# POLITECNICO DI TORINO

## Master's Degree in COMPUTER ENGINEERING


Master's Degree Thesis

# Advanced Synthetic Data Generation for ADAS: Infrastructure Sensors and CARLA-Omniverse Integration

**Supervisors**

**Prof. Massimo VIOLANTE**

**Dott. Alessandro TESSUTI**

**Dott. Simone MARAGLIULO**

**Candidate**

**Thomas BARACCO**

**Academic Year 2024-2025**

**Abstract**

Advanced Driver Assistance Systems (ADAS) validation necessitates large datasets from a variety of driving situations, yet conventional data gathering techniques have serious drawbacks in terms of cost, safety, and scenario coverage. This thesis builds on existing CARLA-based frameworks for generating synthetic data by adding two important new features aimed at improving dataset quality and realism. First, an infrastructure-based sensor simulation system was created, expanding beyond vehicle-mounted sensors to include fixed sensors on traffic lights and road infrastructure, with a dedicated management tool for dynamic sensor placement and configuration while retaining nuScenes dataset compatibility. Second, a novel framework was developed to export complete CARLA simulations to Universal Scene Description (USD) format for integration with NVIDIA Isaac Sim. This framework captures entire simulations as temporal sequences with keyframed animations, allowing deterministic replay in Omniverse's photorealistic rendering environment. An Isaac Sim extension was created to facilitate the management of imported simulations and the collection of sensor data in nuScenes format. CARLA's official USD export framework for version 0.9.16 was announced during development, prompting the strategic decision to prepare for integration with the forthcoming official solution. Despite the fact that the custom USD implementation has material limitations in the current visual output, the framework that has been developed establishes a robust integration pipeline between CARLA and Isaac Sim. This pipeline will facilitate improved photorealistic rendering capabilities and enhanced synchronization when combined with the official CARLA USD framework. This methodology offers a scalable foundation for ADAS validation that maintains industry-standard compatibility and combines the potential for near-photorealistic sensor simulation with the flexibility of virtual environments.

# Table of Contents

# List of Figures

# Acronyms

**2D**
Two-Dimensional

**3D**
Three-Dimensional

**AD**
Autonomous Driving

**ADAS**
Advanced Driver Assistance Systems

**AI**
Artificial Intelligence

**API**
Application Programming Interface

**AV**
Autonomous Vehicles

**CARLA**
Car Learning to Act

**CPU**
Central Processing Unit

**CSV**
Comma-Separated Values

**FOV**

Field of View

**FPS**

Frames Per Second

**GNSS**

Global Navigation Satellite System

**GPS**

Global Positioning System

**GPU**

Graphics Processing Unit

**GSR**

General Safety Regulation

**HIL**

Hardware-in-the-Loop

**IMU**

Inertial Measurement Unit

**JSON**

JavaScript Object Notation

**LiDAR**

Light Detection and Ranging

**MIL**

Model-in-the-Loop

**PBR**

Physically Based Rendering

**RADAR**

Radio Detection and Ranging

**RGB**

Red Green Blue

**RTX**

Ray Tracing Texel eXtreme

**SIL**

Software-in-the-Loop

**UI**

User Interface

**USD**

Universal Scene Description

**V2I**

Vehicle-to-Infrastructure

**V2X**

Vehicle-to-Everything

**VIL**

Vehicle-in-the-Loop

**XML**

Extensible Markup Language

# Chapter 1

# Introduction

The advancement of Advanced Driver Assistance Systems (ADAS) and autonomous vehicles represents a transformative shift in automotive technology, with the potential to reduce accidents, improve mobility, and revolutionize transportation [1, 2]. However, deploying safety-critical systems requires rigorous validation across diverse scenarios and edge cases that are impractical to test physically [3].

Modern perception systems require vast amounts of labeled training data representing real-world driving scenarios [4]. Traditional validation through physical testing faces significant limitations in cost, time, repeatability, and safety [5]. Synthetic data generation through high-fidelity simulation has emerged as a complementary approach, enabling controlled, repeatable testing with flexible dataset generation [6]. The effectiveness of simulation-based validation depends critically on two factors: the comprehensiveness of sensor perspectives captured, and the visual fidelity of rendered synthetic data.

## 1.1 Motivation and Problem Statement

### 1.1.1 Limitations of Vehicle-Centric Sensing

This research builds upon Federico Stella's thesis [7], which developed a framework for vehicle-centric synthetic data generation compatible with the nuScenes dataset format. Stella's work demonstrated the feasibility of generating high-quality synthetic sensor data using CARLA, implementing sensor synchronization, nuScenes-compatible organization, and flexible sensor configuration [6, 8].

While providing a solid foundation for ADAS validation with multiple sensor modalities (RGB cameras, LiDAR, radar) and proper calibration, the vehicle-centric approach has fundamental limitations:

- **Coverage Limitations**: Vehicle-mounted sensors are constrained by their

platform's position and trajectory. In complex intersections and scenarios with occlusions, the ego vehicle may miss critical information about traffic participants and hazards [9].

- **Limited Perspective Diversity**: Vehicle-only configurations lack the elevated, strategically positioned perspectives that infrastructure sensors provide. Infrastructure sensors offer wider fields of view and reduced occlusion effects [10].

- **Inability to Validate Cooperative Systems**: The framework cannot validate Vehicle-to-Everything (V2X) communication and cooperative perception architectures that leverage multi-source data fusion [11], critical as the industry moves toward cooperative intelligent transportation systems.

### 1.1.2 Visual Fidelity Constraints

Beyond sensor perspective limitations, visual fidelity represents another critical challenge. While CARLA provides excellent functional simulation with realistic physics and traffic management, its rendering pipeline has limitations affecting camera imagery photorealism [7]. These include simplified material models, limited dynamic lighting accuracy, absence of advanced rendering features (physically-accurate reflections, refractions), and computational trade-offs between performance and quality [12].

These constraints impact vision-based perception algorithm training. Research shows that training data visual characteristics significantly influence neural network performance, with models trained on synthetic data sometimes showing degraded real-world accuracy [13, 14]. This "simulation-to-reality gap" limits the reliability of purely simulation-validated algorithms.

While the industry has responded with domain randomization, photorealistic rendering engines, and hybrid approaches [15], the lack of integrated workflows for seamlessly transferring functional simulations to advanced rendering environments remains a significant gap.

## 1.2 Research Contributions

This thesis presents two major contributions addressing these limitations while maintaining full backward compatibility with established workflows:

### 1.2.1 Infrastructure-Cooperative Sensor Simulation Framework

The first contribution develops a comprehensive infrastructure sensor simulation system extending Stella's vehicle-centric approach to enable hybrid configurations with both mobile vehicle sensors and fixed infrastructure sensors. The framework consists of three integrated components:

- **Interactive Sensor Placement Tool**: A real-time graphical application providing intuitive infrastructure sensor deployment control with automatic intersection detection, intelligent sensor-to-intersection association, real-time visualization, and comprehensive management capabilities [16]. The tool supports multiple sensor types: elevated LiDAR (LIDAR_TOP), traffic cameras (CAM_TRAFFIC), directional LiDAR (LIDAR_TRAFFIC), and radar (RADAR_TRAFFIC).
- **Infrastructure Sensor Spawning and Lifecycle Management**: Infrastructure sensors maintain fixed world coordinates throughout simulations, providing persistent monitoring regardless of vehicle movements, with specialized coordinate transformation in the nuScenes pipeline.
- **nuScenes Format Extensions with Backward Compatibility**: Comprehensive extensions enabling infrastructure sensor data integration while preserving all existing vehicle-mounted capabilities and ensuring seamless integration with existing tools [17].

This enables previously inaccessible validation scenarios including complex intersection monitoring, occlusion analysis, multi-perspective perception, and V2I communication testing [18, 19].

### 1.2.2 CARLA to Isaac Sim Integration Pipeline

The second contribution addresses visual fidelity through a framework exporting complete CARLA simulations to Universal Scene Description (USD) format, enabling NVIDIA Isaac Sim integration. The pipeline consists of:

- **USD Export Framework**: A comprehensive export system converting CARLA simulations to USD with temporal animation, capturing vehicle trajectories, pedestrian movements, sensor configurations, environmental conditions, and scene geometry with coordinate transformations [20, 21]. The framework employs strategic asset integration leveraging official CARLA USD exports when available, with procedural road generation as fallback.

- **Isaac Sim Extension for Data Collection**: A custom Omniverse extension (`omni.carla.sim_controller`) providing simulation replay and data collection with automatic sensor discovery, precise temporal control, synchronized multi-modal capture, and automated nuScenes generation [22, 23]. The extension seamlessly supports vehicle-only and infrastructure-only deployment modes.

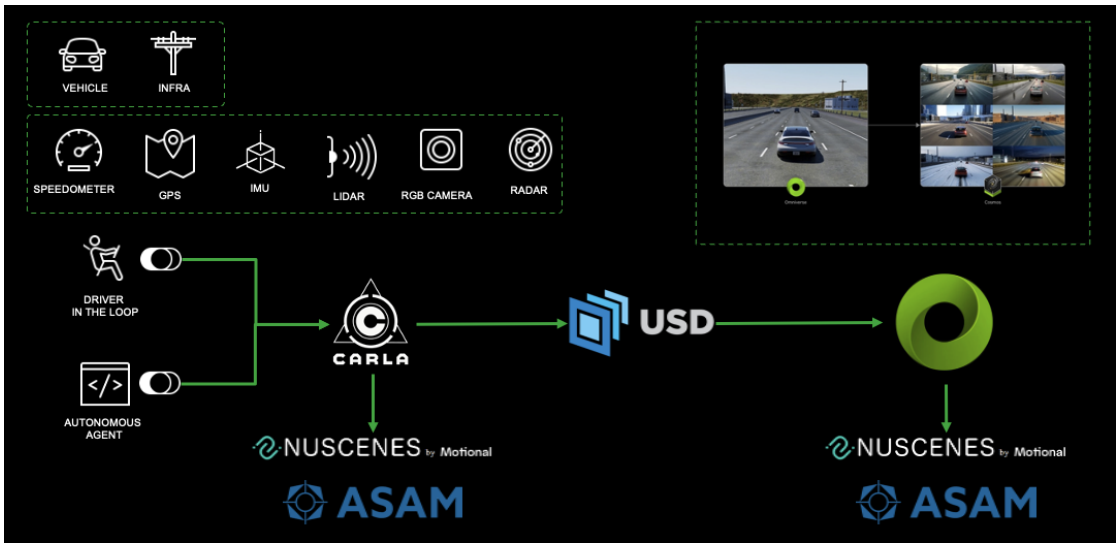


**Figure 1.1:** CARLA to Isaac Sim Integration Pipeline

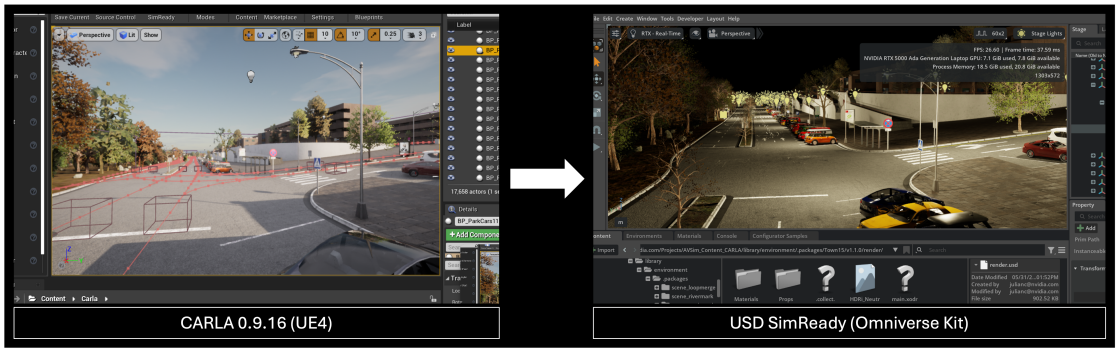


**Figure 1.2:** Official CARLA SimReady USD Exporter [24]

## 1.3 Scope and Limitations

While advancing synthetic data generation capabilities, several intentional scope limitations should be acknowledged:

- **Infrastructure Deployment Scope**: The framework focuses on intersection-based scenarios and traffic monitoring where fixed sensors provide clear advantages [19]. The sensor placement tool is optimized for urban intersection monitoring.
- **Visual Fidelity as Foundation**: The CARLA-Isaac Sim integration establishes technical infrastructure rather than delivering immediate photorealistic improvements. The focus is on robust data transformation, temporal accuracy, and format compatibility. The late release of CARLA's official USD export limited time for rendering optimization.
- **Performance Considerations**: Both enhancements introduce computational overhead. Infrastructure sensors increase concurrent data streams, while USD export and Isaac Sim replay require substantial GPU resources. Current implementations are optimized for research applications.
- **Sensor Modality Coverage**: Current implementations support RGB cameras and LiDAR. While extensible to additional modalities (radar, thermal, event cameras), these are not currently implemented.

## 1.4 Expected Impact and Applications

The contributions enable several important applications:

- **Enhanced Validation Coverage**: Infrastructure-cooperative sensing enables validation of complex scenarios with severe occlusions, multi-perspective validation, wide-area monitoring, and V2I testing [25, 9].
- **Improved Training Data Diversity**: Infrastructure perspectives provide greater diversity with elevated viewpoints reducing occlusions, complementary coverage minimizing blind spots, and multi-modal fusion scenarios for robust algorithms [10].
- **Flexible Validation Workflows**: CARLA-Isaac Sim integration enables functional testing in CARLA followed by visual enhancement in Isaac Sim, comparative studies, and future integration of additional tools.

- **Foundation for Future Enhancements**: Both contributions are extensible to additional sensor modalities, deployment scenarios, advanced rendering techniques, domain randomization, and emerging technologies [26].

The methodologies, tools, and frameworks are designed to be practical and applicable to real-world ADAS validation while maintaining compatibility with established standards, ensuring gradual integration into industrial workflows.

# Chapter 2

# Literature Review and Related Work

ADAS validation is a critical challenge as autonomous vehicle systems grow in complexity and assume greater control responsibilities [5]. This chapter examines current ADAS validation approaches, synthetic data generation technologies, infrastructure-cooperative sensing paradigms, and simulation-to-reality gap challenges, establishing the theoretical foundation for this thesis's contributions.

## 2.1 ADAS Validation Methodologies

### 2.1.1 Traditional Testing Paradigms and Limitations

ADAS validation historically follows a hierarchical X-in-the-Loop approach, progressing from Model-in-the-Loop (MiL) through Software-in-the-Loop (SiL), Hardware-in-the-Loop (HiL), to Vehicle-in-the-Loop (ViL) testing. The European General Safety Regulation (GSR) [1], implemented in phases from July 2022 to January 2029, mandates comprehensive validation for systems including Intelligent Speed Assistance, Emergency Lane Keeping, Advanced Emergency Braking, and Driver Drowsiness Warning.

Traditional validation faces fundamental scalability challenges. Physical testing provides the highest fidelity but requires extensive resources to cover operational scenarios. RAND Corporation research demonstrates that autonomous vehicles would need hundreds of millions—potentially billions—of miles of driving to achieve statistically significant safety validation [3], rendering purely physical testing impractical.

### 2.1.2 Regulatory Framework and Standardization

The regulatory landscape has evolved from informal guidelines to comprehensive protocols. Euro NCAP has established detailed testing procedures for AEB, Lane Support Systems, and Speed Assistance [27]. International efforts include UNECE Global Technical Regulation No. 22 for driver monitoring, ISO 26262 [28] for functional safety, and ISO 21448 (SOTIF) [29] addressing system limitations, edge cases, and sensor performance in complex scenarios.

## 2.2 Synthetic Data Generation for Autonomous Driving

### 2.2.1 Evolution of Simulation Platforms

Autonomous driving simulation has evolved from early physics-based simulators to sophisticated platforms capable of high-fidelity multi-modal sensor simulation. CARLA Simulator [6], built on Unreal Engine 4.26 (for the version 0.9.16 [24]), has emerged as a leading open-source platform with realistic physics, traffic management, environmental controls, and multi-sensor support (cameras, LiDAR, radar). Its client-server architecture enables distributed data generation.

LGSVL Simulator [30] emphasized photorealistic rendering and integration with Apollo and Autoware, though development was discontinued in 2022. Recent advances focus on visual fidelity through neural rendering techniques like NeuRAD [26] and S-NeRF [31], which leverage neural radiance fields for photorealistic scene generation, though at significant computational cost.

### 2.2.2 The nuScenes Dataset Format

The nuScenes dataset [8], introduced by Motional, has become the de facto standard for autonomous driving perception benchmarks. It comprises over 1000 20-second driving scenes from Boston and Singapore with challenging urban environments. The sensor suite includes six cameras providing 360-degree coverage, one spinning LiDAR (Velodyne HDL-32E), and five radar sensors.

The annotation schema supports 23 object classes with detailed 3D bounding boxes, tracking identities, and visibility states, provided at 2 Hz. The standardized database schema using relational JSON structures facilitates integration across data sources and enables transfer learning between synthetic and real datasets [32, 33].

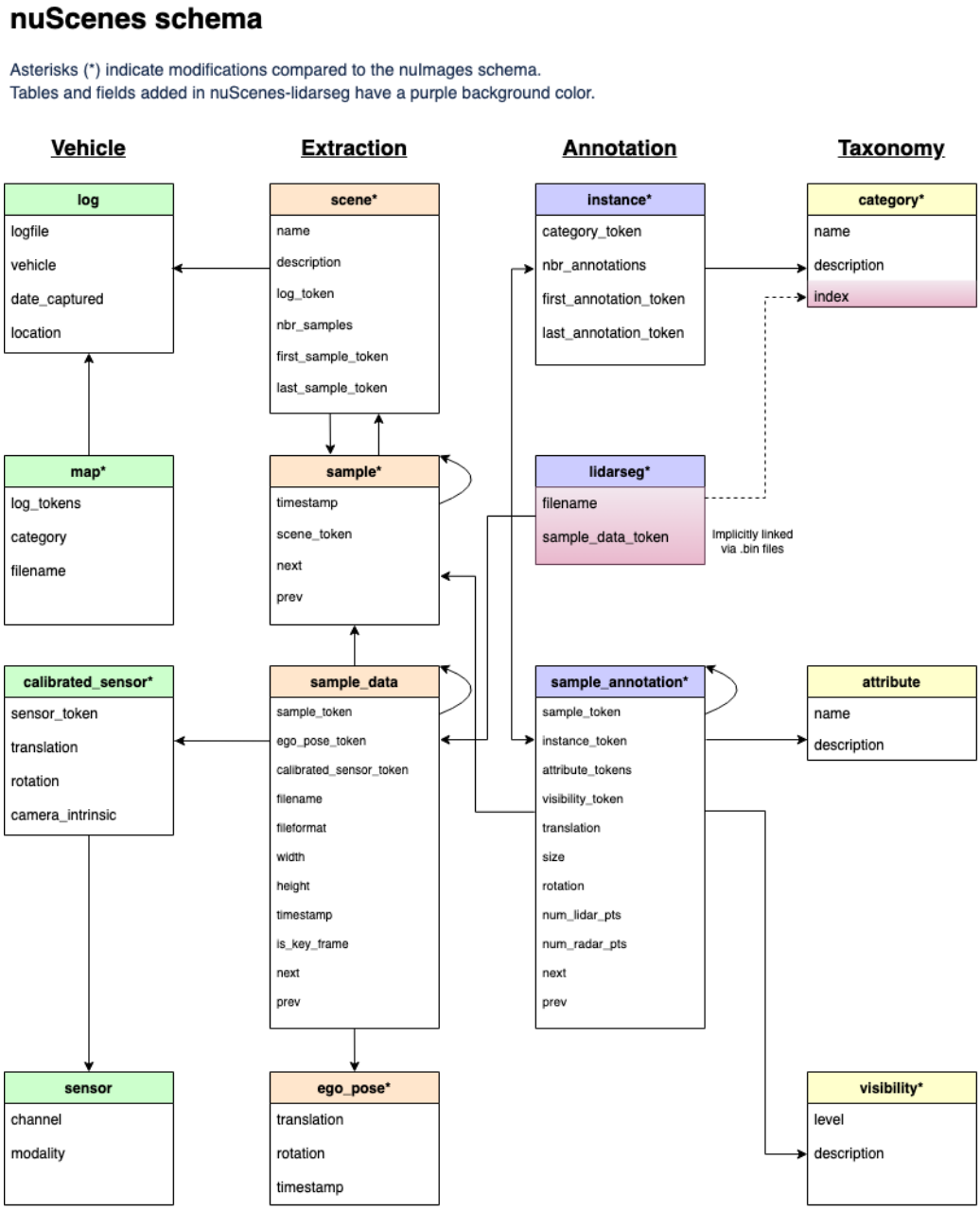


**Figure 2.1:** nuScenes Schema [8]

### 2.2.3 Photorealistic Rendering Technologies

Physically-based rendering (PBR) approaches provide high-fidelity synthetic imagery through accurate material properties, light transport simulation, and atmospheric effects [34]. Advanced techniques like path tracing and photon mapping enable complex lighting phenomena including reflections, refractions, and global illumination.

Hardware-accelerated ray tracing through NVIDIA RTX technology [35] has transformed photorealistic rendering from offline to interactive capability. Studies show synthetic datasets using ray-traced rendering exhibit significantly improved transfer learning performance compared to traditional rasterization [12].

### 2.2.4 Temporal Consistency and Deterministic Replay

Temporal consistency is critical for training algorithms that process sequential data. Modern ADAS systems rely heavily on temporal information for tracking, motion prediction, and scene understanding. Advanced animation systems supporting keyframe interpolation and motion capture enable realistic movement representations [15].

Deterministic replay systems enable exact scenario reproduction, facilitating reproducible benchmarking and regression testing. The Universal Scene Description (USD) format, developed by Pixar, provides a standardized framework for representing animated 3D scenes with temporal consistency guarantees.

### 2.2.5 Simulation-to-Reality Gap

Despite advances, the simulation-to-reality gap persists across visual domain differences, sensor characteristic variations, and behavioral discrepancies [36]. Models trained exclusively on synthetic data often exhibit degraded real-world performance, particularly in challenging conditions.

Contemporary approaches include domain randomization [37], which varies rendering parameters to encourage robust learning; adversarial training [38] using GANs to align feature distributions; and style transfer methods that translate synthetic imagery while preserving semantic content.

## 2.3 Infrastructure-Cooperative Sensing Systems

### 2.3.1 V2X Communication Technologies

Vehicle-to-Everything (V2X) communication encompasses V2V, V2I, V2P, and V2N modes. Cellular V2X (C-V2X), standardized through 3GPP Release 14

[39], provides direct communication (PC5) for low-latency safety applications and network-based communication (Uu) for broader connectivity. Integration with 5G promises enhanced bandwidth, reduced latency, and improved reliability.

V2X enables cooperative perception architectures extending beyond vehicle-centric approaches. By sharing sensor data and perception outputs, cooperative systems extend sensing ranges, overcome occlusions, and provide redundant detections [11].

### 2.3.2 Cooperative Perception Architectures

Cooperative perception systems employ various architectural approaches:

- **Early Fusion**: Shares raw sensor data or low-level features, providing maximum information but requiring substantial bandwidth.
- **Late Fusion**: Shares high-level detections, reducing bandwidth requirements but potentially losing valuable low-level information.
- **Intermediate Fusion**: Shares feature-level representations, balancing information content with communication efficiency [10].

### 2.3.3 Infrastructure-Based Sensing and Datasets

Infrastructure-mounted sensors provide fundamentally different capabilities through elevated positions offering wider fields of view, reduced occlusions, and persistent coverage of critical areas. The DAIR-V2X dataset [25] provides the first large-scale real-world dataset for vehicle-infrastructure cooperative 3D detection with 464,000 annotations. TUMTraf V2X [40] extends this with Munich traffic data.

Optimal infrastructure sensor placement requires considering traffic patterns, intersection geometry, communication constraints, and line-of-sight requirements [16].

## 2.4 Gaps in Current Research

### 2.4.1 Limited Infrastructure Sensing Support in Simulation

Despite recognized benefits of infrastructure-cooperative sensing, current simulation platforms provide limited support for infrastructure-mounted sensors. Existing frameworks, including Stella's work [7], primarily focus on vehicle-centric sensing. Researchers lack tools for configuring infrastructure placements, visualizing coverage, managing parameters, and generating datasets including both vehicle and infrastructure perspectives.

### 2.4.2 Visual Fidelity Limitations

While CARLA provides extensive functionality, its rendering capabilities exhibit limitations:

- **Material and Lighting**: Simplified models lacking real-world surface property complexity and full lighting phenomena [7].
- **Weather Effects**: Approximate representations of fog, rain, and nighttime illumination differing noticeably from reality.
- **Performance Trade-offs**: Computational constraints create fundamental trade-offs between speed and visual fidelity [30].

### 2.4.3 Fragmented Workflows

Fragmentation between simulation and rendering platforms creates workflow challenges. Manually reconstructing scenarios when transferring between platforms limits scalability and creates overhead. The integration gap between functional simulation (CARLA) and advanced rendering (NVIDIA Omniverse) represents a fundamental challenge.

## 2.5 Summary and Research Motivation

This review identifies three critical gaps motivating this thesis:

- **Infrastructure Sensing Gap**: Lack of comprehensive infrastructure-cooperative sensing support in simulation frameworks, limiting evaluation of next-generation cooperative perception systems.
- **Visual Fidelity Gap**: Open-source simulator rendering limitations contributing to the simulation-to-reality gap and potentially impacting synthetic training data quality.
- **Integration Gap**: Fragmentation between functional simulation and advanced rendering engines constraining practical application of photorealistic techniques.

The following chapters present contributions addressing these gaps through:

1. An infrastructure-cooperative sensor simulation framework (Chapter 4) extending Stella's vehicle-centric approach with infrastructure sensing capabilities, sensor placement tools, and nuScenes extensions maintaining backward compatibility.

2. A CARLA-to-Isaac Sim integration pipeline (Chapters 5-6) establishing infrastructure for exporting CARLA simulations to NVIDIA Omniverse, enabling deterministic replay and providing a foundation for visual fidelity enhancements.

These contributions maintain practical applicability through compatibility with established formats, incremental enhancement of existing frameworks, and pragmatic adaptation to ecosystem developments such as CARLA's official USD export support.

# Chapter 3

# Baseline Framework: Vehicle-Centric Synthetic Data Generation

This chapter presents Federico Stella's foundational framework [7], which provides baseline capabilities for vehicle-mounted sensor simulation and nuScenes-compatible data generation in CARLA. Understanding this system is essential for contextualizing the infrastructure-cooperative sensing and Isaac Sim integration contributions in subsequent chapters.

Stella's vehicle-centric approach successfully demonstrated feasibility of generating high-quality synthetic sensor data, implementing precise sensor synchronization, nuScenes-compatible organization, and flexible configuration interfaces. However, the paradigm exhibits limitations in sensor perspective diversity, coverage completeness, and visual fidelity that constrain applicability for next-generation cooperative sensing systems. These limitations motivate this thesis's two major contributions: the infrastructure-cooperative sensor framework (Chapter 4) and the CARLA-to-Isaac Sim integration pipeline (Chapters 5-6).

## 3.1 CARLA Simulator Architecture

### 3.1.1 Overview and Design Philosophy

CARLA (Car Learning to Act) is an open-source simulator designed for autonomous urban driving research [6]. Built on Unreal Engine 4.26 (for the 0.9.16 version [24]), CARLA provides realistic, customizable virtual environments supporting training, prototyping, and validation of autonomous systems. The simulator enables flexible configuration of environmental conditions, traffic scenarios, and sensor specifications,

generating critical signals including GPS coordinates, IMU measurements, LiDAR and radar point clouds, camera imagery, and collision detection.

CARLA's design emphasizes modularity and extensibility, enabling customization of virtually every simulation aspect. The platform supports diverse research applications from perception algorithm training to end-to-end driving policy learning, with particular strength in generating large-scale labeled datasets.

### 3.1.2 Client-Server Architecture

CARLA implements a client-server architecture where the server manages the simulation environment while clients interact through a comprehensive API [6].

The **server component**, built on Unreal Engine 4, handles:

- **Physics Simulation**: NVIDIA PhysX integration provides realistic vehicle dynamics, collision detection, and rigid body interactions
- **Scene Rendering**: Unreal Engine's pipeline generates sensor imagery with configurable quality settings
- **Sensor Data Generation**: Processes sensor models to produce synchronized outputs
- **Environmental State Management**: Tracks all actors, positions, velocities, and states for deterministic reproduction

The **client component**, typically implemented in Python, communicates via TCP. Clients control simulation execution, spawn and configure actors, place and configure sensors, retrieve sensor data through callbacks, and manage environmental conditions.

### 3.1.3 Synchronous Operation Mode

For multi-sensor data collection, temporal synchronization is critical. CARLA supports two modes:

- **Asynchronous Mode** (default): Server executes simulation as rapidly as possible, maximizing throughput but introducing temporal inconsistencies
- **Synchronous Mode**: Server advances only when commanded by client through `world.tick()`. This guarantees perfect temporal alignment across all sensors

Stella's framework implements synchronous mode to ensure data collection integrity. The synchronous workflow ensures each sensor's data corresponds to the same simulation timestamp, preventing temporal misalignment that would compromise multi-modal sensor fusion.

## 3.2 Vehicle-Centric Sensor Configuration

### 3.2.1 nuScenes Sensor Suite Replication

Stella's primary design goal was replicating the exact sensor configuration used in the real-world nuScenes dataset [8], ensuring synthetic data maintains compatibility with established analysis tools and validation workflows.

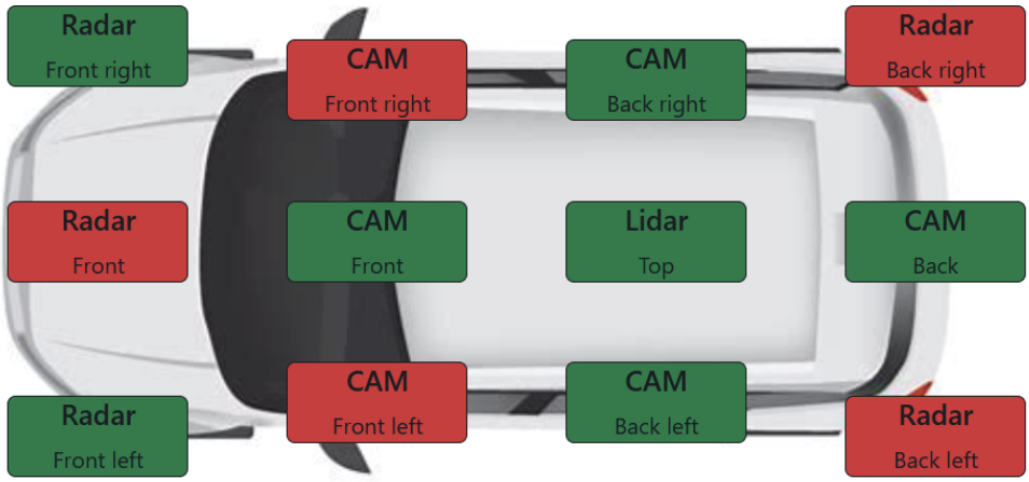


**Figure 3.1:** Ego Vehicle sensor placement [7]

#### Camera Configuration

Six RGB cameras provide comprehensive coverage:

- **CAM_FRONT**: Forward-facing, primary view for obstacle detection
- **CAM_FRONT_LEFT/RIGHT**: Side-forward cameras for intersections and turns
- **CAM_BACK_LEFT/RIGHT**: Rear-quarter cameras for blind spots and lane changes
- **CAM_BACK**: Rear-facing for rearward awareness

Each camera is configured with intrinsic parameters including resolution (1600 x 900 pixels), field of view, and lens distortion parameters matching nuScenes specifications.

#### LiDAR Configuration

The **LIDAR_TOP** sensor, roof-mounted, provides 3D point cloud data with vertical channels (32 or 64), rotation frequency, points per second, and maximum range. CARLA uses ray-casting to simulate light detection, computing intersections between emitted rays and scene geometry.

#### Supplementary Sensors

Additional sensors include:

- **RADAR**: Provides range-rate information through Doppler simulation
- **IMU**: Supplies acceleration and angular velocity measurements
- **GNSS**: Provides global position estimates

### 3.2.2 Sensor Attachment and Coordinate Frames

All sensors attach to the ego vehicle using CARLA's transform system. Each sensor is defined by a six-degree-of-freedom pose (position vector and rotation vector) relative to the vehicle's coordinate frame. This attachment ensures sensors move rigidly with the vehicle, maintaining relative positions throughout simulation.

This paradigm creates naturally ego-centric datasets but inherently limits the ability to capture stationary perspectives or maintain persistent monitoring of fixed locations—motivating the infrastructure sensor extensions in Chapter 4.

## 3.3 Synchronous Data Collection Pipeline

### 3.3.1 Callback-Based Data Acquisition

Stella's framework implements a callback-based architecture balancing performance with temporal consistency. Each sensor registers a callback function executing when new data becomes available. When simulation advances via `world.tick()`, the server generates sensor data and invokes registered callbacks. Each callback receives raw data, processes it (including coordinate transformations and format conversions), and buffers processed data to prevent disk I/O from blocking subsequent operations. [7]

A synchronization barrier ensures all sensor callbacks complete before the next simulation tick, guaranteeing temporal coherence across sensor modalities. [7]

### 3.3.2 Data Processing and Transformation

Raw sensor data requires several processing steps [7]:

- **Camera Data**: RGB images converted from BGRA to RGB format, encoded into PNG/JPEG, with intrinsic calibration matrices computed and metadata associated
- **LiDAR Data**: Point clouds transformed from CARLA's left-handed coordinate system to nuScenes' right-handed convention and encoded into binary formats.
- **Coordinate System Transformations**: Critical transformations manage differences between CARLA (left-handed, Z-up, centimeters) and nuScenes (right-handed, Z-up, meters). The pipeline implements axis remapping $(X, Y, Z)_{\text{CARLA}} \rightarrow (Y, -X, Z)_{\text{nuScenes}}$ with appropriate scaling.

### 3.3.3 Ground Truth Annotation

Synthetic data generation provides perfect ground truth annotations without manual labeling. During each tick, the framework queries CARLA for all actors and their properties: semantic category, 3D bounding box, position, orientation, velocity, and additional attributes. This information is automatically processed to generate nuScenes-compatible annotations with coordinate transformations and unique instance identifiers for tracking. [7]

## 3.4 nuScenes Format Integration

### 3.4.1 Dataset Structure and Organization

The nuScenes dataset format [8] defines a comprehensive organizational structure for multi-modal autonomous driving data. The hierarchical structure (Figure 2.1) includes:

- **scene**: Continuous driving sequence (~20 seconds)
- **sample**: Temporal keyframes at 2 Hz
- **sample_data**: Individual sensor outputs with file paths, timestamps, ego-vehicle pose, and calibration references
- **sensor**: Sensor definitions (camera, lidar, radar)
- **calibrated_sensor**: Intrinsic and extrinsic calibration information

- **ego_pose**: Vehicle pose in global coordinate frame (2D localization in x–y; z=0)
- **instance**: Enumeration of object instances (not preserved across scenes)
- **sample_annotation**: 3D bounding boxes, attributes, visibility, and temporal links (prev/next) per instance at each sample
- **category**: Taxonomy of object classes and sub-classes
- **attribute**: Dynamic properties (e.g., vehicle parked/stopped/moving; cycle with/without rider)
- **log**: Acquisition metadata (vehicle, date, location)
- **map**: Semantic top-down masks (e.g., drivable surface, sidewalk)
- **visibility**: Discrete visibility levels (0–40%, 40–60%, 60–80%, 80–100%)
- **lidarseg**: Links to per-point lidar semantic labels (.bin files) for keyframes

This relational structure enables efficient querying and ensures compatibility with nuScenes-devkit tools.

### 3.4.2 Calibration and Ego Pose Management

The framework generates comprehensive calibration data:

**Camera Intrinsic Calibration**: Camera intrinsic matrices $K$ define projection from 3D camera coordinates to 2D image coordinates:

$$K = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$$

**Extrinsic Calibration**: Defines rigid transformation from sensor to vehicle coordinates using rotation matrix $R$ and translation vector $t$.

**Ego Pose Tracking**: Records vehicle pose in global frame at each timestamp: position $(x, y, z)$ and orientation quaternion $(q_w, q_x, q_y, q_z)$.

## 3.5 Environmental Simulation Capabilities

### 3.5.1 Dynamic Weather and Lighting Control

CARLA provides comprehensive environmental control with parameters including precipitation, fog density, wetness, wind, sun position, and cloudiness. The lighting system implements physically-based rendering with accurate sun/sky models, street lights, vehicle lights, and indirect lighting from environmental reflections. [6]

### 3.5.2 Traffic and Scenario Generation

CARLA's Traffic Manager orchestrates NPCs and pedestrians, implementing lane-keeping behaviors, traffic light compliance, overtaking maneuvers, and collision avoidance. The system supports diverse traffic densities and enables scripting of specific scenarios including emergency braking, cut-in maneuvers, intersection conflicts, and pedestrian crossings. Scenarios can be deterministically reproduced through synchronous mode combined with fixed random seeds. [6]

## 3.6 Limitations of the Vehicle-Centric Approach

While Stella's framework successfully established comprehensive capabilities, several limitations constrain applicability for next-generation ADAS validation:

### 3.6.1 Coverage Limitations and Blind Spots

Vehicle-mounted sensors are constrained by their mobile platform's position. Dense urban intersections create extensive occlusion zones where critical areas remain unobserved until the vehicle is committed to a maneuver. Limited predictive context prevents generation of datasets for long-horizon prediction algorithms. Fixed sensor configurations represent compromises rather than optimal placements for specific scenarios.

### 3.6.2 Lack of Infrastructure Sensing Perspectives

The vehicle-centric paradigm lacks elevated and strategically positioned perspectives from infrastructure-mounted sensors, which offer persistent monitoring of specific locations, elevated vantage points reducing occlusions, and complementary external perspectives. This prevents validation of cooperative perception algorithms and V2I communication systems—motivating Chapter 4's infrastructure-cooperative sensor framework.

### 3.6.3 Visual Rendering Fidelity Constraints

CARLA's visual fidelity exhibits limitations contributing to simulation-to-reality gap:

- **Material and Lighting**: Simplified models lacking real-world surface complexity [7]
- **Texture and Detail**: Lower-resolution textures and simplified geometry
- **Weather Effects**: Incomplete rendering of complex photometric interactions

These limitations affect vision-based perception algorithms, motivating the CARLA-to-Isaac Sim integration in Chapters 5-6.

### 3.6.4 Computational Performance Constraints

The comprehensive sensor suite imposes significant computational demands. Camera rendering overhead can reduce frame rates by up to 70

## 3.7 Summary and Transition

This chapter presented Stella's comprehensive vehicle-centric framework, demonstrating feasibility and utility of synthetic data for ADAS validation. However, identified limitations in coverage, perspective diversity, visual fidelity, and computational performance motivate this thesis's two contributions:

1. The **infrastructure-cooperative sensor framework** (Chapter 4) extends the vehicle-centric approach with fixed infrastructure sensors, addressing coverage limitations

2. The **CARLA-to-Isaac Sim integration pipeline** (Chapters 5-6) establishes infrastructure for enhanced visual fidelity through advanced rendering

Both contributions maintain full backward compatibility with Stella's baseline framework, ensuring existing workflows remain functional while enabling new capabilities.

# Chapter 4

# Infrastructure-Cooperative Sensor Simulation

## 4.1 Introduction and Motivation

The infrastructure-cooperative sensor framework represents the first major contribution of this thesis, introducing a fundamental evolution from Federico Stella's vehicle-centric approach [7] to a hybrid sensing system that integrates mobile vehicle-mounted sensors with strategically positioned fixed infrastructure sensors. This chapter presents the technical architecture, implementation details, and integration methodologies that enable cooperative sensing scenarios within the CARLA simulation environment.

While Stella's framework established comprehensive capabilities for vehicle-mounted sensor simulation and nuScenes-compatible dataset generation, it inherently limited validation scenarios to perspectives achievable from the ego vehicle. This constraint becomes particularly problematic in complex urban intersections where occlusions, limited sensor range, and geometric constraints can compromise detection capabilities. Infrastructure-mounted sensors address these limitations by providing persistent, elevated vantage points that complement mobile vehicle sensors, enabling validation of cooperative perception algorithms and vehicle-to-infrastructure (V2I) communication systems.

The framework consists of three integrated components: (1) a real-time sensor placement management tool that provides intuitive graphical control over infrastructure sensor deployment with automatic intersection detection capabilities, (2) infrastructure sensor spawning mechanisms that complement vehicle-mounted sensors in the CARLA simulation, and (3) extensions to the nuScenes data pipeline that maintain backward compatibility while supporting infrastructure sensor data collection. This modular architecture ensures that the infrastructure sensing capabilities

enhance rather than replace the existing vehicle-centric framework.

## 4.2 Sensor Placement Tool

### 4.2.1 Tool Overview and Architecture

The sensor placement management tool implements a real-time interactive application that bridges CARLA's simulation environment with intuitive graphical controls for infrastructure sensor deployment. The tool provides a comprehensive interface for managing infrastructure sensors throughout their lifecycle: from initial placement and configuration, through real-time visualization and verification, to final configuration persistence and data collection deployment.

The tool architecture follows a model-view-controller pattern where the CARLA world represents the model, the Pygame interface provides the view, and the event handling system serves as the controller. The tool operates through a continuous event loop that maintains synchronization between the graphical interface and the CARLA simulation state. At each iteration, the system: (1) queries the CARLA spectator position and orientation to determine the current viewport, (2) processes user input events including keyboard commands and mouse interactions, (3) updates the graphical interface to reflect current sensor placements and system state, and (4) executes requested actions such as sensor placement or configuration saving. The 30 FPS target provides smooth visual feedback without overwhelming the system with excessive update frequency.

The complete system architecture integrates the CARLA simulator, the sensor placement management tool, the configuration persistence layer, and the nuScenes data generation pipeline. The architecture maintains strict separation between the interactive sensor placement phase and the data collection phase, ensuring that infrastructure sensor configurations can be established independently and reused across multiple simulation runs.

### 4.2.2 User Interface Components

The graphical interface consists of four integrated panels that provide comprehensive visualization and control capabilities. Figure 4.1 shows the main interface with deployed sensors, Figure 4.2 shows the sensor perspective view feature, and Figure 4.4 demonstrates the intersection management overlay.

The **Top View Panel** displays a 300×300 pixel minimap representation of the current CARLA map, loaded from pre-generated PNG images stored in the project directory structure. The minimap provides real-time visualization of the spectator position as a cyan marker with directional indication, showing the user's current location and orientation within the simulation environment. Deployed sensors

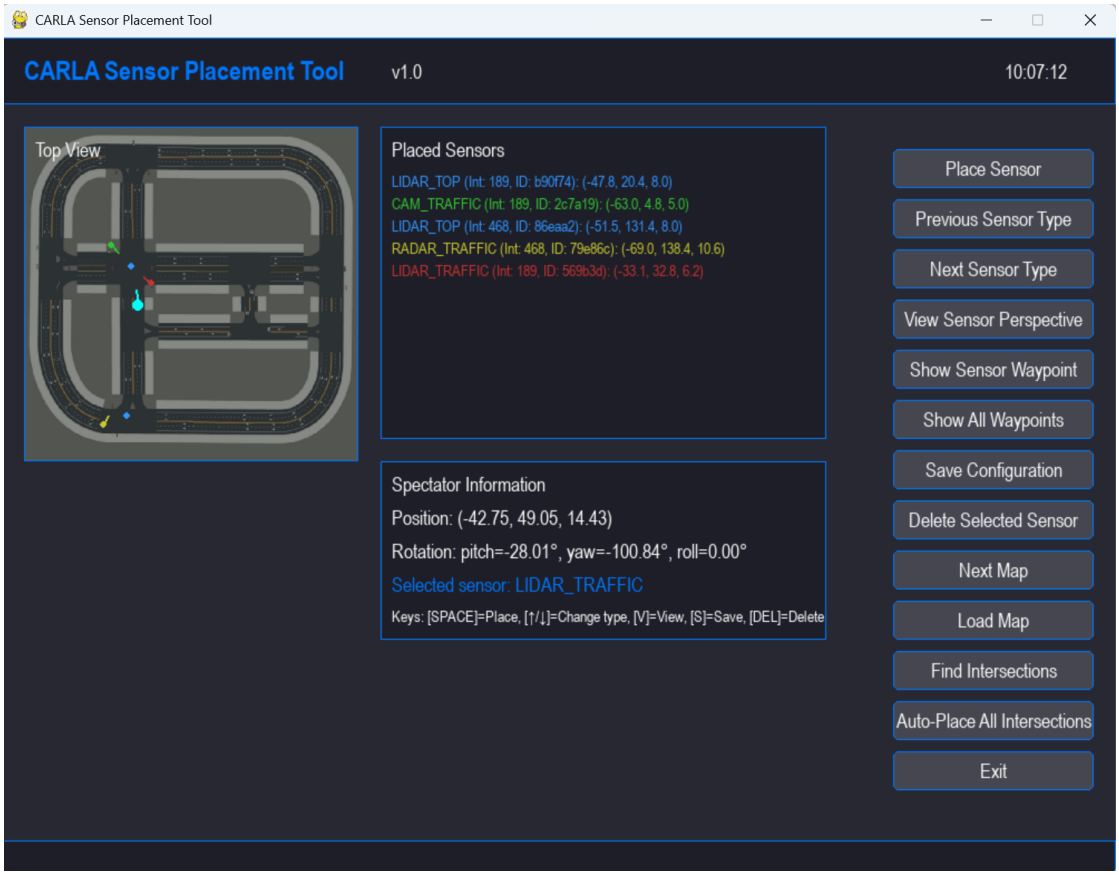


**Figure 4.1:** Main interface of the CARLA Sensor Placement Tool


**Figure 4.2:** Sensor perspective view overlay

appear as colored markers indicating their types and positions. The minimap updates at each frame to reflect spectator movement and sensor placement actions. Figure 4.3 shows a detailed view of the minimap panel with deployed sensors.


**Figure 4.3:** Top View minimap panel

The **Placed Sensors Panel** maintains a scrollable list of all deployed infrastructure sensors on the current map. Each entry displays the sensor type, associated intersection ID, unique sensor identifier, and world coordinates. The panel uses color coding to distinguish sensor types: cyan for LIDAR_TOP, green for CAM_TRAFFIC, red for LIDAR_TRAFFIC, and yellow for RADAR_TRAFFIC. Users can select sensors from this list to view their perspectives, remove sensors, or visualize their coverage areas in the CARLA 3D view.

The **Spectator Information Panel** displays real-time feedback about the current spectator state, including world position coordinates (x, y, z), rotation angles (pitch, yaw, roll), and the currently selected sensor type for placement. This panel provides essential context for understanding the sensor placement operation and verifying coordinate accuracy.

The **Control Button Panel** provides access to all tool functionality through a vertical array of interactive buttons. Key functions include sensor placement, sensor type selection, sensor view activation, configuration saving, sensor deletion, intersection detection, and automatic placement. The button interface implements hover effects and visual feedback to enhance usability.

### 4.2.3 Supported Sensor Types

The system supports four primary sensor types specifically designed for infrastructure deployment, each optimized for different monitoring requirements:

- **CAM_TRAFFIC**: RGB cameras positioned at traffic intersections for visual monitoring. Default configuration includes 800×600 pixel resolution and 90-degree horizontal field of view. These sensors provide photorealistic image data suitable for vision-based perception algorithm validation and can be positioned at any height according to monitoring requirements.
- **LIDAR_TRAFFIC**: Directional LiDAR sensors with 100-meter range, 32 vertical channels, and 56,000 points per second sampling rate. These sensors provide focused coverage of specific intersection approaches, complementing the omnidirectional LIDAR_TOP sensors with higher point density in targeted directions.
- **RADAR_TRAFFIC**: Traffic monitoring radar with 30-degree horizontal FOV, 10-degree vertical FOV, and 100-meter detection range. Radar sensors provide robust detection under adverse weather conditions and enable velocity measurement for approaching vehicles.
- **LIDAR_TOP**: 360-degree elevated LiDAR sensors positioned at 8 meters above ground level, providing complete intersection coverage. These sensors replace the ego vehicle's top-mounted LIDAR in the nuScenes annotation pipeline while offering superior coverage from stationary elevated positions. The LIDAR_TOP configuration includes 60-meter range, 32 channels, 150,000 points per second, and 10 Hz rotation frequency.

## 4.3 Tool Features and Functionality

### 4.3.1 Intersection Detection

The intersection detection system implements automatic identification of strategic sensor placement locations by analyzing CARLA's road network topology. The algorithm extracts junction information from the map's waypoint graph and computes geometric properties essential for sensor placement decisions. Listing A.1 presents the core intersection detection algorithm.

The algorithm processes the map topology in three stages. First, it identifies all waypoints that belong to junction areas by querying the `is_junction` property of each waypoint in the topology graph. Second, it groups waypoints by their associated junction ID to handle junctions that may have multiple waypoint representations. Third, it calculates geometric properties for each unique junction,

including the center point (computed as the centroid of all junction waypoints), bounding box dimensions (derived from the minimum and maximum waypoint coordinates), and a characteristic size metric (used for sensor placement scoring).

This automated approach ensures consistent intersection identification across different CARLA map environments without requiring manual annotation. The geometric information enables intelligent sensor placement algorithms and provides spatial context for sensor association during data collection.

Figure 4.4 shows the intersection management interface that provides comprehensive control over detected intersections, allowing users to highlight specific intersections in the 3D view or directly place sensors at selected locations.

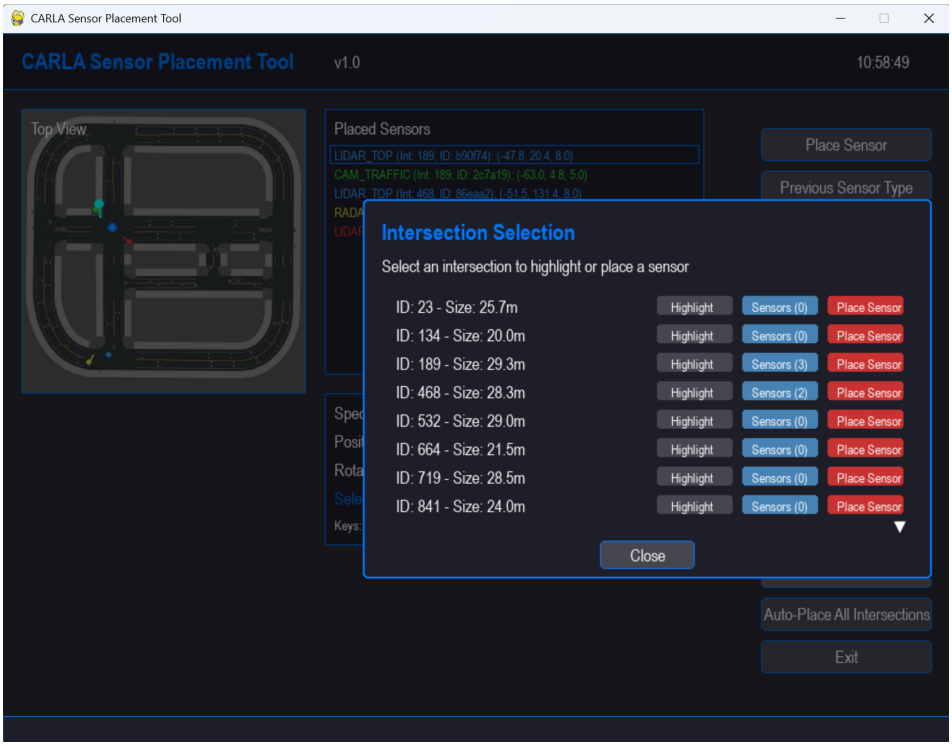


**Figure 4.4:** Intersection Selection overlay

### 4.3.2 Intelligent Sensor Placement

The sensor placement algorithm implements a sophisticated scoring mechanism that evaluates optimal sensor-to-intersection associations based on both proximity and directional alignment. When a user places a sensor at a specific location and orientation (controlled through the spectator position), the system automatically determines which intersection the sensor should monitor.

The algorithm calculates a composite score for each detected intersection using the formula:

$$\text{Score} = (0.7 \times \text{Direction\_Score}) + (0.3 \times \text{Distance\_Score}) \tag{4.1}$$

where the Direction_Score represents the dot product between the sensor's forward vector (derived from spectator rotation) and the normalized vector pointing toward the intersection center, and the Distance_Score provides normalized proximity weighting calculated as $1.0 - (\text{distance}/\text{max\_distance})$ with a maximum consideration distance of 100 meters.

The scoring mechanism prioritizes directional alignment over pure proximity through empirically determined weights (0.7 for direction, 0.3 for distance). This design decision ensures that sensors are associated with intersections that align with their intended monitoring direction rather than simply the closest available intersection. The algorithm selects the intersection with the highest combined score for sensor assignment, enabling intuitive sensor placement where users position the spectator camera to define both sensor location and viewing direction.

### 4.3.3 Real-time Visualization in CARLA

The tool maintains continuous bidirectional communication with the CARLA simulator through the Python API. This integration enables real-time queries of world state, dynamic sensor spawning for preview purposes, and visual debugging through CARLA's built-in debug drawing system.

The debug drawing system visualizes sensor placements by rendering temporary markers, arrows, and lines directly in the 3D simulation view. For each sensor, the system draws a position marker, an orientation arrow indicating the sensor's forward direction, and optional coverage cones for directional sensors. When a sensor is associated with an intersection, the system also draws a connection line between the sensor location and the intersection center, providing clear visual feedback about sensor-intersection relationships.

The CARLA debug drawing system provides temporary visual elements that render directly in the 3D simulation view, enabling users to visualize sensor placements, coverage areas, and intersection associations without permanently modifying the world. The `life_time` parameter controls how long these debug visualizations persist, with typical values of 5.0 seconds providing sufficient visibility without cluttering the view.

Figure 4.5 demonstrates the intersection highlighting feature in the CARLA 3D view, while Figure 4.6 shows the visual feedback provided for deployed sensors.

For sensor preview functionality, the tool temporarily spawns actual CARLA sensor actors at selected positions to capture real-time images from their perspectives. This capability enables users to verify sensor placement effectiveness before


**Figure 4.5:** Intersection highlighting in CARLA 3D view


**Figure 4.6:** Real-time sensor visualization in CARLA 3D view

committing configurations. The preview system maintains proper lifecycle management for temporary sensor actors, ensuring that preview sensors are destroyed when no longer needed to prevent resource leaks. The image processing callback converts CARLA's raw image data into Pygame-compatible surfaces with appropriate color space conversions and geometric transformations.

### 4.3.4 Configuration Persistence

Infrastructure sensor configurations persist to JSON files organized by CARLA map name, enabling reuse of carefully designed sensor networks across multiple simulation sessions. The configuration format stores complete sensor specifications including type, position, orientation, associated intersection ID, and sensor-specific parameters. Listing A.2 demonstrates the JSON configuration structure.

The configuration system implements atomic write operations to ensure file consistency during save operations. The JSON format provides human-readable configurations enabling manual inspection, debugging, and external processing by analysis tools or batch processing scripts. Each sensor receives a unique identifier combining the sensor type, associated intersection ID, and a random hexadecimal suffix to prevent ID collisions.

Configuration loading occurs automatically when the tool starts or when users switch maps. The system filters configurations by map name to display only relevant sensors for the current environment. The atomic write operation using temporary files prevents corruption of existing configurations in case of interruption during save operations. The configuration system maintains map-specific files to organize sensor deployments by environment, facilitating reuse of carefully planned sensor networks across multiple simulation scenarios.

## 4.4 Web-Based Configuration Interface

While the Sensor Placement Tool provides real-time interactive sensor deployment within the CARLA 3D environment, the framework also includes a complementary web-based configuration interface that enables comprehensive simulation setup and sensor management through a browser-based application. This Driver-in-the-Loop frontend extends the original vehicle-centric interface [7] with dedicated support for infrastructure sensor configuration.

### 4.4.1 Fixed Sensors View

The web interface implements a dedicated "Fixed Sensors" view that provides centralized management of infrastructure sensor networks (Figure 4.7). This interface complements the Sensor Placement Tool by offering an alternative workflow

for users who prefer configuration through structured forms rather than interactive 3D placement.

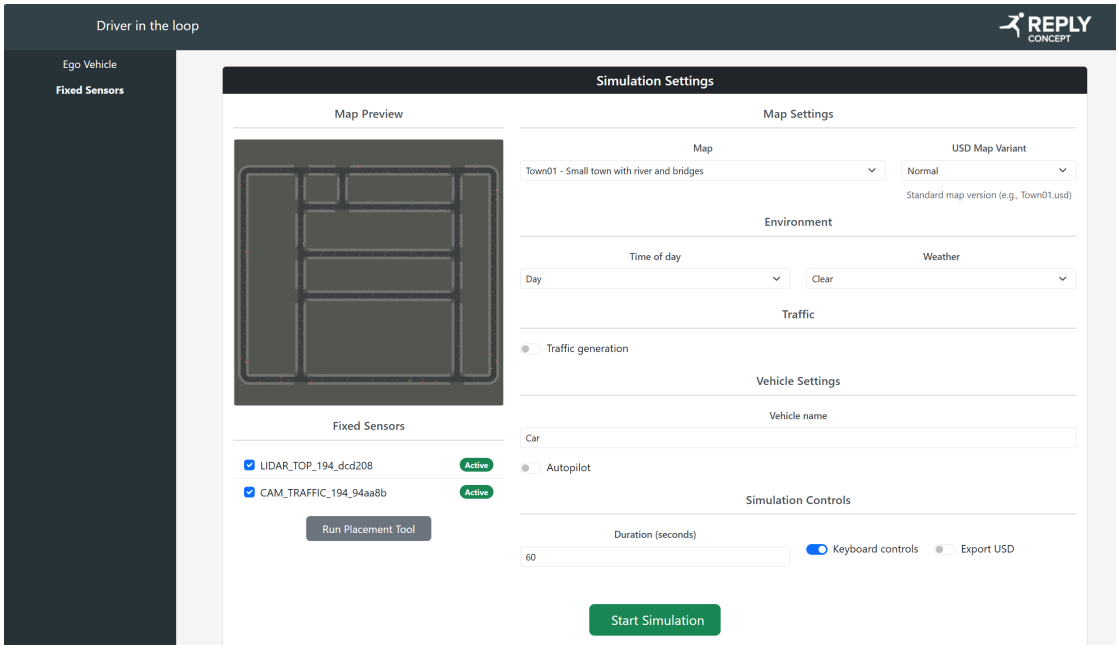


**Figure 4.7:** Web-based frontend interface for Fixed Sensors simulation

The Fixed Sensors panel displays the currently loaded infrastructure sensor configuration from the JSON files generated by the Sensor Placement Tool. Each sensor entry shows its type and unique identifier, with visual status indicators (green "Active" badges) confirming sensor availability for data collection. Users can toggle sensor activation, review sensor deployment across the map preview, and configure simulation parameters including traffic generation, environmental conditions, and USD export options.

The interface integrates seamlessly with the Sensor Placement Tool workflow: configurations created through the interactive 3D tool automatically appear in the web frontend's sensor list, while the frontend provides access to broader simulation parameters not available in the placement tool. The "Run Placement Tool" button enables direct launching of the interactive sensor placement application from within the web interface, facilitating smooth transitions between configuration approaches.

This dual-interface approach accommodates diverse user preferences and use cases. The interactive 3D tool excels for initial sensor placement with real-time visual feedback, while the web interface provides convenient access for configuration review, sensor activation management, and integration with broader simulation parameter setup. Both interfaces operate on the same underlying JSON configuration files, ensuring consistency regardless of which tool users employ for sensor

management.

## 4.5 Infrastructure Sensor Spawning

A critical distinction between infrastructure and vehicle-mounted sensors lies in their spawning mechanisms and lifecycle management within the CARLA simulation. Vehicle-mounted sensors attach to the ego vehicle actor, inheriting its transform and movement throughout the simulation. Infrastructure sensors, conversely, spawn as independent actors at fixed world coordinates, maintaining stationary positions regardless of vehicle movement. Listing A.3 demonstrates the spawning implementation differences.

The spawning distinction propagates through the entire data collection pipeline. Infrastructure sensors require absolute world coordinate specifications from the configuration files, while vehicle sensors use relative offsets from the ego vehicle's coordinate frame. This architectural separation enables mixing infrastructure and vehicle sensors within a single simulation, supporting cooperative sensing scenarios where both fixed and mobile perspectives contribute to the complete scene understanding.

Infrastructure sensors spawn once at simulation initialization and persist throughout the entire data collection session, providing consistent viewpoints across all recorded frames. Vehicle sensors, attached to the moving ego vehicle, spawn once but capture data from continuously changing positions as the vehicle traverses the environment. This fundamental difference in movement behavior necessitates distinct coordinate transformation procedures in the nuScenes data generation pipeline.

## 4.6 Integration with nuScenes Data Pipeline

### 4.6.1 Backward Compatibility Strategy

The infrastructure sensor framework maintains full compatibility with Federico Stella's existing nuScenes data generation pipeline [7, 8] while extending functionality to support fixed infrastructure sensors. The integration strategy preserves all existing vehicle-mounted sensor capabilities while adding infrastructure sensing without disrupting established workflows or data formats.

The framework extends the existing sensor management architecture through modular additions that complement rather than replace the original vehicle-centric approach. Infrastructure sensors utilize the same nuScenes data structures, file formats, and database schema established in the original implementation, ensuring seamless integration with existing analysis tools and validation pipelines.

The extension maintains the original nuScenes database tables including `sensor`, `calibrated_sensor`, and `sample_data` structures, with infrastructure sensors populating these tables using procedures adapted to handle world-fixed coordinate frames [8].

### 4.6.2 nuScenes Schema Extensions

To support infrastructure sensors within the existing nuScenes [8] framework, the implementation introduces a critical extension to the sensor metadata: the `intersection_id` field. This field associates each infrastructure sensor with its corresponding intersection, enabling spatial queries and multi-sensor fusion based on geographic context.

The `intersection_id` field stores the CARLA junction ID assigned during sensor placement, creating a persistent association between sensors and their monitored intersections. This association enables advanced validation scenarios including multi-sensor coverage analysis, intersection-specific performance evaluation, and cooperative perception algorithm testing. The extension modifies both the `sensor` and `calibrated_sensor` tables to include the intersection ID alongside traditional sensor metadata, ensuring that intersection associations propagate through the entire nuScenes data hierarchy.

The `calibrated_sensor` table receives similar extensions, storing the intersection association alongside traditional calibration parameters. This dual representation ensures that intersection associations propagate through the entire nuScenes data hierarchy.

### 4.6.3 Coordinate System Transformations

Infrastructure sensors require specialized coordinate transformation procedures to maintain consistency with the nuScenes format while handling world-fixed coordinate frames. Unlike vehicle-mounted sensors that use ego-vehicle-relative coordinates, infrastructure sensors operate in absolute world coordinates that remain constant throughout the simulation.

The transformation pipeline implements three distinct coordinate frames: (1) CARLA world coordinates (left-handed, Z-up), (2) nuScenes world coordinates (right-handed, Z-up), and (3) sensor-local coordinates for point cloud and image data. The transformation pipeline converts between these frames while preserving geometric relationships.

Listing A.4 presents the coordinate transformation implementation for infrastructure sensors.

The coordinate transformation implementation handles three critical aspects: (1) translation conversion between CARLA [6] and nuScenes world frames through

axis-specific operations, (2) rotation matrix transformation accounting for handedness differences between coordinate systems, and (3) preservation of sensor intrinsic parameters including camera intrinsic matrices for camera sensors. The transformation ensures that infrastructure sensor data integrates correctly with vehicle sensor data in the unified nuScenes dataset, maintaining geometric consistency across heterogeneous sensor sources.

For LiDAR sensors, the coordinate transformation extends to individual point clouds, converting each 3D point from CARLA's left-handed coordinate system to nuScenes' right-handed convention. This point-wise transformation preserves the relative spatial relationships within point clouds while aligning the global coordinate frame with nuScenes standards. The implementation maintains intensity values and other point attributes unchanged during coordinate transformation.

### 4.6.4 Data Collection Pipeline Integration

Infrastructure sensors integrate into the existing data collection pipeline through callback-based mechanisms that parallel vehicle-mounted sensor handling. The system extends Stella's synchronous data acquisition approach to support simultaneous collection from both vehicle-mounted and infrastructure sensors, maintaining temporal coherence across all sensor modalities.

During each simulation tick, the data collection pipeline executes a coordinated sequence: (1) freeze simulation state to ensure consistency, (2) trigger callbacks for all registered sensors (both vehicle-mounted and infrastructure), (3) process and buffer collected data with appropriate coordinate transformations, and (4) advance simulation only after all sensors confirm data capture completion. The pipeline applies different coordinate transformation procedures based on sensor mounting type—infrastructure sensors use world-fixed coordinates while vehicle sensors use ego-relative coordinates—ensuring proper integration into the unified nuScenes sample structure.

The data collection pipeline maintains proper separation between infrastructure and vehicle sensor processing while ensuring both integrate into the unified nuScenes sample structure. Each sample represents a temporal snapshot containing data from all sensors—both vehicle-mounted and infrastructure—captured at the same simulation timestamp.

### 4.6.5 File Organization and Naming Conventions

File organization follows the established nuScenes directory structure with extensions to accommodate infrastructure sensors. The system creates separate subdirectories for each sensor modality (cameras, LiDAR, radar) and uses consistent naming conventions that incorporate sensor type, intersection ID (for infrastructure

sensors), and timestamp information. Infrastructure sensor data files use the prefix pattern `SENSORTYPE_INTERSECTIONID_` to distinguish them from vehicle sensor files while maintaining compatibility with nuScenes loading utilities.

Sample data entries in the nuScenes database link to these files through the `filename` field, which stores relative paths from the dataset root. The system maintains the original nuScenes sample structure where each sample contains references to data from all available sensors at a specific timestamp. Infrastructure sensor sample_data entries appear alongside vehicle sensor entries within the same sample, with the `sensor_token` field distinguishing between different sensor sources.

This organization ensures compatibility with existing nuScenes data loading scripts while clearly identifying the source of each data file through the filename prefix and sensor channel designation.

# Chapter 5

# CARLA to USD Export Framework

## 5.1 Introduction and Motivation

The integration between CARLA simulation and NVIDIA Isaac Sim represents the second major contribution of this thesis, establishing a foundation for advanced synthetic data generation workflows. This framework addresses the need to bridge CARLA's functional simulation capabilities with environments capable of enhanced rendering quality. While CARLA excels in physics simulation, traffic management, and sensor modeling, the integration with NVIDIA Omniverse ecosystem opens pathways for future enhancements in rendering fidelity through Isaac Sim's advanced RTX-based rendering capabilities.

The primary contribution of this work lies not in achieving complete photo-realistic rendering, but rather in establishing the technical infrastructure and methodology for CARLA-to-USD export with temporal animation support. This framework provides a foundation upon which future work can build, including the integration of enhanced visual quality, improved material systems, and more sophisticated rendering techniques. The focus of this chapter is on the technical pipeline, data transformation accuracy, and temporal animation capabilities that enable deterministic simulation replay in Omniverse environments.

### 5.1.1 Motivation for USD-Based Integration

The autonomous driving industry increasingly recognizes the importance of high-fidelity synthetic data for training and validating perception systems. Modern computer vision algorithms, particularly deep convolutional neural networks, demonstrate sensitivity to the visual characteristics of training data. The phenomenon

known as the "simulation-to-reality gap" or "sim-to-real gap" manifests when models trained on synthetic data exhibit degraded performance when deployed on real-world systems [13, 14].

This framework establishes the technical foundation for addressing these challenges by enabling CARLA simulation data to be exported and replayed in environments with advanced rendering capabilities. The focus of this work is on developing robust data transformation pipelines, ensuring temporal accuracy, and maintaining compatibility with industry-standard formats. Future work can build upon this foundation to leverage Isaac Sim's advanced rendering features including ray-traced lighting, physically-based materials, and realistic environmental effects.

The value of this contribution lies primarily in:

- **Technical Infrastructure**: A complete pipeline for CARLA-to-USD conversion with temporal animation support
- **Data Transformation Accuracy**: Robust coordinate system conversion and timeline management
- **Extensibility**: A foundation that future work can enhance with improved rendering quality
- **Industry Standard Integration**: Compatibility with USD and Omniverse ecosystems

### 5.1.2 Universal Scene Description and NVIDIA Omniverse

Universal Scene Description (USD) is an open-source framework developed by Pixar Animation Studios that has emerged as the industry standard for describing, composing, simulating, and collaborating on 3D scenes [21]. USD provides a comprehensive ecosystem for managing complex 3D content with support for hierarchical scene composition, temporal animation data through keyframing, sophisticated material and lighting systems, and cross-platform compatibility across major 3D software applications.

NVIDIA Omniverse builds upon USD to create a collaborative platform for 3D workflows, with Isaac Sim serving as a robotics-focused simulation environment within the Omniverse ecosystem [22]. Isaac Sim provides advanced rendering capabilities through NVIDIA's RTX technology and sophisticated sensor simulation models.

The integration developed in this thesis establishes the technical pipeline for CARLA simulation data to be exported, transformed, and replayed within the Omniverse ecosystem. This creates opportunities for future enhancements including:

- **Advanced Rendering**: Future integration with Isaac Sim's ray-tracing capabilities for improved visual fidelity

- **Enhanced Materials**: Potential to leverage Omniverse's material libraries and PBR systems
- **Sensor Simulation**: Foundation for integrating Isaac Sim's advanced sensor models
- **Collaborative Workflows**: Access to Omniverse's multi-user collaboration features

This work focuses on establishing the core technical infrastructure—accurate data transformation, temporal animation, and format compatibility—upon which these future enhancements can be built.

### 5.1.3 Evolution of the Export Framework Approach

During the initial development phases of this thesis work, CARLA version 0.9.15 was the latest available release, lacking native USD export capabilities. This motivated the development of a custom USD export framework to enable CARLA-Isaac Sim integration. The initial framework design emphasized proof-of-concept functionality over production optimization, intentionally maintaining a simplified implementation suitable for demonstrating integration concepts.

However, during the final stages of thesis development, CARLA released version 0.9.16 with official USD export functionality [24]. Comprehensive testing of this official implementation revealed both its capabilities and its limitations.

**Official CARLA USD Export Capabilities:**

- High-quality static map exports with proper materials and geometric detail
- Individual vehicle models exported as standalone USD assets
- Proper material definitions with PBR parameters
- Optimized mesh representations with appropriate level-of-detail
- Integration with Omniverse asset libraries

**Critical Limitations of Official Export:**

- *No temporal animation support*: Exports only static scenes without timeline data
- *No simulation replay capability*: Cannot capture and replay complete simulation sequences

- *No multi-agent animation*: Does not support animated vehicle trajectories
- *No sensor configuration export*: Sensor placements and parameters not included
- *No synchronized data capture*: Cannot maintain temporal consistency with nuScenes format

These limitations necessitate the continued development of a complementary export framework. The approach evolved from a complete replacement system to a hybrid strategy that leverages official CARLA USD exports for static content while implementing custom functionality for temporal animation and simulation replay.

### 5.1.4 Hybrid Framework Architecture

The final framework design adopts a hybrid approach that combines the strengths of both official CARLA USD exports and custom temporal animation systems:

1. **Static Asset Utilization**: The framework can optionally use official CARLA USD map exports, benefiting from improved material quality and geometric detail
2. **Custom Animation Pipeline**: A complete temporal animation system captures vehicle trajectories, sensor configurations, and simulation state over time
3. **Dynamic Scene Composition**: The system composes complete animated USD stages by combining static assets with temporal animation data
4. **nuScenes Integration**: Maintains compatibility with nuScenes dataset format and sensor synchronization requirements

This hybrid approach provides the best of both worlds: leveraging official tools where available while filling critical gaps in temporal animation and simulation replay capabilities. The framework remains essential for generating time-synchronized synthetic datasets suitable for ADAS validation workflows.

## 5.2 Universal Scene Description Fundamentals

### 5.2.1 USD Technology Overview

Universal Scene Description provides a comprehensive framework for representing 3D scene data with particular strengths in handling temporal animations and hierarchical scene composition [20]. The technology consists of several key components that make it particularly suitable for autonomous driving simulation applications.

- **Scene Composition and Layering:** USD implements a sophisticated layering system that enables non-destructive composition of scene elements. Multiple USD files can be composed together, with later layers overriding or extending earlier layers without modifying the original source files. This capability proves valuable for managing complex simulation scenarios where vehicle trajectories, sensor configurations, and environmental conditions can be managed as separate compositional layers.

- **Temporal Animation Support:** USD includes native support for time-varying data through its attribute animation system. Any scene attribute can be keyframed over time, enabling precise control of object transformations, property changes, and state transitions. The framework supports various interpolation modes including linear, held, and custom interpolation functions, ensuring smooth and accurate motion representation.

- **Hierarchical Scene Organization:** USD scenes are organized as hierarchical stage graphs where scene elements (prims) can contain other prims in a tree structure. This hierarchy naturally represents the parent-child relationships common in vehicle simulation, such as sensors attached to vehicle bodies or wheels connected to chassis components. Transform hierarchies ensure that child elements move correctly with their parents.

- **Metadata and Custom Attributes:** USD provides extensive metadata capabilities, allowing arbitrary custom data to be associated with scene elements. This feature enables preservation of CARLA-specific parameters, sensor configurations, and simulation metadata alongside the geometric and animation data.

### 5.2.2 USD in Autonomous Driving Simulation

The autonomous driving industry has increasingly adopted USD as a standard format for simulation data exchange and collaboration. Several characteristics make USD particularly well-suited to ADAS development workflows:

- **Deterministic Replay:** USD's support for keyframed animation enables exact reproduction of complex multi-agent scenarios. A complete simulation can be captured as a temporal sequence and replayed with frame-accurate precision, ensuring reproducible validation results across different tools and platforms.

- **Multi-Tool Workflows:** USD's open standard and broad industry adoption enable seamless data exchange between simulation, validation, and visualization tools. A scenario captured in CARLA can be visualized in Isaac Sim,

analyzed in custom tools, and reviewed in standard 3D applications without format conversions.

- **Scalability and Performance:** USD's design emphasizes efficient handling of large-scale scenes with millions of elements. Lazy loading, instancing, and sophisticated caching mechanisms enable real-time interaction with complex urban environments containing numerous vehicles, pedestrians, and infrastructure elements.

- **Industry Standardization:** Major automotive OEMs and simulation providers have adopted USD as a standard format for scenario description and synthetic data exchange. This standardization facilitates collaboration between organizations and reduces toolchain integration complexity.

## 5.3 System Architecture and Pipeline Overview

### 5.3.1 High-Level Architecture

The CARLA to USD export framework implements a modular pipeline architecture that processes simulation data through multiple stages, transforming CARLA's runtime representation into deterministically replayable USD scenes. The system design emphasizes separation of concerns, with distinct components responsible for data capture, temporal management, coordinate transformation, and USD stage generation.

The system operates through a clearly defined sequence of processing stages, with data flowing from CARLA's simulation environment through multiple transformation and composition phases before final USD stage generation.

The architecture consists of three primary subsystems:

- **Data Acquisition Layer:** This layer interfaces directly with CARLA, extracting complete scene state information at each simulation timestep. The `DataCapture` component queries CARLA's world state, retrieving vehicle transforms, sensor configurations, environmental conditions, and map topology. The capture system operates synchronously with CARLA's simulation loop, ensuring temporal consistency across all captured data.

- **Processing and Transformation Layer:** Captured data undergoes multiple processing stages to prepare it for USD representation. The `TimelineManager` converts CARLA's microsecond-based timestamps to USD time codes, establishing frame-accurate temporal mapping. The `TransformUtils` component performs coordinate system conversions, transforming CARLA's left-handed coordinate system to USD's right-handed convention. The `ActorProcessor`

extracts and organizes vehicle-specific data including animation keyframes and mesh references.

- **USD Generation Layer:** The final layer assembles processed data into hierarchical USD stage structures. The `USDStageGenerator` orchestrates stage creation, coordinating specialized components for vehicle animation (`VehicleCreator`), environment setup (`EnvironmentCreator`), and sensor representation (`SensorCreator`). The `AssetManager` resolves vehicle mesh references and handles material assignment.

### 5.3.2 Component Interaction

The framework implements a coordinator pattern where the `USDSceneExporter` class serves as the central orchestrator, managing interactions between specialized components.

The `USDSceneExporter` maintains references to all major subsystems and coordinates their operations through a well-defined sequence. Listing B.1 presents the core structure.

Component interactions follow a clear dependency hierarchy. Low-level utilities (`TransformUtils`, `FileUtils`) provide foundational services to processing components (`ActorProcessor`, `RoadProcessor`). Processing components prepare data for consumption by USD generation components (`VehicleCreator`, `EnvironmentCreator`). The `USDStageGenerator` coordinates all generation components while the `USDSceneExporter` manages the entire pipeline.

### 5.3.3 Data Flow Through the Pipeline

Data progresses through the pipeline in three distinct phases: capture, processing, and generation.

- **Capture Phase:** During CARLA simulation execution, the framework captures scene state at regular intervals synchronized with the simulation timestep. The first captured frame includes comprehensive scene information including road network geometry (when using dynamic road export mode), while subsequent frames focus on dynamic elements such as vehicle positions and states. Each capture operation produces a structured data dictionary containing all information necessary for later USD reconstruction.

- **Processing Phase:** After simulation completion, captured data undergoes batch processing to prepare it for USD representation. The `TimelineManager` analyzes the complete temporal sequence, computing frame timing relationships and establishing the target playback framerate (24 FPS). The

`ActorProcessor` organizes vehicle data into per-vehicle timelines, applying coordinate transformations to all spatial data. Sensor configurations are extracted and formatted for USD representation.

- **Generation Phase:** The final phase assembles the USD stage structure. The `USDStageGenerator` creates the stage hierarchy, beginning with environment setup (either loading official CARLA USD maps or generating dynamic road geometry). The `VehicleCreator` instantiates vehicle prims with appropriate mesh references and applies temporal animation keyframes. The `SensorCreator` adds sensor visualization prims at correct positions. Material properties are assigned through the `MaterialManager`, and the complete stage is written to disk as a binary USD file.

## 5.4 Core Export System Implementation

### 5.4.1 USDSceneExporter Architecture

The `USDSceneExporter` class implements the main export orchestration logic, providing a high-level interface for CARLA integration while managing the complex pipeline of data capture, processing, and USD generation.

The class maintains several critical state variables throughout the export process:

- `carlaWorld`: Reference to the CARLA world instance, enabling queries of simulation state
- `export_map`: Boolean flag controlling environment export strategy (true = use official USD map, false = generate dynamic road network)
- `scene_data`: Accumulated list of captured frame data, representing complete simulation history
- `timeline_manager`: Handles temporal coordination and timestamp-to-timecode conversion
- `data_capture`: Interfaces with CARLA to extract scene state information

The exporter provides two primary methods for external interaction:

`capture_scene_state(timestamp, frame_number)`: Called during simulation execution to capture the current scene state. This method should be invoked after each CARLA world tick, typically within the main simulation loop. The method records timing information, extracts complete scene data, and accumulates it for final export.

`export_to_usd_stage(output_path)`: Called after simulation completion to generate the final USD file. This method instantiates the `USDStageGenerator`, processes all accumulated scene data, and writes the resulting USD stage to disk.

### 5.4.2 Data Capture System

The `DataCapture` component implements comprehensive scene state extraction, interfacing directly with CARLA's Python API to query world state. The capture system must handle various actor types, sensor configurations, and environmental conditions while maintaining data consistency across frames. Listing B.2 presents the implementation.

The capture system implements several optimization strategies to minimize performance impact on simulation execution. Vehicle data capture focuses on transform and velocity information, avoiding expensive queries for properties that remain static across frames. Road network extraction occurs only once (first frame) when using dynamic export mode, as road geometry does not change during simulation. Sensor configurations are captured comprehensively but only for sensors attached to the ego vehicle, following nuScenes dataset conventions.

### 5.4.3 Timeline Management and Temporal Coordination

Accurate temporal representation presents a significant challenge due to fundamental differences between CARLA's simulation time model and USD's animation system. CARLA operates with microsecond-precision timestamps that may not correspond to uniform frame intervals, while USD animations typically target a specific playback framerate (24 FPS in this implementation).

The `TimelineManager` addresses this challenge through a systematic timestamp-to-timecode conversion strategy. The manager tracks the simulation's start timestamp and converts subsequent CARLA timestamps (in microseconds) to USD time codes by calculating elapsed time and multiplying by a target framerate of 24 FPS. This approach ensures consistent playback timing in Omniverse regardless of the original CARLA simulation framerate, enabling deterministic replay of recorded scenarios.

The timeline system maintains all recorded timestamps to enable accurate conversion during USD generation. The `timestamp_to_time_code` method implements the core conversion algorithm, calculating elapsed time from simulation start and scaling it to achieve the target playback framerate. This approach ensures that relative timing relationships between simulation events are preserved exactly, even though absolute frame numbers may differ from the original CARLA capture rate.

The hardcoded 24 FPS target framerate represents a deliberate design choice balancing animation smoothness with file size considerations. Higher framerates

would provide smoother playback but significantly increase USD file sizes due to additional keyframe data. The 24 FPS rate provides acceptable motion quality for validation workflows while maintaining manageable file sizes.

## 5.5 Coordinate System Transformation

### 5.5.1 CARLA to USD Coordinate System Mapping

One of the most critical and technically challenging aspects of the USD export framework involves accurate coordinate system transformation. CARLA inherits Unreal Engine 4's left-handed coordinate system, while USD and the broader Omniverse ecosystem employ a right-handed coordinate system following industry standards for 3D graphics and robotics applications. This fundamental difference requires careful transformation of all spatial data to maintain correct spatial relationships and orientations.

**Coordinate System Characteristics:**

- CARLA (Left-Handed System):
  - X-axis: Forward direction (vehicle heading)
  - Y-axis: Right direction (perpendicular to heading)
  - Z-axis: Up direction (vertical)
  - Rotation: Counter-clockwise positive when viewed from positive axis
- USD/Omniverse (Right-Handed System):
  - X-axis: Right direction (or sometimes forward, depending on convention)
  - Y-axis: Up direction (vertical)
  - Z-axis: Forward direction (or backward, depending on convention)
  - Rotation: Counter-clockwise positive when viewed from positive axis (following right-hand rule)

### 5.5.2 Transform Conversion Algorithms

The `TransformUtils` class implements comprehensive transformation algorithms ensuring precise spatial relationships are maintained during coordinate system conversion. The transformation involves three distinct operations: position vector conversion, rotation quaternion transformation, and velocity vector remapping. Listing B.3 presents the implementation.

The transformation algorithm implements several critical operations:

- **Position Vector Transformation:** The Y-coordinate is inverted while X and Z remain unchanged. This inversion accounts for the difference in handedness between the two systems. In CARLA, positive Y points to the vehicle's right; in USD, positive Y points upward (or in some conventions, to the left).

- **Rotation Conversion:** Rotations require more complex handling due to the interaction between coordinate system handedness and rotation conventions. The algorithm first converts CARLA's Euler angles (in degrees) to radians, then negates the yaw angle to account for handedness change. The Euler angles are then converted to quaternion representation, which USD uses for rotations due to its advantages in interpolation and gimbal lock avoidance.

- **Velocity Vector Transformation:** Velocity vectors undergo the same axis remapping as position vectors, ensuring that motion vectors remain correctly oriented in the transformed coordinate system.

### 5.5.3 Validation of Spatial Accuracy

The coordinate transformation system includes comprehensive validation mechanisms to ensure accuracy and consistency of spatial data throughout the conversion process.

- **Round-Trip Testing:** Transformation accuracy is verified through round-trip conversion tests where CARLA transforms are converted to USD format and then back to CARLA coordinates. The test validates that the round-trip conversion produces results within floating-point precision tolerances (typically $< 10^{-6}$) of the original values.

- **Reference Point Validation:** Known reference points with well-defined coordinates in both systems are used for validation. For example, the origin point $(0, 0, 0)$ should remain unchanged except for coordinate axis remapping, and unit vectors along each axis should transform according to expected mathematical relationships.

- **Spatial Relationship Preservation:** Relative distances and angles between objects must remain constant through transformation. The validation system computes distances between pairs of vehicles in both CARLA and USD representations, verifying that relative spatial relationships are preserved exactly.

- **Orientation Consistency:** Vehicle heading directions are validated by comparing forward vectors in both coordinate systems. A vehicle facing north (positive X direction) in CARLA should maintain equivalent heading in USD after transformation.

## 5.6 Vehicle Animation System

### 5.6.1 Keyframe Generation Strategy

The vehicle animation system generates smooth, temporally accurate motion representations by converting captured vehicle trajectories into USD keyframed animations. The system must handle varying capture rates, coordinate transformations, and USD's specific animation representation requirements.

The keyframe generation process follows a systematic pipeline that transforms raw capture data into properly formatted USD animation curves.

The `VehicleCreator` component implements the animation generation logic. Listing B.4 presents the implementation.

The animation system creates keyframes at each captured simulation frame, with USD's interpolation system handling smooth motion between keyframes. This approach ensures that vehicle trajectories are reproduced accurately while minimizing animation data size.

### 5.6.2 Transform and Velocity Processing

Vehicle state data extracted from CARLA includes not only position and orientation but also velocity information critical for physics-accurate replay and analysis. The `ActorProcessor` extracts complete vehicle state from CARLA actors through the coordinate transformation pipeline. Each vehicle's transform (position, rotation) and velocity undergo coordinate system conversion via `TransformUtils` (see Listing B.3), with additional metadata including vehicle ID, blueprint type, and mesh information for asset resolution. This comprehensive state capture enables both visual replay and physics-based analysis within USD environments.

Velocity information, while not directly used in USD animation keyframes, is preserved in the exported data for potential use in physics-based replay systems or for analysis purposes.

## 5.7 Asset Management and Visual Representation

### 5.7.1 Hybrid Asset Strategy

The framework implements a hybrid asset management strategy that leverages both official CARLA USD exports and custom fallback mechanisms to ensure robust visual representation across all scenarios.

With the release of CARLA 0.9.16's official USD export capability, the framework can now utilize high-quality vehicle and map assets when available. However, to maintain compatibility with earlier CARLA versions and handle cases where official USD assets are unavailable, the system retains a comprehensive fallback system.

### 5.7.2 Vehicle USD Asset Library

The system maintains a library of USD vehicle asset references corresponding to CARLA's vehicle blueprints. When creating vehicle representations, the framework attempts to reference official CARLA USD assets first, falling back to custom assets or procedural geometry if necessary. Listing B.5 presents the implementation.

This three-tier asset resolution strategy prioritizes official CARLA USD assets to ensure the highest visual quality when available, while fallback mechanisms guarantee that the framework remains functional across all scenarios.

### 5.7.3 Material Properties and Visual Fidelity

The material system implements different strategies depending on the asset source:

- **Official CARLA USD Assets:** When using official CARLA exports, materials are fully defined with PBR (Physically Based Rendering) parameters including base color, metallic properties, roughness values, and normal maps. These materials are compatible with Isaac Sim's RTX-enabled rendering pipeline, providing a foundation for future visual quality enhancements.

- **Custom Assets:** Custom vehicle USD assets use simplified material definitions with basic PBR parameters. While less detailed than official assets, these materials still provide reasonable visual quality suitable for validation workflows.

- **Fallback Geometry:** Procedurally generated fallback geometry uses solid color materials assigned deterministically based on vehicle IDs, ensuring visual consistency and distinctiveness across the simulation.

The deterministic color generation ensures that vehicles maintain consistent appearance across multiple exports of the same simulation, facilitating visual comparison and analysis.

### 5.7.4 Asset Coverage and Statistics

The framework includes asset coverage analysis functionality that tracks the availability of USD representations for encountered vehicle types. The system maintains statistics on vehicle types encountered during export, categorizing them into those with available USD assets versus those requiring fallback geometry. Upon export completion, the framework reports coverage percentages and lists vehicle types that utilized fallback representations, providing visibility into asset library completeness and identifying opportunities for asset acquisition or creation.

These statistics provide valuable feedback for identifying gaps in asset coverage and prioritizing asset acquisition or creation efforts.

## 5.8 Sensor Integration in USD Format

### 5.8.1 Sensor Representation Strategy

Sensors in the USD export serve dual purposes: they provide visual representation for scene understanding and they preserve sensor configuration metadata for potential use in Isaac Sim's sensor simulation system. The export framework implements sensor representations that balance visual clarity with metadata completeness.

### 5.8.2 Camera Sensor Export

Camera sensors are exported as USD Camera prims with comprehensive parameter preservation. Listing B.6 presents the implementation.

The camera export preserves critical parameters including field of view, resolution, and focal length calculations that enable accurate sensor simulation in Isaac Sim.

### 5.8.3 LiDAR and Radar Sensor Integration

LiDAR sensors receive special handling to leverage Omniverse's sensor asset library when possible. Listing B.7 presents the implementation.

The LiDAR integration attempts to reference realistic sensor models from Omniverse's asset library, providing visual accuracy while preserving the technical parameters needed for potential sensor simulation in Isaac Sim.

### 5.8.4 Sensor Parameter Preservation and Metadata

All sensor types include comprehensive metadata preservation to enable future integration with Isaac Sim's sensor simulation capabilities:

| Sensor Type | USD Representation | Preserved Parameters |
|---|---|---|
| RGB Camera | UsdGeom.Camera | FOV, resolution, focal length, aperture, clipping range |
| Semantic Camera | UsdGeom.Camera + metadata | Same as RGB + segmentation class mapping |
| Depth Camera | UsdGeom.Camera + metadata | Same as RGB + depth range parameters |
| LiDAR | Asset reference or cylinder | Channels, range, rotation frequency, points per second |
| Radar | Cylinder geometry | Range, horizontal/vertical FOV, points per second |

**Table 5.1:** Sensor type mapping and parameter preservation

This comprehensive parameter preservation ensures that the USD export maintains all information necessary for potential high-fidelity sensor simulation in Isaac Sim, even though the current implementation focuses primarily on visual representation and animation replay.

## 5.9 Environment Export and Map Integration

### 5.9.1 Hybrid Approach to Environment Representation

The environment export system implements a flexible hybrid approach that can leverage official CARLA USD map exports (when using CARLA 0.9.16+) or generate dynamic road network representations (for compatibility with earlier versions or when official maps are unavailable).

This hybrid strategy is controlled through the `export_map` parameter:

- `export_map=True`: Load and reference official CARLA USD map assets
- `export_map=False`: Generate dynamic road network from CARLA waypoint data

### 5.9.2 Official CARLA USD Map Integration

When using official CARLA USD map exports, the environment setup process becomes streamlined. Listing B.8 presents the implementation.

Official USD maps provide several significant advantages:

- **Visual Fidelity**: Complete building geometry, detailed road textures, proper materials with PBR parameters
- **Lighting Accuracy**: Pre-configured lighting setups appropriate for the environment
- **Asset Consistency**: Maps exported directly from CARLA maintain perfect geometric alignment with simulation
- **Professional Quality**: Maps benefit from CARLA's professional 3D asset pipeline

However, these advantages come with trade-offs:

- **File Size**: Complete maps can be several gigabytes
- **Version Dependency**: Requires CARLA 0.9.16 or later
- **Static Content Only**: Official exports do not include animated elements

### 5.9.3 Dynamic Road Network Generation

For scenarios where official USD maps are unavailable or when a lightweight representation is preferred, the framework can generate road network geometry from CARLA's waypoint topology. The `RoadProcessor` samples waypoints across the map at regular intervals (typically 2 meters) and groups them by road and lane identifiers. For each lane, the processor generates a triangle mesh strip by calculating road edge positions perpendicular to waypoint directions, using an approximate lane width of 4 meters. Consecutive waypoint pairs are connected through quad faces (two triangles), creating a continuous road surface mesh. The resulting geometry undergoes coordinate transformation to USD conventions and is exported with basic materials, providing a functional spatial reference for vehicle motion even when high-fidelity environmental assets are unavailable.

Dynamic road generation provides complementary advantages:

- **Lightweight**: Significantly smaller file sizes (megabytes vs gigabytes)
- **Version Independence**: Works with any CARLA version
- **Customizability**: Road appearance can be adjusted programmatically

- **Runtime Generation**: Can adapt to custom or procedurally generated maps

The trade-offs include:

- **Visual Simplicity**: Only road surfaces, no buildings or detailed environment
- **Basic Materials**: Simple texture mapping without advanced PBR
- **Generation Overhead**: Requires processing time during export

### 5.9.4 Hybrid Strategy Selection Guidelines

The choice between official USD maps and dynamic road generation depends on specific use case requirements:

- **Use Official USD Maps When:**
  - Visual realism is critical for perception algorithm training
  - Large file sizes are acceptable for the workflow
  - Using CARLA 0.9.16 or later
  - Creating datasets for publication or external sharing
  - Need consistency with CARLA's official visualization
- **Use Dynamic Road Generation When:**
  - File size constraints are important
  - Working with CARLA versions before 0.9.16
  - Focus is on vehicle trajectory analysis rather than perception
  - Need maximum compatibility across CARLA versions
  - Implementing custom or procedurally generated maps
- **Hybrid Approach (Both):** The framework also supports a hybrid approach where official USD maps provide environmental context while custom elements (such as dynamically placed infrastructure sensors from Chapter 4) are added programmatically. This combines visual fidelity with customization flexibility.

# Chapter 6

# Isaac Sim Extension Development

## 6.1 Introduction: Bridging USD Export and Data Collection

The USD export framework presented in Chapter 5 established the technical foundation for converting CARLA simulations into deterministic, replayable scenes within NVIDIA Isaac Sim. This conversion creates temporal USD animations that preserve all spatial relationships, vehicle trajectories, and sensor configurations from the original CARLA simulation. However, the USD export alone represents only half of the integration pipeline—a mechanism is required to replay these imported simulations, manage sensor data collection, and generate industry-standard datasets compatible with existing ADAS validation workflows.

This chapter presents the complementary software component that completes the CARLA-to-Isaac Sim integration: the `omni.carla.sim_controller` extension, a custom Isaac Sim extension designed to bridge the gap between USD-exported simulations and practical synthetic data generation requirements. The extension provides comprehensive sensor management capabilities, intuitive simulation control, and automated nuScenes dataset generation—all while maintaining full backward compatibility with both Federico Stella's vehicle-centric approach and this thesis's infrastructure sensing extensions.

The extension serves multiple critical functions within the overall pipeline:

- **Sensor Discovery and Management**: Automatic detection and configuration of both vehicle-mounted and infrastructure sensors within imported USD scenes

- **Simulation Replay Control**: Precise temporal control over imported CARLA animations with configurable playback parameters
- **Multi-Modal Data Capture**: Synchronized collection of camera and LiDAR data across all detected sensors
- **nuScenes Dataset Generation**: Automated export of collected data in nuScenes format with complete metadata structure
- **Cooperative Sensing Support**: Native integration of infrastructure sensor data alongside vehicle sensor data within unified datasets

A key innovation of this extension lies in its seamless support for infrastructure-cooperative sensing scenarios. While Stella's original framework focused exclusively on vehicle-mounted sensors, this extension natively handles both vehicle-attached and world-fixed sensors within the same data collection workflow. This capability enables validation of cooperative perception algorithms, V2I communication systems, and multi-perspective sensing scenarios that would be impossible with vehicle-centric approaches alone.

The extension architecture prioritizes modularity, maintainability, and extensibility. Each major functional component—sensor management, timeline control, user interface, and nuScenes dataset generation—operates as an independent module with well-defined interfaces. This design facilitates future enhancements, debugging, and integration with additional Omniverse capabilities as the ecosystem evolves.

## 6.2 Complete Workflow: From CARLA to nuScenes Dataset

Before detailing the extension's technical architecture, it is essential to understand how this component integrates within the complete data generation pipeline. The workflow spans multiple software environments and processes, with the Isaac Sim extension serving as the final orchestrator of data collection and dataset generation.

### 6.2.1 End-to-End Pipeline Overview

The complete workflow from initial scenario design to final nuScenes dataset involves seven distinct phases:

- **Phase 1: Scenario Configuration in CARLA.** Users configure the simulation scenario within CARLA, defining map selection, weather conditions, traffic density, and ego vehicle behavior (if present). This phase utilizes the standard CARLA Python API and configuration tools established in previous work [7].

- **Phase 2: Sensor Configuration (Flexible Deployment).** The framework supports two distinct sensor deployment modes, enabling flexible validation scenarios:

  - **Vehicle-Only Mode**: Sensors are configured exclusively on the ego vehicle using the sensor placement interface described in previous work [7]. This mode replicates the original framework capabilities with cameras for 360-degree coverage, roof-mounted LiDAR sensors, and additional perception sensors mounted on the vehicle.
  - **Infrastructure-Only Mode**: Sensors are deployed exclusively on fixed infrastructure positions using the sensor placement management tool developed in Chapter 4. This mode enables traffic monitoring scenarios, intersection surveillance validation, and infrastructure-centric perception testing without requiring an ego vehicle. The tool provides real-time visualization of sensor perspectives, automatic intersection detection, and configuration management for each infrastructure sensor.

  The choice of deployment mode depends on the specific validation objectives. Vehicle-only mode maintains backward compatibility with the original framework, while infrastructure-only mode enables new monitoring and surveillance scenarios. While the framework's architecture supports the theoretical combination of both modes simultaneously (cooperative sensing), the current implementation focuses on these two distinct operational modes.

- **Phase 3: Simulation Execution and USD Export.** The configured CARLA simulation executes in synchronous mode with fixed time steps, ensuring deterministic behavior. The USD export framework (Chapter 5) captures the complete simulation state at each time step, recording vehicle trajectories (if ego vehicle present), sensor configurations (both vehicle and/or infrastructure), actor positions, and environmental parameters. Upon simulation completion, the framework generates a single USD file containing the entire temporal sequence as keyframed animations.

- **Phase 4: USD Import into Isaac Sim.** The generated USD file is imported into NVIDIA Isaac Sim, where it appears as a fully replayable animation within the Omniverse environment. The import process preserves all spatial relationships, temporal synchronization, and sensor metadata from the original CARLA simulation. Users can inspect the imported scene using Isaac Sim's standard viewport tools before proceeding to data collection.

- **Phase 5: Extension-Based Data Collection.** The `sim_controller` extension is activated within Isaac Sim, automatically scanning the imported

USD scene to detect all available sensors regardless of their mounting type (vehicle-attached or infrastructure-fixed). Users select which sensors to activate for data collection and configure sampling parameters (frame rate, output paths, nuScenes metadata). The extension then replays the USD animation while capturing synchronized sensor data from all selected sensors.

- **Phase 6: nuScenes Dataset Generation.** Upon completion of the simulation replay and data collection, the extension automatically generates a complete nuScenes-compatible dataset. This includes all required JSON metadata files defining scene structure, sensor calibrations, sample relationships, and temporal organization. The dataset structure adapts to the deployment mode used:
  - Vehicle-only datasets follow the standard nuScenes structure with ego-vehicle-centric annotations
  - Infrastructure-only datasets use world-fixed coordinate frames with infrastructure sensor metadata

  The resulting dataset is immediately compatible with existing nuScenes devkit tools, perception algorithms, and validation pipelines without requiring any format conversion or post-processing.

### 6.2.2 Key Advantages of the Integrated Approach

This integrated workflow provides several significant advantages over traditional approaches:

- **Deterministic Reproducibility**: The USD-based approach enables perfect reproduction of simulation scenarios. Once a USD animation is generated, it can be replayed indefinitely with identical results, facilitating controlled experiments and comparative analysis.
- **Separation of Concerns**: The separation between simulation execution (CARLA) and data collection (Isaac Sim) enables independent optimization of each component. CARLA focuses on physics accuracy and scenario generation, while Isaac Sim concentrates on rendering quality and sensor simulation.
- **Enhanced Visual Fidelity Foundation**: While the current implementation utilizes basic USD materials, the framework establishes the foundation for future integration with Isaac Sim's advanced rendering capabilities, including ray-traced lighting and physically-based materials.

- **Flexible Data Collection**: Users can replay the same simulation multiple times with different sensor configurations, sampling rates, or processing parameters without re-executing the computationally expensive CARLA simulation.
- **Multiple Deployment Modes**: The framework's support for vehicle-only and infrastructure-only sensing modes enables validation of diverse scenarios:
  - Traditional ego-vehicle perception testing (vehicle-only mode)
  - Traffic monitoring and surveillance validation (infrastructure-only mode)

  While the architecture is designed to potentially support simultaneous vehicle and infrastructure sensors in future work, the current implementation focuses on these two distinct operational modes.
- **Backward Compatibility**: Vehicle-only mode maintains full compatibility with the original framework [7] and existing validation workflows, while infrastructure-only mode provides new capabilities without disrupting established processes.

## 6.3 Extension Architecture and Component Design

### 6.3.1 Modular Component Architecture

The Isaac Sim extension implements a modular architecture pattern that cleanly separates functional concerns across specialized components. This separation ensures maintainability, facilitates debugging, and enables independent enhancement of individual subsystems. The architecture consists of four primary components that work in coordination through well-defined interfaces and event-driven communication patterns:

- The **Sensor Manager** (`sensor_manager.py`) implements comprehensive sensor discovery, selection, and data collection capabilities. This component automatically scans imported USD scenes to identify available sensors, manages user-specified sensor selection, and orchestrates synchronized data capture from multiple sensor modalities. The Sensor Manager handles both camera and LiDAR sensors while maintaining separate processing pipelines for each modality to optimize performance and ensure proper data formatting.
- The **Timeline Controller** (`timeline_controller.py`) provides precise control over simulation playback and temporal synchronization. This component

interfaces with Isaac Sim's timeline system to manage play/pause/stop operations, monitors current frame position, and triggers data collection events at appropriate intervals. The Timeline Controller ensures that sensor data capture remains synchronized with animation playback while preventing frame duplication or data loss.

- The **UI Controller** (`ui_controller.py`) implements the user interface layer, providing intuitive graphical controls for all extension operations. The interface design prioritizes accessibility for users with varying technical expertise while providing advanced configuration options for power users. The UI Controller manages sensor selection checkboxes, playback control buttons, configuration parameters, and real-time status displays.

- The **nuScenes Manager** (`nuscenes_manager.py`) handles the complex task of generating nuScenes-compatible datasets from collected sensor data. This component maintains the complete nuScenes database schema, manages token generation for all dataset elements, organizes collected data into proper directory structures, and exports all metadata in JSON format. The nuScenes Manager also implements the infrastructure sensor extensions developed in Chapter 4, ensuring that both vehicle and infrastructure sensor data integrate seamlessly within unified datasets.

Listing C.1 presents the extension initialization and component coordination.

### 6.3.2 Event-Driven Communication Architecture

The extension implements an event-driven communication pattern that enables loose coupling between components while maintaining responsive user interaction and reliable data collection. This architecture ensures that intensive operations do not block the user interface while guaranteeing proper synchronization between simulation playback and data capture.

Timeline events serve as the primary coordination mechanism. When Isaac Sim's timeline advances during playback, it generates events that the Timeline Controller captures and processes. For each event, the controller determines whether data collection should occur based on the configured sampling rate. If collection is required, the controller invokes the Sensor Manager to capture data from all active sensors and updates the nuScenes Manager with new sample information.

The callback-based design ensures that data collection operations execute asynchronously relative to the main UI thread. This prevents the interface from freezing during intensive sensor data processing, particularly when handling multiple high-resolution cameras or large LiDAR point clouds simultaneously. Status updates

propagate back to the UI Controller through return values and state queries, enabling real-time progress visualization.

Component interactions are mediated through well-defined callback mechanisms rather than direct method invocation. This approach facilitates testing, debugging, and future enhancement by minimizing dependencies between components. Each component exposes a clear API surface while encapsulating internal implementation details.

## 6.4 Infrastructure-Cooperative Sensor Integration

### 6.4.1 Extended nuScenes Format for Cooperative Sensing

The most significant innovation of the Isaac Sim extension lies in its native support for infrastructure-cooperative sensing scenarios. While Federico Stella's framework established comprehensive capabilities for vehicle-mounted sensor simulation, it was inherently designed around a mobile ego vehicle perspective. This chapter's extension fundamentally evolves that approach to seamlessly integrate both vehicle-attached and world-fixed sensors within unified datasets.

The integration required careful extension of the nuScenes data structures to accommodate infrastructure sensors while maintaining complete backward compatibility with existing tools and workflows. The solution implements a minimally invasive approach that adds infrastructure sensor support through optional metadata fields rather than structural changes to core database tables.

#### Enhanced DataLog Structure

The `DataLog` class, which serves as the primary data accumulation structure during simulation replay, was extended to distinguish between vehicle-mounted and infrastructure-based LiDAR sensors. Listing C.2 presents the enhanced structure.

The key innovation is the `_LID_INTER_val` field, which maintains a separate collection for infrastructure LiDAR sensor data. This separation is necessary because infrastructure sensors require different coordinate transformation procedures compared to vehicle sensors. Vehicle-mounted sensors use ego-vehicle-relative coordinates that change with vehicle motion, while infrastructure sensors use world-fixed coordinates that remain constant throughout the simulation.

The separation also facilitates proper sensor calibration metadata generation. Each sensor type requires different calibration parameters in the nuScenes format—vehicle sensors include ego-to-sensor transformations that vary over time, while infrastructure sensors maintain constant world-to-sensor transformations.

### 6.4.2 Coordinate System Handling for Infrastructure Sensors

Infrastructure sensor integration requires careful handling of coordinate system transformations to maintain compatibility with the nuScenes format while properly representing world-fixed sensor perspectives. The implementation addresses three distinct coordinate frames:

1. **Isaac Sim World Coordinates**: The native coordinate system of the Omniverse environment, which may use different conventions than CARLA

2. **nuScenes World Coordinates**: The standardized right-handed, Z-up coordinate system required by the nuScenes format

3. **Sensor-Local Coordinates**: The local coordinate frame of each individual sensor, used for point cloud and image data representation

The extension automatically detects whether each sensor is vehicle-mounted or infrastructure-based by analyzing its USD hierarchy path. Sensors attached to the ego vehicle follow a specific path pattern (e.g., `/World/Car/SensorName`), while infrastructure sensors are positioned directly in the world hierarchy. This automatic detection enables proper selection of coordinate transformation procedures without requiring manual configuration. Listing C.3 presents the implementation.

### 6.4.3 Infrastructure Sensor Annotations in nuScenes Format

Infrastructure sensor data must be properly annotated within the nuScenes database structure to enable correct interpretation by analysis tools. The extension implements specialized annotation procedures that maintain temporal continuity while clearly identifying infrastructure sensor sources.

The `sensor` table entries for infrastructure sensors include the `intersection_id` metadata field introduced in Chapter 4, associating each sensor with its monitored intersection. This association enables spatial queries and multi-sensor fusion based on geographic context. Listing C.4 presents the implementation.

The `is_fixed` parameter serves as the primary discriminator enabling downstream tools to apply appropriate processing procedures. Analysis pipelines can query this field to determine whether sensor data should be interpreted using ego-relative or world-fixed coordinate frames.

### 6.4.4 Backward Compatibility and Migration Path

Critical to the infrastructure sensor integration is complete backward compatibility with existing nuScenes tools and datasets. The extension achieves this through several design decisions:

- **Optional Metadata Fields**: All infrastructure-specific fields (`is_fixed`, `intersection_id`, `sensor_type`) are implemented as optional metadata rather than required schema modifications. Datasets containing only vehicle sensors omit these fields entirely, ensuring compatibility with original nuScenes tools.
- **Standard Table Structure**: No modifications were made to the core nuScenes table structures (`sample`, `sample_data`, `ego_pose`, etc.). Infrastructure sensors populate these tables using the same procedures as vehicle sensors, with infrastructure-specific handling occurring only in coordinate transformation and calibration generation.
- **Sensor Channel Naming**: Infrastructure sensors use distinct channel naming conventions (e.g., `LIDAR_TRAFFIC_`, `CAM_TRAFFIC_`) that clearly identify their type while remaining compatible with nuScenes string-based sensor identification.
- **Separate Processing Paths**: The extension implements separate processing paths for vehicle and infrastructure sensors internally, but these paths converge to generate identical output structures in the final dataset. This approach ensures that infrastructure sensor data can be analyzed using the same tools as vehicle sensor data.

This backward compatibility enables gradual adoption of infrastructure sensing capabilities within existing ADAS validation pipelines without requiring wholesale replacement of established tools and workflows.

## 6.5 Sensor Management System

### 6.5.1 Automatic Sensor Discovery

The Sensor Manager implements sophisticated automatic sensor discovery capabilities that can identify and categorize available sensors within imported USD scenes. The discovery system must handle diverse sensor representations resulting from different export procedures, varying sensor types, and both vehicle-mounted and infrastructure-based deployments.

The discovery process operates through multiple detection methods applied in sequence, with each method targeting specific sensor representation patterns. This multi-method approach ensures robust detection across different USD export configurations and sensor types.

- **Method 1: Camera Primitive Detection.** The first detection method searches for USD Camera primitives (`UsdGeom.Camera`) within the scene graph. Camera primitives represent the standard USD representation for camera sensors and contain intrinsic parameters such as focal length, sensor size, and projection type. The detection traverses the entire scene hierarchy, identifying all camera primitives regardless of their attachment point (vehicle or world).

- **Method 2: LiDAR Asset Reference Detection.** LiDAR sensors exported by the USD framework may appear as asset references (links to external USD files containing sensor geometry). The discovery system searches for primitive names matching LiDAR patterns (`LIDAR`, `LID`, `RTX`, `Ouster`) and attempts to resolve their asset references. This method handles both CARLA's native LiDAR sensors and Isaac Sim's RTX LiDAR implementations.

- **Method 3: Geometric Proxy Detection.** For LiDAR sensors that were exported as simple geometric proxies (cylinders or spheres representing sensor positions), the discovery system searches for appropriately named mesh primitives. This fallback method ensures detection of sensors that lack sophisticated USD representations but maintain correct spatial positioning.

- **Method 4: Metadata-Based Detection.** Some sensor representations include custom metadata attributes that explicitly identify sensor type and parameters. The discovery system queries these metadata fields to supplement information from geometric and primitive-based detection.

Listing C.5 presents the multi-method sensor discovery implementation.

### 6.5.2 Sensor Selection and Configuration Management

Following sensor discovery, users interact with the UI Controller to select which sensors should be activated for data collection. The Sensor Manager maintains the selection state and ensures that only activated sensors participate in data capture operations.

The selection system provides individual control over each detected sensor, enabling users to create custom sensor configurations for specific validation scenarios. For example, users might select only front-facing cameras for forward-collision

testing, or activate all infrastructure sensors to evaluate cooperative perception coverage.

Configuration parameters managed by the Sensor Manager include:

- **Camera Resolution**: Configurable image resolution for camera sensors, with default 500×350 pixels matching nuScenes specifications
- **Sampling Rate**: Frame interval for data collection (e.g., every 5th frame), enabling control over dataset density
- **Output Paths**: Directory locations for sensor data storage, organized by sensor type and name
- **Data Formats**: Format selection for each sensor type (PNG for cameras, NPY/CSV for LiDAR)

### 6.5.3 Multi-Modal Data Collection

The Sensor Manager orchestrates synchronized data collection from all selected sensors during simulation replay. Data collection occurs through timeline event callbacks, ensuring proper temporal alignment across all sensors.

#### Camera Data Collection

Camera data collection implements a robust pipeline that handles various image formats and color spaces while ensuring consistent output quality. Listing C.6 presents the implementation.

#### LiDAR Data Collection

LiDAR data collection presents additional complexity due to the variety of LiDAR sensor implementations within Isaac Sim and the need to handle both vehicle-mounted and infrastructure-based sensors. Listing C.7 presents the implementation.

## 6.6 Timeline Control and Synchronization

### 6.6.1 Isaac Sim Timeline Integration

The Timeline Controller implements precise integration with Isaac Sim's animation timeline system, ensuring synchronized data collection during simulation replay. Isaac Sim represents USD animations through a timeline interface that controls playback of all animated properties within the scene.

The Timeline Controller registers callbacks with Isaac Sim's timeline event system, receiving notifications for each frame advance during playback. These callbacks provide the frame number and playback state (playing/paused/stopped), enabling the extension to coordinate data collection activities. Listing C.8 presents the implementation.

### 6.6.2 Synchronized Data Collection

The synchronization mechanism ensures that all sensor data collected at a given frame corresponds to the exact same simulation state. This temporal consistency is critical for multi-sensor fusion algorithms and perception system validation.

The data collection callback implements a sampling rate mechanism that controls collection frequency. Listing C.9 presents the implementation.

## 6.7 nuScenes Dataset Generation and Export

### 6.7.1 Complete nuScenes Database Structure

Upon completion of simulation replay and data collection, the nuScenes Manager generates the complete database structure required by the nuScenes format. This process transforms the accumulated sensor data and metadata into the thirteen interconnected JSON tables that comprise a valid nuScenes dataset.

The generation process occurs in a specific order to ensure referential integrity between tables. Listing C.10 presents the orchestrated dataset generation.

### 6.7.2 Infrastructure Sensor Integration in Sample Structure

A critical aspect of the nuScenes generation is proper integration of infrastructure sensor data within the sample structure. Each `sample` represents a temporal keyframe containing references to data from all available sensors at that moment.

The implementation ensures that infrastructure sensor data appears alongside vehicle sensor data within each sample, with proper coordination through the `sample_data` table. Listing C.11 presents the unified sample structure implementation.

The resulting dataset structure seamlessly integrates infrastructure sensor data within the standard nuScenes format, enabling analysis tools to process vehicle and infrastructure sensors uniformly while allowing specialized handling when needed.

## 6.8 User Interface Design and Interaction

### 6.8.1 Intuitive Sensor Management Interface

The UI Controller provides a user-friendly graphical interface that makes the extension accessible to users with varying levels of technical expertise. The interface design prioritizes clarity, organization, and responsive feedback while providing access to advanced configuration options.

The interface (Figure 6.1) is organized into collapsible sections that group related functionality, providing a clean and scalable layout that accommodates varying numbers of detected sensors. The implementation demonstrates the detection of both vehicle-mounted sensors (OmniverseKit series) and infrastructure sensors (CAM_TRAFFIC, LIDAR_TOP), showcasing the extension's capability to handle heterogeneous sensor configurations. Listing C.12 presents the implementation.

### 6.8.2 Real-Time Status Feedback

The interface provides continuous feedback about extension state, data collection progress, and any errors or warnings that occur. During active recording, the status display shows the current frame number and count of active sensors (e.g., "Recording: Frame 150 | 2 camera(s) | 1 LiDAR(s)"). When idle, the display indicates readiness and reports the total number of detected sensors. This real-time feedback enables users to monitor data collection without interrupting the simulation workflow.

The interface design makes the extension immediately usable while providing sufficient control for advanced users who need fine-grained configuration.

## 6.9 Performance Optimization and Error Handling

### 6.9.1 Efficient Resource Management

The extension implements several performance optimization strategies to ensure responsive operation during intensive data collection operations:

- **Frame Tracking and Deduplication.** A set-based tracking mechanism prevents redundant processing of frames that have already been captured. Before processing each frame, the extension checks if the frame number exists in a tracking set; if present, processing is skipped. This approach is particularly important during pause/resume operations or when manually

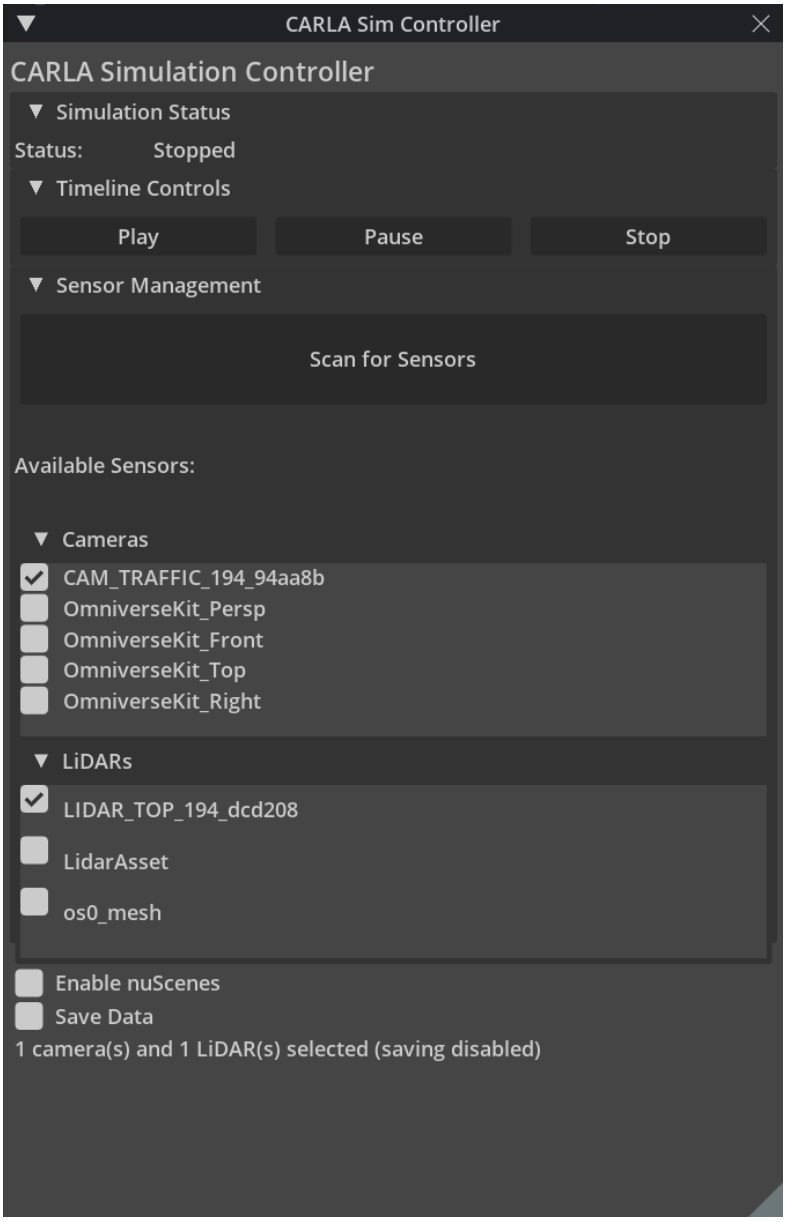


**Figure 6.1:** Isaac Sim Extension User Interface

stepping through the timeline, preventing duplicate data collection that would waste computational resources and storage space.

- **Memory Management and Cleanup.** The extension implements proper memory management to prevent memory leaks during extended data collection sessions. Upon timeline stop, the extension clears frame tracking sets, resets the nuScenes manager's internal buffers, and releases sensor data caches. This cleanup ensures that memory consumption remains bounded regardless of session duration, enabling long-duration data collection without performance degradation.

- **Independent Sensor Processing.** Data collection operations execute independently for each sensor, with per-sensor error handling ensuring that failures in individual sensors do not halt entire collection cycles. Camera and LiDAR collection occur in separate try-catch blocks, allowing the system to continue capturing data from functioning sensors even if others encounter errors. This architecture maintains system robustness during data collection while preserving interface responsiveness.

- **Selective Processing.** Users can selectively enable or disable individual sensors to optimize performance based on specific requirements. Disabling unused sensors reduces processing overhead and storage requirements, enabling users to tailor data collection to their validation scenarios.

### 6.9.2 Comprehensive Error Handling

The extension implements robust error handling throughout all components to ensure continued operation even when individual sensors or operations encounter problems.

This multi-level error handling strategy ensures that the extension remains operational even when encountering unexpected conditions, preventing total failure due to individual sensor issues.

# Chapter 7

# Conclusions and Future Developments

## 7.1 Summary of Contributions

This thesis advances synthetic data generation for ADAS validation by addressing two critical limitations: vehicle-centric sensor restrictions and visual fidelity gaps. Building upon Stella's foundational work [7], this research introduces infrastructure-cooperative sensing and establishes a CARLA-Omniverse integration pipeline.

### 7.1.1 Infrastructure-Cooperative Sensor Simulation

The first contribution develops a comprehensive framework for infrastructure-based sensor simulation. The **sensor placement management tool** provides intuitive interface for positioning infrastructure sensors with automatic intersection detection, intelligent sensor-to-intersection association, and real-time visualization. Supported sensors include LIDAR_TOP, CAM_TRAFFIC, LIDAR_TRAFFIC, and RADAR_TRAFFIC.

The spawning mechanism differentiates between vehicle-attached and world-fixed sensors, implementing independent lifecycle management for stationary infrastructure sensors maintaining fixed world coordinates throughout simulations.

The **nuScenes format extensions** maintain full backward compatibility through an `intersection_id` metadata field, specialized coordinate transformation procedures, and processing paths converging to identical dataset structures, ensuring seamless integration with existing nuScenes devkit tools.

The framework enables previously challenging validation scenarios: complex intersection monitoring, occlusion analysis from elevated vantage points, cooperative perception algorithm testing, and V2I communication validation.

### 7.1.2 CARLA to Isaac Sim Integration Pipeline

The second contribution establishes technical foundation for bridging CARLA's simulation capabilities with environments offering advanced rendering potential.

The **USD export framework** captures entire CARLA simulations as temporal sequences with keyframed animations, implementing comprehensive scene capture including vehicle trajectories, sensor configurations for vehicle-mounted and infrastructure-based deployments, environmental conditions, and temporal synchronization. The framework handles coordinate system transformations between CARLA's left-handed and USD's right-handed conventions.

During development, CARLA 0.9.16 was released with official USD export capabilities [24]. While providing high-quality static map exports and proper material definitions, it lacks critical temporal animation support for simulation replay, multi-agent animations, sensor configuration export, and synchronized nuScenes-compatible data capture. These limitations validated continued development of the custom USD export framework focusing on temporal animation and simulation replay capabilities.

The implementation leverages official CARLA USD assets when available with a three-tier asset resolution strategy providing fallback mechanisms for compatibility across CARLA versions.

The **Isaac Sim extension** (`omni.carla.sim_controller`) completes the pipeline with simulation replay and data collection capabilities, implementing automatic sensor discovery, precise temporal control, synchronized multi-modal data capture, and automated nuScenes dataset generation.

A key innovation is seamless support for both vehicle-only and infrastructure-only deployment modes, enabling diverse validation requirements from traditional perception testing to infrastructure-centric monitoring applications.

## 7.2 Achievement of Research Objectives

Research objectives have been successfully achieved with strategic adaptations based on CARLA ecosystem developments.

The **infrastructure sensing capability** has been fully realized through production-ready sensor placement tool and complete nuScenes integration. The **CARLA-Isaac Sim integration** has been achieved through functional USD export pipeline and comprehensive Isaac Sim extension. The **backward compatibility** objective has been maintained throughout all implementations.

CARLA's official USD export framework announcement during development prompted a strategic pivot toward developing complementary capabilities addressing temporal animation and simulation replay—functionality not provided by the official framework, demonstrating pragmatic engineering decision-making.

## 7.3 Technical Limitations and Constraints

### 7.3.1 Infrastructure Sensing Limitations

The framework exhibits several constraints. The emphasis on **intersection-based scenarios** means highway monitoring and tunnel surveillance receive less optimization. The **computational overhead** from infrastructure sensors compounds performance challenges—each camera sensor incurs significant GPU-to-CPU transfer costs. The **fixed sensor type catalog** (LIDAR_TOP, CAM_TRAFFIC, LIDAR_TRAFFIC, RADAR_TRAFFIC) does not yet support emerging modalities like thermal imaging or acoustic sensors.

### 7.3.2 USD Export and Visual Fidelity Limitations

The framework successfully leverages official CARLA USD assets but exhibits limitations. The primary limitation is **static geometry representation**—while the framework animates vehicle positions, rotations, and velocities, USD assets are static meshes. Vehicle wheels do not rotate, steering wheels remain neutral, suspension systems do not compress, and doors remain closed. This impacts visual realism particularly in slow-speed maneuvers or sharp turns.

The export system does not handle *parked vehicles* (represented as static meshes rather than dynamic actors) or *pedestrians* (requiring skeletal mesh animation conversion to USD SkelAnimation format). Additional limitations include lack of support for weather effects, dynamic lighting, and traffic control device state changes. The **temporal animation system** uses linear interpolation that may not perfectly replicate CARLA's physics-based motion in high-dynamic maneuvers.

### 7.3.3 Isaac Sim Extension Limitations

The extension represents an initial proof-of-concept implementation with significant limitations. **Computational performance and loading times** are substantial—USD files with complete CARLA maps can require several minutes for initial loading. **Runtime performance** struggles to achieve real-time playback speeds (20-30 fps) with complete map exports and multiple active sensors. CARLA's official USD export arrived during final thesis stages, severely limiting time for **optimization and visual enhancement**. The extension demonstrates feasibility but requires extensive future work to achieve production-ready performance and robustness.

## 7.4 Impact and Practical Applications

### 7.4.1 Industrial Relevance

The collaboration with Reply - Concept Quality provided direct insights into industry validation requirements. The infrastructure sensing framework responds to automotive industry's increasing interest in V2I communication systems and cooperative perception architectures. The nuScenes format compatibility ensures datasets can be immediately utilized with existing validation pipelines, reducing adoption barriers.

### 7.4.2 Research and Development Applications

The frameworks enable research applications beyond traditional ADAS validation: cooperative perception algorithm development with synchronized multi-perspective datasets, urban traffic monitoring research for traffic flow analysis and intersection safety studies, and sim-to-real transfer learning to explore domain adaptation techniques.

## 7.5 Future Development Directions

### 7.5.1 Short-Term Enhancements

#### Performance Optimization for Isaac Sim Integration

The most critical near-term development involves addressing computational performance limitations. Priority efforts should focus on **scene loading performance** through LOD systems, selective asset loading, Omniverse instancing, and asset streaming. **Runtime performance optimization** should target render pipeline optimization, sensor scheduling strategies, configurable quality presets, and distributed rendering approaches.

#### Articulated Vehicle Components

Implementing articulated vehicle components would enhance visual realism. Priority features include wheel rotation animation, steering animation, suspension dynamics, and turn signal/brake light activation.

#### Extended Sensor Type Support

Adding semantic segmentation cameras, depth cameras, and thermal imaging simulation would expand framework applicability.

#### Pedestrian and Parked Vehicle Support

Pedestrian support requires converting CARLA's skeletal mesh animations to USD SkelAnimation format. Parked vehicle detection necessitates extending the framework to identify static mesh instances.

### 7.5.2 Medium-Term Extensions

#### Cooperative Sensing Mode

Extending to support simultaneous vehicle and infrastructure sensors would enable true cooperative sensing scenarios, requiring unified coordinate transformation procedures, sensor fusion ground truth generation, and nuScenes format extensions.

#### Dynamic Infrastructure Sensor Control

Implementing runtime control would enable sensors to activate/deactivate based on traffic conditions, adjust parameters dynamically, and simulate sensor failures.

#### Integration with Traffic Simulation Tools

Integration with tools like SUMO would enable realistic traffic patterns, complex multi-vehicle scenarios, and large-scale urban simulations.

### 7.5.3 Long-Term Research Directions

#### Advanced Rendering Integration

Full integration with Isaac Sim's RTX ray-traced rendering for photorealistic lighting, physically-based material systems, and Omniverse's asset libraries.

#### Procedural Scenario Generation

Automated scenario generation with parameterized complexity, automatic sensor placement based on coverage optimization, and edge case generation.

#### Real-World Validation and Calibration

Collecting paired real-world and synthetic datasets, developing metrics quantifying sim-to-real gap, and implementing calibration procedures.

### Standardization and Interoperability

Contributing to industry standards: proposing nuScenes format extensions for infrastructure sensors, participating in USD working groups, and collaborating with ASAM on OpenSCENARIO and OpenDRIVE integration.

## 7.6 Concluding Remarks

This thesis demonstrates that advanced synthetic data generation capabilities can be developed while maintaining compatibility with established validation frameworks and industry-standard data formats. The infrastructure-cooperative sensing framework addresses critical coverage limitations in vehicle-centric validation approaches. The CARLA-to-Isaac Sim integration establishes technical infrastructure that, while facing performance challenges, provides a foundation for production-ready ADAS validation workflows as both ecosystems evolve.

The pragmatic approach—maintaining backward compatibility, building upon existing frameworks, and adapting to ecosystem developments—ensures contributions remain relevant and adoptable by industry stakeholders. The frameworks are stepping stones in the ongoing evolution of ADAS validation methodologies.

The collaboration with Reply - Concept Quality ensured solutions address practical industry challenges. As ADAS technologies advance and regulatory requirements intensify, the frameworks provide tools addressing current needs and anticipating future requirements.

The future of ADAS validation lies in intelligent integration of synthetic and real-world approaches. This thesis advances that integration by providing new capabilities while maintaining compatibility with existing validation ecosystems. The technical foundations—particularly the infrastructure sensing framework and CARLA-Omniverse integration pipeline—provide platforms upon which the research community and industry practitioners can build increasingly sophisticated validation methodologies.

# Appendix A

# Infrastructure-Cooperative Sensor Simulation Code

## A.1 Intersection Detection

**Listing A.1:** Intersection Detection Algorithm

```
def find_and_mark_intersections(self):
    """
    Automatically detect and characterize road intersections
    in CARLA map.

    Uses CARLA's topology system to identify junction
   waypoints, groups them
    by junction ID, and calculates geometric properties for
   sensor placement.
    """
    carla_map = self.world.get_map()
    topology = carla_map.get_topology()

    # Extract all waypoints that are part of junctions
    junction_waypoints = []
    for w1, w2 in topology:
        if w1.is_junction:
            junction_waypoints.append(w1)
        if w2.is_junction:
            junction_waypoints.append(w2)

    # Group waypoints by their junction ID
    unique_junctions = {}
```

```
    for wp in junction_waypoints:
        jid = wp.get_junction().id
        if jid not in unique_junctions:
            unique_junctions[jid] = []
        unique_junctions[jid].append(wp)

    # Calculate geometric center and bounding extent for each junction
    self.intersection_centers = []
    for junction_id, waypoints in unique_junctions.items():
        # Compute centroid
        center_x = sum(w.transform.location.x for w in waypoints) / len(waypoints)
        center_y = sum(w.transform.location.y for w in waypoints) / len(waypoints)

        # Compute bounding box
        x_coords = [w.transform.location.x for w in waypoints]
        y_coords = [w.transform.location.y for w in waypoints]
        extent_x = (max(x_coords) - min(x_coords)) / 2.0
        extent_y = (max(y_coords) - min(y_coords)) / 2.0

        self.intersection_centers.append({
            'id': junction_id,
            'center': carla.Location(center_x, center_y, waypoints[0].transform.location.z),
            'extent': (extent_x, extent_y),
            'size': max(extent_x, extent_y) * 2.0
        })
```

## A.2 Configuration Management

**Listing A.2:** Infrastructure Sensor Configuration Format

```
{
  "LIDAR_TOP_189_b90f74": {
    "type": "LIDAR_TOP",
    "map": "Town10HD",
    "intersection_id": 189,
    "x": -47.8,
    "y": 20.4,
```

```
    "z": 8.0,
    "pitch": 0.0,
    "yaw": -90.0,
    "roll": 0.0,
    "range": "60.0",
    "nChannels": "32",
    "pointsPerSecond": "150000",
    "rotFrequency": "10.0"
  },
  "CAM_TRAFFIC_189_2e7a19": {
    "type": "CAM_TRAFFIC",
    "map": "Town10HD",
    "intersection_id": 189,
    "x": -63.0,
    "y": 4.6,
    "z": 5.0,
    "pitch": -15.0,
    "yaw": -120.0,
    "roll": 0.0,
    "resolutionW": "800",
    "resolutionH": "600",
    "fov": "90.0"
  }
}
```

## A.3 Sensor Spawning

**Listing A.3:** Infrastructure vs Vehicle-Mounted Sensor Spawning

```
def spawn_infrastructure_sensor(world, sensor_config):
    """
    Spawn infrastructure sensor at fixed world coordinates.

    Infrastructure sensors are independent actors that
   remain stationary
    throughout the simulation, providing persistent
   monitoring of specific
    locations regardless of vehicle movements.

    Args:
        world: CARLA world object
        sensor_config: Dictionary containing sensor
   configuration
```

```
12
13      Returns:
14          Spawned sensor actor
15      """
16      blueprint = world.blueprint_library.find(sensor_config['
    blueprint'])
17
18      # Configure sensor-specific parameters
19      for key, value in sensor_config['attributes'].items():
20          blueprint.set_attribute(key, str(value))
21
22      # Create transform using absolute world coordinates
23      transform = carla.Transform(
24          carla.Location(
25              x=sensor_config['x'],
26              y=sensor_config['y'],
27              z=sensor_config['z']
28          ),
29          carla.Rotation(
30              pitch=sensor_config['pitch'],
31              yaw=sensor_config['yaw'],
32              roll=sensor_config['roll']
33          )
34      )
35
36      # Spawn as independent actor (not attached to any
    vehicle)
37      sensor = world.spawn_actor(blueprint, transform)
38      sensor.listen(lambda data: process_sensor_data(data,
    sensor_config))
39
40      return sensor
41
42  def spawn_vehicle_sensor(world, ego_vehicle, sensor_config):
43      """
44      Spawn vehicle sensor attached to ego vehicle.
45
46      Vehicle sensors move with the ego vehicle and use
    relative coordinate
47      offsets from the vehicle's center. This is the
    traditional approach
48      used in Federico Stella's thesis.
49
50      Args:
51          world: CARLA world object
```

```
52          ego_vehicle: Vehicle actor to attach sensor to
53          sensor_config: Dictionary containing sensor
    configuration
54
55      Returns:
56          Spawned and attached sensor actor
57      """
58      blueprint = world.blueprint_library.find(sensor_config['
    blueprint'])
59
60      # Configure sensor-specific parameters
61      for key, value in sensor_config['attributes'].items():
62          blueprint.set_attribute(key, str(value))
63
64      # Create transform relative to vehicle center
65      relative_transform = carla.Transform(
66          carla.Location(
67              x=sensor_config['offset_x'],
68              y=sensor_config['offset_y'],
69              z=sensor_config['offset_z']
70          ),
71          carla.Rotation(
72              pitch=sensor_config['pitch'],
73              yaw=sensor_config['yaw'],
74              roll=sensor_config['roll']
75          )
76      )
77
78      # Attach to ego vehicle - sensor will follow vehicle
    motion
79      sensor = world.spawn_actor(
80          blueprint,
81          relative_transform,
82          attach_to=ego_vehicle
83      )
84      sensor.listen(lambda data: process_sensor_data(data,
    sensor_config))
85
86      return sensor
```

## A.4 Coordinate Transformations

**Listing A.4:** Infrastructure Sensor Coordinate Transformations

```
def create_calibrated_sensor_infrastructure(sensor_type, sensor_config):
    """
    Create calibrated_sensor entry for infrastructure sensor.

    Infrastructure sensors use world-fixed coordinates, requiring explicit
    transformation from CARLA's left-handed coordinate system to nuScenes'
    right-handed coordinate system. Unlike vehicle sensors, infrastructure
    sensors maintain constant world coordinates throughout the simulation.

    Coordinate System Conventions:
    - CARLA: Left-handed (X forward, Y right, Z up)
    - nuScenes: Right-handed (X right, Y forward, Z up)

    Args:
        sensor_type: Sensor channel identifier (e.g., 'LIDAR_TRAFFIC_189')
        sensor_config: Dictionary containing sensor configuration

    Returns:
        Dictionary containing calibrated sensor entry for nuScenes format
    """
    # Extract sensor pose in CARLA world coordinates
    carla_location = carla.Location(
        x=sensor_config['x'],
        y=sensor_config['y'],
        z=sensor_config['z']
    )

    carla_rotation = carla.Rotation(
        pitch=sensor_config['pitch'],
        yaw=sensor_config['yaw'],
        roll=sensor_config['roll']
    )

    # Convert CARLA rotation to rotation matrix for transformation
```

```
35      carla_transform = carla.Transform(carla_location,
    carla_rotation)
36      rotation_matrix = carla_transform.get_matrix()
37
38      # Apply coordinate frame conversion: CARLA to nuScenes
39      # This involves both translation and rotation
    transformations
40      translation_nuscenes = [
41          carla_location.x,       # nuScenes X = CARLA X
42          -carla_location.y,      # nuScenes Y = -CARLA Y (left
    -to-right hand)
43          carla_location.z        # nuScenes Z = CARLA Z
44      ]
45
46      # Apply rotation matrix transformation for handedness
    change
47      # Transformation matrix T converts CARLA rotation to
    nuScenes:
48      # T = [[1, 0, 0], [0, -1, 0], [0, 0, 1]]
49      # R_nuscenes = T @ R_carla @ T^T
50      R_carla = rotation_matrix[:3, :3]
51      T = np.array([[1, 0, 0], [0, -1, 0], [0, 0, 1]])
52      rotation_nuscenes = T @ R_carla @ T.T
53
54      # Create calibrated sensor entry compatible with
    nuScenes schema
55      calibrated_sensor = {
56          'token': generate_token(),
57          'sensor_token': sensor_config['sensor_token'],
58          'translation': translation_nuscenes,
59          'rotation': rotation_matrix_to_quaternion(
    rotation_nuscenes),
60          'camera_intrinsic': get_camera_intrinsic(sensor_type
    , sensor_config)
61                              if 'CAM_' in sensor_type else []
62      }
63
64      # Add infrastructure-specific metadata (optional
    extension fields)
65      if 'intersection_id' in sensor_config:
66          calibrated_sensor['intersection_id'] = sensor_config
    ['intersection_id']
67
68      return calibrated_sensor
```

# Appendix B

# CARLA to USD Export Framework Code

## B.1 USD Scene Exporter

**Listing B.1:** USDSceneExporter Core Structure

```
class USDSceneExporter:
    """
    Main exporter orchestrating USD conversion pipeline.

    Coordinates data capture from CARLA, temporal
   synchronization,
    and USD stage generation for Omniverse integration.
    """

    def __init__(self, carlaWorld, export_map=True):
        """
        Initialize USD exporter with CARLA world reference.

        Args:
            carlaWorld: CARLA world object containing
   simulation state
            export_map: If True, use official CARLA USD map
   exports;
                        if False, generate dynamic road
   network
        """
        self.carlaWorld = carlaWorld
        self.export_map = export_map
```

```
20
21          # Initialize core managers
22          self.timeline_manager = TimelineManager()
23          self.data_capture = DataCapture(carlaWorld,
    export_map)
24
25          # Scene data accumulator
26          self.scene_data = []
27          self.current_frame = 0
28
29      def capture_scene_state(self, timestamp, frame_number):
30          """
31          Capture complete scene state at current simulation
    timestep.
32
33          Extracts all relevant data from CARLA including
    vehicle states,
34          sensor configurations, and environmental conditions.
35
36          Args:
37              timestamp: CARLA simulation timestamp (
    microseconds)
38              frame_number: Sequential frame identifier
39
40          Returns:
41              Dictionary containing complete scene state
42          """
43          # Record timing information for temporal
    synchronization
44          self.timeline_manager.record_timestamp(timestamp)
45
46          # Determine if road geometry should be captured (
    first frame only)
47          include_roads = (frame_number == 0) and not self.
    export_map
48
49          # Extract complete scene data from CARLA simulation
50          scene_state = self.data_capture.capture_frame(
51              timestamp,
52              frame_number,
53              include_roads=include_roads
54          )
55
56          # Accumulate for final USD export
57          self.scene_data.append(scene_state)
```

```
58          self.current_frame = frame_number
59
60          return scene_state
```

## B.2 Data Capture

**Listing B.2:** Scene State Capture Implementation

```
1  class DataCapture:
2      """
3      Captures complete scene state from CARLA simulation.
4
5      Extracts vehicle trajectories, sensor configurations,
       environmental
6      conditions, and optionally road network geometry for USD
       export.
7      """
8
9      def __init__(self, carlaWorld, export_map):
10         self.world = carlaWorld
11         self.export_map = export_map
12         self.actor_processor = ActorProcessor()
13         self.road_processor = RoadProcessor(carlaWorld.
       get_map())
14
15     def capture_frame(self, timestamp, frame_number,
       include_roads=False):
16         """
17         Extract complete scene state at current frame.
18
19         Args:
20             timestamp: CARLA timestamp for this frame
21             frame_number: Sequential frame identifier
22             include_roads: Whether to include road network
       geometry
23
24         Returns:
25             Dictionary containing all scene data for this
       frame
26         """
27         frame_data = {
28             'timestamp': timestamp,
29             'frame_number': frame_number,
```

```
            'vehicles': [],
            'sensors': [],
            'weather': None,
            'roads': None
        }

        # Capture all vehicle states and transformations
        for actor in self.world.get_actors().filter('vehicle.*'):
            vehicle_data = self.actor_processor.process_actor(actor)
            if vehicle_data:
                frame_data['vehicles'].append(vehicle_data)

        # Capture sensor configurations (attached to ego vehicle)
        ego_vehicle = self.world.get_actors().filter('vehicle.*')[0]
        for sensor in self.world.get_actors().filter('sensor.*'):
            if sensor.parent and sensor.parent.id == ego_vehicle.id:
                sensor_data = self._process_sensor(sensor)
                frame_data['sensors'].append(sensor_data)

        # Capture road network geometry (first frame only, if using dynamic export)
        if include_roads and not self.export_map:
            frame_data['roads'] = self.road_processor.extract_road_network()

        # Capture current weather and lighting conditions
        frame_data['weather'] = self._capture_weather()

        return frame_data

    def _process_sensor(self, sensor):
        """Extract sensor configuration and spatial parameters"""
        return {
            'id': sensor.id,
            'type': sensor.type_id,
            'transform': sensor.get_transform(),
            'attributes': dict(sensor.attributes)
        }
```

```
66
67      def _capture_weather(self):
68          """Extract current weather parameters from CARLA
    world"""
69          weather = self.world.get_weather()
70          return {
71              'cloudiness': weather.cloudiness,
72              'precipitation': weather.precipitation,
73              'sun_altitude_angle': weather.sun_altitude_angle
    ,
74              'sun_azimuth_angle': weather.sun_azimuth_angle
75          }
```

## B.3 Coordinate System Transformations

**Listing B.3:** Coordinate System Transformation Implementation

```
1  class TransformUtils:
2      """Utility class for coordinate system conversions
    between CARLA and USD"""
3
4      @staticmethod
5      def convert_carla_to_usd_transform(carla_transform):
6          """
7          Convert CARLA transform to USD coordinate system.
8
9          Coordinate System Conventions:
10         - CARLA: Left-handed (X forward, Y right, Z up)
11         - USD: Right-handed (X right, Y up, Z back)
12
13         This conversion is critical for proper spatial
    representation in
14         Omniverse and other USD-based tools.
15
16         Args:
17             carla_transform: carla.Transform object
18
19         Returns:
20             Dictionary with 'position' and 'quaternion' in
    USD coordinates
21         """
22         location = carla_transform.location
23         rotation = carla_transform.rotation
```

```
24
25          # Convert position: Invert Y-axis for handedness
    change
26          position = [
27              location.x,    # X unchanged
28              -location.y,   # Y inverted (right -> left
    handedness)
29              location.z     # Z unchanged
30          ]
31
32          # Convert rotation to quaternion for USD
33          # CARLA uses degrees, USD uses radians
34          pitch_rad = math.radians(rotation.pitch)
35          yaw_rad = math.radians(-rotation.yaw)  # Negate yaw
    for handedness
36          roll_rad = math.radians(rotation.roll)
37
38          # Compute quaternion from Euler angles (ZYX rotation
     order)
39          cy = math.cos(yaw_rad * 0.5)
40          sy = math.sin(yaw_rad * 0.5)
41          cp = math.cos(pitch_rad * 0.5)
42          sp = math.sin(pitch_rad * 0.5)
43          cr = math.cos(roll_rad * 0.5)
44          sr = math.sin(roll_rad * 0.5)
45
46          qw = cr * cp * cy + sr * sp * sy
47          qx = sr * cp * cy - cr * sp * sy
48          qy = cr * sp * cy + sr * cp * sy
49          qz = cr * cp * sy - sr * sp * cy
50
51          # Normalize quaternion to ensure unit length
52          magnitude = math.sqrt(qw*qw + qx*qx + qy*qy + qz*qz)
53          quaternion = [qx/magnitude, qy/magnitude, qz/
    magnitude, qw/magnitude]
54
55          return {
56              'position': position,
57              'quaternion': quaternion
58          }
59
60      @staticmethod
61      def convert_velocity_to_usd(carla_velocity):
62          """
```

```
        Convert velocity vector from CARLA to USD coordinates.

        Args:
            carla_velocity: carla.Vector3D velocity vector

        Returns:
            List of three floats representing velocity in USD coordinates
        """
        return [
            carla_velocity.x,   # X component unchanged
            -carla_velocity.y,  # Y component inverted
            carla_velocity.z    # Z component unchanged
        ]
```

## B.4 Vehicle Animation

**Listing B.4:** Vehicle Animation Keyframe Generation

```
def _create_vehicle_animation(self, vehicle_xform, vehicle_timeline):
    """
    Generate keyframed animation for vehicle trajectory in USD.

    Creates temporal animation by setting transformation keyframes at
    computed USD time codes. This enables deterministic replay of the
    CARLA simulation in Omniverse.

    Args:
        vehicle_xform: USD Xform primitive for the vehicle
        vehicle_timeline: List of vehicle states across all frames
    """

    # Get or create USD transform operations
    # These define how the vehicle's position and orientation change over time
    translate_attr = vehicle_xform.AddTranslateOp()
    orient_attr = vehicle_xform.AddOrientOp()
```

```
18      scale_attr = vehicle_xform.AddScaleOp()
19
20      # Apply keyframes for each captured frame
21      for frame_data in vehicle_timeline:
22          # Convert CARLA timestamp to USD time code
23          # This ensures temporal alignment across all
    animated elements
24          usd_time = self.timeline_manager.
    timestamp_to_time_code(
25              frame_data['timestamp']
26          )
27
28          # Extract transformed position and rotation
29          position = frame_data['transform']['position']
30          quaternion = frame_data['transform']['quaternion']
31
32          # Set keyframe values at computed time code
33          translate_attr.Set(
34              Gf.Vec3f(position[0], position[1], position[2]),
35              time=usd_time
36          )
37
38          # USD quaternion format: (w, x, y, z)
39          orient_attr.Set(
40              Gf.Quatf(quaternion[3], quaternion[0],
    quaternion[1], quaternion[2]),
41              time=usd_time
42          )
43
44          # Scale remains constant (vehicle dimensions don't
    change)
45          if usd_time == 0:
46              scale_attr.Set(Gf.Vec3f(1.0, 1.0, 1.0))
```

## B.5 Asset Management

**Listing B.5:** Vehicle Asset Resolution System

```
1  def _create_vehicle_mesh(self, vehicle_xform, vehicle_data):
2      """
3      Create or reference vehicle mesh geometry using
    hierarchical fallback strategy.
4
```

```
    Attempts multiple asset resolution methods in order of visual quality:
    1. Official CARLA USD exports (CARLA 0.9.16+) - highest fidelity
    2. Custom USD asset library - intermediate quality
    3. Procedural bounding box - lowest quality fallback

    Args:
        vehicle_xform: USD Xform primitive for vehicle placement
        vehicle_data: Dictionary containing vehicle metadata
    """
    vehicle_path = str(vehicle_xform.GetPath())
    blueprint_id = vehicle_data.get('blueprint', 'unknown')

    # Attempt 1: Reference official CARLA USD export (CARLA 0.9.16+)
    official_usd_path = self._get_official_carla_usd_path(blueprint_id)
    if official_usd_path and os.path.exists(official_usd_path):
        vehicle_ref = self.stage.DefinePrim(f"{vehicle_path}/VehicleAsset")
        vehicle_ref.GetReferences().AddReference(official_usd_path)
        print(f"Using official CARLA USD asset: {official_usd_path}")
        return

    # Attempt 2: Reference custom USD asset from library
    custom_asset_url = self._get_custom_asset_url(blueprint_id)
    if custom_asset_url:
        try:
            vehicle_ref = self.stage.DefinePrim(f"{vehicle_path}/VehicleAsset")
            vehicle_ref.GetReferences().AddReference(custom_asset_url)
            print(f"Using custom USD asset: {custom_asset_url}")
            return
        except Exception as e:
            print(f"Custom asset reference failed: {e}")

    # Fallback: Generate simple bounding box geometry
```

```
37      # This ensures visual representation even without asset
    library
38      print(f"Using fallback geometry for {blueprint_id}")
39      self._create_fallback_geometry(vehicle_path,
    vehicle_data)
40
41  def _create_fallback_geometry(self, vehicle_path,
    vehicle_data):
42      """
43      Create simple bounding box representation for vehicle.
44
45      Generates colored box geometry as visual placeholder
    when
46      high-fidelity assets are unavailable.
47      """
48      mesh_info = vehicle_data.get('mesh_info', {})
49      extent = mesh_info.get('extent', [2.0, 1.0, 0.75])  #
    Default car dimensions
50
51      # Create box geometry
52      box_path = f"{vehicle_path}/Geometry"
53      box = UsdGeom.Cube.Define(self.stage, box_path)
54      box.CreateSizeAttr().Set(1.0)
55      box.CreateExtentAttr().Set([(-extent[0], -extent[1], -
    extent[2]),
56                                  (extent[0], extent[1],
    extent[2])])
57
58      # Apply distinctive color based on vehicle ID
59      color = self.get_vehicle_color(vehicle_data['id'])
60      box.CreateDisplayColorAttr().Set([Gf.Vec3f(color[0],
    color[1], color[2])])
```

## B.6 Sensor Export

**Listing B.6:** Camera Sensor USD Creation

```
1   def _create_camera_sensor(self, sensor_id, sensor_config,
    base_path, attached=False):
2       """
3       Create USD Camera representation with complete intrinsic
        parameters.
4
```

```
    Generates proper USD Camera primitive with focal length, aperture,
    and field of view parameters compatible with Isaac Sim rendering.

    Args:
        sensor_id: Unique sensor identifier
        sensor_config: Dictionary containing camera parameters
        base_path: USD path for sensor placement
        attached: Whether sensor is attached to vehicle or world-fixed

    Returns:
        Created UsdGeom.Xform primitive
    """
    camera_path = f"{base_path}/{sensor_id}"
    camera_xform = UsdGeom.Xform.Define(self.stage, camera_path)

    # Create Camera primitive
    camera = UsdGeom.Camera.Define(self.stage, f"{camera_path}/Camera")

    # Extract camera intrinsic parameters from CARLA configuration
    width = sensor_config.get('image_size_x', 1920)
    height = sensor_config.get('image_size_y', 1080)
    fov = sensor_config.get('fov', 90.0)

    # Calculate focal length from FOV and sensor dimensions
    # Standard pinhole camera model: focal_length = (sensor_width / 2) / tan(fov / 2)
    sensor_width = 36.0  # mm, standard full-frame sensor
    focal_length = (sensor_width / 2.0) / math.tan(math.radians(fov / 2.0))

    # Set USD camera intrinsic attributes
    camera.CreateFocalLengthAttr().Set(focal_length)
    camera.CreateHorizontalApertureAttr().Set(sensor_width)
    camera.CreateVerticalApertureAttr().Set(sensor_width * height / width)

    # Set camera spatial transform
    transform = sensor_config.get('transform', {})
```

```
        self._apply_sensor_transform(camera_xform, transform, attached)

        # Set clipping planes for rendering
        camera.CreateClippingRangeAttr().Set(Gf.Vec2f(0.1, 1000.0))

        return camera_xform
```

**Listing B.7:** LiDAR Sensor USD Creation

```
def _create_lidar_sensor(self, sensor_id, sensor_config, base_path, attached=False):
    """
    Create USD LiDAR representation with Omniverse asset reference.

    Attempts to reference realistic LiDAR sensor models from Omniverse
    asset library. Falls back to simple geometric proxy if assets unavailable.

    Args:
        sensor_id: Unique sensor identifier
        sensor_config: Dictionary containing LiDAR parameters
        base_path: USD path for sensor placement
        attached: Whether sensor is attached to vehicle or world-fixed

    Returns:
        Created UsdGeom.Xform primitive
    """
    lidar_path = f"{base_path}/{sensor_id}"
    lidar_xform = UsdGeom.Xform.Define(self.stage, lidar_path)

    # Attempt to reference Omniverse LiDAR asset (Ouster OS0 model)
    lidar_asset_url = "omniverse://localhost/NVIDIA/Assets/Isaac/4.5/" \
                      "Isaac/Sensors/Ouster/OS0/OS0_REV6_128_10hz.usd"

    try:
```

```
25          lidar_ref = self.stage.DefinePrim(f"{lidar_path}/
    LidarAsset")
26          lidar_ref.GetReferences().AddReference(
    lidar_asset_url)
27          print(f"Referenced LiDAR asset from Omniverse
    library")
28      except Exception as e:
29          # Fallback: Create simple cylinder geometry as
    visual placeholder
30          print(f"LiDAR asset reference failed ({e}), using
    fallback geometry")
31          lidar_geom = UsdGeom.Cylinder.Define(self.stage,
32                                               f"{lidar_path}/
    LidarGeometry")
33          lidar_geom.CreateHeightAttr().Set(0.1)
34          lidar_geom.CreateRadiusAttr().Set(0.05)
35          lidar_geom.CreateDisplayColorAttr().Set([Gf.Vec3f
    (0.0, 1.0, 0.0)])
36
37      # Apply spatial transform
38      transform = sensor_config.get('transform', {})
39      self._apply_sensor_transform(lidar_xform, transform,
    attached)
40
41      # Store LiDAR technical parameters as USD metadata
42      # This enables potential integration with Isaac Sim
    sensor simulation
43      lidar_prim = self.stage.GetPrimAtPath(lidar_path)
44      lidar_prim.SetMetadata('lidar:channels',
45                             sensor_config.get('channels', 64)
    )
46      lidar_prim.SetMetadata('lidar:range',
47                             sensor_config.get('range', 100.0)
    )
48      lidar_prim.SetMetadata('lidar:rotation_frequency',
49                             sensor_config.get('
    rotation_frequency', 10.0))
50
51      return lidar_xform
```

## B.7 Environment Export

**Listing B.8:** Official USD Map Loading

```
def _setup_environment_with_official_map(self, map_name):
    """
    Load official CARLA USD map export (CARLA 0.9.16+).

    Leverages NVIDIA's official CARLA-to-USD export pipeline for
    high-fidelity environmental representation. Falls back to dynamic
    road generation if official exports unavailable.

    Args:
        map_name: CARLA map identifier (e.g., 'Town01', 'Town10HD')

    Returns:
        Boolean indicating success of official map loading
    """
    # Construct path to official CARLA USD map export
    # CARLA 0.9.16+ exports maps to: carla/Export/Maps/
    carla_install_path = os.environ.get('CARLA_ROOT', '/opt/carla')
    map_usd_path = f"{carla_install_path}/Export/Maps/{map_name}.usd"

    if not os.path.exists(map_usd_path):
        print(f"Official USD map not found: {map_usd_path}")
        print("Falling back to dynamic road generation")
        return False

    # Reference the map USD into the environment hierarchy
    # This creates a link to the external USD file rather than copying content
    env_path = "/World/Environment"
    env_xform = UsdGeom.Xform.Define(self.stage, env_path)

    map_ref_path = f"{env_path}/Map"
    map_prim = self.stage.DefinePrim(map_ref_path)
    map_prim.GetReferences().AddReference(map_usd_path)

    print(f"Successfully loaded official USD map: {map_name}")
    return True
```

# Appendix C

# Isaac Sim Extension Development Code

## C.1 Extension Initialization

**Listing C.1:** Extension Initialization and Component Coordination

```
class SimControllerExtension(omni.ext.IExt):
    """
    Main extension class coordinating all simulation control components.

    Orchestrates sensor management, timeline control, nuScenes dataset
    generation, and user interface within Isaac Sim environment.
    """

    def on_startup(self, ext_id):
        """
        Initialize all components in proper dependency order.

        The initialization sequence ensures that dependencies are satisfied
        before dependent components are created:
        1. nuScenes Manager (data format specification)
        2. Sensor Manager (data collection, depends on nuScenes format)
```

```
        3. Timeline Controller (playback control, triggers data collection)
        4. UI Controller (user interface, orchestrates all other components)
        """
        try:
            # Step 1: Initialize nuScenes manager for dataset generation
            # Provides infrastructure sensor support via is_fixed parameter
            self._nuscenes_manager = NuScenesManager(
                vehicle_name="Car",
                map="Town10HD",
                is_fixed=True  # Enable infrastructure sensor support
            )

            # Step 2: Initialize sensor manager with nuScenes integration
            # Links sensor data collection to dataset generation pipeline
            self._sensor_manager = SensorManager(
                nuscenes_manager=self._nuscenes_manager
            )

            # Step 3: Initialize timeline controller for playback management
            # Registers callbacks for frame-by-frame data collection
            self._timeline_controller = TimelineController(
                on_timeline_event_callback=self._on_timeline_event,
                on_timeline_stop_callback=self._stop_simulation,
            )

            # Step 4: Initialize UI controller as coordination layer
            # Provides user interface for all other components
            self._ui_controller = UIController(
                self._sensor_manager,
                self._timeline_controller,
                self._nuscenes_manager
            )
```

```
49
50              print("[CARLA Sim Controller] All components
    initialized successfully")
51
52          except Exception as e:
53              print(f"[CARLA Sim Controller] Initialization
    error: {e}")
54              # Proper error handling ensures graceful
    degradation
```

## C.2 Infrastructure Sensor Support

**Listing C.2:** Enhanced DataLog Structure Supporting Infrastructure Sensors

```
1  class DataLog:
2      """
3      Extended data accumulation structure supporting
    cooperative sensing.
4
5      Maintains all sensor data captured during a single
    simulation frame,
6      including both vehicle-mounted and infrastructure
    sensors. The structure
7      extends Federico Stella's original implementation with
    infrastructure
8      sensor support through the _LID_INTER_val field.
9      """
10
11     def __init__(self, sample):
12         self._sample = sample                # Sample/frame
    identifier
13         self._timestamp = None               # Unix
    timestamp in microseconds
14
15         # Camera data from all sensors (vehicle +
    infrastructure)
16         # Combined storage enables unified processing
    pipeline
17         self._RGB_val = []
18
19         # Vehicle-mounted LiDAR data (original Stella
    implementation)
20         self._LID_val = []
```

```
21
22          # Infrastructure-mounted LiDAR data (NEW - this
    thesis contribution)
23          # Separate storage enables appropriate coordinate
    transformation
24          self._LID_INTER_val = []
25
26          # Ego vehicle transformation (pose) for vehicle-
    relative coordinates
27          self._egoTransform = None
28
29          # Scene actors for 3D bounding box annotation
    generation
30          self._actors = []
```

**Listing C.3:** Infrastructure Sensor Detection and Coordinate Handling

```
1  def detect_sensor_type(sensor_path: str) -> str:
2      """
3      Determine if sensor is vehicle-mounted or infrastructure
    -based.
4
5      Uses USD scene graph path analysis to classify sensor
    attachment type.
6      This classification is critical for applying appropriate
     coordinate
7      transformations during data processing.
8
9      Args:
10         sensor_path: Full USD scene graph path to sensor
11
12     Returns:
13         'vehicle' for ego-vehicle-attached sensors
14         'infrastructure' for world-fixed sensors
15     """
16     # Vehicle sensors follow pattern: /World/Car/...
17     if '/Car/' in sensor_path or sensor_path.startswith('/
    Car'):
18         return 'vehicle'
19
20     # Infrastructure sensors are placed directly in world
    hierarchy
21     # Typical naming patterns: /World/LIDAR_TRAFFIC_... or /
    World/CAM_TRAFFIC_...
22     if 'TRAFFIC' in sensor_path or 'INTER' in sensor_path:
```

```
        return 'infrastructure'

    # Default to vehicle for backward compatibility with Stella's implementation
    return 'vehicle'

def apply_coordinate_transform(point_cloud, sensor_type, ego_transform):
    """
    Apply appropriate coordinate transformation based on sensor type.

    Infrastructure and vehicle sensors require different coordinate handling:
    - Infrastructure sensors: Already in world coordinates, only need
      coordinate convention adjustment (Isaac Sim -> nuScenes)
    - Vehicle sensors: Require transformation to ego-relative frame,
      then coordinate convention adjustment

    Args:
        point_cloud: Nx4 numpy array of LiDAR points (x, y, z, intensity)
        sensor_type: 'vehicle' or 'infrastructure'
        ego_transform: Current ego vehicle transformation matrix

    Returns:
        Transformed point cloud in nuScenes coordinate convention
    """
    if sensor_type == 'infrastructure':
        # Infrastructure sensors already in world coordinates
        # Only apply nuScenes coordinate convention adjustment
        return convert_to_nuscenes_coords(point_cloud)
    else:
        # Vehicle sensors need ego-relative transformation first
        ego_relative = transform_to_ego_frame(point_cloud, ego_transform)
        return convert_to_nuscenes_coords(ego_relative)
```

**Listing C.4:** Infrastructure Sensor Metadata in nuScenes Format

```
def generate_infrastructure_sensor_entry(sensor_info):
    """
    Generate sensor table entry with infrastructure-specific
    metadata.

    Creates nuScenes-compatible sensor entry with optional
    extension fields
    that identify infrastructure sensors and associate them
    with monitored
    intersections. These optional fields maintain backward
    compatibility
    while enabling infrastructure sensor support.

    Args:
        sensor_info: Dictionary containing sensor
    configuration

    Returns:
        Dictionary formatted as nuScenes sensor table entry
    """
    return {
        'token': generate_unique_token(),
        'channel': sensor_info['channel'],  # e.g., '
    LIDAR_TRAFFIC_189'
        'modality': sensor_info['modality'], # 'lidar' or '
    camera'

        # Standard nuScenes fields
        'is_ego_vehicle': False,  # Infrastructure sensors
    not on ego vehicle

        # Infrastructure-specific extension fields (optional
    )
        # These fields are ignored by standard nuScenes
    tools but enable
        # infrastructure-aware processing in custom analysis
     pipelines
        'is_fixed': True,                    # World-fixed
    sensor indicator
        'intersection_id': sensor_info['intersection_id'],
    # Junction association
```

```
        'sensor_type': 'infrastructure',      # Explicit type identification
    }
```

## C.3 Sensor Discovery and Management

**Listing C.5:** Multi-Method Sensor Discovery Implementation

```
def scan_for_sensors(self):
    """
    Comprehensive sensor discovery using multiple detection methods.

    Implements robust sensor detection that handles various USD representations
    resulting from different export procedures. Uses hierarchical detection
    strategy with multiple fallback methods to ensure comprehensive coverage.

    Detection Methods:
    1. USD Camera primitive type detection (standard USD cameras)
    2. Name pattern matching for LiDAR sensors
    3. Asset reference detection for complex sensors
    4. Metadata-based detection for explicitly tagged sensors

    Returns:
        Dictionary with counts: {'cameras': int, 'lidars': int}
    """
    stage = get_current_stage()
    discovered_cameras = {}
    discovered_lidars = {}

    # Method 1: Traverse all primitives in scene hierarchy
    for prim in Usd.PrimRange(stage.GetPseudoRoot()):
        prim_path = str(prim.GetPath())

        # Camera detection via USD Camera primitive type
        if prim.IsA(UsdGeom.Camera):
            camera_name = prim.GetName()
```

```
29              try:
30                  # Initialize Isaac Sim Camera wrapper for
    data access
31                  camera_object = Camera(prim_path)
32                  camera_object.initialize()
33                  discovered_cameras[camera_name] =
    camera_object
34                  print(f"[Sensor Manager] Detected camera: {
    camera_name}")
35              except Exception as e:
36                  print(f"[Sensor Manager] Camera
    initialization failed "
37                        f"for {camera_name}: {e}")
38
39          # LiDAR detection via name pattern matching
40          # Supports multiple naming conventions: LIDAR, LID,
    RTX, Ouster
41          if any(pattern in prim_path.upper() for pattern in
42                 ['LIDAR', 'LID_', 'RTX', 'OUSTER']):
43              lidar_name = prim.GetName()
44              lidar_info = {
45                  'path': prim_path,
46                  'type': self._determine_lidar_type(prim),
47                  'prim': prim
48              }
49              discovered_lidars[lidar_name] = lidar_info
50              print(f"[Sensor Manager] Detected LiDAR: {
    lidar_name}")
51
52      # Update internal state with discovered sensors
53      self.available_cameras = discovered_cameras
54      self.available_lidars = discovered_lidars
55
56      # Initialize selection state (all sensors disabled by
    default)
57      self.selected_cameras = {name: False for name in
    discovered_cameras}
58      self.selected_lidars = {name: False for name in
    discovered_lidars}
59
60      return {
61          'cameras': len(discovered_cameras),
62          'lidars': len(discovered_lidars)
63      }
```

## C.4 Data Collection

**Listing C.6:** Camera Data Collection and Processing

```
def save_camera_data(self, current_frame: int, save_data: bool = True):
    """
    Capture and save data from all selected cameras.

    Implements robust per-camera error handling to ensure failures in
    individual cameras don't halt entire collection process.

    Args:
        current_frame: Current simulation frame number
        save_data: If True, write data to disk; if False, only process
    """
    try:
        selected_cameras = [name for name, selected in self.selected_cameras.items()
                            if selected]

        for camera_name in selected_cameras:
            try:
                camera = self.available_cameras[camera_name]

                # Initialize camera if not already initialized
                if not camera.is_initialized():
                    self._initialize_camera(camera, camera_name)

                # Retrieve raw camera data from Isaac Sim
                camera_data = self._get_camera_data(camera, camera_name)

                if camera_data is not None and 'rgba' in camera_data:
                    if save_data:
                        # Process and save image data
                        self._save_camera_image(
                            camera_data['rgba'],
                            camera_name,
                            current_frame
```

```
34                         )
35
36                     # Update collection tracking
37                     self.last_saved_frames[f"cam_{
    camera_name}"] = current_frame
38                 else:
39                     print(f"[Sensor Manager] No valid data
    for camera: {camera_name}")
40
41             except Exception as cam_error:
42                 # Per-camera error handling prevents total
    collection failure
43                 print(f"[Sensor Manager] Error processing {
    camera_name}: {cam_error}")
44
45     except Exception as e:
46         print(f"[Sensor Manager] Camera collection error: {e
    }")
47
48 def _save_camera_image(self, rgb_data, camera_name: str,
    current_frame: int):
49     """Process and save camera image with format
    normalization"""
50
51     if not isinstance(rgb_data, np.ndarray) or rgb_data.size
     == 0:
52         return
53
54     # Normalize data type to uint8 for consistent image
    format
55     if rgb_data.dtype in [np.float32, np.float64]:
56         # Float data assumed to be in [0, 1] range
57         rgb_data = (np.clip(rgb_data, 0, 1) * 255).astype(np
    .uint8)
58     elif rgb_data.dtype != np.uint8:
59         rgb_data = rgb_data.astype(np.uint8)
60
61     # Handle different channel configurations
62     if len(rgb_data.shape) == 3 and rgb_data.shape[:2] !=
    (0, 0):
63         if rgb_data.shape[2] == 4:
64             # RGBA to RGB conversion (discard alpha channel)
65             rgb_data = rgb_data[:, :, :3]
66
67         # Convert numpy array to PIL Image for saving
```

```
68          image = Image.fromarray(rgb_data)
69
70          # Create nuScenes-compatible directory structure
71          camera_dir = os.path.join(self.save_path,
    camera_name)
72          os.makedirs(camera_dir, exist_ok=True)
73
74          # Generate filename with zero-padded frame number
75          filename = f"frame_{current_frame:06d}.png"
76          filepath = os.path.join(camera_dir, filename)
77
78          # Save with high quality PNG compression
79          image.save(filepath, quality=95)
80
81          # Record in nuScenes manager for metadata generation
82          self.nuscenes_manager.record_camera_sample(
83              camera_name, filepath, current_frame
84          )
```

**Listing C.7:** LiDAR Data Collection with Infrastructure Support

```
1  def save_lidar_data(self, current_frame: int, save_data:
       bool = True):
2      """
3      Capture and save LiDAR point cloud data.
4
5      Handles both vehicle-mounted and infrastructure LiDAR
       sensors with
6      appropriate coordinate transformations for each type.
       Implements
7      multi-method data retrieval to handle various Isaac Sim
       LiDAR interfaces.
8
9      Args:
10         current_frame: Current simulation frame number
11         save_data: If True, write data to disk; if False,
       only process
12     """
13     try:
14         selected_lidars = [name for name, selected in self.
       selected_lidars.items()
15                            if selected]
16
17         for lidar_name in selected_lidars:
18             try:
```

```
19                 lidar_info = self.available_lidars[
   lidar_name]
20
21                 # Retrieve point cloud data using
   appropriate interface
22                 point_cloud = self._get_lidar_point_cloud(
   lidar_info)
23
24                 if point_cloud is not None and len(
   point_cloud) > 0:
25                     # Determine sensor type for proper
   coordinate handling
26                     sensor_type = detect_sensor_type(
   lidar_info['path'])
27
28                     if save_data:
29                         # Save point cloud in multiple
   formats
30                         self._save_lidar_pointcloud(
31                             point_cloud,
32                             lidar_name,
33                             current_frame,
34                             sensor_type
35                         )
36
37                     # Record in nuScenes manager with sensor
    type information
38                     if sensor_type == 'infrastructure':
39                         self.nuscenes_manager.
   record_infrastructure_lidar_sample(
40                             lidar_name, point_cloud,
   current_frame
41                         )
42                     else:
43                         self.nuscenes_manager.
   record_vehicle_lidar_sample(
44                             lidar_name, point_cloud,
   current_frame
45                         )
46
47                     self.last_saved_frames[f"lidar_{
   lidar_name}"] = current_frame
48
49             except Exception as lidar_error:
```

```
                print(f"[Sensor Manager] Error processing {lidar_name}: {lidar_error}")

        except Exception as e:
            print(f"[Sensor Manager] LiDAR collection error: {e}")

def _save_lidar_pointcloud(self, point_cloud, lidar_name, current_frame, sensor_type):
    """
    Save LiDAR point cloud in multiple formats for flexibility.

    Generates three output files per point cloud:
    - .npy: Binary NumPy format (efficient, preserves precision)
    - .csv: Text format (human-readable, debugging)
    - .json: Metadata (sensor info, frame info, statistics)
    """
    # Create sensor-specific directory
    lidar_dir = os.path.join(self.save_path, lidar_name)
    os.makedirs(lidar_dir, exist_ok=True)

    base_name = f"frame_{current_frame:06d}"

    # Save NumPy binary format (primary format for processing)
    npy_path = os.path.join(lidar_dir, f"{base_name}.npy")
    np.save(npy_path, point_cloud)

    # Save CSV format (for tools preferring text input)
    csv_path = os.path.join(lidar_dir, f"{base_name}.csv")
    np.savetxt(csv_path, point_cloud, delimiter=',',
               header='x,y,z,intensity', comments='')

    # Save metadata JSON
    metadata = {
        'frame': current_frame,
        'sensor_name': lidar_name,
        'sensor_type': sensor_type,
        'point_count': len(point_cloud),
        'timestamp': int(time.time() * 1_000_000),
        'bounds': {
            'x_min': float(np.min(point_cloud[:, 0])),
            'x_max': float(np.max(point_cloud[:, 0])),
```

```
            'y_min': float(np.min(point_cloud[:, 1])),
            'y_max': float(np.max(point_cloud[:, 1])),
            'z_min': float(np.min(point_cloud[:, 2])),
            'z_max': float(np.max(point_cloud[:, 2]))
        }
    }

    json_path = os.path.join(lidar_dir, f"{base_name}.json")
    with open(json_path, 'w') as f:
        json.dump(metadata, f, indent=2)
```

## C.5 Timeline Control

**Listing C.8:** Timeline Controller Implementation

```
class TimelineController:
    """
    Manages simulation timeline control and data collection
    synchronization.

    Interfaces with Isaac Sim's timeline system to control
    playback and
    trigger synchronized data collection at appropriate
    intervals.
    """

    def __init__(self, on_timeline_event_callback,
    on_timeline_stop_callback):
        """
        Initialize timeline controller with event callbacks.

        Args:
            on_timeline_event_callback: Called on each
    timeline update
            on_timeline_stop_callback: Called when timeline
    stops
        """
        self._timeline = omni.timeline.
    get_timeline_interface()
        self._on_event_callback = on_timeline_event_callback
        self._on_stop_callback = on_timeline_stop_callback
        self._is_playing = False
        self._current_frame = 0
```

```
22
23          # Register with Isaac Sim's timeline event stream
24          self._timeline_event_stream = self._timeline.
    get_timeline_event_stream()
25          self._timeline_subscription = \
26              self._timeline_event_stream.
    create_subscription_to_pop(
27                  self._on_timeline_event
28              )
29
30      def _on_timeline_event(self, event):
31          """
32          Process timeline events from Isaac Sim.
33
34          Handles play/pause state changes, frame position
    updates,
35          and timeline reset events.
36          """
37          if event.type == int(omni.timeline.TimelineEventType
    .PLAY):
38              self._is_playing = True
39              self._current_frame = int(
40                  self._timeline.get_current_time() *
41                  self._timeline.get_time_codes_per_seconds()
42              )
43
44          elif event.type == int(omni.timeline.
    TimelineEventType.PAUSE):
45              self._is_playing = False
46
47          elif event.type == int(omni.timeline.
    TimelineEventType.STOP):
48              self._is_playing = False
49              self._on_stop_callback()  # Trigger dataset
    generation
50
51          elif event.type == int(omni.timeline.
    TimelineEventType.CURRENT_TIME_CHANGED):
52              self._current_frame = int(
53                  self._timeline.get_current_time() *
54                  self._timeline.get_time_codes_per_seconds()
55              )
56
57              # Invoke data collection callback during
    playback
```

```
58              if self._is_playing:
59                  self._on_event_callback(self._current_frame,
    self._is_playing)
60
61      def play(self):
62          """Start timeline playback"""
63          self._timeline.play()
64
65      def pause(self):
66          """Pause timeline playback"""
67          self._timeline.pause()
68
69      def stop(self):
70          """Stop timeline and trigger dataset generation"""
71          self._timeline.stop()
72
73      def get_current_frame(self):
74          """Get current frame number"""
75          return self._current_frame
76
77      def is_playing(self):
78          """Check if timeline is currently playing"""
79          return self._is_playing
```

**Listing C.9:** Frame-Level Data Collection Synchronization

```
1  def _on_timeline_event(self, current_frame: int, is_playing:
       bool):
2      """
3      Handle timeline events and orchestrate synchronized data
       collection.
4
5      Called by Timeline Controller for each frame during
      playback.
6      Implements sampling rate control and prevents duplicate
      collection.
7
8      Args:
9          current_frame: Current frame number from timeline
10         is_playing: Whether timeline is actively playing
11     """
12     try:
13         if not is_playing or not hasattr(self, '
       _sensor_manager'):
14             return
```

```
15
16          # Apply sampling rate to control dataset density
17          # Example: sampling_rate=5 means collect every 5th
    frame
18          if current_frame % self._sampling_rate != 0:
19              return
20
21          # Prevent duplicate collection of same frame
22          if current_frame in self._saved_frames:
23              return
24
25          # Mark frame as collected
26          self._saved_frames.add(current_frame)
27
28          # Create new DataLog entry for nuScenes format
29          data_log = DataLog(current_frame)
30          data_log._timestamp = int(time.time() * 1_000_000)
    # Unix microseconds
31
32          # Capture ego vehicle transformation for this frame
33          data_log._egoTransform = self._get_ego_transform()
34
35          # Add to nuScenes manager's data collection
36          self._nuscenes_manager.data_list.append(data_log)
37
38          # Trigger synchronized multi-modal sensor data
    collection
39          self._sensor_manager.save_sensor_data(current_frame)
40
41          # Update UI with current progress
42          if hasattr(self, '_ui_controller'):
43              self._ui_controller.update_status()
44
45      except Exception as e:
46          print(f"[Extension] Timeline callback error: {e}")
47          # Log error but continue collection for remaining
    frames
```

## C.6 nuScenes Dataset Generation

**Listing C.10:** Orchestrated nuScenes Dataset Generation

```
1 def generate_dataset(self):
```

```
    """
    Generate complete nuScenes dataset from collected data.

    Produces all thirteen required JSON tables with proper referential
    integrity and temporal structure. Infrastructure sensor support is
    integrated throughout the generation process.

    Table Generation Order (ensures referential integrity):
    1. Ontology tables (category, attribute, visibility)
    2. Sensor tables (sensor, calibrated_sensor)
    3. Meta-information (log, map)
    4. Temporal structure (scene, sample, sample_data)
    5. Poses and annotations (ego_pose, instance, sample_annotation)
    """
    if not self.save_enabled:
        print("[nuScenes Manager] Save disabled, skipping dataset generation")
        return

    try:
        print("[nuScenes Manager] Beginning dataset generation...")

        # Phase 1: Generate ontology tables (order-independent)
        self._generate_categories()      # Object classification taxonomy
        self._generate_attributes()      # Object attribute definitions
        self._generate_visibility()      # Visibility level definitions

        # Phase 2: Generate sensor configuration tables
        self._generate_sensors()         # Sensor type definitions
        self._generate_calibrated_sensors()  # Sensor calibration parameters

        # Phase 3: Generate meta-information tables
        log_token = self._generate_log()  # Data collection session info
```

```
34          self._generate_map(log_token)      # Map/environment
    information
35
36          # Phase 4: Generate temporal structure
37          scene_token = self._generate_scene(log_token)  #
    Scene definition
38          self._generate_samples(scene_token)  # Keyframe
    samples
39
40          # Phase 5: Generate sensor data references
41          self._generate_samples_data()  # Link samples to
    sensor files
42
43          # Phase 6: Generate pose and annotation data
44          self._generate_ego_pose()         # Vehicle
    trajectory
45          self._generate_instances()        # Object tracking
    identities
46          self._generate_sample_annotations()  # 3D bounding
    boxes
47
48          # Phase 7: Export all tables to JSON files
49          self._export_all_tables()
50
51          print("[nuScenes Manager] Dataset generation
    complete!")
52          print(f"[nuScenes Manager] Output directory: {self.
    output_path}")
53
54      except Exception as e:
55          print(f"[nuScenes Manager] Dataset generation error:
     {e}")
56          raise
57
58  def _export_all_tables(self):
59      """
60      Export all database tables to JSON files.
61
62      Creates standard nuScenes v1.0-trainval directory
    structure with
63      one JSON file per table.
64      """
65      tables = {
66          'category': self.categories,
67          'attribute': self.attributes,
```

```
        'visibility': self.visibility_levels,
        'sensor': self.sensors,
        'calibrated_sensor': self.calibrated_sensors,
        'log': self.logs,
        'map': self.maps,
        'scene': self.scenes,
        'sample': self.samples,
        'sample_data': self.sample_data_entries,
        'ego_pose': self.ego_poses,
        'instance': self.instances,
        'sample_annotation': self.sample_annotations,
    }

    output_dir = os.path.join(self.output_path, 'v1.0-trainval')
    os.makedirs(output_dir, exist_ok=True)

    for table_name, table_data in tables.items():
        output_file = os.path.join(output_dir, f"{table_name}.json")

        with open(output_file, 'w') as f:
            json.dump(table_data, f, indent=2)

        print(f"[nuScenes Manager] Exported {table_name}.json "
              f"({len(table_data)} entries)")
```

**Listing C.11:** Unified Sample Structure with Infrastructure Sensors

```
def _generate_samples_data(self):
    """
    Generate sample_data table linking samples to sensor files.

    Creates entries for both vehicle and infrastructure sensor data,
    ensuring proper temporal alignment and metadata preservation.
    Each sample_data entry represents one sensor observation at one
    temporal keyframe.
    """
    self.sample_data_entries = []

```

```
12      for sample_info in self.collected_samples:
13          sample_token = sample_info['token']
14          frame_number = sample_info['frame']
15
16          # Process vehicle-mounted camera data (Stella's
    original)
17          for camera_name in self.vehicle_cameras:
18              if camera_name in sample_info['camera_files']:
19                  entry = self.
    _create_camera_sample_data_entry(
20                      sample_token=sample_token,
21                      camera_name=camera_name,
22                      file_path=sample_info['camera_files'][
    camera_name],
23                      frame=frame_number,
24                      is_infrastructure=False
25                  )
26                  self.sample_data_entries.append(entry)
27
28          # Process infrastructure camera data (NEW - this
    thesis)
29          for camera_name in self.infrastructure_cameras:
30              if camera_name in sample_info['camera_files']:
31                  entry = self.
    _create_camera_sample_data_entry(
32                      sample_token=sample_token,
33                      camera_name=camera_name,
34                      file_path=sample_info['camera_files'][
    camera_name],
35                      frame=frame_number,
36                      is_infrastructure=True  # Infrastructure
     sensor flag
37                  )
38                  self.sample_data_entries.append(entry)
39
40          # Process vehicle-mounted LiDAR data (Stella's
    original)
41          if 'lidar_file' in sample_info:
42              entry = self._create_lidar_sample_data_entry(
43                  sample_token=sample_token,
44                  lidar_name='LIDAR_TOP',
45                  file_path=sample_info['lidar_file'],
46                  frame=frame_number,
47                  is_infrastructure=False
48              )
```

```
49              self.sample_data_entries.append(entry)
50
51          # Process infrastructure LiDAR data (NEW - this thesis)
52          for lidar_name in self.infrastructure_lidars:
53              if lidar_name in sample_info['infrastructure_lidar_files']:
54                  entry = self._create_lidar_sample_data_entry(
55                      sample_token=sample_token,
56                      lidar_name=lidar_name,
57                      file_path=sample_info['infrastructure_lidar_files'][lidar_name],
58                      frame=frame_number,
59                      is_infrastructure=True  # Infrastructure sensor flag
60                  )
61                  self.sample_data_entries.append(entry)
62
63  def _create_camera_sample_data_entry(self, sample_token, camera_name,
64                                       file_path, frame, is_infrastructure):
65      """
66      Create sample_data entry for camera with proper metadata.
67
68      Links camera observation to parent sample, ego pose, and sensor
69      calibration while maintaining infrastructure sensor identification.
70      """
71      # Find corresponding sensor and calibrated_sensor tokens
72      sensor_token = self._find_sensor_token(camera_name)
73      calibrated_sensor_token = self._find_calibrated_sensor_token(camera_name)
74
75      # Find ego pose token for this frame
76      ego_pose_token = self._find_ego_pose_token(frame)
77
78      return {
79          'token': generate_unique_token(),
80          'sample_token': sample_token,
81          'ego_pose_token': ego_pose_token,
82          'calibrated_sensor_token': calibrated_sensor_token,
```

```
        'filename': file_path,  # Relative path from dataset root
        'fileformat': 'png',
        'width': self.camera_resolution[0],
        'height': self.camera_resolution[1],
        'timestamp': self._get_timestamp_for_frame(frame),
        'is_key_frame': True,

        # Optional infrastructure sensor metadata (extension field)
        'is_infrastructure_sensor': is_infrastructure,
    }
```

## C.7 User Interface

**Listing C.12:** UI Controller Implementation

```
def _create_sensor_section(self):
    """
    Create the sensor management section of the UI.

    Provides intuitive controls for sensor discovery, selection, and
    debugging within Isaac Sim's UI framework.
    """
    with ui.CollapsableFrame("Sensor Management", collapsed=False, height=0):
        with ui.VStack(spacing=3):
            # Control buttons for sensor discovery
            with ui.HStack(spacing=3):
                ui.Button(
                    "Scan for Sensors",
                    clicked_fn=self._on_scan_sensors,
                    tooltip="Detect all available sensors in the USD scene"
                )

                ui.Button(
                    "Debug Scan",
                    clicked_fn=self._on_debug_scan,
                    width=80,
                    tooltip="Print detailed sensor detection information to console"
```

```
                )

                ui.Button(
                    "Debug LiDAR",
                    clicked_fn=self._on_debug_lidar,
                    width=80,
                    tooltip="Display LiDAR sensor details and data statistics"
                )

            ui.Label("Available Sensors:")

            # Organized sensor lists with collapsible sections
            with ui.VStack():
                # Camera section
                with ui.CollapsableFrame("Cameras", collapsed=False, height=0):
                    self._camera_scroll = ui.ScrollingFrame(height=100)
                    with self._camera_scroll:
                        self._camera_list = ui.VStack(spacing=1)

                # LiDAR section
                with ui.CollapsableFrame("LiDARs", collapsed=False, height=0):
                    self._lidar_scroll = ui.ScrollingFrame(height=100)
                    with self._lidar_scroll:
                        self._lidar_list = ui.VStack(spacing=1)

def _populate_sensor_lists(self):
    """
    Populate sensor selection lists after scanning.

    Creates checkbox for each detected sensor with clear labeling to
    distinguish vehicle sensors from infrastructure sensors.
    """
    # Clear existing lists
    self._camera_list.clear()
    self._lidar_list.clear()

```

```
59      # Populate camera list with checkboxes
60      for camera_name in sorted(self._sensor_manager.
    available_cameras.keys()):
61          with ui.HStack():
62              # Checkbox for sensor activation
63              checkbox = ui.CheckBox()
64              checkbox.model.add_value_changed_fn(
65                  lambda model, name=camera_name:
66                      self._on_camera_selected(name, model.
    get_value_as_bool())
67              )
68
69              # Sensor name label with type indicator
70              sensor_type = "INFRA" if "TRAFFIC" in
    camera_name else "VEHICLE"
71              ui.Label(f"{camera_name} ({sensor_type})")
72
73      # Populate LiDAR list with checkboxes
74      for lidar_name in sorted(self._sensor_manager.
    available_lidars.keys()):
75          with ui.HStack():
76              checkbox = ui.CheckBox()
77              checkbox.model.add_value_changed_fn(
78                  lambda model, name=lidar_name:
79                      self._on_lidar_selected(name, model.
    get_value_as_bool())
80              )
81
82              sensor_type = "INFRA" if "TRAFFIC" in lidar_name
     else "VEHICLE"
83              ui.Label(f"{lidar_name} ({sensor_type})")
84
85  def _create_control_section(self):
86      """Create playback control section"""
87      with ui.CollapsableFrame("Timeline Controls", collapsed=
    False, height=0):
88          with ui.HStack(spacing=3):
89              ui.Button("Play", clicked_fn=self.
    _timeline_controller.play)
90              ui.Button("Pause", clicked_fn=self.
    _timeline_controller.pause)
91              ui.Button("Stop", clicked_fn=self.
    _timeline_controller.stop)
92
93  def _create_settings_section(self):
```

```
94      """Create data collection settings section"""
95      with ui.CollapsableFrame("Data Collection Settings",
    collapsed=False, height=0):
96          with ui.VStack(spacing=3):
97              # Enable nuScenes dataset generation
98              with ui.HStack():
99                  self._nuscenes_checkbox = ui.CheckBox()
100                 self._nuscenes_checkbox.model.
    add_value_changed_fn(
101                     self._on_nuscenes_toggle
102                 )
103                 ui.Label("Enable nuScenes Generation")
104
105             # Enable data saving to disk
106             with ui.HStack():
107                 self._save_checkbox = ui.CheckBox()
108                 self._save_checkbox.model.
    add_value_changed_fn(
109                     self._on_save_toggle
110                 )
111                 ui.Label("Save Sensor Data")
112
113             # Real-time status display
114             self._status_label = ui.Label("Status: Ready")
```

# Bibliography


[1] European Commission. *Commission Regulation (EU) 2019/2144 on type-approval requirements for motor vehicles and their trailers*. Nov. 2019 (cit. on pp. 1, 7).

[2] *Global status report on road safety 2018*. en. URL: https://www.who.int/publications/i/item/9789241565684 (cit. on p. 1).

[3] Nidhi Kalra and Susan M Paddock. «Driving to safety: How many miles of driving would it take to demonstrate autonomous vehicle reliability?» en. In: *Transportation Research Part A: Policy and Practice* 94 (Sept. 2016). DOI: 10.1016/j.tra.2016.09.010. URL: https://doi.org/10.1016/j.tra.2016.09.010 (cit. on pp. 1, 7).

[4] Keli Huang, Botian Shi, Xiang Li, Xin Li, Siyuan Huang, and Yikang Li. *Multi-modal Sensor Fusion for Auto Driving Perception: A Survey*. 2024. arXiv: 2202.02703 [cs.CV]. URL: https://arxiv.org/abs/2202.02703 (cit. on p. 1).

[5] Philip Koopman and Michael Wagner. «Challenges in Autonomous Vehicle Testing and Validation». en. In: *SAE International Journal of Transportation Safety* 04.1 (Apr. 2016), pp. 15–24. ISSN: 2327-5626, 2327-5634. DOI: 10.4271/2016-01-0128. URL: https://saemobilus.sae.org/articles/challenges-autonomous-vehicle-testing-validation-2016-01-0128 (cit. on pp. 1, 7).

[6] Alexey Dosovitskiy, German Ros, Felipe Codevilla, Antonio Lopez, and Vladlen Koltun. «CARLA: An Open Urban Driving Simulator». In: *Proceedings of the 1st Annual Conference on Robot Learning*. 2017, pp. 1–16 (cit. on pp. 1, 8, 14, 15, 19, 20, 34).

[7] F. Stella. «Virtual vehicle sensor's set-up and synchronous logging from CARLA Simulator». MA thesis. Turin, Italy: Politecnico di Torino, Apr. 2025 (cit. on pp. 1, 2, 11, 12, 14, 16–18, 20, 22, 31, 33, 55, 56, 58, 69).

[8] Holger Caesar et al. *nuScenes: A multimodal dataset for autonomous driving*. 2020. arXiv: 1903.11027 [cs.LG]. URL: https://arxiv.org/abs/1903.11027 (cit. on pp. 1, 8, 9, 16, 18, 33, 34).

[9] Manabu Tsukada, Takaharu Oi, Masahiro Kitazawa, and Hiroshi Esaki. «Networked Roadside Perception Units for Autonomous Driving». In: *Sensors* 20.18 (2020). ISSN: 1424-8220. DOI: 10.3390/s20185320. URL: https://www.mdpi.com/1424-8220/20/18/5320 (cit. on pp. 2, 5).

[10] Guangzhen Cui, Weili Zhang, Yanqiu Xiao, Lei Yao, and Zhanpeng Fang. «Cooperative Perception Technology of Autonomous Driving in the Internet of Vehicles Environment: A Review». In: *Sensors* 22.15 (2022). ISSN: 1424-8220. DOI: 10.3390/s22155535. URL: https://www.mdpi.com/1424-8220/22/15/5535 (cit. on pp. 2, 5, 11).

[11] Syed Adnan Yusuf, Arshad Khan, and Riad Souissi. «Vehicle-to-everything (V2X) in the autonomous vehicles domain – A technical review of communication, sensor, and AI technologies for road user safety». In: *Transportation Research Interdisciplinary Perspectives* 23 (2024), p. 100980. ISSN: 2590-1982. DOI: https://doi.org/10.1016/j.trip.2023.100980. URL: https://www.sciencedirect.com/science/article/pii/S2590198223002270 (cit. on pp. 2, 11).

[12] Zhihang Song et al. «Synthetic Datasets for Autonomous Driving: A Survey». In: *IEEE Transactions on Intelligent Vehicles* 9.1 (Jan. 2024), pp. 1847–1864. ISSN: 2379-8858. DOI: 10.1109/tiv.2023.3331024. URL: http://dx.doi.org/10.1109/TIV.2023.3331024 (cit. on pp. 2, 10).

[13] Xiaoyu Chen, Jiachen Hu, Chi Jin, Lihong Li, and Liwei Wang. *Understanding Domain Randomization for Sim-to-real Transfer*. 2022. arXiv: 2110.03239 [cs.LG]. URL: https://arxiv.org/abs/2110.03239 (cit. on pp. 2, 38).

[14] Chinmay Vilas Samak, Tanmay Vilas Samak, Bing Li, and Venkat Krovi. *Sim2Real Diffusion: Learning Cross-Domain Adaptive Representations for Transferable Autonomous Driving*. 2025. arXiv: 2507.00236 [cs.RO]. URL: https://arxiv.org/abs/2507.00236 (cit. on pp. 2, 38).

[15] Stephan R. Richter, Hassan Abu AlHaija, and Vladlen Koltun. *Enhancing Photorealism Enhancement*. 2021. arXiv: 2105.04619 [cs.CV]. URL: https://arxiv.org/abs/2105.04619 (cit. on pp. 2, 10).

[16] Lingyu Zhang, Li Wang, Lili Zhang, Xiao Zhang, and Dehui Sun. «An RSU Deployment Scheme for Vehicle-Infrastructure Cooperated Autonomous Driving». In: *Sustainability* 15.4 (2023). ISSN: 2071-1050. DOI: 10.3390/su15043847. URL: https://www.mdpi.com/2071-1050/15/4/3847 (cit. on pp. 3, 11).

[17] nuTonomy. *nuScenes Development Kit*. https://github.com/nutonomy/nuscenes-devkit. ROS2-based devkit with JSON metadata structure. 2024 (cit. on p. 3).

[18] Tao Huang, Jianan Liu, Xi Zhou, Dinh C. Nguyen, Mostafa Rahimi Azghadi, Yuxuan Xia, Qing-Long Han, and Sumei Sun. *Vehicle-to-Everything Cooperative Perception for Autonomous Driving*. May 2025. DOI: 10.1109/jproc.2025.3600903. URL: http://dx.doi.org/10.1109/JPROC.2025.3600903 (cit. on p. 3).

[19] Lei Chen and Cristofer Englund. «Cooperative Intersection Management: A Survey». In: *IEEE Transactions on Intelligent Transportation Systems* 17.2 (2016), pp. 570–586. DOI: 10.1109/TITS.2015.2471812 (cit. on pp. 3, 5).

[20] Pixar Animation Studios. *Introduction to USD - Universal Scene Description*. https://openusd.org/release/intro.html. Technical documentation on USD format with hierarchical Prims and composition. 2025 (cit. on pp. 3, 40).

[21] Pixar Animation Studios. *Universal Scene Description*. Available: https://graphics.pixar.com/usd. 2023 (cit. on pp. 3, 38).

[22] NVIDIA Corporation. *NVIDIA Omniverse Platform*. Available: https://www.nvidia.com/omniverse. 2024 (cit. on pp. 4, 38).

[23] NVIDIA. *Isaac Sim 5.0 Technical Specifications*. https://github.com/isaac-sim/IsaacLab. RTX-based photorealistic environments with 1000+ SimReady assets. 2025 (cit. on p. 4).

[24] Matt Rowe. *CARLA 0.9.16 Release*. Sept. 2025. URL: https://carla.org/2025/09/16/release-0.9.16/ (visited on 10/13/2025) (cit. on pp. 4, 8, 14, 39, 70).

[25] Haibao Yu et al. *DAIR-V2X: A Large-Scale Dataset for Vehicle-Infrastructure Cooperative 3D Object Detection*. 2022. arXiv: 2204.05575 [cs.CV]. URL: https://arxiv.org/abs/2204.05575 (cit. on pp. 5, 11).

[26] Adam Tonderski, Carl Lindström, Georg Hess, William Ljungbergh, Lennart Svensson, and Christoffer Petersson. *NeuRAD: Neural Rendering for Autonomous Driving*. 2024. arXiv: 2311.15260 [cs.CV]. URL: https://arxiv.org/abs/2311.15260 (cit. on pp. 6, 8).

[27] Euro NCAP. *Assessment Protocol - Safety Assist*. European New Car Assessment Programme. 2023 (cit. on p. 8).

[28] *ISO 26262-1:2018*. en. URL: https://www.iso.org/standard/68383.html (cit. on p. 8).

[29] *ISO 21448:2022*. en. URL: https://www.iso.org/standard/77490.html (cit. on p. 8).

[30] Guodong Rong et al. «LGSVL Simulator: A High Fidelity Simulator for Autonomous Driving». In: (2020). arXiv: 2005.03778 [cs.RO]. URL: https://arxiv.org/abs/2005.03778 (cit. on pp. 8, 12).

[31] Yurui Chen, Junge Zhang, Ziyang Xie, Wenye Li, Feihu Zhang, Jiachen Lu, and Li Zhang. *S-NeRF++: Autonomous Driving Simulation via Neural Reconstruction and Generation*. 2025. arXiv: 2402.02112 [cs.CV]. URL: https://arxiv.org/abs/2402.02112 (cit. on p. 8).

[32] Jiang Yue, Weisong Wen, Jing Han, and Li-Ta Hsu. *LiDAR Data Enrichment Using Deep Learning Based on High-Resolution Image: An Approach to Achieve High-Performance LiDAR SLAM Using Low-cost LiDAR*. 2020. arXiv: 2008.03694 [cs.CV]. URL: https://arxiv.org/abs/2008.03694 (cit. on p. 8).

[33] Yan Wang, Wei-Lun Chao, Divyansh Garg, Bharath Hariharan, Mark Campbell, and Kilian Q. Weinberger. *Pseudo-LiDAR from Visual Depth Estimation: Bridging the Gap in 3D Object Detection for Autonomous Driving*. 2020. arXiv: 1812.07179 [cs.CV]. URL: https://arxiv.org/abs/1812.07179 (cit. on p. 8).

[34] Matt Pharr, Wenzel Jakob, and Greg Humphreys. *Physically Based Rendering: From Theory to Implementation*. 4th. MIT Press, 2023 (cit. on p. 10).

[35] NVIDIA Corporation. *RTX Real-Time Ray Tracing*. NVIDIA Developer Documentation. 2023 (cit. on p. 10).

[36] Aayush Prakash, Shaad Boochoon, Mark Brophy, David Acuna, Eric Cameracci, Gavriel State, Omer Shapira, and Stan Birchfield. *Structured Domain Randomization: Bridging the Reality Gap by Context-Aware Synthetic Data*. 2020. arXiv: 1810.10093 [cs.CV]. URL: https://arxiv.org/abs/1810.10093 (cit. on p. 10).

[37] Josh Tobin, Rachel Fong, Alex Ray, Jonas Schneider, Wojciech Zaremba, and Pieter Abbeel. *Domain Randomization for Transferring Deep Neural Networks from Simulation to the Real World*. 2017. arXiv: 1703.06907 [cs.RO]. URL: https://arxiv.org/abs/1703.06907 (cit. on p. 10).

[38] Judy Hoffman, Eric Tzeng, Taesung Park, Jun-Yan Zhu, Phillip Isola, Kate Saenko, Alexei A. Efros, and Trevor Darrell. *CyCADA: Cycle-Consistent Adversarial Domain Adaptation*. 2017. arXiv: 1711.03213 [cs.CV]. URL: https://arxiv.org/abs/1711.03213 (cit. on p. 10).

[39] 3GPP. *3GPP Release 14: LTE-based V2X Services*. Tech. rep. 3rd Generation Partnership Project, 2017 (cit. on p. 11).

[40] Walter Zimmer, Gerhard Arya Wardana, Suren Sritharan, Xingcheng Zhou, Rui Song, and Alois C. Knoll. *TUMTraf V2X Cooperative Perception Dataset*. 2024. arXiv: 2403.01316 [cs.CV]. URL: https://arxiv.org/abs/2403.01316 (cit. on p. 11).