List of figures

1. INTRODUCTION
2. GOAL OF THE PROJECT
3. OVERVIEW OF CARLA SIMULATOR
4. BUILDING CARLA
   1. LINUX BUILD
   2. WINDOWS BUILD
5. ADDING/MODIFYING A SENSOR IN CARLA
6. MODIFYING RADAR CODE
   1. UNDERSTANDING THE LIDAR SENSOR MODEL
   2. UNDERSTANDING THE RADAR SENSOR MODEL
   3. MODIFYING RADAR SENSOR MODEL
   4. RESULTS
7. DBSCAN CLUSTERING
8. APPLYING DBSCAN ON CARLA RADAR DATA
9. RESULTS
10. PROBLEMS ENCOUNTERED
11. FUTURE SCOPE
12. BIBLIOGRAPHY
13. INTRODUCTION

One of the primary requirements of autonomous driving is a reliable perception of the environment. This is generally provided by highly accurate sensors like Lidar, Radar and Camera. Also, the data is often provided in the form of 3D point clouds. Using different algorithms we can affectively derive important information from these point clouds.

Point Clouds are just datasets of points which represent objects or space in 3D format. Each point in the dataset comprises of X,Y and Y geometric coordinates. Most of the times they are generated by using a 3D laser scanner and LIDAR. Radar gives us the 3D point cloud along with one more important data which is range rate. Point clouds are comparatively easier to edit, display and filter. Also, they are a non-intrusive way to measure and object or its properties. Buildings and sites don’t need to be shut down in order to be measured.

Since, LIDARS emit light in all direction by rotating 360 degrees, more points are reflected which result in a dense collection of points. The scenes reconstructed from these point clouds are more detailed due to the density of the point clouds.

Radars point clouds, on the other hand, are generally sparce. They are generated by radio waves reflecting form the surface of buildings or objects. However, when it comes to dim lighting and weather conditions, they have an advantage over Cameras and LIDARs. A combination of these three sensors is extremely useful for Autonomous Driving.

Any AI neural network requires a lot of labelled data to train the model and work. Also, data for innumerable scenarios and test cases are required. Some of these scenarios are risky to carry out in real life. For example scenarios in which the car needs to get into an uncomfortably close proximity of a person or child. Moreover, the sheer volume of data to train a model is difficult and expensive to collect. This is where simulators come to the rescue.

Carla is one such simulator which is open source. It gives us access to unlimited Radar, Lidar and camera data for any required scenario. There is a drawback with the current Radar in Carla. It does not provide labelled data, which, as mentioned above is required for Machine learning Algorithms. So, our goal is to build Carla and modify the Radar code so that it returns labelled data which can be then used to get data and to be used for various research and development projects.

1. Goal of the Project

The goal of the project is to modify the Radar code so that it returns the ID of the objects being detected by the Radar. With this we will have access to labelled data which can be used to train various machine learning models and applications. After this, create different scenarios so that we can get the radar data, filter moving objects and cluster them.

The main tasks that were carried out are:

1. Build Carla and unreal engine on Windows and Linux PC.
2. Researching how to create/make changes to a sensor in Carla and understanding the Carla Pipeline.
3. Looking through the radar and Lidar code(C++) and making changes to the Radar code.
4. Creating a scenario and collecting simulated Radar data.
5. Filtering stationary and moving objects.
6. Performing clustering of the data using DBSCAN and enclosing them in bounding boxes.
7. OVERVIEW OF CARLA SIMULATOR

Carla(Car Learning to Act) is an open source simulator which has been developed from scratch to support and facilitate training, development and validation of open autonomous driving systems. Being open source, it gives free access to digital assets and has sensors models which can be used to create any scenario the is required.

The Carla simulator has a client server architecture, where the server side deals with all the simulation related functions like physics computation, actors, sensor visualization etc. It is crucial to have a dedicated GPU as Carla aims to make the simulation as real as possible. On the other hand, the client side is made of the modules that control the logic of actors on scene and set the world condition.

As the current goal of the project was to modify an existing sensor model, it was necessary to build Carla and Unreal Engine.

1. BUILDING CARLA

Building Carla from the source code is necessary if there needs be any further development. In the current project additional features needed to be added to the Radar sensor model. Carla can be built in both Linux and Windows environment, and it was built in both to check the viability. The Carla versions built are 0.9.10, 0.9.11 and 0.9.12. The most stable version that was found was 0.9.11 and the steps to build it is written below.

4.1. WINDOWS BUILD

The steps to building Carla are the key problems faced during the build will be mentioned in detail below.

1. Installing the software prerequisites:
   * CMake (3.26.0)
   * Python (3.8.8)
   * Make (3.81)
   * Visual Studio 17: Select ‘Windows 8.1 SDK’ and ‘Desktop development with C++’. Doing this enables the x64 command prompt that will be used to build Carla.

Add CMake, Python, Make to the Path in Environment variables. Also, it’s very important to use the exact same versions because maintaining correct versions is crucial while building Carla in windows. Version mismatches might lead to various Linker issues during the building Carla which are difficult to solve.

1. Setting up Unreal Engine for version till and including 0.9.11.

* Create an account on Epic Games.
* Run Epic Games Launcher using the Epic games account.
* Click on Unreal Engine tab and install version 4.24 or higher.
* After installation add the path to the Environment Variables. To do this, go to Advanced System settings-> Environment Variables and create a new variable by clicking ‘New’. Name the variable as ‘UE4\_ROOT’ and chose the path to the installation folder of the desired UE4 installation.

In Carla versions 0.9.12 and above Unreal Engine has a modified fork of 4.26 especially for Carla, for which the particular branch of unreal needs to be cloned and built on the local pc. The steps for which are given below.

* Link your GitHub account to Unreal Engine account, so that the particular fork of Unreal Engine can be downloaded.
* Clone the Carla Branch of Unreal Engine.

‘ git clone --depth 1 -b carla <https://github.com/CarlaUnreal/UnrealEngine.git> ‘

* Run the configuration scripts ‘Setup.bat’ and ‘GenerateProjectFiles.bat’.
* To compile the Unreal engine which has especially been made for Carla, the ‘UE4.sln’ file needs to be opened with Visual Studio (verify the visual Studios version based on the Carla version being built. Then build the UE4 solution keeping these settings: ‘Development Editor’, ‘Win64’ and ‘UnrealBuildTool’.
* To verify the successful build launch the UE4Editor.exe file and if it has been built correctly all the .uproject files should be linked to it. If not add it to Environment variables by creating the variable ‘UE4\_ROOT’ as mentioned above

1. Build Carla

After setting the Unreal engine, environment variables and all required software, Carla can now be built.

* To build Carla the commands have to be executed via the ‘x64 Native Tools Command Prompt for VS 2019’.
* For commands to build carla to work, it needs to be run in root CARLA folder, by navigating to it using the cd command.
* Now, the PythonAPI Client can be compiled. The PythonAPI Client is required to control the simulation. It gives us the tools required to create actors and scenarios which help to create scenarios customized to the desired test cases. Once the Client is compiled, scripts can be written to create scenarios and interact with the simulation. The command used to compile the PythonAPI Client is ‘make PythonAPI’.
* Once the PythonAPI is compiles it will result in 2 files ‘.egg’ and ‘.whl’ files.

The ‘.egg’ has been used in my case.

* The next step is to compile the server. The command ‘make launch’ is used next to compile and launch Unreal Engine.

1. Starting the simulation

To control the simulation using the python scripts it is necessary to ‘Play’ the simulation.

After which any example, preferably, dynamic weather, as it has very visible changes to the environment, can be used to check if the simulation is working or not.

* 1. Linux build

The second option is to build Carla simulator on a Linux OS. The software requirements will be mentioned in detail.

1. Software Prerequisites:

* Python (3.8.8)
* Cmake
* Clang

The best way to install all these requirements are to run the commands in the figure.

***PUT FIGURE HERE!!!!!!!!!!!!!!!!***

1. Building Unreal Engine

The next step is to get the Unreal Engine ready. For carla version 0.9.11 use Unreal Engine version 4.24. For different versions of Carla use respective Unreal Engine version.

* Clone Unreal Engine 4.24 on the local computer.

‘git clone --depth=1 -b 4.24 [https://github.com/EpicGames/UnrealEngine.git ~/UnrealEngine\_4.24](https://github.com/EpicGames/UnrealEngine.git%20~/UnrealEngine_4.24)’

* Get to the directory where UE 4.24 has been cloned and download the patch for Unreal Engine.

‘wget https://carla-releases.s3.eu-west-3.amazonaws.com/Linux/UE\_Patch/430667-13636743-patch.txt 430667-13636743-patch.txt’

‘patch --strip=4 < 430667-13636743-patch.txt’

* Build the unreal engine using this command.

‘./Setup.sh && ./GenerateProjectFiles.sh && make’

* Set the Unreal Engine environment variable by using the command given below. This will help Carla find Unreal Engine during launch.

‘ export UE4\_ROOT=~/UnrealEngine\_4.24’

* The variable, UE4\_ROOT, should also be added to’ ~/.bashrc’ or ‘~/.profile’ for it to be accessible session-wide.

This can be done by :

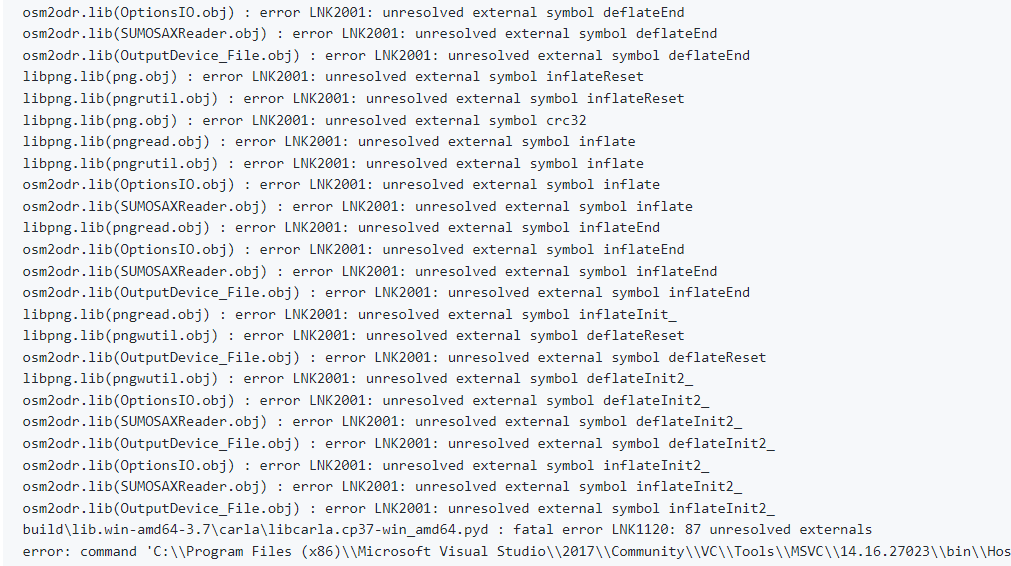
* + - * 1. Opening ‘~/.bashrc’ using command gedit ~/.bashrc
        2. Write the environment variable in the ~/.bashrc file: ‘export UE4\_ROOT=~/UnrealEngine\_4.24’.
        3. Now, save the file and reset the terminal.

1. Building Carla

The process of building Carla remains similar to that of the windows version and can be followed from above.

* 1. Issues faced with Windows build.

1. LNK2001: unresolved external symbol



<https://github.com/carla-simulator/carla/issues/3621>

This is one of the trickiest issues found during building carla. As this can occur due to various reasons and depends on the system being used at the time, it takes a lot of trials to get right.

1. Reasons for this issue:

* Multiple Python versions installed on the local system.
* Wrong version of python or other software.

1. Solution

* Uninstall other versions of python.
* Install only the correct version of python. (in the case of Carla 0.9.11, the most stable python version is python 3.8.8)
* Restore the pc to default factory settings and install all software from scratch. This worked exceptionally well in the current scenario.

1. RuntimeError: time-out of 2000ms while waiting for simulator

This error occurs if you do not press the play button in the Unreal Engine. The simulation environment is not activated and hence the scripts cannot find a simulator to run on.

1. Module ‘carla’ has no attribute ‘Client’.

The reason for this is due to an issue with the latest version of setuptools library in python 3 that builds the .egg files.

The issue can be resolved by using an earlier version of the library. In the above case, the command ‘pip3 install -Iv setuptools==47.3.1’ was used to install the version of setuptools that would solve the error.

* 1. Conclusion

There were no issues faced while compiling Carla in Linux OS. Comparatively, it was observed that the number of issues while compiling in Windows OS were just too many. Also, building Carla on windows is very version oriented. Therefore, any mismatch in version results in errors that cannot be resolved easily. Hence, Carla should be built on a Linux system to avoid these issues.

1. Adding/modifying a sensor in Carla.

A part of the goal of the project is to get more information from the radar sensor in Carla. To achieve this it is necessary to understand how sensors work and the changes that are required to be done in the code. So, by knowing how to create a sensor, the workflow will be clear, which will help in the modification of radar sensor.

In Carla, sensors are a special type of actor that produce a stream of data. This stream of data can be produces continuously, every time the sensor is updated or in the form of interrupts that only produce data after specific events. For example, Radar sensor which produces point clouds on every update, but a collision sensor only returns data when a collision happens.

There are 2 types of sensors in Carla. Client side sensors and Server side sensors. The Client side sensors don’t need to interact with the Unreal Engine(Server side). The best example being LaneInvasion sensor, which notifies every time a lane mark has been crossed. Whereas, the Server side sensors run inside the UE4 and send data back to the Python Client. For doing this they need to cover the whole communication pipeline which is shown in Figure 5.1***.*** Creation of server side sensor is what is required in this case.

A diagram of a server

Description automatically generated

Figure 5.1: Communication pipeline of Carla Simulator. **Image source:** Carla official Website. [1]

The major changes that need to be done in the pipeline can be divided over 5 major parts.

1. Sensor Actor

Sensor actor belongs to the AActor class as it is a special actor that that is used to measure or simulate data. The user can have access to it in the form of Sensor actor.

* It is a part of the UE4 framework which is the server side.
* As we discussed above the sensor actor is a special actor and therefore, it derives from the ‘AActor’ class(which is available in UE4).
* The ‘AActor’ class has a virtual method called Tick which can be used to update the sensor on every update from the simulator.
* Since an actor is any object that can be dropped into the world, the sensor also inherits this property.
* The code(.cpp and .h) files that govern the sensor are found in the path mentioned below. If a new sensor is to be created the .cpp and .h file needs to be created in this path. Since, the goal here is to modify the Radar sensor, the path to the radar sensor model files are given below.

‘Unreal/CarlaUE4/Plugins/Carla/Source/Carla/Sensor/Radar.h’

* The constructor can be used to create the trigger box. The tick can be enabled or disabled here. In our case the tick is required.
* Carla needs to be told what attributes the sensor has. This is done in the ‘GetSensorDefinition’ function. The function is defined in ‘ActorBlueprintFunctionLibrary.cpp’ whose location is provided below.

‘Unreal\CarlaUE4\Plugins\Carla\Source\Carla\Actor\ActorBlueprintFunctionLibrary.cpp’

These attributes are required to spawn the sensor. For example, some of the attributes of radar are horizontal field of view, vertical field of view, range, etc.

* Set Function: It is used to create a sensor on user demand. As soon as the sensor is connected, the set function is called with the parameters requested by the user.
* The Tick function is now used to and data is sent back to the client,

Every sensor has a stream that is used to send data to the client. While using \*sensor.listen(callback function) this stream is what it is subscribed to. The callback function gets triggered every time on the user side when some data is sent. However before this data can be accessed it needs to be serialized.

1. Serializer

It has functions that are used to serialize and deserialize the data generated by the sensor. It runs in the LibCarla which connects both the server and client.

* These are 2 files that belong to the LibCarla part of the pipeline. It consists of 2 static methods, Serialize and Deserialize. Below are the 2 files and their location.

‘LibCarla/source/carla/sensor/s11n/SafeDistanceSerializer.h’

‘LibCarla/source/carla/sensor/s11n/SafeDistanceSerializer.cpp’

* Serialize function: This function has 2 arguments one of which is the sensor and the other is a return buffer. The function gets the arguments that is passed to the ‘Stream.Send(…)’ function. The return buffer is ‘carla::Buffer’ which is just a dynamically allocated piece of raw memory. This is used to send raw data to the client.
* Deserialize function: The next step is to deal with the buffer that is coming to us from the serialize function, but this time it is in the client side. This data is deserialized and packed into a ‘RawData’ object.

But before this can happen an event needs to be defined so that this can be executed. It is done in the next stage i.e. Sensor data object.

1. Sensor Data

It stores the data generated by the sensor. This data will be sent to the final user. To accomplish his a data object is created which represents the data of the sensor.

‘LibCarla/source/carla/sensor/data/RadarData.h’

In the case of radar, the buffer we created in ‘Serialize’ will be interpreted as a structure containing information about range, azimuth, altitude and radial velocity. Our goal is to add one more variable to be returned. This will be shown later on.

1. Register the Sensor

In the above step the pipeline is completed. This sensor can now be registered. This is done in the file ‘LibCarla/source/carla/sensor/SensorRegistry.h’. After this the right data can be dispatched to the right serializer.

1. MODIFYING RADAR CODE

Lidar and Radar work on the same concept of shooting traces to get point clouds of the surrounding environment, and Lidar already has the feature in which it returns the object ids, so it was analyzed to get an understanding of how to proceed with getting the additional information from the Carla radar sensor model.

* 1. Understanding The Lidar Sensor Model

As mentioned above, Lidar sensor in Carla uses the technique of raycasting to simulate behavior similar to real world Lidar. Raycasting is a method in which virtual rays are cast from the Lidar sensor into the environment to measure the distance to the objects. Each of the these rays cover a designated field of view and each ray moves in a specific direction. Whenever a ray intersects an object, the intersection point and the required properties of the object are recorded. After this Carla assigns semantic labels to the detected objects. Semantic labelling is the process in which Carla assigns specific classification to roads, sidewalks, pedestrians etc. These labels provide contextual information about the objects detected by the Lidar.

Using the data collected by the intersection points and the semantic labelling associated with them, Carla is able to generated a point cloud representation of the scene. Each point cloud represents a location in 3D space and includes information such as intensity, position, range and semantic label. This data is then sent via the client server pipeline of carla to be accessed by the user.

The Figure shown below depicts the functional overview of the Ray cast semantic lidar code.

A detailed working of the Lidar code will be discussed below and the part of the code which deals with extracting the object id of the of the objects intersected by the lidar rays will be analyzed so that the same feature can be applied to radars.

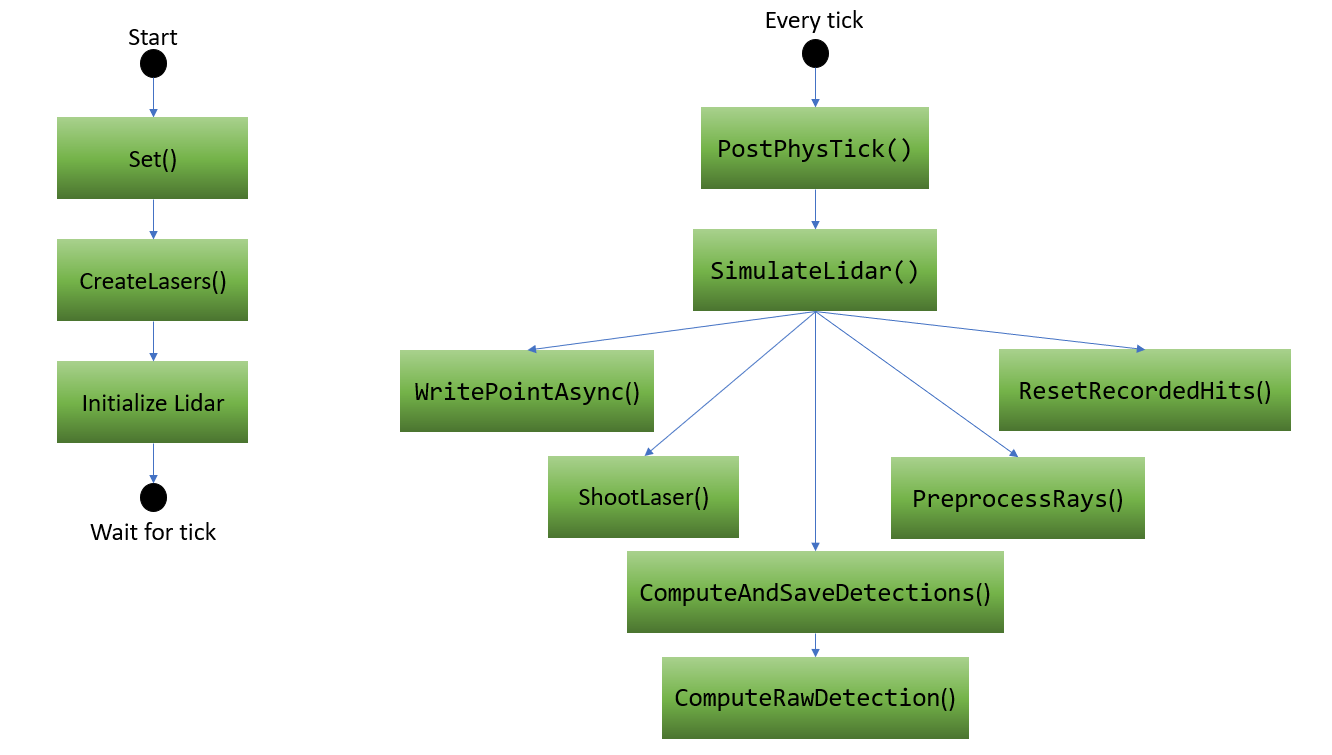


Figure 6.1.1 Flowchart showing the working of the Carla Semantic Lidar Sensor

Let's go through the code and explain the purpose of each function:

1. GetSensorDefinition():

This function returns the definition of the LIDAR sensor as an FActorDefinition object. It is used to define the sensor in the Carla simulator.

1. ARayCastSemanticLidar():

This is a constructor which sets the primary actor tick to be enabled. This enables the tick() function to be called on every frame.

1. Set():

This function sets the properties of the LIDAR sensor based on the provided FLidarDescription. It calls the CreateLaser() function which creates the lasers for the sensor based on the number of channels specified and initializes some data structures.

1. CreateLasers():

This function creates the lasers for the LIDAR sensor based on the number of channels specified in the LIDAR description. It calculates the vertical angles for each laser based on the upper and lower field of view limits and stores these points to be used later on by the SimulateLidar() function. With this function, Initializing the Lidar is done and the code waits for the physics tick of the world which then calls the PostPhysTick() function.

1. PostPhysTick():

This function is called after the physics tick of the world. It calls the SimulateLidar() function which is responsible for simulating the LIDAR sensor and sending the sensor data to the data stream.

1. SimulateLidar():

This function performs the LIDAR simulation for the given delta time. It does so by casting rays from each laser, detecting hits with objects, recording the hits, and generating semantic detections based on the recorded hits. It also handles the synchronization and parallelization of the lidar simulation process. This is a brief overview of how it works:

1. Based on the points per second mentioned in the Lidar attributes and the frame’s delta time, it calculates the number of points to scan with each laser in that particular frame. This ensures that there is consistent number of points per frame being generated regardless of frame rate.
2. If the number of points to scan with one laser is zero or negative then it means that no points were requested in that frame and in this case a warning log message is printed.
3. It gets the current horizontal angle of the lidar and converts it to degrees to represent lidars rotation around the vertical axis. It resets the recorded hits so that new lidar detection results can be stored.
4. After all this information is collected the lidar is ready to shoot traces and record detections. To do this parallel processing is used to iterate over each channel and each point to scan with one laser. Using parallel processing distributes the work across multiple threads to enable efficient computation.
5. The ‘HitResult’ object stores the result of the ray cast done in the shoot laser function. If the preprocess condition is ‘true’ and the laser successfully hits an object then the details about the lidar point stored in ‘HitResult’ object is written asynchronously to the recorded hits vector.

It calls functions ResetRecordedHits(), PreprocessRays(), WritePointAsync(), ComputeAndSaveDetections() and ShootLaser() functions to complete the above tasks. These functions will be discussed in detail below.

1. ShootLaser():

In Semantic Raycast Lidar, the ShootLaser function casts a trace from the lidar body to simulate the lasers detection. It returns “true” if the if the trace hits an object.

Here’s the brief overview of the function which demonstrates its working.

1. Create an object named ‘HitInfo’ of the class ‘FHitResult’ which will store the hit results of the trace.
2. Retrieve the location and rotation of the Lidar body. This will help in determining the location and orientation of the lidar so that the traces can be shot in the right direction.
3. Use the range from the lidar’s ‘Description’ and the forward vector of the laser rotation to calculate the endpoint of the traces.
4. With the above information, the traces can be shot and if the trace hits an object, the information is stored and the function returns ‘true’.
5. If the trace doesn’t hit an object the ‘HitInfo’ is not stored and the function returns ‘false’.

1. ResetRecordedHits():

This function resets the recorded hits data structure to prepare for storing new hits. It resizes the data structure based on the number of channels and the maximum points per channel.

1. PreprocessRays():

This function initializes the preprocess conditions for the rays. It sets all the conditions to true initially.

1. WritePointAsync():

This function asynchronously writes a hit point to the recorded hits data structure for a specific channel.

1. ComputeAndSaveDetections():

This function computes and saves the detections based on the recorded hits. It iterates over the channels and hits, computes the raw detections, and writes them to the semantic LIDAR data structure.

1. ComputeRawDetection():

This function computes the raw detection based on the hit information and sensor transformation. It calculates the detection point, incident angle, and object information for the hit.

As described above, the ShootLaser() function shoots traces into the virtual environment to collect detections. It does this by using LineTraceSingleByChannel() function that’s available in Unreal Engine. This function returns ‘true’ if there is a blocking hit.

The first parameter sent by this function named (in this case) ‘HitInfo’, is of data type ‘struct FHitResult’, which has a function ‘GetActor()’ which has the capability to return the actor that owns the component that was hit by the trace.

Once, the information of all the detection points has been collected, it is passed back to SimulateLaser() function, which then call ComputeAndSaveDetections(). ComputeAndSaveDetections() runs a loop going through the detections. It then sends each detection as a parameter to ComputeRawDetection() which uses the function GetActor() to extract the name of the actor hit by the trace. This name is then run through the Carla directory to ascertain the Object ID associated with the actor name.

Then the object id is then saved in variable called ‘detection’ which is of type ‘FSemanticDetection’. ‘FSemanticDetection’ is an alias of the class ‘SemanticLidarDetection’ whose member variables are used to store the location, angle of incidence of the rays, object ID and Object tag of the actor hit by the trace.

* 1. Understanding The Radar Sensor Model

In the real-world, radar sensors work by emitting radio waves and measuring the reflections or echoes from objects in the environment. Using these reflections we can get information about the distance, position of the detected objects and relative velocity.

In Carla simulator, this behavior is imitated by using ray casting. This means that traces are shot from the position where the radar is positioned in the environment and the details of the objects being hit by the traces are stored as detections. As discussed above, this is also like the working of the Lidar.

Here's an overview of how the Radar sensor works in CARLA:

Radar sensor is used by placing the radar in the Carla virtual environment with the help of python APIs that are available in Carla. The radar can be attached to a vehicle or just be placed in the environment as per the requirements. Various attributes that are specific to the radar sensor need to be defined here. These include field of view, points per second, range and sensor tick which define the horizontal and vertical field of view in degrees, points generated by all lasers per second, maximum distance to ray cast in meters, and simulation seconds between sensor ticks respectively.

To replicate the predefined pattern of the radio waves in the field of view, the simulation performs line traces from the radar sensor location to the maximum detection range. The number of rays emitted per second is determined by the “points per second” attribute that we initially defined while initializing the sensor.

These traces check for collision with objects in the virtual world. If a ray hits an object then a blocking hit or collision is detected. For each successful collision that is detected by the radar sensor, the hit result is then stored data structure and sent to a data stream to be accessed by the user. Using the listen() function we can access this data stream and get the detections from the radar. Each detection point has its own azimuth, elevation, depth and radial velocity.

After implementing the changes to radar sensor model code, the Object ID will also be available. This will help with getting labelled data which is useful for various Machine Learning models.

By simulating the behaviour of real-world radar sensors, the CARLA Radar sensor provides crucial information for perception and object detection algorithms in autonomous driving systems. The collected radar data can be used for various purposes, such as object tracking, collision avoidance, and scene understanding.

A diagram of a computer program

Description automatically generated with medium confidence

Figure 6.1.2 Flowchart showing the working of the Carla Radar Sensor

The flowchart in fig above shows the major functions of the radar sensor model and the order in which they are called. Let's go through the flowchart and describe its functions in detail.

1. GetSensorDefinition():

This function returns the definition of the Radar sensor as an FActorDefinition. It calls function ‘MakeRadarDefinition’ defined within ActorBlueprintFuntionLibrary.cpp. This function is used to define the sensor’s properties and uses default values to initialize the sensor attributes. By invoking the 'MakeRadarDefinition' function, the sensor attributes are systematically defined, adhering to professional standards. These values can be subsequently customized to align with specific user requirements.

1. ARadar():

This is the constructor of the ARadar class which is used to enable ticking by setting ‘PrimaryActorTick.bCanEverTick’ to true.

1. Set():

The Set() function calls the SetRadar() function defined in ActorBlueprintFunctionLibrary.cpp and sends it the attributes specified by the user. The SetRadar() function assigns these values to the attributes by using the functions SetHorizontalFOV(), SetVerticalFOV(), SetRange(), and SetPointsPerSecond().

1. SetHorizontalFOV(float NewHorizontalFOV):

This function sets the horizontal field of view (FOV) of the Radar sensor.

1. SetVerticalFOV(float NewVerticalFOV):

This function sets the vertical FOV of the Radar sensor.

1. SetRange(float NewRange):

This function sets the maximum detection range of the Radar sensor.

1. SetPointsPerSecond(int NewPointsPerSecond):

This function sets the number of radar points or rays emitted per second.

1. BeginPlay():

This function is called when the Radar sensor actor begins play. It initializes the ‘PrevLocation’ member variable with the initial location of the actor.

1. PostPhysTick():

The function is called after every physics tick of the world.

1. It calls CalculateCurrentVelocity() which returns the current velocity of the radar.
2. The data structure used to store detections of the previous frame is reset.
3. Calls the SendLineTraces() function which simulates the entire radar. It shall be explored in detail later.
4. Lastly, it sends collected radar data through the data stream.
5. CalculateCurrentVelocity(const float DeltaTime):

The function calculates the current velocity of the Radar sensor based on its location changes. It uses the previous and current location of the sensor to estimate the velocity.

1. SendLineTraces(float DeltaTime):
2. It takes ‘DeltaTime’ as a parameter so that it can shoot traces and gather detections for this period.
3. The max radar radius in the horizontal and vertical direction is calculated based on the initial attributes given by the user, i.e., horizontal and vertical field of view (FOV) angles and range of the radar.
4. The number of traces to be generated for raycasting is calculated by multiplying the delta time and points per second and then converting it to an integer.
5. A critical section(‘Mutex’) is acquired to ensure thread safety. Read access is locked in the physics scene.
6. A parallel loop is run for all the rays that must be traced.
7. The end location of each ray is determined based on factors such as the radar's position, rotation, range, and the maximum radii in the horizontal and vertical directions.
8. The LineTraceSingleByChannel() function performs a line trace from the radars location to the end location and store the result of the hit status and actor.
9. If the trace hits an object the value is set to ‘true’.
10. The critical section is now unlocked for read access.
11. Finally, a loop iterates through the stored ray values, and only the rays that hit an object are retained as detections.
12. CalculateRelativeVelocity():

This function uses the velocity of the vehicle with the radar sensor and the target object to calculate the relative velocity.

* 1. Modifying Radar Sensor Model

In the above sections, the functionality of the radar and lidar sensors has been described. In this section, the modification to the radar sensor code will be covered along with where the changes need to be made to the entire Carla pipeline. This information will form the backbone of modifying the existing sensors in Carla to add features and tailor them to specific research requirements or use cases. In this case, getting labelled data from the radar sensor which can be used for a variety of other machine learning algorithms.

* + 1. Modifying radar features(radar.cpp)

The moment a trace hits the object in the environment, data about the object gets stored in the ‘LineTraceSingleByChannel’ function and details can be extracted for further use. This function is called in the ‘SendLineTraces’ method.

As the name suggests, the ‘SendLineTraces’ function deals with shooting traces from a small area, outward in multiple directions to mimic the electromagnetic pulse which gets sent out by a radar sensor. This is done by using a different thread for each trace. A mutex(critical section) is also used to ensure thread safety. As read access is locked here, the code modification to add any features which involves extracting actor or episode details shouldn’t be done in this loop. Doing so will result in a the simulator crashing without a significantly helpful error message. Once the threads have finished executing and the critical section is free the ’Rays’ array will hold information of all the traces shot by the simulator. This is where the code to add the feature to the sensor will be implemented.

The logic of the ‘SendLineTrace’ function can be seen in the Figure 6.3.1.a. The code after the ‘Close Threads’ block has been modified. This flowchart shows the logic of the added feature.

A diagram of a process

Description automatically generated

Figure 6.3.1.a: Flowchart of the SendLineTraces Functions.

A computer screen shot of a program code

Description automatically generated

Figure 6.3.1.b: The modified code which extracts and returns the Object ID.

For a ray to be relevant it needs to have collided with an object. The ‘LineTraceSingleByChannel’ function which is used to create the rays, returns a Boolean value. This will be true if the ray has hit an object. Using this, a check is placed to keep information of only the rays with collision.

Now, the actor details have to be extracted from the information stored in the ‘Rays’ array. Individual actor details are stored in the Actor registry which is a part of the Episode.

In Carla simulator, an episode refers to a single run or a discrete run of a simulator scenario each containing a specific scenario or task within the CARLA environment. Every object in Carla has an Episode[1]. During an Episode, various entities such as vehicles, pedestrians and environmental elements interact based on predefined rules and behavior. Episodes serve as important tools for evaluating and testing algorithms, control strategies and autonomous driving systems within Carla. It holds all the information for a scenario for that instance of time and is responsible for various aspects in the duration of the Episode. It takes care of the initialization in which it sets up the environment including configuration of the vehicles, pedestrians and other objects. During an Episode execution, it handles the parameters and rules of the simulated scenario like the interaction between actors and the world. Data collection takes place simultaneously throughout the episode like sensor readings, vehicle telemetry and simulation metrics to facilitate analysis and evaluation. Finally, when the Episode concludes, the data is saved and the simulation environment is reset to prepare for the upcoming episodes.

Therefore, to get the object information, the actor registry in the Carla episode is extracted. The actor information acquired by the ‘LineTraceSingleByChannel’ is matched against the information in the actor registry and the Object ID of the matching actor is stored to return to the User. The code implementation can be seen in Figure 6.3.2. At this point, the object Id has been extracted and to make it accessible to the user, the data has to be provided a path through the pipeline connecting the user to the carla simulator. This will be described in the next section.

* + 1. Modifying the Carla pipeline

As shown by Figure 5.1, for a sensor feature to be successfully implemented, the path of the return data from the sensor should be established through the entire Carla Pipeline. The changes made in the above section was only in the simulator end of the pipeline. Files in the Libcarla and PythonAPI folder also need to be changed so that the simulator knows the changes in the structure storing the detections and also establishes enough space to transfer it through the TCP protocol.

Libcarla:

A screen shot of a computer program

Description automatically generated

Figure 6.3.2.a. Radar detection structure

The ‘RadarData.h’, file, as shown in Figure 6.3.2.a., has the data object for the radar sensor. Up until now, the data structure had 4 values, i.e. azimuth, elevation, relative velocity and range. The Object\_Id variable needs to be added to this structure and the size of the new structure needs to be updated to include the new variable. This structure template is going to reinterpret the buffer that was created in the Serialize method. The serialize method will now accept variables velocity, azimuth, elevation, range and Object ID.

PythonAPI:

A computer screen shot of a program code

Description automatically generated

Figure 6.3.2.b. Output stream from sensor to User

Once the data structure in Libcarla has been modified and updated with a new variable, this change should be conveyed to the PythonAPI part of the pipeline. The PythonAPI is closest to the user. It should be able to parse the data correctly and should be updated with the Object\_ID.

As shown in the figure 6.3.2.b., the file ‘SensorData.cpp’ contains the code which parses the output stream coming from the Unreal engine. The ‘operator<<’ function is used for output stream insertion which allows objects of the ‘RadarDetection’ class to be streamed to an output stream. The variable ‘obj\_id’ needs to be added in this section of the code with the other radar detection variables so that the parser knows that the detections have been updated with an extra value.

* 1. Results

Now that the code has been modified it can be built and launched. The changed made to the code will reflect in the radar detections. The detection will also show object IDs of the actors that come into the field of view of the radar Sensor.

A screenshot of a computer program

Description automatically generated

Figure 6.4.1: The radar detection from unmodified and unmodified Carla simulator.

Figure 6.4.1. depicts the previous radar output and the output after making changes to the Carla Simulator and the object ID can be seen. It is to be noted that some of the Object IDs are ‘0’. This is because Carla returns ‘0’ for stationary objects like trees and buildings[4] whereas, actors created during execution and traffic signs that come with the map have unique Object IDs.

A simple code was written to create a scenario with a couple of actors and the radar mounted on an ego vehicle. This can be used to verify that the Object IDs are correct.

A car on the road

Description automatically generated

Figure 6.4.2: A scenario in Carla Simulator. The Cybertruck is the Ego vehicle.

As shown in Figure 6.4.2, the scenario consists of a ‘Tesla Cybertruck’, which is the Ego vehicle and has the radar mounted in front of it. The other three actors in the front are namely, on the right – ‘Harley-Davidson low-rider’, in the middle – ‘Audi Etron’ and on the left – ‘Pedestrian 0002’. These details are provided in the Blueprint Library in Carla[6].

The simulation is executed and the detections acquired by the radar sensor are stored.

The ‘world’ is an object representing the simulation. It acts as an abstract layer containing the main methods to spawn actors, change the weather, get the current state of the world, etc.[7]. Using the ‘world’ object the list of actors in that simulation can be acquired and the actor matching the Object ID will be returned. If the actors used when creating the scenario match the actor names returned by the Object ID, the code changes made to the simulator can be verified.

A screenshot of a computer program

Description automatically generated

Figure 6.4.3 Code which searches the Actor List in the World Object and stores it in a file.

The radar data returned by the Carla simulator has the detections and is it stored in an array of frames, where each frame corresponds to one episode in the Carla simulator. The function shown in Figure 6.4.3 gets the detections of one frame at a time. It then goes through all the detections an finds the ones that are not ‘0’, as these detections are the stationary objects like trees, buildings etc. Once it gets a non-zero Object ID, it compares it to the actor list which it acquired from the world object. Upon finding a match it stores the details of the actor in the output file. The contents of the file are shown below in Figure 6.4.4.

A computer screen shot of a program

Description automatically generated

Figure 6.4.4: Display the values ​​returned by matching the Object IDs in the Actor list

As shown in Figure 6.4.4, the Object ID from a few detections are taken and the actor details extracted from the Actor list are printed below each respective detection. The names of the actors created and placed in the Carla environment as seen in Figure 6.4.2, can be seen in the output file thus verifying that the Object Ids returned to us by the changes made in the Carla simulator are correct and they can be used to label data for other implementations.

1. DBScan

In the field of radar signal processing, the Density-Based Spatial Clustering of Applications with Noise (DBSCAN) algorithm emerges as a powerful tool, seamlessly addressing the challenges posed by complex and dynamic radar environments. As radar systems become increasingly sophisticated, the need for robust and efficient clustering techniques becomes crucial to extract meaningful information from vast and intricate datasets.

DBSCAN, which was introduced in the field of data mining, has found a natural application in radar technology due to its ability to identify clusters of radar point clouds based on their spatial density. Unlike traditional clustering algorithms, DBSCAN excels in handling irregularly shaped clusters and is particularly adept at discerning outliers, which is crucial in the presence of noise or unexpected radar reflections.

As DBSCAN is a density based algorithm, it is perfect for defining clusters as regions of high radar point cloud density, enabling the identification of distinct objects or phenomena. This adaptability proves invaluable in scenarios where conventional clustering methods may struggle, such as in the presence of variable target shapes, clutter, or interference.

The efficiency and reliability of DBSCAN in radar applications make it an indispensable tool for various purposes, including target detection, tracking, and classification. Its ability to adapt to different radar environments and accommodate varying target densities positions DBSCAN as a versatile solution that enhances the overall performance and accuracy of radar systems.

As radar technology continues to evolve, the integration of advanced clustering techniques like DBSCAN signifies a paradigm shift towards more intelligent and adaptive signal processing. The application of DBSCAN in radar not only enhances the capability to discern relevant information from complex data but also paves the way for further innovations in the ever-evolving field of radar technology.

* 1. Parameters of DBSCAN

A circle with orange dots and green dots

Description automatically generated

* Epsilon: Epsilon is the radius of the circle which needs to be created across each data point to check the density of the cluster.
* *minPoints*: It is the minimum number of data points that needs to be contained in a data points circle for it to be classified as a core point.
  1. Types of points

A diagram of circles and dots

Description automatically generated

DBScan works by creating a circle of length Epsilon radius around every data point and classifying them into core points border points and noise.

* Core Points: A data point is classified as a core point if it has the number of data points as specified by parameters *Minpts* inside the radius Epsilon.
* Border Points: If a data point has points less than the minPts inside the radius Epsilon then it is classified as a border point.
* Noise: If a data point is not in the Epsilon radius of any other point or has no points other than itself inside its neighborhood it is considered as Noise.

As shown in the figure above, we have considered minPts=3 and a circle of radius Epsilon is drawn around each point. The points depicted in Green are the core points as they have at least 3 points in their circle including themselves.

The data points in Red are the border points and they have less than 3 but greater than 1 point in their circle.

Lastly, data points in Blue are the noise/outliers. These data points have no other points apart from itself inside the circle.

* 1. Reachability and Connectivity

A diagram of a diagram of a diagram

Description automatically generated with medium confidence

If a data point is reachable from another data point directly or indirectly, it’s called reachability. Whereas connectivity defines if two data points belong to the same cluster or not. []

* Direct Density Reachable: A point is said to be Direct Density Reachable if it falls within the neighborhood of a core point.
* Density Reachable: If a point is connected to another point through a series of core points, it is said to be Density Reachable.
* Density Connected: If there is a core point which is density reachable from both points, then they are said to be Density Connected. It shows a symmetric relationship that demonstrates connectivity between 2 data points in a cluster.

1. APPLYING DBSCAN ON CARLA RADAR DATA

Write a short introduction

* 1. Scenarios to be tested

<https://www.analyticsvidhya.com/blog/2020/09/how-dbscan-clustering-works/>

<https://open-instruction.com/ml-algorithms/overview-of-dbscan-clustering-algorithm/>

[1] <https://carla.readthedocs.io/en/latest/core_world/>

[1] <https://carla.readthedocs.io/en/latest/tuto_D_create_sensor/>

[3]

[4] <https://github.com/carla-simulator/carla/issues/3191> tells that 0 is returned for stationary objects

[5] <https://carla.readthedocs.io/en/latest/python_api/#carlaworld> gives none if it doesn’t know the actor id

[6] <https://carla.readthedocs.io/en/latest/bp_library/>

[7] https://carla.readthedocs.io/en/latest/foundations/

A black screen with white text

Description automatically generated

A screenshot of a computer program

Description automatically generated

[3]