Smart Applications report

Simulator group
Lavorati Lisa (535658), Morisco Michele (505252),
Ninniri Matteo (543873), Piras Andrea (619640)

December 2021

1 Overview

Unlike the other groups, who have been assigned one single, larger task each, the Simulator group has been assigned several smaller tasks through the course, each one with the common goal of automatizing as much as possible the other groups' experience with the simulator. As a result, the biggest part of our job has been the development of scripts capable of automatizing most of the basic tasks executed by the other groups although, in the final part of the course, we have also helped the other groups by pointing out issues found while integrating the various modules, as well as helping them by testing their solutions and by passing .bag files to the groups which were unable to run the simulator because of hardware limitations. This is the reason this report won't describe an architecture of any kind, as most of our scripts are stand-alone components with no real architecture, but it will instead list all the various tasks assigned to the Simulator group as well as how they were resolved, as well as insights on preliminary solutions and extra tasks executed autonomously by the group by our own initiative.

2 Preliminary analysis

2.1 AirSim

The first task assigned to our group was to analyze the first version of the ETeam simulator, which was created using Microsoft's AirSim simulator. The main objectives of the assignment were to:

- Assess the usability of the simulator on the middle to low-end machines such as the average laptop used by a university student
- Run the ROS bridge
- Detect any issue in the simulator, and propose possible solutions
- Propose possible enhancements

The four subtasks have been documented in the first report released by our team on November 2021 and, as a result, they will not be described in detail in this document. A short summary is as it follows:

- The simulator was unusable when run by using the Vulkan graphics library, and it was only capable to run with the OpenGL library.
- Even when run with OpenGL, the simulator was still unable to run satisfactorily on low-end machines. A partial solution to this problem was to edit the configuration file named `GameUserSettings.ini` in order to reduce the rendering resolution down to 50% or less, by adding the line `sg.ResolutionQuality=50.000000` to the `ScalabilityGroups`.
- The ROS bridge couldn't be run at all, which was confirmed by the ETeam as well.
- There were countless physics bugs in the various tracks.
- We have proposed a track generator as a mean of testing the car under unpredictable scenarios.

After examining the simulator, we have been asked to search for possible alternatives to the AirSim package. We have tested three possible alternatives:

- AirSim for Unity3D
- Carla
- LG SVL

Briefly, AirSim for Unity3D was plagued with bugs as its development was still at an early stage. LG SVL seemed good but, eventually, the E-Team decided to go with Carla, although we were able to run the ROS bridge on LG SVL as well, as described in the following section.

2.2 LG SVL

SVL is a simulator developed by LG Electronics America R&D Lab.
It is a multi-robot simulator based on HDRP Unity for autonomous vehicle developers. This simulator was well-suited for our purpose because it uses Unity, a lighter graphics engine than Unreal, and it supports ROS.
As in AirSim, SVL's recommended requirements are:

- at least 4 GHz Quad core CPU
- NVIDIA GTX 1080 (8GB memory) or higher
- Windows 10 (64-bit), Ubuntu 18.04 (64-bit), or Ubuntu 20.04 (64-bit)

In any case, our tests have shown that it was capable of running even on lower specs computers, such as a laptop running with a GTX 960m or a GTX 920mx.
Indeed, we ran the simulator with a low spec PC with a **2GB** GPU and it ran properly. The simulator is currently supported for Windows (64-bit) and Linux (64-bit), but the developers recommend using Windows for optimal performance.
The last requirement of this simulator on Windows is a graphics card that supports *DirectX 11.* Instead of Linux, it requires a graphics card that supports *Vulkan 1.1.*

In our experience with SVL, as mentioned previously, it runs smoothly on a low-end PC.

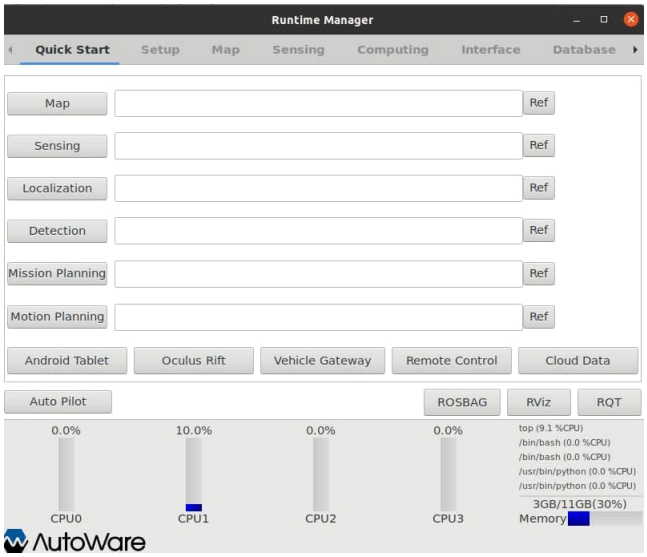


Figure 1: Autoware menu screen.

Using this simulator, you need to register for free on the manufacturer's website, this allows each user to launch the simulator in a cloud or local environment. On the simulator website, it is possible to use assets created by other users, completely free, such as maps, sensors, and plugins. To create the simulation just insert the cluster, which is the physical computing setup used to run a single simulation. After that, you can create your simulation by adding a map, the car, and its sensors. Since SVL is open-source it is possible to modify some parameters and add your own assets using the Unity editor.
In our experience, we had some problems importing the project on Unity because in the documentation they specify that the version is *2020.3.3f1*, but it is better to use the **2020.3.19f1** version. Then, the recommended version generates errors due to version migration. After importing the project on Unity, you can add your assets in the *Assets/External* folder after that you have to create a new build of the simulator in order to test it.
The biggest problem that we encountered with this simulator is the use of the ROS bridge proposed by the LG developers that did not work properly. But, we have found a solution with **Autoware.AI**, which is open-source software for self-driving vehicles that provides a ROSBAG-based simulation environment. This software can communicate with the SVL simulator using the ROS bridge, which provides JSON interfacing with ROS publishers/subscribers.
Thanks to *Autoware*, we were able to launch the ROS bridge with Rviz, indeed the bridge was able to communicate correctly and the car could move automatically in a predetermined path, even if only slightly.
In conclusion, SVL was a good simulator for our purpose but we could not do more tests about it because the E-Team crew chose the CARLA simulator for their purposes.

3 Carla analysis

After selecting Carla as the base for the next version of the simulator, our group received several more tasks:

- Learning to run the Carla simulator

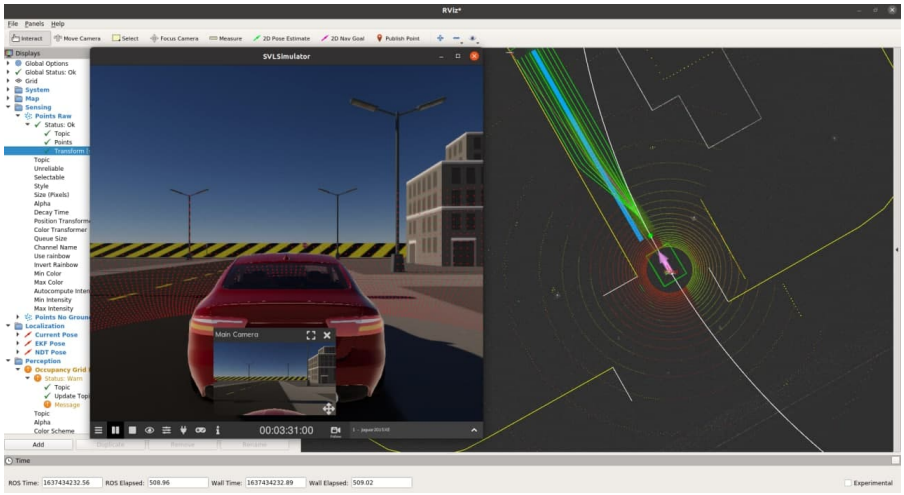


Figure 2: Rviz and simulator with ROS bridge and Autoware.

- Learning to run the Carla ROS bridge
- Import the ETeam car's sensors data into Carla from AirSim
- Trying, if possible, to move the AirSim simulator's assets into Carla.

The first three tasks have been completed successfully in a day. The fourth task required installing the simulator on the Unreal Engine 4 game engine, which was a slightly harder task to perform as it was highly demanding both in terms of disk space and computational resources. Also, Carla requires a custom version of a specific version of the engine in order to run correctly, which is slightly trickier to install as it doesn't follow the standard commands used to install the "stock" version of the program. At the same time, importing the ETeam car into the simulator was deemed impossible for the simple fact that the source code of the simulator's previous version, made with AirSim, didn't include the relative assets. Regardless, we were able to perform small experiments with the importing of the tracks before the ETeam released its own official version of the simulator.

4 Running the ETeam Simulator

Right after the official version of the ETeam simulator, running on Carla, was made available, we were tasked with running it on our machine to check if everything was working as expected.
Our task was hindered by the fact that the ETeam used a different installation procedure than the nominal one recommended by the Carla documentation, making our previous work with the simulator basically useless. The biggest issue was related to the Carla ROS bridge and was caused by a version mismatch between such bridge and the ETeam's simulator. This issue was solved by modifying a line in the file `bridge.py` of the Carla ROS bridge package, an action which was not required when running the ROS bridge on the "vanilla" simulator.

5 Automate installation and execution

Once we were finally able to run the final version of the simulator on a machine, we have been tasked with what could be considered the main task of our group, which was focusing on automatizing as much as possible the other groups' experience with the simulator:

- Create a bash script capable of automatizing the installation of the simulator on a machine
- Create a bash script capable to run the simulator with the requested modules

5.1 The setup script

This bash script is to be run on the root folder of the simulator. Once it is opened, what it does is essentially to download and install:

- ROS
- The Carla ROS bridge
- MEGAcmd and the Simulator's extra data
- Our autorun script's extra files
- The Smart Application groups' requirements
- The Smart Application groups' packages

As simple as it might seem, especially considering that for most of those packages there is plenty of documentation associated, this procedure was not trivial. For example, we had to keep in mind that:

- The ROS bridge's `bridge.py` file needs a specific row to be modified in order to solve the version conflict issue mentioned in the last section
- Our autorun files, as described below, requires to be modified at install time in order to add some of the machine's informations inside them
- There have been several issues with the Smart Application's dependencies and folder structure which required several modifications

Our installation file, namely `autorun_setup.sh`, was tested and found to be working on several machines equipped with different types of NVidia GPUS. It has not been tested on AMD machines both because some of the Smart Application modules requires CUDA to work and also because we couldn't find anyone with an AMD GPU to test it. Still, we believe we have found as many issues as possible, especially considering that we tested the script on a clean Ubuntu install. Running it is as simple as running `./autorun_setup.sh` on a shell.

5.2 The autorun script

The required autorun script had to feature three different modes:

- Run the simulator with the ROS bridge
- Run the simulator with the ROS bridge and the manual controls enabled
- Run the simulator with the Smart Applications' autopilot

All the modes implemented the same core idea, which is:

- Run the core simulator
- For each module requested, open a new bash console and run it

The modules varied from mode to mode, but all of them needs the two ROS launch files `spawn_vehicle.launch` and `spawn_track.launch`. As our code base grew and we needed to perform more complex tasks such as enabling the manual controls or disabling the UI elements, we have progressively realized that both launch files were relatively limiting for our needs as we needed to run different python files than the ones hard coded on `spawn_vehicle.py`, or loading different tracks than the default one. As a result, we have decided to create two customized launch files capable to open our modified scripts as well. When not parameterized, both files could theoretically substitute the ETeam launch files and give their respective expected results. However, we have decided to keep them separated for the time being. The first one runs the ROS bridge and spawns the ETeam's car. The second one instantiates the track's cones.
The main difficulty in running the ROS bridge is that, essentially, it requires the core simulator to be up and running (which is signaled by the fact that the map "Town10HD_Opt" is being displayed by the simulator, when run without the "render offscreen" option). Otherwise, it will throw an exception and close itself. We have searched for the existence of some sort of flag set by the simulator once it is running so that we could have just waited for it before running the rest of the nodes, but to no success. One colleague suggested to wait for port number 2000, where the simulator places itself, to be occupied, but it was revealed to be unsuccessful when we attempted to use such procedure outside of Docker. As a result, the only way to wait for the simulator to be up and running before proceeding is recurring to `sleep` commands. The only issue of this command is that each machine is different and, as a result, it opens the simulator with a different amount of time. This is why the amount of time to sleep has been left as a parameter.
The autorun script is called `autorun.sh` and running it is as simple as calling `./autorun.sh` on a bash window. Once started, the first thing that the script will ask us is which mode to launch. The three main modes are as described in the next subsections.

5.2.1 Mode 1

Mode 1 ("run the core simulator") was, obviously, the easiest one: all it needs to do is to run the afromented `spawn_vehicle_manual_drive_nohud.launch` and `spawn_track_absolute_position.launch` files.

5.2.2 Mode 2

Mode 2 ("manual controls") turned out to be the hardest. Normally, the manual control is enabled, while using the ROS bridge, by pressing the "B" key. Simulating the pressure of such button was deemed to be too unreliable, as it can work only if the ROS bridge window is currently highlighted. It is possible to highlight the window through a script command, but the command has been found to not work in some machines. A list of preliminary solutions is the following:

Simulating a key press through pygame pygame is the library used by the ROS bridge. By simulating a key press event through such library, theoretically, it would have been detected by the ROS bridge's event manager, which would have acted accordingly. However, running separated pygame scripts has been found to be cause of crashes on some machines and, as a result, it had been deemed too risky.

Sending a ROS topic Following what has been said above about pygame, what this library does once it detects a key press is to publish a ROS topic. Such topic is, interestingly enough, detected by the same script on a second moment, which will act accordingly. As a result, we have tried to simply emulate the sending of such ROS topic. However, we were unsuccessful: although the "manual control" box of the ROS Bridge's UI gets checked once it receives such topic, suggesting that the manual control has been activated, it doesn't actually set to true the actual boolean which enables the manual control itself (which is done by pygame on a second moment)

Final solution The final solution was to manually change, inside the `carla_manual_control.py` script, the above mentioned "manual control" boolean to true at the beginning of the "main" method. This approach works. However, since we were very restraining to touch any part of the source code, we have instead created a copy of the script and placed it in the same folder (this is done at installation time by our setup script). Then, the `spawn_vehicle` launch file is simply instructed to run our version of the `carla_manual_control.py` script when requested.

5.2.3 Mode 3

Mode 3 runs, in order, the sensory, SLAM, planning and execution modules as well. All it does is to simply run the various modules on a separated window (since their outputs might be individually analyzed, especially by the data analysis group).

5.3 The recorder script

Later on, we have been tasked to extend our autorun solution with a feature which would have allowed to record the video output of the ROS bridge, as well as the output from the various modules. The issue with such task was that Carla does not have an out-of-the-box solution to record the output of the video and, as a result, we were forced to use an external library. Under the advice of the course's assistant, we have decided to use the `ffmpeg` library (which is installed by our setup file as well). However, there were some things to consider:

- The ROS Bridge's HUD had to be removed, and we needed to do so automatically

- `ffmpeg` cannot record windows but only screen rectangles. As a result, we needed to know the window's coordinates

The first issue was resolved by making yet another version of the `carla_manual_control.py` script which disabled the HUD from the very beginning. The second one was more complex, and it was eventually solved by simply making the simulator go fullscreen and record the entire screen. The ROS bridge can go full screen, once again, by modifying some parameters of the `carla_manual_control.py` script. However, there is a catch: although modifying such files modifies the ROS bridge to occupy the whole screen, it does not modify the actual size of the camera used by the ROS bridge. This results in a mostly black window with an 800x600 pixels wide rectangle on its top left corner displaying the ROS bridge output. This was solved by modifying the sensors' configuration file `objects.json`, assigning to the `rgb_front` camera sensor a resolution equal to the one of the monitor (calculated at install time by our setup script).

5.3.1 Extra features

Some of the features we deemed useful although not requested by our professor were:

Map selection we have noticed that the other groups needed to run different maps than the default one in order to try different tracks. As a result, we have thought that it would have been useful to them to have an option, in our autorun script, which allowed to choose which map to load. There are three different kinds of maps available:

- The stock maps Town01 and Town04
- An input .xodr map
- Our empty track

When the user chooses to open our empty track, we also ask him wheter he'd like to create, on the fly, a random track using our random map generator. If he accepts, the script proceeds to ask him the tracks' parameters (which can be left to the default ones by simply pressing the enter key). It then creates the track and automatically spawns the car at the beginning of the circuit, already oriented in the right position. The user doesn't have to do anything.

6 The empty track

Once we finished the tasks assigned to us, we tried to add extra features. The first thing we tried to add was an empty track, in order to allow the creation and testing of differently shaped circuits (since the default track is a simple straight line) and, possibly, reduce the computational load by running a simpler map. We tried different solutions:

6.1 OpenStreetMap and .xodr

OpenStreetMap is an online tool that allows the user to export real world street data into a .ods file. Such file extension can then be converted into a .xodr file, which is accepted by the Carla simulator. By experimenting with it, we were able to understand the .xodr file format well enough

to create a simple map made by a simple, large flat terrain with no buildings or other useless details.

The issue with this approach is that the SLAM module fails to detect the position of the car on the map, causing the `pose_stamped` topic to not be published. This was also observed by running the module on a .xodr version of the default map (hence, without buildings or textures of any kind). After a long analysis done alongside the SLAM group, we have hypothesized that this happens because the `laser_scan_matcher` module is not used to work in almost empty environments, causing a situation in which, essentially, more than 90% of the module's "rays" to go to infinity, causing an internal ROS module to throw an error. As this is deemed illegal by an internal ROS module, it's not anyone's fault. We also found the exact line which caused such error and we believe that modifying it in order to accept up to 93% of illegal rays would have solved the error without side-effects, but the line was inside a .so file in the ROS library and we were consequently unable to change it. Still, the empty track has been shown to work on rare occasions and we decided to keep it available for future uses, in case such issue would be solved.

6.2 Opt maps

Essentially, the Carla simulator has two versions of each map: a layered and a not layered version. The "layered" version is split in several layers, where each one of them contains one between buildings, trees, parked vehicles, etc. By removing all of them, the only thing left is the textured ground. We tried to create an empty space with enough stuff around it by simply removing everything around the map and then look for a large enough spot where we could have placed the cones. However, this experiment was soon discarded: not only most layered maps crashes when run alongside the ROS bridge, but the few which worked crashed when we attempted to spawn the cones. By searching for a solution, we have confirmed that this is associated to a bug in the core simulator itself and, as a result, it could not be fixed by us.

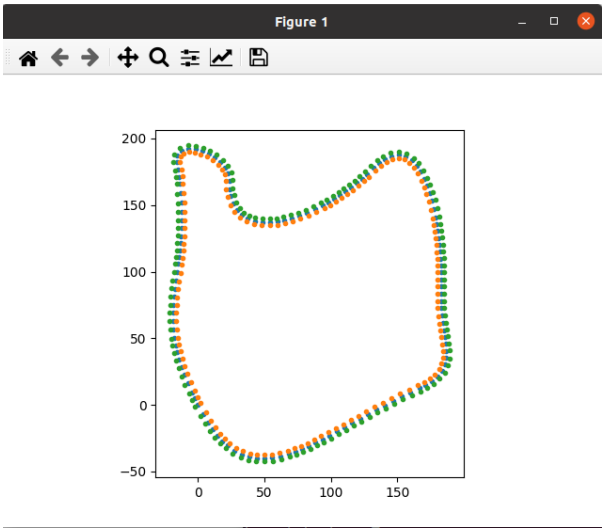
6.3 RoadRunner

RoadRunner is the official Matlab package used to create and import new maps for Carla. Our last idea was to simply create a fairly large terrain. However, RoadRunner requires a separated Matlab licence from the one we have been given by the University. We both asked Mathworks for a trial licence and the University's Matlab Licence administrator if we could get it, but we didn't received an answer from both of them and, as a result, we were unable to proceed with our experiments.

7 The circuit generator

One last thing that we wanted to add ever since the first assignment, but that we were able to do only later on, was a track generator. The idea was to generate a track in the form of a set of cones coordinates, save them into a .yaml file (which is the format used by the standard "acceleration" track) and eventually load them through the `spawn_track` launch file.
The general idea behind this script is the following:

- Using a third-party library, we create a set of points shaped like an irregular polygon. The parameters allow us to choose the number of vertices (hence, of curves) and how "smooth" they should be. Currently, there are no definitions of left/right cones



(a) A randomly generated track

(b) And its result on an empty track

- The issue with the current circuit is that both short edges and long edges have the same number of points, resulting in short straights in the track having high densities of points and the long straights to have low densities. What we want is, instead, to have a set of points equidistant between each other, at least in an approximate way. One solution is to generate, from the beginning, a lot of points, such that the distance between one point and its nearest is less than the chosen distance. Then, starting from the first one, we sum the distance between consecutive points using Euclide's theorem. When we arrive at a point such that the current sum is greater than our threshold, it means that the current point is distant (in tracks terms) that threshold from the starting position and we delete all the intermediate points. We then repeat the same process starting from the current point until we reach the last point in the circuit.

- We now need to create the left and right cones. First of all, we need to decide at which angle to place them. One good solution is that, given a point, its predecessor and its successor on the track, we can place the two cones alongside the bisector line passing from that point, dividing the angle between its predecessor and its successor in two equal parts. The bisector line equation is then calculated by translating the current point to the point (0, 0) alongside its two neighbors, and calculating the average of the angle of the two points against the origin of the system. This angle is the bisector's line angle against the origin. The equation is calculated by finding, through sine and cosine, a point of such line passing through the origin and then by calculating the line passing through both points.
  The two cones' coordinates are then found by finding the coordinates of the two points passing through the bisector line, distant a certain amount from the origin of the system (hence, our current point). The whole system is then translated back to its original coordinates and we proceed with the next point until we have calculated the cones' coordinates for the whole circuit's points.

- Finally, the data is saved into a .yaml file.

Figure 4: Town04 with our custom track initialized. The .yaml file has been created with a customized version of our track generator.

7.0.1 Generating single curves

Since the empty track cannot be currently used because of the above mentioned issues, we have been unable to test the autopilot against the tracks generated by our script, as they require large spaces. Still, we were able to help the other groups by using our circuit generator in order to create simple tracks with curves in Carla's stock circuits. The trick was, essentially, to start not from random points in order to make a shape, but from manually annotated points obtained by driving the ETeam car through the planned path on the stock circuit. Then, by carefully setting some of the generator's parameters, it was able to create the intermediate cones passing through such points, making an effective curved shaped track as requested. Such track was then used in order to test the autopilot against curves.

8 Enabling the "true low resolution" quality

The simulator offers two different quality settings: "High" and "Low". The issue is that, even if run under the "Low" setup, the simulator is often unable to run even on mid-range machines. Using the "render off screen" option doesn't solve the issue in some machines. A possible solution is to reduce even more the rendering quality, for example by setting the main simulator's resolution to 1% of the maximum quality, and render it inside a very small window (for example, a 100x100 window). Since the ROS bridge runs on a separate window, it won't affect the rendering quality of the window we actually use for this course's purposes. Such settings are to be placed inside a file named `GameUserSettings.ini`, which is inside a hidden folder placed outside of the simulator's

directory. We have created a script named `enable_true_lowres.sh` which, once launched, will append such settings inside the required file. We recommend running it only once in order to avoid unexpected behaviors.

9 Issues through the course

Our job was strongly hindered by the fact that Carla is a highly demanding program, with the minimum requirements being a GPU with at least 8GB of memory. Modifying the `GameUserSettings.ini` file once again made the task much more accessible to our lower-end machines, but even when run with the lowest possible settings, we were often subject to crashes due to overheated machines or out-of-memory crashes. A version of Carla prior to 0.9.12, would have allowed us to run the simulator using the less demanding OpenGL library. A docker version of the simulator, capable of running such simulator on Carla 0.9.11, was made by another student outside of our group. Such a version, however, did not include every single feature of the simulator, such as the ETeam's car. At the same time, we have been personally asked by a member of the ETeam to focus on a docker-less version of the simulator, which brought us to the decision to focus on such version.
Still, we believe that we have done everything possible *with what we had* (in terms of computational resources).

10 Appendix A: files created.

Below, a list of the files created by our team, their locations and how to run them. The scripts have been saved in the `automation-scripts` branch of the simulator. However, all the files which could have been inserted directly in the develop branch of the simulator (which means everything with the exception of the three customized `carla_manual_control.py` scripts and `Town_04.yaml`, which have to be placed in folders created by the `Update.sh` script) have been also placed in their destination folders.

10.1 `autorun.sh`

Our autorun script. Must be placed in the simulator's root folder.
Location:
`https://github.com/unipi-smartapp-2021/ETEAM-MIRROR-etdv_simulator/tree/develop`
How to run: `./autorun.sh -s <sleep seconds>`. The rest of the parameters are asked directly at runtime.

10.2 `autorun_setup.sh`

Our auto installation script. Must be placed in the simulator's root folder.
Location:
`https://github.com/unipi-smartapp-2021/ETEAM-MIRROR-etdv_simulator/tree/develop`
How to run: `./autorun_setup.sh`.

10.3 `update_modules.sh`

Reinstalls the autopilot modules.
Location:
`https://github.com/unipi-smartapp-2021/ETEAM-MIRROR-etdv_simulator/tree/develop`
How to run: `./update_modules.sh`.

10.4 `record_utils.sh`

Recording utilities. Used by `autorun.sh`, but it can work as a standalone script as well. The bag and video files are saved respectively in the /src/VideoRecordings and /src/Bags folder inside the simulator.
Location: `https:`
`//github.com/unipi-smartapp-2021/ETEAM-MIRROR-etdv_simulator/tree/develop/src`
How to run: `./record_utils.sh <fps>`.

10.5 `enable_true_lowres.sh`

Lowers the rendering quality of the simulator even more (the ROS Bridge's window is unaffected). Useful when even the `RenderOffScreen` option doesn't help.
Location:
`https://github.com/unipi-smartapp-2021/ETEAM-MIRROR-etdv_simulator/tree/develop`
How to run: `./enable_true_lowres.sh`.

10.6 objects_nohud.json

Sensors file used when we want to run the ROS bridge full screen. Modifies the `sensor.camera.rgb` sensor to adapt to the size of the screen (as determined by `autorun_setup.sh`).
Location: `https://github.com/unipi-smartapp-2021/ETEAM-MIRROR-etdv_simulator/tree/develop/src/config`

10.7 spawn_track_absolute_position.launch

Modified `spawn_track.launch` with the capabilities of setting the .yaml track file and path, as well as allowing to choose on whether to use the car's relative position on the track to spawn the cones or their absolute coordinates.
Location: `https://github.com/unipi-smartapp-2021/ETEAM-MIRROR-etdv_simulator/tree/develop/src/launch`
How to run: `roslaunch etdv_simulator spawn_track_absolute_position.launch track_name:=<.yaml filename> track_file:=<.yaml file path> absolute_position:=<'' for relative position, '_absolute_position' for absolute position>`

10.8 spawn_vehicle_manual_drive_nohud.launch

Modified `spawn_vehicle.launch` with also the capabilities of allowing to choose on whether to use `carla_manual_control.py` or our customized versions of such file
Location: `https://github.com/unipi-smartapp-2021/ETEAM-MIRROR-etdv_simulator/tree/develop/src/launch`
How to run: `roslaunch etdv_simulator spawn_vehicle_manual_drive_nohud.launch manual_drive:=<'' for no manual drive, '_manual_drive' for enabling it> hud:=<'' for enabling the hud, '_nohud' to disable it>`

10.9 spawn_track_absolute_position.py

Modified `spawn_track.py` file which removes two lines, enabling it to spawn the cones as defined by their the absolute position as defined by the .yaml file, instead of their relative position.
Location: `https://github.com/unipi-smartapp-2021/ETEAM-MIRROR-etdv_simulator/tree/develop/src/scripts`

10.10 carla_manual_control_manual_drive_nohud.py, carla_manual_control_nohud.py and carla_manual_control_manual_drive.py

Modified `carla_manual_control.py` files which either enables the manual controls on startup or removes the hud. Or both. Installed by the setup script.
Location: `https://github.com/unipi-smartapp-2021/ETEAM-MIRROR-etdv_simulator/tree/automation-scripts/`

10.11 EmptyTrack.xodr

Empty track, composed by a large grey plane with nothing around.
Location: https://github.com/unipi-smartapp-2021/ETEAM-MIRROR-etdv_simulator/tree/automation-scripts/

10.12 Town04.yaml

Cones definitions for the large curve in the map "Town04", used to test the planning module against a curved track. The file defines the cones'absolute position.
Location: https://github.com/unipi-smartapp-2021/ETEAM-MIRROR-etdv_simulator/tree/automation-scripts/

10.13 circuitmaker.py

Generates a random circuit according to a set of parameters.
Location: https://github.com/unipi-smartapp-2021/ETEAM-MIRROR-etdv_simulator/tree/develop/src/scripts
How to run: `python3 circuitmaker.py`. Use `python3 circuitmaker.py -h` for a full list of parameters.