Lab 0: Access Carla

EE267 Introduction to Autonomous Driving Systems

Student 1 ID: 862161452
Student 1 Name: Kamren James

Student 2 ID:862620026
Student 2 Name: Shashwat Shah

Objective

This lab prepares the simulation environment for the following labs and the final project. The goal is to set up a working environment, get familiar with basic APIs, understand the inner workings of the simulator, and program it to let it simulate scenarios serving your purpose.

Part 1: Running *Carla Server*

To establish the necessary simulation environment, we successfully launched the CARLA server, focusing on a headless setup to optimize resource usage.

We initialized the server by navigating to the CARLA root directory and running the following command:

Bash

- bash CarlaUE4.sh -RenderOffScreen

The critical flag we employed here was:

- `-RenderOffScreen`: This flag is essential because it forces the Unreal Engine to render graphics directly into a memory buffer rather than to a physical display. This allows the server to run reliably on a machine without a dedicated graphics interface, which is ideal for a remote or computational environment.

Once executed, the server environment initializes the Unreal Engine and begins listening for client connections. The successful launch, confirming the server's readiness, is documented in the attached screenshot (Figure X).

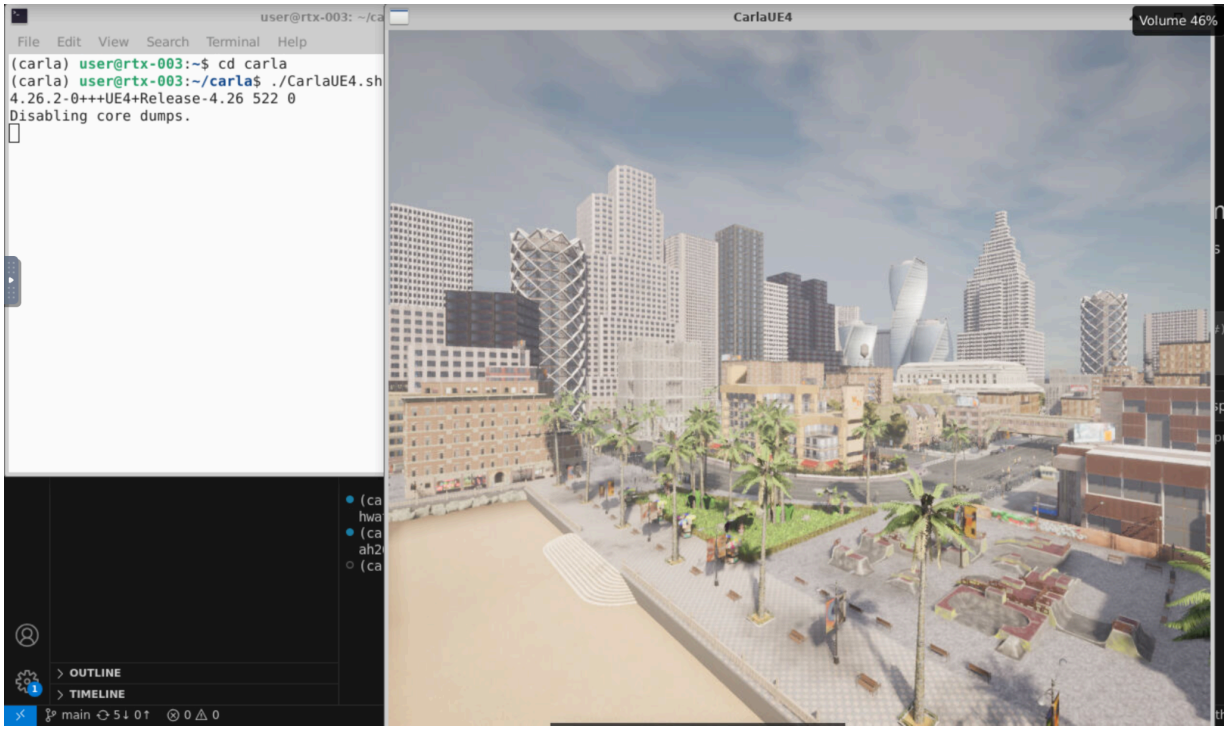

user@rtx-003: ~/ca
File Edit View Search Terminal Help
(carla) user@rtx-003:~$ cd carla
(carla) user@rtx-003:~/carla$ ./CarlaUE4.sh
4.26.2-0+++UE4+Release-4.26 522 0
Disabling core dumps.
OUTLINE
TIMELINE
main
CarlaUE4
Volume 46%

Fig:1

Part 2: Drive a Car using *Carla Client*

With the CARLA server running, the next stage involved setting up the client environment and executing basic control scripts to interact with the simulation.

Client Environment Setup

We established a dedicated and isolated environment for the client-side code to manage dependencies effectively.

CARLA Client Installation: The official CARLA client library was installed using `pip` from the Python Package Index (PyPI), providing the necessary API bindings for communication with the running server:
Bash
pip install carla

1. **Repository Setup:** The project assignment repository was cloned locally, containing the necessary client scripts (`manual_control.py` and `automatic_control.py`).

Initial Driving Tests

We performed two initial tests to confirm the client's ability to connect and command vehicles within the simulation.

1. Manual Control Test

The client was first launched using the `manual_control.py` script. (Note: For environments accessing shared GPU resources, the `vglrun` prefix was used for rendering.)

python manual_control.py

vglrun python manual_control.py

Fig:2,3,4 Display's Manual control mode

This successfully opened a Pygame window, displaying the rendered environment and sensor feeds. We used the keyboard to directly interact with the vehicle, testing basic throttle, braking, and steering inputs, as well as cycling through different sensor views (e.g., camera, LiDAR).

CODE 1:

2. Automatic Control Test

Following the manual test, we executed the `automatic_control.py` script to observe the vehicle operating under a predefined or simple algorithmic control system:

```
python automatic_control.py

# OR (if required)

vglrun python automatic_control.py
```

This script demonstrated the use of the CARLA API to send continuous, automated commands to the vehicle, allowing it to navigate autonomously within the simulated environment. A screenshot capturing the view during this automated operation has been attached to confirm the successful client-server communication and rendering

pygame window
Server: 21 FPS
Client: 20 FPS
Vehicle: Volkswagen T2 2021
Map: Town10HD_Opt
Simulation time: 0:00:21
Speed: 21 km/h
Heading: -90° W
Location: (-103.0, -14.7)
GNSS: (0.000141, -0.000925)
Height: -0 m
Throttle:
Steer:
Brake:
Reverse:
Hand brake:
Manual:
Gear: 1
Collision:
Number of vehicles: 1

pygame window
Server: 20 FPS
Client: 16 FPS
Vehicle: Volkswagen T2 2021
Map: Town10HD_Opt
Simulation time: 0:00:32
Speed: 0 km/h
Heading: -4° NW
Location: (-65.6, -58.1)
GNSS: (0.000522, -0.000580)
Height: -0 m
Throttle:
Steer:
Brake:
Reverse:
Hand brake:
Manual:
Gear: 1
Collision:
Number of vehicles: 1

FIg:5,6

Part 3: Using Carla API

To create a realistic environment for testing autonomous driving agents, we utilized the **Traffic Manager (TM)** module to procedurally generate and control a population of Non-Player Character (NPC) vehicles and pedestrians.

Traffic Manager Initialization

The first step in generating autonomous traffic was to initialize the Traffic Manager, which acts as the control hub for all self-driving actors.

1. **Client Connection:** We ensured a connection to the CARLA server was maintained, and the main `carla.World` object was retrieved.

Traffic Manager Retrieval: The Traffic Manager instance was obtained from the client. By default, the TM is an independent module that manages the autopilot behavior of actors.
Python
traffic_manager = client.get_trafficmanager()

2.
3. **Synchronization (Optional but Recommended):** For precise control over the simulation state—especially when capturing sensor data—the TM was configured to run in synchronous mode, matching the server's tick rate.

Spawning Vehicles and Pedestrians (Background Traffic)

We then proceeded to populate the world with a high volume of random traffic, including both vehicles and pedestrians.

1. **Blueprint and Spawn Point Retrieval:** We collected a library of available vehicle and pedestrian blueprints, along with a list of recommended spawn points provided by the map.
2. **Vehicle Spawning:** We iteratively selected a random vehicle blueprint and a random spawn point to create a specific number of background vehicles.
3. **Enabling Autopilot:** Crucially, each spawned vehicle was then handed over to the Traffic Manager by enabling its autopilot feature, which delegates control to the TM's internal path planning and collision avoidance logic.
   Python
   # Simplified Python API snippet
   # world.spawn_actor(vehicle_bp, spawn_point)
   # vehicle.set_autopilot(True)

4. **Pedestrian Spawning and Control:** Similarly, we spawned a number of pedestrian actors. Pedestrian movement requires a separate setup involving the `carla.WalkerControl` and a dedicated `carla.WalkerAIController` to manage

their pathing and animation. The AI controller was instructed to start walking to give the appearance of an active, bustling environment.

The successful implementation of this script resulted in a bustling simulation with multiple vehicles and pedestrians navigating the road network autonomously, as confirmed in the attached screenshots. This populated environment serves as the standard scenario for all subsequent testing.

CarlaUE4

CarlaUE4

Fig:7,8,9

Part 4: Bounding Boxes

To generate ground-truth data suitable for training and evaluating object detection models, we implemented the capability to render 3D bounding boxes around all non-player vehicles and export this data in a standard format. This required developing custom geometric projection logic based on the camera's intrinsic parameters.

Geometric Transformation Implementation

The primary challenge was translating the 3D world coordinates of objects into 2D pixel coordinates visible on the sensor image.

1. **Projection Matrix Calculation:** We implemented the `build_projection_matrix` function, which uses the camera's field of view (FoV), image width, and image height to construct a standard **3x3 projection matrix (K)**. This matrix maps points from the 3D camera coordinate system to the 2D pixel space.
2. **3D to 2D Point Projection:** The `get_image_point` function was created to perform the full geometric transformation:
    - It takes a 3D world point and the camera's transformation matrix.
    - It transforms the 3D world point into the camera's local coordinate system (Camera View).
    - It then applies the calculated projection matrix (K) to get the final 2D pixel coordinates.
    - This function includes crucial filtering to ensure only points *in front* of the camera (positive Z-coordinate) are considered valid.

Bounding Box Implementation

The logic for generating the boxes and their projection was integrated into the client's game loop.

1. **Camera Setup and Matrix Calculation:** We set up the required camera specifications and calculated the static projection matrix K for the entire session.
2. **Vehicle Spawning and Annotation:** We augmented the background traffic spawning process (from Part 3) to ensure a robust set of vehicles was present for annotation.
3. **Rendering Loop Modification:** We modified the client's game loop to perform the following steps on every tick:
    - **Retrieve Actors:** Get a list of all active vehicle actors in the world.
    - **Generate Corners:** For each vehicle, we calculated the eight 3D corner points of its bounding box using its extent (dimensions) and current location/rotation.
    - **Projection and Drawing:** We applied the `get_image_point` function to project all eight 3D corner points onto the 2D image plane. These projected points were then connected using Pygame's drawing utilities to render a 3D box on the sensor display.
4. **Filtering for Accuracy:** To avoid noise and ensure realistic data, we implemented a filtering mechanism:
    - **Distance Check:** We filtered out vehicles exceeding a certain distance (e.g., 50 meters) from the ego-vehicle/camera, as distant boxes offer limited training value.
    - **Field of View (FoV) Check:** We used the projection results to ensure at least a minimum number of box corners were visible within the camera's viewport boundaries before drawing the box.

Exporting Bounding Boxes (Pascal VOC Format)

To make the generated ground-truth data usable for external machine learning pipelines, we implemented an export functionality.

1. **Bounding Box Calculation:** After projecting the 3D corners to 2D, we calculated the tightly fitted **2D axis-aligned bounding box (AABB)** that enclosed all visible projected points. This yields the standard `(xmin, ymin, xmax, ymax)` coordinates.
2. **Pascal VOC Export:** We created a function to write the image file and an accompanying XML annotation file for each frame. This annotation file adheres to the **Pascal VOC format**, which includes:
    - Image filename, width, and height.
    - For each detected object (vehicle): its class (e.g., 'car') and the calculated 2D bounding box coordinates.

The accompanying screenshots showcase the successful implementation, displaying the sensor feed with the dynamically rendered 3D bounding boxes correctly tracking the vehicles in the simulated environment.

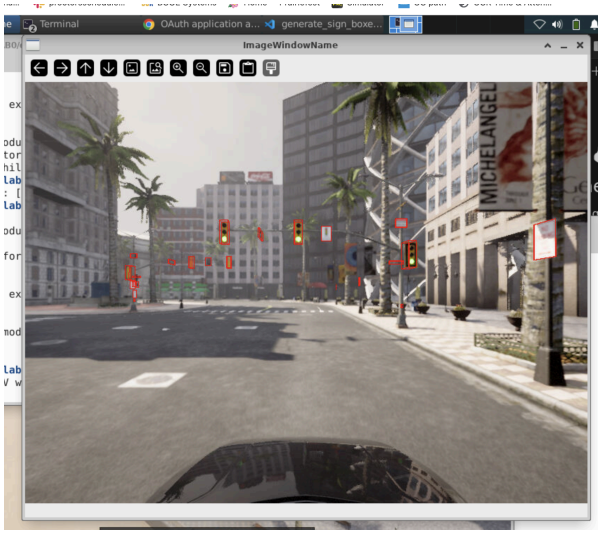

Terminal
OAuth application a...
generate_sign_boxe...
ImageWindowName
MICHELANGEL

Fig:10,11,12,13,14

Part 5: Instance Segmentation

To acquire pixel-level labels that distinguish individual objects, we implemented the **Instance Segmentation** camera sensor, a high-fidelity ground-truth sensor provided by the CARLA simulator.

Camera Setup and Scene Population

We began by configuring and deploying the specialized camera and ensuring the scene was sufficiently populated for testing.

1. **Camera Blueprint and Spawning:** We retrieved the dedicated blueprint, `sensor.camera.instance_segmentation`, and set its resolution and field of view to match our bounding box camera (Part 4). We then chose a suitable static location and

rotation, and saved its `carla.Transform` to ensure the same viewpoint could be used across multiple frames and for analysis.

2. **Scene Population:** To stress-test the instance segmentation, we retrieved the list of available spawn points on the map. We then implemented a loop to selectively spawn a large number of vehicles only within an **80-meter radius** of the fixed camera location. This guarantees that the vehicles being annotated are visible and relevant to the sensor. *A screenshot of the populated scene from the spectator view, confirming the density of traffic, is included (Figure α).*

Generating the Instance Segmentation Image

The data acquisition was managed using the synchronous simulation mode established earlier.

1. **Data Queue Setup:** We instantiated a Python `queue` object to receive the image data asynchronously from the sensor.
2. **Listening and Ticking:** The camera's `listen()` method was implemented to feed its output directly into the queue. We then executed a single `world.tick()` to force the simulation to advance one frame, generating a new image.
3. **Image Retrieval and Export:** The instance segmentation image was retrieved from the queue and saved to disk as `'instance_segmentation.png'`.

Analyzing the Instance Segmentation Output

The saved image contains the ground-truth information encoded across its color channels.

Instance ID Encoding:

The unique instance ID for each object (e.g., each individual car, each pedestrian) is encoded in the image's BGR color channels.

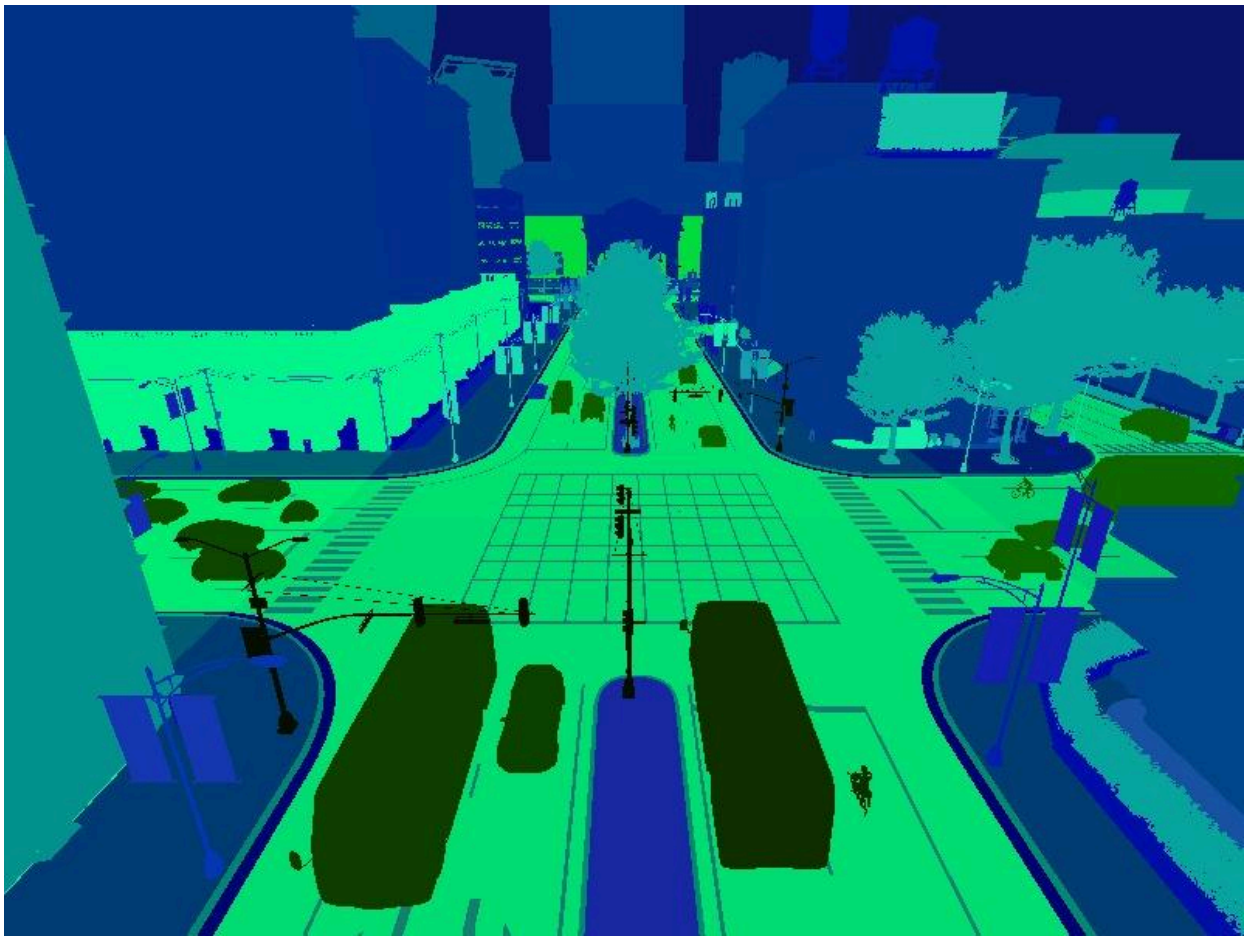
- The **Red (R)** channel contains the standard **Semantic ID** (e.g., all pixels belonging to the *class* 'Vehicle' have the same R-value).
- The **Green (G)** and **Blue (B)** channels encode the **Instance ID**. The full ID is reconstructed by treating the G and B channels as the higher and lower bytes of a two-byte integer, using the formula:
  $$\text{InstanceID} = G \times 256 + B$$
  This encoding allows the simulator to assign a unique ID to every single actor (up to 65,536 instances) in the scene, regardless of its semantic class.

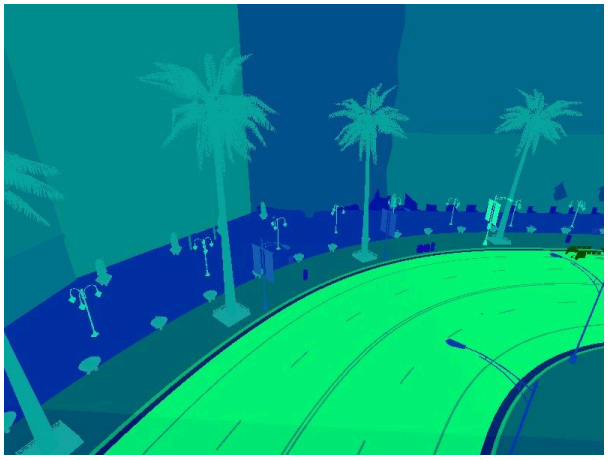
Potential Applications in Autonomous Driving

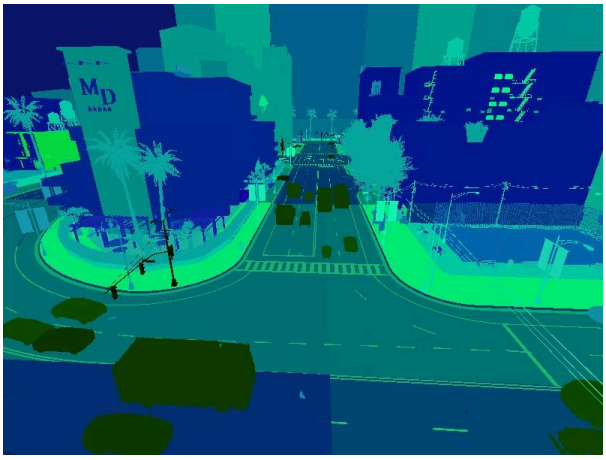
Instance segmentation is crucial for autonomous driving perception systems. It enables **per-instance tracking and prediction**, which is vital for safety-critical tasks. For example, it allows the driving agent to:

1. **Differentiate Obstacles:** Separate two adjacent or overlapping vehicles, allowing the prediction system to track each car's unique trajectory, speed, and intent.
2. **Precise Planning:** Accurately measure the distance to the *closest point* of a specific vehicle (e.g., the car directly in front) versus a vehicle in a neighboring lane.

The screenshot of the instance segmentation output (Figure β) visually demonstrates how pixels belonging to two different, yet identically classified, vehicles are rendered with distinct encoded colors.







Fig;15,16,17