuze repräsentieren den geplanten Pfad des Autos. Die blauen Kreuze zeigen den Pfad des Containers. Dies gilt trotz der statischen Eigenschaften für die meisten Objekte. Der Balken über dem Fahrzeug symbolisiert die Aggressivität: Grün steht für entspannt, Rot für aggressiv. Dieser Wert beeinflusst das Einhalten von Abständen, das Brems- und Beschleunigungsverhalten, die Bereitschaft zum Spurwechsel usw. Die durchgehenden roten bzw. grünen Linien zeigen die durch das .xodr-Format definierten Straßen.*

Interaktion mit anderen Verkehrsteilnehmern

Fahrzeuge reagieren auf andere Verkehrsteilnehmer ähnlich wie auf statische Objekte. Stehende oder langsamere Fahrzeuge sowie Fußgänger werden erkannt. Es wird abgebremst und – je nach Aggressivitätslevel – entweder überholt oder gewartet (siehe Abbildung 5.2). Komplexere Manöver, wie etwa mehrfache Spurwechsel, sind jedoch nicht möglich (siehe Abbildung 5.3). Kollisionen haben – sofern im Szenario nicht explizit anders definiert – kaum Auswirkungen. Die betroffenen Fahrzeuge versuchen ihren Weg, soweit möglich, fortzusetzen.

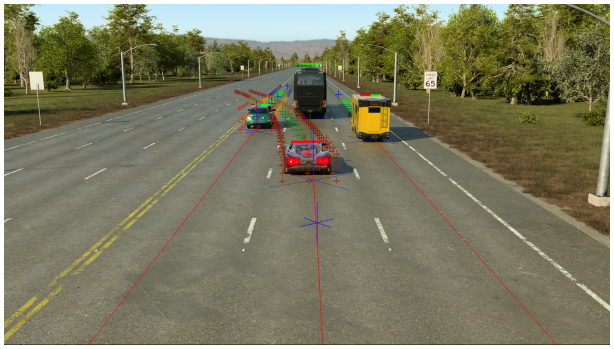


Abbildung 5.2: *Der vorausfahrende Bus und der Krankenwagen fahren langsamer als der Sportwagen und das türkisfarbene Auto. Der Sportwagen bremst ab, wechselt automatisch die Spur und überholt. Die grünen Kreuze repräsentieren die Wahrnehmung des Fahrzeugs.*

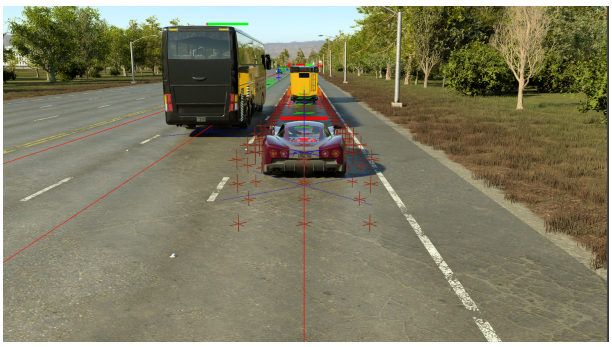


Abbildung 5.3: *Gleiches Szenario, jedoch mit veränderter Ausgangsposition des Sportwagens. Ein doppelter Spurwechsel, um die vorausfahrenden Fahrzeuge zu umgehen, ist nicht möglich.*

Einhaltung der Verkehrsregeln

Die maximal zulässige Fahrgeschwindigkeit wird durch das .xodr-Straßenformat definiert. Fahrzeuge beschleunigen bis zur jeweiligen Begrenzung. An Kreuzungen wird gewartet, und Vorfahrtsregeln werden eingehalten. Ampeln werden ebenfalls korrekt beachtet. Grundlegende Verkehrsregeln werden somit zuverlässig umgesetzt. Verstöße gegen diese Regeln sind nahezu ausschließlich über den Szenariopfad erzwingbar. Ein sehr hohes Aggressivitätslevel kann ähnliche Effekte erzeugen. Straßenschilder hingegen werden derzeit nicht berücksichtigt.

Fußgängerverhalten

Fußgängermodelle können, wie bereits erwähnt, beliebig in E2E platziert werden. Die verfügbaren Animationen beschränken sich auf Gehen und Laufen.

Pathfinding

Fußgänger bewegen sich ausschließlich entlang vordefinierter Wegpunkte. Statische Objekte auf dem Weg können nur eingeschränkt umgangen werden (siehe Abbildung 5.4). Ein selbstständiges, regelkonformes Überqueren von Straßen oder Zebrastreifen ist derzeit nicht implementiert.

Abbildung 5.4: *Die Fußgängerin weicht dem Objekt aus. Die grünen Kreuze zeigen die Wahrnehmung, die blauen Kreuze den geplanten Weg.*

Interaktion mit anderen Verkehrsteilnehmern

Fußgänger können sich gegenseitig dynamisch ausweichen. Bei vorbeifahrenden Autos halten sie an und setzen ihren Weg anschließend fort.

5.2 Realismus der von Kameras aufgenommenen Bilddaten

Die folgenden Bilddaten wurden mithilfe von Replicator kreiert. Grundlegend wurden Basisszenarien verwendet. Leichte Veränderungen wurden an Anzahl der gespawnten Fahrzuege per Frame und verfügbaren Fahrzeugmodellen vorgenommen. Die verwendeten Kameras sind die gegebenen Basismodelle camera_fov60 und camera_fov120.

5.2.1 Optische Eigenschaften

Die Ray Traycing-Technologie von NVIDIA erlaubt sehr realitätsnahe Aufnahmen. Lens Flares, Reflexionen usw. sind vorhanden und realitätsnah umgesetzt (siehe Abbildungen 5.5 und 5.6). Größere Reflexionen, z.B. an Glasfronten, werden nicht unterstützt. Optische Verdeckungen durch Regen oder Schnee sind ebenfalls nicht implementiert. Der Simulator unterstützt lediglich die Simulation von Nebel und nasser Fahrbahn (siehe Abbildung 5.7).

Optische Effekte durch extreme Temperaturen sind ebenfalls nicht simulierbar. Keiner der simulierbaren Wetterffekte hat Einfluss auf die erzeugten Bounding Boxes (soiehe Abbildungen 5.5/5.7)

Abbildung 5.5: *Direkte Lichtquellen erzeugen Lens Flares sowie einen radialen Glow bzw. Bloom-Effekt. Reflexionen an Autos, aber auch an der Fahrbahn, sind realistisch umgesetzt.*

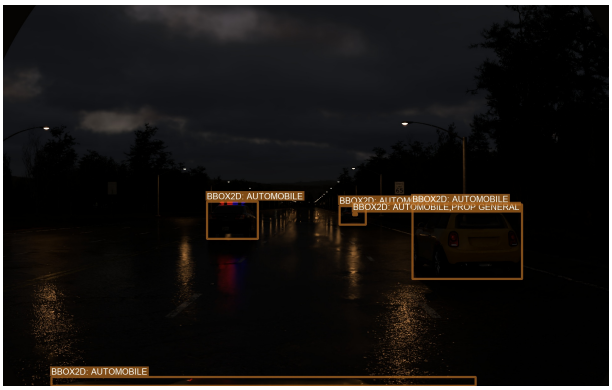


Abbildung 5.6: *Reflexionen verschiedener Lichtquellen, z.B. der Straßenlaternen, werden ebenfalls simuliert. Nässe auf der Fahrbahn ist realistisch umgesetzt.*

Abbildung 5.7: *Wettereffekte werden nur begrenzt simuliert. Nebel, Wolkendichte und Sonnenstand sind die einzigen modifizierbaren Aspekte. Hier zu sehen ist die maximale Nebeldichte. Diese ist zwar realistisch dargestellt, aber zu dünn, um einen nennenswerten Effekt zu erzeugen.*

5.2.2 Vielfalt & Komplexität der Szenen

Replicator ermöglicht das schnelle Erstellen unterschiedlichster Szenarien. Diese folgen dabei jedoch keiner tieferen Logik. Positionen und Modelle von Straßenschildern, Fußgängern und Autos sind zufällig und führen zu logischen Fehlern im abgebildeten Fahrszenario (siehe Abbildung 5.8).

Um komplexere, in sich schlüssige Szenen zu erstellen, müssen die Argumente für Replicator feinjustiert oder die Szenen manuell mit E2E und komplexeren ScenarioGraphs erstellt werden.

Abbildung 5.8: *Ein Auszug aus dem Basiszenario Sign. Durch die ungesteuerete Randomisierung von Replicator kommt es zu Fehlern in der Scenariologik.*

5.2.3 Realwelt-Evaluation eines simulativ trainierten YOLO-Netzwerks

Für den folgenden Vergleich wurden mithilfe vortrainierter YOLO-Netzwerke, wie im Kapitel 3 beschrieben, eigene Netze ausschließlich auf Simulatorbildern trainiert. Basis war das YOLOv11m-Modell von Ultralytics. Der verschiedenen Datensätze bestanden aus etwa 3500 Bildern, erstellt aus Basisszenarien (siehe Abbildung 5.9). Die Verteilung zwischen Training und Validierung betrug 80% zu 20% . Das Netzwerk wurde nacheinander auf diese trainiert.

Trainiert wurde auf einer NVIDIA RTX A6000. Die Startparameter wurden an die Hardware angepasst (siehe A.10/A.11 für die Ausgangsargumente/Skripte).

Abbildung 5.9: *Ein Beispiel für die im Datensatz enthaltenen Bilder. Erstellt mithilfe des Basisszenarios seq_multi_maps, ergänzt durch Szenen aus signs.*

Die Trainingszeit betrug insgesamt 40 Minuten. Anschließend wurde das Netzwerk mit realen Bildern getestet (siehe Abbildung 5.10/5.11/5.12/5.13).

Abbildung 5.10: *Erkennung auf einem Testbild. Hier wurde eine der ersten Generationgesteste. Es kommt zu vielen Falschidentifikation. Die Testbilder wurden sehr breit gehalten, im Bezug auf Lichtverhältnissen, Szenarien usw. um die Domain-Lücke zu minimieren.*

Abbildung 5.11: *Ein weiteres Ergebniss. Das geteste Netzwerk ist die 6. Generation. Es kommt zu weniger Falschidentifikation.*

Abbildung 5.12: *Besondere Lichtverhältnisse oder Scenariokomplexität, etc. führen oft zu keiner Identifiktaion. Dies passiert unabhängig von der Netzwerkgeneration.*

Abbildung 5.13: *Im Gegenzug zu Realbildern erkennen die Netzwerke jeder Generation, bei Simulatorbilder alle Objekte mit hoher Sicherheit. Im gezeigten Beispiel sind die Confidence Scores, wie zu sehen ist in den hohen 90% wiederzufinden. Fehlidentifikationen kommen kaum vor.*

Für eine Übersicht über die Netzwerk siehe A.1/A.2. Unabhängig von der Netzwerkgeneration sind häufigen Falschidentifikation oder fehlende Detektionen zu beobachten Es ist eine Sim-Real Gap zu erkennen.

Sim-Real-Gap

Keine der getesteten Netzwerkkonfigurationen konnte eine erfolgreiche Objekterkennung auf den allen realen Bildern erzielen.

Die Ursache liegt vermutlich in der grundsätzlichen Diskrepanz zwischen simulierten und realen Bilddaten. Während die Simulationsbilder durch perfekte Ausleuchtung, scharfe Kanten und saubere Labels geprägt sind, weisen reale Aufnahmen typischerweise Bildrauschen, Bewegungsunschärfe, Verzerrungen und visuelle Inkonsistenzen auf.

Es ist möglich anzunehmen,das das Netzwerk spezifische Merkmale von Simulationsbildern lernt und dadurch eine begrenzte Generalisierungsfähigkeit gegenüber realen

Szenarien entwickelt. Diese Einschränkung kann zusätzlich durch die teils fehlende Szenenstruktur in den Simulationsdaten sowie durch kleinere Unstimmigkeiten in deren Qualität verstärkt werden.

Für DriveSim bedeutet das: Rein auf Bais des Simulators Neuronale Netze realweltfähig zu trainieren – ist (zumindest in der getesteten Konfiguration) teilweiße erfüllt. Ein nahtloses Testen realer KI-Komponenten oder Sensorik auf Basis von Simulationsdaten erscheint derzeit aber noch nicht einfach umsetzbar.

Ausblick

Im Folgenden werden Lösungsansätze zur Realwelttauglichkeit der Netzwerke aufgezeigt, deren Umsetzung nicht in den Rahmen dieser Arbeit fällt.

Ein möglicher Ansatz Ansatz wäre ein Fine-Tuning mit Bildern aus der realen Welt – entweder gemischt im ursprünglichen Datensatz oder separat als zusätzliches Training.

Ein weiterer Ansatz wäre die Veränderung der Simulationsdaten selbst, um sie der Realität besser anzupassen. Dies kann durch Anpassungen im Python-Code erfolgen. (siehe Anhang A.1) Alternativ können Veränderungen an Licht, Reflexionen, usw. der Szenarien über E2E angepasst werden oder die Sensoren abgestimmt werden.

6 Zusammenfassung

Ziel dieser Arbeit war es, die Simulationsplattform NVIDIA DriveSim hinsichtlich ihrer Eignung zur Datengenerierung für KI-basierte autonome Fahrsysteme zu evaluieren. Im Fokus standen dabei einerseits das Verhalten der simulierten Verkehrsteilnehmer, andererseits die Realitätsnähe der generierten Kameradaten. Als übergeordnetes Kriterium diente die Frage, ob sich die Plattform für den Einsatz in einem universitären Forschungskontext eignet – insbesondere zur Vorbereitung, zum Training oder zur Validierung von Deep-Learning-basierten Self-Driving-Stacks.

Im Rahmen der Arbeit wurden mit DriveSim E2E individuelle Fahrszenarien erstellt und mithilfe von Replicator in synthetische Bilddaten mit Bounding Boxes umgewandelt. Anschließend wurde ein neuronales Netz zur Objekterkennung (YOLOv11) ausschließlich auf diesen Daten trainiert. Der anschließende Test mit realen Bildern zeigte, dass nur teilweise übertragbaren Erkennungsergebnisse erzielt wurden. Damit konnte eine Sim-to-Real-Gap nachgewiesen werden.

Die qualitative Analyse des Verkehrsverhaltens innerhalb der Simulation ergab, dass einfache Regelverhalten wie Spurhaltung, Bremsverhalten und das Erkennen anderer Verkehrsteilnehmer grundsätzlich umgesetzt werden. Komplexere Interaktionen, wie etwa das selbstständige Umfahren dynamischer Hindernisse, sind jedoch nur begrenzt realistisch umgesetzt.

Darüber hinaus traten im praktischen Umgang mit der Plattform mehrere Einschränkungen auf. Dazu zählen instabile Writer im Datengenerierungsprozess, fehlerhafte Abhängigkeiten innerhalb des Containers und eine nur sehr eingeschränkt gepflegte Dokumentation. Wesentliche Funktionen – wie etwa die Simulation von Regen oder Schnee, das Laden eigener Karten im .xodr-Format oder die Anbindung an ROS – sind nicht oder nur teilweise verfügbar. Auch fehlt eine aktive Open-Source-Community, die einen Erfahrungsaustausch außerhalb von Großkunden ermöglicht.

Positiv hervorzuheben sind die realistische physikalische Fahrzeugmodellierung, das klar strukturierte Sensormodell und die Integration in die Omniverse-Plattform. Die Replicator-Komponente erlaubt zudem eine flexible Skalierung von Datensätzen unter kontrollierten Bedingungen.

Zusammenfassend lässt sich feststellen, dass NVIDIA DriveSim zwar Potenzial für industrielle Anwendungen bietet, die Plattform in ihrer aktuellen Form jedoch nur eingeschränkt für Forschungseinrichtungen geeignet ist. Insbesondere die dokumentierte Sim-to-Real-Gap und die eingeschränkte Transparenz bei der Szenenlogik stellen zentrale Hindernisse dar. Die in dieser Arbeit durchgeführte Untersuchung zeigt, dass Trainingsdaten aus DriveSim aktuell nicht ohne Weiteres für die Entwicklung realweltlicher KI-Modelle im Bereich autonomes Fahren nutzbar sind.

A Anhang

A.1 Python Skripte

Listing A.1: *Argumente für Änderungen am Datensatz: Macht Simulierte Bilder unschärfer, etc. und simuliert so mehr die Reale Welt.*

```
auto_augment: randaugment
erasing: 0.4                # Simuliert Verdeckung
hsv_v: 0.4                  # Veraender die Helligkeit
perspective: 0.001          # Verzerrt die Geometry
```

Listing A.2: *Um Änderungen an Container langfristig vorzunehemn müssen diese als ein neues Image("Name") gespeichert werden*

```
docker commit <container-name-oder-id> mein-neues-image:tag         #speichert
    den Container mit Aenderungen als ein Neues Images ab. Der unveraenderte
    Container ist weiterhin unter dem alten Image/ID ausfuehrbar.
```

Listing A.3: *Das Script started Replicator und bindet/Mounted automatisch die Output Ornder*

```
#!/bin/bash

set -e
# User input for docker links
echo "Enter path to DRIVE SIM container from NGC"
read -p 'DRIVE SIM path: ' IMAGE
read -p 'Are you using an NGC hosted Nucleus?(y/n): ' NGC_NUCLEUS       #bei
    localem omniverse Server gebe n ein

case $NGC_NUCLEUS in
    [yY] ) docker run -it --rm --gpus all -v /usr/share/nvidia:/usr/share/
        nvidia/ --name=ds2 --network=host --ulimit nofile=65535:65535 --env "
        ACCEPT_EULA=Y" --env DISPLAY -e NGC_API_KEY=$NGC_API_KEY --env nuc=
        omniverse://c2981f53-74dc-4c0e-a8af-99fe16d81521.cne.ngc.nvidia.com/ -v
         ~/ovcache:/var/ovcache $IMAGE /bin/bash
esac        #Started den Container bei y Eingabe

case $NGC_NUCLEUS in
        [nN] ) read -p 'Enter Nucleus server Username:' USERNAME;      #
            Benutzername des Omniverse Server Nutzers
    read -p 'Enter Nucleus server Password:' PASSWORD;      #Passwort des
        Omniverse Server Nutzers
    OUT_DIR="/home/vima/nvidia_local_data/Output"       #Weg zum Output Ordner
        auf dem lokalen System

    docker run -it --rm --gpus all  \       #started den Container bei Eingabe
        n. --gpus all gibt dem docker das Recht auf die grafikkarte voll
        zuzugreifen
    --network=host --ulimit nofile=65535:65535 \
    -v /usr/share/nvidia:/usr/share/nvidia/ \
    -v ~/ovcache:/var/ovcache \
    --env "ACCEPT_EULA=Y" --env DISPLAY  \
    --env NVIDIA_DOC_PORT=8080 \
    -v "${OUT_DIR}":/tmp/replicator_out \       #Mounted unser vorher
        definierten lokalen Output Ordner in den Output Ordner des Containers.
        Erstellte Daten werden in diesen und damit automatisch in den Output
        Ordner im lokalen System gespeichert.
    -e OMNI_USER=${USERNAME} -e OMNI_PASS=${PASSWORD} \
    ${IMAGE} \
    bash
esac
```

Listing A.4: *Config.toml des Basisscenarios Sign. Alle Config Files der basisscenarios sind im Container unter* /Files/drivesim-ov/_build/linux-x86_64/ext/omni.drivesim.replicator.domainrand/omni/drivesim/replicator/domainrand/config/ *zu finden.*

```
[Assets]
root = "{/app/drivesim/defaultNucleusRoot}/"
sets = [
    "ego.toml",
    "road_signs.toml",
    "static_traffic_vehicles.toml",
    "static_public_vehicles.toml",
    "static_oversize_vehicles.toml",
    "cones.toml",
]       #sets enthaelt die Namen der Asset.toml Datein. (siehe

[Events.config]
enable_script = true
enable_periodic = true      #Replicator untersheidet in Zwei Modie: Periodisch,
     in dem jeder Frame Radomisiert wird; und sequentiell, hier wird das
    Scenario einmal am Anfang Randomisiert.
periodic_start_frame = 0
collisions_enabled = true
enable_ego_teleportation = true     #bei Periodischen Scenarion 'faehrt' das
    Auto nichtmehr. F r Fortbewegung auf der Stra e zwischen den Frames muss
    es  ber  eine Funktion telepotiert werden. bei Sequenziellen Scenarion
    w rde hier 'false'stehen und das Auto w rde sich wie gewohn fortbewegen.

####Setup-Phase####: In dieser Phase weden z.B. 'Populationen' aus den Asset.
    toml erstellt. Diese dienen danach in der Periodischen Phase als
    Anhaltepunkt f r die Funktionenen.

[[Events.setup]]
type='EgoRand'
func='setup'

[[Events.setup]]                                    #Anfang einer Setup
    Funktion
type= 'VehicleRandStat'                             #Definition des Typs
func= 'setup'                                       #Definition der Funktion;
    hier wird eine Population erstellt
pop_name='traffic_vehicles'                         #Name der erstellten
    Population
asset_file='static_traffic_vehicles.toml'           #Asset.toml Datei aus der
    die Population ertsellt werden soll
asset_sel_type='range'                              #Definition der Auswahlart
asset_sel=[0,10]                                    #Hier wird einfach
    zufaellig zwischen den ersten 11 in der .toml definierten Assets ausgesucht
    (Index beginnt bei 0)

[[Events.setup]]
type= 'VehicleRandStat'
func= 'setup'
pop_name='oversize_vehicles'
asset_file='static_oversize_vehicles.toml'
asset_sel_type='range'
asset_sel=[0,5]

[[Events.setup]]
type='SignRand'
func='setup'
pop_name='road_signs'
asset_file='road_signs.toml'
asset_sel_type='index'
asset_sel=[3,14,15,16,17]

#### PERIODIC EVENTS ####: Periodisceh Phase. Die folgenden Funktionen werden,
    wenn nicht anderst definiert f r jeden neuen Frame ausgef hrt.
```

```
[[Events.periodic.1]]        #die 1 hinter periodic definiert nach wie vielen
    Frames diese Funktion ausgef hrt werden soll->1=nach jedem Frame, 2=alle 2
     Frames
type='EgoRand'               #Definition des Typs
func='place_along_road'      #Definition der Funktion; Fas Ego Fahrzeug wird
    telepotiert
distance=5                   #zu telepotierende Distanz in Meter

[[Events.periodic.1]]
type='VehicleRandStat'           #Definition des Typs
func='randomize_position'        #Definition der Funktion; hier werden zufaellig
     Objekte gespawnt
pop_name='traffic_vehicles'      #die zu spawnend Objekte werden aus der vorher
    definierten Population entnommen
region_center='ahead_ego'        #Bereich in dem die Objekte gespawnt werden
    sollen
shuffle_order=true
num_placements='random'          #Anzahl der zu spawnenden Objekte

[[Events.periodic.1]]
type='VehicleRandStat'
func='randomize_color'
pop_name='traffic_vehicles'

[[Events.periodic.1]]
type='VehicleRandStat'
func='randomize_position'
pop_name='oversize_vehicles'
region_center='ahead_ego'
shuffle_order=true
num_placements=2

[[Events.periodic.1]]
type='SignRand'
func='randomize_position'
pop_name='road_signs'
region_center='ahead_ego'
orientation='face_ego'
min_distance=3
# offset makes signs closer to ego
offset=-4
shuffle_order=true
surface_type=['median', 'sidewalk']
num_placements='random'
```

Listing A.5: *Beipiel für die Asset Datei Ego.toml des Basisscenarios sings. Es enthält die Dateipfade zu den einzelen Modellen bzw. deren .usda Dateien. Die Asset.toml Datein aller Basisscenarios sind zu finden unter /Files/drivesim-ov/_build/linux-x86_64/ext/* omni.drivesim.replicator.domainrand/omni/drivesim/replicator/domainrand/config/assets/*. Die Modellpfade verweisen alle auf den Omniverse Server.*

```
[Assets.EgoRand]
objects = """
Projects/ds2_scenarios/production/ego/SDG/Generic/generic_sedan_sdg_1cam.usda
Projects/ds2_scenarios/production/ego/SDG/Generic/
    generic_sedan_sdg_1lidar_5radar.usda
Projects/ds2_scenarios/production/ego/SDG/Generic/generic_sedan_sdg_1lidar.usda
Projects/ds2_scenarios/production/ego/SDG/Generic/generic_sedan_sdg_4cam.usda
Projects/ds2_scenarios/production/ego/SDG/Generic/generic_sedan_sdg_8cam.usda
"""
```

Listing A.6: *Dieser base_ds_parser behebt Probleme mit der Standartversion indem er die 3d Bounding Boxes umgeht.*

```
import concurrent
```

```
import glob
import json
import multiprocessing
import os
from collections import defaultdict

import cv2
import fiftyone as fo
import matplotlib.pyplot as plt
import numpy as np
import tqdm
from PIL import Image

from .pp_utils.datatypes import CameraParams, ObjectDetection
from .pp_utils.utils import _read_exr
from .pp_utils.utils_3d import proj_3dbbox_to_image_coords


class BasicDSParser:
    def __init__(self, data_dir, config=None) -> None:
        self.data_dir = data_dir
        self.config = config
        self.session_hash = None
        self.cameras = None
        self._groups = defaultdict(fo.Group)
        self.grouped_dataset = None

    def setup(self):
        g = os.walk(f"{self.data_dir}/frames")
        self.session_hash = next(g)[1]
        self.cameras = next(g)[1]
        self.grouped_dataset = len(self.cameras) > 1
        assert len(self.session_hash) == 1, "Multiple session hashes found"
        assert len(self.cameras) >= 1, "At least one camera expected"

        self.session_hash = self.session_hash[0]

    def _get_camera_path(self, camera):
        return os.path.join(self.data_dir, "frames", self.session_hash, camera)

    def parse_dataset(self):
        samples = []
        for camera in self.cameras:
            print(f"{camera=}")
            camera_path = self._get_camera_path(camera)
            images = glob.glob(os.path.join(camera_path, "ldr_color/*"))
            if not len(images) >= 1:
                print(f"{camera} folder doesn't contain any RGB data")
                continue

            os.makedirs(os.path.join(camera_path, "semantic_segmentation_fo"),
                exist_ok=True)
            os.makedirs(os.path.join(camera_path, "depth_fo"), exist_ok=True)

            lock = multiprocessing.Lock()

            with tqdm.tqdm(len(images)) as pbar:
                with concurrent.futures.ProcessPoolExecutor() as executor:
                    futures = [executor.submit(self.parse_frame, camera_path,
                        image) for image in images]

                    for _idx, future in enumerate(concurrent.futures.
                        as_completed(futures)):
                        try:
                            result, frame_num = future.result()
                        except Exception as e:
                            print(f"Error parsing frame: {e}")
                            continue

                        sample = fo.Sample.from_dict(result)
```

```
                        result["camera"] = camera
                        if self.grouped_dataset:
                            with lock:
                                sample.set_field("group", self._groups[
                                    frame_num].element(camera))
                                samples.append(sample)
                                pbar.update(1)
                        else:
                            samples.append(sample)
                            pbar.update(1)

        return samples

    def parse_2d_bbox(self, camera_path, filename):
        bbox_data = np.load(os.path.join(camera_path, f"bounding_box_2d_tight/{
            filename}.npy"))

        with open(os.path.join(camera_path, f"bounding_box_2d_tight/labels_{
            filename}.json")) as file:
            labels_data = json.load(file)

        with open(os.path.join(camera_path, f"camera_params/{filename}.json"))
            as file:
            camera_params = json.load(file)

        camera_params = CameraParams(camera_params)

        im_height = camera_params.cameraFisheyeNominalHeight
        im_width = camera_params.cameraFisheyeNominalWidth

        if im_width == 0 or im_height == 0:
            print(f"Skipping 2D bbox for {filename} due to zero image size")
            return fo.Detections(detections=[])

        num_detections = bbox_data["semanticId"].shape[0]

        bbox_data_norm = np.array(
            [
                bbox_data["x_min"].astype(int) / im_width,
                bbox_data["y_min"].astype(int) / im_height,
                (bbox_data["x_max"].astype(int) - bbox_data["x_min"].astype(int
                    )) / im_width,
                (bbox_data["y_max"].astype(int) - bbox_data["y_min"].astype(int
                    )) / im_height,
            ]
        )

        detections = []
        for i in range(num_detections):
            label = labels_data[f"{bbox_data['semanticId'][i]}"]["class"]
            detections.append(
                fo.Detection(
                    label=label,
                    bounding_box=bbox_data_norm[:, i].tolist(),
                )
            )

        return fo.Detections(detections=detections)

    def parse_semantic_segmentation(self, camera_path, filename):
        mask_targets = {}
        sem_seg = plt.imread(os.path.join(camera_path, "semantic_segmentation",
             f"{filename}.png"))

        with open(os.path.join(camera_path, "semantic_segmentation", f"mapping_
            {filename}.json")) as file:
            mapping = json.load(file)

        for k in mapping.keys():
            if k not in mask_targets:
```

```
                mask_targets[int(k)] = mapping[k]["class"]

        seg_path = os.path.join(camera_path, "semantic_segmentation_fo", f"{
            filename}.png")
        cv2.imwrite(seg_path, sem_seg[:, :, 0] * 255)

        return seg_path, mask_targets

    def parse_depth(self, camera_path, filename):
        NEAR = 0.01
        FAR = 100

        depth_data = _read_exr(os.path.join(camera_path, "distance_to_camera",
            f"{filename}.exr"))
        depth_data = np.clip(depth_data, NEAR, FAR)
        depth_data = (np.log(depth_data) - np.log(NEAR)) / (np.log(FAR) - np.
            log(NEAR))
        depth_data_uint8 = (depth_data * 255).astype(np.uint8)

        img = Image.fromarray(depth_data_uint8)
        map_path = os.path.join(camera_path, "depth_fo", f"{filename}.png")
        img.save(map_path)

        return map_path

    def parse_frame(self, camera_path, filename):
        img_path = os.path.abspath(filename)
        filename = os.path.splitext(os.path.basename(filename))[0]

        sample = {"filepath": img_path}

        if os.path.exists(os.path.join(camera_path, "bounding_box_2d_tight")):
            sample["bbox2d"] = self.parse_2d_bbox(camera_path, filename)

        #     3D bounding box parsing intentionally skipped
        # if os.path.exists(os.path.join(camera_path, "bounding_box_3d")):
        #     sample["bbox3d"] = self.parse_3d_bbox(camera_path, filename)

        if os.path.exists(os.path.join(camera_path, "semantic_segmentation")):
            mask_path, mask_targets = self.parse_semantic_segmentation(
                camera_path, filename)
            sample["segmentations"] = fo.Segmentation(mask_path=mask_path)

        if os.path.exists(os.path.join(camera_path, "distance_to_camera")):
            map_path = self.parse_depth(camera_path, filename)
            sample["depth"] = fo.Heatmap(map_path=map_path)

        return sample, filename
\textbf{}
```

Listing A.7: *Das FiftyoneCore Skript. Kann die erstellten Datensets impotieren und als Fifty1, Yolo oder Coco Datensetz exportieren. YOLO und COCO führten in der Umsetzung aber wiederholt zu Problemen. Das Skript exportiert nach /tmp/replicator_out. Unsere Schnittstelle mit dem Container.*

```
import os
import fiftyone as fo
import toml
from .basic_ds_parser import BasicDSParser
from .pointcloud_parser import PointCloudParser


def get_parser_from_writer(writer_name):
    if writer_name == "BasicDSWriterFull":
        return BasicDSParser
    elif writer_name == "PointCloudWriter":
        return PointCloudParser
    else:
```

```
        return


def get_dataset_name(data_dir, writer):
    return f"{os.path.basename(os.path.normpath(data_dir))}_{writer}"


def export_dataset(dataset, export_dir, dataset_format="fiftyone"):
    """
    Export dataset in the specified format.
    Supported formats: fiftyone, coco, yolo
    """
    print(f"Exporting dataset to {export_dir} as format: {dataset_format}")

    if dataset_format == "coco":
        dataset.export(
            export_dir=export_dir,
            dataset_type=fo.types.COCODetectionDataset,
            label_field="bbox2d"
        )
    elif dataset_format == "yolo":
        dataset.export(
            export_dir=export_dir,
            dataset_type=fo.types.YOLOv5Dataset,
            label_field="bbox2d"
        )
    elif dataset_format == "fiftyone":
        dataset.export(
            export_dir=export_dir,
            dataset_type=fo.types.FiftyOneDataset
        )
    else:
        raise ValueError(f"Unsupported export format: {dataset_format}")


def main(*args, **kwargs):
    config_file = os.path.join(os.path.dirname(os.path.abspath(__file__)), "
        config.toml")

    with open(config_file, "r") as f:
        cfg = toml.load(f)

    fo.config.default_app_address = cfg.get("default_app_address", "0.0.0.0")

    app_config = fo.app_config.copy()
    app_config = app_config.from_dict(cfg.get("fiftyone_app_config", {}))

    print("Starting FiftyOne with the following config:")
    print(app_config)

    data_path = kwargs.get("data_path")
    writer = kwargs.get("writer")
    export_base_dir = kwargs.get("export_dir", "/tmp/replicator_out")       #
        Output
    dataset_format = kwargs.get("dataset_format", "fiftyone")

    parser = get_parser_from_writer(writer)

    if parser:
        parser = parser(data_dir=data_path)
        parser.setup()

        fo_dataset_cfg = cfg.get("fiftyone_dataset_config", {})
        if not fo_dataset_cfg.get("name"):
            fo_dataset_cfg["name"] = get_dataset_name(data_path, writer)

        dataset_name = fo_dataset_cfg["name"]

        if fo.dataset_exists(dataset_name):
            print("Dataset already exists. Loading from memory.")
```

```
            dataset = fo.load_dataset(dataset_name)
        else:
            samples = parser.parse_dataset()
            dataset = fo.Dataset(**fo_dataset_cfg)
            dataset.add_samples(samples)

        #     Create a new folder for the dataset export
        export_dir = os.path.join(export_base_dir, dataset_name)
        os.makedirs(export_dir, exist_ok=True)

        #     Export dataset in user-defined or default format
        export_dataset(dataset, export_dir, dataset_format)

        # Optionally launch the app
        #sess = fo.launch_app(dataset)
        #sess.wait()

    else:
        print("Parser not found or unsupported writer.")
        exit(1)
```

Listing A.8: *Das Skript Impotiert Fifty1 Datensätze und kann YOLO, oder COCO Datensätze Exportieren. Zusätzlich werden die Frames mit gezeichneten Bounding Boxes und Labeln als .jpg exportiert. Um das geplante Training zu beschleunigen kann nach labeln und Bounding Box Art gefiltert werden.*

```
import os
import shutil
import fiftyone as fo
from fiftyone import ViewField as F
from fiftyone import types

# === Load Dataset from JSON ===
dataset_dir = <Weg_zum_Ausgangsdatensatz>                          #Input setzen

dataset = fo.Dataset.from_dir(
    dataset_dir=dataset_dir,
    dataset_type=fo.types.FiftyOneDataset,
)

# === Filter only 2D BBoxes with selected labels ===
target_labels = {"automobile", "person", "road_sign"}              #Hier kann nach
     Labels gefiltert werden

def filter_detections(sample):                                      #Filtert nur
    nach 2d Bounding Boxes. 3d Bounding Boxes k nnen funktionieren, f hrten
    aber wiederholt zu problemen und waren f  r  das KI-Training irrelevant.
    if sample.has_field("bbox2d"):
        detections = sample["bbox2d"].detections
        filtered_detections = [
            d for d in detections if d.label.split(",")[0].strip() in
                target_labels
        ]
        sample["bbox2d"].detections = filtered_detections
        sample.save()

# Apply filtering to each sample
for sample in dataset:
    filter_detections(sample)

# Filter samples that still have at least one detection
filtered_dataset = dataset.match(F("bbox2d.detections").length() > 0)

# === Set Output Root Directory ===
dataset_name = filtered_dataset.name or "FilteredDataset"
output_root = os.path.join(<Weg_zum_Outputordner>, dataset_name)        #Output
     Setzen
os.makedirs(output_root, exist_ok=True)
```

```
# === EXPORT 1: Images with Bounding Boxes Drawn (PNG) ===                #
    Exportiert die .jpgs
export_dir_bb = os.path.join(output_root, "with_bboxes")
os.makedirs(export_dir_bb, exist_ok=True)

# Check if the dataset is grouped (older version = media_type is string)
print("Media type:", dataset.media_type)
print("Group slices (raw):", dataset.group_slices)

if dataset.media_type == "group":
    # If grouped, select a slice (use the first or a known one like "image" or
        "rgb")
    slice_name = dataset.group_slices[0]  # Replace if needed
    image_slice = filtered_dataset.select_group_slices(slice_name)
    image_slice.compute_metadata(overwrite=True)
    image_slice.draw_labels(
        output_dir=export_dir_bb,
        label_fields="bbox2d",
        overwrite=True
    )
else:
    # If not grouped, proceed normally
    filtered_dataset.compute_metadata(overwrite=True)
    filtered_dataset.draw_labels(
        output_dir=export_dir_bb,
        label_fields="bbox2d",
        overwrite=True
    )

print(f"\ n   Images with bounding boxes saved to: {export_dir_bb}")


# === EXPORT 2: YOLO Format ===                                      #Expotiiert
    das YOLO Format
export_dir_yolo = os.path.join(output_root, "yolo_export")
images_root = os.path.join(export_dir_yolo, "images")
labels_root = os.path.join(export_dir_yolo, "labels")

os.makedirs(images_root, exist_ok=True)
os.makedirs(labels_root, exist_ok=True)

# Split into train/val
filtered_dataset.shuffle()
train_dataset = filtered_dataset[:int(0.8 * len(filtered_dataset))]
val_dataset = filtered_dataset[int(0.8 * len(filtered_dataset)):]

# === Label mapping ===
class_labels = sorted(target_labels)

def export_yolo(ds, split):
    images_out = os.path.join(images_root, split)
    labels_out = os.path.join(labels_root, split)

    os.makedirs(images_out, exist_ok=True)
    os.makedirs(labels_out, exist_ok=True)

    # Clean labels + copy images
    for sample in ds:
        if sample.has_field("bbox2d"):
            detections = sample.bbox2d
            for det in detections.detections:
                if det.label:
                    det.label = det.label.split(",")[0].strip()
            sample.save()
```

```
        # Copy image manually
        src_img = sample.filepath
        filename = os.path.basename(src_img)
        dst_img = os.path.join(images_out, filename)
        shutil.copy2(src_img, dst_img)

    # Export labels to a temp folder
    temp_export = os.path.join(export_dir_yolo, f"temp_labels_{split}")
    os.makedirs(temp_export, exist_ok=True)

    ds.export(
        export_dir=temp_export,
        dataset_type=fo.types.YOLOv5Dataset,
        label_field="bbox2d",
        split=split,
        classes=class_labels,
        export_media=False,
        overwrite=True,
    )

    # Move .txt files from temp_export/labels/split -> labels/train or labels/
        val
    nested_label_dir = os.path.join(temp_export, "labels", split)
    for file in os.listdir(nested_label_dir):
        if file.endswith(".txt"):
            shutil.move(os.path.join(nested_label_dir, file), os.path.join(
                labels_out, file))

    shutil.rmtree(temp_export)

export_yolo(train_dataset, "train")
export_yolo(val_dataset, "val")

print(f"\ n   YOLOv8 format exported to: {export_dir_yolo}")

# === Generate dataset.yaml for YOLOv8 ===
names_block = "\n".join([f"  {i}: {name}" for i, name in enumerate(class_labels
    )])

yaml_content = f"""\
path: {export_dir_yolo}
train: images/train
val: images/val
test:  # optional
names:
{names_block}
"""

yaml_path = os.path.join(export_dir_yolo, "dataset.yaml")
with open(yaml_path, "w") as f:
    f.write(yaml_content)

print(f"\ n   YOLOv8-compatible dataset.yaml saved to: {yaml_path}")
```

Listing A.9: *Format eines YOLO Datensatses*

```
MergedDataset/
          dataset.yaml
          images/
                train/
                val/
          labels/
              train/
              val/
```

Listing A.10: *Die Argumente für Trainig des YOLO Netzerkes wurden auf die verwendete Hardware angepasst. Als Ausgangspunkt wurde das angesporchen YOLOv11 Medium Netzwerk gewählt.*

```
from ultralytics import YOLO

model = YOLO('yolo11m.pt')

model.train(
    data='/home/tas/Desktop/Yolo Enviroment/datasets/ds1/dataset.yaml',
    epochs=50,
    imgsz=640,
    batch=32,
    device=0,
    project='runs/detect',
    name='train_A'
)
```

Listing A.11: *Die zweite Generation wird auf basis der Gewichtungen des Ersten weitertrainiert. Hier können die Netzwerke gefeintuned werden.*

```
from ultralytics import YOLO

model = YOLO('/home/tas/Desktop/Yolo Enviroment/runs/detect/train_A/weights/
    best.pt')

model.train(
    data='/home/tas/Desktop/Yolo Enviroment/datasets/ds2/dataset.yaml',
    epochs=50,
    imgsz=640,
    batch=32,
    device=0,
    project='runs/detect',
    name='train_B'
)
```

A.2 Tabellen

Tabelle A.1: *Mögliche Argumente aus generate_SDG_headless_args.py*

Argument	Required	Default	Help
--scenario	False	None	scenario name or path. (default: None)
-l, --list-scenes	False	False	List available scenario presets.
--num-frames	False	0	Frame numbers (default: 100)
-w, --writer	False	BasicDSWriterFull	Writer (default: BasicDSWriterFull)
-o, --output-path	False	None	Set output path.
--rand-config	False	None	Name of randomization TOML file. (default: None)
--rand-config-dir	False	None	Path to the folder that holds randomization TOML file.
--rand-asset-dir	False	None	Path to the folder that holds asset files to be used with randomization TOML file.
--show-ui	False	False	Show ui. (default: False)
--run-count, -r	False	1	Number of frame generation sessions to schedule.

Argument	Required	Default	Help
--ds-config	False	release	Datastudio build configuration. [debug|release]
--kit-config	False	release	Kit build configuration. [debug|release]
-q, --render-level	False	ultralow	Rendering level preset. [ultralow, low, medium, high, ultra]
--allow-root	False	False	Allow root.
--texture-streaming	False	False	Enable texture streaming (for VRAM-constrained environments)
--no-livestream	False	False	Disable livestream (default: False)
--smart-display-env	False	False	Smartly set DISPLAY variable before launching
--nucleus-path	False	None	Set nucleus path. (default: omniverse://drivesim2-rel.ov.nvidia.com)
-p, --print-command	False	False	Print kit command before running
--print-command-only	False	False	Print kit command but don't run
--custom-env-setting	False	omni_env _settings.json	Introduce a config json file to load environment variables
--monitor-usd	False	False	Monitors USD for modifications during runtime
--seed-value	False	None	Set seed value for repeatable randomization
--manual-play	False	False	Disable automatic start of autotest script and launch Datastudio GUI for debugging
--disable-autotest	False	False	Disable autotest script and launch Datastudio GUI for debugging
--log-file	False	None	Path to alternative logging directory.
--profile	False	False	Enable Tracy profiling
--usd-teleportation	False	False	Switch all prim teleportation to USD
--export-usd-log-path	False	None	Export USD log to following relative or abs path (token resolution not supported)

Tabelle A.2: *Verfügbare, vordefinierte Basisscenarion aus generate_SDG_headless.py. Weg zu den Assets, der Config Datei und teilweise auch Argumenten wie dem Writer, Rendereinstellungen, usw. sind hier definiert.*

Scenario
sdg_pyds_loopmerge
multi_maps
seq_multi_maps
rivermark
rivermark_peds
signs
seq_signs
seq_bikes
seq_rivermark
seq_rivermark_ego_spawned
pedgate_hamburg
cones
weather_example_low_res
weather_example
night_rand
seq_night
generic_lidar
generic_lidar_2
generic_radar
slope_parking
intersection_lane
loop_merge_ego_spawned
loop_merge_seq_spawned
cloud_ego_spawned
hub_signs
scene_bends_ccm_exp_example
scene_bends_ccm_exp_hdr
hello_world
basic_oval

Tabelle A.3: *Übersicht der verfügbaren Writer und deren Ausgabeformate*

Writer	**Description**	**Output**
BasicDSWriterFull	All-inclusive writer used for demos	bounding_box_2d_loose(.npy, .json), bounding_box_2d_tight(.npy, .json), bounding_box_3d(.npy, .json), camera_params(.json), semantic_segmentation(.png, .json), instance_segmentation(.png, .json), dynamics(.p), ldr_color(.png), normals(.png), occlusion(.npy), transforms(.p)

Writer	Description	Output
PointCloudWriter	Writer For Point-Clouds. Used for Lidar/Radars.	Does not Work right now! Leads to crashes in the Replicator Process.
BasicDSWriterCloud	Fewer GTs enabled. Allows operation in VRAM limited environments, such as cloud instances	bounding_box_2d_loose(.npy, .json), bounding_box_2d_tight(.npy, .json), bounding_box_3d(.npy, .json), camera_params(.json), semantic_segmentation(.png, .json), instance_segmentation(.png, .json), dynamics(.p), ldr_color(.png), transforms(.p)
KittiWriter	Writer for Kitti dataset format	rgb (.png), depth (.png), object_detection (.txt), instance_segmentation (.png), semantic (.png), semantic_segmentation (.png)
KittiWriter	Writer for Kitti dataset in semantic-kitti format. Set use_kitti_dir_names to True to write dataset in this format	depth (.png), image_02 (.png), instance (.png), label_02 (.txt), semantic (.png), semantic_rgb (.png)
HubWriter	Writer of tensor-based hub datasets (hubsets). Will output GT as selected in configuration file. Configurable semantic-type filtering. Supports direct storage on AWS S3 buckets. Besides annotators, it supports OpenDrive road lane GT.	One hubset per sensor render product (.hub)
HubReader	Hubset (.hub) converter. Supports all GT output by HubWriter. Output format is applied by callback function provided. Callback functions can be added to support any output format and layout (*). Includes example COCO callback.	bounding_box_2d_loose(*), bounding_box_2d_tight(*), bounding_box_3d(*), camera_params(*), semantic_segmentation(*), instance_segmentation(*), ldr_color(*), transforms(*), distance_to_camera(*), road_lanes(*)

Tabelle A.4: *Übersicht über die Basissensoren. Spezifische Kammeravarianten können über das 'Property' Menü ereicht werden. Die Art der Ausgegeben Daten wird über den Writer des Replicators bestimmt. Lidar/Radar Sensoren führen derzeit zu Abstürzen.*

Kategorie	**Sensor**	
cameras	Generic_hyperion8_1	
		IMX623_HY81_FOV180_GEN.sensor.usda
		IMX728_HY81_FOV120_GEN.sensor.usda
		IMX728_HY81_FOV70_GEN.sensor.usda
		IMX728_HY81_FOV30_GEN.sensor.usda
general	GPS/IMU	hires_gps_imu_can.sensor.usda
lidar	Example_Rotary.sensor.usda	
	Example_Solid_State.sensor.usda	
radar	Example.sensor.usda	
ultrasonic	Example.sensor.usda	

A.3 Abbildungen

Abbildung A.1: *Netzwerkwerte der 1. Generation.*

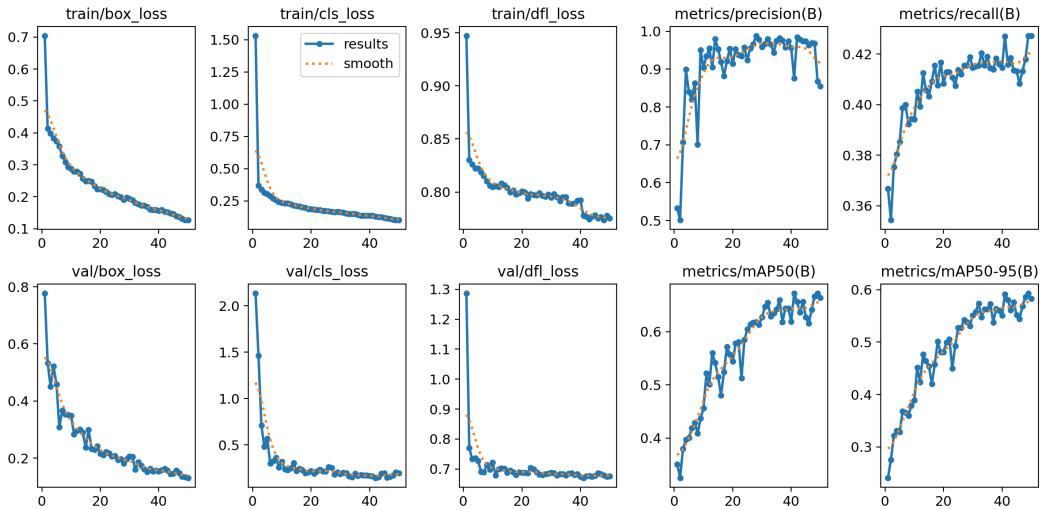
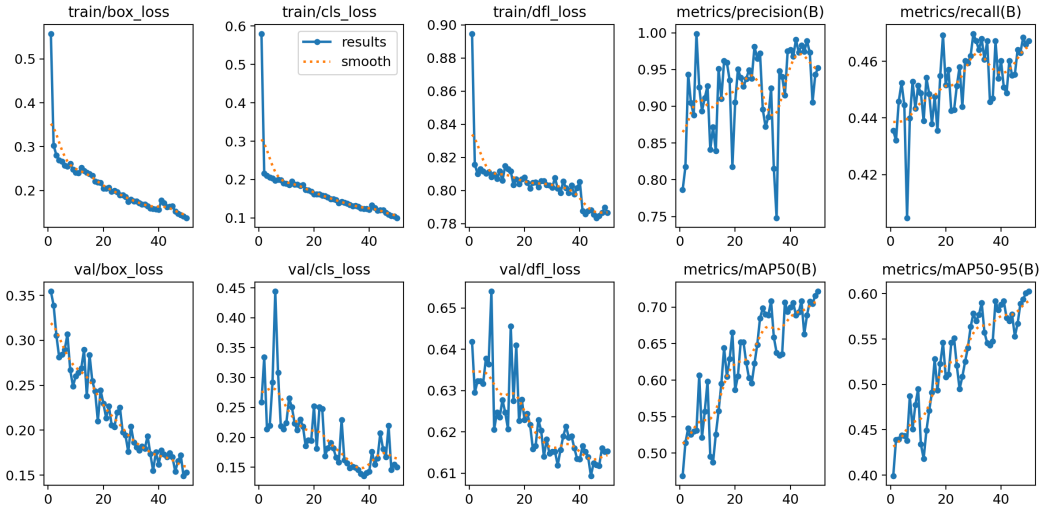


Abbildung A.2: *Netzwerkwerte der 1. Generation.*



Abbildungsverzeichnis

2.1 Replicator Workflow 6
4.1 Basic UI 12
4.2 E2E Placer 12
4.3 Scenario Outliner 13
4.4 Property-Menü 13
4.5 ScenarioGraph Example 14
4.6 Ego Vehicle Sensor-Anpassung 15
5.1 Fahrzeugverhalten 20
5.2 Überholmanöver durch Auto 20
5.3 Fehlgeschlagenes Überholmanöver 21
5.4 Fußgängerverhalten 22
5.5 Beispiel für Lichtquellen 23
5.6 Beispiel für Reflexionen 23
5.7 Beispiel für Nebel 23
5.8 Replicator-Fehler 24
5.9 Bild aus Datensatz 25
5.10 Bild aus dem Testsatz - frühe Generation 25
5.11 Bild aus dem Testsatz - späte Generation 25
5.12 Bild aus dem Testsatz - späte Generation - Keine Identifikation . . 26
5.13 Bild aus dem Simulator - späte Generation 26
A.1 Netzwerkwerte der 1. Generation. 45
A.2 Netzwerkwerte der 6. Generation. 46

Tabellenverzeichnis

A.1 generate_SDG_headless_args.py Argumente 41
A.2 Replicator Szenarios 43
A.3 Replicator Writer 43
A.4 Basissensoren 45

Literaturverzeichnis

[1] "NVIDIA DRIVE Sim." https://developer.nvidia.com/drive/simulation/getting-started.

[2] "Scenario Editor GUI — drivesim 25.1 documentation." https://developer.nvidia.com/docs/drivesim/latest/docs/drive_sim_basics/scenarios/scenario_editor.html.

[3] "DRIVE Replicator Overview and Setup — drivesim 25.1 documentation." https://developer.nvidia.com/docs/drivesim/latest/docs/drive_replicator/sdg_overview_and_setup.html.

[4] "OVERVIEW — drivesim 25.1 documentation." https://developer.nvidia.com/docs/drivesim/latest/docs/sdk/sdk.html.

[5] R. Rankin, "GAIA-2: Pushing the Boundaries of Video Generative Models for Safer Assisted and Automated Driving." https://wayve.ai/thinking/gaia-2/, Mar. 2025.

[6] L. Russell, A. Hu, L. Bertoni, G. Fedoseev, J. Shotton, E. Arani, and G. Corrado, "GAIA-2: A Controllable Multi-View Generative World Model for Autonomous Driving," Mar. 2025. Comment: Technical Report.

[7] "Opendrive." https://www.asam.net/standards/detail/opendrive/.

[8] "Docker: Accelerated Container Application Development," May 2022.

[9] "FiftyOne Basics — FiftyOne 1.4.1 documentation." https://docs.voxel51.com/user_guide/basics.html.

[10] "Discover Ultralytics YOLO models | State-of-the-Art Computer Vision." https://www.ultralytics.com/yolo.

[11] "Scenario Editor GUI — drivesim 25.1 documentation." https://developer.nvidia.com/docs/drivesim/latest/docs/drive_sim_basics/scenarios/scenario_editor.html#navigating-the-map.

[12] "Scenario Editor GUI — drivesim 25.1 documentation." https://developer.nvidia.com/docs/drivesim/latest/docs/drive_sim_basics/scenarios/scenario_editor.html#creation-tab.

[13] "Scenario Editor GUI — drivesim 25.1 documentation." https://developer.nvidia.com/docs/drivesim/latest/docs/drive_sim_basics/scenarios/scenario_editor.html#changing-scenario-parameters-via-ui-recommended.

[14] "Scenario Editor GUI — drivesim 25.1 documentation." https://developer.nvidia.com/docs/drivesim/latest/docs/drive_sim_basics/scenarios/scenario_editor.html#adding-an-action-by-visual-scripting-recommended.

[15] "Ego Config — drivesim 25.1 documentation." https://developer.nvidia.com/docs/drivesim/latest/docs/drive_sim_basics/ego_config/ego_config.html#creating-a-new-ego-config.

[16] "SENSORS — drivesim 25.1 documentation." https://developer.nvidia.com/docs/drivesim/latest/docs/drive_sim_basics/sensors/sensors.html.

[17] "Designing your DRIVE Sim scenario — drivesim 25.1 documentation." https://developer.nvidia.com/docs/drivesim/latest/docs/drive_replicator/sdg_design_scenario.html.

[18] "Using Replicator Output — drivesim 25.1 documentation." https://developer.nvidia.com/docs/drivesim/latest/docs/drive_replicator/sdg_replicator_output.html#option-2-voxel-51.

[19] "FiftyOne Installation — FiftyOne 1.4.1 documentation." https://docs.voxel51.com/getting_started/install.html.

[20] "Exporting FiftyOne Datasets — FiftyOne 1.4.1 documentation." https://docs.voxel51.com/user_guide/export_datasets.html#supported-export-formats.