TEC2017-88169-R MobiNetVideo (2018-2020)

Visual Analysis for Practical Deployment of Cooperative Mobile Camera Networks

D1.2 v1

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
1.0	14/07/2019	Juan Carlos San Miguel	Contributions
1.0	21/07/2019	José M. Martínez	Editorial checking
1.0	22/07/2019		First 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 5: MSS APIs
- Chapter 6: MSS experiments
- Chapter 7: Conclusions

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 a given 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 a given 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.

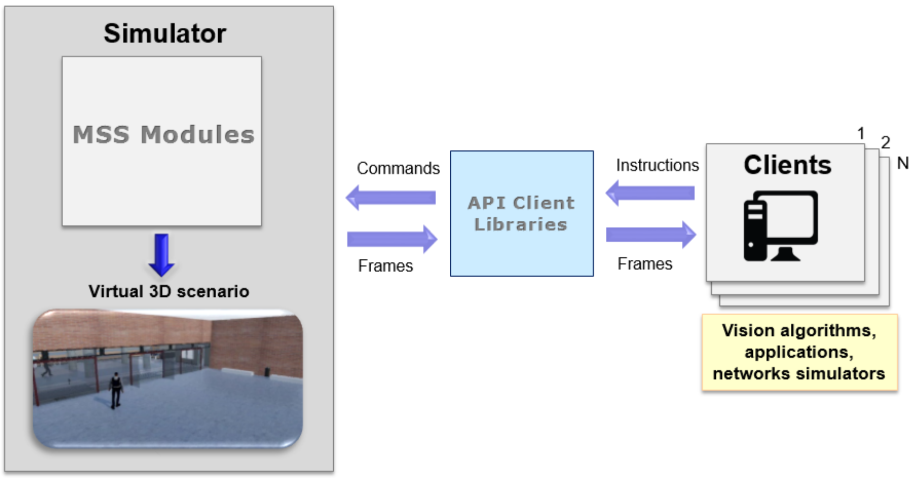


Figure 1. MSS incorporates a Client-Server architecture allowing multiple connections.

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 in to 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 in to 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.

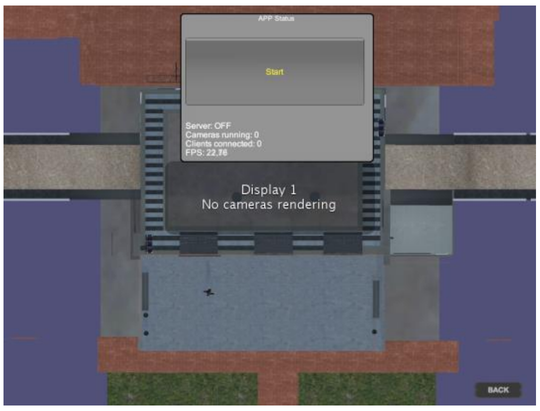
- Several characters walking around.

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**.

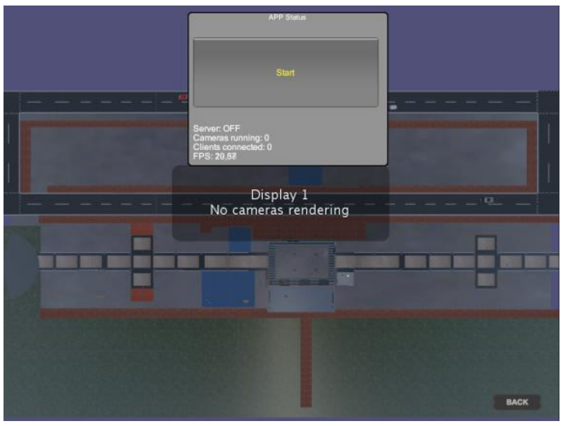


Server: OFF
Cameras running: 0
Clients connected: 0
FPS: 22.76

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.



Server: OFF
Cameras running: 0
Clients connected: 0
FPS: 20.57

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 bottle-necks. 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.

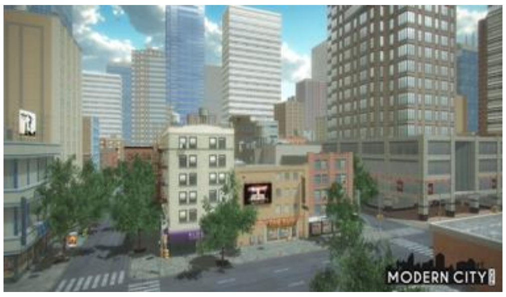


Figure 4. Image of the scenario Modern City Pack Day.

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.

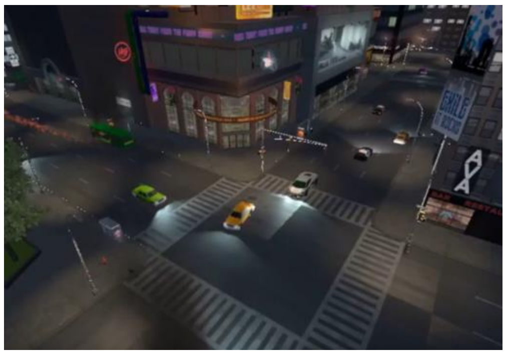


Figure 5. Image of the scenario Modern City Pack Night.

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 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 instanced a camera object in their code by simply following an oriented object programming. Further, it includes methods for 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 located 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.

5. 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 Table 1.

Table 1 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

5.1. 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.

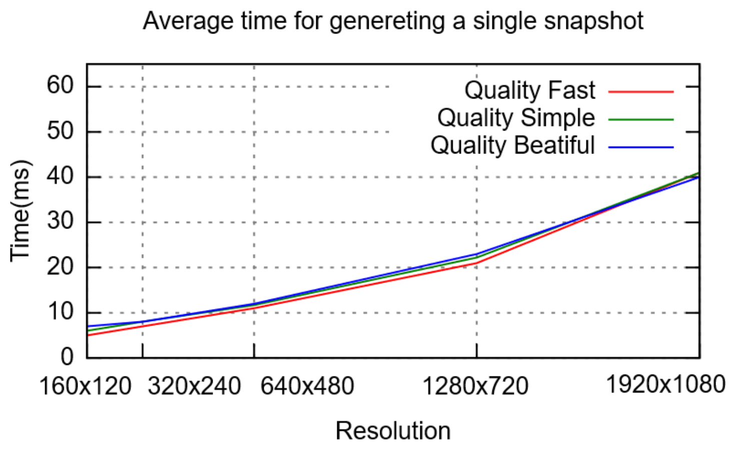


Figure 6. Time to generate one RAW frame for different resolutions and graphic qualities.

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).

5.2. 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.

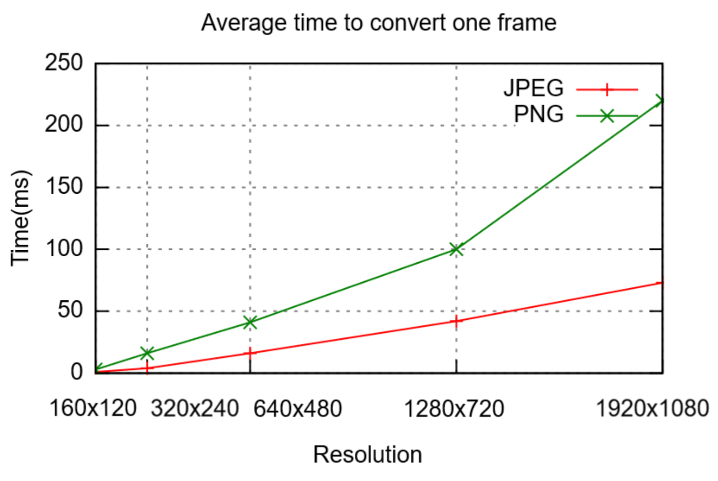


Figure 7. Average conversion time for different resolutions and encoders (JPEG and PNG).

6. 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, Febrero 2017.

[2] Francisco Lobo García, “Desarrollo de escenarios para simulador multi-cámara basado en Unity”, Trabajo Fin de Grado, Junio 2018.