Lab 0: Access Carla

EE267 - Introduction to Autonomous Driving Systems

Bhagya Patel - 862548221
Parth Kalathiya - 862488463

Part 01 - Running Carla Server

We start the Carla simulator by going into the `~/carla` directory and using the following command:

`./CarlaUE4.sh` - for a simulator with a GUI and `./CarlaUE4.sh -RenderOffScreen` for without GUI. This starts the simulation in spectator mode. One can navigate within the simulator by pressing keyboard keys like WASD to move forward, right, backward, and left, respectively. The keys Q and E help the spectator go up and down. The documentation also suggests dragging the screen with the left mouse click to change the direction of the camera, but as we control the simulation on BCOE systems via NoVNC, the WebSocket connection does not handle mouse clicks properly.

Part 02 - Drive a car using the Carla client

We create a conda environment named `trojan` and set up Python 3.9 in it. We installed the following packages:

carla==0.9.15
filelock==3.13.1
fsspec==2024.6.1
Jinja2==3.1.4
MarkupSafe==2.1.5
mpmath==1.3.0
networkx==3.2.1
numpy==2.0.2
pillow==11.0.0
pygame==2.6.1
shapely==2.0.7
sympy==1.13.3
typing_extensions==4.12.2
opencv-python==4.12.0.88
pascal-voc-writer==0.1.4

And then cloned our project repository inside the `carla` folder. To test the simulator, we first started the simulator in the background, as mentioned in [Part 01 - Running Carla Server](). In another terminal tab, we started with the `manual_control.py` file and captured different scenes as shown below.

Figure 2.1: RGB Camera

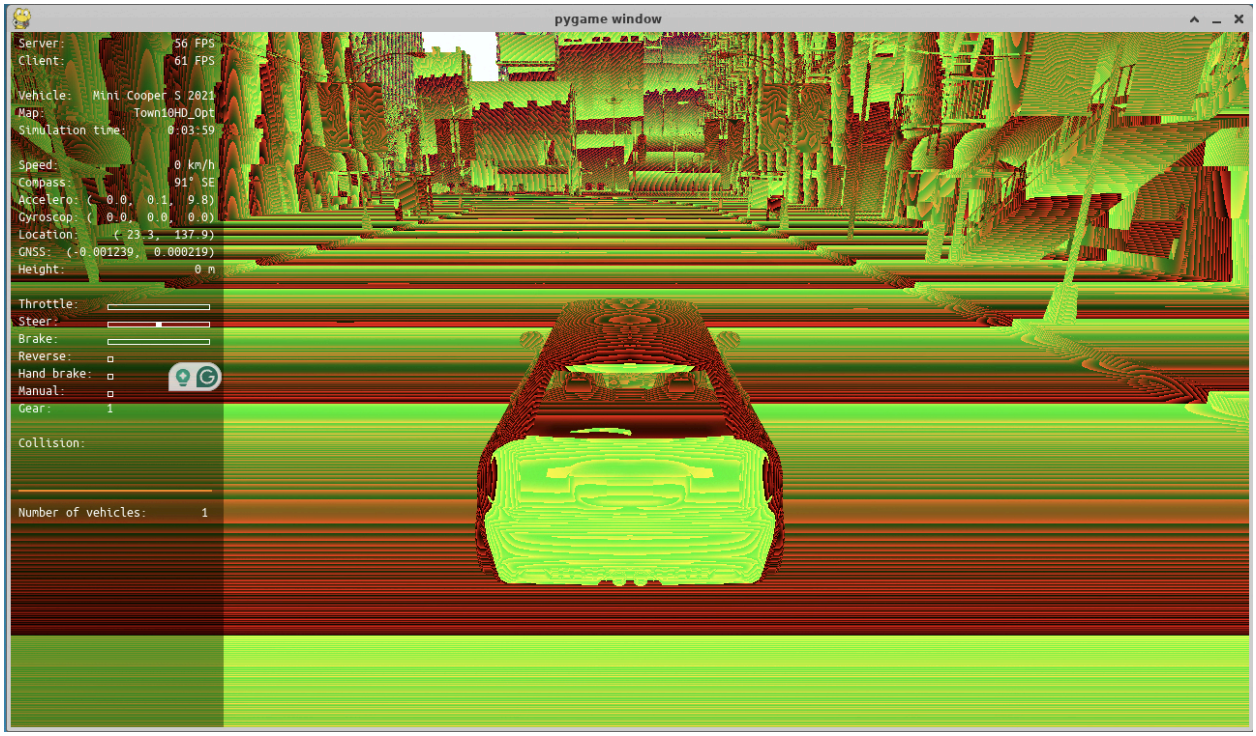

Figure 2.2: Camera Depth (Raw)

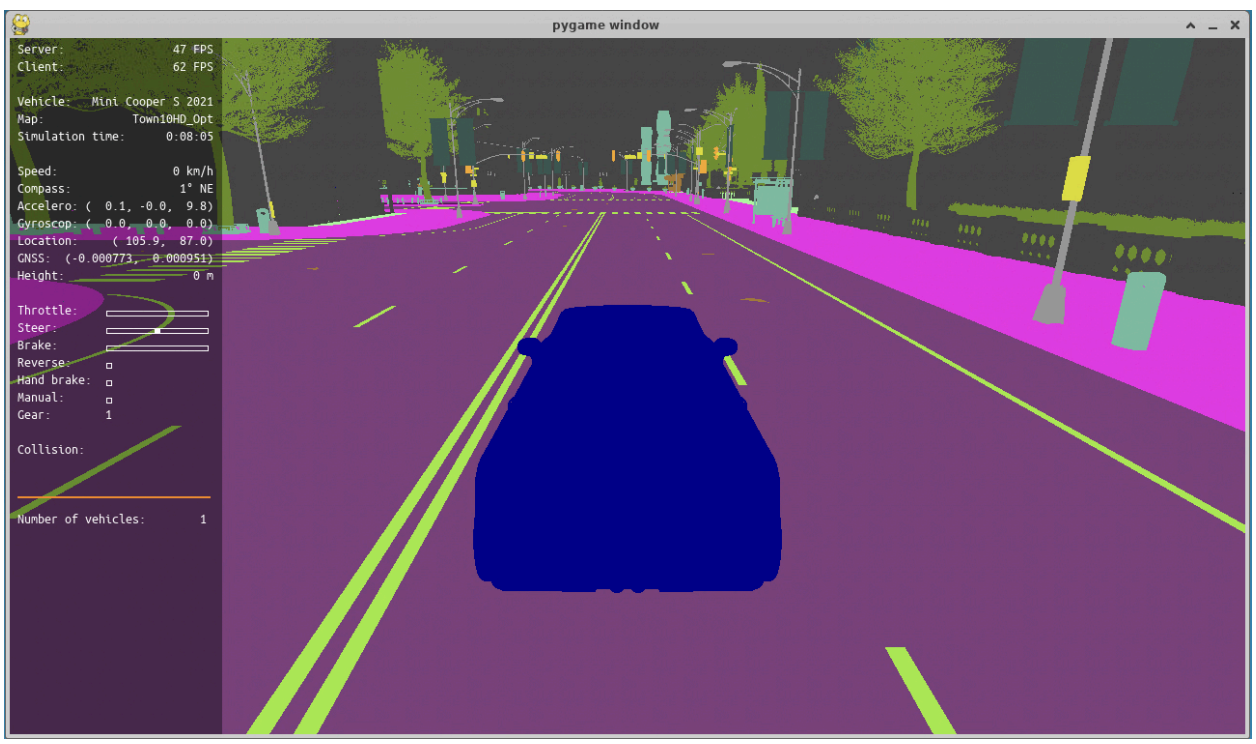

Figure 2.3: Camera Semantic Segmentation (CityScapes Palette)

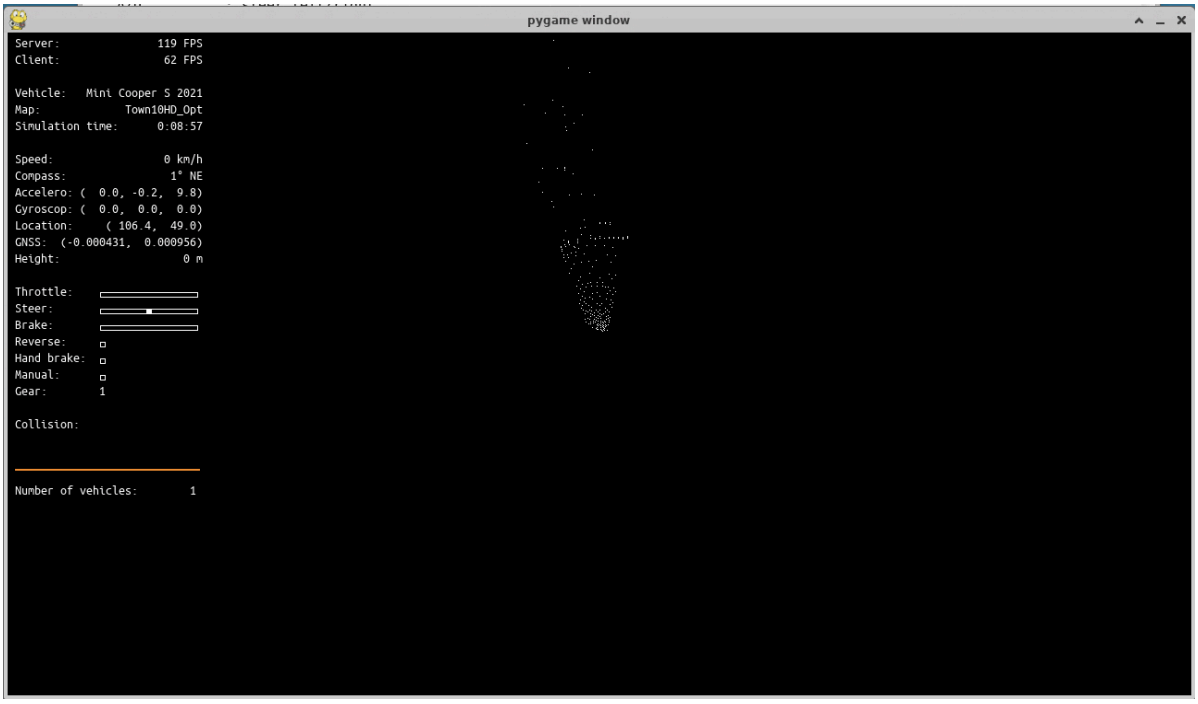

Figure 2.4: LIDAR

Running the `automatic_control.py` file, we observed that the vehicle moved autonomously, navigating the environment's roads properly. It stopped at stop signs, it stopped at traffic lights, switched lanes smoothly, and reached its destination.

Figure 2.5: Automatic Control

Part 03 - Using the Carla API

This part introduces writing some code to interact with the Carla environment. Following the steps in part 1, we can navigate the environment with our keyboard, and we find the environment empty at first.

Figure 3.1: Empty environment (without traffic)

We use the Carla API to add some traffic to this environment with the traffic manager. Following the tutorial provided in the instructions, we wrote the `traffic_manager.py` file. We first set the world to be synchronous and set its random seed to 0. We obtain the traffic spawn points with the API and then spawn random vehicles with their blueprints. We also spawn pedestrians (background traffic) on the road, as there were no spawn points present off the road. We gather walker blueprints and add them to the spawn list. The vehicles are set on autopilot, but the walkers remain stationary. We get the following result.

Figure 3.2: Environment with traffic (Vehicles and Pedestrians)

Part 04 - Bounding Boxes

1. Simulator Setup:

Figure 4.1: Camera Setup

The simulator setup is straightforward: We start the CARLA server locally and connect to it from your Python script. First, we import the CARLA library and create a client that connects to `localhost:2000`. Then we load the world instance and its blueprint library, which contains definitions for vehicles and sensors. We pick a spawn point at random and place our ego vehicle there. Next, we attach an RGB camera sensor about two meters above the vehicle and configure it to run in synchronous mode so that each simulation step matches our script's frame rate. Finally, we set up a simple queue to receive camera images for processing. This setup gives us a running CARLA environment with a camera feed ready for further tasks like rendering bounding boxes.

2. Implement geometric transformations:

Implementing geometric transformations simply means converting points from the 3D world into the 2D camera image so we can draw boxes around vehicles. We do this with two helper functions:

1. build_projection_matrix(width, height, fov)
    - This function computes the camera's intrinsic matrix (called K) from its field of view and image size. Internally, it figures out how "wide" the camera sees and translates that into focal lengths along the x- and y-axes.
2. get_image_point(point_3d, K, world_to_camera)
    - This function takes a single 3D point (for example, one corner of a vehicle's bounding box), applies the world_to_camera transformation (which moves points from the CARLA world into the camera's coordinate system), and then projects that into pixel coordinates using K. The result is a 2D (x, y) location in the camera image.

By calling `build_projection_matrix` once at setup, then repeatedly using `get_image_point` for each vertex of every vehicle's 3D box, we get the 2D corners needed to draw rectangles on our camera feed.

3. Implement vehicle bounding boxes:

First, we spawn 1000 additional vehicles by selecting blueprints from CARLA's library and placing NPC cars at random map locations, all set to autopilot so they move independently. Second, we modify the game loop to process each simulation tick: capture the latest camera image, then iterate over every actor labeled as a vehicle and retrieve its 3D bounding box corners in world coordinates.

For projecting and drawing, for each corner of a vehicle's 3D box, we call a helper function to convert its 3D world position into 2D pixel coordinates on the camera image. By finding the minimum and maximum x and y values among those projected points, we form a tight 2D rectangle around the vehicle. Finally, to keep the display focused and efficient, we filter vehicles by both distance and field of view: we skip any car more than a preset range (for example, 60 meters away) and any car behind the camera (determined via a simple dot-product test). The result is a clean, real-time overlay of bounding boxes around only the nearby, visible vehicles in our CARLA simulation.

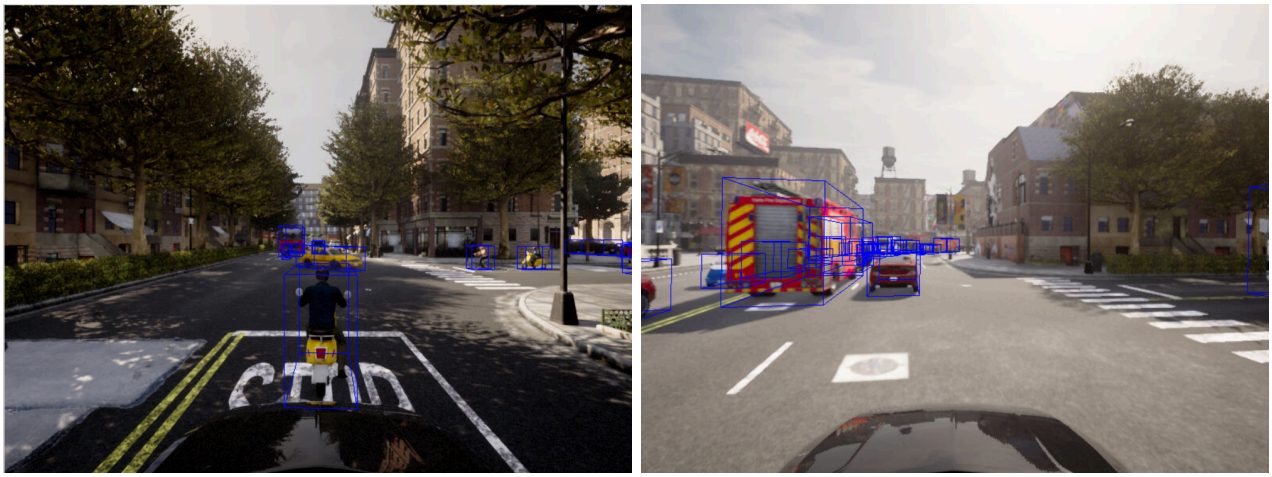

Figure 4.2: 3D Bounding Boxes

4. Export bounding boxes:

- Pascal VOC format implementation with XML annotations and PNG image saving.
- Pascal VOC format only stores 2D bounding box data, so we need to convert 2D bounding box to 3D bonding box to store annotated data.

Figure 4.3: 2D Bounding Boxes

The implementation leverages the `pascal_voc_writer` library to generate and save Pascal VOC compliant XML annotation files which, along with the captured images, can be found in the repository's `output` folder for use in training and evaluation.

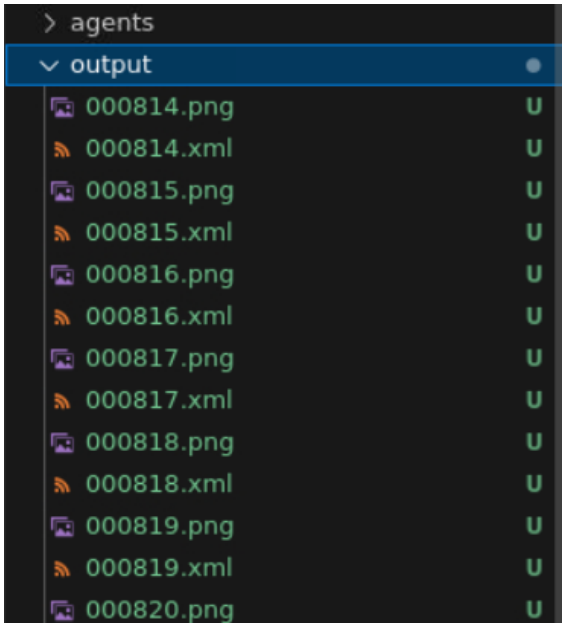

Figure 4.4 Pascal VOC format file structure

Part 05 - Instance Segmentation

We first set the instance segmentation camera at the same location as in the tutorial. After setup, we spawned vehicles in spawn points within an 80-meter radius of the camera.

Figure 5.1: Spawned traffic spectator view

Then, we set up a queue to capture images and added a listener to the camera. After one tick, the camera captured an image, which was pushed to the queue. We obtain that image from the queue and save it as `instance_segmentation.png`

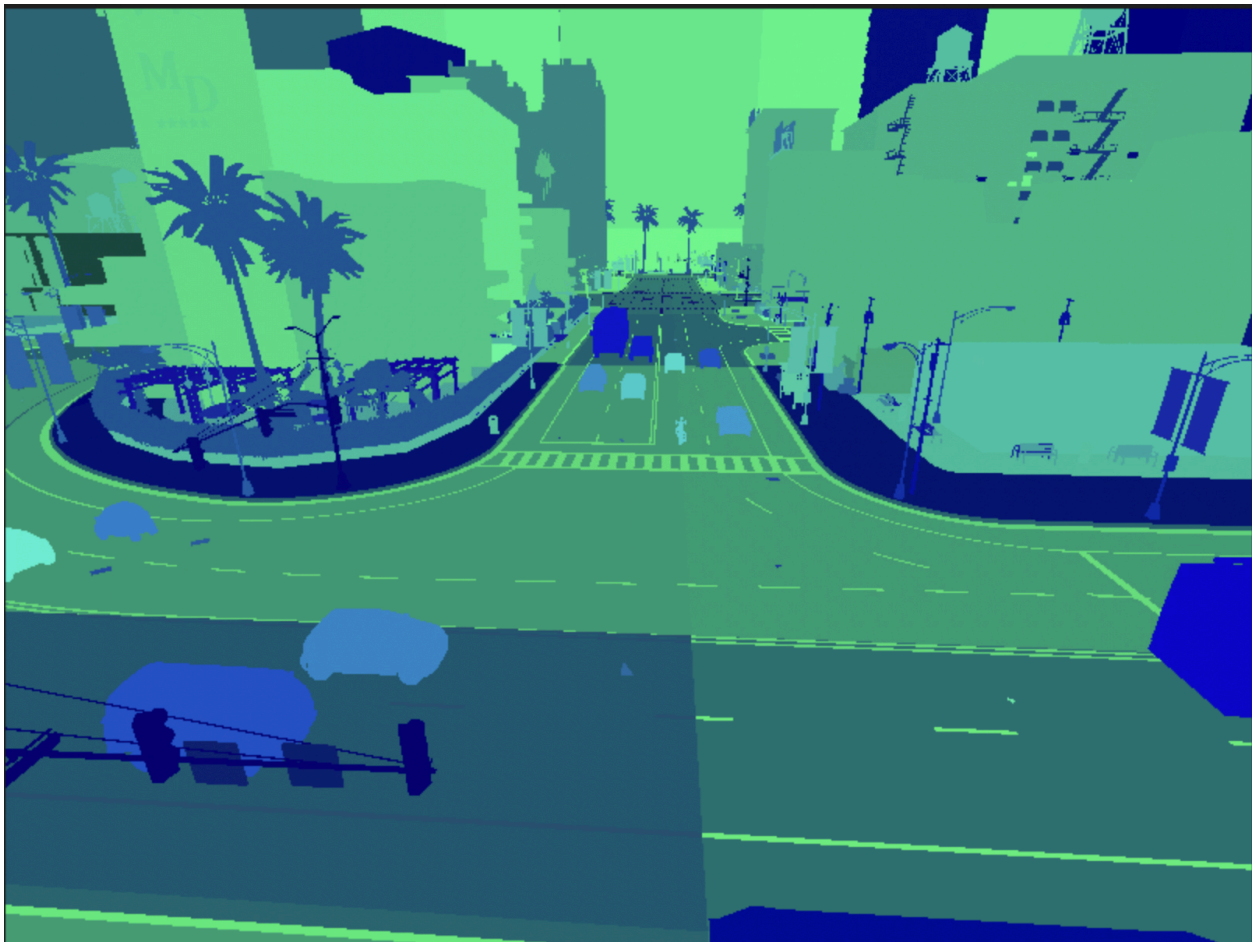

Figure 5.2: “instance_segmentation.png”

How does instance segmentation differ from semantic segmentation? and its potential applications in autonomous driving.

Semantic segmentation classifies every pixel in an image into categories like road, car, or pedestrian, treating all objects of the same class as one. In contrast, instance segmentation not only identifies object categories but also distinguishes between individual instances, such as separating each car or pedestrian. In autonomous driving, semantic segmentation helps identify drivable areas and scene context, while instance segmentation enables tracking and predicting the movement of specific objects for safer navigation and decision-making.

In the segmented image, each object - such as vehicles, roads, buildings, and pedestrians is assigned a distinct color mask. This highlights that the model has correctly separated individual instances within the same class. Such segmentation enables an autonomous vehicle to differentiate and track individual objects, estimate distances, and plan paths safely.