UNIVERSIDAD AUTÓNOMA DE MADRID
ESCUELA POLITÉCNICA SUPERIOR

Grado en Ingeniería Informática

TRABAJO FIN DE GRADO

SISTEMA MULTI-CÁMARA DISTRIBUIDO BASADO EN UNITY

Mario González Jiménez.
Tutor: Juan Carlos San Miguel Avedillo.
Ponente: José María Martínez Sánchez.

Enero 2017

SISTEMA MULTI-CÁMARA DISTRIBUIDO BASADO EN UNITY

Mario González Jiménez
Tutor: Juan Carlos San Miguel Avedillo
Ponente: José María Martínez Sánchez

Video Processing and Understanding Lab
Departamento de Tecnología Electrónica y de las Comunicaciones
Escuela Politécnica Superior
Universidad Autónoma de Madrid

Enero 2017

Trabajo parcialmente financiado por el Ministerio de Economía y Competitividad del Gobierno de España bajo el proyecto TEC2014-53176-R (HAVideo) (2015-2017)

Resumen

Este Trabajo Fin de Grado (TFG) presenta una herramienta software para simular sistemas de visión artificial inteligente con múltiples cámaras en entornos virtuales 3D generados mediante técnicas de gráficos. Esta herramienta aporta una alternativa en la investigación en el ámbito de visión artificial (*Computer Vision*) que trata de solventar las limitaciones técnicas y flexibles de los entornos de pruebas más comunes (e.g. videos grabados o un sistema de múltiples cámaras en red). El objetivo de este TFG consiste en ofrecer un entorno de pruebas con datos sintéticos para la investigación de algoritmos en *Computer Vision* que sirve para simular situaciones en una gran variedad de entornos y condiciones, algo difícilmente replicable en el mundo real. Esta herramienta está basada en el motor gráfico Unity que permite diseñar escenarios y entornos 3D realistas y extiende su funcionalidad con un sistema capaz de manejar múltiples cámaras virtuales distribuidas en tiempo real, así como la configuración de las propiedades de dichas cámaras y la posibilidad de transmitir a otras aplicaciones los frames capturados por cada una. Mediante una arquitectura cliente-servidor, la herramienta permite la configuración de las camaras y su uso tanto en remoto como en localhost. Por último, se evalúa el rendimiento del sistema desarrollado y se discuten los resultados, estableciendo configuraciones recomendadas y descubriendo los límites técnicos del simulador.

Palabras clave

Visión Artificial, visión por computador, simulación, sistema distribuido de múltiples cámaras, diseño 3D, Unity.

Abstract

This Bachelor Thesis presents a software tool to simulate multi-camera systems using 3D virtual data. This tool offers an alternative to traditional Computer Vision research by addressing the technical and flexibility limitations of common testing environments that are based on pre-recorded videos or live video streams from network cameras. The objective of this Thesis is to provide a flexible testing environment for Computer Vision algorithms where it can be simulated virtual scenarios which cannot be easily replicated with real world. This tool is based on the Unity game engine that allows to design and to load realistic 3D environments and extends its funcionality with a multi-camera system able to simulate the management of several distributed virtual cameras in real time, as well as setting their properties and transmitting the captured frames to third party applications (i.e. algorithms). By using a Client-Server architecture, this tool allows the remote configuration and use of the system as well as in localhost mode. Finally, system performance is evaluated and results are discussed, establishing recommended configurations and technical limitations for successful operation.

Keywords

Computer Vision, Simulation, Distributed Multi-Camera System, 3D Enviroment, Unity.

Agradecimientos

En primer lugar, me gustaría agradecer a mi tutor del TFG Juan Carlos San Miguel Avedillo su ayuda, tiempo y disponibilidad durante más de un año de trabajo manteniendo siempre el entusiasmo e interés por el trabajo, así como la posibilidad de conocer y trabajar en el grupo de investigación VPU, departamento que se encuentra en el edificio C de la Escuela Politécnica Superior y que no conocía previamente a la realización del TFG, lo cual, en mi opinión, es una muestra del deficiente pero mejorable funcionamiento de la universidad en su labor como lugar de divulgación e investigación científica para los alumnos.

También me gustaría agradecer y dedicar a mis padres este trabajo, por el apoyo económico que he necesitado todos estos años mientras cursaba el grado y que de otra forma quizá no podría haber cursado.

Por último y más importante, me gustaría agradecer a varios profesores de la escuela por su trabajo, a destacar Xavier Alamán, Miren Idoia Alarcón Rodríguez, Pablo Fernández Gallardo y Fernando Maestre Miranda, los cuales no se dedican a impartir un temario determinado, si no que transmiten mucho más: pasión por tu profesión, la enseñanza, el trato humano, el conocimiento, la informática... Entre todos ellos me han ayudado a crecer como persona e intelectualmente y a valorar esta profesión.

Contents

Resumen v

Abstract vi

Agradecimientos vii

1 Introduction 1
1.1 Motivation . . . 1
1.2 Objectives . . . 2
1.3 Document Structure . . . 3

2 State of the art 4
2.1 Game Engines . . . 4
2.1.1 Comparison . . . 4
2.2 Camera Simulators . . . 5
2.2.1 Introduction . . . 5
2.2.2 OVVV . . . 5
2.2.3 UnrealCV . . . 6
2.2.4 SLCNR . . . 7
2.2.5 Virtual Pedestrian . . . 7
2.2.6 Conclusion . . . 7

3 System design 9
3.1 Requirements analysis . . . 9
3.1.1 Functional requirements . . . 9
3.1.2 Non-functional requirements . . . 10
3.2 System Overview . . . 10
3.3 Communication between the simulator and clients . . . 11
3.4 Logic View . . . 12
3.5 Unity extension . . . 13
3.6 3D Models . . . 14
3.6.1 Camera . . . 14
3.6.2 Scenario . . . 15

4 Implementation 17
4.1 Hardware architecture . . . 17
4.2 Software architecture . . . 18
4.3 Module 1: Virtual World . . . 19
4.3.1 Unity scripting API . . . 19
4.3.2 Camera object . . . 19
4.3.3 Camera class . . . 19

4.3.4 Cameras synchronization logic . . . 20
4.3.5 Cameras Controller Class . . . 22
4.4 Module 2: Buffer . . . 22
4.4.1 Buffer class . . . 22
4.4.2 Encoder class . . . 23
4.5 Module 3: Server . . . 24
4.5.1 Server TCP . . . 24
4.5.2 Commands Manager Class . . . 24
4.6 API Client Libraries . . . 25
4.7 Communication protocol . . . 26
4.8 Applications . . . 27
4.8.1 Dummy camera . . . 27
4.8.2 Pedestrian detection . . . 27

5 Experimental work 28
5.1 Set-Up conditions . . . 28
5.2 System Testing . . . 29
5.3 Simulated Scenario . . . 30
5.4 Frame generation performance . . . 31
5.5 Frame conversion performance . . . 32
5.6 CPU and GPU usage . . . 33
5.6.1 Framerate . . . 33
5.6.2 Resolution . . . 34
5.6.3 Several Cameras . . . 35
5.7 Network Usage . . . 38
5.8 Application example: people detection . . . 38
5.9 Conclusion . . . 39

6 Conclusions and future work 40
6.1 Conclusions . . . 40
6.2 Future work . . . 41

Bibliography 42

A Tutorial for adding the simulator in to a project in Unity 45

B Game engines 49
B.1 Introduction . . . 49
B.2 Unreal Engine . . . 50
B.3 Amazon Lumberyard . . . 50
B.4 Unity . . . 51
B.5 Uses in research . . . 51
B.6 Conclusions . . . 51

List of Figures

1.1 Image recognition example . . . 1
1.2 Synthetic data used in Computer Vision simulators: (a) Moving points ([1]) and (b) virtual videos ([2]). . . . 2

2.1 OVVV Simulator example . . . 6
2.2 UnrealCV architecture . . . 6
2.3 View of the virtual world of SLCNR . . . 7

3.1 MSS architecture . . . 11
3.2 MSS default screen . . . 12
3.3 MSS Modules . . . 13
3.4 Camera FOV scheme . . . 15

4.1 Hardware architecture . . . 18
4.2 MSS Modules . . . 18
4.3 Camera GameObject . . . 20
4.4 Cameras synchronization . . . 21
4.5 Cameras Controller Diagram . . . 22
4.6 ThreadPool Scheme . . . 22
4.7 Buffer module logic view . . . 23
4.8 Server Module Logic View . . . 24
4.9 Server TCP diagram . . . 25
4.10 Scene Simulation . . . 26
4.11 Screenshots of the dummy application . . . 27

5.1 Examples of 3D humans models in the scenario: (a) police and (b) student. . . . 30
5.2 Comparative between (a) pictures of the 'Alan Turing' building and (b) virtual scenario modeled. . . . 30
5.3 . Different graphic quality options: (a) Fast (b) Simple and (c) Beautiful. . . . 31
5.4 Time to generate one RAW frame for different image resolutions and graphics qualities. . . . 31
5.5 Average time to convert a single frame for different resolutions and coding algorithms (JPEG and PNG) . . . 32
5.6 Average conversion size for different resolutions and encoders (JPEG and PNG). 33
5.7 Comparative of image quality with a 300% zoom applied. . . . 33
5.8 CPU usage with one camera and different framerates. . . . 34
5.9 CPU usage with one camera and different resolution. . . . 35
5.10 CPU usage with several cameras simultaneously. . . . 36
5.11 Real Framerate regarding several cameras working simultaneously. . . . 37
5.12 Pedestrian detection results. . . . 39

A.1 MSS source folder structure 46
A.2 Unity start screen 46
A.3 Unity User Interface 47
A.4 Unity User Interface 48
A.5 Unity User Interface 48

B.1 Game Engine components 49
B.2 Unreal Engine Tech demo 50
B.3 Virtual Kitty dataset example 52

List of Tables

2.1 Table comparative of game engines. 4
2.2 Table comparative between visual camera simulator. MSS is the simulator that this work presents. 8

3.1 Table comparative. 14

5.1 Tools used in the experiments. 28
5.2 Functionality testing. 29
5.3 GPU Resources with one camera grouped by framerate 34
5.4 GPU Resources with one camera grouped by resolution 35
5.5 GPU Resources with several cameras working. 37
5.6 Network Usage in different cases. 38

B.1 Table comparative of game engines. 52

Chapter 1

Introduction

1.1 Motivation

Computer Vision is an interdisciplinary area with the objective of a high-level understanding from digital images or videos through computer algorithms. Computer vision has an enormous potential to change our lifestyle and it has recently become important in a variety of application areas, ranging from established areas like visual surveillance, multimedia and machine vision to emerging areas [3] like driver assistance, vision-guided surgery, assistive robotics, and even marine fishery [4]. For example, Figure 1.1 shows an on-board car camera detecting objects in real time while driving. Computer vision research is challenging due to many factors such as the huge variety and complexity of visual content, real-time requirements, limited resources and intercommunication between systems.

To design vision algorithms and test new applications, a suitable simulator is useful to support the development before deployment. This is particularly important for evaluation and training due to the complexity involved in obtaining real-world controlled environments where the situations of interest may need to be repeated as many times as necessary in order to test or to train algorithms. For example, we would like to identify a person leaving an object (e.g. bag, luggage) on an airport and track him across multiple views obtained from a multi-camera system. In this scenario, there are many cameras involved that can be used in different ways depending on algorithm used (e.g. selecting views where the person is fully visible, choosing the best view for tracking or even moving the cameras as the person moves). For best results, it may be necessary to repeat the same scenario with different parameters (e.g. density of people, complexity of the action) and perhaps different camera movements. Real-world data does not allow such kind of repetition for algorithm testing.

In this context, simulation tools provide flexible testing environments instead of real-world

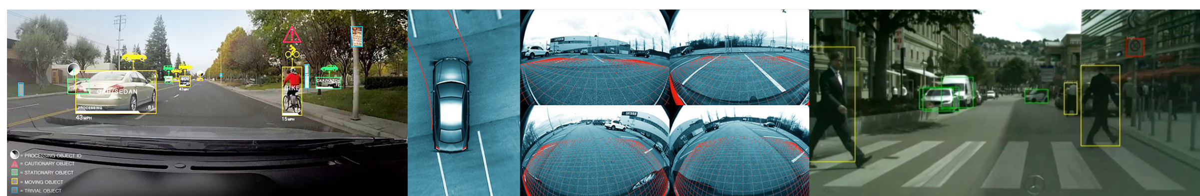
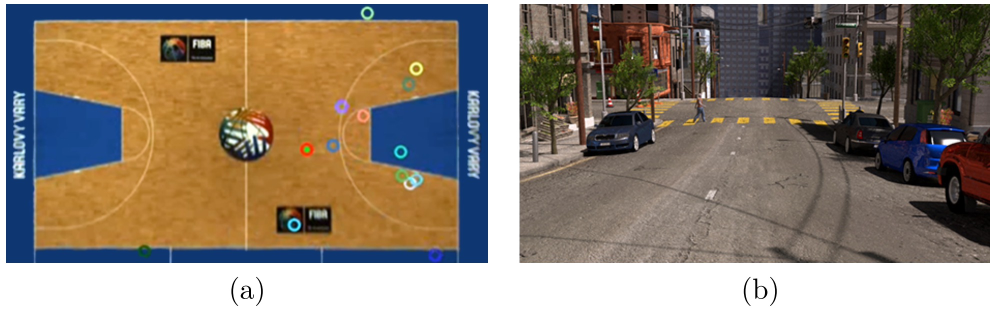


Figure 1.1: Image recognition examples provided by the Nvidia Driver PX2 system, an open AI platform for autonomous vehicles. Source: [5]



(a) (b)

Figure 1.2: Synthetic data used in Computer Vision simulators: (a) Moving points ([1]) and (b) virtual videos ([2]).

systems in order to emulate real scenarios. Such tools use synthetic data that can be broadly categorized into virtual videos or moving points (example in Figure 1.2).

- **Moving points:** objects are represented as simple moving points on a common coordinate system (e.g. ground plane or orthographic projection [1]).
- **Virtual videos:** video data from virtual 3D worlds generates with game engines such as Unity[2]. They must mimic real conditions so the obtained algorithm performance can be extrapolated to real-world systems.

One of the primary advantages for using simulators is that a wide range of visual phenomena can be modeled in a highly controlled fashion (e.g. the amount of people in a certain scenario). Moreover, the effectiveness of a system can be determined before is actually deployed. There is no installation and hardware cost, in many times only one computer may be needed to install and run the simulator. Another key benefit is the dynamic configuration of the system according to the technical details of a particular algorithm (e.g. move the camera to left) or properties required at each moment (e.g. zoom in the face of a person). Finally, simulators can be a effective tool in education where students learn concepts and behaviour of the simulated system's components such as for integrating video-surveillance technologies.

It is important that simulation tools incorporate mechanisms to control the configuration of cameras individually (e.g. 3D position and rotation), to capture information in real time (e.g. frames and ground truth), to synchronize all sub-systems in the simulator and to manage the data flow.

1.2 Objectives

The main objective of this Bachelor Thesis is to develop a simulator which provides virtual video from multiple cameras. This virtual system must be able to manage several distributed cameras in real time, configure camera's properties and possibility to transmit data (e.g. frames) to external applications. The Unity game engine[3] is used as the starting point to simulate realistic 3D environments which is extended with the desired multi-camera functionality.

[1] https://en.wikipedia.org/wiki/Orthographic_projection
[2] https://unity3d.com/es
[3] A game engine is a software framework designed for the creation and development of video games

In order to achieve the main objective, the following sub-goals are defined:

- To study of the related state of the art including related multi-camera simulators and available game engines to motivate the selection of Unity.
- To design and implement a system for controlling cameras remotely with different location configurations (e.g. position, orientation) and acquisition parameters (e.g. frame rate, resolution) to get information from the 3D environment.
- To design and implement a system for supporting the simultaneous connection of several third party applications (e.g. computer vision algorithms) to the multi-camera simulator. This system uses a client-server architecture and considers a server (integrated within the simulator), a communication protocol and an API that will be integrated and used by future applications (i.e. clients).
- To design and implement a proof-of-concept example containing a 3D scenario modeling the hall of the 'Escuela Politécnica Superior' ('A' building) and two simple application making use of the simulator using standard computer vision libraries such as OpenCV[4].
- To evaluate the functionality and performance of the system developed.

1.3 Document Structure

This documents is structured as follow:

- Chapter 1. This chapter introduces the motivation and the objectives of this Bachelor Thesis.
- Chapter 2. This chapters gives an overview of related works and game engines.
- Chapter 3. This chapter describes the design of the system developed and models of the virtual scenario.
- Chapter 4. This chapter describes the system architecture and implementation.
- Chapter 5. This chapter presents and discuss the experiment results.
- Chapter 6. This chapter summarizes the main achievements of this work and gives suggestions for future work.
- Bibliographic.

At the end, a appendix go into further details:

- Appendix A. This appendix provides a simple tutorial for including this simulator in an existing or in a new Unity project.
- Appendix B. This appendix provides further information of game engines.

[4] http://opencv.org/

Chapter 2

State of the art

This chapter overviews related works for multi-camera simulation. In section 2.1, we briefly explain the key elements of a game engine and we describe the most popular ones. Then, we present existing visual camera simulators related to this Thesis in section 2.2.

2.1 Game Engines

2.1.1 Comparison

In the last years, the computer and video games industry has grown so far that the most sophisticated rendering pipelines, interactive or physics simulations are no longer exclusively running on an expensive specialized scientific machines. They are now operating on personal computers or game devices at significantly lower cost. This is possible due to the widespread of software frameworks where games are designed and created: game engines. This frameworks can be employed in scientific research, recently increasing the interest in realistic simulation.

There are various important engines available in the market with free license: Unreal Engine or UE4 (https://www.unrealengine.com/), Amazon Lumberyard (https://aws.amazon.com/lumberyard/) and Unity (https://unity3d.com/). These games engines share common features but they have key distinctive aspects. Further descriptions are provided in the Appendix B including examples of use in research. Table 2.1 shows a comparative with some distinctive aspects. Although UE4 and Unity may be similar, we decided to use Unity for one reason: as UE4 full source code is provided, it has not an extensively developed API (i.e. you may need to create you own methods for your requirements).

Table 2.1: Table comparative of game engines.

Game Engine	Free License	Source Code availability	Marketplace	Extra Services	Script Language
UE4 4	Yes	Yes	Yes	Only Third-party	C++
Amazon Lumberyard	Yes	Yes	No	Amazon Web Services, Twitch integration	C++
Unity	Yes	No	Yes	Unity Ads, Multiplayer, Performance Reporting,...	C# and JavaScript

2.2 Camera Simulators

2.2.1 Introduction

In Computer Vision research, simulators are important as they can be a reference to design algorithms. We make a distinction in two different types: smart-camera networks simulators (SCN) and visual simulators (VS).

A smart-camera is a sophisticated vision system, complete with image sensors, communication interfaces and on-board processing. With a smart-camera network (SCN) we are capable to coverage extensive areas for search and rescue missions, smart cities, security and surveillance. SCN are a increase area research, and complexity in the develop of these system motivate the development of simulators tools like the next examples.

CamSim [6] is a SCN simulator which enables to test self-adaptation and self-organisation algorithms without the difficulties of deploying a physical smart-camera network. As key benefits of CamSim, it is easy to generate for different testing scenarios, with no virtual cameras and objects limits. Moreover, camera behaviour model and several communication techniques can be configured dynamically at run-time.

WiSE-Mnet++ [7] is another SCN simulator to simulate a range of different scenarios, including free place cameras, collaborative sensing and models for data exchange. This holistic SCN simulator models sensing, processing, communication and self-decision operations. Moreover, it simulates realistic smart-camera networks with real world and synthetic datasets.

In the Computer Vision community, virtual video from simulated scenarios used to be not considered because it usually removes artifacts that capture digital cameras and it does not offer enough realistic graphic simulation. But with modern game engines we are able to simulate photo-realistic scenarios, and this is the reason why visual simulator become popular in the last years. A visual simulator used in the appropriate way for designing, debugging and testing, it saves significant time and cost. As this Bachelor Thesis presents a multi-camera visual simulator, this chapter is focussed on these systems. In the next sections, we review existing visual camera simulators and discuss their features and inconveniences.

2.2.2 OVVV

The ObjectVideo Virtual Video (OVVV) [8] is a simulator tool to generate video from virtual cameras in a virtual 3D world. It has two components: Virtual Video Mod (a Half-Life 2 game mod, based on Source Engine, both developed by Valve Software) and C API (C library with functions to stream virtual video and control cameras by applications).

ObjectVideo Virtual Video is free distributed for research and development purposes. However, since one of its component is based on a commercial game, a license is necessary. After purchasing Half Life 2, you can obtain the Source SDK engine (including map editor and source code) for free. Figure 2.1 shows an example of the virtual 3D world and the graphic quality. Its main features are: (1) Freely place, configure underlying parameters independently for each camera; (2) Synchronized video streams in real time from independent cameras observing the same virtual world; (3) Automatic target ground truth generation and repeatable testing.

Figure 2.1: OVVV example: an indoor office scenario with several employers. Source: [8]

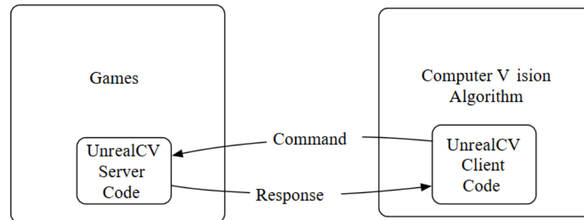


Figure 2.2: UnrealCV architecture. Source: http://unrealcv.org/

2.2.2.1 OVVV extension

The main problem of OVVV is the absence of handling cameras in a distributed computing environment. In order to build a complete simulator, in 2015, Luis Pérez presented an extension of the OVVV simulator for his Bachelor Thesis[9]. This extension incorporate to OVVV a Client-Server architecture allowing communication from the simulator to external applications. Further, a cameras controller was developed which has methods to create, to delete and to configure multiple cameras in real-time and simultaneously. Additionally, a feature for visualizing cameras is provided using the OpenCV library.

2.2.3 UnrealCV

UnrealCV [10] is an open source project created by Weichao Qiu that helps computer vision researchers to build virtual worlds using UE4 4 (UE4). It is distributed like a plug-in which extend UE4's functionality with two features: (1) A set of commands to interact with the virtual world and (2) Communication between UE4 and an external program using a TCP Server. They also provided a compiled binary as a simple application with UnrealCV embedded for testing purpose with any previous knowledge required.

The main work of UnrealCV is to do IPC (Inter Process Communication) between a simulation and a computer vision algorithms. The data flow is represented in 2.2. A simulation created on UE4 can be extended by adding UnrealCV Server module. It automatically starts a TCP server and waits for commands from external applications which must use the UnrealCV client code to generate these commands. Since it has a Client-Server architecture, it is prepared to cross-platform and support multiple programming languages.

UnrealCV does not support multiple cameras and it has another important disadvantage. We can not generate and transmit images in real-time because data is saved on disk first, not in main memory, implying high latencies not allowing real-time processing.

2.2.4 SLCNR

Software Laboratory for Camera Networks Research (SLCNR) [11] is a virtual vision simulator capable of generating synthetic video from multiple cameras in customize 3D virtual worlds with a tool included for simulating pedestrian traffic which also supports pedestrian detection and tracking. From this tool must be mentioned that can be deployed over a networks of computers to simulate large complex scenes. Figure 2.3 shows an example of the virtual world that can be simulated in this simulator in an indoor scenario.

This virtual vision simulator has three modules. First, a Virtual world engine (VW) based on Panda3D is capable of simulating 3D scenes including dynamic objects as pedestrian, automobiles, etc. It supports passive, wide Field-of-View (FOV) and active PTZ cameras. Second, a Video analyses pipeline (VP) processes images captured by cameras located on VW. Each camera has its own VP module making up a virtual smart camera. Third, a synchronization unit (SYNC) that can be executed over a network of computers running instances of VW and VP modules. The SYNC unit syncs all modules with each other.

Figure 2.3: Examples of the SLCNR virtual world with several pedestrian walking. Source: [11]

2.2.5 Virtual Pedestrian

In recent years, pedestrian detection has become an important area research, according to needs on intelligent video surveillance systems or the potential for improving safety systems. In ref [12], authors perform an extensive evaluation of the state of the art.

Usually, pedestrian detector are trained over large datasets from the real world in different scenarios. However, this datasets are not always suitable. For example, in a rescue scenario it is important to have the appropriate pedestrian detector according to the environment. To generate a real world dataset in this scenario, takes times, because it is necessary to record video but also it is implicated actors and other parameters that must be manually replicated. By considering the problem of training a pedestrian detector algorithm in a scenario where is not easy to generate a dataset, in [13], authors propose a method which is able to generate virtual pedestrian models into a scene according the perspective geometry . This project provides a system for training pedestrian detectors in specific scenes. A difference from the previous simulators, this tool uses real data as input to generate synthetic data.

2.2.6 Conclusion

The previous simulators reviewed are an interesting tools for Computer Vision research but they have two main inconveniences. First, UnrealCV and Virtual Pedestrian are not multi-camera supported. This is a big inconvenience because, by using a simulation tools, we have the

possibility to establish direct communication with the cameras, allowing dynamic configuration and as result, we can repeat one scenario in different situations or parameters that can not be easy replicate in the real world. On the other hand, UnrealCV, SLCNR and Virtual Pedestrian are not prepare to work in real-time. The goal of these simulators is to generate virtual video datasets that can be used for training and testing vision algorithms. This limitation causes that they can not be coupled to a SCN compatible simulator like WiSE-Mnet++. By combining the appropriate visual simulator and the appropriate SCN simulator, we may have a holistic simulator for smart-camera networks.

This Bachelor Thesis presents a simulator called Multi-Camera System Simulator (MSS). In Table 2.2, we compare the main features of the previous simulators and our work. The main advantage of our simulator is to based it in a modern game engine with high graphics capabilities, a multi-camera system and real-time broadcast from individual cameras.

Table 2.2: Table comparative between visual camera simulator. MSS is the simulator that this work presents.

Game Engine	Free	Source Code availability	Multiple cameras	Cameras remote controller	Custom scenarios	Game Engine based	Real-Time Broadcast
OVVV [8]	Yes	No	Yes	No	Yes	Source	Yes
[9]	Yes	Yes	Yes	Yes	Yes	Source	Yes
UnrealCV [10]	Yes	Yes	No	Yes	Yes	UE4	No
SLCNR [11]	No	No	Yes	Yes	Yes	Panda3D	No
Virtual Pedestrian [13]	No	No	No	No	Yes	Custom	No
MSS (proposed)	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**	**Unity**	**Yes**

Chapter 3

System design

This chapter outlines the requirements, design of the system and some concepts of the simulator. These requirements determines the technologies appropriates to develop our system and the modular design of our simulator.

3.1 Requirements analysis

Along this section, the goals of the system presented in this document are described. To facilitate the compression of the requirements, we make a difference into these categories: functional requirements and non-functional requirements.

3.1.1 Functional requirements

The requirements described below are the functionalities that must be developed to achieve the goals of this project. In order to validate all requirements, in chapter 5 we design a functional testing based in a list of different possible actions available in the simulator such as to create a camera.

FR1: 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.

FR3: The simulator will based on a modern engine where we can recreate photo-realistic scenarios.

FR4: Dynamic objects will be supported such as pedestrians, automobiles or drones.

FR5: Customizable artificial intelligence will be supported for autonomous characters.

FR6: Scheduled events and characters actions can be programmed.

FR7: Cameras located freely in the scenario including position and height.

FR8: Camera rotation on itself in two axes: up-down and left-right.

FR9: To generate frames (images) from the cameras.

- FR9.1: Several image formats will be available.

FR9.2: Frames resolution will be able to adjust.

FR10: Images per second (framerate) generated will be able to adjust between 1 and 30 for each camera.

FR11: The area of the observable world (field of view) that is seen in a given moment will be able to adjust.

FR12: Each camera will have a buffer where frames temporary will be saved on main memory.

FR13: All the parameters mentioned above will be able to configure individually and dynamically.

FR14: To Create and to delete cameras.

FR15: A broadcast for each camera.

FR16: All cameras must be synced. This means that different cameras see the same world in a given moment.

3.1.2 Non-functional requirements

In order to build a reliable simulator for research, these requirements must be considered:

NFR1 **Performance**: Performance in our system is critical because must be designed to work in real time: simulation, frames generated and response to clients. It must be developed with algorithmic efficiency, exploit the computer resources and making parallel computing in all processes that allow it.

NFR2 **Availability**: Our system must be executed during large time without reliability problems.

NFR3 **Reliability**: The simulator may operate over long periods of time without failures.

NFR4 **Maintainability**: Well documented code and distributed on modules for future extensions.

NFR5 **Documentation**: An extend documentation must be included so no previous knowledge of simulators or game engines are necessary.

NFR6 **Usability**: An API client code must be provided to ease the use of the simulator.

3.2 System Overview

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 3.1, 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 are able to communicate and to receive information through the methods included.

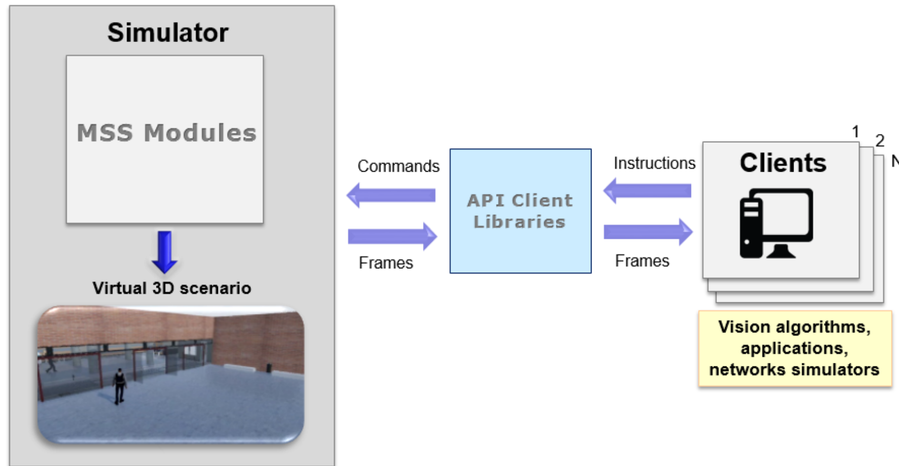


Figure 3.1: MSS incorporates a Client-Server architecture allowing multiple connections.

The MSS modules extend the native Unity's features with a multi-camera system, an asynchronous server TCP and more. We explain details in section 3.5. These modules work as a plug-in, making possible to incorporate the simulator in an existing Unity project or in a new one. To incorporate it in an existing project, you only need to copy the MSS's source folder which contains all the MSS's classes and code in to the root Unity project folder. This process is visually explained as a tutorial in Appendix A.

On the other hand, we also provide a compiled version which is used for experiments in chapter 5. This version is a stand-alone application for windows that can be used without previous knowledge and includes all functionality of the simulator and one scenario. By default, Unity only provides an empty scenario. As it is not useful to test the simulator over a empty scenario, we design an example scenario which is detailed in section 3.6.2. To run it is quite simple: launch the .exe, configure the quality graphics options and start it. After the application loads, a black screen appears with a button on the top left (Figure 3.2). When you press the button, the simulator is loaded and initialized and the simulator is waiting for clients on background.

3.3 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 of all, 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:

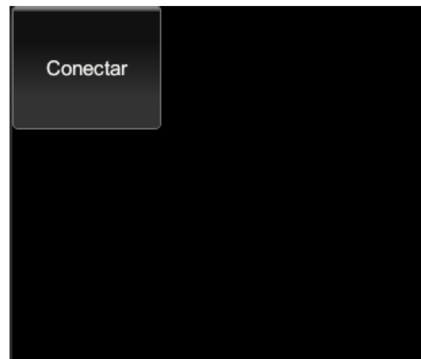


Figure 3.2: The MSS's default screen when is executed. If the button is pressed, the simulator is initialized and ready to use.

1. 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.

2. We read the first frame of the video, and we give it to the pedestrian detection algorithm.

3. 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.

4. 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.

5. We read the second frame of the video and repeat the process. We repeat these steps until all frames of the video are processed.

6. Finally, with the results of all frames of the video, we can conclude the effectiveness of the algorithm.

The idea of our simulator is that an application like this example uses the frames generated by a virtual camera instead of a pre recorded video. These frames are generated in real time, exactly with the same behaviour that if we connect to a real multi-camera system (e.g. surveillance). To adapt this simple application to the use of the simulator, we modify some steps. First, we still need a dataset with the difference that this dataset is generated in real-time. So, we load some scenario in the simulator and create a camera placed where it has been decided. After, we request a frame to the simulator. When the frame is received, we apply a conversion and give it to the algorithm. All the following steps are equivalent to the process explained before.

3.4 Logic View

The development of this simulator is complex due to the implication of a variety of technologies that work for specific purposes: image processing, multithreading, sockets, synchronization logic, GPU programming and more. To an optimal implementation, we make a modular design (Figure 3.3). The simulator is composed by three modules:

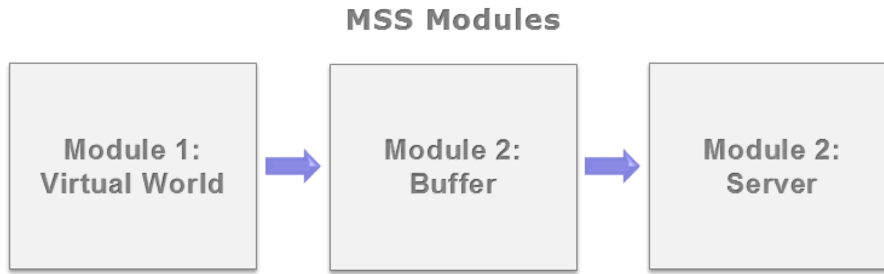


Figure 3.3: MSS Modules logic view.

- **Virtual Word**: this module is responsible for all operations related to the virtual world and therefore a direct integration with the game engine (Unity). It contains the virtual cameras, the cameras controller and all logic which implies using the Unity Scripting API. By the Unity technical limitations, this entire module works sequentially, synced and following an object oriented programming. This module involves mainly GPU programming and synchronization logic.

- **Buffer**: the main purpose of this module is to store frames in main memory temporarily. In order to get the maximum performance possible, this module is implemented with multithreading. As it is explained in the next chapter, before storing a frame in main memory is necessary an image conversion. So, this module involves multithreading and image processing.

- **Server**: this module contains an asynchronous server that allows multiple client connections. It receives commands from clients, processes them and gives a response. In order to attend the clients immediately, this module also is implemented with multithreading. In this way, with independent threads, the performance of the previous modules does not affect to server and the connection logic. In this modules is involved sockets, multithreading and a *ThreadPool* which is explained in the next chapter.

3.5 Unity extension

This simulator is based in the Unity game engine. Unity has the capacity to simulate realistic scenarios, with customizable artificial intelligence (AI) and support for dynamic objects such as pedestrian and automobiles. Further, by using the Unity scripting API we are able to interact with the virtual world through the methods and classes included. In particular, Unity has an special object which is employed for rendering the scene. This special object are the camera, that we extend for creating our custom camera and we use it at starting point to develop our system.

The Unity scripting API does not contains classes for sockets, multithreading and image processing. These extra functionalities that are requirements for our simulator, is what we extend with the MSS modules. In Table 3.1 we find an elemental comparative between the native features of Unity and the features that our simulator includes. It is important to mention that Unity is not multithread safe. This means that we can not use Unity API methods across different threads, and this is the reason because we have a module exclusive for the operations

Table 3.1: Table comparative.

Features	Unity	Extension with MSS's modules
Realistic 3D Scenarios Support	Yes	Yes
Distributed Multi-Cameras	Yes	Yes
Position, Rotation and FOV	Yes	Yes
Framerate and Resolution	Yes	No
Capture Frames Realtime	No	Yes
Multithreading	No	Yes
Server TCP	No	Yes
Buffer for frames	No	Yes
Cameras Controller	No	Yes
Cameras Synchronization	No	Yes

relational to the virtual world. Moreover, as the language programming of Unity is C#, the Unity compiler allows to use .Net framework so we can create threads like any .Net application without the previously mentioned constraint. Therefore, we taken advantage of such .Net feature for implementing multithreading in the logic of the buffer and the server modules.

3.6 3D Models

3.6.1 Camera

For the implementation of our Cameras, we design a virtual pinhole camera model. A pinhole 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 in to 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 in to 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. The FOV is the most important property of a camera. It is the extension area of the observable world that is seen in a specific moment. A visual representation of a camera and the FOV property is represented in Figure 3.4. There are three important components. First, the near plane is the plain that cut the observable area perpendicular to the viewing direction. The objects closer to the camera than the near plane are not visible. The far plane is just the opposite function to the near plane. Sometimes, the far plane is placed in an infinitely far away space from the camera so all objects can be seen regardless of their distance from the camera. Finally, the horizontal and vertical FOV properties are the width of the camera's view angle, usually measured in degrees.

Others camera properties that are modelled according to the Unity API parameters available [1] are:

- **Perspective projection**. In perspective projection, the distance between the aperture (center of projection) and the far plane is finite. The size of the objects varies according to the distance which gives a realistic aspect.

[1] https://docs.unity3d.com/Manual/class-Camera.html

- **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.

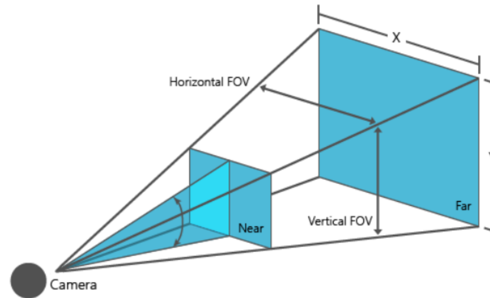


Figure 3.4: A visual representation of a camera and the FOV property. The tridimensional space between Near and Far, and the angle which defines the horizontal and vertical FOV is the observable world. Source: https://msdn.microsoft.com/en-us/library/dn479430(v=vs.85).aspx

3.6.2 Scenario

As it has explained in section 3.5, Unity only provides a predetermine empty scenario. By default, there is an 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 Politecnica 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.

3.6.2.1 Dynamic Objects

A dynamic objects is an object that has an action or an animation implemented trough a script, also called shader, which contains mathematical calculations and algorithms that modified the material and the texture of the object (e.g. liquids, reflections, explosions). In particular, in this scenario we have autonomous pedestrian walking around with a basic artificial intelligence (AI) script.

For the implementation of this characters, it is necessary three components:

- 3D Human model. We use free humans models from [2][3]. Not all 3D models are compatible with animations. For animated it is necessary a model rigged or a model with animations incorporated. A model rig is essentially a digital skeleton bound to the object. Like a real skeleton, a rig is made up of joints and bones, each of which act as a point of reference that the animation used can used.
- Animations clips. An animation clip is an isolated piece of motion, the basic action for movement such as run, walk or jump. They can be modified and combined in several ways to produce lively final characters.
- AI scripts. We use a basic AI script to keep the character walking around constantly or following a predefined route along the scenario.

3.6.2.2 Static Objects

The static objects are those elements which compose the building. This means floor, ceiling and wall but also furniture objects such as chairs or windows. The building structure is built by planes for floor and ceiling, cubes for walls and cylinders for columns. In total, more than 100 different objects have been manually generated, placed it and textured according to the structure and proportions of the real building. It is used free resources from [4][5][6] for getting 3D furniture models and building the interior space.

[2] https://www.assetstore.unity3d.com/en/#!/content/31711
[3] https://www.cgtrader.com/3d-models/character
[4] http://tf3dm.com/
[5] http://www.turbosquid.com/Search/3D-Models/free
[6] https://www.cgtrader.com/free-3d-models

Chapter 4

Implementation

Based on the requirements described and the logic view explained in the last chapter, in this chapter we introduce the hardware and the software architecture of our system. Later, we explain the implementation and the interconnection of the three modules that composed the simulator. After that, we detail the API client library and give two examples client applications by using the API developed.

4.1 Hardware architecture

In this section we detail the hardware architecture of the simulator. This system involves the communication between hardware unit due to the diverse technologies involves such as GPU programming, image processing and sockets. Figure 4.1 shows the hardware architecture and the relation between physic computer components.

- **GPU**. The GPU unit is the most important unit in our system because it has two important functions. First, GPU is where the virtual scenario is loaded including the cameras. If we simulate complex scenarios, a powerful graphic card is a requirement. On the hand, GPU is the unit that generates the frames of the cameras. This is vital because depending of the graphic cards it can generate more or less frames in one second (maximum framerate).

- **RAM**. After a frame is generated by the GPU unit, it is sent to main memory to store it temporarily. The reason of storing in main memory is that this simulator has the goal to provide frames in real-time. If we use disk instead main memory, the implicit latency will not allow real-time processing.

- **CPU**. The frames that GPU generates has an specific image type which is not adequate to work with it. Because of this, all frames need to be processed to convert this image type in to an standard one like JPEG.

- **Network interface card**.The NIC is the hardware unit that allows to communicate the simulator with external applications. We must take into account that is needed a sufficient bandwidth to broadcast cameras. In chapter 5 we analyse how much is necessary with different conditions.

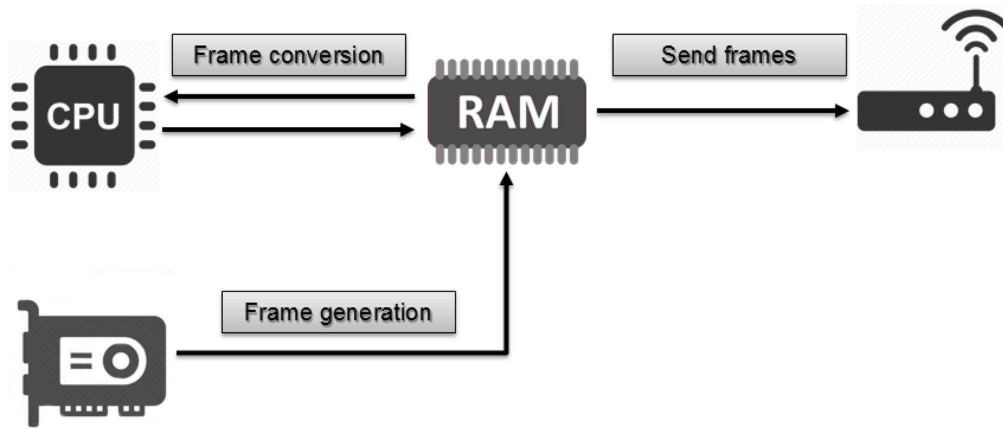


Figure 4.1: Hardware architecture and relations between physic components.

4.2 Software architecture

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. 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 a optimums image processing and avoid bottle-necks. Finally, the Server module has an asynchronous server prepared for multiple connections. Commands Manager is responsible for handling the data input and output of the simulator. Figure 4.2 shows the architecture and the data flow.

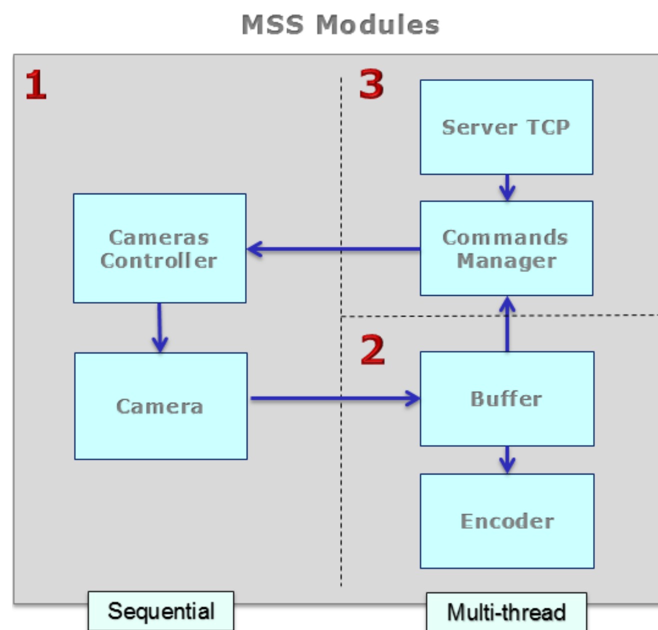


Figure 4.2: MSS Modules architecture and data flow diagram.

4.3 Module 1: Virtual World

4.3.1 Unity scripting API

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 [1] with 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.

A *GameObject* is the fundamental object in Unity. They have not functionality by themselves but they act as containers for components. These components can be 3D models, script, textures, sounds, characters and another GameObject. Every script attached to a GameObject is active and automatically executed with predetermined event functions following a sequential logic [2]. The base function is Update, which is automatically called once per cycle. It usually contains the application logic. The times that this cycle is executed per second is known as framerate (fps). This is vital because the virtual world is updated once per cycle too. We use this logic to sync the classes and objects of this module.

4.3.2 Camera object

For the implementation of our Cameras, we create a new GameObject with a special component. This component is the virtual pinhole camera model that generate frames of the scenario trough its field of view. It has a complete list of configurable variables where we find properties such as framerate, FOV and world position. It could be many cameras in the scenario, but only one acts as the main camera which is used for display the scene on screen. As the simulator sends frames to external applications, whatever shown on the screen is not relevant: the scenario is loaded on memory whether or not we have any camera active. Figure 4.3 shows the 'MainCamera' GameObject with a virtual camera component.

At this point, we want to save this object in our project to later to create instances and this is possible by creating a custom prefab. A prefab is an object which allows to store a GameObject with all its components, scripts and properties in the project as a resource. The prefab acts as a template from which you can create instances in the scene. Any edit made to a prefab will be reflected in all of its instances but you can also override components and settings for each instance individually. We call this prefab 'CameraSimulator' and it is the final object of our project.

4.3.3 Camera class

The 'CameraSimulator' prefab also contains a script with the implementation of our custom camera class. Apart from methods to customize it dynamically, the camera class implements the

[1] https://docs.unity3d.com/ScriptReference/Camera.html
[2] https://docs.unity3d.com/Manual/ExecutionOrder.html

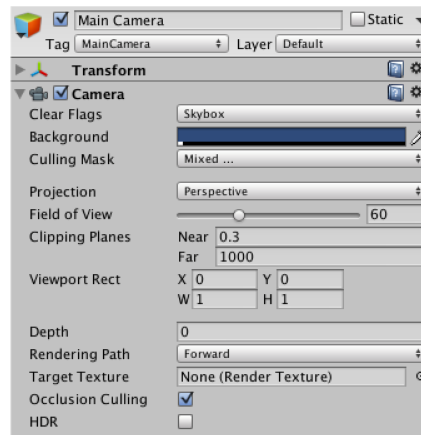


Figure 4.3: This GameObject called 'Main Camera' has two components attached. Every GameObject has a transform component which store the position, rotation and scale of the object. Finally, the camera component is a virtual camera model.

Update function which is automatically called once per frame and contains the image generator code. As the main objective of the simulator is to provide frames to clients, this function is essential. Every time it is called, a flag, used for the synchronization logic (section 4.3.4), is active or not depending on the framerate property. If flag is active, it is generated a frame following this process:

1. First, we render the camera. By rendering, it refreshes whatever the camera is seeing on that moment. In this step, only GPU is involved.

2. Later, data is saved on GPU memory using a Texture2D variable, a Unity class for texture handling.

3. In order to transfer the image data from GPU memory to main memory, we use the 'GetRawTextureData' function provided by the Unity API[3] which returns the image data saved it on GPU into a byte array. For this transfer, it takes many milliseconds depending on the image resolution.

At this point, the frame is stored in main memory in RAW format. Considering that the frame will be sent over the network, this image type is not appropriate due to it takes up much more space than compress image formats such as JPG. Additionally, RAW is a not a standardized format type across different manufacturers. So, we need to convert the RAW image into a standard type. However, image conversion takes a huge computational cost that may caused a big bottleneck in our simulator. Regarding to this problem, we parallel this conversion trough the buffer module. In this way, we can obtain the next image while the actual is being processed.

4.3.4 Cameras synchronization logic

As we say before, the scripts that extends methods and classed from the Unity scripting API follows a sequential logic trough an execution order of event functions. We use this functions to sync the frames generation between cameras. The idea is that each camera has a script

[3] https://docs.unity3d.com/ScriptReference/Texture2D.GetRawTextureData.html

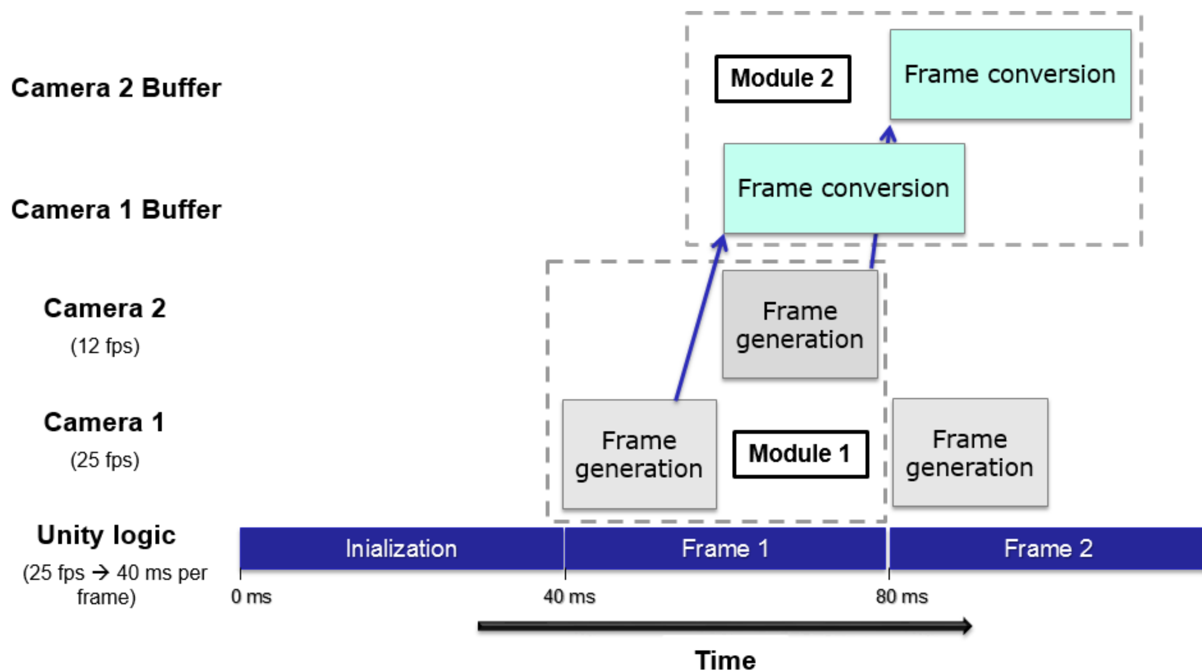


Figure 4.4: Camera 1 and Unity has the same fps, so one image is generated per frame. Camera 2 has 12 fps, so it is generated once per twice frames.

(camera class) with the code developed in to the Update function. By doing this, we obtain one frame per cycle for each camera and we obtain cameras working at the same time as the Unity logic. Therefore, all frames generated by different cameras in the same cycle display the same virtual world moment but with their own perspective. But there is another important detail: the camera's framerate is an individual property. For example, suppose a case in what we have two cameras: one running at 20 fps and one running at 10 fps. We can not generate 20 frames + 20 frames because this will imply a waste of resources that cannot be afforded. To solve this problem, we design this process as follow (Fig 4.4) :

1. If there are no cameras created, we force the Unity logic cycle to work at 30 fps.

2. If there are one camera, we force the Unity logic cycle to work at the fps property of the camera.

3. If there are several cameras, we look for the maximum fps that the cameras have and force Unity to work at that fps. The rest of the cameras use that fps and their own fps property.

The process implemented presents a big problem. The computational cost of generating a single image is high and we want to generate until 30 images per second. This number is multiplied by the number of cameras we have in the scene. So, what happens if we can't generate so many images in one second? Unity's fps will start to decrease and consequently, the framerate of all cameras. Note that situation will not only affect the cameras, this decrease in speed will be noticeable in all scripts. The positive thing with this process is that all cameras will decrease their framerate in the same proportion so we still have a synced system.

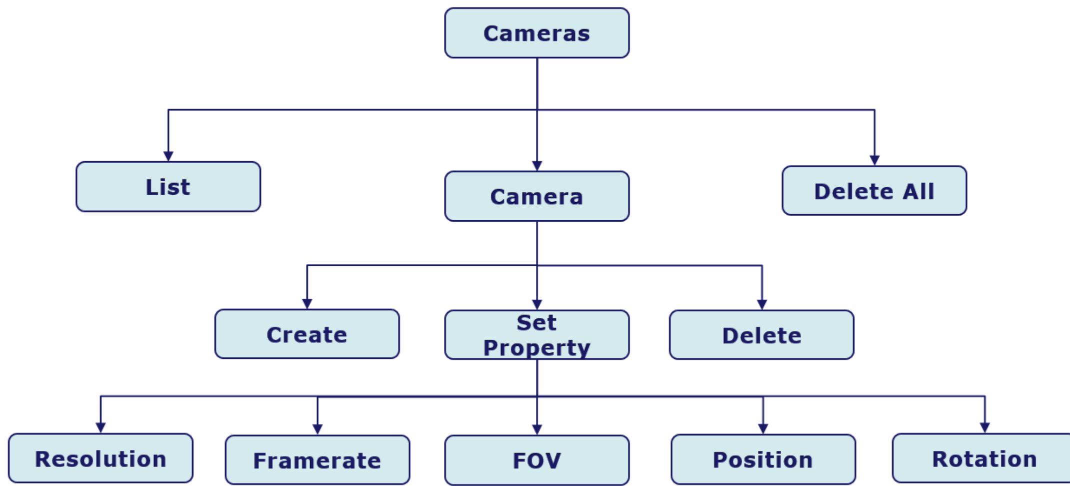


Figure 4.5: Cameras Controller functionality diagram.

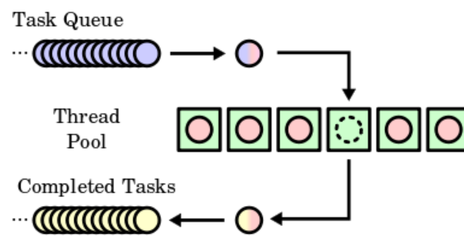


Figure 4.6: Commands are represented by the waiting tasks(blue). The Cameras Controller class reads the ThreadPool once per cycle and processes them. Source: https://en.wikipedia.org/wiki/Thread_pool .

4.3.5 Cameras Controller Class

In our design, clients have not direct connection with the cameras because we have a central server which manages all connections.. Instead, we have a Cameras Controller class that handles the cameras by the instructions sended from clients. So, this class receives commands from clients and interacts directly with the cameras. The commands implemented is represented in Fig 4.5. By using the Update function, this class reads once per cycle the ThreadPool where the commands are stored momentarily and processes them. We develop a ThreadPool because a task queue is the best option to communicate this class to the Server module which works asynchronous. Fig 4.6 explains the ThreadPool the data flow. When the server receives a command relational with the cameras (e.g. create, delete) from a client, it is saved in the Task Queue. Then, this class reads all commands received and processed them.

4.4 Module 2: Buffer

4.4.1 Buffer class

When a camera is created, it initialize an independent thread with the Buffer class using .Net framework. Each camera has its own buffer where the frames are stored on main memory. Figure 4.7 shows the logic view of this module. The buffer has direct connection with its

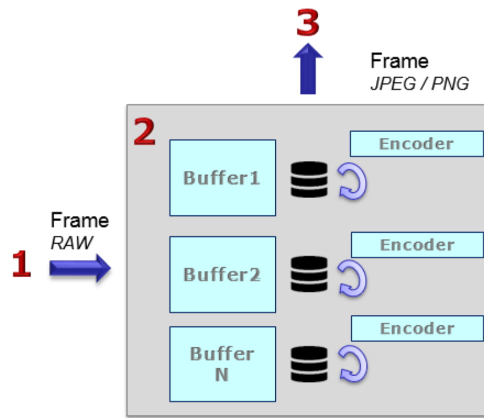


Figure 4.7: Buffer module logic view.

camera associate. With the Encoder class we convert the frames received in to a standard image type.

By default, this thread sleeps until receives a frame directly from the camera which is assigned. When it receives a new frame, the thread wakes up and calls to the Encoder class for doing a frame conversion. Depending on the frame resolution and the image format configured, this conversion takes from many milliseconds to hundreds. As this process is executed in an independent thread, the execution of the other modules of the simulator is not affect by this execution thread. The result of this conversion is the frame (image) that the clients receives. After the conversion process, it checks if any new frame has been received while the previous one was being processed. In this case, this process is repeated, if not, the thread sleeps again.

The MSS simulator is prepared to work in real time so we do not need to store many frames on memory. In fact, we only need to store the last one. Because of this, the buffer size allows only to save the last frame processed. When a new one is processed, the buffer is overridden.

4.4.2 Encoder class

The Encoder class receives a frame from the Buffer class and returns it converted into one of the following formats implemented:

- **JPG**: the most common image file format. This format is the perfect option if you are looking the best relation quality/size or time conversion.

- **PNG**: file format with lossless data compression. This must be your option if you are looking no detriment to the quality. Unfortunately, this format takes a much longer time conversion so it can be a bottleneck.

Both formats use the image encoders integrated on GDI+ API, a graphic device interface for Windows. A graphic device interface allows to applications to display information on screen with the abstraction of the display device installed. In order to use the GDI+'s encoders, it is necessary to include the 'drawing.dll' library in our project.

As it is explained in section 4.3.2, the frames generated by the cameras have a RAW format. This not standard format uses a property implementation from Unity company which is not compatible with GDI+ library. Regarding to this problem, it is used the image processing

library EmguCV, a cross platform wrapper to the OpenCV library compatible with Unity's compiler. This library contains the appropriate method to read a RAW image data and to build a GDI+'s compatible class.

4.5 Module 3: Server

This module contains the asynchronous TCP server class and the Commands Manager class. The async server allows to support a large number of clients which avoid blocking methods except the exactly moment when a client connects but this time is insignificant. The server operates as a multithread socket server. Further, it helps to improve the performance and responsiveness of the simulator. In fact, responsiveness is a requirements for this project because clients must received a response immediately in order to reduce the existing delay between to send a command and to receive a response. On the other hand, the Commands Manager class manages the data received by the server and handles the list of the clients connected to the simulator. Fig 4.8 shows the logic view of this module. When the Commands Manager class receives an instruction relational with the cameras, it sends it to the ThreadPool as it is explained in section 4.3.5. If it receives a data request such as get a frame, it gets the last frame generated stored in the appropriate buffer and sends it to the client.

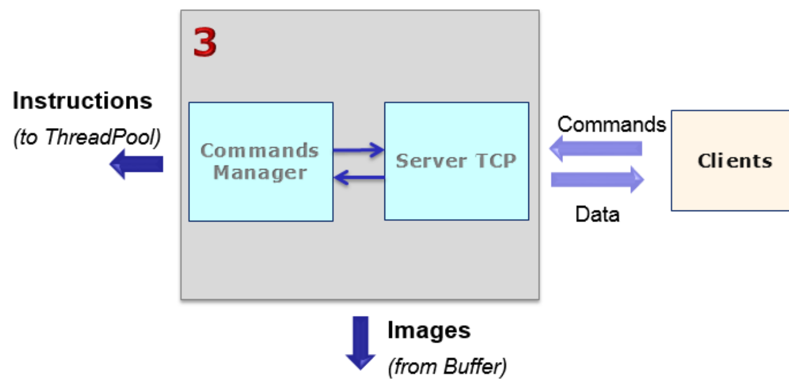


Figure 4.8: Module logic view. Instructions represents the commands, and images the frames.

4.5.1 Server TCP

The Server TCP is implemented with an independent thread based on Microsoft .Net[4]. It is developed with an asynchronous socket, the best option for multiple client connection. In an asynchronous socket, the thread is not blocked while it is waiting for incoming data. Instead, each client has an instance of the *NetworkStream* class which provides methods for sending and receiving data in background using a buffer. The main execution of the thread always waits for new client connections and creates a new NetworkStream's instance for each one. The logic of this class is represented in Figure 4.9.

4.5.2 Commands Manager Class

To give an immediate response to the client commands and the best performance possible, the *Commads Manager* class is developed with another independent thread. This class is always

[4] https://msdn.microsoft.com/es-es/library/fx6588te(v=vs.110).aspx

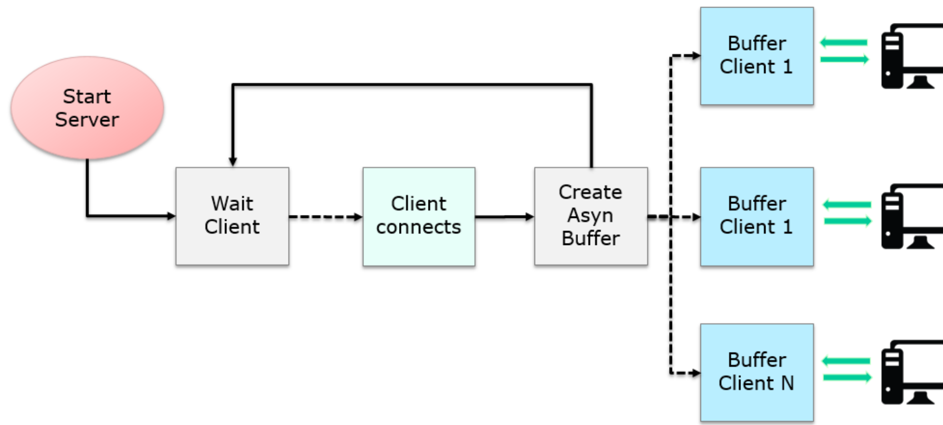


Figure 4.9: Server TCP diagram.

checking if new incoming data has been received reading the buffer that each client has. There are two kinds of commands:

- **Order**: related to camera's functionality. For example, create a new camera or change some property. In this case, the commands is sent to the Cameras Controller class thought the ThreadPool explained in section 4.3.5.

- **Petition**: clients ask for data or information. The common case is to request a frame from one specific camera. This class has not directly connection with the Camera class but it does with the Buffer class. So, it looks for the appropriate buffer using the camera's identifier (name) and sends to the client the last frame stored.

4.6 API Client Libraries

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. It includes libraries for:

- **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. It is also possible to initialize cameras reading a file thanks to 'createCameraFromFile' method. This method allows to create cameras easily and automatically through a script.

- **Connection**: this library contains all the methods relational with the communication between clients and the simulator using sockets. The idea is that an application can configured the camera object in its code and then apply this changes in the simulator. With this library, clients can request information like request a frame from an specific camera,

- **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. OpenCV support image treatment (e.g. contrast, brightness, threshold) and object detection (e.g. recognition, tracking). With this library, the frames that clients received can be convert in to a OpenCV compatible class that allows image processing.

- **User Interface**: when a frame is received and converted with the Image Processing library, clients can used this library to display it on screen. Further, it includes controls that allows an user to handle a camera in real-time. This controls are connect with the Camera and the Connection library and both implement the logic to modify the camera in the client and simulator. An example is showed in Figure 4.11b. With these buttons we can move, rotate and change camera's properties visually and without a line of code. This library is developed by using the 'Qt OpenCV GUI' module integrated in Opencv API.

4.7 Communication protocol

Now that we explain the implementation of the simulator and its different components, we give an communication protocol example (Figure 4.10). On this example, a client is already connect and suppose a camera running. The action represented in this Figure is to request a frame. On one hand, the camera running generates frames periodically at the framerate configured. The camera generates frames all the time even a client does not request them because if we generates a frame on the moment that the client requested it, we would have an enormous delay (frame generation + frame conversion + data transfer). With our method, we reduce this delay to the minimum possible (data transfer).

On the other hand, the client request one frame from the camera -always the last frame generated like we are working with a real network camera by sending a command to the simulator. Server receives this command and stores it in the buffer that each client has, as we detail in section 4.5.1. After that, the Commands Manager class reads this instruction, gets the last frame stored in the appropriate buffer and send it to the client using the server. This process support multiple client's request simultaneously.

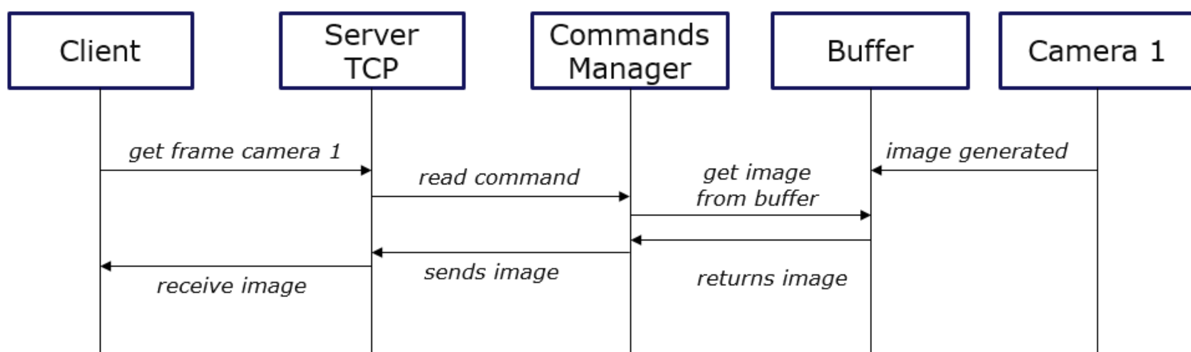


Figure 4.10: Sample diagram for communication.

(a) Menu using the console. (b) Dummy user interface.

Figure 4.11: Screenshots of the dummy application

4.8 Applications

4.8.1 Dummy camera

In order to test the functionality of the simulator, a dummy console application for Linux is built using the API provided. This application connects to the simulator trough an IP passed it on argument, and allows to create a camera, to generate a broadcast and to handle it visually trough the User Interface library. Once the application is started, a minimalist menu appears on console (Fig 4.11a) where you can find options to establish connection to the MSS simulator, create a single camera, change its properties and delete the camera from the simulator.

When you select the broadcast option, a new thread is created where it is continually asking for frames on background at the same speed as the framerate property. The frames received are display by the screen of this Figure 4.11b. The camera can be placed along the scenario with the buttons below. If you need to control more cameras, you can execute as many dummy instances as necessary.

4.8.2 Pedestrian detection

To demonstrate the potential use of the simulator for research, we build a pedestrian detection application. This application connects to the simulator, creates a new camera into a specific place and starts a streaming where we can see pedestrian localized in real time. The algorithm implemented for the pedestrian detection, based on this example [5], analyses each frame looking for the shape and the model corresponding to a person. In chapter 5, we analyse the effectiveness of the algorithm.

[5] http://www.magicandlove.com/blog/2011/12/04/people-detection-sample-from-opencv/

Chapter 5

Experimental work

This chapter presents the experiments performed to test the simulator described in chapters 3 and 4. First, we check the overall system and functionality developed. Later, we evaluate the performance of the frame generation process including the frame conversion process (section 5.5) to identify how many frames the simulator is able to generate depending on resolution and image format parameters. In sec 5.6, we test the computational resources employed. Finally in sec 5.7, we monitor the network usage in different situations to test a distributed computing environment.

5.1 Set-Up conditions

In all experiments we use a compiled version from the project explained in section 3.2 which includes the three modules developed and the EPS building scenario designed. This version is an 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).

For each experiment in this chapter, we create a new instance in the simulator and we connect one client that configures the experiment conditions. Each experiment runs for a large period of time (2-3 hours) with the purpose to get valid results and to collect the data necessary for graphics and tables. The tools employed in the experiments are depicted in Table 5.1.

Table 5.1: Tools used in the experiments.

Tool	Developer	Purpose
Performance Monitor	Microsoft	To collect CPU and main memory usage while runs each experiment.
GPU-Z	TechpowerUp	To collect GPU resources while runs each experiment.
Gnuplot	Opensource	To generate figures with average data.
MatLab	Mathworks	To generate figures with boxplot and to calculate the mean and variance.
Excel	Microsoft	To prepare the data collected for Gnuplot and MatLab.

5.2 System Testing

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 chapter 3. 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.

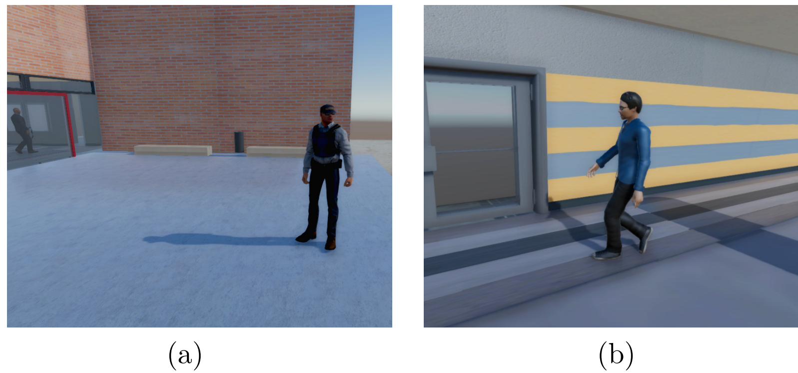
In this experiment we design 12 different common actions and the output expected. After, researchers from the VPULab research group (Universidad Autónoma de Madrid) tested several times each action and log the results. This results are depicted in Table 5.2.

Table 5.2: 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 a 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
A broadcast from the camera.	A GUI screen where the frames are display.	OK
Create several cameras.	Confirmation message on the console with all cameras.	OK
A broadcast from several cameras simultaneously.	Several screens with the broadcast from each camera.	OK
Place the camera dynamically while is broadcasting with the buttons available in the GUI screen.	Changes will be reflected in the screen.	OK
Change camera's properties (FOV, framerate) while is broadcasting.	Changes will be reflected in the screen.	OK
Test both image types JPEG and PNG.	Fluent broadcast in both cases.	It has been detected some problems with PNG type.

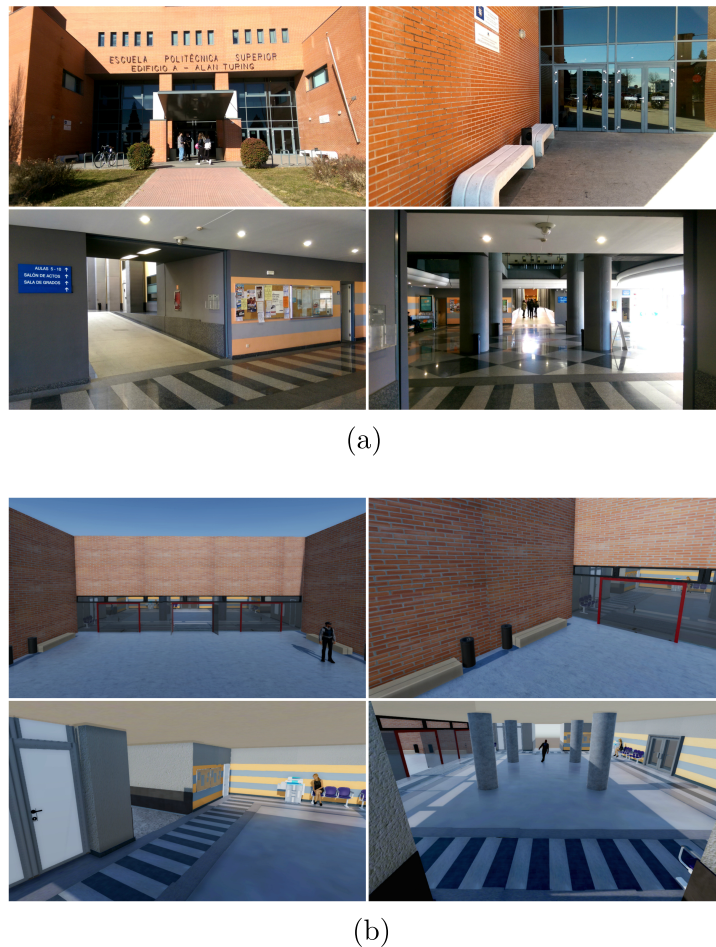
5.3 Simulated Scenario

In all experiments, we use the modeled scenario for the EPS building. Examples for the pedestrians included in the scenario are presented in Figure 5.1 and comparisons between real pictures and the modeled scenario are provided in Figure 5.2.



(a) (b)

Figure 5.1: Examples of 3D humans models in the scenario: (a) police and (b) student.



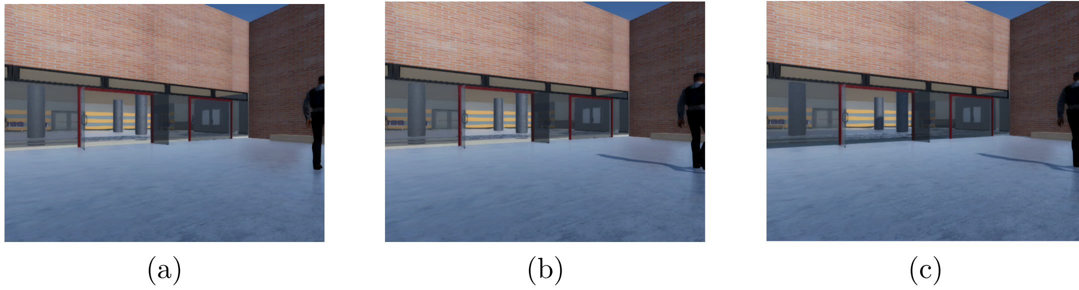
(a)

(b)

Figure 5.2: Comparative between (a) pictures of the 'Alan Turing' building and (b) virtual scenario modeled.

In the scenario, there are three different quality graphic options that can be chosen when

we start the simulator (see Figure 5.3). The *Fast* option is the lower configuration, with no shadows and a poor detail. The *Simple* option is the recommend quality option in the most cases, with a good compromise between quality and GPU consumption. Finally, the *Beautiful* option has the biggest graphics quality but is only recommended for powerful GPUs. We use the *Simple* option in all experiments except in the frame generation performance experiment.



(a) (b) (c)

Figure 5.3: . Different graphic quality options: (a) Fast (b) Simple and (c) Beautiful.

5.4 Frame generation performance

To evaluate the number of frames generated in one second, we measure the process explained in section 4.3.2. In this experiment, we focus in the frame generation process and not the frame conversion process so only module 1 is involved.

In Figure 5.4, 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.

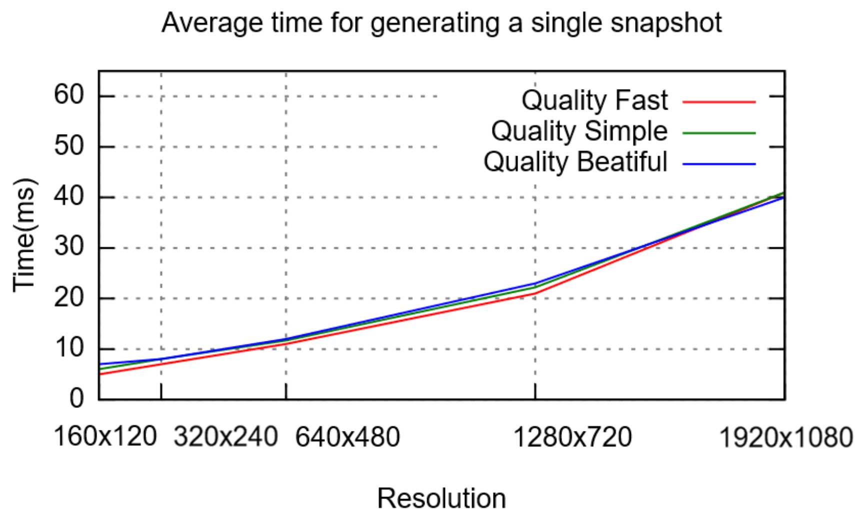


Figure 5.4: Time to generate one RAW frame for different resolutions and graphic qualities.

5.5 Frame conversion performance

We evaluate the frame conversion process (Module 2) 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 5.5, 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.

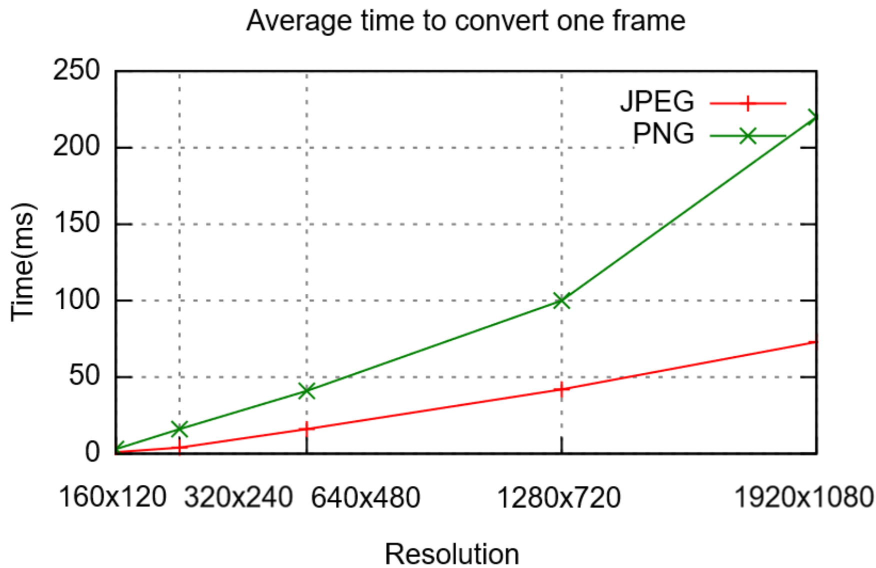


Figure 5.5: Average conversion time for different resolutions and encoders (JPEG and PNG).

In Figure 5.6, we appreciate a significant contrast in the image size depending on the format defined. For example, a 3MB (1280x720) image or 6 MB (1920x1080) is not optimal to be send over the network so this configuration is certainly not recommend in the most cases. On the other hand, the JPEG decoder implements an important compression with images of 85 KB (640x480), 170 KB (1280x720) and 341 KB (1920x1080). Finally, we present Figure 5.7 to check the final image quality. We can observe noise and less detail on JPEG format but it has to be considered that a zoom level of 300% is applied for this figure because we can not visually appreciate any significant difference in full size.

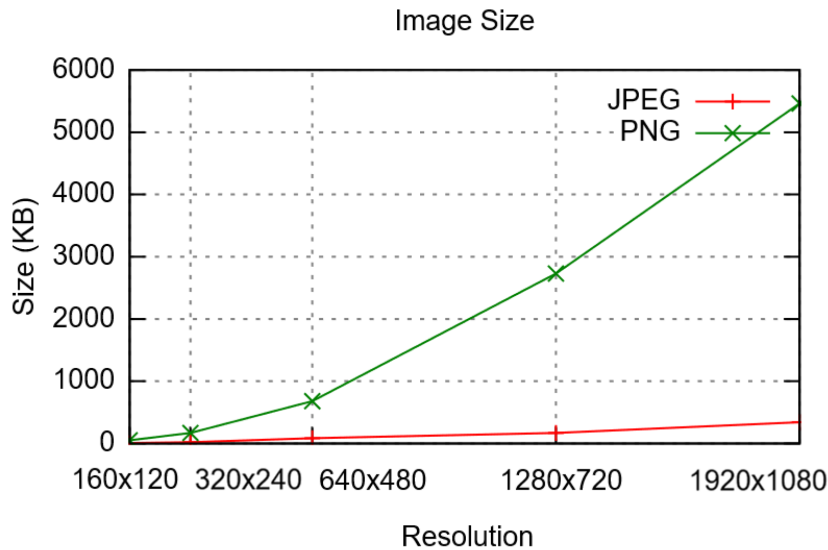
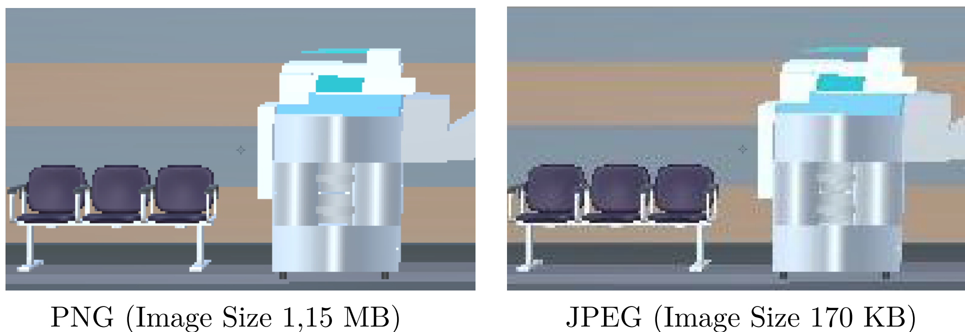


Figure 5.6: Average conversion size for different resolutions and encoders (JPEG and PNG).



PNG (Image Size 1,15 MB) JPEG (Image Size 170 KB)

Figure 5.7: Comparative of image quality with a 300% zoom applied.

5.6 CPU and GPU usage

We evaluate the performance of the simulator for one or multiple cameras in terms of the processing resources employed. We measure the CPU and GPU usage for three experiments concerning the framerate, resolution and multiple cameras. We also measure the RAM usage in all experiments but it is not referenced on this document because we do not observe any significant change (less than 1GB in any experiment).

5.6.1 Framerate

In this experiment we measure processing resources employed by the simulator running one camera with different framerates. The conditions of the experiment are:

- One camera is placed and configured with 640x480 resolution and image format JPEG.
- Camera's framerate starts at 5 fps and changes periodically with increments of 5 fps.
- No clients connects while the experiment is running.

The CPU usage results are depicted in Figure 5.8 grouped by framerate. This CPU Usage is calculated based on the following equation $CPU - usage = 100\frac{CPU\,time}{Total\,time}$.

For the maximum framerate (30 fps) and one camera, there is a 8% CPU usage. Each framerate increase has an equivalent CPU increment of 1%. Although we use a powerful computer for tests, this trend of 1% increase number is reasonable for most computers.

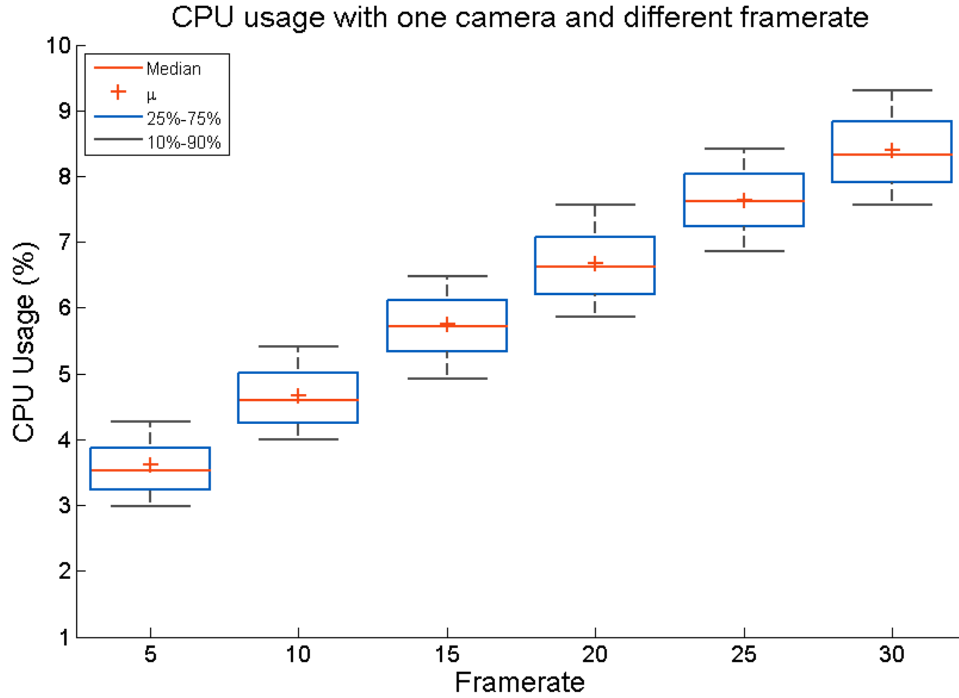


Figure 5.8: CPU usage with one camera and different framerates.

The GPU resources for the same experiment are depicted in Table 5.3. The maximum GPU Load presents about 12% but we should consider that the graphics card runs the whole experiment with its minimum frequency (135 MHz). Based on this results, most computers are able to run it with at least one camera with 640x480 resolution and the maximum framerate.

Table 5.3: GPU Resources with one camera grouped by framerate

	Resources		
Framerate	GPU Load (%)	Memory Load (%)	GPU Frequency (MHz)
5	2.4 ± 0.76	2.8 ± 0.36	135 ± 0
10	4.1 ± 1.44	1.6 ± 0.23	135 ± 0
15	6.2 ± 3.0	3.5 ± 1.5	135 ± 0
20	8.3 ± 2.2	4.8 ± 0.11	135 ± 0
25	10.1 ± 1.02	5.3 ± 1.04	135 ± 0
30	12.4 ± 0.5	4.9 ± 0.3	135 ± 0

5.6.2 Resolution

We measure processing resources employed by the simulator running one camera with different resolutions. The conditions for the experiment are:

- One camera is placed and configured with 15 fps and image format JPEG.
- Resolution starts at 160x120 and periodically it is changed to the next standard resolution employed in Computer Vision: 320x240, 640x480, 1280x720 and 1920x1080.

- No clients connected while the experiment is running.

The CPU usage results in the experiment are depicted in Figure 5.9 grouped by resolution. With Full HD images it presents about a 12% usage which can be considered a high CPU consumption for a single camera. However, smallest resolutions present no difference on the CPU Usage so we can understand that the simulator by itself spends about 3%-4% CPU usage.

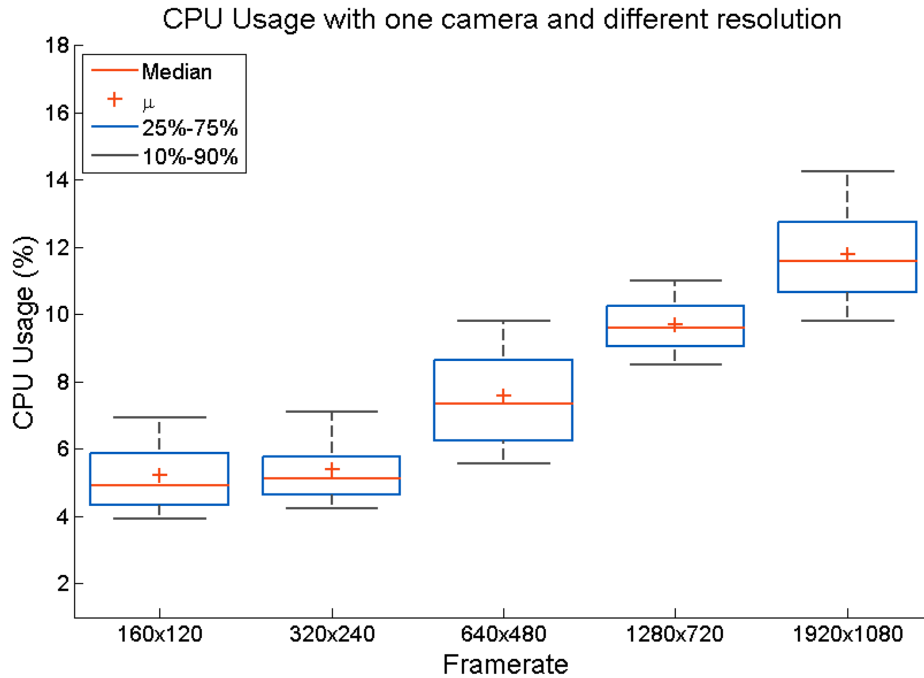


Figure 5.9: CPU Usage with one camera and different resolution.

The GPU resources for the same experiment are depicted in Table 5.4. In this experiment, the graphic cards also runs with the minimum frequency. The difference of GPU Load between 1280x720 and 1920x1080 is an enormous 11% so we conclude that this configuration is only recommended for special cases.

Table 5.4: GPU Resources with one camera grouped by resolution

	Resources		
Resolution	GPU Load (%)	Memory Load (%)	GPU Frequency (MHz)
160x120	3.7 ± 0.7	1.6 ± 0.5	135 ± 0
320x240	4.1 ± 0.88	1.9 ± 0.1	135 ± 0
640x480	6.8 ± 1.5	2.4 ± 0.2	135 ± 0
1280x720	11.5 ± 4.6	4.1 ± 0.7	135 ± 0
1920x1080	22.4 ± 16.4	7.9 ± 3.06	135 ± 0

5.6.3 Several Cameras

We measure the processing resources employed by the simulator running several cameras simultaneously with the purpose to identify the maximum number of cameras. The conditions of the experiment are:

- All cameras have the same configuration: 640x480 @ 25 fps. We choose this configuration because is a standard in video surveillance. In particular, this configuration is often used in some cameras installed in the campus of the Universidad Autónoma de Madrid.

- A new camera is periodically created in the simulator.

- No connected clients while the experiment is running except the creation of new cameras.

- We force the graphic card (Titan X) to work at the maximum performance mode.

The CPU Usage results in the experiment are depicted in Figure 5.10 grouped by number of cameras active simultaneously in the simulator. It starts with 5% which is a number that corresponds with the previous experiments and an increment of 4%-2% for each new camera until the seventh camera, where we see a lineal usage until the end of the experiment. This is caused because we reach the maximum of our system are it is explain in the next paragraph.

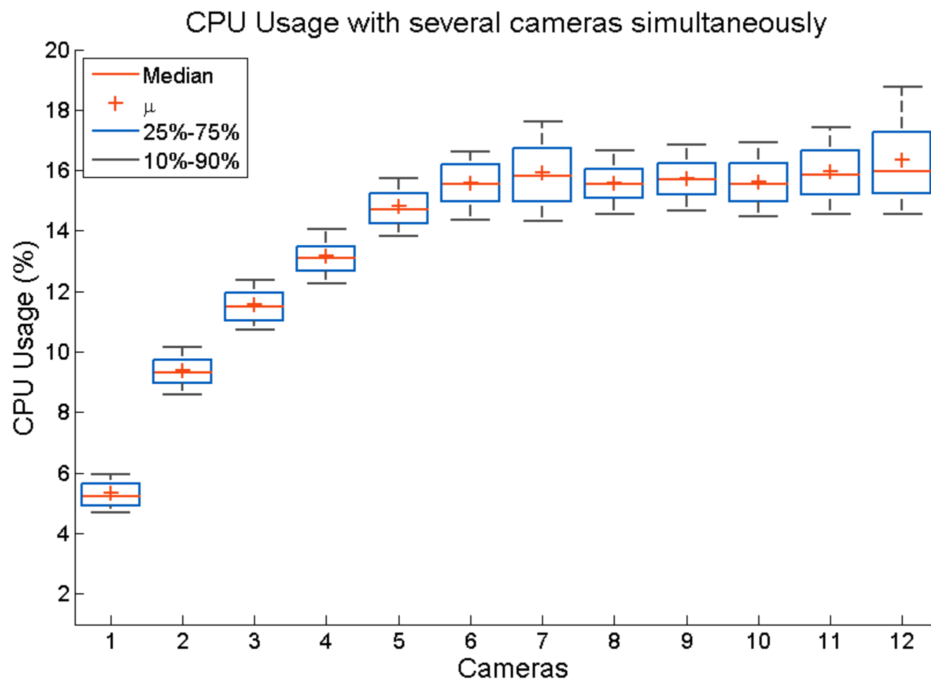


Figure 5.10: CPU usage with several cameras simultaneously configured at 640x480 @ 25 fps.

The GPU resources for the same experiment are depicted in Table 5.5. It is important to consider that in this experiment we manually force the graphic card to work at its maximum frequency (1151 MHz) and this is the reason why the maximum of frames does not match with experiment 1. But anyway, the results are equivalent to the previous figures. From beginning to the sixth camera present a increment for each new camera, but for the successive not. Although we add more cameras, the simulator does not generate more frames. When the simulator achieves the maximum throughput of frames, such number of frames is allocated between the old and new cameras added to the simulator.

Table 5.5: GPU Resources with several cameras working.

	Resources		
Cameras	GPU Load (%)	Memory Load (%)	GPU Frequency (MHz)
1	2.1 ± 0.14	1.0 ± 0.0	1151 ± 0
2	4.4 ± 0.24	1.0 ± 0.1	1151 ± 0
3	6.4 ± 0.24	1.0 ± 0.1	1151 ± 0
4	8.2 ± 0.16	2.0 ± 0.1	1151 ± 0
5	9.9 ± 0.05	2.0 ± 0.1	1151 ± 0
6	11.34 ± 0.27	2.5 ± 0.24	1151 ± 0
7	11.7 ± 0.18	2.8 ± 0.21	1151 ± 0
8	11.3 ± 0.23	3.0 ± 0.1	1151 ± 0
9	11.7 ± 0.16	2.7 ± 0.17	1151 ± 0
10	11.4 ± 0.26	2.4 ± 0.24	1151 ± 0
11	11.7 ± 0.22	2.6 ± 0.23	1151 ± 0
12	11.5 ± 0.62	2.7 ± 0.20	1151 ± 0

As we said at the beginning, we placed cameras configured with 25 fps. But we know for the first experiment that our system has a maximum number of frames which is able to generate in one second depending on the resolution. Therefore, there is a situation where the simulator reaches the maximum throughput of frames (i.e. generating the desired number of frames takes almost the same time indicated by the framerate) and adding new cameras implies that the simulator needs to decrease the framerate proportionally. This results are depicted in Figure 5.11. In this case, we reach the maximum with the sixth camera as we see with a tiny decrement. This decrement is proportional to the successive cameras. For example, with 7 cameras we have 20 fps x 7 cameras = 140 frames generated in one second. With 12 cameras, we have 11 fps x 12 cameras = 132 frames generated. We can infer that these numbers are equivalents.

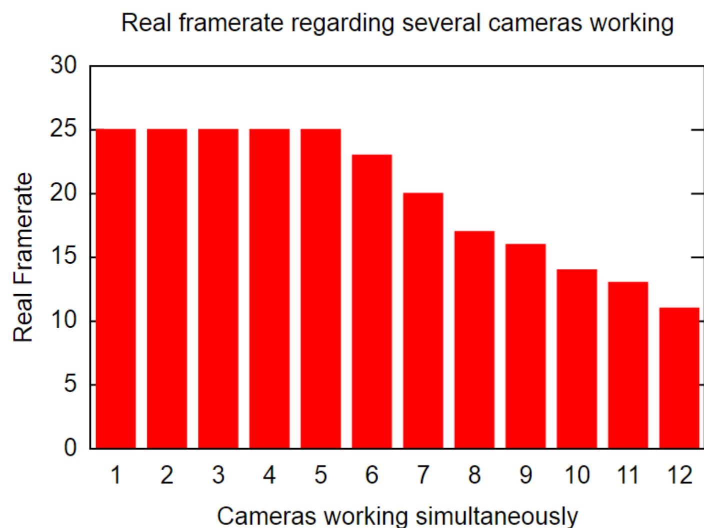


Figure 5.11: Real Framerate regarding several cameras working simultaneously. All cameras have the same configuration: 640x480 @ 25. We expect 25 fps until it reach the maximum of frames that our system is able to generate in one second.

5.7 Network Usage

In this experiment we measure the network usage in different situations in order to determinate how much bandwidth is necessary when we use the simulator in a distributed environment (the alternative is localhost). We use the Dummy application to create cameras and to generate a broadcast for each camera. Two computers are employed to host the simulator and the Dummy application, having both 100Mb/s Ethernet connections.

The results are depicted in Table 5.7. In the second experiment we obtain that the size for one 640x480 frame is 85 KB so to transmit 25 frames we theoretically need 85 KB x 25 = 2.04 MB/s. This is not what we see in the tablebecause data compression is applied before the frame is sent to clients using a Base64 encoder. It is interesting that our computer does not have enough bandwidth to transmit 25 fps in 1920x1080 resolution. In any case, the network usage is a key resource when streaming data to algorithms or clients.

Table 5.6: Network Usage in different cases.

Description	Streaming speed	Image Format	Network/Bandwidth Usage
1 cam @ 320x240	25 fps	JPEG	270 KB/s
2 cam @ 320x240	25 fps	JPEG	530 KB/s
1 cam @ 640x480	10 fps	JPEG	460 KB/s
1 cam @ 640x480	25 fps	JPEG	1 MB/s
2 cam @ 640x480	25 fps	JPEG	2.2 MB/s
1 cam @ 1280x720	25 fps	JPEG	2.8 MB/s
1 cam @ 1920x1080	10 fps	JPEG	1.7 MB/s
1 cam @ 1920x1080	25 fps	JPEG	Not enough bandwidth

5.8 Application example: people detection

In this experiment, we check the effectiveness of the pedestrian detection application by creating three different cameras in the same place that they are placed on the real surveillance system installed in the EPS building. Each camera incorporate the pedestrian detection algorithm. The goal of this experiment is to identify all the pedestrian in the scenario without false cases.

The results are depicted in Figure 5.12 and show an algorithm reliability about 50% correct. It presents two cases where can not correctly detect pedestrians. As this algorithm employs human shapes, a common mistake is to identify columns as humans. This happens in some cases because the shape of columns and humans is very similar in size and structure. An area of improvement may be to look for parts of humans instead the complete body such as the head, trunk and legs. On the other hand, this algorithm is calibrated for a minimum and maximum human size. Thus it does not detect human who are far away from the camera. We could adjust this calibration for an optimal result.

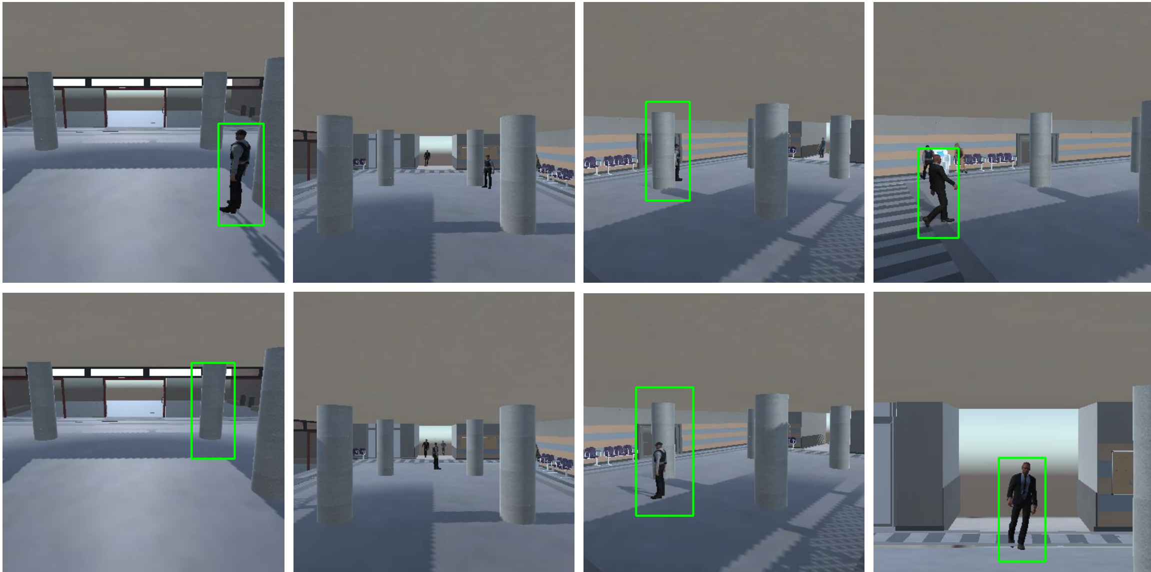


Figure 5.12: Pedestrian detection results.

5.9 Conclusion

Based on the experimental results, we identify two bottlenecks in the simulator.

First, a limit in the maximum of frames that can be generated in one second. The process of generating a frame is complex involving the GPU and CPU. The graphic card generates the frame, but we need it in main memory so there is a data transfer between GPU and CPU which takes time. Usually, GPU and CPU works asynchronous so *Framerate = max(CPU time, GPU time)*. But when we want to transfer data between these units it introduces a "sync point" that causes a stall. This means that both CPU and GPU must take turns to run while the other one sits idle. As result, framerate is CPU time + GPU time at this moment. The frame generation process is not a process that can be parallelized so *this frame generation bottleneck stablishes the maximum of cameras that can run simultaneously in the simulator.*

On the other hand, another bottleneck is the image conversion process which has a high computational cost for both JPEG and PNG. Fortunately, we parallel this process through the buffer module to decrease the overall cost. However, *this image-conversion bottleneck stablishes the maximum framerate of the cameras.* We can improve the performance n two ways: better graphics card (will help to generate more frames per second, so we would able to run more cameras or a higher configuration) and better processor (will help to encode more images per second). Better improvements will be observed for higher resolutions.

Finally, it is important to mention that the network is key in a distributed computing environment. For most cases, it should be enough a Gigabit or optical fiber connectivity.

Chapter 6

Conclusions and future work

6.1 Conclusions

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 that we review on chapter 2. 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.

On the other hand, the modular design of the system allows future extensions. For example, if you want to add a new camera property (e.g. zoom), you only need to add the new parameter in the Camera class and create two methods for getting and setting this property. Later, you could add a new command for this property by developing a few lines of code in the Cameras Controller class and Commands Manager class. Finally, it can be modeled others type of cameras by designing a new camera object. In any case, the logic and the behavior of the system are not necessary to modify even to understand by the programmer. To facility future extensions, a documentation of the simulator's classes is included using Doxygen tool.

A summary of the features developed are:

- Multiple virtual cameras distributed modeling a pinhole camera. Parameters configurable available: freely location configuration (position, rotation), framerate, resolution, image type and FOV.
- A system manages the cameras that allows controlling them remotely and dynamic configuration of camera's properties.
- A Buffer for each camera which stores the frames generated on main memory.
- An Image Processing class that converts the frames type into a standard image type such as JPEG and PNG. This class can be extended in the future for more functionalities (e.g. others formats, image analysis, etc).

- A server for simultaneous connection of several third party applications (e.g. computer vision algorithms).
- A virtual scenario is included, modeling the hall of the 'Escuela Politecnica Superior' building with autonomous pedestrians.

6.2 Future work

As we detail in chapter 5, this system presents some technical limitations. The process of generating a frame takes a high computational cost because involves GPU programming, CPU operations and transfer data between this units and main memory. In order to improve this process, we propose:

- The performance in the generation frame process (Module 1) is limited mainly by the time spent in the data transfer between the GPU and the CPU which causes a stall. To reduce the time spent in this stall, an alternative is to pipeline the process explained in section 4.3.3. This alternative is based in the publication of Jeremy Cowles in [1]. He proposes a method where the frame generation process can be pipelined and executed in parallel which reduces the stall. This method is fundament in if we introduces a tiny delay between each step of the frame generation process, GPU and CPU that runs in parallel, finish their operation and when we do the data transfer, it takes less time. By improving this process, we will be able to generate more frames in one second which increases the number of cameras to be running in the simulator.
- To improve the frame conversion process (Module 1) is needed a faster decoders. Usually, the faster decoders in the market are private but we find an alternative to the GDI+ API library. This alternative is the open-source libjpeg-turbo library[2]. This library contains a JPEG decoder that accelerates the JPEG compression and decompression on x86, x86-64 and ARM systems. The performance of libjpeg-turbo rivals to some proprietary high-speed JPEG codecs. According to the authors, with this library we can expected a 2x-6x improve in the frame conversion process. The main disadvantage of this library is that not includes decoders for other types than JPEG.
- An interesting feature that can be develop in the future is a system which allows to create and place cameras visually. This features facilities the use of the simulator for people with limit technical knowledge or simply can help in this process saving time.

[1] https://goo.gl/hsJJiC
[2] http://www.libjpeg-turbo.org/

Bibliography

[1] X. Wang, E. TÃŒretken, F. Fleuret, and P. Fua, "Tracking interacting objects using intertwined flows," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 38, pp. 2312–2326, Nov 2016. xi, 2

[2] A. Gaidon, Q. Wang, Y. Cabon, and E. Vig, "Virtual worlds as proxy for multi-object tracking analysis," in *CVPR*, 2016. xi, 2, 51

[3] J. Peddie, K. Akeley, P. Debevec, E. Fonseka, M. Mangan, and M. Raphael, "A vision for computer vision: Emerging technologies," in *ACM SIGGRAPH 2016 Panels*, SIGGRAPH '16, (New York, NY, USA), pp. 2:1–2:2, ACM, 2016. 1

[4] E. Lantsova, T. Voitiuk, T. Zudilova, and A. Kaarna, "Using low-quality video sequences for fish detection and tracking," in *2016 SAI Computing Conference (SAI)*, pp. 426–433, July 2016. 1

[5] N. Corporation, "The ai car computer for self-driving vehicles," 2016. 1

[6] L. Esterle, P. R. Lewis, H. Caine, X. Yao, and B. Rinner, "Camsim: A distributed smart camera network simulator," in *2013 IEEE 7th International Conference on Self-Adaptation and Self-Organizing Systems Workshops*, pp. 19–20, Sept 2013. 5

[7] J. SanMiguel and A. Cavallaro, "Networked computer vision: the importance of a holistic simulator," *IEEE Computer*, pp. 1–7, (to appear) 2017. 5

[8] G. R. Taylor, A. J. Chosak, and P. C. Brewer, "Ovvv: Using virtual worlds to design and evaluate surveillance systems," in *2007 IEEE Conference on Computer Vision and Pattern Recognition*, pp. 1–8, June 2007. 5, 6, 8

[9] L. P. Llorente, "Simulador virtual para sistemas multi-camara distribuidos," Master's thesis, Escuela Politecnica Superior (Universidad Autonoma de Madrid), July 2015. 6, 8

[10] W. Qiu and A. Yuille, "Unrealcv: Connecting computer vision to unreal engine," *arXiv preprint arXiv:1609.01326*, 2016. 6, 8

[11] W. Starzyk and F. Z. Qureshi, "Software laboratory for camera networks research," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 3, pp. 284–293, June 2013. 7, 8

[12] P. Dollar, C. Wojek, B. Schiele, and P. Perona, "Pedestrian detection: An evaluation of the state of the art," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 34, pp. 743–761, April 2012. 7

[13] H. Hattori, V. Naresh Boddeti, K. M. Kitani, and T. Kanade, "Learning scene-specific pedestrian detectors without real data," in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2015. 7, 8

[14] M. Lewis, J. Jacobson, and C. based Games, "Game engines in scientific research," 2002. 49

[15] D. BENOIT, "Unreal paris," 2015. 50

[16] S. R. Richter, V. Vineet, S. Roth, and V. Koltun, “Playing for data: Ground truth from computer games,” in *European Conference on Computer Vision (ECCV)* (B. Leibe, J. Matas, N. Sebe, and M. Welling, eds.), vol. 9906 of *LNCS*, pp. 102–118, Springer International Publishing, 2016. 51

[17] J. E. Laird, M. Assanie, B. Bachelor, N. Benninghoff, S. Enam, B. Jones, A. Kerfoot, C. Lauver, B. Magerko, J. Sheiman, D. Stokes, and S. Wallace, “A test bed for developing intelligent synthetic characters,” in *Artificial Intelligence and Interactive Entertainment II* (K. Forbus and M. S. El-Nasr, eds.), no. SS-02-01 in AAAI Technical Report, (Menlo Park, California), pp. 52–56, AAAI Press, 2002. 51

[18] W. Knight, “Minecraft is a testing ground for human-ai collaboration,” 2016. 51

[19] W. Sheng, K. Ishikawa, H. T. Tanaka, A. Tsukamoto, and S. Tanaka, “Photorealistic vr space reproductions of historical kyoto sites based on a next-generation 3d game engine,” *Journal of Advanced Simulation in Science and Engineering*, vol. 1, no. 1, pp. 188–204, 2015. 51

[20] J. W. Hu, C. Feng, Y. Liu, and R. Y. Zhu, “Utse: A game engine-based simulation environemnt for agent,” in *Frontiers of Manufacturing and Design Science IV*, vol. 496 of *Applied Mechanics and Materials*, pp. 2142–2145, Trans Tech Publications, 4 2014. 51

[21] C. M. D. Science and T. G. Australia, “Simulating a flatfish unmanned underwater vehicle in a unity 3d maritime environment,” in *Simulating a Flatfish Unmanned Underwater Vehicle in a Unity 3D Maritime Environment*, 2016. 51

Appendix A

Tutorial for adding the simulator in to a project in Unity

This tutorial shows the steps necessary to incorporate the simulator in to an existing Unity project or in a new one. For this tutorial is necessary a previous installation of Unity and a copy of the MSS source folder.

First, localized the MSS source folder. This folder follows this structure:

- Documentation folder: contains the documentation generated with doxygen and tutorials for an easy implementation in your project.

- Plugins: third-party libraries that are necessary to incorporate. For example, the System.Drawing.dll contains the encoders for the frame conversion to JPEG and PNG.

- Resource: some extra material such as images or files configuration.

- Simulador Code: contains all classes developed and the simulator logic.

- Simulator Objects: contains the objects relative with the simulator.

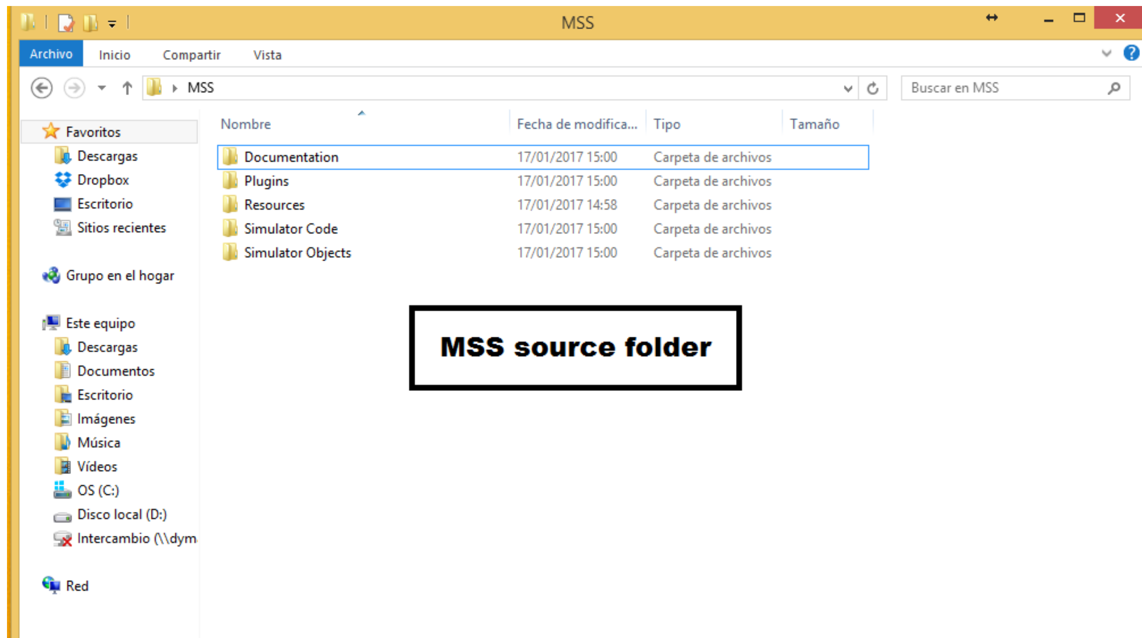


Figure A.1: MSS source folder structure

Open Unity. The version installed is 5.6.0 and this screen may be a little different in the future. Now, select your project or create a new one. At this example, we create a new project. If Unity asks, it is important to select 3D option.

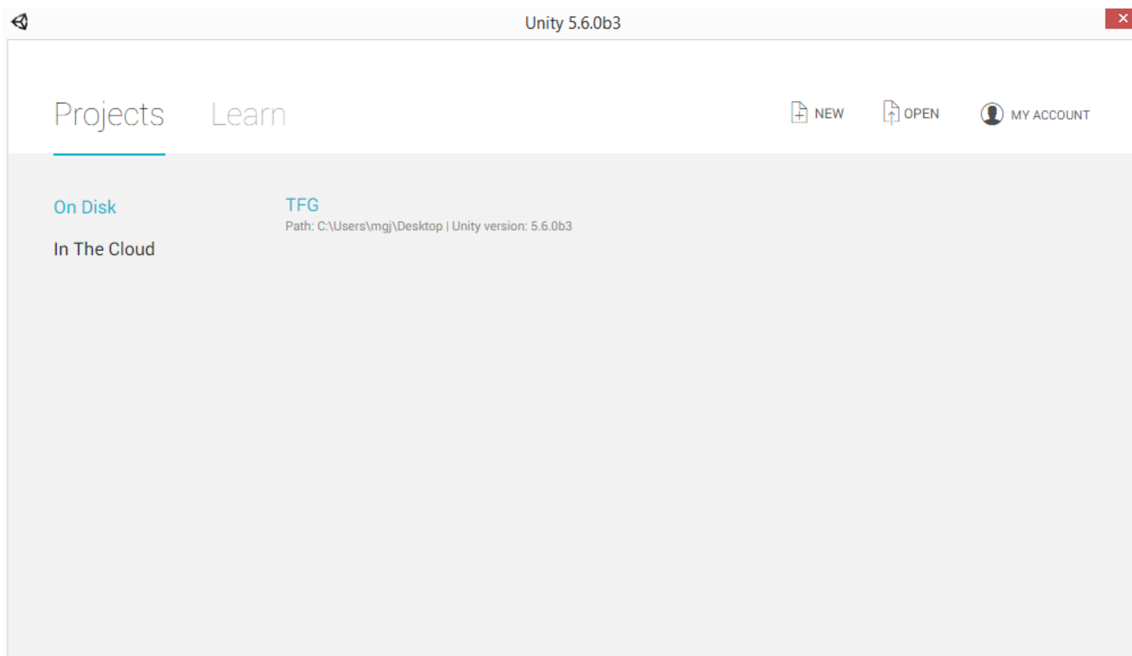


Figure A.2: Unity start screen

This screen is the Unity user interface. It is organised in different panels. We mark the most relevant:

1. This panel shows your source folder. Here you have your scripts, 3D models, objects, textures and plugins. In this case is empty because we just create the project. You can

organised all content as you want except the plugins that must be in the 'Plugins' folder.

2. This panel shows the virtual word. You can select objects and navigate with the mouse.

3. This panel shows a list of all GameObjects active in the scene. A scene is a level, equivalent to a main class. You can have several scenes in the project. By default, there are a camera GameObject and a light.

4. This panel shows the GameObject selected properties. Here you find all components attached including scripts.

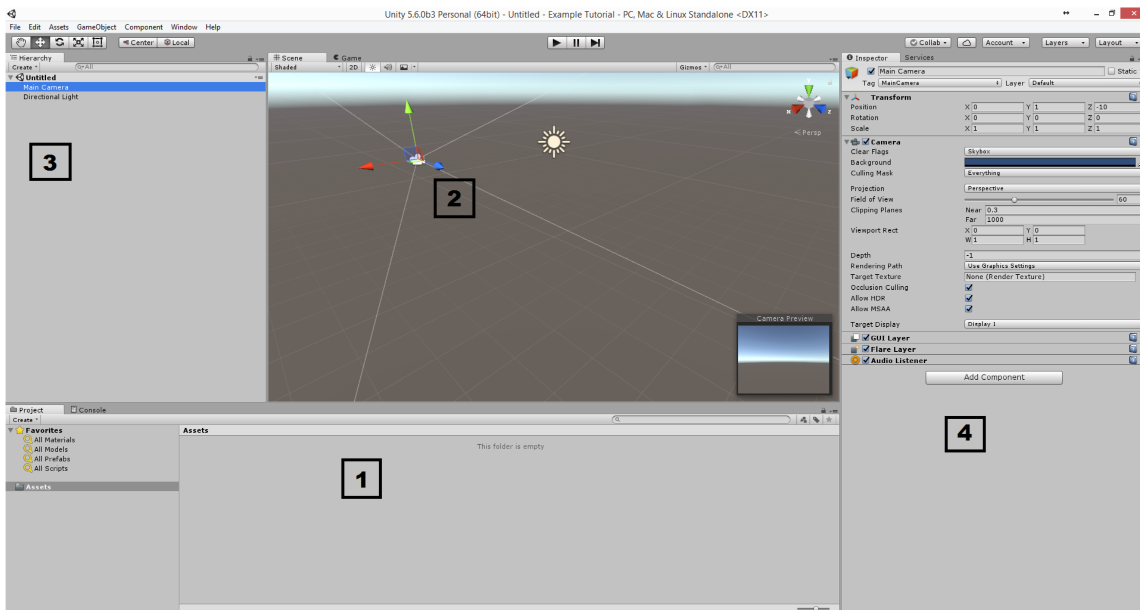


Figure A.3: Unity User Interface

Now, go to your folder project which has this structure. 'Assets' is the only folder that you will use'. This folder is the source folder that its mark in the previous Figure. Copy inside this folder all folders from MSS (documentation, resources, plugins, simulator code and simulator objects).

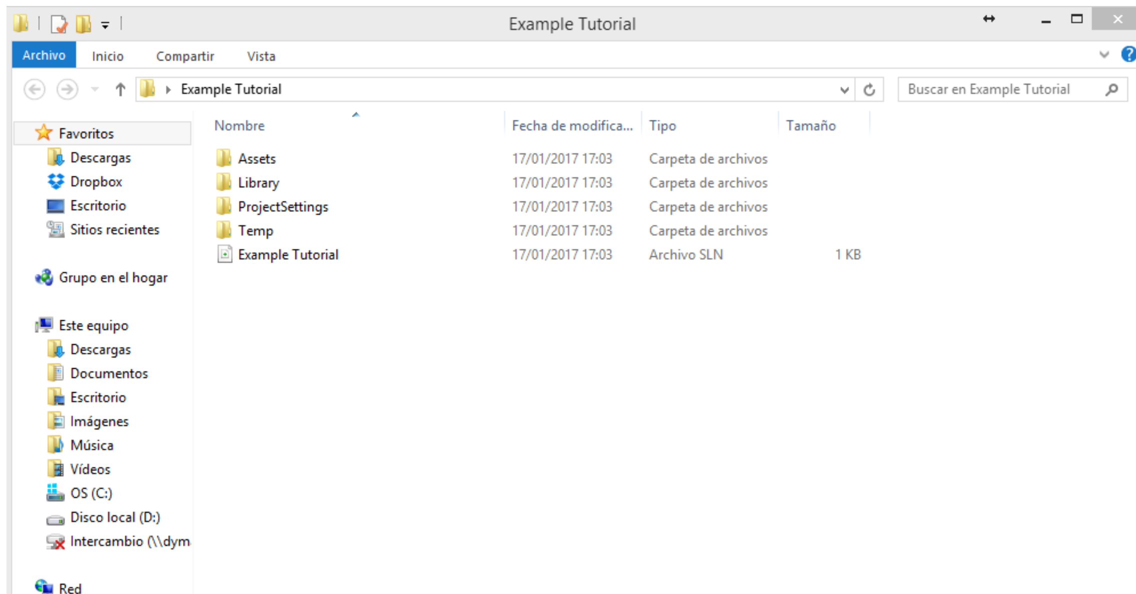


Figure A.4: Unity User Interface

At this moment, Unity automatically detect the new content added in the project and recompile it. Now, you can use the simulator. To add the simulator in to the scene, go to the 'Simulator Objects' folder, select 'StartSimulator' and drag and drop in to the left panel.

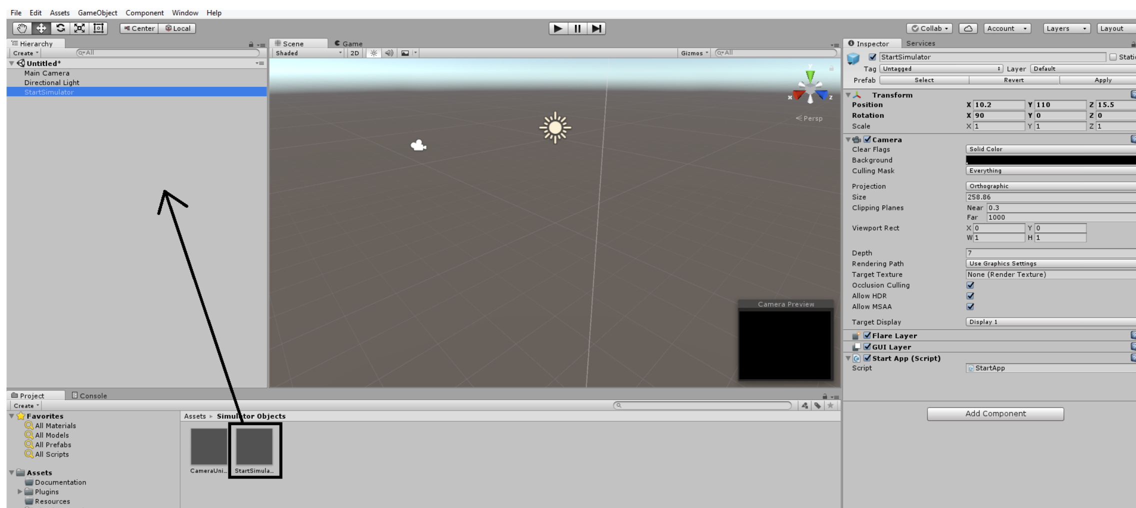


Figure A.5: Unity User Interface

Now, you are ready to use the simulator when you press the 'play' bottom on the top or when you export your project.

Appendix B

Game engines

B.1 Introduction

In the last years, the computer and video games industry has grown so far that the most sophisticated rendering pipelines, interactive or physics simulations are no longer exclusively running on an expensive specialized scientific machines. They are now operating on personal computers or game devices at significantly lower cost. This is possible due to the widespread of software frameworks where games are designed and created.

Games are constantly evolving, offering new graphics techniques and realistic simulations using artists, programmers and even real actors. Traditionally, the cost to develop these games was really high that even the biggest companies were not able to return the investment in one single game. In this context appears the concept of a game engine as a software framework to create video games. Companies have been forced to use modular systems enabling to reuse code from one game to another or for a family of similar games. Such modular components may be employed in scientific research, recently increasing the interest in realistic simulation.

A game engine is a collection of modules of simulation code that do not directly specify the game's behaviour (game logic) or game's environment (level data). The engine includes modules for rendering (generate images), input devices, sound, networking and physics. However, the games behaviour and environment must be specified by the game programmer. In Figure B.1 we can see a simplified scheme and the relations between components.

For research purposes, it is crucial precise behaviour of all processes involved, and all game

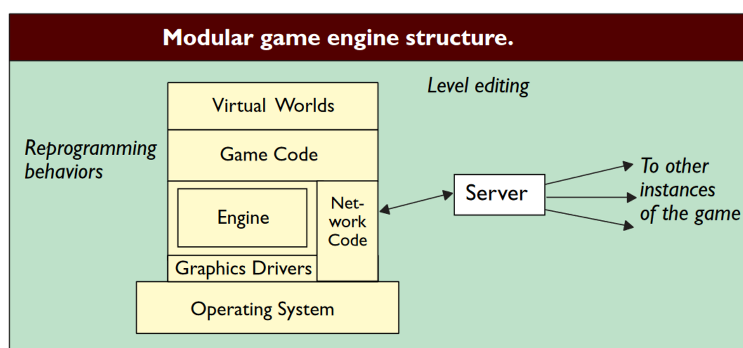


Figure B.1: Game Engine components. Source: [14]

Figure B.2: Unreal Engine 4 Tech Demo Unreal Paris by Benoît Dereau. Source: [15]

engines do not provide the same features, so you need to look the most appropriate engine for a specific task. In the next section we review some popular not private game engines

B.2 Unreal Engine

Unreal Engine or UE4 (https://www.unrealengine.com/) is a game engine developed by Epic Games, initially released on 1998. Its most recent version is known as UE4 4 (UE4). Although originally was developed for first-person shooters, it has been successfully used in a variety of other genres. It's the non-private game engine most used on the game industry due to the full source code access allowing to customize and to extend all Unreal tools and subsystems. It has a royalty license: a 5% royalty after the first $3,000 of revenue per product per quarter on games and applications you release. No royalty is paid for film projects, contracting and consulting projects such as architecture, simulation and visualization. UE4 enables to deploy games for many platforms: Windows PC, PlayStation 4, Xbox One, Android, etc.

Some important features are the compatibility with modern API graphics (DirectX 12[1], Vulkan[2]), support for Virtual Reality (VR) devices (HTC Vive[3], SteamVR[4]) and great expressiveness to create beautiful visuals for architectural visualizations, simulations or digital films as we can see in Figure B.2.

B.3 Amazon Lumberyard

CryEngine designed by the German Crytek company (https://aws.amazon.com/lumberyard/) has been the most technical and powerful game engine available on the market during years. On 2016, Crytek had financial troubles so the software was bought by Amazon and renamed as *Amazon Lumberyard*, a free cross-platform engine. The main advantage of Amazon Lumberyard has is a completely free license, including its full source code, to make PC and console games with any kind of royalty or subscription fee. It has only a restriction: you need to use Amazon Web Services (AWS Cloud) for multiplayer games. Interesting features included are a

[1] http://www.amd.com/en-us/innovations/software-technologies/directx12
[2] https://www.toptal.com/api-developers/a-brief-overview-of-vulkan-api
[3] https://www.vive.com/eu/
[4] http://store.steampowered.com/steamvr

drag-and-drop visual scripting tool and a library of 22 pre-built features that can be used to start new projects or prototype ideas.

B.4 Unity

Unity (https://unity3d.com/) is a cross-platform game engine developed by Unity Technologies and is used to develop games for PC, consoles and mainly mobiles devices. It became popular by offering the first industry-leading multiplatform support including Windows, Linux, Android and others. At beginning, it got the attention from the community thanks to Unity Web Player: a plugin to view 3D content created with Unity directly on your web browser. It also has the same license as UE4 but also has a subscription license with additional features.

Unity is the most widely used VR development platform because supports all VR devices: Oculus Rift, Gear VR, Playstation VR,Steam VR, HTC Vive and Microsoft HoloLens. Microsoft HoloLens ([5]) is Unity's exclusive thanks to Microsoft partner. Others important features are Unity Ads (monetize games), Unity Collaborate (teamwork) and integration to external plugins.

B.5 Uses in research

Hyper-realistic computer games may be used to teach Artificial Intelligence about the real world. For example, researchers from Intel Labs and Darmstadt University [16] use the Grand Theft Auto game to train machine-learning algorithms and to classify many objects such ascars and pedestrians. In [17], the authors present a virtual environment to investigate the role and behaviour of intelligent synthetic characters. Another example can be find on Project Palmo [18], a platform for AI research based on Minecraft game which provides a wide range of experimentation scenarios, ranging from survival to collaboration tasks.

For the UE4, we can find several projects in Virtual Reality such as photorealistic reproductions of historical places in [19]. In [20], authors describe a platform to develop virtual environments for agents.

Unity is another interesting tool in research. As we can see here [21], a Defence Science and Technology Group from Australia have used Unity for the implementation of an Unmanned Underwater Vehicle (UUV) model into 3D virtual maritime environment. In [2], authors propose an efficient method to clone real into virtual world including features for automatic tracking and objects detection. We can see an example from the 'Virtual Kitty' dataset in Figure B.3.

B.6 Conclusions

The reviewed games engines share similar specifications and features but they have key distinctive aspects. For example, UE4 and Unity have an official marketplace where you find free and paid content such as plug-ins, 3D models, scripts and more. This is very useful because you have a database where you quickly can find content ready to use on your projects. Besides, you can also sell your own content and earn money.

[5] https://www.microsoft.com/microsoft-hololens/en-us

Figure B.3: On top, a frame of a video from the Kitty tracking benchmark. On bottom, the same frame rendered with the synthetic clone tool.

On the other hand, Amazon Lumberyard has a common problem with new software released on the market: it does not have an extensive documentation and the developing community is small which helps and tutorials. For these reasons, we discarded Amazon Lumberyard.

UE4 as well as Unity have a great active community behind providing free resources like these extended tutorials for beginners [6] [7]. But we decided to use Unity for one reason: as UE4 full source code is provided, it has not an extensively developed API (i.e. you may need to create you own methods for your requirements). This is the opposite case on Unity, and we think it is easier to develop using the Unity API. Further, it includes extend cameras functionalities that we use in our system as it is explained in chapter 3. As conclusion, Table B.1 shows a comparative with some distinctive aspects.

Table B.1: Table comparative of game engines.

Game Engine	Free License	Source Code Access	Marketplace	Extra Services	Script Language
UE4 4	Yes	Yes	Yes	Only Third-party	C++
Amazon Lumberyard	Yes	Yes	No	Amazon Web Services, Twitch integration	C++
Unity	Yes	No	Yes	Unity Ads, Multiplayer, Performance Reporting,...	C# and JavaScript

[6]https://docs.unrealengine.com/latest/INT/Videos/
[7]http://www.virtualgamelab.com/unity-resources.html