Think Green, Think Clean

Final Project Report by Benjamin Ratin

Table of Contents

0. Abstract

The project described in this report introduced a crawler vehicle that can drive autonomously and cover every street in a neighborhood to perform a useful task. The project is built using the Carla simulator and Carla APIs. It relies on Carla libraries to perform low-level driving tasks while focusing on routing. A detailed description of the routing algorithm is included in the report. In addition an object detection system is implemented to assist driving and performing tasks in the future. The feasibility of building a vehicle that is able to plan and navigate a route that covers every street in both directions has been successfully explored. While the vehicle is able to crawl the map successfully, there is room for further improvement. The route planner can be better optimized to reduce the number of times the vehicle drives over the same roads. The object detection also could be enhanced, especially in the area of training the model on Carla representation of real-world objects.

1. Introduction

Today, a number of routine tasks in the neighbourhoods are performed by human drivers. Activities like garbage collection, street cleaning, checking streets for dangerously parked cars or making sure streets are safe after a storm could be automated.

I built a simulated prototype of an autonomous vehicle that can drive on neighborhood streets and perform whatever tasks may be necessary. My vehicle has level 5 autonomy. The challenges that a developer of such a vehicle has to solve are: routing, navigation, lane detection, intersection recognition, mapping, object detection, traffic light recognition. In addition to driving, the vehicle would have to perform a useful task, so there are challenges related to that, but they are out of scope of my project.

1.1 Goal of the Project

The goal of the project was to build a crawler vehicle that can drive autonomously and cover every street in a neighborhood to perform a useful task. A project like this requires a large team and a lot of time. It is impossible to build the entire autonomous system in a few weeks and I relied on a lot of prior work (see below). I was looking at the project's goal from two directions: to

learn as much as I can about autonomous vehicles and to solve some specific challenges for my project while relying on prior work to have solved a number of other challenges.

A unique challenge in my project is vehicle routing and navigation. Since I am developing a “neighborhood crawler” that needs to cover every street in a neighborhood as opposed to a more usual task of driving from point A to point B, I needed to develop my own routing algorithm and a navigation system. This is where my primary focus was.

1.2 Level of Autonomy

My vehicle has level 5 autonomy.

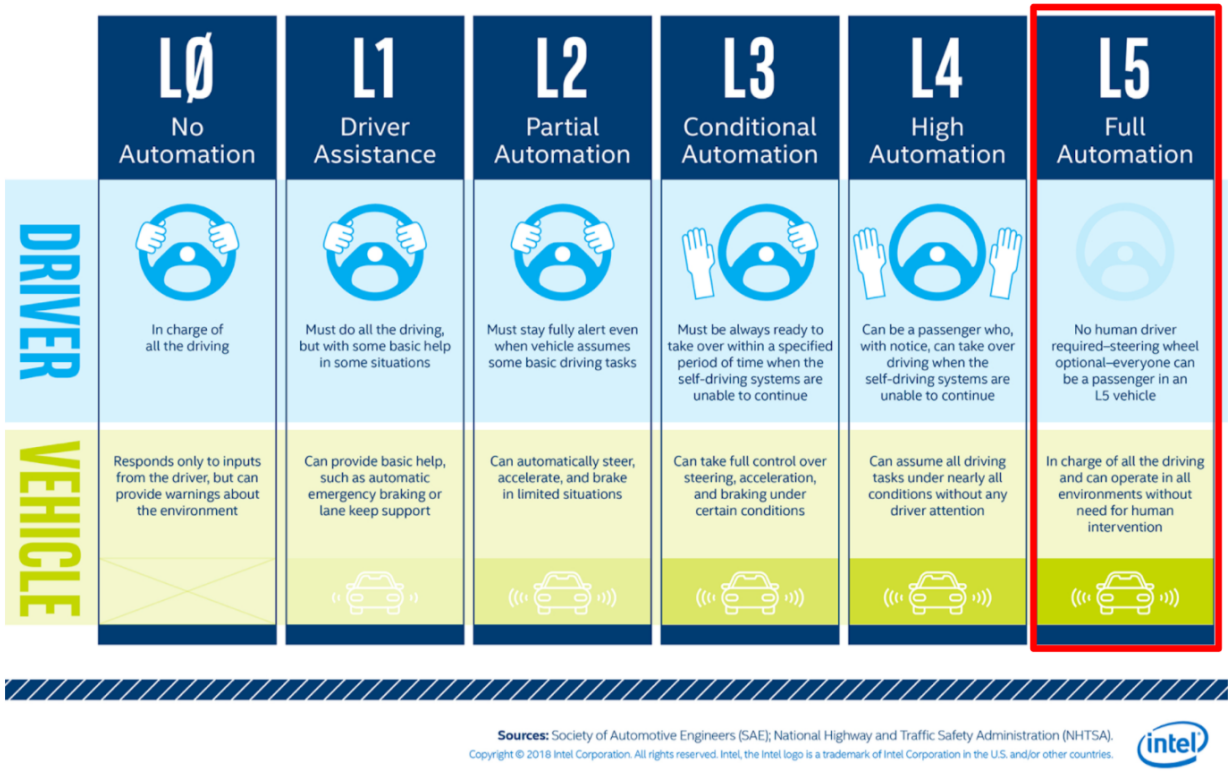


Figure 1: Level of Autonomy

I believe that one day such vehicles will be deployed in neighborhoods to drive completely autonomously. I think this adoption could start in enclosed environments, such as golf courses for example, where the overall setup is pretty well controlled. From there, these vehicles could be used in small neighborhoods and from there in more open environments, like towns and cities.

1.3 Description

1.3.1 Inputs

The initial inputs to the vehicle is:

- A map of the area
- A region within the map the vehicle is tasked to cover (in my project, I am using the whole map)
- An actual task to be performed (out of scope for my project)

1.3.2 Architecture

The first step in the process is for the vehicle to compute the route, based on the inputs. Once the route is computed, the vehicle drives along it from one waypoint to the next.

A fully autonomous vehicle architecture is shown in Figure 2 below:

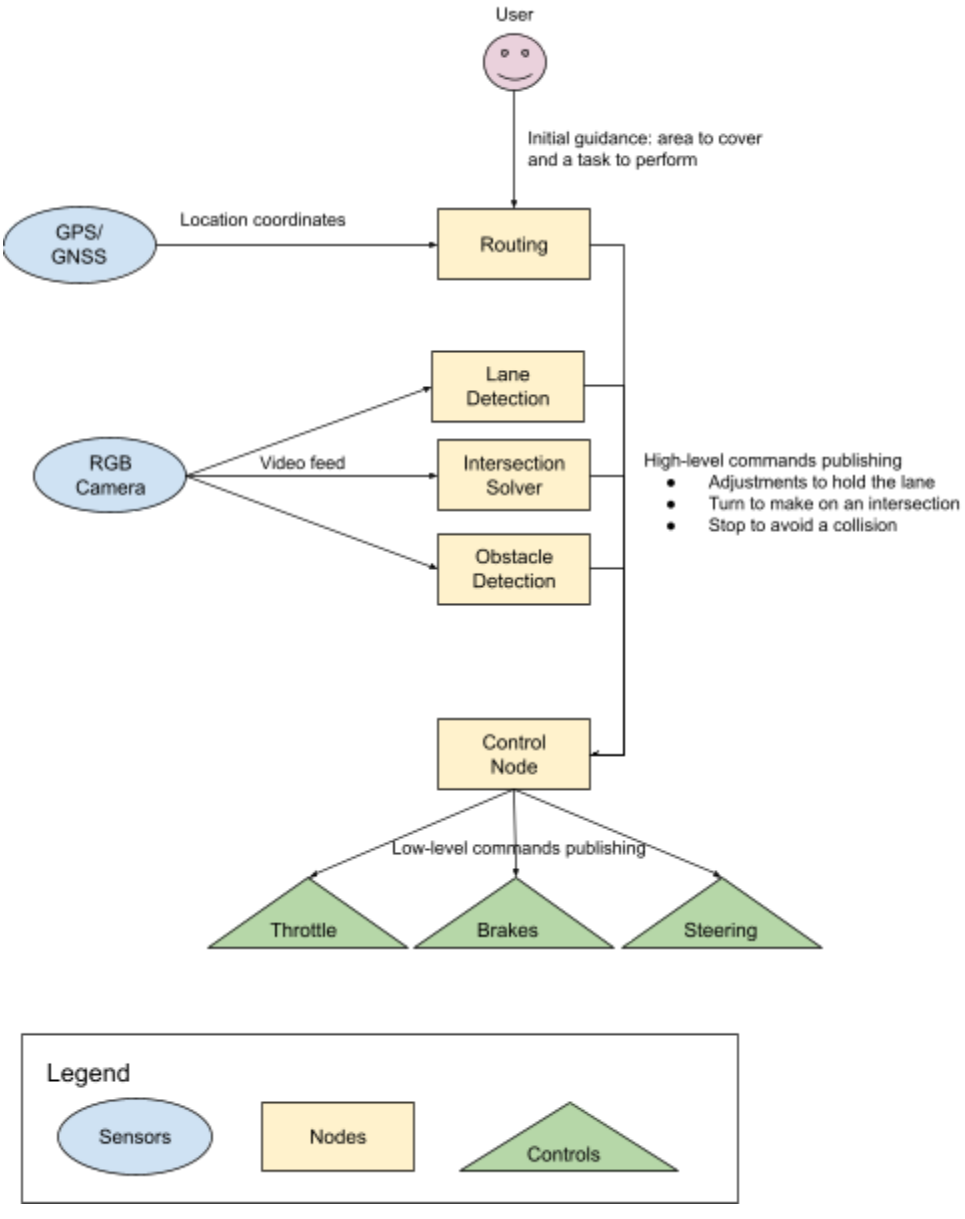


Figure 2: A fully autonomous vehicle high level architecture

The diagram above describes the following high-level architecture elements and relationships between them:

- The system is initialized by a user who sets up an area to cover and a task to perform.
- An RGB camera is feeding images into the nodes that are responsible for driving functions (lane detection, intersection, obstacle detection)
- A GPS/GNSS sensor is feeding current coordinates to the routing node for navigation
- The commands from the higher-level nodes are at the level of making a turn or stopping.
- These commands are issued to the control node that is responsible for relatively small adjustments of the vehicle controls.
- In turn the control node is responsible for command execution and translation of those high-level commands into smaller commands to adjust the vehicle speed and direction.
- Low-level commands are sent down to the vehicle controls

Note: the diagram above does not illustrate the “payload” modules like the actual garbage collection or location of illegally parked vehicles. It is easy to imagine how those nodes can be added to the overall system as required.

For my project I focused on the higher-level tasks such as planning the route and I left low-level driving to Carla libraries. The class hierarchy in my project is shown in Figure 3 below:

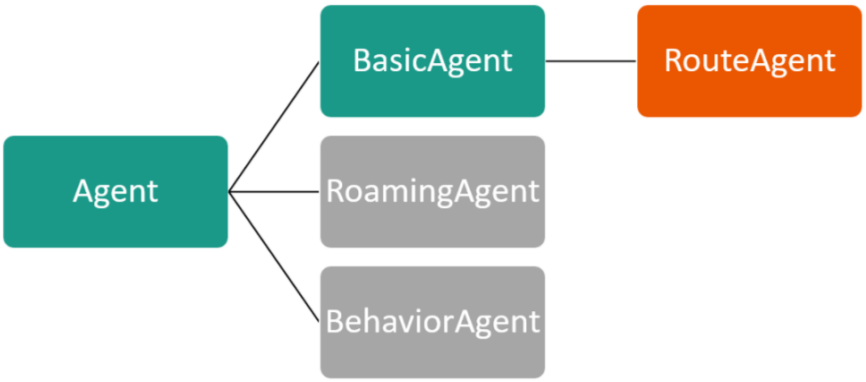


Figure 3: Carla API Agent class hierarchy

The Agent class in Carla represents basic functionality of all Agents that can control actors within the simulation. BasicAgent class, inherited from Agent, and supporting classes implement the ability of the vehicle to drive to a given destination. I wrote a new class, RouteAgent, that inherits from BasicAgent to add functionality such as routing and handling a list of destinations.

The two classes in gray, RoamingAgent and BehaviorAgent are not used in my code.

My project's high level architecture is shown in Figure 4 below.

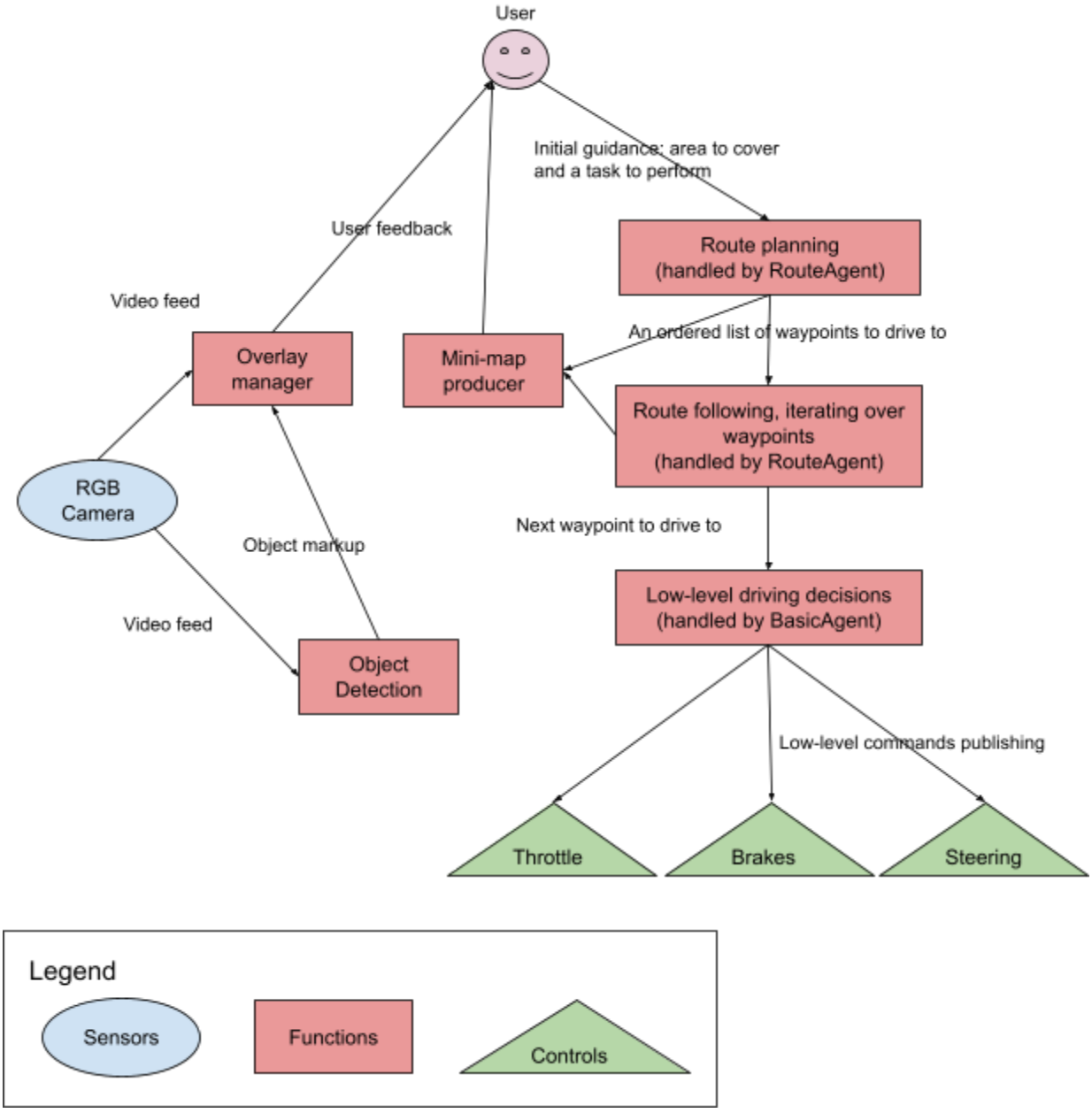


Figure 4: Project Architecture

The diagram shows the relationships between route planning, iterations over waypoints and the use of BasicAgent to perform the low-level driving and navigation. The RGB camera sensor is used to capture images in front of the vehicle. Those images are augmented by the object detector and are presented to the observer in a video stream. In addition, a minimap with the current status is produced.

1.3.3 Route Planning

Before the vehicle starts driving, it needs to play the route through the neighborhood. My algorithm, looks like this:

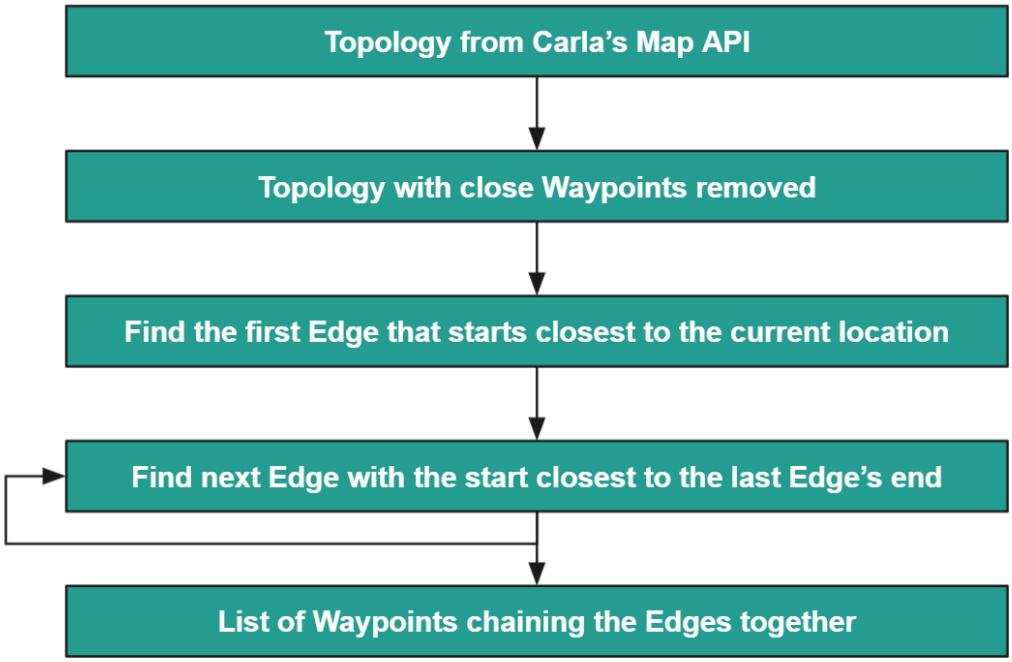
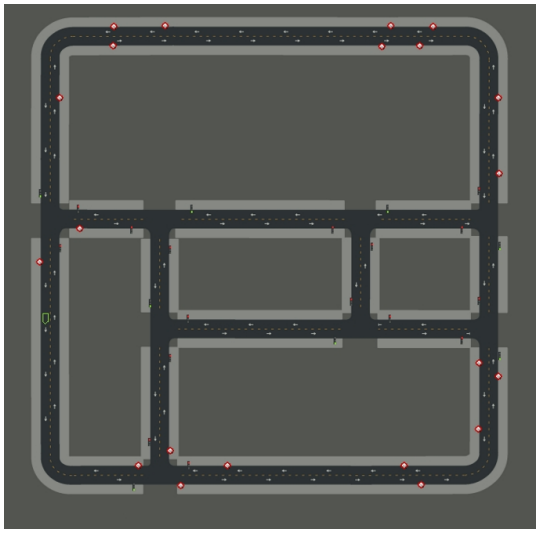
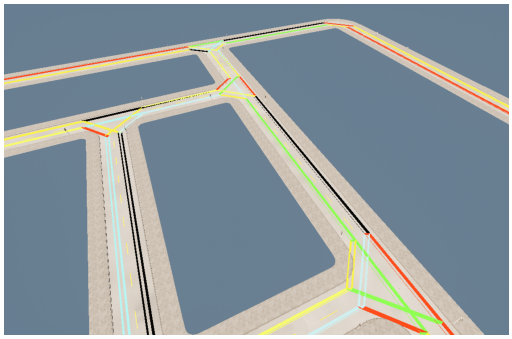


Figure 5: Route Planning Algorithm

- First I get topology from the Carla API. I worked a lot with "Town02" map:



- Carla's representation of topology is a list of pairs of waypoints that are directly connected. If a given street has two lanes, there will be 4 waypoints, 2 on each lane at the ends of the street, connected by two segments. A visual representation of the topology looks like this:



- My next step is to combine waypoints that are close to each other. Carla topology has some waypoints within map centimeters of each other and I do not need that level of precision for my vehicle.
- I then create a list of edges between each pair of waypoints. The idea is that if I have my vehicle drive along each edge I will then cover the entire map in both directions.
- To determine the sequencing, I first find the edge with the start closest to the initial location of the vehicle. Carla's API has a function that determines how far two waypoints are from each other in terms of the route between them, so I rely on that function to measure the distance.
- Once I've identified the first edge that the vehicle should drive on, I take the end of that edge and find the next edge with the start closest to that point.
- I repeat this process until I have covered all the edges. Since the edges are directional, meaning that there are two edges for each two-way street, once the vehicle has driven over all edges, the entire map will be covered.
- Finally, I create a list of waypoints that represent the starts and ends of the edges in sequence. I now have my route. A visual representation of the full route looks like this:

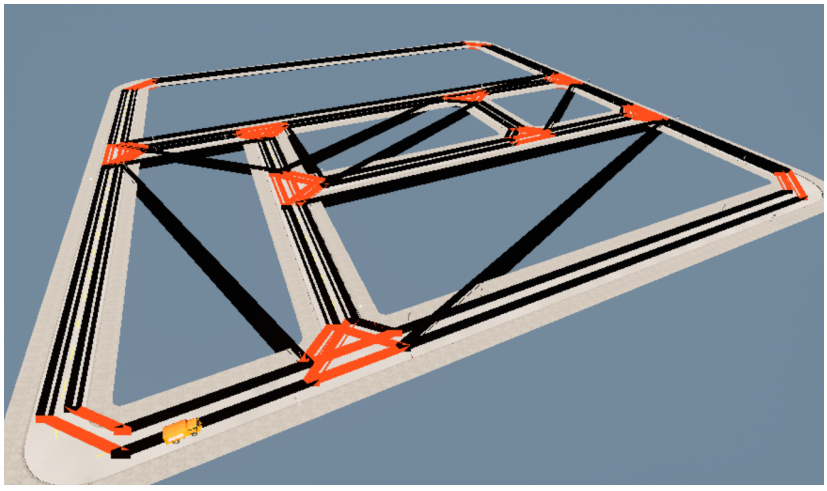


Figure 6: Full Route produced by the Planner

You can see from the image that I have full coverage of the map. Please note that this is not the low-level driving route, but a graph that connects higher level waypoints. As such certain lines do not go over roads. Sometimes in order to get to the start of a segment, the vehicle will need to go over an already covered part of the route, which is an opportunity for further optimization of my routing algorithm.

- I have also built a mini-map feature that visually shows the route plan. In the screenshot below:
    - Green points and lines show waypoints and segments that have already been covered
    - White point and lines show future waypoints and segments to be driven over
    - Yellow point shows the next destination waypoint
    - Yellow line show the current segment being driven on
    - Read circle represents the vehicle moving around

  This map is updated in real time

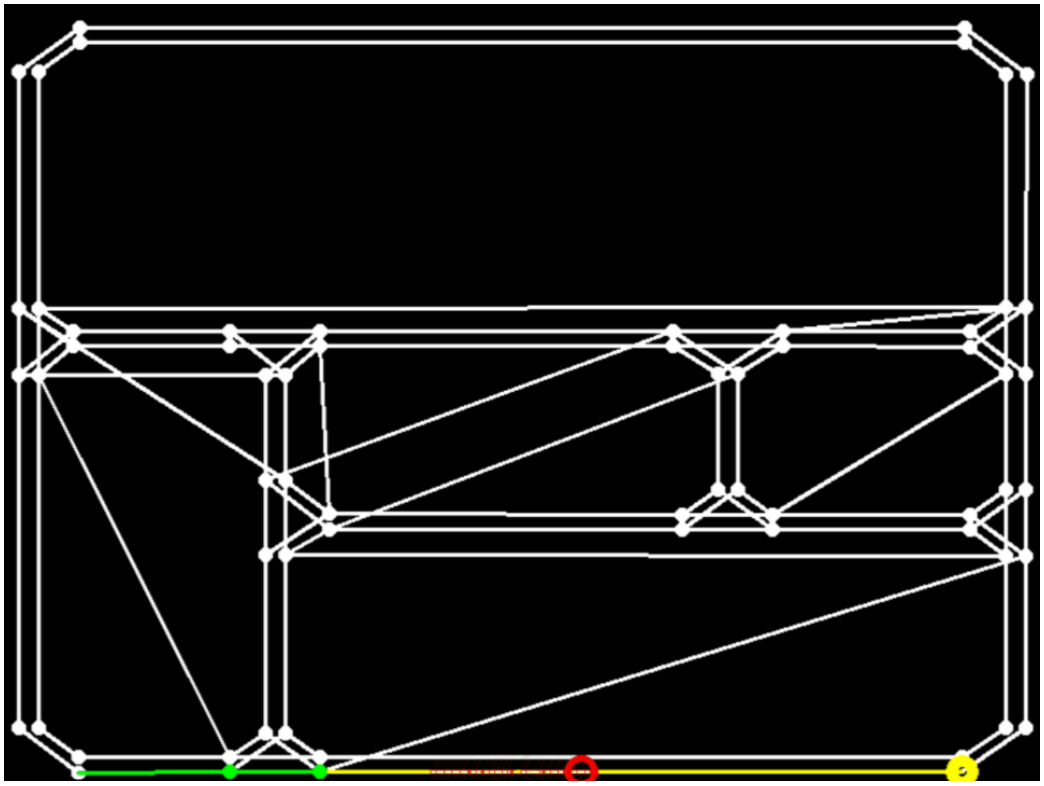


Figure 7: Mini-map with the current vehicle status

1.3.4 Object Detection

Object detection for the vehicle is required for both driving and for performing its task.

For object detection I rely on OpenCV dnn_DetectionModel API. I found pre-trained models online that allow for object detection in real time:

- https://www.youtube.com/watch?v=HXDD7-EnGBY
- https://github.com/ankityddv/ObjectDetector-OpenCV.git

I used Carla's RGB camera sensor's output to capture images and pass them to the detection API and to overlay bounding boxes on top of the images from the camera. I referenced my sources at the bottom of the slide. The resulting output looks like this:

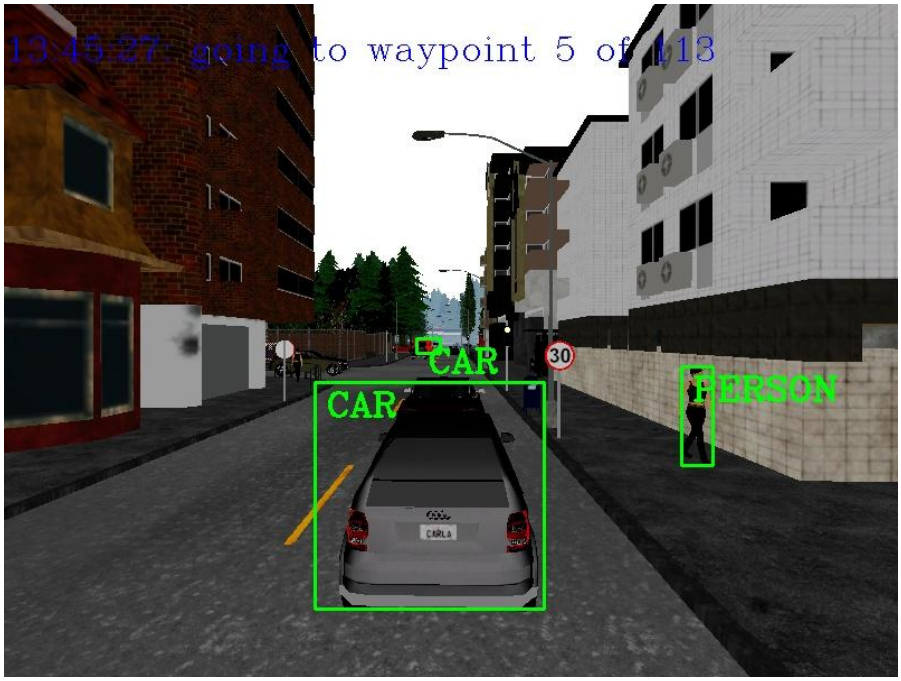


Figure 8: Object Detection Screenshot

2. Related Work

I was impressed with the work done by "Team Delamain" on the "Interceptor 4" project presented in the first lecture. At the same time, I was thinking that to implement a system like this in real life requires so much time and effort that it would be more practical to consider some more incremental steps, like slower-driving vehicles in more controlled environments. This is

where my idea came from. I was also impressed by the capabilities of Carla simulation environment that I learned about from that presentation (and since then with a lot of research).

I consider Carla built-in autonomous driving capabilities part of the related work. While I focused on the functionality that is unique to my project, I relied on a lot of existing features within Carla itself. There are powerful tutorials that I have been following to learn more about specific of Carla environment:

- Carla documentation: https://carla.readthedocs.io/en/0.9.11/
- Useful tutorials: https://www.youtube.com/watch?v=J1F32aVSYaU (multiple parts)

For object detection I relied on the following tutorial and the training model:

- https://www.youtube.com/watch?v=HXDD7-EnGBY
- https://github.com/ankityddv/ObjectDetector-OpenCV.git

Other related work includes:

- The material we are have studied in the course
- OpenCV documentation and tutorials
- NumPy documentation
- ROS architecture and tutorials
- Some ideas from Gazebo and Unity
- Self-driving car research and development done in both Universities and the Industry
- Some basic prior knowledge I have about graph theory and linear algebra

3. Team Organization

I worked on this project by myself, while relying on the material learned in this course, documentation and online tutorials as well as the existing Carla and other libraries.

4. Software and Development Tools

4.1. Software

- Carla simulator (based on Unreal Engine)
- Python 3 (on both Windows and Linux)
- Multiple Python libraries
    - Carla Python API and all of the underlying libraries
    - OpenCV/cv2
    - NumPy
    - Standard Python libraries
- Visual Studio Code
- Google Drive
- GitHub to keep my code under source control and for backup purposes
- ROS (for learning purposes; I did not use ROS directly in the project)

4.2. Laptop / Desktop setup

- A Windows Desktop Intel Core i7, 32 GB RAM, NVidia GTX 970 GPU
- An Ubuntu virtual machine running on top of Windows/VMware Player (I also experimented with Virtual Box), hosted on the Desktop above. I used it only for learning purposes; the final version of the project does not require Ubuntu.

4.3. Hardware

- I used a simulator environment.

4.4. Simulator

- Carla simulator, latest version 0.9.11 (I also experimented with Gazebo and Unity, but so far I found Carla to be the most useful for my project)

5. List of Milestones

As mentioned above, I will be focusing on navigation and routing aspects first. Once that is done, I will see if I can make more improvements to the project. In the meantime, I will be learning about how Carla implements driving primitives. It's analogous to what would have happened if I had other members of the team focusing on other aspects of the work: I would want to know how their parts of the project work, but I would focus mostly on my parts.

Milestones	Date(s)	Comments
Familiarize myself with robotics development tools, like ROS	6/22 - 7/10	
Pick a simulation engine	7/3-7/13	Looked at Unity, Gazebo and a few other options
Setup development environments	7/3-7/14	Setup a Windows and an Ubuntu dev environments with all the tools ready.
Basic self-driving car using existing Carla tools	7/16	Have basics working and going through more complex tutorials
Routing/navigation module prototype	7/25	I will start with a standalone set of functions that can plot the route among waypoints
Routing/navigation module fully integrated	7-29-8/1	
Identification of possible	Ongoing	

improvements to the overall system		
Work on the presentation with demo video(s)	7/28-8/3	
Final presentation	8/3 or 8/5	

I worked on the project on my own.

6. Results and Discussion

6.1. Routing

After a number of iterations I came up with the routing algorithm described above. My algorithm works well in terms covering every lane of every street on the map. In terms of performance, it takes about 10 seconds to plan the route through a somewhat simple map of Town02 and the complexity will grow proportionally to the number of segments on the map. I am concerned that with very complex maps it may take a long time to finish planning. On the other hand, once the route is known, it can be reused, so the vehicle does not have to recompute the route everytime it starts driving.

In terms of the route efficiency, I tried to minimize the number of times that that vehicle driver over the same lane as well as the total length of the route. My algorithm works well, but I think there is room for improvement in the routing algorithm.

6.2. Object Detection

I did not have too much time to spend on object detection and I relied primarily on pre-existing work. I did adjust some thresholds to make it work a bit better within the Carla environment. As it is, since the model was trained on real-life objects which do not look exactly the same within the Carla environment, there are definitely some object detection errors. While I think on the whole detection works well, I think if I had the time to retrain the model with Carla object I would have achieved better results. I also did not have the time to implement object tracking and I treat each frame separately. I am sure with object tracking from frame to frame I would achieve much better results for the overall object detection.

7. Conclusions

During this project I came to the following conclusions:

- It is possible to build a level-5 autonomous vehicle to cover every lane on every street in a neighborhood.

- I developed a routing algorithm that works well, but can be further optimized.
- Existing OpenCV object detection API is powerful and fast enough to do object detection in real time, even without optimizations like object tracking.
- Modern simulators, like Carla, are very powerful and work great to support experimentation such as in my project.

8. Future Work

The future work on this project would include the following items:

- Include a task execution, like a garbage bin, parked car, etc.
- Improvements to the route selection algorithm
- Better training on the object recognition model (on Carla objects)
- Object tracking
- Real AI driving based on sensors
- Re-routing in case of road obstruction
- Usage of ROS bridge and open source ROS Nodes

9. References

Overview:

- The material we are have studied in the course

Carla documentation and tutorials:

- Carla documentation: https://carla.readthedocs.io/en/0.9.11/
- Useful tutorials: https://www.youtube.com/watch?v=J1F32aVSYaU (multiple parts)

For object detection I relied on the following tutorial and the training model:

- https://www.youtube.com/watch?v=HXDD7-EnGBY
- https://github.com/ankityddv/ObjectDetector-OpenCV.git

Python libraries

- Carla API: https://carla.readthedocs.io/en/0.9.11/python_api/
- OpenCV documentation and tutorials: https://docs.opencv.org/ and https://docs.opencv.org/master/d9/df8/tutorial_root.html
- NumPy documentation: https://numpy.org/doc/stable/reference/index.html
- ROS architecture and tutorials: https://www.ros.org/
- Some ideas from Gazebo and Unity: http://gazebosim.org/tutorials and https://docs.unity3d.com/Packages/com.unity.device-simulator@3.0/manual/index.html
- Python 3.7 documentation (on both Windows and Linux) https://docs.python.org/3.7/library/
- Visual Studio Code: https://code.visualstudio.com/
- Google Drive: https://www.google.com/drive/
- GitHub to keep my code under source control and for backup purposes: https://github.com/

10. Software Installation

10.1 Requirements

- Windows 10 (this may work on Ubuntu as well, but I did not run my software on Ubuntu, since I do not have an Ubuntu machine with a GPU)
- A GPU supported by Carla (most modern GPUs are supported)
- Carla software version 0.9.11
- Python 3.7 (the only version of Python supported by Carla's Python API library on Windows as of Carla version 0.9.11)
- OpenCV module for Python

10.2 Installation

- Install Carla 0.9.11. Following the instructions here:
    - Download from here: https://carla-releases.s3.eu-west-3.amazonaws.com/Windows/CARLA_0.9.11.zip
    - https://carla.readthedocs.io/en/0.9.11/start_quickstart/

      Note: Carla 0.9.12 was released on 8/3/21. I did all of my development and testing with Carla version 0.9.11, referenced above.

- Install my software in a directory within PythonAPI directory under Carla root

This is what Carla installation looks like:

- carla ← the root directory of the installation
    - PythonAPI
        - carla ← location of Carla API modules
        - *my-project* ← a directory where my project should be installed

An easy way to achieve this is to go into the "PythonAPI" directory and from there run 'git clone' command like this (in this case, I installed Carla into E:\carla):

E:\carla\PythonAPI> git clone
git@github.com:berratin123/DGMD-S17-submission-Benjamin-Ratin.git

After this, you can cd into the cloned directory and run my script from there, see below.

Note:

When my main.py starts it needs to find an ‘egg’ file with the Carla API. It’s looking for the file in the ‘../carla/dist’ directory. This means that when my software is run, the working directory needs to be a “sibling” directory of ‘carla’. The easiest way to achieve this is to install the software in the sibling directory and run it from there.

In addition, the only ‘egg’ file available in Carla’s installation for Windows is for Python 3.7, which is why all Carla tools need to run under Python 3.7. If you have multiple version of Python installed you can run like this ‘py -3.7 *script*.py’

11. Running the Software

11.1 Carla Server

Carla has a client-server architecture. The first thing that you need to start is the Carla server, which is achieved by:

- Go into the top directory of Carla installation (E:\carla in my case)
- Run .\CarlaUE4.exe [parameters]

I run my Carla server with the following parameters to make things run faster:

E:\carla> .\CarlaUE4.exe -quality-level=Low -fps=15

11.2 My project

As mentioned above, my software needs to be started from a directory that’s a “sibling” of ‘carla’ within the Python API. Change directory into the cloned directory, and start it like this:

E:\carla\PythonAPI\DGMD-S17-submission-Benjamin-Ratin> **py -3.7 .\main.py**

As stated above, Python version 3.7 is a requirement for Carla Python API on Windows.

Note: the client/server communication timeout is set to 10 seconds. Still, when the machine is busy, it sometimes takes more than 10 seconds to load the initial map. If you get an error from main.py that it times out:

- Make sure the Carla server is running
- Try again. The second time it does not need to reload the map and the connection is significantly faster.

If you want the project vehicle to **ignore traffic lights** to make it run faster, you would need to modify the BasicAgent class that is responsible for it within Carla’s library. You can open:

E:\carla\PythonAPI\carla\agents\navigation\basic_agent.py file and add a line

```
light_state = False
```

On line 107 (before “if light_state:” line)

You can stop my software by pressing Ctrl-C.

11.3 Other Carla Tools

There are a number of useful examples inn the examples directory here:

E:\carla\PythonAPI\examples>

The most interesting tool for my project is the one that allows to swans other vehicles and pedestrians within the simulation: You can run it like this:

PS E:\carla\PythonAPI\examples> py -3.7 .\spawn_npc.py -n 50

Where ‘50’ is the number of actors you want to spawn.
Just like with my software, the examples rely on the ‘egg’ file to be in the ‘../carla/dist’ directory, so I run the examples after going into the directory where they are located.