UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH

Design and evaluation of virtual environments for testing Advanced Driver Assistance Systems

A Degree Thesis

Submitted to the Faculty of the

Escola Tècnica d'Enginyeria de Telecomunicació de Barcelona

Universitat Politècnica de Catalunya

by

Miguel Ángel Bueno Sánchez

In partial fulfilment

of the requirements for the degree in

TELECOMMUNICATIONS TECHNOLOGIES AND SERVICES ENGINEERING

Advisors: Isaac Agustí Ventura

Ferran Silva Martinez

Barcelona, January 2019

Abstract

The ADAS (Advanced Driver Assistance Systems) industry and development are growing very fast and new tools are needed in order to design and evaluate newly created systems. Before the tendency of the industry to rely on virtual simulations to design such technology, an evaluation of Unreal Engine 4, and CARLA (Car Learning to Act) more specifically, are proposed as software that can bring real value when developing ADAS.

The work here presented focuses on ADAS that rely on cameras, such as rear view or side view camera systems, and seeks a way of integrating virtual cameras into synthetic vehicles in the CARLA simulator despite its limitations.

Open source software is explored for future development of other systems. Also, new ways of designing ADAS systems arise. Engineers will now have material to work with at predevelopment stages, even with no tangible cameras or cars are available. Working in virtual environments allows to model ADAS systems accurately with a unique resource: a computer.

Resum

La indústria d'ADAS (Sistemes Avançats d'Ajuda al Conductor) i el seu desenvolupament han experimentat un creixement exponencial, de manera que noves eines són necessàries per a poder dissenyar i evaluar sistemes d'aquesta mena. Davant la tendència de la indústria a fer servir cada cop més simulacions de carácter virtual per a dissenyar la tecnologia anteriorment esmentada, es proposa l'evaluació d'Unreal Engine 4 i, més concretament, de l'entorn de simulació CARLA (Vehicle aprenent a actuar), com a programari que pot resultar de gran utilitat de cara a desenvolupar ADAS.

La tesis se centra principalment en ADAS que usen càmeres, com càmeres de visió posterior o sistemes digitals com a retrovisors, i es focalitzen tots els esforços en integrar càmeres virtuals en vehicles sintètics dins l'entorn CARLA fent front a totes les limitacions que pugui haver-hi.

S'explora programari lliure per al desenvolupament futur d'altres sistemes. S'obren també noves vies de dissenyar ADAS, ja que els enginyers podran començar a treballar amb material virtual en etapes molt inicials dels projectes, quan ni tan sols es disposa de càmeres o vehicles reals disponibles. A més, per a dur a terme tot això tan sols es necessita un ordinador.

Resumen

La industria ADAS (Sistemas Avanzados de Ayuda al Conductor) y su desarrollo han experimentado un crecimiento exponencial, de manera que nuevas herramientas son necesarias para poder diseñar y evaluar sistemas de dicha índole. Ante la tendencia de la industria a usar cada vez más simulaciones virtuales para diseñar la tecnología anteriormente comentada, se propone la evaluación de Unreal Engine 4 y, más concretamente, del entorno de simulación CARLA (Vehículo aprendiendo a actuar), como programas que podrían resultar de gran utilidad de cara a desarrollar ADAS.

La tesis se centra principalmente en ADAS que usan cámaras, como sistemas de visión trasera o sistemas digitales a modo de retrovisores, y se focalizan todos los esfuerzos en integrar cámaras virtuales en vehículos sintéticos dentro del entorno CARLA pese a las limitaciones que se puedan encontrar.

Se exploran programas libres para el desarrollo futuro de otros sistemas. A la vez, se abren nuevas vías de diseño ADAS, pues los ingenieros tendrán la capacidad de empezar a trabajar con material virtual en etapas muy iniciales de los proyectos, cuando no se disponen de muestras de cámaras o vehículos reales. Además, todo ello se puede llevar a cabo con muy poco material: un ordenador.

Dedicated to my family and friends.

Special mention to Isaac Agustí and Ferran Silva, who helped me a lot to carry out this work.

Acknowledgements

The thesis was proposed by Isaac Agustí, who has been supervising day after day the work done. Most of the technical aspects on this thesis are thanks to him.

There has also been great advice and support from Ferran Silva, who concisely explained everything one need to know to accomplish this work.

This work has been financed and conducted at FICOSA ADAS SL.

Revision history and approval record

Revision	Date	Purpose
0	22/12/2018	Document creation
1	25/12/2018	Added draft introduction, abstract and acknowledgements
2	04/01/2019	Added budget, environment impact, state of the art and draft methodology
3	08/01/2019	Added list of figures, list of tables and glossary
4	10/01/2019	Added bibliography and appendices A, B, C and D
5	16/01/2019	Restructured document and added results and next steps

DOCUMENT DISTRIBUTION LIST

Name	e-mail
Miguel Ángel Bueno Sánchez	miguelangelbuenos@gmail.com
Isaac Agustí Ventura	isaac.agusti@ficosa.com
Ferran Silva Martínez	ferran.silva@etsetb.upc.edu

Written by: Miguel Ángel Bueno Sánchez		Reviewed and approved by: Isaac Agustí Ventura and Ferran Silva Martínez	
Date	02/01/2019	Date	24/01/2019
Name	Miguel Ángel Bueno Sánchez	Name	Isaac Agustí Ventura and Ferran Silva Martínez
Position	Project Author	Position	Project Supervisors

Table of contents

List of Figures

List of Tables

1. Introduction

Advanced Driver Assistance Systems –ADAS for short- are increasingly being incorporated into modern vehicles in order to evolve to the next automobiles generation. They basically help the driver to be more effective thanks to technology.

FICOSA[1] is currently working to improve such systems. This task requires new design tools and technology. This project is part of the development process of the company and is focused on finding and exploring a virtual simulation environment that allows the study and design of ADAS based on cameras.

Specifically, this work is based on CARLA[2], which is a project/layer built on Unreal Engine 4[3], a powerful graphic engine very used in the industry as a tool to develop autonomous cars.

FICOSA is developing three main camera based products: Rear View Systems (RVS) that help drivers to park a car or avoid obstacles at the rear of the vehicle thanks to the vision it grants; first generation of Camera Monitor Systems[4] (CMS), which are mounted on cars replacing traditional external side mirrors by cameras and Surround View Systems (SVS), which allow the driver to have a bird view of the vehicle thanks to the vision four cameras strategically placed on the bodywork of the vehicle grant.

The main purpose of the thesis is to integrate virtual cameras in a virtual environment that resembles as much to reality as possible so that further study and tuning can be performed in order to have material to start designing ADAS even before real cameras samples are available. This work focuses on the design of a fisheye camera in CARLA environment, which are not available by default due to limitations on the simulator.

Static simulations involving cars and cameras are already being performed with a software combination of Blender[5] and MATLAB[6]. This thesis goes to the next level applying dynamism to the environment. This allows a more precise study of the qualities of the cameras and results closer to the final product.

In the next sections it is explained what makes a camera different than others (even if they are equal on the paper) and how do we simulate such cameras to get captured images very close to the images a real camera would take.

To sum up, this thesis focuses on integrating virtual cameras in a dynamic environment held in CARLA –and therefore in Unreal Engine 4- in order to obtain captures of the synthetic world with the same characteristics of a real camera. The captured frames are meant to serve FICOSA engineers as a starting point to design ADAS systems based on cameras. Such frames provide information of mechanical interference provoked by the vehicle integration, field of view, regulation compliance, etc.

Next chapters go deeper into all the above explained.

[1] FICOSA International S.A., https://www.ficosa.com/
[2] Official website: http://carla.org/
[3] More information at its website: https://www.unrealengine.com
[4] Further information on CMS: http://www.imatest.com/solutions/camera-monitor-systems/
[5] Official website: https://www.blender.org/
[6] Proprietary software of The Mathworks Inc., https://es.mathworks.com/products/matlab.html

1.1. Image Engineering simulations tasks

There is a sub division in the Image Engineering team with the mission of performing different kind of simulations to check whether the integration of cameras into vehicles is valid taking into account field of view regulation and customers' specifications.

Workflow is the following:

1. Customer sends vehicle information as CAD to FICOSA.
2. Image Engineering team runs simulations with Blender. Such simulations consist in placing virtual cameras on the vehicle as specified by the client. Output images are taken from the cameras' perspective. Usually, each camera sees part of the vehicle's bodywork and part of a synthetic world that helps when analysing the results.
3. Depending on the camera type, the output images can have narrow angle (pinhole model) or wide angle (fisheye) lens projection. In case of fisheye, the equisolid lens projection is simulated in Blender because it allows to increase the field of view up to 360 degrees and has a known lens distortion.
   As the output synthetic images have a known distortion, they are distorted again according to the custom lens that is used for the project. That can be done with a specific MATLAB that the company owns.

Lens distortion, field of view, interferences caused by vehicle, blind spots and other parameters are evaluated with the outcome of the mentioned static simulations. Also, everything must fulfil regulation such as FMVSS111, ISO 16505, ECE R46, etc.

All this work is carried out with static simulation environments. This thesis brings onto the table a new simulation environment (CARLA) that introduces dynamism thanks to Unreal Engine 4. Such dynamism can be very helpful when designing ADAS.

For instance, it allow to see in a somewhat realistic way how a camera performs in a crossroad. Also, outputted video can be used to replace videos recorded by real cameras when they are not available yet at early stages of the project.

To sum up, the aim of this work is to gather all points above presented in a unique tool that introduces new features to help designing and evaluating ADAS.

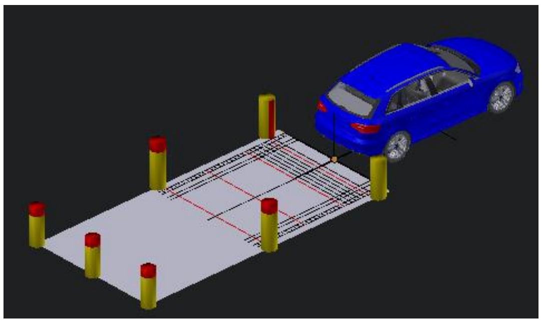


Figure 1. Blender simulation environment from a perspective view.

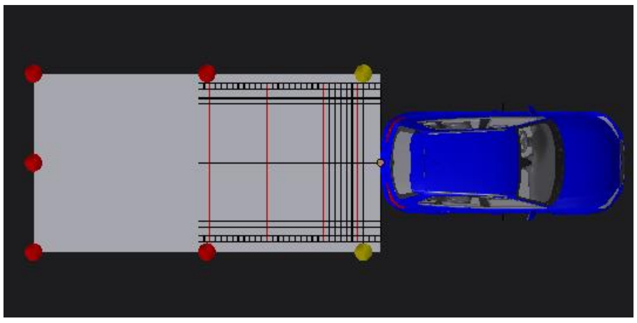


Figure 2. Blender simulation environment top view.

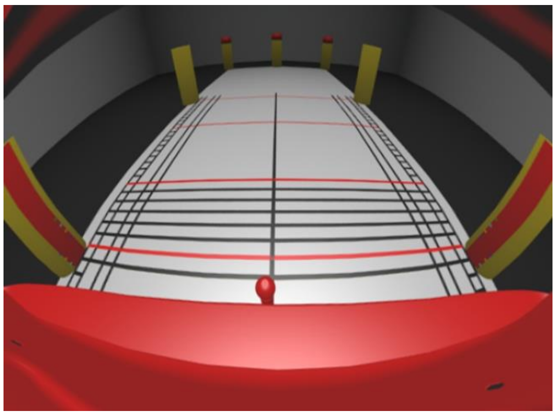


Figure 3. Result of a RVS simulation in Blender.

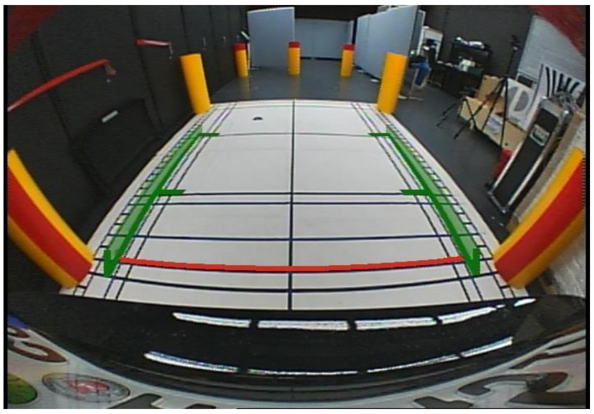


Figure 4. RVS mounted on a real vehicle.

1.2. <u>Unreal Engine 4</u>

Unreal Engine 4 –UE4 from now on- is a graphic engine developed by Epic Games[7] very used in the videogames world. It is also increasingly being used in the autonomous driving industry as a tool to design and test ADAS and AI algorithms. The main reason is its real-time technology, which allows to perform high fidelity synthetic simulations.

It is written entirely in C++ and is served as a way of applying realistic physics and dynamism to static meshes and virtual worlds.

In UE4 everything are C++ objects, meaning everything belongs to a class. That means that in order to import something into the virtual world a static mesh must be imported and then a C++ class generated. The class is what gives functionality to the object. This gives a lot of personalization possibilities and even allows to modify the core elements of the engine itself, although some code has restricted access due to security and stability reasons.

The natural workflow for someone to create a videogame or to create simulation environments starts by creating the elements of the world as static meshes in modelling

[7] Further information of the company can be found at https://www.epicgames.com/

software such as Blender or Maya. Then, the models are imported into the UE4 platform (see Figure 5). After that, it allows to apply different materials and physics to the meshes to bring them into life with a realistic look. Models can also be programmed by generating C++ classes or using and internal visual scripting language called Blueprints (see example on Figure 6).

UE4 allows the designer to give basic functionality to agents, but very complex models can be programmed. A whole AI system, for instance. Control over agents can be passed to the user as if it was a videogame. Even vehicles can be imported with realistic physics.

There are also other elements available, like cameras, which are very helpful for the purpose of this work. It may seem trivial to integrate cameras into the environment mounted on a moving vehicle to record the surrounding virtual world. However, there is a big limitation in UE4 regarding cameras: they can be configured in a unique lens projection (rectilinear or pin-hole camera) with a limited field of view up to 170 degrees. ADAS applications that FICOSA is currently developing demand wide angle fisheye lenses with fields of view way higher than 170. Besides that, each lens is unique in the distortion it introduces in the generated image. It is proposed a way to increase the field of view of cameras and to generate a correct distortion according to the models used in the company for every recorded image.

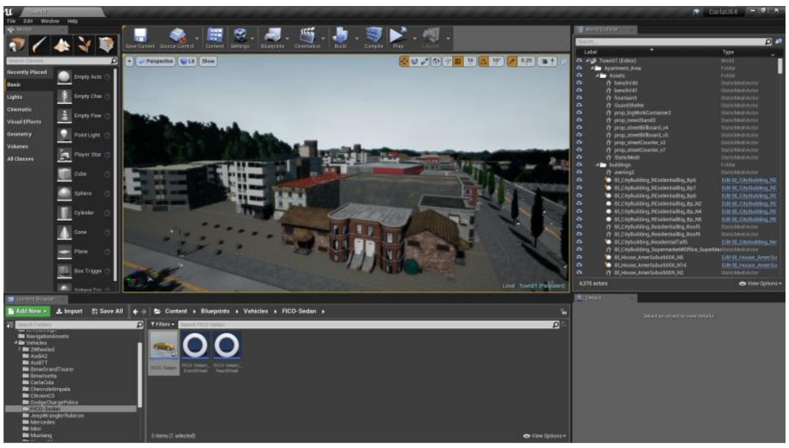
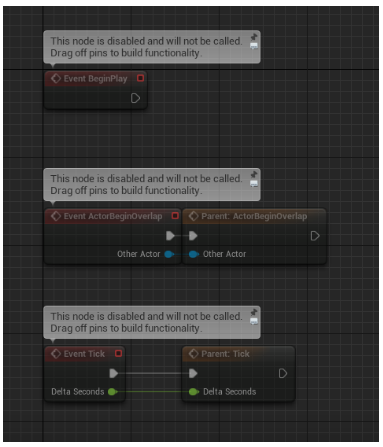


Figure 5. Unreal Engine 4 Main User Interface.


This node is disabled and will not be called.
Drag off pins to build functionality.

This node is disabled and will not be called.
Drag off pins to build functionality.

This node is disabled and will not be called.
Drag off pins to build functionality.

Figure 6. Blueprint.

1.3. CARLA

Car Learning to Act (CARLA) is an open-source urban simulator developed at the Computer Vision Center[8] (CVC) that allows the design, study, evaluation and validation of trained autonomous agents. It is being developed as a free alternative for the autonomous driving research.

It allows to test AI algorithms and ADAS with no cost nor manpower. When designing and tuning ADAS based on cameras, the systems must be integrated on real cars when available in order to perform road tests. Such tests consist in acquiring images with the

[8] Established by Universitat Autònoma de Barcelona. More information: http://www.cvc.uab.cat/

cameras under different conditions to see if they perform as expected in each one of the case scenarios. They must be tested under poor light conditions, different weathers, etc. Many of these cases are easy to cover, but some other are harder. Consider for instance a test case where it needs to be rainy under low light conditions. Besides, there is a large investment in getting the car and set it up, as well as in manpower.

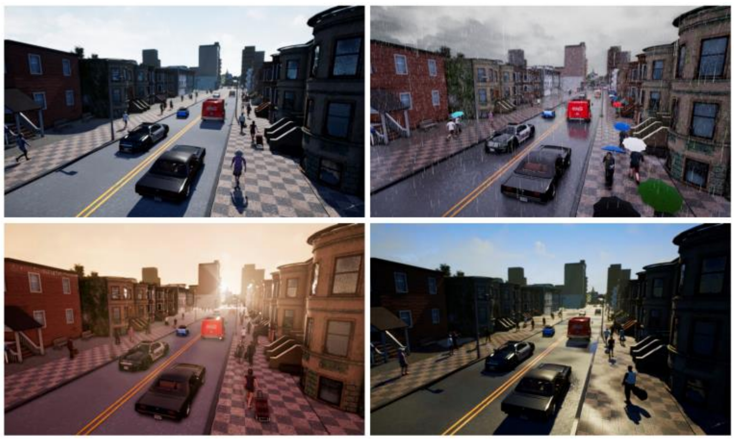


Figure 7. Spectator images of CARLA environment.

Obviously, all these tests must be done even if CARLA is used, because a simulation environment will never be a validation tool. However, it eases the task of designing the whole system in the first place avoiding possible future loops in the project regarding cameras design, which includes intrinsic camera parameters, integration position on vehicle and others.

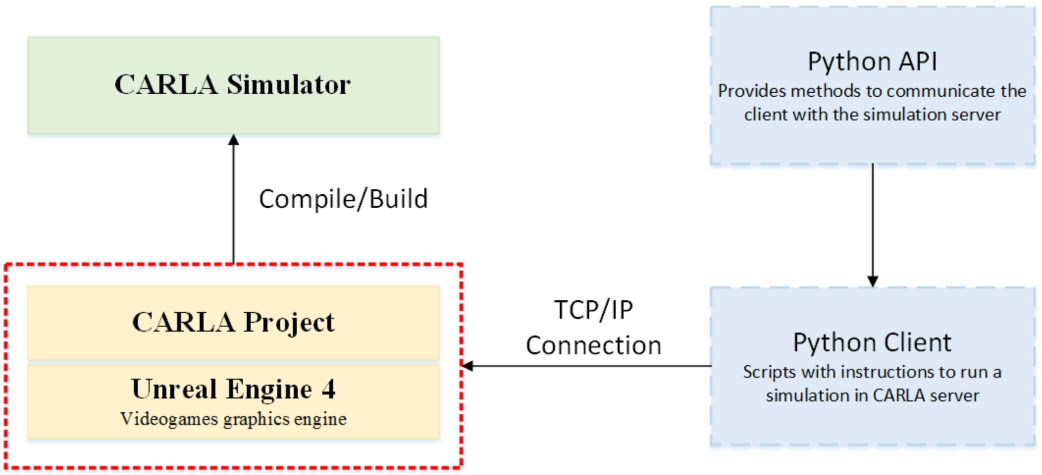
CARLA is built as a project in Unreal Engine 4 and it includes a Python API that allows scripting to manage the behaviour of the different agents present in the environment. It consists on two virtual villages with pedestrians and cars moving around randomly. One can integrate and deploy its own vehicle and attach sensors to it. That way, that vehicle or controlled agent will retrieve data of the world at each time step of the simulation. There are various available sensors: LIDAR, depth cameras and RGB cameras. For this particular project we are interested on the third one and specifically on the RGB images it captures.

What makes CARLA special is that it provides an already created environment to directly start working with and a Python API and file system that allows the user to change the settings of the simulation and tweaking UE4 parameters without even touching UE4. The Python API is directly connected to UE4 and scripting in Python translates into modifications in the engine configuration as well as generation of new C++ code to give functionality to agents. It makes the whole process of configuring and running a simulation much easy and direct.

CARLA simulator acts as a server when running and the user has the possibility of connecting a client running on Python via TCP/IP protocol[9]. If the simulator is launched it freezes until a Python script derived from the CARLA Python API tries to connect as a client. Then, the simulator follows the guidelines proposed by the script and adapts its settings according to it. Such settings include weather settings, agents spawn location, agents' behaviour, etc.

[9] More information: https://en.wikipedia.org/wiki/Internet_protocol_suite

There is also the possibility to launch a simulation in standalone mode. That way, CARLA does not freeze when initializing and the user can control the agents with the keyboard, as in a videogame.



CARLA Simulator
Compile/Build
CARLA Project
Unreal Engine 4
Videogames graphics engine
TCP/IP Connection
Python API
Provides methods to communicate the client with the simulation server
Python Client
Scripts with instructions to run a simulation in CARLA server

Figure 8. CARLA's structure.

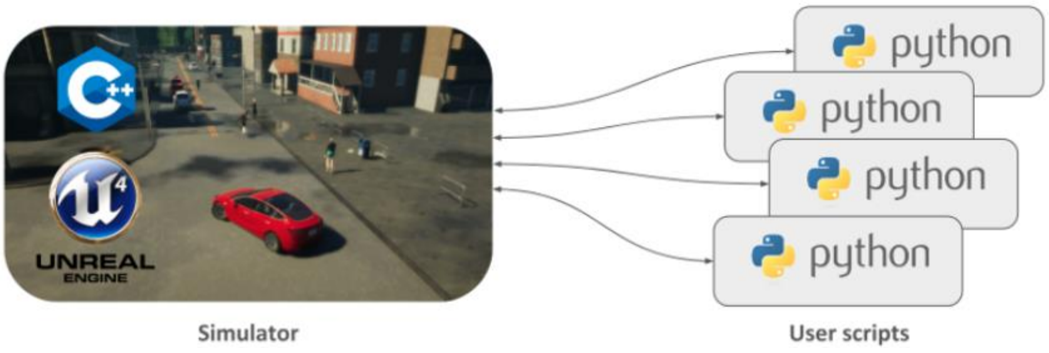


Figure 9. Example of how Unreal Engine 4 and CARLA interact.

1.4. <u>Requirements</u>

For the sake of clarity of next sections and to be able to do some maths in the future, some characteristics of lens and sensor already being used in a FICOSA system out in the market are listed.

Note that lens and sensor must always be specified. It makes no sense to just give information of one of them because the final processed image of a camera depends of its combination. Same lens with different sensors can produce different outputs.

Virtual cameras can and must be modelled with specifications listed below. Also, the simulation environment must be capable of retrieving data such as videos and images according to requirements, which are the ones real FICOSA systems have.

Lens data:

- Name: Sunny AT102A
- Effective focal length (EFL): 0.948 mm
- Entrance pupil (EP): 0.948 mm
- Horizontal field of view (HFOV): 196 degrees
- Vertical field of view (VFOV): 150 degrees

Sensor data:

- Name: On Semi AR0143
- Resolution: 1344 x 968 pixels
- Pixel size: 0.003 mm
- Array size: 4.032 x 2.904 mm

Video quality:

- Frame rate: 60 frames per second
- Colour depth: 8 bits (3 colour channels RGB)

Camera data:

- Extrinsic parameters[10]

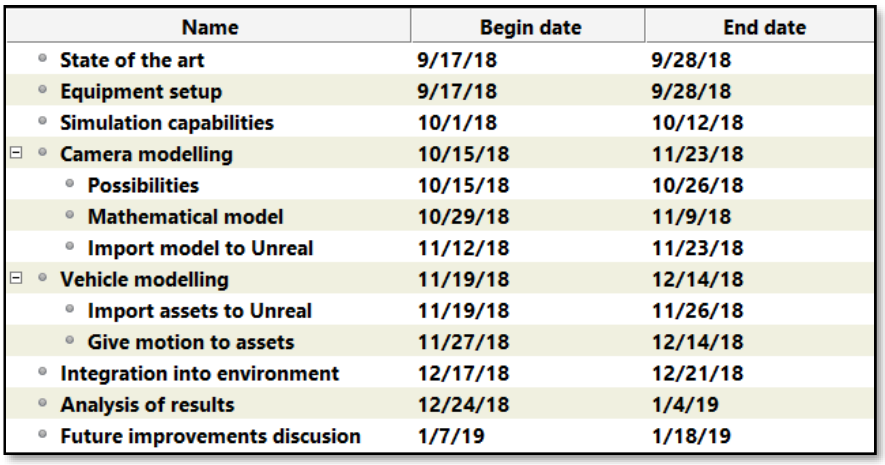
It is also interesting to obtain CAN data from the simulations:

- Speed
- Individual wheel speed
- Gear information

[10] Position of cameras from an origin reference point on vehicle.

1.5. Work plan

Proposed Gantt diagram



Name	Begin date	End date
State of the art	9/17/18	9/28/18
Equipment setup	9/17/18	9/28/18
Simulation capabilities	10/1/18	10/12/18
Camera modelling	10/15/18	11/23/18
Possibilities	10/15/18	10/26/18
Mathematical model	10/29/18	11/9/18
Import model to Unreal	11/12/18	11/23/18
Vehicle modelling	11/19/18	12/14/18
Import assets to Unreal	11/19/18	11/26/18
Give motion to assets	11/27/18	12/14/18
Integration into environment	12/17/18	12/21/18
Analysis of results	12/24/18	1/4/19
Future improvements discusion	1/7/19	1/18/19

Figure 10. Planned Gantt diagram work plan.

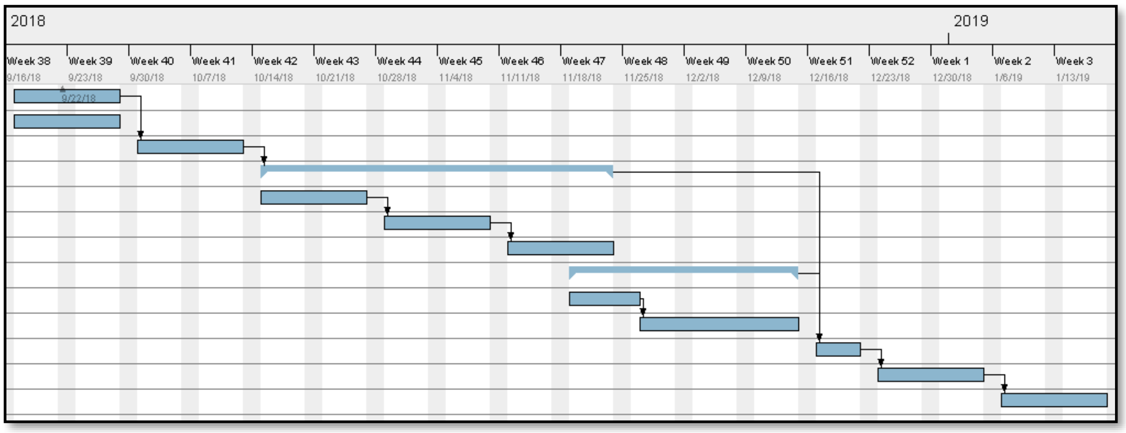
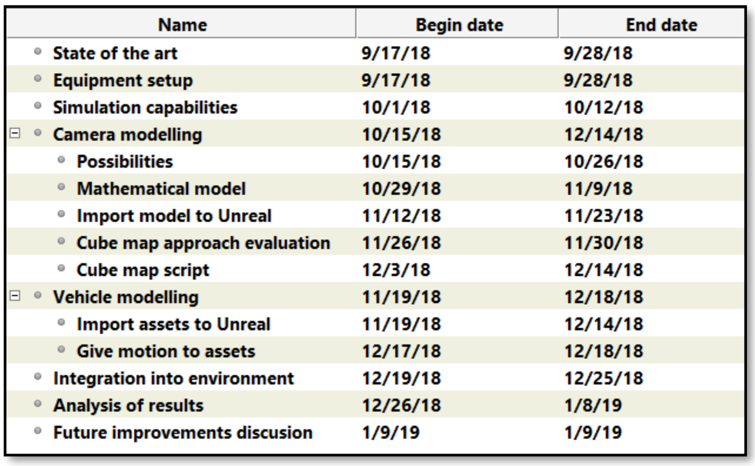


Figure 11. Planned Gantt diagram timeline.

Real Gantt diagram



Name	Begin date	End date
State of the art	9/17/18	9/28/18
Equipment setup	9/17/18	9/28/18
Simulation capabilities	10/1/18	10/12/18
Camera modelling	10/15/18	12/14/18
Possibilities	10/15/18	10/26/18
Mathematical model	10/29/18	11/9/18
Import model to Unreal	11/12/18	11/23/18
Cube map approach evaluation	11/26/18	11/30/18
Cube map script	12/3/18	12/14/18
Vehicle modelling	11/19/18	12/18/18
Import assets to Unreal	11/19/18	12/14/18
Give motion to assets	12/17/18	12/18/18
Integration into environment	12/19/18	12/25/18
Analysis of results	12/26/18	1/8/19
Future improvements discusion	1/9/19	1/9/19

Figure 12. Real Gantt diagram work plan.

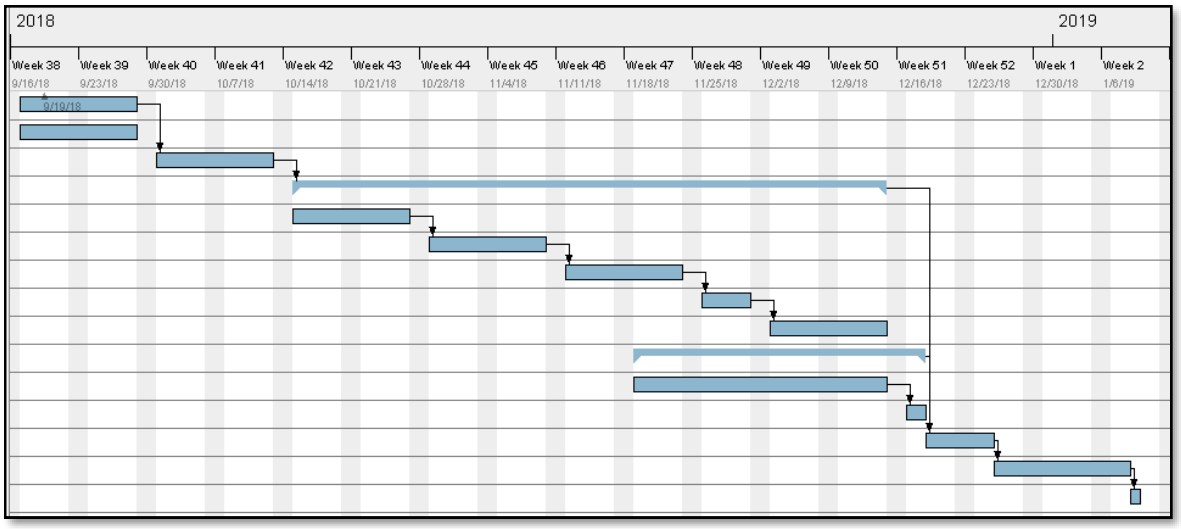


Figure 13. Real Gantt diagram timeline.

Some tasks such as camera modelling and vehicle integration have spanned more than was initially expected due to limitations encountered in Unreal Engine 4 and CARLA. For instance, different approaches had to be tested in order to obtain a fisheye virtual camera model to obtain the desired result. Before such situation, importing vehicles into the environment tasks were started so that they could be finished once the camera issue was resolved.

Initial idea consisted in modifying Unreal Engine 4 so that camera's maximum field of view could be increased programmatically at low level by modifying the code of the engine itself. After a lot of research and navigating through the engine's source code, I found out that there was actually plenty of people working on the issue and that it is not trivial to find an optimal solution.

Therefore, I designed a new way of modelling the camera: using an orthographic camera and a mirror ball right in front of it so that the camera captures the reflections of the environment on the ball. Although I did the entire math in order to set up the system, Unreal was a limitation on this approach. Reflections are captured at a very low resolution in order to not compromise performance and the orthographic camera is somehow bugged and does not work properly.

Eventually, I decided to implement another solution consisting in generating a cube map on the simulator using six perspective cameras properly placed and rotated. Then, do pixel transformations on the resulting images to obtain the desired lens distortion.

2. State of the art of the technology used or applied in this thesis:

The fast growth of ADAS in the last few years has caused the creation of some other simulators with common features that allow autonomous driving testing, cameras car integration, simulating of light conditions, etc.

Each simulator is different though and has strengths and weaknesses. The reason why CARLA has been chosen over its competitors is its realism and high quality textures, which allows to have a final result very close to reality. Also, the fact that it is open-source and is in constant development make this option the better choice.

2.1. CarMaker[11]

Simulator by IPG that excels in its low computational load so that real time simulations can be carried out on almost any decent device. This characteristic sacrifices graphics quality though. It is actually pretty poor: flat terrains and buildings built with very basic geometry with applied textures.

Although the graphics limitations, the environment is very customisable and a lot of real life situations can be reproduced. It is very simple to generate new road patterns and all aspects and characteristics of vehicles can be changed as desired.

There are plenty of sensors to use with several parameters to customize and fisheye lenses are supported.

Another good aspect of the simulator is its ability create visualizations of the data gathered by the sensors attached to the vehicle as well as basic information of the car's performance such as speed, acceleration, etc. These visualizations can be generated on the web or connecting the simulator to MATLAB Simulink[12].

Overall is a very good simulator, but it lacks the graphics quality that other simulators offer.

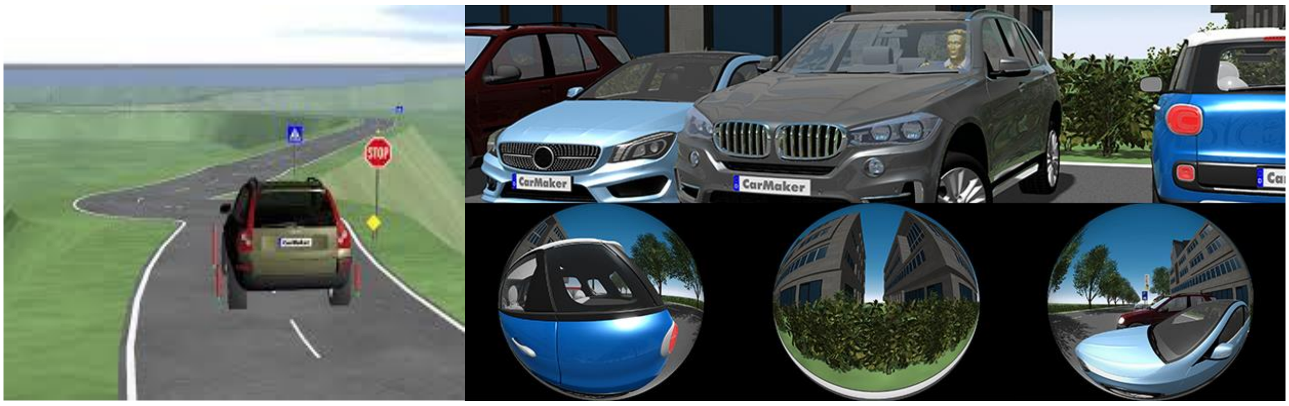


Figure 14. Car Maker simulation images.

[11] By IPG Automotive, https://ipg-automotive.com/products-services/simulation-software/carmaker/
[12] Plug-in that eases the task of simulating complicated systems using MATLAB. More information on its website: https://es.mathworks.com/products/simulink.html

2.2. AirSim

A simulator developed by Microsoft[13]. It is also based on Unreal Engine 4 and has a Python API, so it is very similar to CARLA in its structure. Graphics quality is very similar because they are both built on the same platform.

It presents some drawbacks though. There does not exist a fixed time step simulation option, which basically means that simulations must run at real time and may not be perfectly accurate. If the machine running the simulation has not enough computational power to move the simulation at real time, some frames and information are lost because the system gets overloaded.

Also, AI for AirSim[14] is a little weaker and no red traffic lights are present in the environment, making que simulation less realistic and reducing the number of possible test case scenarios. CARLA has a most robust AI system and more realistic weather settings, as well as interfaces to tweak them as desired.

On the other hand, AirSim has a better map quality with more variate situations.

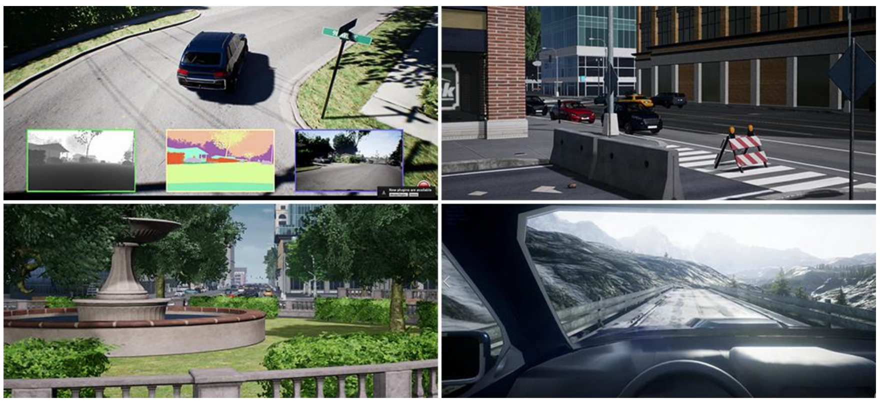


Figure 15. AirSim's different scenarios.

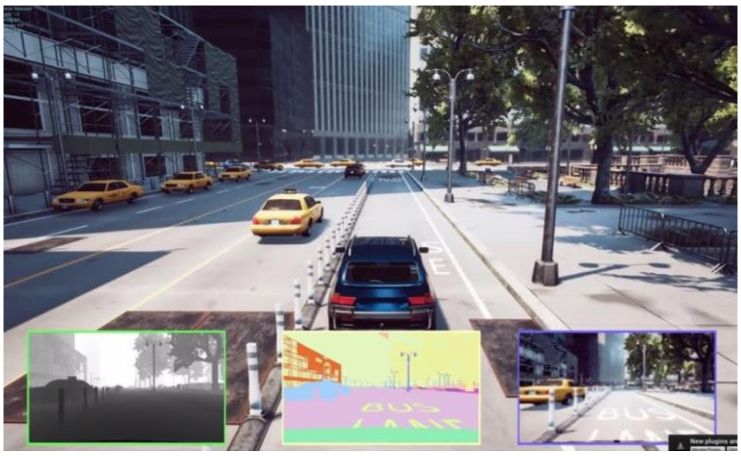


Figure 16. AirSim picture of a running simulation.

[13] https://www.microsoft.com/es-es/
[14] Official repository of the project: https://github.com/Microsoft/AirSim

2.3. <u>MATLAB</u>

Mathworks is working hard to build applications that satisfy the necessities of ADAS and AI research. They have on the market some software packages that allow testing systems based on sensor attached to cars.

Main problems are that such software is at very initial stages. Although a good variety of different sensors can be implemented, like LIDAR, radar and RGB cameras they are not configurable, so they can only be used for general purpose work. No specific lenses or radars can be tested, so fisheye lenses are not supported. Also, maps are too simple. They are mostly a single road with boxes that model cars.

The company is currently starting to use Unreal Engine 4 as a base layer for its software, connecting MATLAB to it. However, they are way behind CARLA or AirSim, which have already been developed in Unreal Engine 4 for years.

Advantages are that MATLAB allows an easy creation of neural networks and AI systems that work on the cloud and can convert MATLAB code to CUDA code for better performance on GPUs. This is not useful for camera based ADAS though.

Also, the price must be considered. Standard MATLAB version costs 2000€ and a lot of toolboxes such as Simulink must be added in order to properly use all the features above explained.

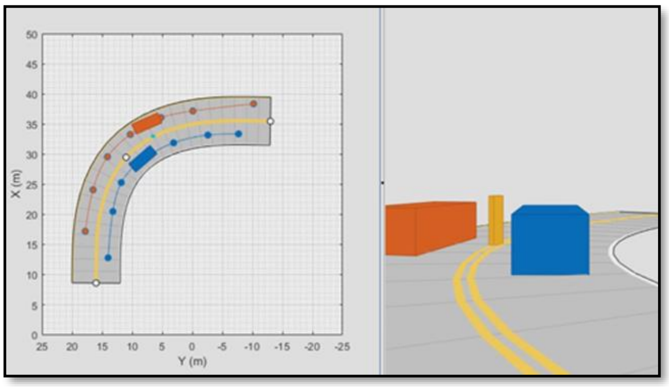


Figure 17. MATLAB tool for simulating automotive environments.

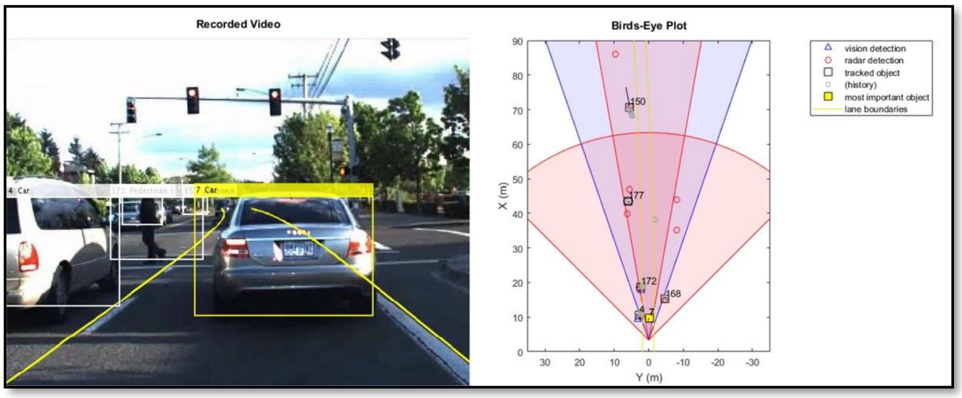


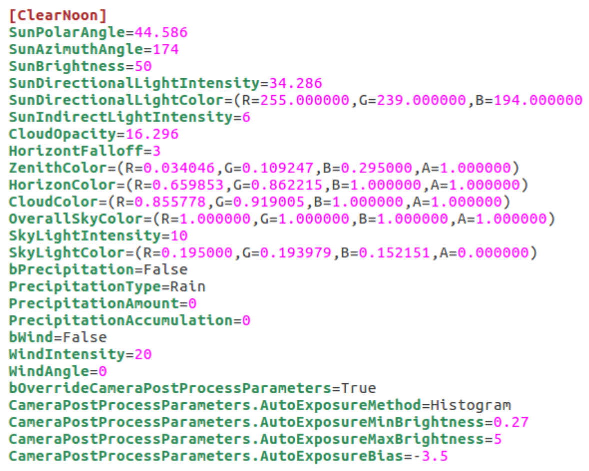
Figure 18. MATLAB autonomous tagging and detection toolbox.

2.4. CARLA

CARLA, just like AirSim, has high quality graphics thanks to Unreal Engine 4. Also, the big community behind it and a strong development team allow for periodic updates that bring many new important features. Also, it has the advantage of being open source and free to use, which were the downsides of CarMaker and MATLAB.

Remarkable things CARLA has are the variety of scenarios, pedestrians, car models and environment conditions that are available. For instance, the following are some weather pre-sets that can be selected for a simulation: clear noon, cloudy noon, soft rain sunset, hard rain noon, clear sunset, etc.

The above mentioned are all the available weather configurations for CARLA v0.8.4, the version this thesis is based on. However, more can be generated manually by personalizing the CarlaWeather.ini, a file included in the CARLA project that allows tweaking all parameters that make a weather configuration.



```
[ClearNoon]
SunPolarAngle=44.586
SunAzimuthAngle=174
SunBrightness=50
SunDirectionalLightIntensity=34.286
SunDirectionalLightColor=(R=255.000000,G=239.000000,B=194.000000
SunIndirectLightIntensity=6
CloudOpacity=16.296
HorizontFalloff=3
ZenithColor=(R=0.034046,G=0.109247,B=0.295000,A=1.000000)
HorizonColor=(R=0.659853,G=0.862215,B=1.000000,A=1.000000)
CloudColor=(R=0.855778,G=0.919005,B=1.000000,A=1.000000)
OverallSkyColor=(R=1.000000,G=1.000000,B=1.000000,A=1.000000)
SkyLightIntensity=10
SkyLightColor=(R=0.195000,G=0.193979,B=0.152151,A=0.000000)
bPrecipitation=False
PrecipitationType=Rain
PrecipitationAmount=0
PrecipitationAccumulation=0
bWind=False
WindIntensity=20
WindAngle=0
bOverrideCameraPostProcessParameters=True
CameraPostProcessParameters.AutoExposureMethod=Histogram
CameraPostProcessParameters.AutoExposureMinBrightness=0.27
CameraPostProcessParameters.AutoExposureMaxBrightness=5
CameraPostProcessParameters.AutoExposureBias=-3.5
```

Figure 19. Weather configuration parameters available to customize CARLA simulation environment.

CARLA outputs some CAN (Controller Area Network) data from the simulations along with recorded images. For each frame, parameters such as vehicle speed, acceleration and steering are stored. Although limited, it is a possibility all other simulators in this section do not offer.

There exists also the possibility of running simulations in variable time step (default) or fixed time step. The first option makes the simulator try to keep up to real time. At each frame, CARLA outputs CAN data, images and information stored by attached sensors. That means that CARLA has a finite amount of time to perform all this calculations. Consequences are that some information is lost if that time is not enough. On the other site, fixed time steps simulations allow CARLA to take all the needed time to compute all information for each frame. That way, the simulation does not run on runtime but outputs data with more accuracy. Fixed time steps simulations are ideal for physics and accuracy simulations. This feature is a great advantage of CARLA because AirSim does not support it.

As mentioned before, CARLA is an Unreal Engine 4 project, which means that has all the limitations the former has. After all, agents and scenarios that appear in the simulator have been created with modelling software such as Blender separately and then imported into Unreal Engine 4. What the graphic engine does is providing a base on which to join all models and give functionality to agents.

One of the most severe limitations of CARLA is that it only supports pinhole cameras up to 170 degrees of field of view. When a perspective/rectilinear image is generated (as in CARLA), field of view cannot have a too high value. Otherwise, resulting image will be distorted in a way that it will look like objects are moved towards the center of the image and inwards along the plane's normal. That causes a lot of information of the image to be concentred at the center of the image with low pixel density. If one tries to resize or undistort the picture will find out that poor resolution is seen at those zones. See Figure 20 for a graphic description of the effect on a CARLA generated image.

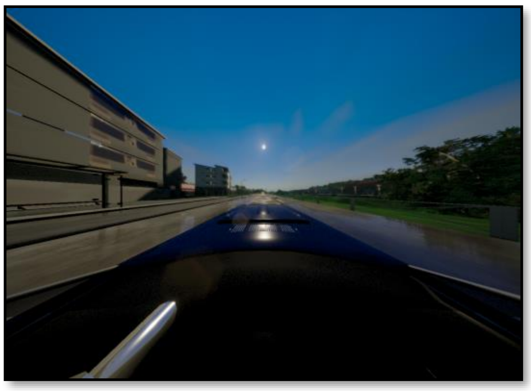


Figure 20. Image with a field of view of 170 degrees in CARLA.

Other drawbacks are the no capability of changing weather and other settings of the simulation on the run. CARLA generates simulations organized in episodes. Each episode has a determined and finite number of frames or time steps and a unique behaviour of the environment and other non-controlled agents governed by a seed. The configuration of each episode can be scripted thanks to the Python API, which connects to Unreal Engine 4.

3. Methodology / project development:

3.1. Environment set-up

The first step to evaluate Unreal Engine 4 and CARLA as simulation tools is actually installing and setting them up. Although this could look like a trivial task, the truth is a lot of problems raised when performing it.

Linux, and more concretely Ubuntu 16.04 LTS, is the operating system of the computer where Unreal Engine 4 and CARLA are installed. That is, because CARLA performs better on Linux and is less likely to crash than in Windows.

CARLA is used under the 0.8.4 version. Although there exist newer ones under development this is the stable one. Such version runs on Unreal Engine 4.18. Each version of the project runs on different UE4 versions and there is generally no cross compatibility. This happens because UE4 evolves very quickly and some things are deprecated from one version to another. That is why the correct version of Unreal Engine 4 must be used with each CARLA version in order to everything work properly and avoid crashes.

3.1.2 Build Unreal Engine 4 from source

First of all, Unreal Engine 4 must be installed on the machine. In order to do that, a GitHub (a very used cloud repository) account and an Epic Games (the developing team behind UE4) account must be created. Once done this, the GitHub account must be connected to the Epic Games account through the Epic Games profile. Unreal Engine 4 repositories at GitHub are private. That is the reason why this connection must be done. Otherwise, it will be impossible to the user to download UE4 source code.

Building Unreal Engine 4 from source is frankly easy. Just a couple of commands must be submitted on a terminal session:

```
git clone -b 4.18 https://github.com/EpicGames/UnrealEngine.git
cd UnrealEngine
./Setup.sh
./GenerateProjectFiles.sh
make
```

It can take quite some time though to complete the whole process depending on the computer it is being installed.

3.1.3 Build CARLA from source

Once Unreal Engine 4 is installed, CARLA can be downloaded. Before that, some dependencies must be installed on the computer so that the process completes successfully:

```
sudo apt-get install build-essential clang-3.9 git cmake ninja-build
python3-requests python-dev tzdata sed curl wget unzip autoconf
```

The command above installs necessary tools such as clang-3.9 (a compiler), git (control version software), Python 3, etc.

Dependencies and all required project files can be downloaded from GitHub with the following command:

```
git clone https://github.com/carla-simulator/carla
```

A script must be run inside the new CARLA folder:

```
./Setup.sh
```

It will download all necessary assets for the virtual world from the cloud and will set everything up so that they can be integrated into Unreal Engine 4. Take into account that the package used is of size 3GB. In order to accelerate the whole process, the `–jobs=8` flag can be appended to the *Setup.sh* command to take advantage of multi-threading.

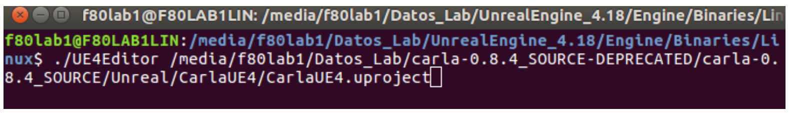
As a last step, indicating where UE4 is installed and running a building script is required:

```
UE4_ROOT=~/UnrealEngine_4.18 ./Rebuild.sh
```

Although the whole process of setting CARLA takes time to complete, it is only needed to execute a bunch of commands. However, just one tiny connection error with the web server when downloading or a non-compatibility between the dependencies installed required to compile can lead to a lot of different errors that state the non-triviality of the task.
In my case, many connection problems raised due to proxy settings at FICOSA. Also, I had to install external compilers in order to carry out the compiling process because of incompatibilities on the computer.

3.1.4 Opening up the project and the simulator

Once Unreal Engine 4 and CARLA are properly built one can load the second into the graphics engine. To do so, one must navigate to UnrealEngine_4.18/Engine/Binaries/Linux and enter the following command:



```
f80lab1@F80LAB1LIN: /media/f80lab1/Datos_Lab/UnrealEngine_4.18/Engine/Binaries/Lin
f80lab1@F80LAB1LIN:/media/f80lab1/Datos_Lab/UnrealEngine_4.18/Engine/Binaries/Li
nux$ ./UE4Editor /media/f80lab1/Datos_Lab/carla-0.8.4_SOURCE-DEPRECATED/carla-0.
8.4_SOURCE/Unreal/CarlaUE4/CarlaUE4.uproject
```

Figure 21. Command that opens CARLA project in Unreal Engine 4 suite.

./UE4Editor opens up UE4's suite. The link that follows is the absolute path to CarlaUE4.uproject, CARLA's Unreal Engine 4 project itself.

After some seconds, the interface of the graphics engine will pop up:

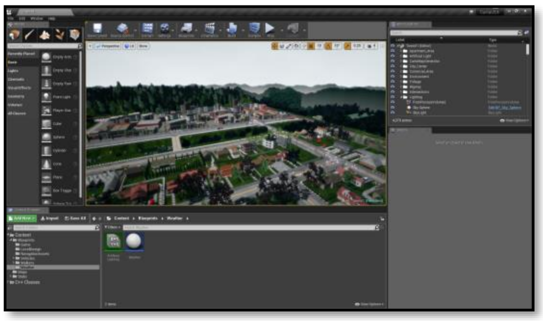


Figure 22. Unreal Engine 4 main interface.

3.2. Camera modelling

One of the biggest constraints of Unreal Engine 4 for our purposes is the lack of tools to simulate virtual cameras that incorporate ultra-wide-angle lenses or fisheye lenses. Many applications on the automotive world use such lenses in order to have a wider field of view, which translates into more information of the scene being recorded. This is extremely useful because, for instance, allows top view systems to gather all information surrounding a vehicle with only four cameras placed on the front, rear and side-mirrors of the cars.

Unreal Engine 4 only allows simulating cameras with a perspective projection so the main purpose of this project is to bring fisheye lenses into the engine's scene. To do such thing, some possibilities were evaluated:

- Modifying the inner renderer of the engine in order to change the projection distortion applied to the rendered image. In other words, make UE4 natively render images with a fisheye distortion at real time.
  This is the ideal solution, because it is straightforward and accelerates the process of rendering. However, there are some constraints to be considered: the engine only allows setting field of view up to 170 degrees, which is not enough to cover the 196 degrees field of view of our camera. Also, pixel density is highly decreased due to the pin-hole distortion camera model. The higher the field of view, the tinier the information on the center of the final image. When undistorting such image, that region with low pixel density but with a lot of data must be expanded by interpolation. In such process, resolution of the final undistorted image suffers a steep fall.
  Another problem to consider is that some code of the engine has restricted access and cannot be accessed. Moreover, as it is a closed API everything is connected so even doing little changes to the code results in massive amount of errors.

- Creating some kind of geometrical structure that allows increasing field of view and applying some sort of distortion to the final rendered image. If an orthographic camera is placed in front of a mirror-ball, it would capture the world reflected on the ball. Such reflections would form a fisheye image with field of view up to 360 degrees.

- A cube map could be generated. That structure consists of 6 perspective cameras placed in a way that each one covers a part of the ultra-wide-angle lens field of view. All the parts combined programmatically leads to a final image with the desired field of view and distortion. This approach has a big drawback, which is that it is computationally heavy because for each virtual camera to be simulated there will be six extra cameras to generate the cube map.

3.2.1 Mirror ball

After analysing all the options available, the mirror-ball and orthographic camera system seemed the easier and more effective to implement. It only requires Unreal Engine 4 to place a single orthographic camera and render the reflection of the world on the mirror-ball's surface, so it is considerably more computationally affordable than the cube map approach. In addition, its implementation is much simpler than modifying the renderer of the engine.

The method detailed:

The mirror-ball is simply a perfect sphere with a mirror material applied to it. The idea is to place an orthographic camera right in front of it so that the reflected light rays can reach a wider zone increasing that way the field of view of the camera artificially. Thanks to the form of the ball, everything projected on its surface produces an image with fisheye equisolid distortion.

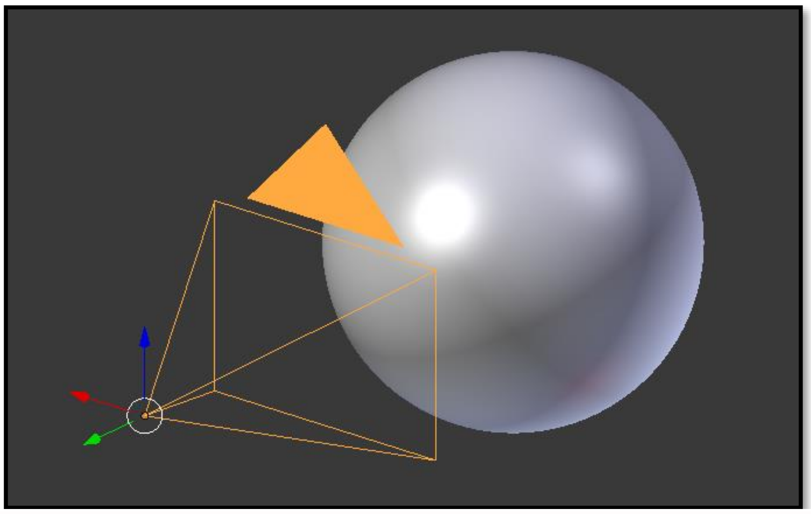


Figure 23. Blender example of a camera in front of a perfect mirror ball.

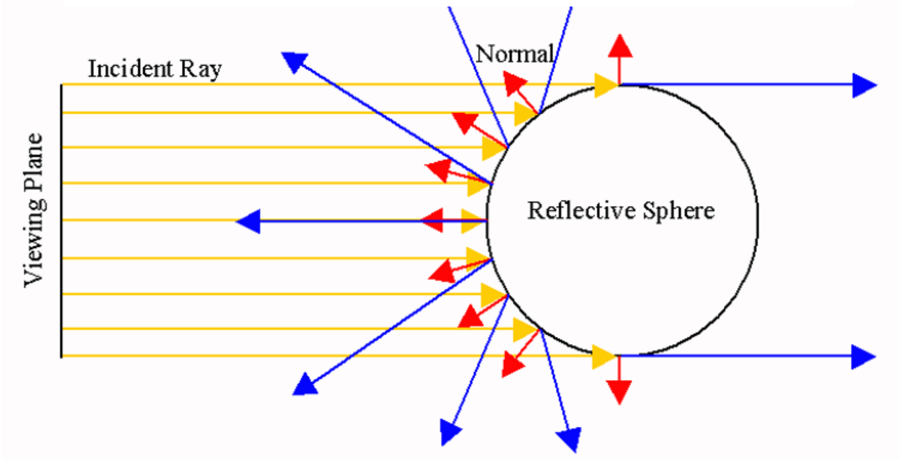


Figure 24. Illustrative scheme of the orthographic camera and mirror ball system.

The camera is modelled as a plane that casts rays uniformly distributed over its surface onto the sphere. The ball is a perfect mirror, so rays get reflected off at the very same angle they hit the surface of the sphere. Such angle depends on the normal of the area differential of the ball where the ray hits. That way, the further the ray collides from the center of the sphere the more deviated. A very high field of view can then be achieved. However, the more rays get deviated the more distortion is introduced to the resulting image.

It is also important that the mirror ball is very small (ideally infinitely small). This is because the plane that casts rays or orthographic camera must be as tiny as possible so that it can be seen as a spot. By doing this other kind of undesired distortions due to different initial starting points of the rays casted are avoided. If this condition is fulfilled, the model is able to capture almost the 360 degrees surrounding environment (ideally 360 degrees if the ball was infinitely small).

To build the whole simulation model (orthographic camera and mirror ball) some calculus must be made. Taking the specifications of the sensor and lens stated at the introduction:

There is interest in finding the focal length that a pure equisolid lens would have in order to present the same 196 degrees field of view of our custom lens given the same sensor.

To do such thing, the equisolid formula is used:

$$r = R \cdot \sin(\frac{\theta}{2})$$

Where r is the distance of each pixel to the center of the image or image height, R the double of the focal length to find and θ half of the field of view of the lens. See appendix A for more details on lens projections and their characterization.

Focal length is constant, so taking half the maximum horizontal field of view and half the horizontal size of the sensor should suffice for the purpose of finding it.

$$d = \frac{horizontal\ sensor\ size}{2} = \frac{4.032\ mm}{2} = 2.016\ mm$$

$$halfFOV = \frac{HFOV}{2} = \frac{196\ degrees}{2} = 98\ degrees$$

Then,

$$R = \frac{r}{\sin(\frac{\theta}{2})} = \frac{d}{\sin(\frac{halfFOV}{2})} = 2.671\ mm$$

$$f_{equisolid} = \frac{R}{2} = \frac{2.671\ mm}{2} = 1.335\ mm$$

It can be appreciated that a scale has been applied to the focal length in order to adapt it to the new equisolid lens model:

$$scale = \frac{f}{f_{equisolid}} = \frac{0.948\,mm}{1.335\,mm} = 0.710$$

If we consider the mirror ball to be infinitely small, the only thing to calculate is the orthographic scale of the camera. It controls the size of objects projected on the image. In other words, it sets the size of the plane that models the orthographic camera.

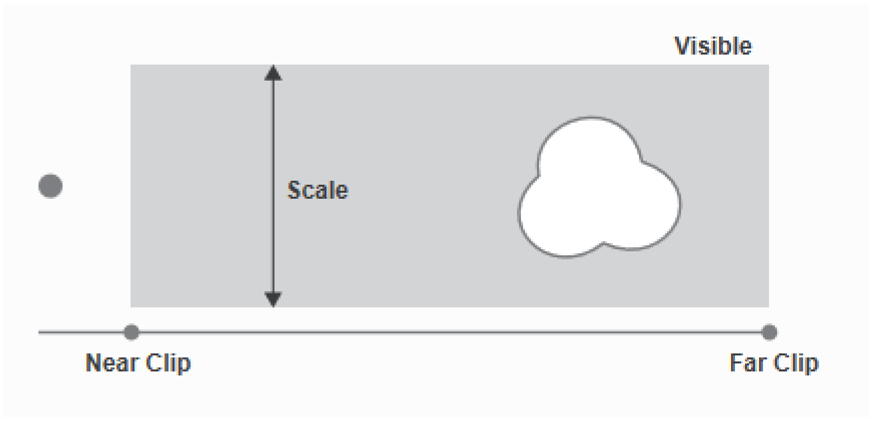


Figure 25. Blender's orthographic camera parameters[15]

The appropriate orthographic scale of the camera to get correct sized objects on the final projected image can be found performing some tests on Unreal Engine 4 and Blender. The way both software handle the scale is a bit opaque and there is no analytical way to perform the task. To do so, some markers are placed angularly equally spaced around a virtual orthographic camera placed in the middle of the scene. Right in front of the camera there is a mirror ball of diameter 0.5 mm. The camera is facing the mirror ball in a way that its optical axis goes through the center of the ball. All that the camera captures is the reflected environment on the surface of the mirror. The last marker (yellow) is placed at the desired half horizontal field of view, in our case 98 degrees from the camera's optical axis. Then, one must play with the orthographic scale parameter in order to see the last marker on the horizontal central line of pixels on the output image.

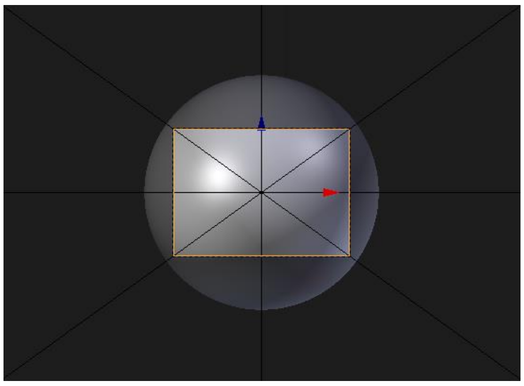


Figure 26. Size of the camera projected on the mirror ball. Orthographic scale.

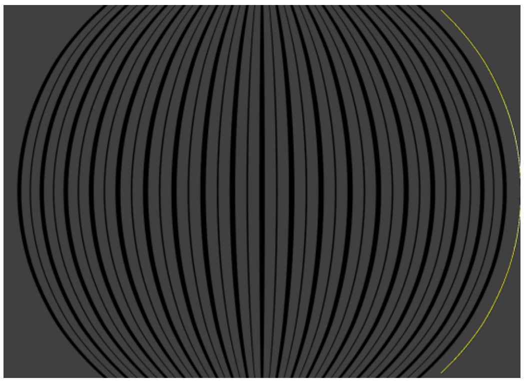


Figure 27. Image captured by orthographic camera looking at mirror ball with orthographic scale matching 196 degrees FOV.

[15] https://docs.blender.org/manual/en/latest/render/cycles/camera.html

For the AT102A lens, the correct orthographic scale is 0.758 with a mirror ball diameter of 1 mm.

Earlier it is said that the mirror ball must be infinitely small. That is somehow an ideal statement. However, in order to implement it, the only thing to consider is that it is very small compared to the other objects in the scene. For instance, if the virtual world consists in cars, buildings, roads, etc. and arbitrary good choice would be half a millimetre. It is up to the designer though to choose such value. The bigger the ball is the more undesired distortion is introduced into the final image. Take into account though that changing the mirror ball size changes requires a change on the orthographic scale of the camera because they are dependent on each other.

In order to determine the accuracy of the presented model some test have been run in a simulation software that called Blender. This tool incorporates simulations with fisheye equisolid lenses.

One way of validating the model is to generate an equisolid virtual camera with the sensor and lens parameters to be tested and take a picture of a certain environment. After that, we replace the camera with our solution and take another picture of the same environment. The proposed environment:

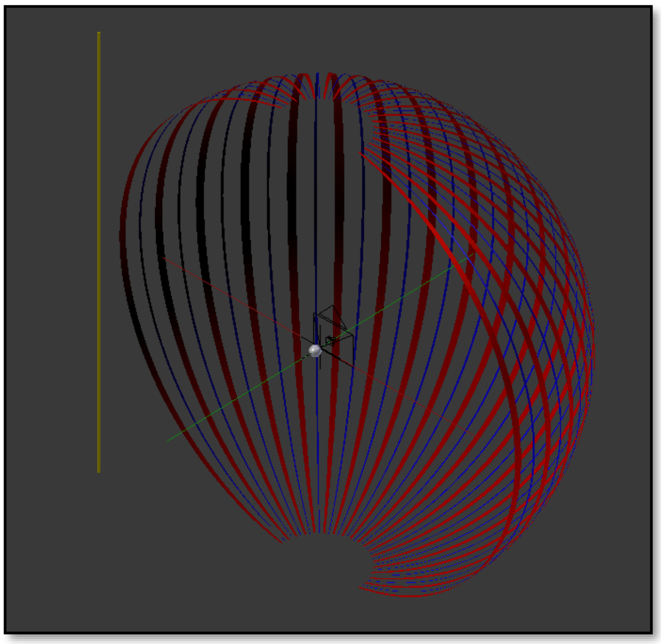


Figure 28. Blender environment to find the right scale for the orthographic camera and validating the model.

Note that the environment is a half-sphere with latitude lines only. The center of the scene are the camera models, so lines are 5 degrees separated. The yellow cylinder at the left is located at 188 degrees and allows us to see if the model reaches the 196 degrees of horizontal field of view.

Be careful though, because the set of images generated correspond to an equisolid projection. However, we are interested in having a custom distortion according to the specific lens manufacturer. In order to distort the image in the right way a MATLAB script is used. It performs the operation of converting the image from equisolid projection to custom projection according to some input lens parameters such as horizontal field of view, focal length, pixel and sensor size and distortion curve among others.

Now, by comparing both pictures it can be evaluated how far is the model from the ideal one.

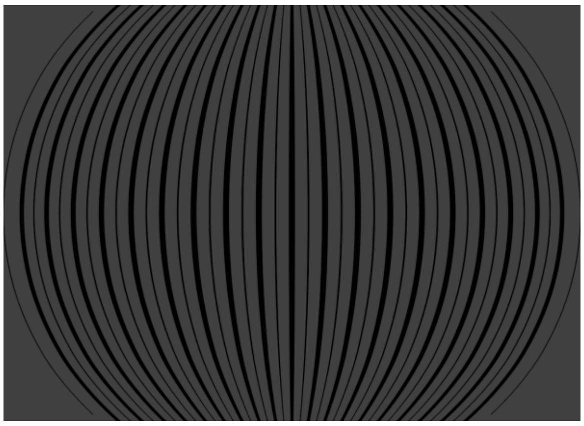


Figure 29. Image obtained with the mirror-ball system.

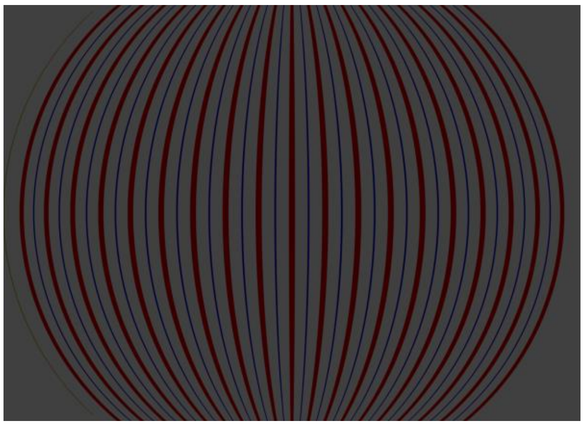


Figure 30. Image obtained with a fisheye equisolid camera.

We have seen the amount of pixels that are from the center of the image to the middle to each of the reference lines separated 5 degrees. These pixels can be translated into millimetres thanks to the pixel size information of the sensor. Then, we can check if the position of each line is correct on the sensor by looking at the lens distortion table. All the measures and the deviation from the theoretical model of the lens are listed below on Table 1.

Table 1. Comparison between designed mirror ball models and theoretical lens projection.

angle theta [deg]	real height Y [mm]	[Model 1] r = 1 scale = 0.758	[Model 1] Error	[Model 2] r = 0.454 scale = 0.3399	[Model 2] Error
0	0.0000	0.0000	0.000%	0.0000	0.000%
5	0.0827	0.0840	1.562%	0.0840	1.562%
10	0.1658	0.1650	-0.512%	0.1650	-0.512%
15	0.2499	0.2490	-0.342%	0.2490	-0.342%
20	0.3351	0.3330	-0.639%	0.3330	-0.639%
25	0.4221	0.4200	-0.502%	0.4200	-0.502%
30	0.5112	0.5070	-0.817%	0.5070	-0.817%
35	0.6027	0.6000	-0.443%	0.6000	-0.443%
40	0.6969	0.6930	-0.564%	0.6930	-0.564%
45	0.7942	0.7890	-0.661%	0.7890	-0.661%

50	0.8949	0.8910	-0.432%	0.8910	-0.432%
55	0.9990	0.9930	-0.598%	0.9960	-0.298%
60	1.1067	1.1010	-0.514%	1.1040	-0.243%
65	1.2181	1.2150	-0.252%	1.2150	-0.252%
70	1.3331	1.3290	-0.305%	1.3290	-0.305%
75	1.4515	1.4460	-0.380%	1.4520	0.033%
80	1.5731	1.5690	-0.261%	1.5750	0.120%
85	1.6972	1.6950	-0.129%	1.7010	0.224%
90	1.8225	1.8210	-0.083%	1.8270	0.246%
98	2.0198	2.0160	-0.188%	2.0160	-0.188%

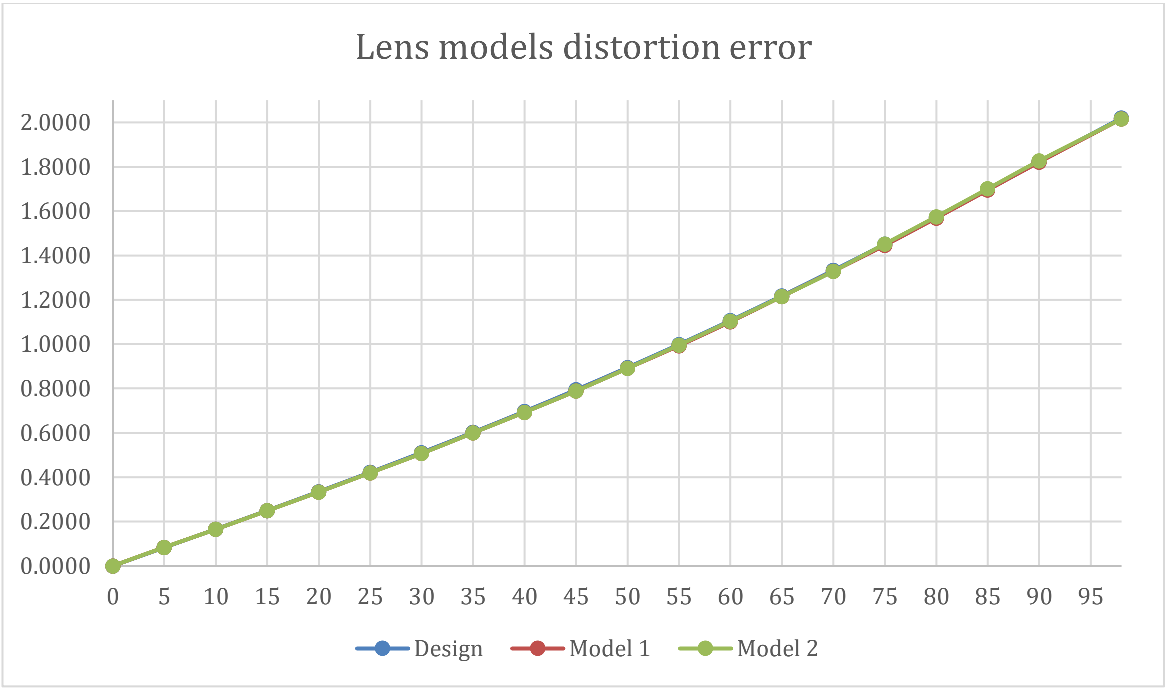


Figure 31. Graphic representation of Table 6.

As it can be seen, both models are very accurate and allow for a great resulting fisheye image. However, there is a big constraint imposed by Unreal Engine 4: reflections are rendered at a very poor 132x132 resolution in order to save computational power. This fact makes the above explained approach useless to generate a fisheye image, because the final image would have a very poor quality. Figure 32 reflects all this. As can be seen, the quality of the reflections is too poor to generate high quality images. The following image is an imported mirror ball in the Unreal Engine 4 environment:

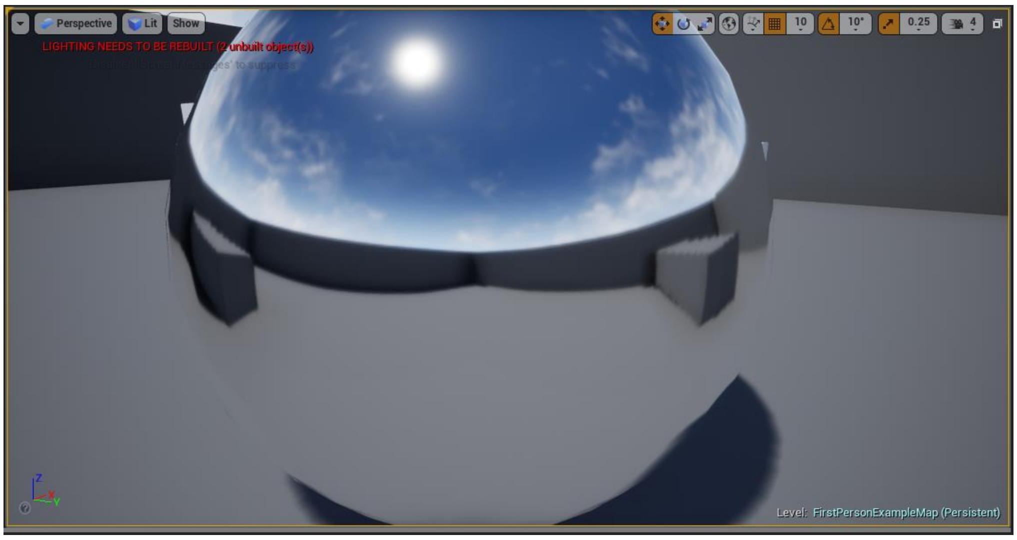

Perspective
Lit
Show
LIGHTING NEEDS TO BE REBUILT (2 unbuilt object(s))
Level: FirstPersonExampleMap (Persistent)

Figure 32. Picture that illustrates the low resolution UE4 applies to mirrors.

There is another big limitation for this model to be used in UE4: the orthographic camera is somehow bugged and does not work properly. Sometimes it retrieves completely blank images with no justification.

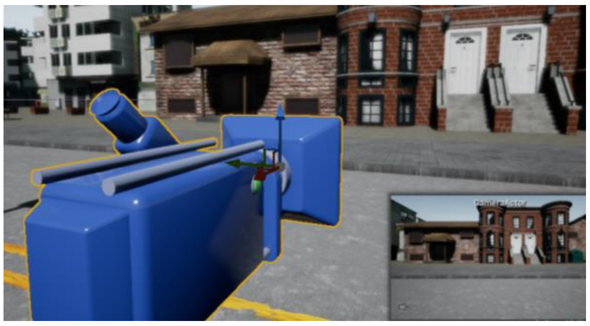


Figure 33. Perspective camera in UE4.

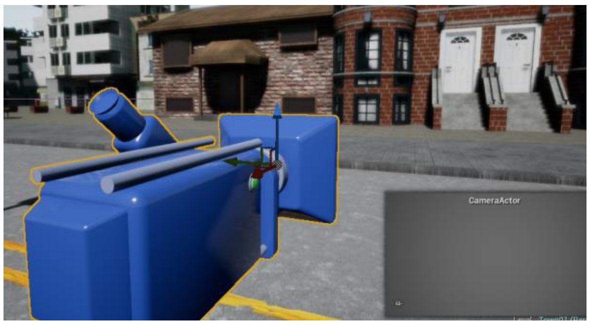


Figure 34. Orthographic camera in UE4. Note that it shows nothing.

3.2.2 Cube map

An alternative approach to solve the field of view and distortion problem of the virtual camera is to generate a cube map. It consists in a composition of six perspective images that form a 360 degrees panorama. Such projection is then transformed using image processing to a fisheye panorama, so after all it is just a transition step. An example of a cube map:

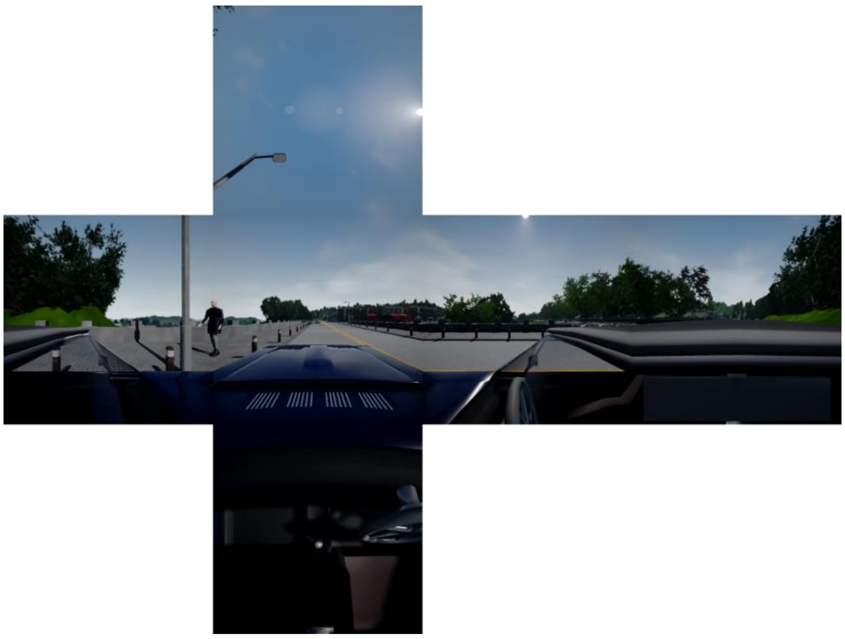


Figure 35. Cube map example made with CARLA generated images.

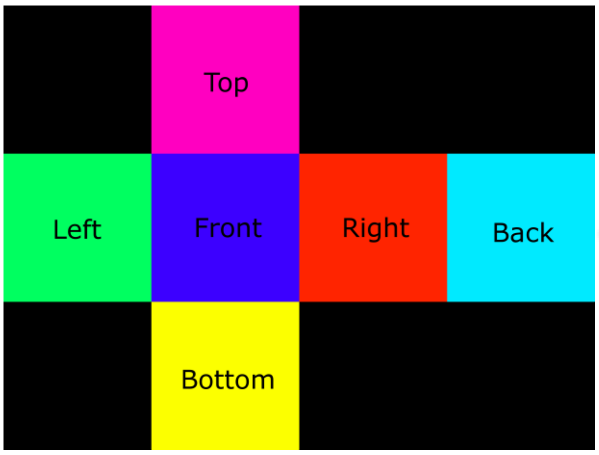


Figure 36. Illustrative example of a cube map.

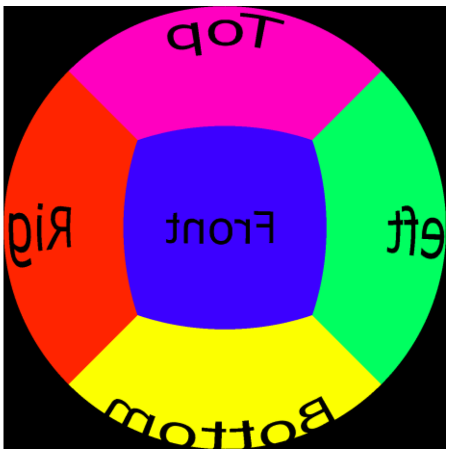


Figure 37. Fisheye form when converting from a cube map.

Images can be structured in different ways, but the chosen form is the one seen in Figure 36. However, one can arrange images so that they form a vertical cross, for instance.

Note that the final fisheye image is flipped horizontally. This is intentional because rear view cameras used in ADAS generate images that must be flipped to help the visualization to the driver in a similar way as conventional rear view mirrors.

The method to generate a cube map consists in placing six perspective cameras in the simulator with the very same location but rotated 90 degrees from each other. That way, if we place cameras facing front, left, right, back, top and bottom a 360 degrees panorama is recorded each frame. The reason why it is possible to place six cameras at the same point is because they are modelled as spots that cast rays in the simulator.

Note that all six cameras must have a field of view both horizontal and vertical of 90 degrees. This constraints the resolution of the image to be squared. A good resolution choice would be 1024x1024, because it is good enough to generate output 1344x1344 images. It is easily seen on Figure 37 because the front tile roughly occupies half of the final fisheye image. That means that 1024 pixels are used to build just half of a 1344 pixel image.

See Figure 38 for a better understanding of the positioning of cameras in the scene:

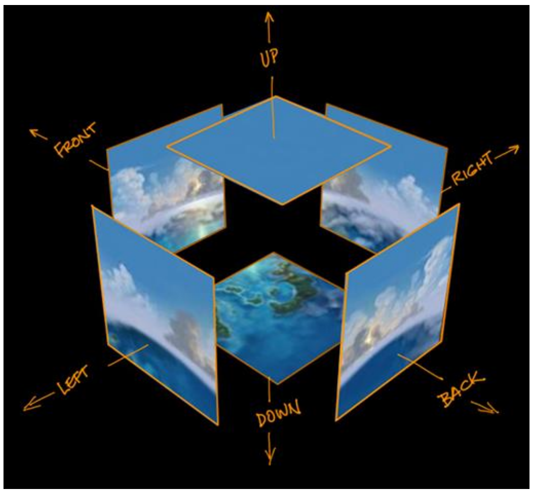


Figure 38. Cube map faces configuration.

The Python API makes it very simple to place a camera in the environment by scripting:

```
camera0 = Camera('front', PostProcessing='SceneFinal')
camera0.set_image_size(x_resolution, y_resolution)
camera0.set_position(x, y, z)
camera0.set_rotation(yaw=yaw, pitch=pitch, roll=roll)
settings.add_sensor(camera0)
```

Figure 39. Python code to set virtual cameras in CARLA.

That way, when launching a simulation and connecting the client with the code of Figure 39 CARLA saves the generated images in well-organized folders.

Those pictures must then be processed by an external tool consisting in a Python script that creates a cube map out of the six images for every frame or time step. Also, it directly transforms all cube maps recently generated into a fisheye image.

Python is the coding language chosen because it is multiplatform, easy to use and has some powerful libraries such as OpenCV[16] and numpy[17] that are written in C at very low level, resulting in good performance.

3.2.2.1 Cube map to equirectangular script

The first idea was to transform the cube map into and equirectangular panorama image. Such image would be a 360 degrees representation of the virtual world captured by cameras, with a peculiar distortion similar to the one found on maps. Then, the image would be cropped appropriately in order to get the desired output field of view. With such actions an equidistance fisheye image was meant to be outputted. There is just one step more: using the MATLAB script to adapt the image to the custom distortion to get the desired lens distortion.

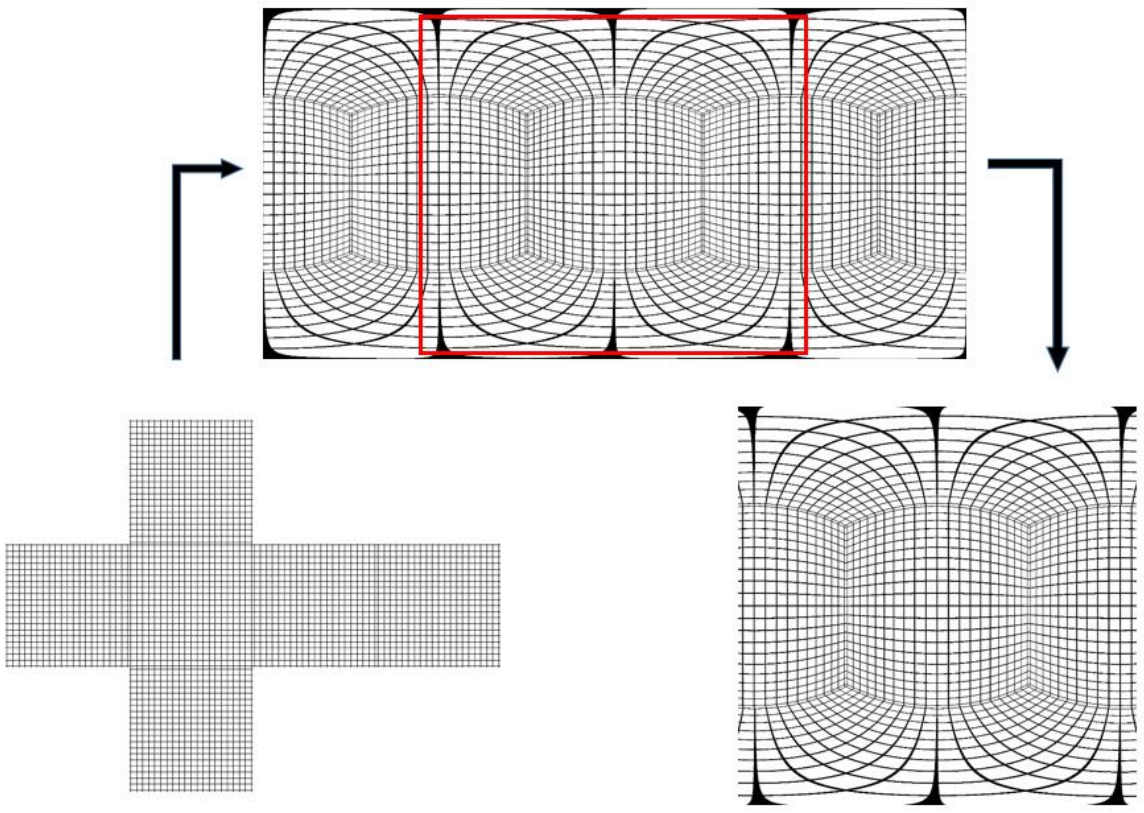


Figure 40. Cube map to equirectangular panorama with desired field of view process.

[16] https://opencv.org/
[17] http://www.numpy.org/

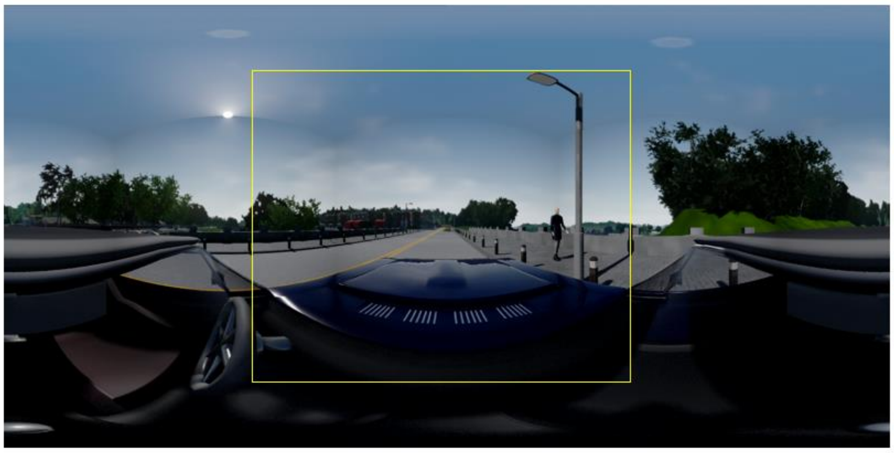


Figure 41. Equirectangular image with images from CARLA. Information inside the yellow rectangle is the one of interest.

However, after performing some tests and validating the results, it can be seen that they are not the expected ones. Although pretty close, there is different distortion at all points of the image, especially at the top and bottom parts of it. Moreover, some lines are blended on the opposite expected direction.

The validation process consists in generating an equirectangular image with the Python script using a cube map made of a grid and comparing it to an equisolid image from Blender. To create the last mentioned picture a cube with a grid applied to each face as texture is constructed and a fisheye camera is placed right at the middle, inside it. The camera outputs and image with an equisolid projection. Both the images from Blender and from the Python script are transformed by the MATLAB script to get the custom lens projection of the AT102A.

Figure 42 and Figure 43 show the final result of the whole process. Images do not perfectly match, which means that this approach to acquire fisheye images with a specific lens distortion from CARLA is not valid.

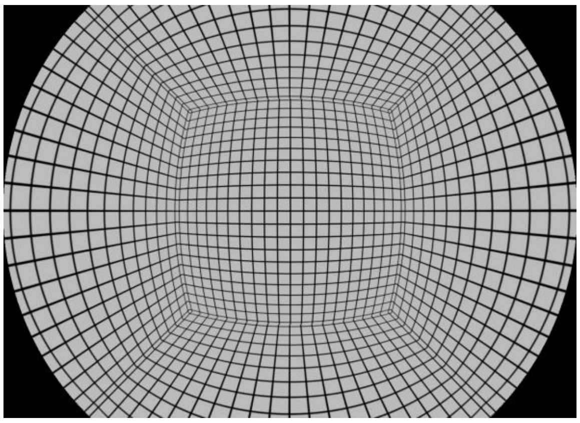


Figure 42. Blender generated image with AT102A distortion.

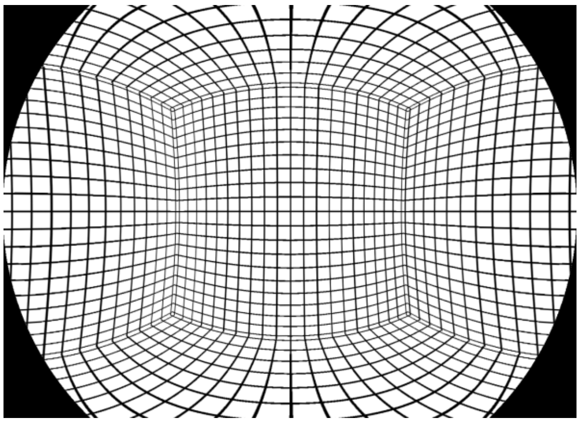


Figure 43. Equirectangular image cropped and distorted to match AT02A.

See more details on appendix D for more details on how the script works and why it does not produce the desired result for this application.

3.2.2.2 Cube map to fisheye script

This time, instead of converting the cube map into an equirectangular panorama as an intermediate step to then compute the fisheye distortion, the script takes as input the images generated by the six cameras and outputs fisheye equisolid images.

When CARLA records the world with its cameras and saves the images on disk it creates six folders, one for each camera, and stores in there all the frames of every camera.

The program starts by loading all this images and generating cube maps out of them for each frame. That way, a 360 degrees panorama is generated for each simulation instant.

The whole code can be found in appendix C.

Mathematically, the process consists in placing an imaginary virtual camera at the center of a cube. Such figure is a folded cube map wrapped with the images taken by the virtual cameras of CARLA as textures. Rays are casted from that spot onto the inside cube surface. Pixels hit by the ray are then placed appropriately onto a plane that will eventually become the final fisheye image. Note that, although this is a 3 dimensional view of the whole process of transforming a cube map into a fisheye image, cube maps are actually stored unfolded as 2 dimensional images.

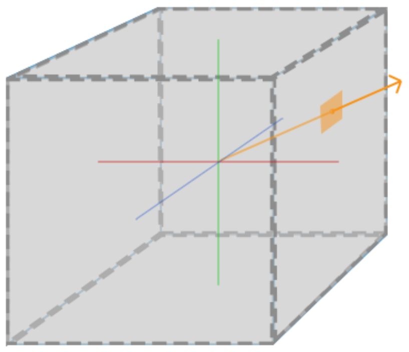


Figure 44. 3D representation of the ray casting process that the script carries out.

Programmatically, spherical coordinates are calculated on the output image plane in the first place. Such coordinates must have its origin at the center of the canvas, which is why polar coordinates are appropriate for this task. Therefore, it is needed to compute two parameters: r and φ. The calculus are easily made thanks to the numpy library for Python. It has a very similar syntax to MATLAB and allows making operations on matrices without looping over them.

Using matrix subtraction and function *meshgrid*[18], the script moves the origin of the coordinate system to the center (it is placed at top left corner in Python as default) and normalizes the coordinates.

[18] https://docs.scipy.org/doc/numpy-1.15.0/reference/generated/numpy.meshgrid.html

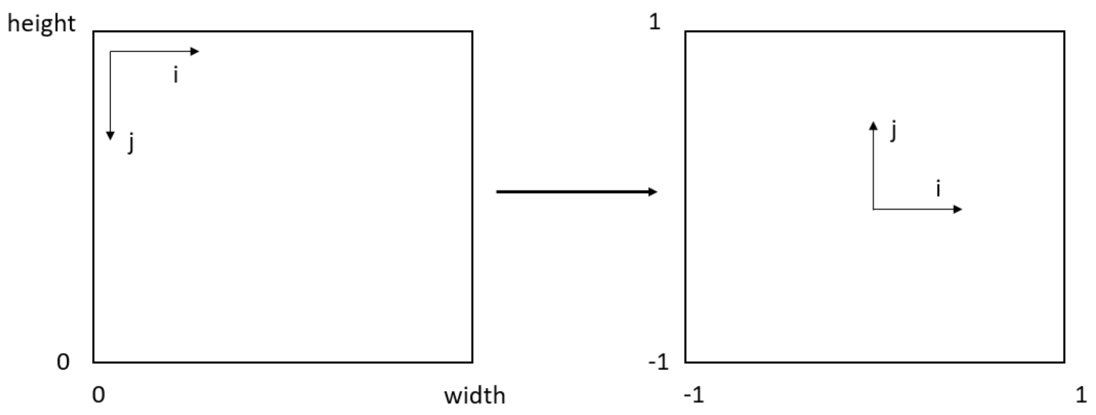


Figure 45. Original image coordinates (left) and normalized ones (right).

Polar coordinates *r* and φ are calculated according to the following scheme:

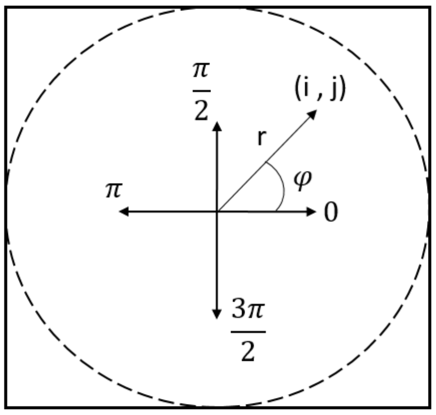


Figure 46. Representation of spherical coordinates on a fisheye output image plane.

$$r = \sqrt{i^2 + j^2}$$

$$\varphi\ (phi) = \begin{cases} 0 & r = 0 \\ \pi - \operatorname{asin}\left(\frac{j}{r}\right) & i < 0 \\ \operatorname{asin}\left(\frac{j}{r}\right) & i \geq 0 \end{cases}$$

The dashed circle represents the boundary where $r = 1$. Everything outwards that line is of no interest for the purpose of creating a fisheye image. That is because the cube map is projected onto a half sphere represented on a plane. Therefore, the resulting shape must be circular with radius not greater than one for a cube map of dimensions 1x1x1. The dashed circle represents the lens and the image plane the sensor. Depending on the size of the lens-sensor system, the circle and black corners will be drawn in one way or another[19].

Get_spherical_coordinates returns the spherical coordinates for the image plane. Such coordinates must be mapped to the 3-dimensional cube map in order to get the corresponding pixel.

This function gets the spherical coordinates previously calculated and computes a 3 dimensional ray that hits the surface of the cube map at point x, y, z. That way, for each pixel at the output image plane with a certain spherical coordinate a pixel on the cube map is assigned.

Parameter θ only depends on the desired output field of view. Figure 47 shows what each variable represent:

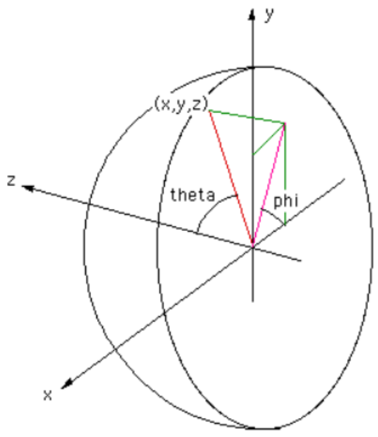


Figure 47. 3D scheme that illustrates what theta and phi represent.

The coordinate system used in this work for the cube map is the following:

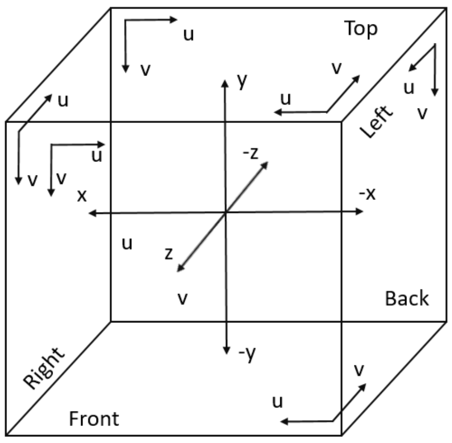


Figure 48. Cube map coordinates and tiles composition.

[19] See appendices A and C for more information on black corners.

For such system, the equations that transform r, φ, θ to x, y, z are:

$$x = \sin(\theta)\cos(\varphi)$$

$$y = \sin(\theta)\sin(\varphi)$$

$$z = \cos(\theta)$$

However, this is just a point in the 3 dimensional space. It must be converted to the 2 dimensional space where each cube map resides. The first thing to do is seeing at which tile that vector is pointing at. Function *get_face* does that work. It is easy to determine such thing depending on the magnitude of x, y and z. Each tile of the cube map has an axis normal to it. The one with higher absolute value is the side where the vector points to.

Defining the coordinates systems as in Figure 48 makes it easy to map 3 dimensional points to the 2 dimensional cube map image. Depending on the face the vector is pointing to, the program assigns different values for the uv coordinates. For instance, if vector is (x,y,z) = (0.99, 0.54, 0.31), it points to positive x direction, which corresponds to Left face according to the coordinate system convention. The reason is that the greater absolute value of the vector is x. According to the schemes, if the selected face is Left, u = z and v = -y. Therefore, final coordinates u = 0.31 and v = 0.54 are obtained. Obviously, those coordinates are normalized and must adapted to the size of the image:

$$u = u_{norm} \cdot width_{image} = 0.31 \cdot 1024px \approx 317$$

$$v = v_{norm} \cdot height_{image} = 0.54 \cdot 1024px \approx 552$$

If the origin coordinates on the 2D cube map image of the Left tile are added, the final coordinates where to find the pixels are found:

$$col = u + x_{origin} = 317 + 0 = 317$$

$$row = v + y_{origin} = 552 + 1024 = 1576$$

Colour information of pixel (1576, 317) of the 2D cube map image is then placed on the location of the pixel on the output canvas fisheye image that was used to compute the spherical coordinates on the first place.

That way, a pixel is extracted from the cube map and placed on the output fisheye image where it corresponds, according to the calculated coordinates. This whole process must be repeated for each pixel of the output image until the whole picture is built.

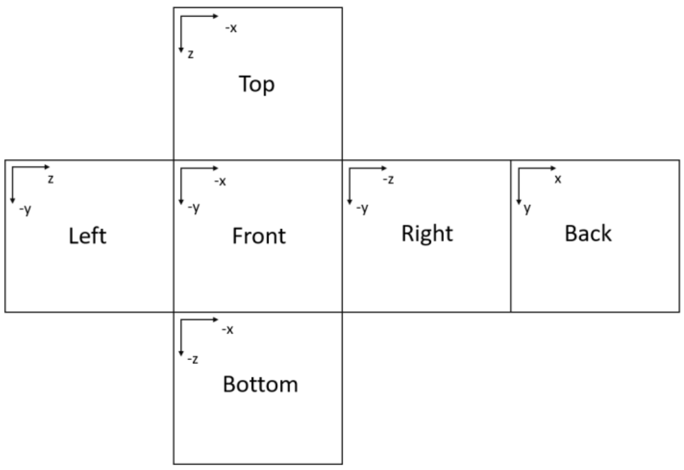


Figure 49. Tiles' uv coordinates of a cube map.

One last thing to keep in mind is that the input images for the script must have at least half the resolution of the output in order to obtain decent quality. That is because a single tile occupies half the output image approximately.

Note that, this time, the conversion from cube map to fisheye gives as result an equisolid image like the one obtained with Blender:

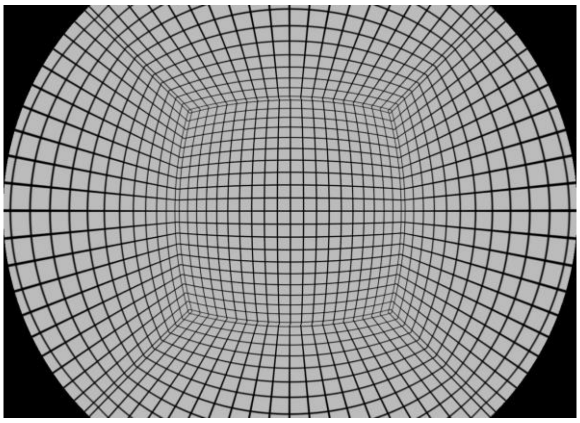


Figure 50. Blender generated image with AT102A distortion for fisheye validation.

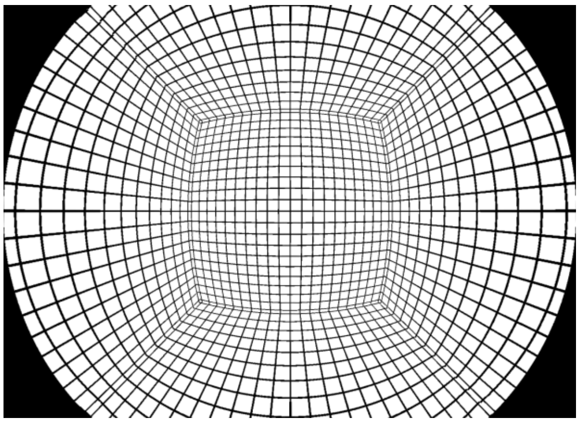


Figure 51. Output of the script for a grid image.

Figure 50 and Figure 51 validate the results obtained by the Python cube map to fisheye tool.

3.3. <u>Vehicle integration</u>

Once the environment is set up and there is a way of generating fisheye images even though the known limitations of Unreal Engine 4, there is only one thing left: integrating custom vehicles.

Vehicle CAD files provided by FICOSA customers are converted from CATIA proprietary formats (i.e. CATPart or CATProduct) to standard 3D formats like STL. Therefore, this files can be used in other simulation environments like CARLA to run simulations. In this case, customers send CADs of vehicles that are imported into CARLA. Integrating virtual cameras into these virtual cars allows to study mechanical interferences derived from the integration. Visual obstruction caused by bodywork of different vehicles is unique, meaning that an exact same ADAS mounted on different vehicles retrieve different results.

Importing vehicles into the simulation environment must be done through Unreal Engine 4.

First of all, the vehicle object must be correctly modelled in Blender. Bodywork and wheels have to be different meshes joined by an armature that will allow Unreal Engine 4 to apply physics to the asset.

Once the model is finished, it is exported as an FBX format so that it can be properly imported into Unreal Engine 4.

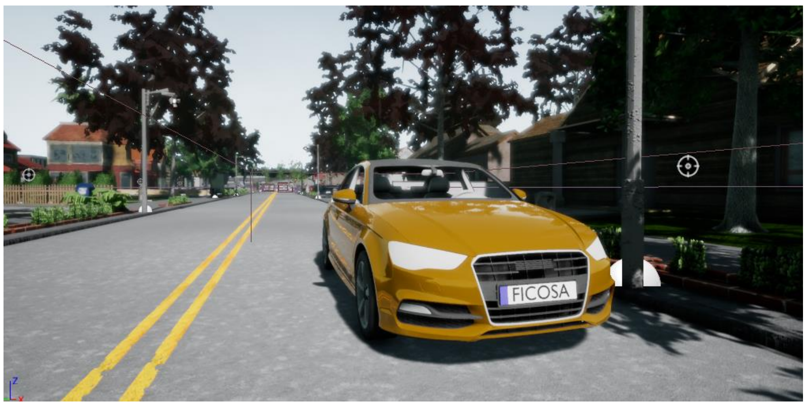


Figure 52. FICOSA's vehicle in CARLA

Some blueprints and configuration have to be performed so that the vehicle is fully operative and simulations can be run. Otherwise, the vehicle will be susceptible of being placed on the CARLA environment but there will not be the possibility of running simulations with that asset in the world.

In order to have further detail and insight on the whole process of creating and importing vehicles into CARLA, refer to appendix E.

4. Results

Some important achievements can be highlighted from this work. A simulation tool in constant development such as CARLA has been evaluated finding as a big limitation the not possibility of setting higher fields of view than 170 degrees to cameras. Also, the availability of only pin-hole (perspective) camera models was critical taking into account that ADAS based on cameras usually rely on fisheye lens.

A solution for the camera issue is considered: rendering a 360 degrees cube map and transforming it into a fisheye equisolid image. Such picture is then transformed to custom lens distortion according to requirements of every project. The result of converting a cube map into a fisheye image is the following:

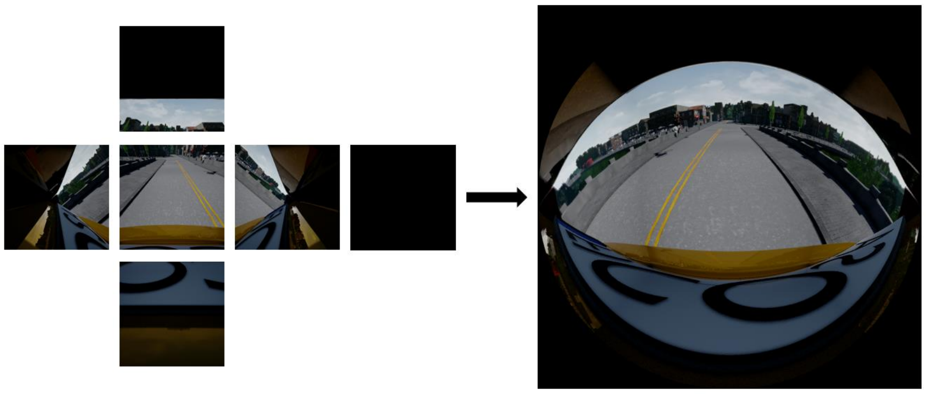


Figure 53. Cube map to fisheye equisolid conversion results.

Fisheye equisolid image on Figure 53 processed with the MATLAB script to obtain AT102A lens distortion:

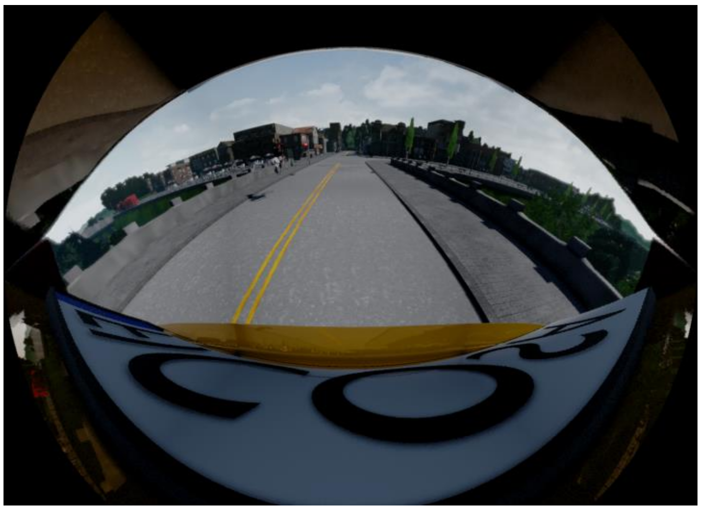


Figure 54. Fisheye image with AT102A distortion.

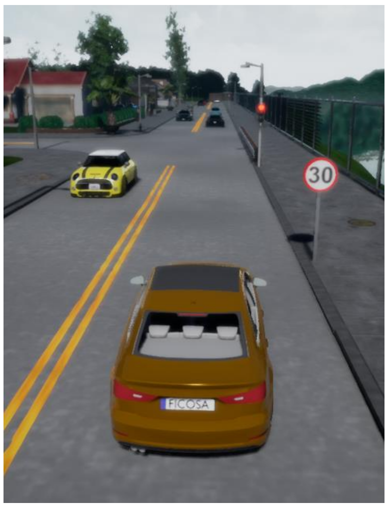


Figure 55. FICOSA's demo vehicle in CARLA environment.

Generated synthetic images with CARLA are very valuable because –among other things- are based on cameras mounted on a self-imported custom vehicle, which is another achievement of this work.

All the above together will allow FICOSA to design RVS and CMS.

Although the solutions found satisfy FICOSA’s needs, there exist certain limitations. One must be very careful when choosing a weather for the simulation because some effects like vignette, blooming and lens flares can cause artefacts on the final image. After all, a cube map is a combination of six pin-hole cameras. Each one of them will record the environment with different light conditions and perspectives. Because of that, some faces of the cube map can have slightly different light conditions. Furthermore, if a vignette effect is introduced, there will appear a darkened line on the border of each perspective image that will result in a highlighting of the stitching borders of the final fisheye image.

In order to minimize the above explained limitations, some actions such as eliminating blooming, lens flares and vignette effects can be performed in Unreal Engine 4. Also, choosing low light weathers with the sun remaining invisible can improve quality of images providing a more homogenous light to all tiles of the cube map.

It is far from ideal the fact that an external tool to Unreal Engine 4 must exist. A better solution would be to modify the inner renderer system of the graphics engine. However, this is a drawback that only Epic Games can deal with.

5. Budget

Table of software and material costs is the following:

Table 2. List of software and components used during the project development.

Description	***Quantity***	***Unitary Cost (€)***	***Total Cost (€)***
MATLAB license	1	2000	2000
MATLAB Image Processing Toolbox	1	1000	1000
Laptop	1	500	500
PC for simulations	1	2500	2500
Microsoft EXCEL	1	100	100
TOTAL			6100

Other software such as Blender has been used but it is open source under the GNU license.
Unreal Engine 4 is also free to use for the purpose of this work. Its entire source code can be found on the web.
CARLA is an open source project for Unreal Engine 4 under an MIT license.

Cost of the working ours dedicated to the thesis:

Table 3. Manpower costs.

Project hours	**Hourly wage (€)**	**Total price (€)**
150	40	6000

6. Conclusions and future development:

This work shows how to perform simulations in order to evaluate ADAS. With the tool up and running, a lot of possibilities arise.

Top view systems, rear view systems and CMS can already be evaluated with CARLA thanks to the creation of an external tool that allows to simulate fisheye cameras in Unreal Engine 4.

With such a complex simulation tool, style departments of FICOSA will be able to build a virtual reality environment that will allow the user to see from the inside of the vehicle how a CMS looks like, for instance. An immersive experience like that could be priceless at early stages of the project, because it would be real material to work with, not only guesses or ideas of how the final product will feel. Such application could be also interesting for commercial purposes.

One of the most important things in ADAS is that new technology gets homologated. In order to check whether a system is legal -and therefore fulfils requirements- is to evaluate them in virtual environments. Some test can be performed on static environments but others require dynamism. For instance, it is required an environment like CARLA to artificially place pedestrians on the middle of the road and observe how the vehicle stops before hitting it and its behaviour in front of a difficulty or obstacle like that.

Vision systems FICOSA's department can be greatly benefitted of CARLA. It would allow to work with high fidelity video material to develop algorithms to be used later both on synthetic images and real cars. Such programs typically perform extrinsic and intrinsic calibration of cameras, ground truths, temperature studies of the system, etc. Also, dynamic simulation environments open the door to new lines of investigation such as object detection and tagging. Robustness of algorithms can also be tested by adding noise, lens flares and other non-desirable effects that occur in real life and affect ADAS.

CARLA only has two maps available –Town01 and Town02-. An important future improvement should involve importing other environments into the simulator. The Unreal Marketplace is full of different maps that could be very useful to test ADAS in different conditions of light, road distribution, etc. It is possible to do that through Unreal Engine 4, in a very similar way to how to import vehicles.

Videos recorded in CARLA and CAN information could be injected on HIL systems in order to evaluate a camera's behaviour without performing road test or having real samples of the final system. Other sensors that can be studied are LIDAR (also included in the environment) and depth cameras. These sensors in ADAS are increasingly being used for applications like alerting the driver if there is something on the side mirrors' blind spots

To sum up, this thesis will help to the developing of smart systems that will make the automotive industry evolve towards intelligent vehicles. A new set of simulation tools are presented in order to evaluate and design such advanced systems. As presented in this section, many actions can be made to improve the tools performance and possibilities that they offer. All this actions define the future development and are evaluated internally at FICOSA.

Bibliography:

[1] P. Bourke. "Computer generated angular fisheye projections". [Online] Available: http://paulbourke.net/dome/fisheye/. [Accessed: 10 October 2018].

[2] P. Bourke. "Image warping for off axis fisheye lens/projections". [Online] Available: http://paulbourke.net/dome/fisheyewarp/ [Accessed: 15 October 2018].

[3] P. Bourke. "Converting to/from cube maps". [Online] Available: http://paulbourke.net/miscellaneous/cubemaps/. [Accessed: 5 October 2018].

[4] Dagon. "Using Blender to make cube maps". [Online] Available: https://github.com/Senscape/Dagon/wiki/Using-Blender-to-make-cube-maps. [Accessed: 25 October 2018].

[5] PanoTools. "Projections". [Online] Available: https://wiki.panotools.org/Projections. [Accessed: 30 September 2018].

[6] StackOverflow. "Converting a cube map into equirectangular panorama". [Online] Available: https://stackoverflow.com/questions/34250742/converting-a-cubemap-into-equirectangular-panorama. [Accessed: 10 November 2018].

[7] OpenCV foundation. "Converting a cube map into equirectangular panorama". [Online] Available: http://answers.opencv.org/question/180430/convert-cubemap-pixel-coordinates-to-equivalents-in-equirectangular/. [Accessed: 10 November 2018].

[8] Wikipedia Foundation Inc., "Cube mapping". [Online] Available: https://en.wikipedia.org/wiki/Cube_mapping. [Accessed: 10 November 2018].

[9] Epic Games. "Unreal Engine 4 Documentation". [Online] Available: https://docs.unrealengine.com/en-us/. [Accessed: 10 December 2018].

[10] CARLA Team, CVC. "CARLA Documentation". [Online] Available: https://carla.readthedocs.io/en/latest/. [Accessed: 20 September 2018].

[11] Deep Drive. "Rebuilding Deepdrive on Unreal Engine 4". [Online] Available: https://deepdrive.io/blog/building-on-unreal.html. [Accessed: 20 September 2018].

[12] Shaun Lebron. "Visualizing projections". [Online] Available: http://shaunlebron.github.io/visualizing-projections/. [Accessed: 20 September 2018].

[13] Stanford University. "Python numpy tutorial". [Online] Available: http://cs231n.github.io/python-numpy-tutorial/. [Accessed: 10 November 2018].

[14] The Mathworks Inc., "Automated driving system toolbox". [Online] Available: https://es.mathworks.com/products/automated-driving.html. [Accessed: 1 October 2018].

[15] Microsoft Corporation. "AirSim". [Online] Available: https://github.com/Microsoft/AirSim. [Accessed: 1 October 2018].

[16] IPG Automotive GmbH. "CarMaker: Virtual testing of automobiles and light-duty vehicles". [Online] Available: https://ipg-automotive.com/products-services/simulation-software/carmaker/. [Accessed: 1 October 2018].

[17] *United Nations, Regulation No 46 Revision 5*. ECE R46, August 2013.

[18] U.S. Department of Transportation, National Highway Traffic Safety Administration (NHTSA), Vehicle Rear view Image Field of View and Quality Measurement (FMVSS 111), DOT HS 811 512, September 2011.

[19] Aleksandar M. Dimitrijevic, M. Lambers, Dejan D. Rancic. "Comparison of spherical cube map projections used in planet-sized terrain rendering". Vol. 45, no. 2, pp. 259-297. 2016

[20] S. Kriglstein, G. Wallner. "Environment mapping".

[21] John P. Snyder, U.S. Bureau of Mines. "Map projections – a working manual". 1987, Washington, USA.

[22] P. Zimmons, U.S. Bureau of Mines. "Spherical, cubic, and parabolic environment mappings". December 1999.

Glossary

ADAS	Advanced Driver Assistance Systems
PC	Personal Computer
UE4	Unreal Engine 4
CARLA	Car Learning to Act
CMS	Camera Monitor Systems
CAD	Computer-Aided Design
STL	Standard Template Library
FBX	Filmbox
3D	3 Dimensions
CATIA	Computer-Aided Three Dimensional Interactive Application
MATLAB	Matrix Laboratory
OpenCV	Open Computer Vision Library
API	Application Programming Interface
LTS	Long Term Support
CVC	Computer Vision Center
TCP/IP	Transmission Control Protocol / Internet Protocol
AI	Artificial Intelligence
RGB	Red Green Blue
LIDAR	Light Detection and Ranging
FOV	Field of View
EFL	Effective Focal Length
EP	Entrance Pupil
UV (Coordinates)	Horizontal Vertical
RVS	Rear View System

HIL	Hardware in the Loop
CAN	Controller Area Network