Installation of Carla :

For Installation of Carla it is highly advisable to follow the Latest Document for Windows or Linux Build.

Windows: https://carla.readthedocs.io/en/latest/build_windows/
Linux: https://carla.readthedocs.io/en/latest/build_linux/

We need to make sure we are following the Same set of Commands for the Specific version of the OS installed. During Windows build if we have installed Windows 11/10 then we have to install the SPecific SDK when installing Visual Studio Code.

This process may take around 10 hours depending on the Internet Speed and the System speed.

Roadrunner

Roadrunner is a tool by Mathworks for Installing of Virtual Maps which can be imported to simulators like Carla and More.

Using the interactive editor RoadRunner, you may create 3D environments for testing and modeling automated driving systems. By designing road signs and markings unique to a certain region, you may personalize roadway scenery. Along with vegetation, buildings, and other 3D models, you may add signs, signals, guardrails, and damage to the road. At intersections, RoadRunner offers options for adjusting traffic signal timing, phases, and vehicle routes.

Roadrunner installation requires first purchasing a license from their website using an official Mathworks account that is logged in.

We need to follow the Updated instructions for Installation of Roadrunner on this link.

After successfully installing Roadrunner we will follow this video for getting started with Roadrunner.

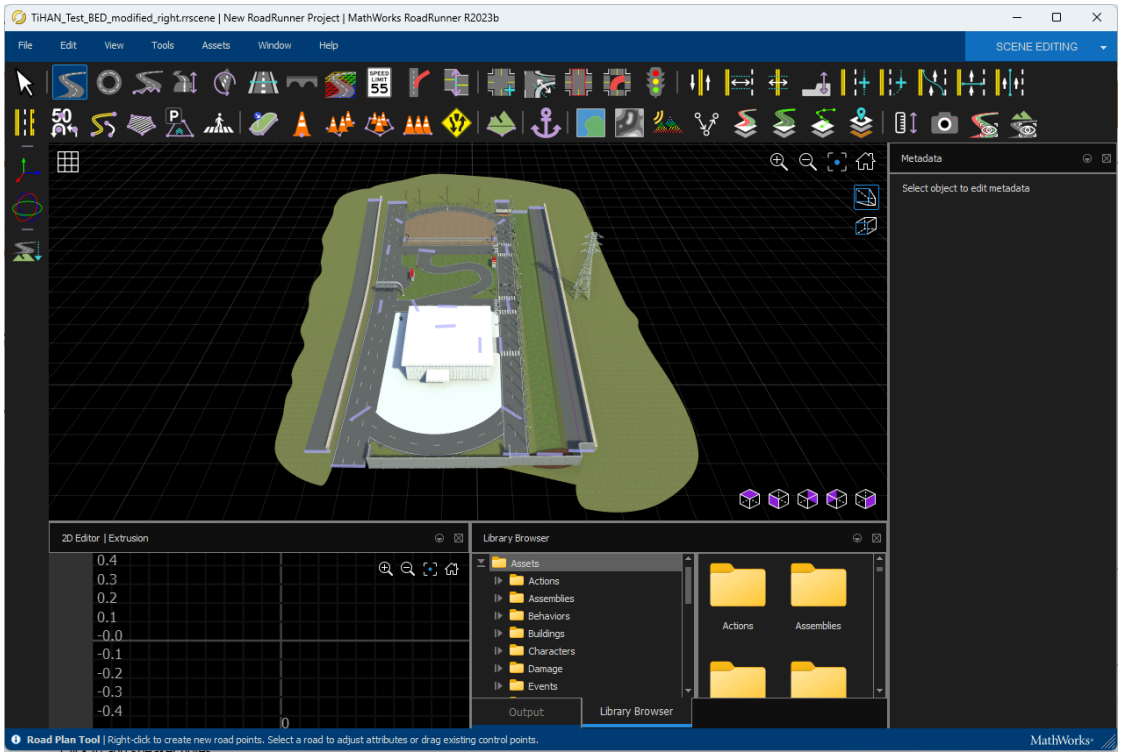

Figure 1: Overview of Mathworks Roadrunner Tool

Creation of MAP

1. Roads Creation

We use the Road Tool in Roadrunner to create Roads and the Tutorial for the Same can be found here.

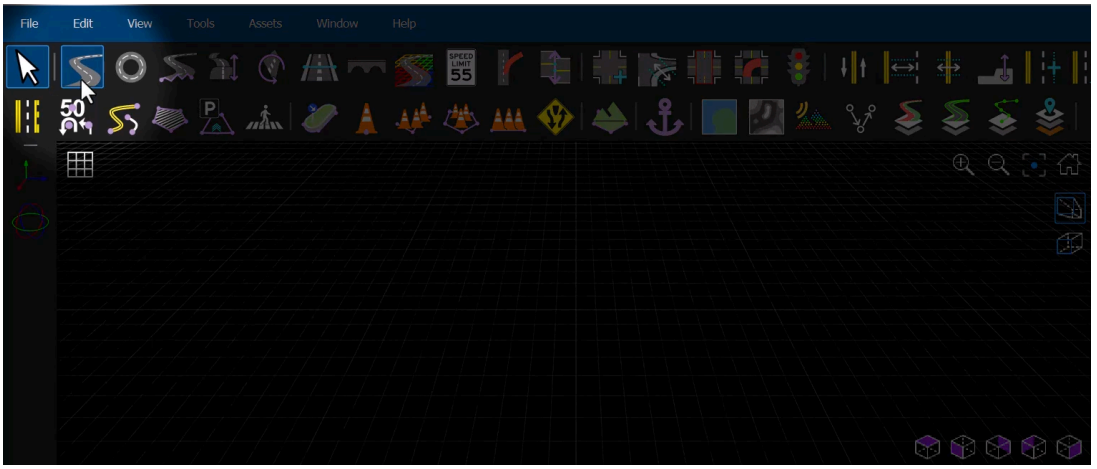

Figure 2: Pointing out RoadRunner Road Tool

We need to click on the tool and then draw a road by clicking on the editor. For detailed explanation of the same follow this video Mathworks: Tutorial for Creation of Roads .

2. Creating Terrains and Adding Props

The Surface Tool of the Mathworks is used to create terrains such as mud, grass, and potholes.

Initially, we sketch the outline of our own form and then fill it in with a substance such as gravel, grass, or sand.

Check out the video below to get a proper understanding of how to use the Surface Tool. Even though they demonstrate how to make a 3D prop in the video below, we used the same method to create our testbed.

Video URL : Mathworks: Tutorial for Terrains and 3D Props

3. Customizing of Lanes

Several Road tools from Mathworks are used to create different kinds of lanes, such as two-lane and four-lane pathways.

We also have the ability to alter the road's altitude. Its Curvature can also be changed. The video below delves into detail on each of these.

Video URL: Mathworks: Tutorial for Road Customization

We follow another video for creation of Junctions, here we use the Junction Tool for connecting 2 intersections of Roads.

Video URL: Mathworks: Tutorial for Creation of Junctions

4. Exporting of Carla Maps

For exporting the map created to Carla or any other Simulator we use the following documentation. Here they clearly explain the steps to follow for the Same.

Please note that there are no errors when exporting a map. We need to look for the error in their helpage to fix the error.

Carla Software

CARLA is an open-source simulation platform designed for developing, training, and validating autonomous driving systems. It offers a range of digital assets such as urban layouts, buildings, and vehicles, along with customizable sensor suites and environmental conditions. With full control over static and dynamic actors, CARLA provides a versatile environment for testing and refining autonomous driving algorithms.

1. Importing Custom Map from Roadrunner

After we successfully created a Map in Roadrunner then we need to install Few Plugins in Carla so that we can import the Map.
The latest plugins need to be downloaded from here.

After downloading and extracting them we need to place them in the Plugins folder under the CarlaUE4 project directory, located at <carla>/Unreal/CarlaUE4/Plugins (next to the Carla folder).

For further steps and Instructions it is strictly advised to follow the Document attached.
Mathworks Document to Import a Map in Carla

Please do follow each step slowly and at the end we will have our map successfully imported to Carla.
Then we can Run some Python scripts to check whether the Map is working correctly or not.

This process may take 30 mins initially to download everything and rebuild them for the first time, but later we can import a map within 5 mins.

Figure 3: Working of a Custom Map in Carla

2. Scenario Creation

First we need to follow the Mathworks Document to Import the map into Carla. Then We need to Remove all the RoutePlanners and SpawnPoints. To do the same we need to goto World Outliner in carla and search for the same and delete them all.

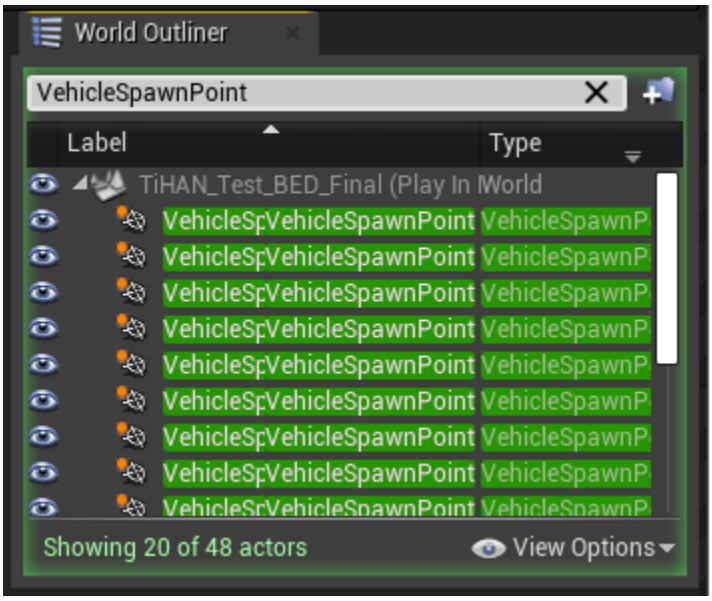

Figure 4: World Outliner in Carla

To facilitate exploration of the map, 18 new spawn spots have been added. We can add a spawn point using the PlaceActors Menu in the Unreal Editor.

Following the addition of these spawn places, we utilize Carla's built-in BasicAgent function to move through them in order. I've included a simple method of using these spawn points to navigate the map in the final code.

Additionally, Carla has a function called autopilot that we can activate after creating a vehicle to allow it to freely go across the map.

We can create a lot of scenarios with these two attributes.Since it is challenging to operate two vehicles at once that spawn at distinct time_stamps, we are unable to develop sophisticated scenarios like Cut-in and Cut-out. The logic in the code will gradually grow more complicated, slowing down and prone to errors.

To Navigate the Car through the map we just need to send the SpawnPoints to the BasicAgent of Carla, this BasicAgent finds the Shortest path between the Present Path and the Destination point and calculates a set of Waypoints based on Pure Pursuit Algorithm(This algorithm may change based on the Version of Carla Used).

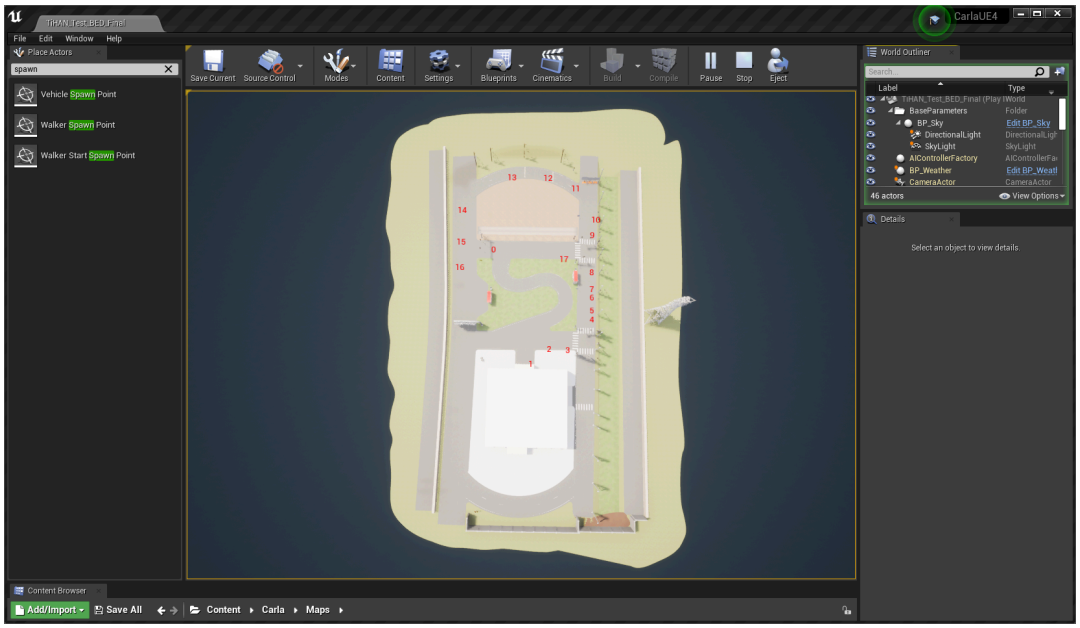

Figure 5: Place Actors Menu in the UnrealEditor.

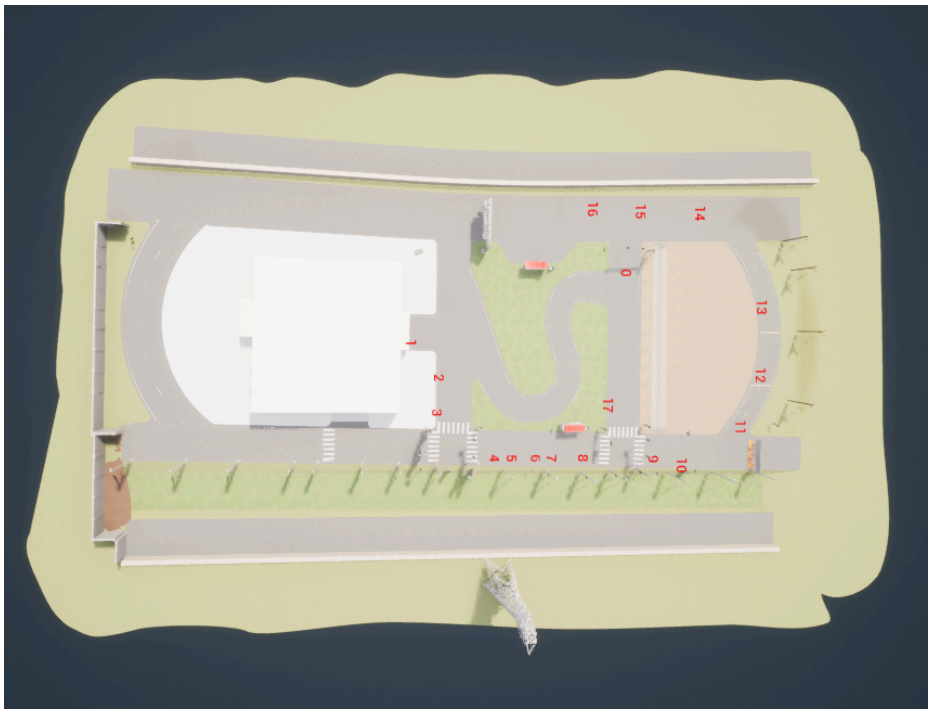

Figure 6: Different Spawn Points on the Map

3. YOLO Model Integration

First we need to Create an RGB Sensor and attach it to the Vehicle. (Detailed Explanation in the Final Code).
We can Fine Tune the parameters of the RGB Sensor as well.

```
# Find the blueprint of the sensor.
blueprint =
world.get_blueprint_library().find('sensor.camera.rgb')
# Modify the attributes of the blueprint to set image
resolution and field of view.
blueprint.set_attribute('image_size_x', '1920')
blueprint.set_attribute('image_size_y', '1080')
blueprint.set_attribute('fov', '110')
# Set the time in seconds between sensor captures
blueprint.set_attribute('sensor_tick', '1.0')

transform = carla.Transform(carla.Location(x=0.8, z=1.7))
sensor = world.spawn_actor(blueprint, transform,
attach_to=my_vehicle)
```

The above code lets us Create and spawn an RGB Sensor.
Then we take the Data from the Sensor using the Code below.

```
sensor.listen(lambda data: yolo(data))
```

Here display_yolo is a function which is called everytime a new Image is generated. We can write our own Function here and the data is directly sent to the function.
For More Customisation of the RGB Camera use this Document.

After receiving the data we pass it to the YOLO model and get the output from it and display the Same on a CV2 Window.
We use the Ultralytics Library to Load the YOLO model. And CV2 library to draw bounding boxes on Images after analyzing the Output of YOLO Model.

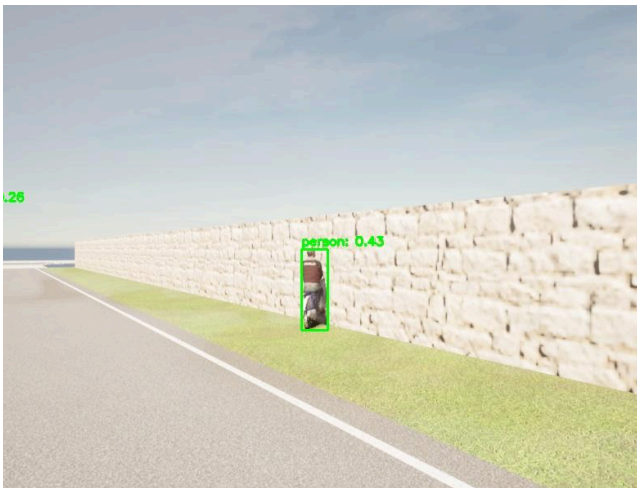

Figure 7: Detection of a Person after Integration of YOLO Model

4. ZED Camera Integration

As there is no direct sensor which works like a Stereo Vision Camera i created a new Sensor by fusing outputs from 2 sensors.

The ZEDCamera integration involves setting up RGB and Depth cameras in the CARLA simulation environment to provide visual data for the vehicle. This integration is essential for applications such as object detection, depth estimation, and visual navigation.

The ZEDCamera class initializes and configures two types of cameras: an RGB camera and a Depth camera. These cameras are attached to the vehicle to capture real-time visual data.

The RGB camera captures color images with a resolution of 800x600 pixels and a field of view of 90 degrees.It is spawned and attached to the vehicle at a specific location and orientation.

The Depth camera captures depth information, providing a 3D representation of the environment.Similar to the RGB camera, it has a resolution of 800x600 pixels and is attached to the vehicle.

The RGB and Depth cameras have callback functions (process_rgb_image and process_depth_image) that process and store the latest images captured by the sensors.The RGB images are converted to a numpy array and reshaped to exclude the alpha channel.The depth images are also converted to a numpy array for further processing.

After combining all outputs we get a 4 channel output where the first 3 channels correspond to RGB camera Data and the 4th one is the Normalized Depth Image data.

Then we apply the YOLO model on the first 3 channels and get Bounding box coordinates which are then used to calculate the Depth value of the Box with the help of the Depth Channel.

This process is computationally intensive due to which we may have a slight delay in the displaying time.

Figure 8: Output Image from ZED Camera with YOLO Bounding Boxes.

5. Vehicle Actuation

For vehicle actuators, we start with the input from the ZEDCamera, feed the first three RGB channels to the appropriate YOLO model based on the scenario, then extract the correct depth values from the fourth channel using the bounding box values that the model provides.

Since our bounding box is not limited to the object alone and contains a few outside object elements, we used np.min to get the Bounding box's Least Distance instead of np.mean, which will result in problems.

I have a set of object classes that the system will respond to, and we issue Warnings on the Screen based on the Depth values.

We apply the brakes on the automobile to stop it if the identified object is within the 0 to 8 meter range and is part of the class. The object is then displayed in red in the CV2 window. We cut the speed in half to slow down the vehicle if the item is between 8 and 15 meters away. The object is displayed in yellow in the CV2 pane.

Figure 9: Detection of Car with Red Warning Box

Figure 10: Detection of Car with Yellow Warning Box

Final Code

The provided Python script represents an advanced implementation for simulating various automated driving scenarios within the CARLA simulator. This comprehensive script facilitates the setup and execution of different scenarios, such as Automatic Emergency Braking (AEB) with a car or bike, collision warning, and speed bump detection. It achieves this by integrating several key components, including the CARLA client and world, RGB and depth cameras, and YOLO models for object detection.

Overview of the Script

The script begins by importing necessary libraries such as carla, cv2, numpy, and time. Additionally, it imports YOLO from the ultralytics package for object detection and the BasicAgent class from CARLA's navigation module to manage vehicle movements. Tkinter is used to create a graphical user interface (GUI) for scenario selection.

ZEDCamera Class

The ZEDCamera class is responsible for setting up and managing the RGB and depth cameras. The constructor (__init__) initializes the cameras using CARLA's blueprint library and attaches them to the vehicle. The RGB camera is configured with specific attributes such as image size and field of view. Similarly, a depth camera is set up with the same image size attributes. Both cameras are spawned in the world and attached to the vehicle.

The class includes methods to process images captured by these cameras. The process_rgb_image method converts the RGB image to a numpy array, while the process_depth_image method converts the depth image and calculates normalized depth values from the encoded RGB values. These depth values are then converted to real-world distances in meters. The latest RGB and depth images are stored as instance variables.

YOLO Model Loading

The load_yolo_model function is employed to load a YOLO model for object detection. It supports loading a pretrained model (yolov8n.pt) or

a custom model specified by the path. This flexibility allows users to experiment with different YOLO models based on their requirements.

Image Processing and Display

The process_and_display_images function is pivotal in the script. It processes images from the cameras, applies the YOLO model for object detection, and displays the results in CV windows. The function takes the ZEDCamera instance, YOLO model, and a title for the window as inputs. It starts by retrieving the latest RGB and depth images from the ZEDCamera. If valid images are available, it processes them using the YOLO model to detect objects.

For each detected object, the function calculates the minimum depth within the bounding box and evaluates whether the object poses a threat based on its distance and confidence score. Depending on the threat level, it sets a warning level and changes the color of the bounding box accordingly. These bounding boxes and labels (including class, confidence, and depth) are then drawn on the frame, which is displayed in a CV window. The function returns a flag indicating whether it should proceed and a boolean indicating if it is safe to proceed based on the detected objects.

GUI for Scenario Selection

A GUI is created using Tkinter to allow users to select different driving scenarios. The scenario_selection_ui function sets up this interface. The GUI includes buttons for selecting scenarios such as AEB with a car, AEB with a bike, collision warning, speed bump detection, and a full demo. Upon selection, the chosen scenario is displayed in a message box, and the GUI window is closed.

Main Function

The main function orchestrates the entire simulation. It starts by calling the scenario selection UI to get the user's choice. The CARLA client and world are then created, and any existing vehicles are deleted to ensure a clean slate for the simulation.

The blueprint library is used to find the vehicle blueprint (vehicle.mercedes.sprinter) and set up the spawn points. Based on the selected scenario, different spawn points and YOLO models are configured. The vehicle is spawned at the first defined spawn point and set to manual control (autopilot disabled).

For scenario 5 (full demo), an additional RGB camera is attached to the vehicle to demonstrate multiple tasks simultaneously. The ZEDCamera instance is then created, which sets up and manages the RGB and depth cameras.

The YOLO models are loaded using the load_yolo_model function. If scenario 5 is selected, two YOLO models are loaded: one for collision warning and another for speed bump detection. This allows the script to display outputs from both models in separate CV windows.

The BasicAgent class is instantiated to manage the vehicle's movement. The agent's destination is set to the next defined spawn point. A delay is added to ensure the map and sensors are fully initialized before proceeding.

Simulation Loop

The main simulation loop continuously processes the camera images and updates the vehicle's control based on the detected objects and their distances. The process_and_display_images function is called to process and display the images using the YOLO model. If scenario 5 is selected, a second call to process_and_display_images is made to process and display images using the second YOLO model for speed bump detection.

The agent's control commands are updated based on the processed images. If an object is detected too close to the vehicle, the script applies the brakes to demonstrate automatic emergency braking. The test vehicle used for the scenario is removed once the demonstration is complete.

If the agent reaches its destination, the script sets a new target from the defined spawn points. For certain scenarios, such as AEB with a car or bike, collision warning, or the full demo, additional test vehicles are created and added to the simulation.

The loop continues until all targets are reached or the user quits the simulation by pressing 'q'. After the simulation ends, the ZEDCamera instance is destroyed, and all CV windows are closed. Any remaining

vehicles and sensors in the world are also destroyed to clean up the environment.

Conclusion

Overall, this script showcases an advanced application of computer vision and machine learning in autonomous driving. By integrating real-time object detection and decision-making, it enables the vehicle to navigate complex environments safely. The script's modular design allows for easy customization and extension, making it a powerful tool for simulating and demonstrating various automated driving scenarios within the CARLA simulator.

Limitations Observed in Carla

- **Limited sensor customization**: CARLA offers a good selection of sensors, but advanced users might find it challenging to integrate and manage data from a wider variety of custom or non-standard sensors.
- **Data latency**: When using multiple sensors simultaneously, there can be a slight delay in data acquisition, which could impact real-time applications.
- **Limited Weather Representation** : CARLA provides a range of weather conditions, but may not fully capture the subtleties and dynamic interactions of real-world weather phenomena.
- **Limited scenario scripting**: Creating intricate scenarios involving multiple interacting vehicles (e.g., cut-in maneuvers) can be difficult within CARLA alone. Additional tools might be necessary for robust scenario development.
- **Right-hand traffic only**: CARLA currently only supports right-hand traffic configurations, which may not be suitable for regions with left-hand driving standards like India.
- **Platform Support :** We noticed that sometimes it is tough to install Carla on

Linux and sometimes it's tough to do the same on Windows. This happens especially when we try to install Older Version. It is always recommended to install the Latest version of the Software.

- **Computationally Expensive :** Carla takes a lot of Computer resources like CPU, RAM and GPU. Due to which there may be frame Drops and calculation errors in different computers. Results obtained may not look the same in Different computers.

- **Limited Actors Support** : The Current Code has a limitation for 2 actors, adding more is causing the system to drop a lot of frames and the results from the Sensors are getting bad.