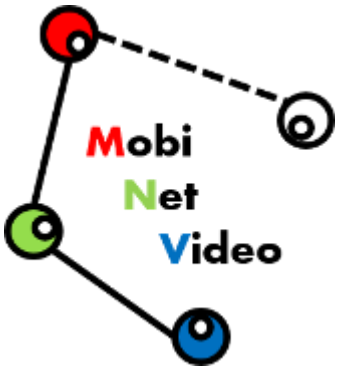

TEC2017-88169-R MobiNetVideo (2018-2020)

Visual Analysis for Practical Deployment of Cooperative Mobile Camera Networks

D1.2 v2

Camera Simulation

Video Processing and Understanding Lab

Escuela Politécnica Superior

Universidad Autónoma de Madrid

Supported by

AUTHORS LIST

Juan C. SanMiguel	Juancarlos.sanmiguel@uam.es

HISTORY

Version	Date	Editor	Description
0.1	29/04/2019	José M. Martínez	Initial draft version
0.2	14/07/2019	Juan Carlos San Miguel	Contributions
0.3	21/07/2019	José M. Martínez	Editorial checking
1.0	22/07/2019		First version
1.1	10/12/2020	Juan Carlos San Miguel	Contributions (new sections 4.2, 5, and 6.3)
1.2	14/12/2020	José M. Martínez	Editorial checking
2.0	14/12/2020		Second version

CONTENTS:

1. Introduction

1.1. Motivation

Work package 1 (WP1) aims at the initial establishment and maintenance of a development framework for the remaining work packages.

This deliverable describes the work related with the task T.1.2 Cameras network simulation which supports other tasks for generating test data. We focus on the simulator "Multi-camera System Simulator (MSS)" [1] to describe its structure and the developed features within the context of this project. Moreover, we also integrated other developments for the MSS simulator [2]

1.2. Document structure

This document contains the following chapters:

- Chapter 1: Introduction to this document
- Chapter 2: MSS description
- Chapter 3: MSS test scenarios
- Chapter 4: MSS APIs
- Chapter 5: MSS extensions
- Chapter 6: Framework performance
- Chapter 7: Conclusions and future work

2. MSS description

2.1. Software requirements

The requirements described below are the functionalities that must be developed to achieve the goals of this project.

- Remote work will be supported. The simulator will work as a server so one or more applications and algorithms will be able to use it simultaneously.
- The simulator will have based on a modern engine where we can recreate photo-realistic scenarios.
- Dynamic objects will be supported such as pedestrians, automobiles, or drones.
- Frames resolution will be able to adjust.
- Images per second (framerate) generated will be able to adjust between 1 and 30 for each camera.
- The area of the observable world (field of view) that is seen in each moment will be able to adjust.
- Each camera will have a buffer where frames temporary will be saved on main memory.
- All the parameters mentioned above will be able to configure individually and dynamically.
- To Create and to delete cameras.
- A broadcast for each camera.
- All cameras must be synced. This means that different cameras see the same world in each moment.
- There must be different speeds of the vehicles, overtaking on the streets.

The main purpose of this work is to develop a simulation tool which allows to handle multiple cameras into a virtual scenario. Additionally, broadcast messages can be generated from each camera (by sending frames through sockets) to external applications and vision algorithms. By using the Unity game engine where we find the appropriates features for the start point developing our system, we build up a complete multi camera visual simulator. This simulator is referenced as 'Multi-Camera System Simulator' (MSS). In **Figure 1**, the MSS architecture are outlined with a multi-client server design. The Client Server architecture makes remote work possible as well as local work. With the API client library developed, clients can communicate and to receive information through the methods included.

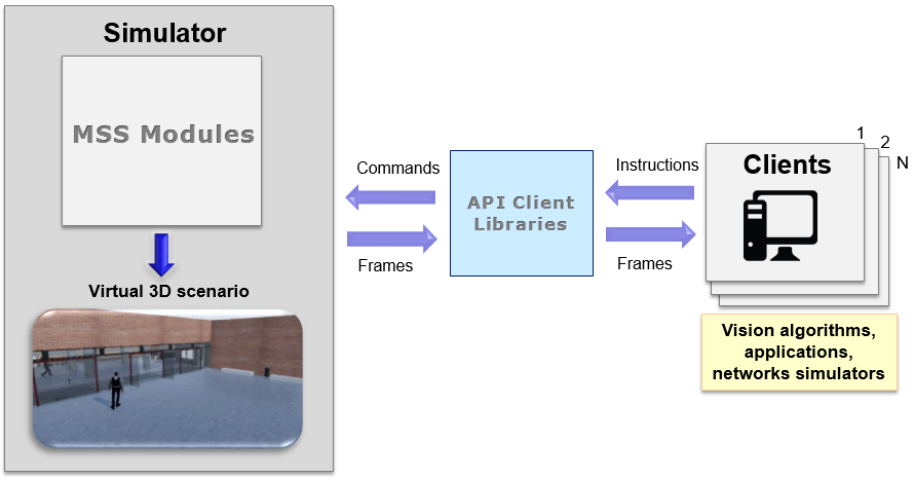

Figure 1. MSS incorporates a Client-Server architecture allowing multiple connections.

2.2. Communication between the simulator and clients

Before we explain the communication flow between clients and the simulator, we explain how works Computer Vision research conceptually. For example, we suppose that want to test a pedestrian detection algorithm. First, we need a dataset (images or videos) for giving material to the algorithm. A common dataset in Computer Vision research is a clip of a pre-recorded video. Now that we have the dataset and the algorithm, we develop a simple application with this process:

- First, we need to use some image processing library such as OpenCV for Reading the video frames and converting it in some object or class for further processing. A video is a sequential of frames that we can get one by one with this library.
- We read the first frame of the video, and we give it to the pedestrian detection algorithm.
- The algorithm analyses the frame looking for the shape corresponding to a human. It returns the frame but marked with the shapes that it has found with a rectangle.
- Now, we use a method that the image processing library contains for display a frame on screen. At this moment, we can appreciate the effectiveness of the algorithm visually trough the marks.
- We read the second frame of the video and repeat the process. We repeat these steps until all frames of the video are processed.
- Finally, with the results of all frames of the video, we can conclude the effectiveness of the algorithm.

2.3. 3D Models

2.3.1. Camera

For the implementation of our Cameras, we design a virtual pinhole camera model. Apinhole camera is a camera which has a small hole in the front called aperture or center of projection. The lights reflected for the objects passes through the aperture and projects an inverted image into a light-sensitive film paper. In a virtual pinhole camera, instead of a film paper, it has an object called render texture. When we take a snapshot in a virtual pinhole camera, it generates projection into the render texture of the objects that they are into its field of view (FOV). The size and shape of the objects in the render must match to the real size and shape of the objects.

- **Perspective projection**. In perspective projection, the distance between the apertura (center of projection) and the far plane is finite. The size of the objects varies according to the distance which gives a realistic aspect.
- **World points (X, Y, Height, Width).** These four values indicates where the camera is located into the virtual world, measured in absolute coordinates.
- **Target texture**. Reference to the render texture that contains the projection of the camera view, equivalent to the film paper explained before. We use this texture to generate frames and then store these frames in main memory temporary.
- **Framerate**. The numbers of frames generated in one second.

2.3.2. Scenario

Unity only provides a predetermine empty scenario. By default, there is an endless tridimensional space with any object created. If we render this scenario, we only can see a black screen because there are, literally, nothing to show. Because of this, we decided to develop an example scenario for testing our simulator. This scenario is the hall of the 'Escuela Politécnica Superior' (EPS) building 'Alan Turing'.

The scenario is composed by:

- A room with approximately 600 square meters.
- The entry of the building.
- Several characters walking around.

In the modelling of this scenario there are two kind of components: static objects and dynamic objects.

3. MSS test scenarios

3.1. EPS Lite

This system involves the communication between hardware unit due to the diverse technologies involved: GPU programming, image processing and sockets. This is how a scenario called EPS-Lite is developed for the MSS simulator as shown in **Figure 2**.

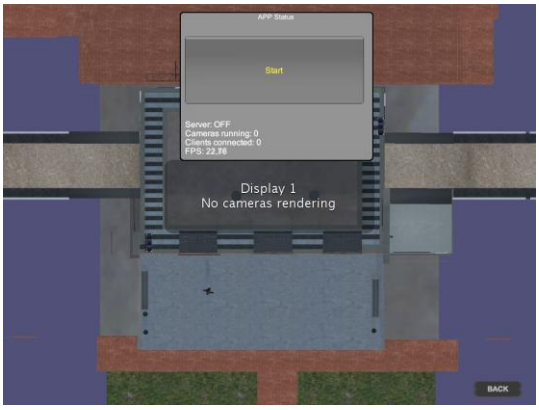

Figure 2. Scenario EPS-Lite.

3.2. EPS Full

As we see in the last chapter, the MSS simulator is composed by three different modules: Virtual Word, Buffer and Server. The main purpose of the Virtual World module is to interact with the virtual scenario trough the camera object which generates the frames that are sent to the applications and algorithms.

Figure 3. Scenario EPS-Full.

All classes of this module are synced and work sequentially. The Buffer module is where the frames generated by the cameras are saved temporary in main memory. It is necessary a previous image conversion. This module is implemented with multithreading due to an optimums image processing and avoid bottlenecks. Finally, the Server module has an asynchronous server prepared for multiple connections. and the scenario is extended to EPS-Full, view in Figure 3.

3.3. City Day

For the development of a scenario in which a city is simulated, we decided to start from a complete scenario Modern City Pack 4. It is a high-quality stage, with numerous static and periodical objects, details and textures, which is intended to add dynamic objects, responsible for providing events and situations of interest, view the Figure 4.

Figure 4. Image of the scenario Modern City Pack Day.

3.4. City Night

It also has two versions of the same stage, day and night. The approximate extension of the stage is about 65,000 m2. In Figure 5 you can see the scenario ModernCity Pack.

Figure 5. Image of the scenario Modern City Pack Night.

4. MSS APIs

Unity provides a scripting API with some interesting classes and methods which allows the interaction between the virtual scenario and your code. For example, there is a camera class 1with some basic properties useful for our simulator that we extend to develop our custom camera class with extra functionalities. Unity scripting API does not support multi-thread user code, but it tolerates. This means that we can use threads like any application, but threads can't use any Unity scripting API methods and classes. We only can employ threads for tasks which do not affect directly to the virtual scenario. For example, it is not possible to create one thread for each camera. However, it is possible to use threads for some classes like the server TCP. This is the reason only this module can use the API and works sequentially.

4.1. C/C++ API

In order to facilitate the communication between the simulator and applications, we supply client libraries that help to use the simulator and it reduces the amount of code needed in the development. The API is programmed in C++ which is a widely used language for Computer Vision research

Camera: in this library, the Camera struct is defined, the representation of the simulator camera object which is used for handling a camera in the client-side. With this library, clients can instance a camera object in their code by simply following an oriented object programming. Further, it includes methods to get and set properties.

Connection: this library contains all the methods relational with the communication between clients and the simulator using sockets.

Image Processing: image processing is the process of manipulating an image data in order to make it suitable for vision algorithms or applications. For example, an image conversion or changing contrast is a image processing task. In Computer Vision research, a wide image processing library is OpenCV.

User Interface: when a frame is received and converted with the Image Processing library, clients can have used this library to display it on screen. Further, it includes controls that allows a user to handle a camera in real-time.

The APIs developed as clients for the MSS simulator have a series of classes and at the same time certain functions for the creation of cameras in the Unity virtual world, these functions add cameras both manually and graphically, and it is done in such a way that the user can create an object with default parameters or in turn indicating the desired configuration such as location, position, resolution, image size, quality, among others.

The API implemented in Visual Studio contains the following Classes:

- **MSScam.cpp**: This class allows instantiating a camera-type object where it is assigned a name and all its previously mentioned values to be in the space within the simulator.
- **MSScam_control.cpp**: Provide an interface to move cameras on the MSS platform through a GUI (source file), this library includes third-party code (OpenCV GUI tools). You can observe the following functions.
- **MSScam_insert.cpp**: provides an interface to insert cameras in the MSS platform through a GUI.
- **MSSclient.cpp**: For basic functionality of a client that connects to the MSS server (source file)
- **MSSutils.cpp**: This is a private library for functions needed to use in some parts of code: conversions, maths methods (source file) and PropertyFileReader.cpp: Implementation of the property class.

4.2. Python API

The development of an API allows easy communication between applications, in this case it is provided libraries in the Python client to interact with the simulator scenarios. The API allows to have a very reduced code for the development of different applications in the client. The API programmed in the Python language brings competitive advantages in high-level programming,

since it is widely used today for machine vision research. Within the development of the API you will find the following libraries:

- **MSScam.py:** This library defines the structure of the cameras through which handle and control instances of the simulator camera object. Inside are different functions that allow, on the client side, to manipulate parameters regarding the properties of the camera. The builder of this class allows to instantiate a generic camera in the simulator, and the user can initialize the object as needed. This file is the which has more functions, since it controls aspects such as: assigning ID to each camera, registering the object in the simulator, manipulate image resolution and coding, frame transmission, position and rotation movements, etc.

- **MSSclient.py**: This class contains all the methods that relate to communication and creation between clients and the simulator. The communication established is through sockets. The library allows the API to manage parameters related to the simulator, for example: establish the connection, remove the client from the simulator, manage, manage in the simulation for frame transmission, number of cameras that the customer has created in the simulator, etc.

- **MSScam_control.py**: This library allows you to manage a graphical user interface for display the frames received from a camera and manage it in real time. This interface shows user options to manipulate the position and rotation of the camera, i.e. through control buttons, commands are sent along with the other classes to act on a camera.

- MSScam_insert.py - With the API, this class allows to instantiate and create a "user camera" through a graphical user interface. This means that a camera object can be inserted into some 3D position of the simulation scenario in a graphic way.

The implementation of the API allows to generate several examples by calling the methods of the incorporated library. This allows to develop examples with a more

generic and simple code of implementation, in the following figure you can see a general block diagram of the implementation and operation of the API in Python and some of the functions of the libraries it contains.

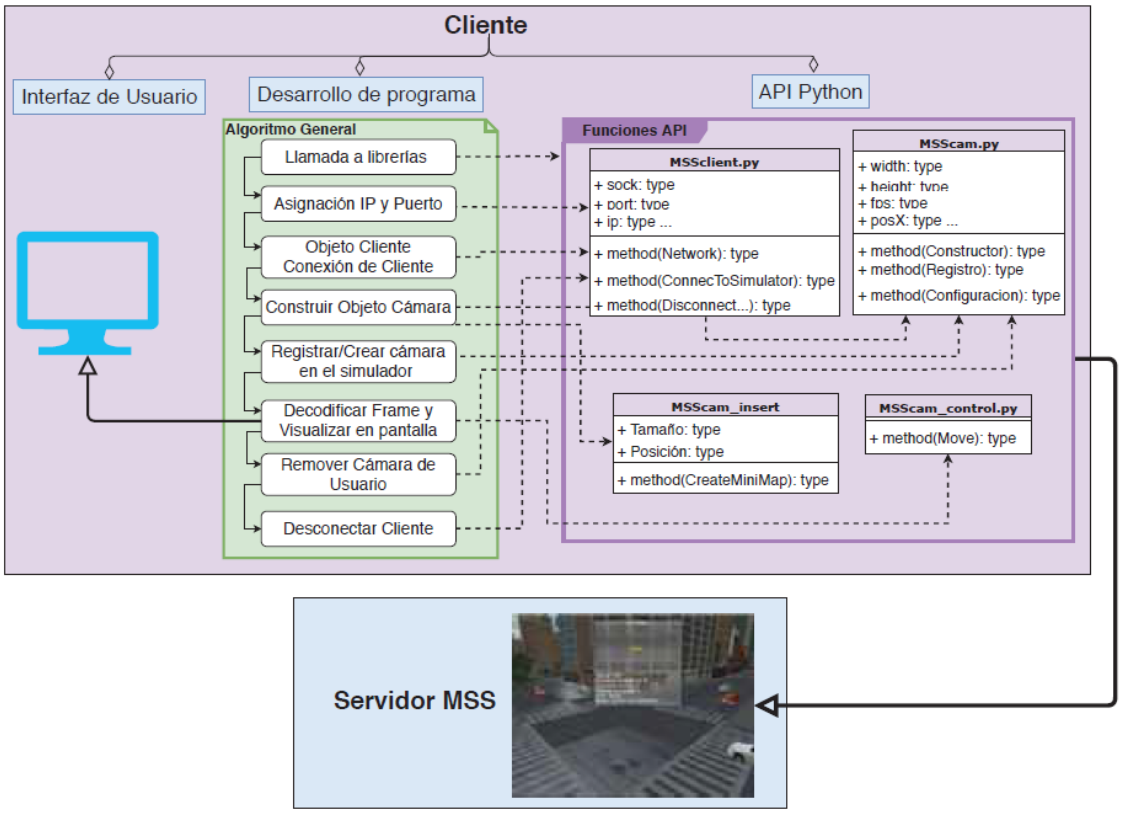

Figure 6. Block diagram of the implemented modules for the Python API.

5. MSS extensions

5.1. Capture modes: continuous and under demand

Because machine vision, deep learning or autonomous driving algorithms require a high computational capacity, this implies that the execution time of these algorithms in specific cases take longer than expected, so if it is executed within the simulator it is possible to lose the analysis of some frames when video is transmitted from the server to the client, due to this the user has the possibility to change the execution mode to type FrameOnDemand (transmitting images on customer demand).

The MSS simulator API incorporates both server and client enhancements to work in the situation described above, under two execution modes: Continuous Simulator and Frame On Demand Simulator.

- *Continuous* mode indicates that the simulator is running at "normal" speed, i.e. according to what is specified by the user when creating cameras on the stage or instantiating the ones is previously in the test environment.

- *FrameOnDemand* mode allows you to insert a "pause" time when server to retransmit each frame as the client requests the sending of the next frame, from so that no image is lost while the customer analyzes or processes some algorithm on the frame that arrived previously. With very small times the simulator will seem to go continuously but as time goes by you will be able to appreciate how the simulator pauses its execution for the requested period of time.

The following figure shows a flow chart showing the steps for generate the process of data capture in the two modes described. You can also observe in the following link https://vimeo.com/405353836 the example script of the execution FRAME ON DEMAND SIMULATOR on the simulator and its operation.

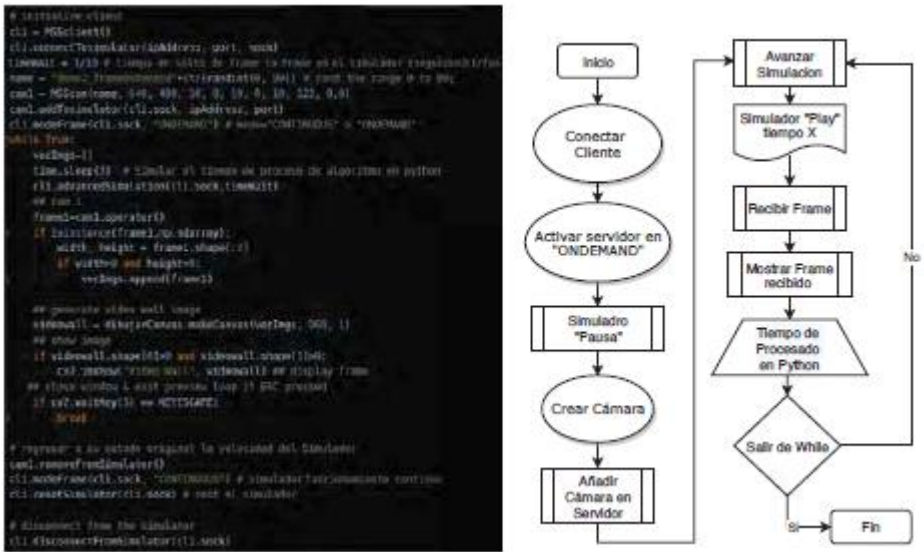

Figure 7. (a)Client Python code and (b) general scheme of operation of the capture mode "Frame on demand".

5.2. Camera types

The initial version of the simulator allows the manipulation and handling of the simulator cameras. In the previous work of MSS, you have a single object to instantiate cameras from the client. The The importance of incorporating improvements in MSS is due to the need to build new environments simulation tools that allow to deliver an added value to the functionalities of the cameras and testing machine vision algorithms. For the improvements made, the initial object (camera) is has called it "User Camera" because the customer is the one who requests to create a new camera in the simulator. With the renewal of the module in the Python API has been incorporated a new camera type object called "Scene Camera". This one has the particularity of being cameras already defined and installed in the scene, only prepared to initialize and transmit them.

5.2.1. Cameras: CameraScene y CameraUser

In the operation of the simulator initially an environment runs and by connecting of one or more clients, one or more cameras can be inserted into the simulator (pre-fab called CameraUser). In addition, a number of parameters must be passed to the "Camera" class as are; position, rotation, (3D spatial location), fps resolution, among others. You can see with more detail in the MSS original work. In a version with new

additions, it is possible to predefine within a 3D test scenario (Server) camera set up at any point or position of the simulator and access them through new functions of the client API (Python), these pre-made objects created in Unity (prefabs) have been assigned as CameraScene. Below you can see in the following cameras pre-installed in the scene (hierarchy of Unity's interface (white lines) and the one the user can create and position in the scene (blue).

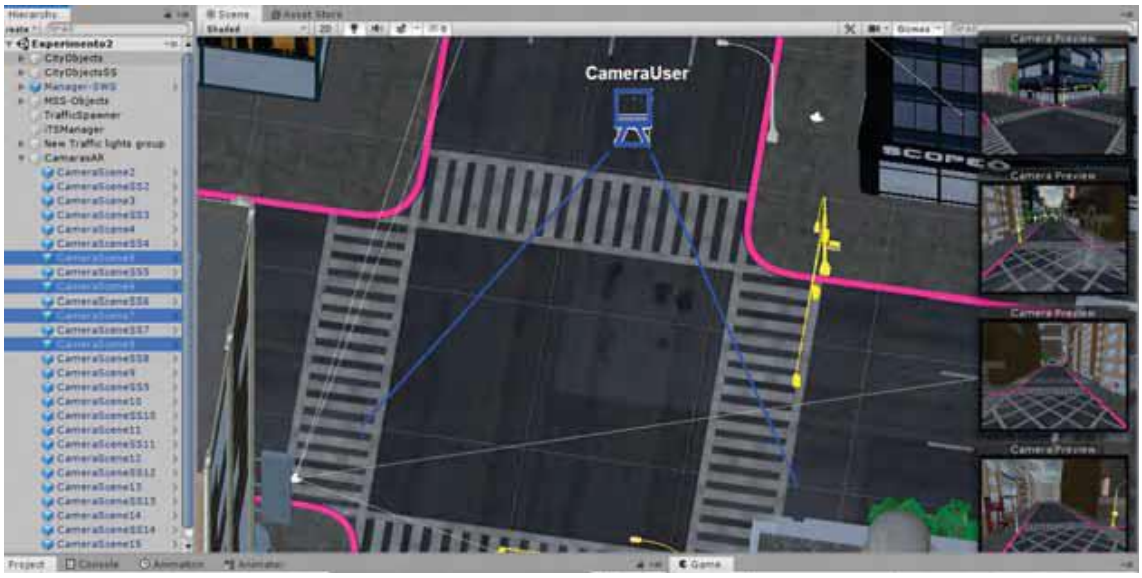

Figure 8. Scenario where the CameraScene type is pre-installed and where the user can create CameraUser

5.2.2. Type: Fixed Camera, Mobile Camera, PTZ Camera

The two prefabs CameraUser and CameraScene also incorporate an additional feature in which you can define what type of camera you want to insert, this badge can independently assign each camera object and there are three types: FixedCam, MobileCam and PTZ.

By default when inserting a camera is defined as FixedCamera which has the characteristic of being a static camera and without the possibility that the user can perform no action on it. If MobileCamera is selected it allows the user to control total over it i.e. by means of commands the user can move or rotate the camera over the whole scenario, and on the contrary a PTZCamera type camera, allows the client only the control over its rotation in two axes (X, Y) blocking completely the possibility of

move to the position camera. Its operation can be observed in the following link https://vimeo.com/405595414 . The following Table shows in detail the characteristics of this functionality in the cameras.

Table 1 **. Features of the different camera types**

Características	Eje	Fixed Camera	Mobile Camera	PTZ Camera
Posición	X	No	Si	No
	Y	No	Si	No
	Z	No	Si	No
Rotación	X	No	Si	Si
	Y	No	Si	Si
	Z	No	No	No

5.2.3. Cameras over moving objects

There is also an option (Special Cam) to improve and adjust the rotation movements in cameras that are inserted over moving objects. Currently there are security cameras located within public or private transportation. This means that these events can be replicated in the simulator so it is possible to find cameras inside a bus, a car or even a person on the move. The most suitable way of using this function is understand with the following steps:

- Insert an object that has some kind of kinematics in the scene (e.g. Helicopter, Car, Bus, etc).
- Program or generate the movement sequences to the inserted object.
- Create a new empty "GameObject" inside the inserted object. The name is assigned "TurnReference".
- Drag the camera prefab "CameraScene" into the newly created object and position it in the way you want. Then activate the Special Cam option.
- Run the simulator and access the camera to control the camera. The steps for the correct operation of this option are shown in more detail in the link: https://vimeo.com/411703677

For these cases the cameras base their position according to the "parent" object (in the hierarchy of Unity) to which it belongs, since they depend on the moving object. That is, its position is relative to the position of the moving object and this is relative to Unity's 3D world for which by activating this function it allows an adjustment in those parameters so that the user can manipulate it with better precision. It should preferably only be activated when a camera is inserted into the GameObject with a scroll in the scene, this function affects directly to PTZ cameras since the user will only have permission to manipulate the rotation of the camera (not its movement or displacement). To observe this function, you can please follow this link: https://vimeo.com/405602423.

5.3. Semantic map

This section briefly describes what semantic segmentation within vision is all about and the protocol carried out for its implementation within the simulator.

5.3.1. Definition

The main objective of the simulator is to create a distributed multi-camera system for the transmission video (set of images at certain FPS) on a client-server network, where the development of the simulations is carried out in exterior spaces (City Scenario), this allows segment in a concrete way the "sky", "ground", "sidewalks", "trees", "cars", "people", etc, which is considered the set of classes necessary for semantic segmentation. The following figure shows an example.

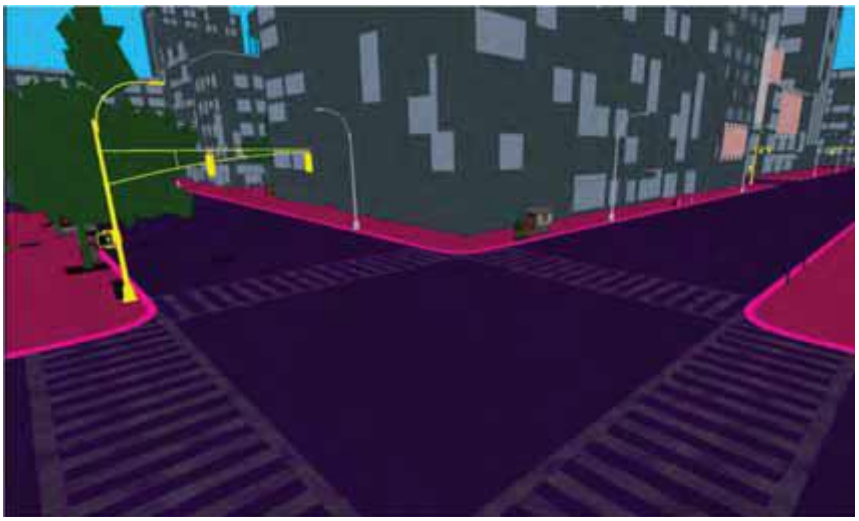

Figure 9. Sample segmentation map for the MSS simulator

5.3.2. Select number of classes

To generate semantic maps in the simulator scenes, a total of 12 classes, these can be added according to the needs and demands that require new scenes, it is say that with other simulations where new objects (or more classes per consequence) are incorporated a new color) such as bicycles, trains, planes, etc. can be added to the simulator.

5.3.3. Create materials and/or textures

Once the number of classes exist in the simulator, the step to follow is to create the different materials and/or textures of the colors that indicate the labeling to assign to each object in the scene, as seen in the following figure the assigned where, for the designer's consideration, a material is created that defines about how the surface should be rendered or otherwise a texture that serves for bitmap images, for more detail you can see Unity's https documentation: http://docs.unity3d.com/en/530/Manual/Shaders.html

Figure 10. Materials and textures to label each object on the stage

For sky labelling (sky or background class) it is necessary to manipulate the Clear Flag in the Camera component of the Camera object to select: skybox or solid Color (assign a background color).

5.3.4. Layer creation

The use of layers in Unity allows the camera to render only a part of the scene, so which can selectively indicate which objects may or may not be viewed by a camera. For which two new layers are created: MainStage (RGB rendering) and SemanticStage (rendering Semantic map).

In order for a camera to render a part or certain objects in the scene, it is necessary the use of Layers, in the city environment two new layers were added, such as can be seen in the following figure, the first MainStage in which all objects have their own material and texture as all the simulation examples (RGB) have been developed, and the second SemanticStage a custom layer where every object in the scene is classified with a label (color) according to the previously defined colors.

Figure 11. Creation of layers for real (RGB) rendering and semantic map

5.3.5. Layer and material assignment to objects

Once the materials and layers have been created, they must be assigned, this means that each material will be set to label the objects and one of the two layers to indicate to the cameras the rendering of certain objects in a scene. As we can see in the following figure the object must be selected and in the inspector in the parameter Layer assign the corresponding layer to it.

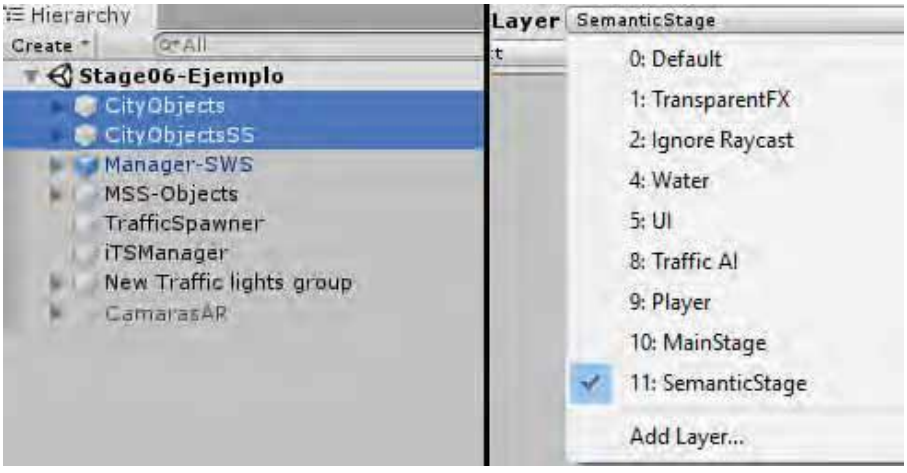

Figure 12. Assigning a certain layer to one or more of the objects in the scene.

5.3.6. Create a scene with tagging and semantic mapping

When implementing a semantic map in the simulations, the procedure followed is:

1. Identify the objects of the "CITY" scenario within Unity's Hierarchy in order to classify them according to their class.

2. Duplicate (see Figure 3.14) all existing objects that will appear in the simulator. This way you will have all the existing objects repeated twice, so one will serve to have a "normal" scene (RGB) and the other to label with colors and have the semantic map (ground-truth). It is necessary to take into account that for a better performance of the simulator it is recommended to have the least amount of possible objects in a scene, so you can look for alternatives to this method to incorporate a semantic layer.

3. Assign to each object the corresponding material according to its class (person, bus, building, vegetation, etc)

4. Assign each duplicate object the SemanticStage layer.

5. Assign the MainStage layer to the main objects, these objects will have the textures and original materials, i.e. RGB colors.

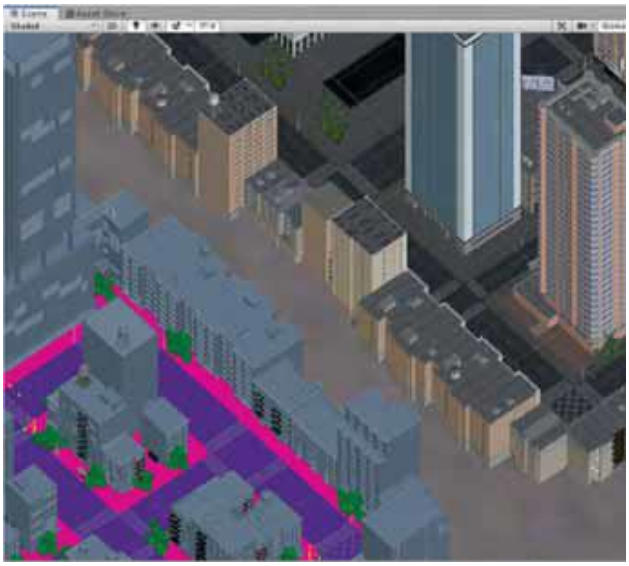

Figure 13. Duplicate all objects (city) to assign different layers an RGB and another semantic map to label each object according to its class.

The purpose of duplicating and assigning layers to the objects in the scene is that by instantiating a camera in the simulator can render the objects on it according to their layer, that is if the camera is instructed to display the MainStage layer, only those objects assigned to it are displayed to that layer, while if SemanticStage is selected the camera will only project the objects assigned to that particular layer.

5.3.7. Visualization and Rendering Management

After assigning the layers and the labeling of the objects it is necessary to manage when observe the objects in RGB and when to display the objects sorted by color, so such management is done with the Management SS script. This code has been developed to add to the CameraUser and CameraScene cameras and control whether the camera renders the objects with the MainStage layer (RGB objects) or objects with the SemanticStage layer (semantic map).

To manage which layer the camera renders, the Culling Mask variable is used, which allows you to customize the way frames are rendered when you instantiate a camera in the scene. To understand in a simple way the creation of a complete scene step by step is available at https://vimeo.com/416959724 .

6. Framework performance

During the development of this project, unit tests has been made. However, as a final test, we want to verify that the simulator works accord requirements analysed. In particular, we realize a black box testing which is technique used on functional testing to examine the functionality develop in a system. With this testing technique, the internal working of the system is not relevant for the tester. Instead, the tester has a list of inputs and what the expected outcomes should be. In this experiment we design some different common actions and the output expected. After, researchers from the VPULab research group (Universidad Autónoma de Madird) tested several times each action and log the results. These results are depicted in the following Table.

Table 2 Functionality testing.

Action	Expected	Result
Connect to the simulator	Confirmation message on the console.	OK
Create a camera with any configuration.	Confirmation message on the console.	OK
Create a camera trying different resolutions.	Confirmation message on the console	It does not work with resolutions lower than 100x100.
Create two cameras with the same name.	An error message in the second camera because the name works as an identifier that must be unique.	Confirmation message in both camera.
Create a camera with a framerate bigger than 30 fps.	Camera's framerate at 30 fps.	OK
Delete the camera created	Confirmation message on the console.	OK

6.1. Scenario performance

In all experiments we use a compiled version from the project which includes the three modules developed and the EPS building scenario designed. This version is a standalone application for windows that we run on this computer running Windows 7 64 bits SP1 with the following specifications: Intel Xeon E5-2630 v3 @ 2.40 GHz (16 cores and 64 GB RAM). Moreover, we use a powerful GPU with the following details: NVIDIA GeForce TITAN X 12GB GDDR5 (3072 CUDA Cores).

In Figure 6, we appreciate that it takes about 40 milliseconds to generate a Full HD image (1920x1080) so the simulator is able to generate a maximum framerate of 25 frames per second (fps) with different camera configurations, for example, one camera at 25 fps or 5 cameras at 5 fps. Although this result seems not to be really impressive, we have to consider that many Computer Vision algorithms use a standard resolution of 640x480 because working with bigger images has an extremely high computational cost. This resolution 640x480 is the best option for our simulator, with a reasonable rate of 80 images per second. On the other hand, between quality graphics option have any difference in the average time. We conclude quality graphic is not a parameter that affect significantly the frame generation process.

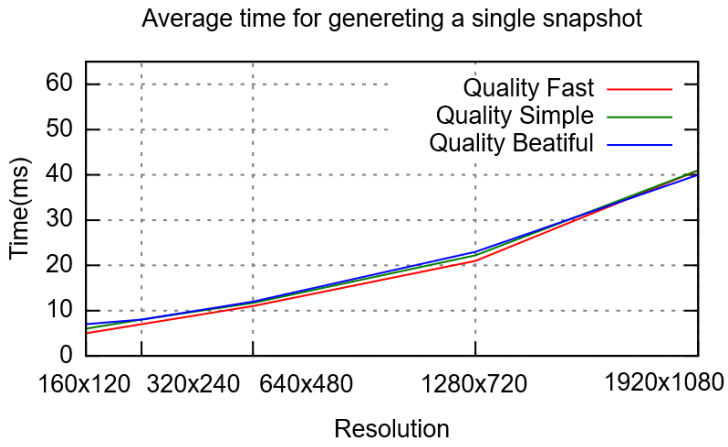

Figure 14. Time to generate one RAW frame for different resolutions and graphic qualities.

In all the experiments we use a compiled version with the scenario EPSlite, as reference or base, and four compiled versions of the city's scenario, three of them with the daytime scenario with different densities of dynamic objects (low, medium and high), and another the night scene. These versions are applications running on a Windows 10 64bit with the following specifications: Intel Core I5-3330 @ 3.00 GHz (4 cores and 8GB RAM), and with a simple graphics card: Intel HD Graphics. For each of the experiments, a client that configures the conditions of the experiment is connected to each compiled instance. Each experiment was executed for a period of time of about 5 minutes, obtaining data every half second, with the purpose of collecting the necessary data for graphics (in total 600 data for each configuration of each experiment).

6.2. C++ API performance

We evaluate the frame conversion process from RAW format to both JPEG and PNG format with the purpose to find out the maximum framerate for each format. First, we measure the average time to convert one frame for different resolutions using a single camera running in the simulator. Later, we compare the image size between formats. The decoders employed in this experiment are the decoders include in the on GDI+ Windows API. As depicted in Figure 7, the time needed to convert an image is not important when operating at small resolution, but changes from 1280x720 resolution or higher. For example, we have a rate of 14 images processed per second (72 ms/frame) with 1920x1080 but it is clear that there is a bottleneck as compared to the frame generation process at different qualities (previous subsection) which takes around 40 ms for the same 1920x1080 resolution. We also observe higher computational cost with the PNG decoder as compared to the JPEG one, specifically for the 640x480 resolution: 41 ms (25 fps) of PNG instead of 16 ms (60 fps) of JPEG.

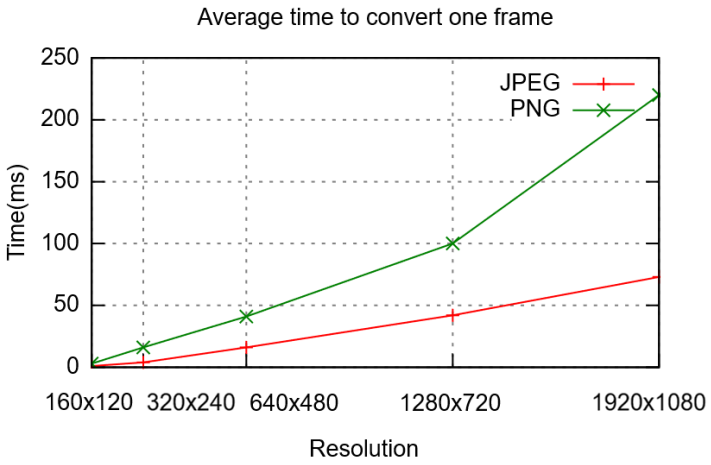

Figure 15. Average conversion time for different resolutions and encoders (JPEG and PNG).

Additional experiments are available at https://repositorio.uam.es/handle/10486/677829 and https://repositorio.uam.es/handle/10486/688104

6.3. Python API performance

This evaluation compiles 3 different scenarios, each of which increases the number of CameraScene (2, 10 and 30), for which each RGB camera that is placed in the simulator at the same time will have another camera with similar characteristics but with its semantic map layer, so the variable scene cameras (CS) will reach a maximum of 30 (15 RGB and 15 with semantic map) which are distributed in the scene.

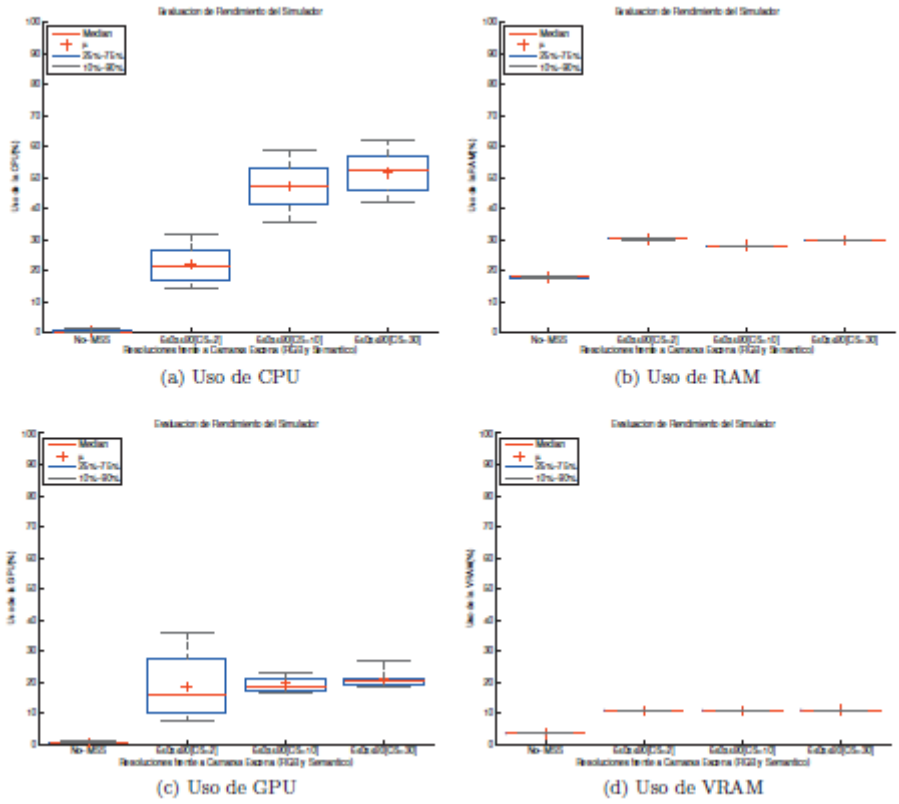

Figure 16. Performance setting medium resolution and simulator compiled with 2, 10 and 30 stage cameras (CameraScene)

In the evaluation of the performance of resources for this experiment it can be seen in the previous figures the 640x480 case [CS=2] is a scenario compiled with two CameraScene type cameras, where its CPU and GPU performance is better than the following two cases presenting a percentage of CPU usage of 20% and GPU usage of %15 (approximately), it is important to note that for this the frame rate delivered by the simulator is FPS=30. In the following cases, 640x480 [CS=10] and 640x480 [CS=30]

clearly shows that the percentage of use increases as the CS variable CameraScene also increases, however the amount of SPF observed in the simulator is reduced at FPS=27 and FPS=11 respectively. Figures b and d have similar performance in all cases, this is because the simulator only runs one window which leads to the resolutions of the images are the same.

For this experiment, a single compilation of the scene has been used. This executable file is the one that was built with 30 stage cameras (CS=30, RGB=15 and Semantics=15), in this experiment the client instances from a script the visualization of: 2, 10 and later 30 cameras. Therefore to each case has been named "CS=30[Cli=2,10,30]" (compiled simulator = 30 cameras [display of the client = X cameras]). The results are presented in the following figure:

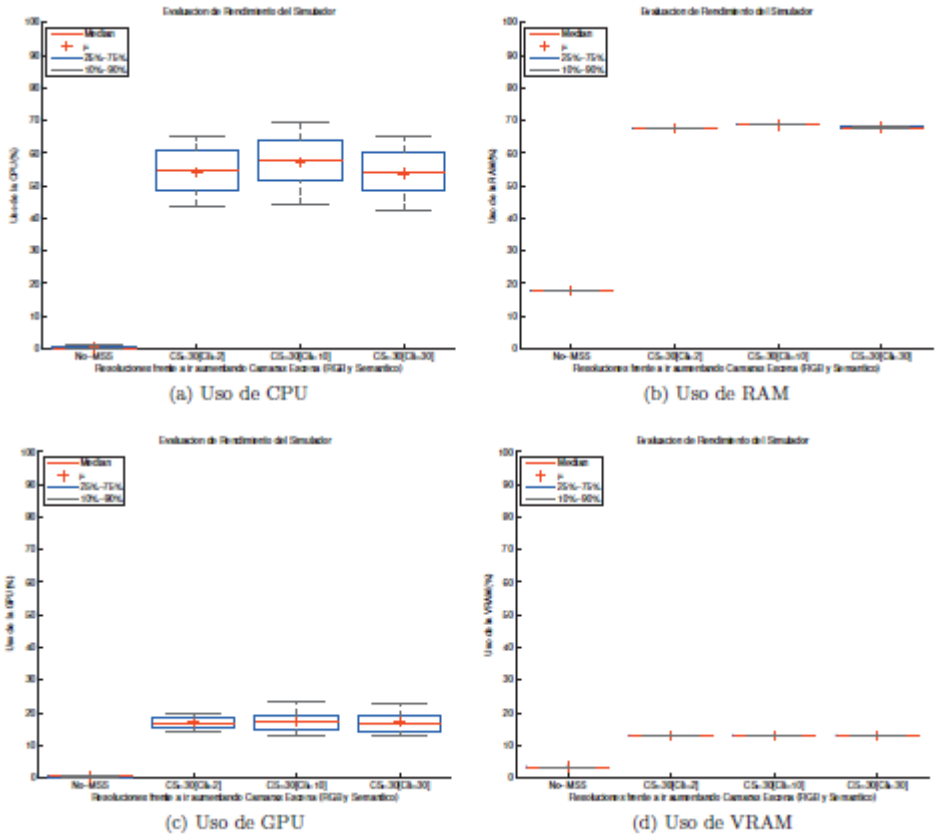

Figure 17. Performance of the simulator compiled with 30 scene cameras (CameraScene) calling 2, 10 and 30 camera instances from client

As you can see the previous subfigures the performance of the simulator is similar since in all three cases the compiled file is the same, this performance remains within the same parameters as the previous experiment with the 640x480 case [CS=30] since it is the same executable. On the other hand in subfigures b and d the performance increases since in this case when using a higher percentage of RAM and VRAM is required to display the cameras, which is reflected in the percentage of use. It is important to note that for these three cases the variable SPF=10 but observing in the experiment this is not so, the amount of FPS reflected by the simulator is between 5-6 FPS so there is a clear reduction in the frame rate

Additional experiments are available at https://repositorio.uam.es/handle/10486/692616

7. Conclusions and future work

In this document we present a suitable simulator tool for Computer Vision research. This simulator can be used for designing, testing and debugging vision algorithms but also can provide input data for smart-camera simulators like WiSE-Mnet++, the holistic SNC simulator. By using the API client libraries developed, you can easily adapt an existing application to communicate to the simulator without change the logic or the behavior of your systems or applications. One of the benefits of this simulator unlike others existing simulators, is that is based in a modern game engine (Unity). Thanks to this game engine, which is license free, one can develop photo-realistic virtual worlds, including customization AI and dynamic objects such as pedestrian or automobiles. Further, it can be programmed weathers conditions or any interaction with the virtual world through the Unity scripting API.

Future works are related to:

- Work on the pre-API developed in multiple programming languages, such as Matlab and puython, checking its operation for the latest versions of the simulator and openCV libraries. Develop examples of clients connected to the MSS simulator using the functions developed in the pre-API worked.
- Simulator improvements, in terms of various aspects; Implement mobile cameras within simulation environments to try out and monitor wide ranges of a scene through a function that allows moving the cameras generated in the MSS. Another aspect is the automatic annotation and finally make a study of the performance of the GPU used for the operation of the simulator and the costs involved in the software implemented
- Application of the City + AI city Challenge track 3 scenario, to make transport systems more intelligent, based on data from traffic sensors, signalling systems, infrastructure and traffic. Unfortunately, progress has been limited for several reasons, including poor data quality, lack of data tags and lack of high quality models that can convert data into actionable

information. There is also a need for platforms that can handle the analysis from edge to the cloud, which will accelerate the development and implementation of these models. Therefore, the MSS tool allows to generate simulated data for the identification of traffic and implement the algorithms developed for the identification of events of interest among these:

- City-scale multi-camera vehicle tracking
- City-scale multi-camera vehicle re-identification
- Traffic anomaly detection – Leveraging unsupervised learning to detect anomalies such as lane violation, illegal U-turns, wrong-direction driving, etc.

References

[1] Mario González Jiménez, “Sistemas multi-cámara distribuidos basados en Unity”, Trabajo Fin de Grado, Escuela Politécnica Superior, Universidad Autónoma de Madrid, Febrero 2017. https://repositorio.uam.es/handle/10486/677829

[2] Francisco Lobo García, “Desarrollo de escenarios para simulador multi-cámara basado en Unity”, Trabajo Fin de Grado, Grado en Ingeniería Informática, Escuela Politécnica Superior, Universidad Autónoma de Madrid, Junio 2018. https://repositorio.uam.es/handle/10486/688104

[3] Vinicio Pazmiño Moya, “Contribuciones a la simulación de sistemas de vídeo-seguridad con múltiples cámaras”, Trabajo Fin de Master, Master en Ingeniera de Telecomunicación, Escuela Politécnica Superior, Universidad Autónoma de Madrid, Septiembre 2020, https://repositorio.uam.es/handle/10486/692616