VIDEO-MONITORIZATION SYSTEM FOR A REALISTIC DRIVING SIMULATOR

Master Thesis

To obtain the degree of Master at Instituto Superior de Engenharia do Porto, public defend on September by

Laura Dapena Formoso

Master in Telecommunication Engineering,
Porto, Portugal
2014
Supervisor:
Prof. Dr. Miguel Leitão

To my parents, and my grandparents.
Without them I had never achieved this challenge.

Author email: 1130928@isep.ipp.pt

Acknowledgments

I would like to thank all the people who support me and help me during this long journey, especially to my parents and my family, who ever support my decisions and choices.

Especial thanks also to my friends (As Pontes, Porto, Madrid, Ljubljana) for joining me along my entire student life and my boyfriend André, who was by my side supporting me in everything.

I would like to express my gratitude also to my supervisor Dr. J. Miguel Leitão, whose expertise, understanding and patience, added considerably to my graduate experience.

Many thanks too for ISEP and ETSIT for making the development of this thesis possible and allow me coming to Porto.

To finish, also say thanks to Mobile Team for support me everyday during the last 9 months.

Abstract

In the last few years, ISEP in collaboration with FEUP and other universities, created a realistic driving simulator called DriS, which had the objective to help in researches of different areas, as civil engineer, computer graphics, psychology, education, etc.

The result of this thesis pretends to help the professionals who analyse the data collected in each driving experience, in order to allow them the study of the driver's reactions at different obstacles during a ride, at the same time.

DriS simulator consists in one white screen where the driving simulators environments are projected, in one real car to make the driving experience and four cameras placed in the car. Of these four cameras, three are inside the car and one of them outside the car. Each camera is focused in one specific and critical part of the driving: the road, the driver, the pedals and the controls (gearshift, steering wheel, wiper controls, etc.).

Each one of the camera records a video that is save in one computer placed in the control room, inside the Laboratório de Análise de Tráfego in FEUP. Also, a text file is saved in this computer. This text file contains some information about the driver's experience, as it can be the car coordinates, the speed of the car, the time, etc.

The work of this thesis arises in order to improve the way on how professionals analyse and perform data collected from a DriS driving experience. For that purpose, was created a video-monitorization system, consists in a video application, that allows load and player four videos simultaneously as well as a text file which contains all the data collected from the experience. All of them will be time-coordinated and the user could move forward and backward through them using a slider. Also, as a basic video player, contains some buttons to control the status of the video (play, stop, pause) allowing the professionals analyse with detail the four videos and the data.

Take advantage of the new progresses in software development, the application was made in C++ using the Qt library, and its integrated development environment the Qt Creator, which made easier the implementation.

At the end of this report (Chapter 4) is attached a user manual in order to explain and help the professionals to use the application.

Resumen

En los últimos años, el ISEP en colaboración con FEUP y otras universidades, han creado un simulador de conducción llamado DriS, el cual tiene como objetivo ayudar en la investigación de diferentes áreas como ingeniería civil, computación gráfica, psicología, educación, etc.

El resultado de este proyecto pretende ayudar a los profesionales de estos sectores quienes analizan los datos recolectados en cada experiencia de conducción, para permitir el estudio simultáneo de las reacciones de los conductores a diferentes obstáculos durante el trayecto.

El simulador DriS consiste en una pantalla blanca donde son proyectados los ambientes de simulación; en un coche real en el cual se llevarán a cabo las experiencias de conducción; y en cuatro cámaras situadas en el coche. De esas cuatro cámaras, tres están situadas dentro del coche, y solo una de ellas fuera del mismo. Cada cámara está centrada en grabar una parte crítica y específica de la conducción: la carretera, el conductor, los pedales y los controles (palanca de cambios, volante, limpiaparabrisas, etc.)

Cada una de las cámaras graba un video, el cual es guardado en un ordenador situado en una de las salas de control, dentro del Laboratorio de Análise de Tráfego de FEUP. También es guardado en ese ordenador un archivo de texto. Este archivo contiene información sobre la experiencia de conducción del usuario, como las coordenadas del coche, la velocidad del mismo, el tiempo, etc.

El trabajo de este proyecto surge con el objetivo de mejorar la forma en la cual los profesionales analizan e interpretan los datos recogidos en una experiencia de conducción en el DriS. Para ese propósito, se ha creado un sistema de video-monitorización que consiste en una aplicación de video la cual permite cargar y reproducir cuatro videos en simultáneo. También consigue leer el archivo de texto que contiene los datos recogidos en la experiencia. Ambos (videos y texto) deben estar

sincronizados temporalmente, permitiendo al usuario mediante el uso de un cursor mover los archivos (video y texto) hacia atrás o hacia delante. Al ser un reproductor de vídeo básico, contiene algunos botones que controlan el estado del video (play, stop, pause), permitiendo a los profesionales analizar con detalle los cuatro vídeos y los datos.

Aprovechando los nuevos progresos en software de desarrollo, la aplicación se ha hecho en C++ usando la biblioteca Qt y su ambiente de desarrollo integrado Qt Creator, el cual hizo más fácil la implementación.

Al final de esta memoria (Capítulo 4) se adjunta un manual de usuario, que tiene como objetivo explicar y ayudar a los profesionales usar y manejar la aplicación.

Resumo

Nos últimos anos, o ISEP em colaboração com a FEUP e outras Universidades, criou um simulador realista de condução chamado DRIS, que tem como objectivo ajudar em trabalhos de investigação de diferentes áreas, como engenharia civil, computação gráfica, psicologia, educação, etc.

O resultado deste trabalho pretende ajudar os profissionais a analisarem os dados recolhidos em cada experiência de condução, a fim de permitir o estudo das reações do motorista em diferentes obstáculos durante um percurso.

O simulador DRIS é constituído por uma tela branca, onde os ambientes de simulação são projetados; um carro real, onde é feita a experiência de condução e quatro câmaras colocadas no carro. Destas quatro câmaras, três estão dentro do carro e uma fora do carro. Cada câmara está focada estrategicamente, em partes críticas da condução: a estrada, o motorista, os pedais e os controles (mudança de marcha, volante, os comandos do limpador, etc).

Cada uma das câmaras grava um vídeo, que é guardado em um computador colocado em uma das salas de controlo, dentro do Laboratório de Análise de Tráfego na FEUP. Além disso, um arquivo de texto é guardado no mesmo computador. Este arquivo de texto contém algumas informações sobre a experiência do motorista, como as coordenadas do carro, a velocidade do carro, o tempo, etc

O trabalho desta Tese surge com a finalidade de melhorar a forma de os profissionais analisar e interpretar os dados recolhidos a partir de uma experiência de condução no DRIS. Para o efeito, foi criado um sistema de vídeo-monitorização, que consiste em uma aplicação de vídeo, que permite a visualização de quatro vídeos simultaneamente, e ler um arquivo de texto, que contém todos os dados recolhidos na experiência. Ambos (vídeo e texto) têm de estar sincronizados com o mesmo tempo de forma a permitir ao utilizador, navegar backward e forward com a ajuda de um cursor. Além disso, como qualquer reprodutor de vídeo básico, contém alguns botões para

controlar o status do vídeo (Play, Stop, Pause) e permiti que os profissionais analisem com detalhe os dados dos quatro vídeos.

Aproveitando os avanços no desenvolvimento de software, a aplicação foi feita em C++ usando a biblioteca Qt, em ambiente de desenvolvimento integrado do Qt Creator, o que tornou mais fácil a implementação.

No fim deste relatório (capítulo 4) é anexado um manual do usuário, a fim de explicar e ajudar os profissionais a usar a aplicação.

Index

Image Index

Acronyms

DriS - Driving Simulator

FEUP - Faculdade de Engenharia da Universidade do Porto

GUI – Graphic User Interface

HD – High Definition

IDE – Integrated Development Environment

ISEP - Instituto Superior de Engenharia do Porto

I/O – Input Output

NADS - National Advanced Driving Simulator

NASA - National Aeronautics and Space Administration

OS – Operative System

SGI – Silicon Graphics Inc.

TDTS - Truck Driver Training Simulator

TRL - Transport Research Laboratories

VESTR - Virtual Environment for Surface Transportation Research

VTI - Swedish National Road and Transport Research Institute

WDF - World Description Files

WYSIWYG – What You See Is What You Get

1 Introduction

Driving simulation is a type of motion simulation that is currently undergoing a big change due to the enormous progress and improvements on this field in the last few decades.

In this thesis, the driving simulator that was used to develop the video-monitorization system was the DriS simulator, that will be presented in the following lines.

1.1. Introduction to DriS

DriS is a driving simulator developed by the FEUP in collaboration with ISEP in the end of the 90s and allocated in the Laboratório de Análise de Tráfego of FEUP. It was developed in order to help in the research of different topics, as road design, driving methods, driving education, or realism of video and images in a driving simulator.

To achieve this purpose, was recreated a virtual simulation ambient, where with access to one real vehicle, the drivers can drive along a virtual road composed by

realistic images generated in a graphical station. Being in a real vehicle provide more realism to the experience, but as the vehicle is fixed in the ground and does not

make any movement, the driver cannot feel as he is in a real road. This feeling happens also due to the images are projected in a screen, so the perception of depth that the driver could have is missing at all.

In the experiences, the driver is placed in a completely dark room to not allow the light influence in the driving. Also, four cameras are recording with some level of detail the road, and the movements of the driver, the pedals and the commands (gearshift, steering wheel, etc.).

In the following sections, the DriS will be presented with more detail. Will be described their facilities, the architecture, the road specifications, the driving station, the image generation sub-system, the sound simulation system and the reports production.

1.1.1. Facilities

The space where DriS is placed concerns about 75 m^2 and it is divided in three rooms (Figure 1).

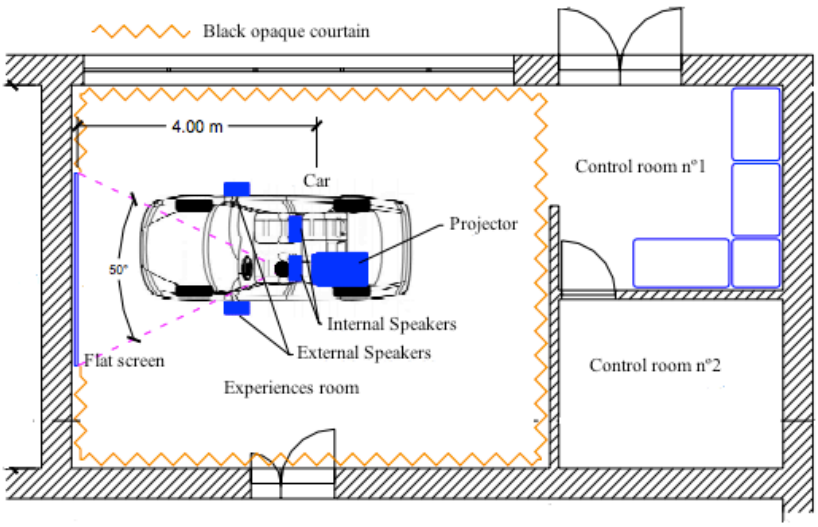


Figure 1: DriS room

The bigger room is the experiences room, and as its own name says, where the experiences are done. It contains a Volvo car, a BARCO video projector and a flat screen (3.40x2m), where the projections are done (Figure 2). This room provided a false roof and it is completely isolated from the outside lights.

Figure 2: DriS experiences room

The other two rooms have more or less the same size (Figure 1) and are control rooms, where the experiences are prepared and the simulator is developed. The control room nº1 has inside it all the computers associated to the simulation, allow the professionals monitoring and control all the experiences. The control room nº2 is used to save the files, the hardware and the peripherals that have not direct access.

1.1.2. Architecture

A driving station (consists on a real vehicle) and a graphic workstation compose the general architecture of DriS (Figure 2). The graphic workstation holds the scene database and performs the simulation and the image synthesis tasks.

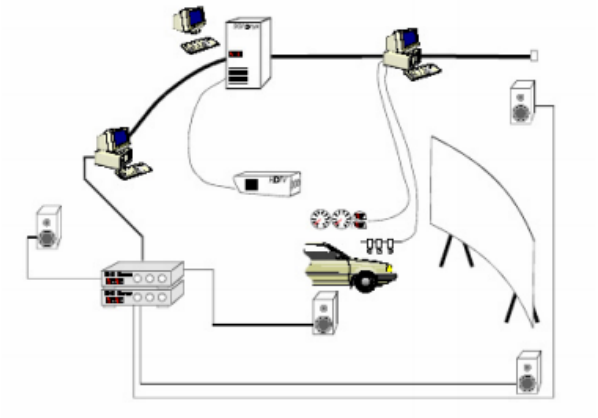


Figure 3: DriS general architecture [8]

The SGI Onyx Reality Engine 2 graphical workstation, runs in linux OS and it is composed by a commercial computer with the following specifications:

- Intel Pentium IV 3.0 GHz Prescot processor
- 1GB DDRAM
- Nvidia GeForce FX 5600 xt graphic card
- ATA 160GB internal hard disk

The same workstation is connected to the sound PC and to the cockpit PC by local area network. At the same time, a Switch D-link 1010G HUB interconnects all the systems.

DriS was developed using Genes, a virtual environment simulation toolkit. This tool is made on top of Performer [9], a well-known visual simulation software from SGI. Genes allows real-time image and sound generation, together with some movement and behaviour controllers.

DriS also includes autonomous vehicles to simulate traffic and provides more realism to the experiences in the driving simulator. The implementation of autonomous vehicles consists on an integrated model, which allows these vehicles to work on three different modes:

1. Being absolutely autonomous
2. Following a route imposed by an external entity
3. Having an intermediate behaviour between the 2 previous modes.

With these three modes, is achieved that the autonomous level of each simulated vehicle could oscillate between the absolute autonomy and the total dependency.

1.1.3. Road specifications

To create and specify the different simulation environments, WDF file texts are used. These files allow the creation of objects, definition of roads and specify movements and behaviours. At the same time, allow some operation with the objects: grouping them hierarchal, associate video cameras to the objects or apply geometric transformations (translations and rotations) to them.

Different types of definitions compose the simulating world. These definitions are: visual objects, animated objects, objects with some specific behaviour and the road.

1.1.4. Driving station

The driving station (Figure 2) is composed by a real vehicle (Volvo 440 turbo), which contains inside it a set of cameras and sensors to monitor the actions made by the driver during an experience.

The steering wheel and the original pedals of the vehicle were removed, and replaced by a specific console, developed exclusively for DriS. That console, presents one steering wheel, the two pedals (accelerator and brake) and some other buttons. The console has some sensors to read the pedals positions and steering wheel orientations, as well as direction changes, ignition key and horn indicators. This console is connected with the central computer to receive the signals collected by the different sensors.

Inside the vehicle are placed three cameras to record with a high-level of detail the movements of the driver, the pedals and the controls (gearshift, steering wheel, etc.). Also, another camera, which records the road, is placed outside of the vehicle, but in the front part of the cart.

1.1.5. Image projection system

The image generated by the principal simulation system is sending to the Barco Data 801s projector. This HD projector, with three colour projection systems independents (RGB), allow a focus calibration independent and balanced, enabling

1280x1024 pixel resolution images with an image update of 20 images per second, and a frame rate of 60 frames per second.

The image is generated in the principal simulation system through a graphic board GeForce FX 5600 xt, with 128bit Nvidia processor and 256MB of intern memory. Then, is sent to the projector through a Barco universal VGA interface, and projected in a white rectangle (3.40x2.00m) flat screen.

1.1.6. Sound simulation system

A sound system provides to the driving simulation more realism, once that the real driving is "contaminated" all the time with sounds of infinite different sources. For this, the DriS adds a sound simulation system to its architecture. This system is composed by: a computer to generate and simulate the sound, two audio amplifiers and four loudspeakers.

The sound system simulation is implemented in one commercial computer with two soundboards Sound Blaster AWE32 from Creative Labs, allowing the sound reproduction in four different channels. Three of these channels are intended for surround sound reproduction, while the other channel is associated to motor rotation sound.

The amplifiers are placed inside the control room, allowing the professionals to change the volume depending on each specific situation. In order to reproduce the most the sound heard by the driver in a real situation, the loudspeakers are distributed by the room and inside the vehicle: two speakers are placed on the side of the front wheels, while the other two speakers are placed inside the vehicle, behind the front seats.

1.1.7. Reports production

One of the big advantages of simulated driving versus the real driving is how easy can be the monitoring, through reports production. The DriS allow the registration of all the variations involved in the simulation process of the driving vehicle as all the

other involved vehicles. The data is recorded in real time to a binary file, being converted to a text file after the experience finish.

1.1.8. Goals and Studies

As we said before, DriS is being developed by a large team of researches in different areas, as computer systems, image analysis, computer graphics, civil engineering and psychology.

The main goals of this project is studying the driver's behaviour and road analyses under conditions that are difficult or almost impossible to reproduce: risk situations, accident, new roads, new managing systems, new road signs and new vehicle models.

To study the goals presented above, some studies and experiences were done. Follows, is presented a list of some different studies done with DriS.

- Vehicle's motion detection with current self motion with three kinds of road pavement: concrete pavement, bituminous pavement and bituminous pavement with chromatic bands.

- Evaluation of the influence of external publicity panels in driving performance.

- Evaluation of driving education methods.

1.2. Context

This thesis arises from the professional's need of study the video results and the reports of the experiences. With a video-monitorization system, which can be a video application that allows reproduce and analyse all videos (four at the moment) and the report (text file) generated in each experience, could be more easy to understand and make conclusions of the driving experiences behaviour.

1.3. Thesis Goals

The main goals of this research thesis are:

- Make a study to determine which programming language or library could be the best to develop a video application.
- Develop a video-monitorization system able to open a directory with four video files and one text file and reproduce them at the same time. This system should allow the users to navigate through the videos, using a time slider.
- Make a user tutorial, to explain the researches which is the best way to use this application.

1.4. Outline of the Thesis

This thesis is organized as follows:

Chapter1 Presents the DriS simulator, its architecture, main goals and some studies done with it. The main context, main goals and timeline of this thesis are also presented in this Chapter.

Chapter 2 introduces the basic concepts of driving simulation and some aspects of their history as well as some examples of the most advanced driving simulators around the world.

Chapter 3 presents and explains the two more important software tools that were consider to implement the main objective of the thesis. Also this chapter contains the advantages and disadvantages of each tool and a conclusion in which is explained the tool that was chosen to develop the video-monitorization system.

Chapter 4 explains some experiences with the software tools mentioned in *Chapter 3*, justifying which tool will be used in the implementation of the thesis solution. Also, is presented in this chapter the whole structure of the application implemented.

Chapter 5 includes the conclusion of all the work that was done and some ideas and improvements that could be implemented in the future.

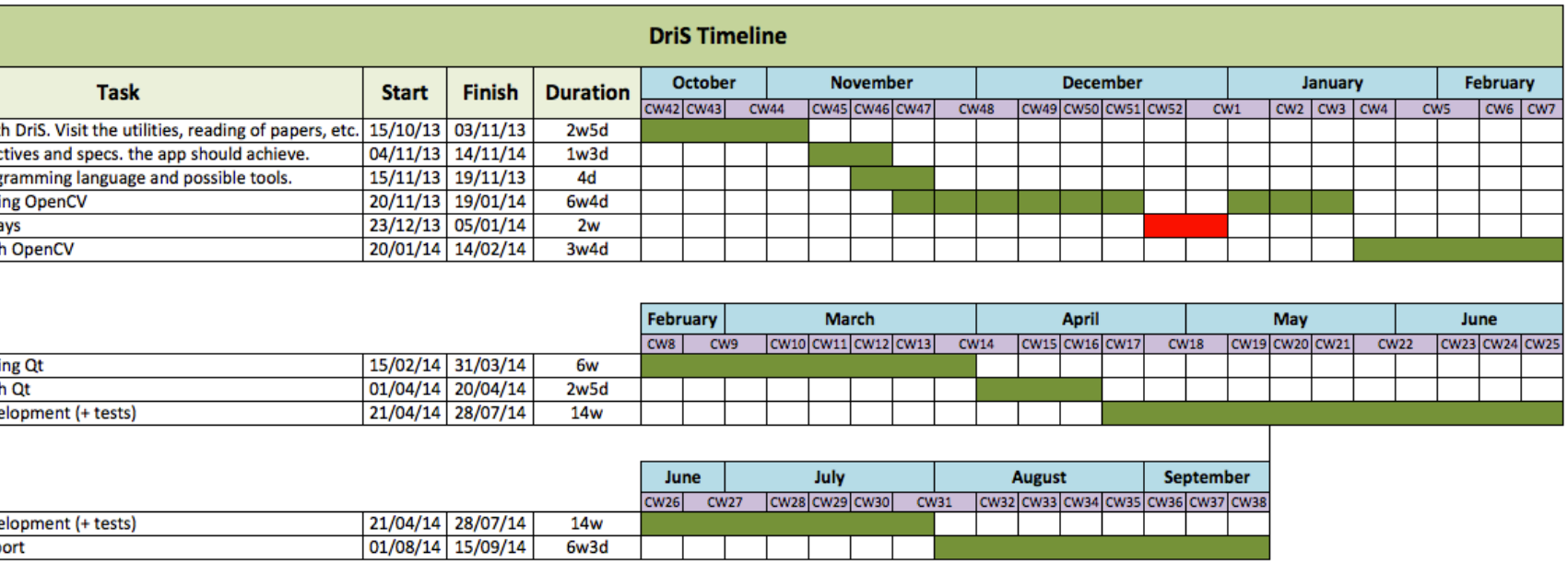
1.5. Timeline

Being the main objective of this thesis the development of a video-monitorization system for a realistic video simulator was needed to divide the work in small tasks in order to complain the objectives proposed and make a good application.

In the next page, is presented a timeline with all the tasks that were development and its duration, start and finish dates. The timeline is presented in weeks, due to the big duration that were needed to complete the objectives of this thesis.

DriS Timeline

Task	Start	Finish	Duration
First contact with DriS. Visit the utilities, reading of papers, etc.	15/10/13	03/11/13	2w5d
Define the objectives and specs. the app should achieve.	04/11/13	14/11/14	1w3d
Choose the programming language and possible tools.	15/11/13	19/11/13	4d
Study and learning OpenCV	20/11/13	19/01/14	6w4d
Christmas holidays	23/12/13	05/01/14	2w
Experiences with OpenCV	20/01/14	14/02/14	3w4d
Study and learning Qt	15/02/14	31/03/14	6w
Experiences with Qt	01/04/14	20/04/14	2w5d
Application development (+ tests)	21/04/14	28/07/14	14w
Application development (+ tests)	21/04/14	28/07/14	14w
Writing final report	01/08/14	15/09/14	6w3d



2 State of art

The first driving simulator system appears in the early 70s. The main cars constructor companies in the USA take advantage of the graphic and computing developments made by NASA in the space career, as well as by the technology developed by the US air force in the military first flight simulators. Using these developments, the car companies built their own driving simulators, first in USA and then in Europe.

Nowadays, there are two mainly lines of development about driving simulators:

- Driving simulators for commercial purposes
- Driving simulators for research and development functions

The architecture of both types of simulators depends on the cost restrictions and the intended application. The most commonly architecture for a driving simulator is based upon a real car in front of a fixed screen where a virtual world is projected (Figure 5).

Figure 4: Southampton University's simulator

Other common architecture is the one that introduces a special platform in the scene. This device usually consists in a piece of vehicle cabin controlled by some electro motors that allow the cabin motion in the three spatial dimensions. In this model, the screen can be placed inside the cabin, and it is composed by several monitors (Figure 6).

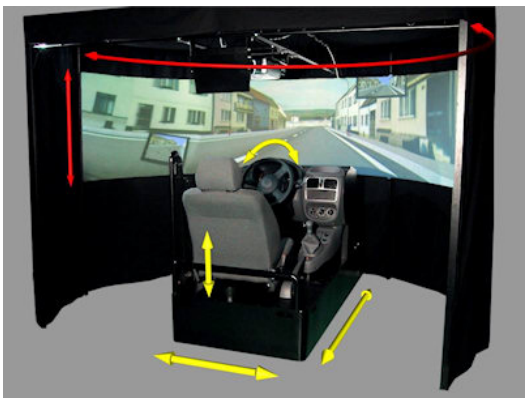


Figure 5: EDAS Driving Simulator

The third type of architecture, and the most expensive, includes a real vehicle cabin inside a mobile platform (Figure 7).

Figure 6: Toyota Driving Simulator

Besides these three main types of driving simulators seems very different (and they are), there is some common components to consider in all driving simulators. In the following section will be described them.

2.1 Common components in all driving simulators

Usually, the five principal components in a driving simulator system are: vehicle model, visuals, motion, traffic and scenarios & instructions.

- **Vehicle model**: At the beginning, driving simulators used simple, special purpose vehicle models, but nowadays vehicle models approach perfection and simulators can use the same models, software tools and programming languages that car engineers use in the design of their vehicles. This, in a commercial context allows studying the comfort, design and security of a vehicle even in the design phase.

- **Visuals:** Driving is a visual dominant task, and present a good visual experience is an important issue in the driving simulators. The fist driving simulators used a camera monitor system hovering a conveyor belt or a maquette. This allowed that the texture of the visual environments recorded on the video was presented with a low resolution as you can see in the figure 8.

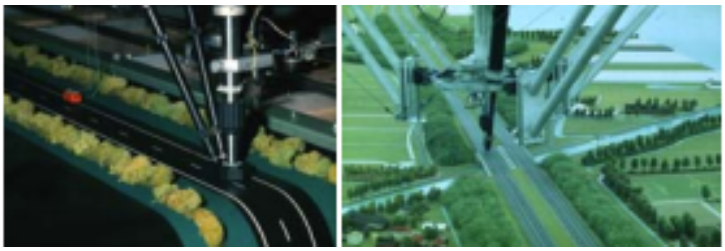


Figure 7: First visual environments for driving simulation

With the improvements in the computer graphics, the visuals in the driving simulators improved as well, making realistic the driving experience to the user, by adding to the scenario natural and detailed textures, people and objects (Figure 9).

Figure 8: Modern visual environment for driving simulation

- **Motion:** While visual perception plays a very important role in a lot of driving tasks, we should consider as well the vestibular perception, which is very important in the control of the vehicle. Cars move strongly and usually exhibits large linear accelerations while accelerating, breaking or in curves. This movements are very complex of reproduce, so even the most advanced driving simulators have difficulties in being realistic in this kind of simulations. The NADS [1] or Toyota [2] driving simulators (described in the following section) could be some good examples of simulators, which reproduce in very realistic way the motion of a vehicle.

- **Traffic model:** As we do not drive in completely isolation on the road, the traffic models are an important part of the driver experience. It exists a lot of traffic models, like a car moving over a track, an intersection or a roundabout for example. To be acceptable, the traffic model has to reproduce driving behaviour at the control, the manoeuvring and the strategic level.

- **Scenarios & instructions:** One of the big advantages in the driving simulators is that you can choose the scenario, the traffic and the obstacles that you want to appear during an experience. This allows to study and to train the driver amongst the different obstacles that he can have during a real drive.

2.2. Examples of real driving simulators

The following section presents some of the most important diving simulators in the world [3], organized geographically:

USA simulators:

- **NADS-1**: Developed at University of Iowa is one of the most advanced ground vehicle simulator at the moment. It consists of a large 24-foot-diameter dome in which entire cars, or cabins of other vehicles can be mounted (Figures 10 and 11).

Figure 9: NADS-1 driving simulator

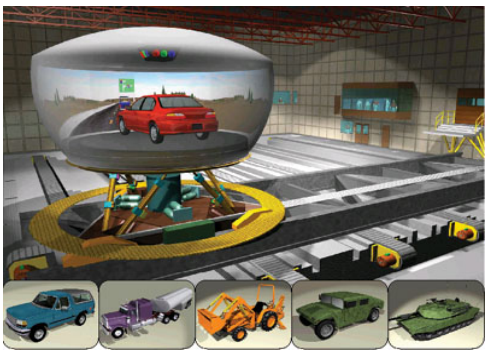


Figure 10: Inside of NADS

At the same time, the motion subsystem, in which the dome is mounted, provides horizontal and longitudinal travel and rotation. This, allow the users feel acceleration, breaking and steering cues as if the drivers were driving a real car. However, as showed in the figure 12, the display system is in 2D, so the depth perception of the driver is very limited, affecting to his reactions as distance, speed and time are key issues while driving.

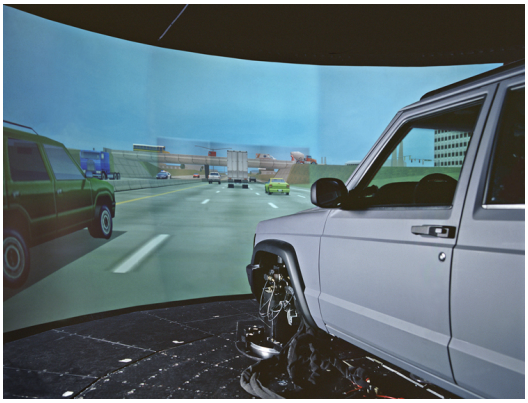


Figure 11: NADS-1 visual system

- **DES**: This simulator born under the HumanFIRST program at the University of Minnesota [4]. The most important difference between this simulator and the model before is that this one has not any movement in the cabin, in spite of the fact that is formed by a real car too. The only movement that the DES provides is the car body vibration and a three-axis electric motion system, which create Z-axis motion with a limited range of movement. Other difference consists in that in DES, the visual scene is projected to a high-resolution five channel210-degree forward field of view with rear and side mirror views are provided by a rear screen and LCD panels (Figure 13).

Figure 12: DES Driving Simulator

European simulators:

- **Sweden**: One of the most important simulators developed in Europe was developed in VTI [5] and is one of the first driving simulators to allow different degrees of freedom in the simulator. It's very similar to American NADS.

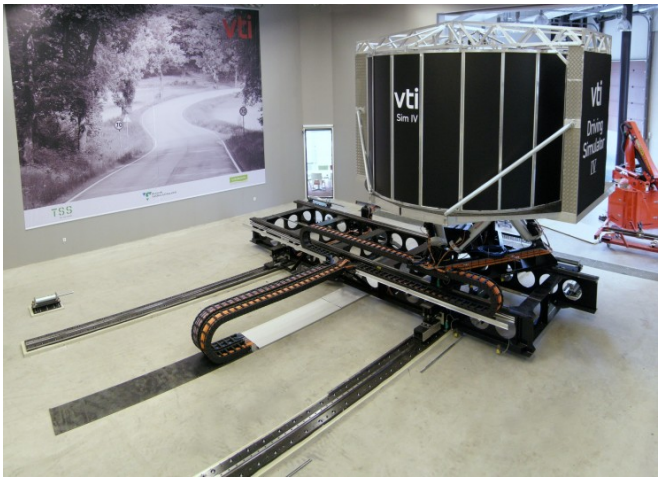


Figure 13: VTI Driving Simulator

- **UK**: Other important driving simulator could be found at the TRL [6]. This specific one offers a variety of validated test scenarios and works in studies about drugs and energy drinks effects in conduction, motorway road works and speed management systems among other applications. Another interesting

development in driving simulators can be the TDTS, also in UK, and consists in a dynamic truck cabin plus a wide display screen (Figure 15).

Figure 14: UK Truck Driver Training Simulator

Other countries simulators:

- **Japan**: In Japan, in 2007 Toyota developed its own driving simulator (Figure 7), very similar with the NADS-1. The simulator consists of a real car positioned on a mechanical platform inside a 7.1meter dome. A tilt mechanism, vibration apparatus and other devices manipulate the dome as the driver operates the vehicle. The dome, as happens in NADS, acts as a giant 360-degree video scree, simulating a real driving experience that includes a sense of speed, acceleration a riding comfort with sound effects thrown in to complete the virtual driving experience.

3 Implementation tools

After presenting the DriS simulator, is easy to understand one of the researchers needs when they have to compare the videos and the reports made by DriS of the same experience. It would be easy for them to have a tool where all the videos and the file text are presented at the same time, and for this reason was decided to create a video application which provide them an easy way to study the experiences.

Before develop the video application, some implementation tools were studied, being stronger the idea to analyse the OpenCV [10] and QT libraries [11], due to both are open source and cross-platform, two good features to have in a research application.

3.1. OpenCV

OpenCV (Open Source Computer Vision) is a well-known computer vision library, developed by Intel in 1999. It works under BSD (Berkeley Software Distribution) license, so is free for academic and commercial use. At the same time it is cross-platform

(works in Linux, Windows, Mac OS and Android) and has included the C++, C, Phyton, Java and Matlab interfaces.

OpenCV was built to provide a common infrastructure for computer vision applications and to accelerate the use of machine perception in the commercial products. At the same time is optimized for computational efficiency with main focus on real-time applications including approximately 2500 different computer vision algorithms. These algorithms can be used for multiple tasks, as face detection and recognition, objects identification, camera movements and moving objects tracking, extract 3D models of objects, etc.

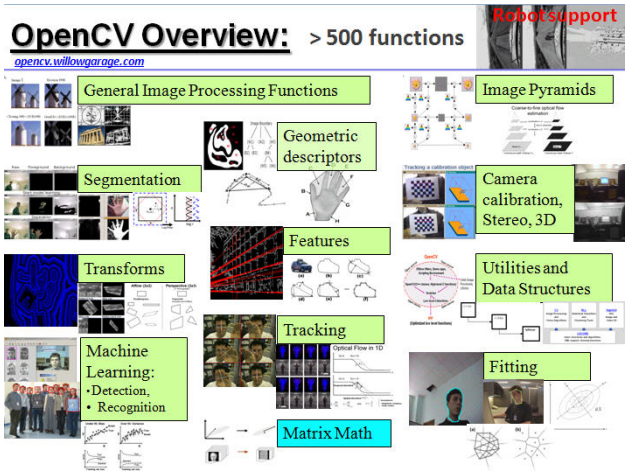


Figure 15: Some algorithms provided by OpenCV

This software has a modular structure, so the package includes several shared or static libraries. The most important of them are the following modules:

- **Core**: in this module, basic data structures and basic image processing functions that can be use by all other modules are defined.

- **Imgproc**: this module includes basic image processing algorithms as image filtering, histograms, image transformations, colour space conversion, etc.

- **Highgui**: provides simple user interface capabilities, several image and video codecs, image and video capturing capabilities, manipulating image windows, handling track bars and mouse events, etc.

- **Video**: video analysis module, which includes motion estimation, background subtraction and object tracking algorithms.

- **objdetect**: its main function is the object detection and recognition (faces, eyes, etc.) of standard objects.

With all of these modules and more libraries that you can see at [12] can be easy to imagine the amount of good computer vision applications that OpenCV allows to develop; but as everything, this library has its good and bad things that are summarize in the following sections.

3.1.1. Advantages

- **Free**: as we said before, OpenCV is an open-source library working under BSD license, being easier to obtain than some other applications (Matlab for example).

- **Cross-platform and portability**: this library works in all computers, wherever the OS of the device, so could be easy to port applications from one OS to another.

- **Speed**: OpenCV is basically a library of functions written in C/C++, so it is closer to provide machine language code to the computer to get executed, getting more image processing done for the computers processing cycles. As a result of this, programs written in OpenCV are quite fast.

3.1.2 Disadvantages

- **Memory manager**: in OpenCV, every time that a piece of memory is allocated, is mandatory to release it again. Saying in other way, if this piece of memory is allocated in a loop the program will use a growing amount of memory until it crashes from no remaining memory.

- **Integrated Development environment**: OpenCv has not an integrated development environment (IDE), so for each OS the user is free to choose which IDE to use. Usually, in Windows the developers use Microsoft Visual Studio or NetBeans; in Linux, Eclipse or NetBeans as well; and in Mac OS Xcode.

- **Difficult to learn**: to use properly OpenCV and make a good application, the developer should read and learn a lot about OpenCV and should have a good knowledge of a high level language and image processing algorithms, making the learning curve steeper.

- **User Interface**: the possibilities that OpenCV offers in terms of user interface is very poor, so to make a good and friendly application is better to use other user interface frameworks.

3.1.3. Conclusion

OpenCV is a very complete and in constant development advanced computer vision library, that allows the users to develop optimized apps for real-time applications. It is almost written in C/C++ to make faster the implementation and it works in the most important OS: Windows, Linux and MacOS. Due to its complexity to learn the basics of OpenCV and due to has not got integrated a development environment, which make harder its installation, some users desists to use this library.

As the purpose of this thesis is making a video-monitorization system application using no real-time videos, and as a video application should have a very friendly user interface that OpenCV does not support, was decided (after doing a few experiences, explained in the next chapter) to not use OpenCV.

3.2. Qt

Qt is a full-developed GUI framework with tools designed to optimize the creation of applications and user interfaces for different platforms and operation systems, being a cross-platform framework easily to port from one OS to another (Figure 17).

Figure 16: Operative systems in which Qt works and can be ported

Qt framework provides to the developer: cross-platform C++ class libraries, choice of user interface approach (Declarative, C++, HTML5) and an integrated development environment (Qt Creator). The Qt Creator is an integrated development environment (editor + compiler + debugger) easy to use, efficient and free that allows the development of applications in different OS. This IDE, uses the C++ language, allows the visual and event-controlled programming, provides a syntax highlighting and code completion, is context sensitive help, etc.

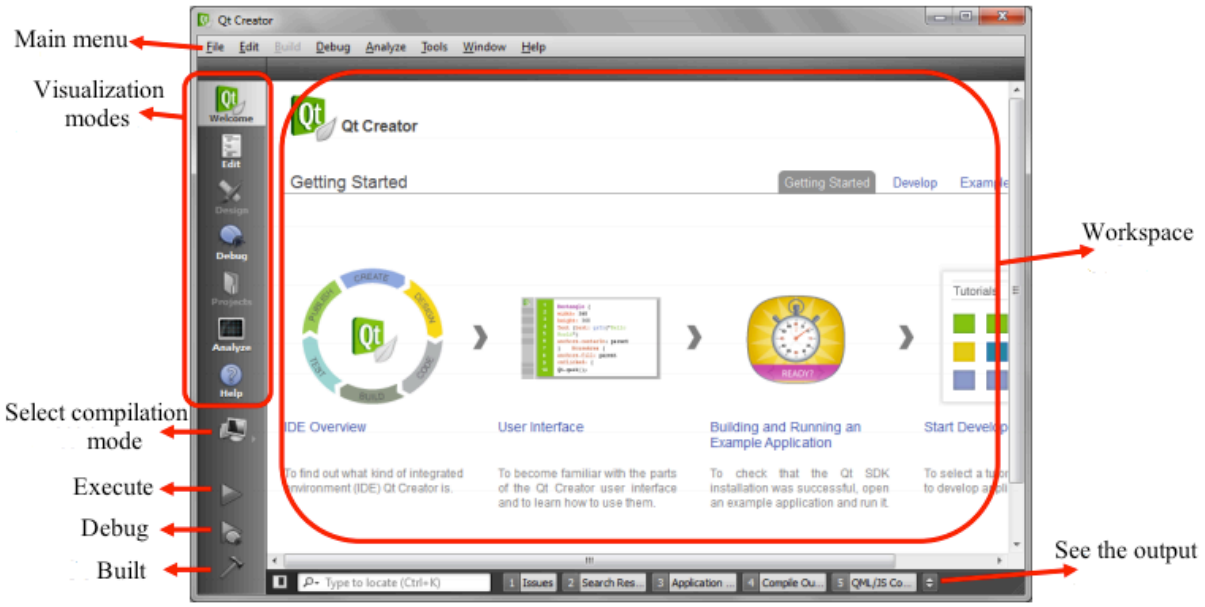


Figure 17: Qt Creator

Regarding the features listing above, can be consider that the two main features that Qt has is its visual programming and its event-control programming.

Visual programming

Qt allows the developer to be focused on design the user interface of the application, the visual elements distribution (widgets) the interaction between them, the different kinds of existing windows, etc. Basically, allows the developer designs the graphic layout of the application following the rule: "What you see is what you get" (WYSIWYG).

One important part of this visual programming is composed by the widgets (as we said before). Widgets are primary elements for creating user interfaces in Qt. They can display data and user information, receive user input and provide a container for other widgets that should be grouped together. (Figure 19) A widget that is not embedded in a parent widget is called a window.

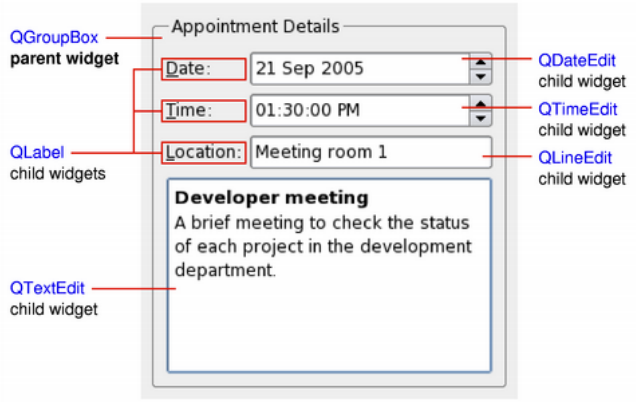


Figure 18: Example of a QWidget object

Event-controlled programming

Qt includes a different way than others GUI frameworks to make the communication between objects. It uses signals and slots. A signal is emitted when a particular event occurs, and a slot is a particular function, called in response to a particular signal. In that way, the developer can assign as many signals as he want to the same slot, and also can assign the same signal to several slots (Figure 20).

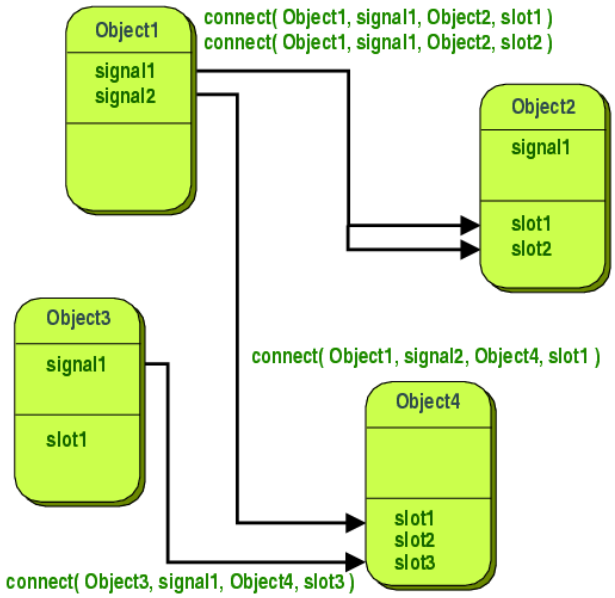


Figure 19: Communication flow between Objects using signals and slots

The signals and slots mechanism is type safe, which means that the signature of signals and related slots should match, although the slot may have a shorter signature than the signal it receives due to it can ignore extra arguments.

Objects emit signals when they change their state in a way that may be interested to other objects. Actually, it does not care whether anything is receiving the signals that the object emits, making this true information encapsulation and ensuring that the object can be used as a software component.

Slots are normal C++ functions used for receiving signals that do not know if they have any signals connected to them. When different slots are connected by the same signal, the slots will be executed one after the other, in the order that they were connected, when the signal is emitted.

Some examples of programs created with Qt libraries are: Adobe Photoshop Album, Google Earth, KDE, Opera, Skype, VLC, Windows Media Player, and a lot of more. Qt easy usability and good results both in user interface and in performance make that

each time more developers and companies use the Qt libraries to develop their applications.

3.2.1 Advantages

- **Free**: although Qt has a paid commercial framework, that offers more possibilities to developers, allows a free download and use of most of their libraries, as said before.

- **Cross-platform**: it is cross-platform, allowing the development of the app in any OS. Also, Qt uses the system's resources to draw windows, controls, etc. to provide a native look to the applications.

- **Easy to port**: besides been cross-platform, Qt allows to port in an easy way the applications from one OS to another.

- **Integrated development environment**: Qt has integrated QtCreator, its integrated development environment. That environment has some advantages, listed in the section above, that make easy the development of the apps using Qt. Also the Qt Creator has integrated a help mode with lot of documentation about different classes, examples, etc. in order to help the users in an easy and fast way.

- **Open source**: Qt is open source and is developed by the Qt Group at Nokia, so a very important enterprise maintain it with support from the community and ensuring its evolution.

3.2.2 Disadvantages

- **Difficult to learn all the possibilities**: Qt offers a lot of possibilities and functions. The number is so high that is complicated to know, remember and understand all of them without external help.

- **Database drivers**: there are some problems with installing database drivers. Is not really Qt's fault, but it would be better if there were a way to do it in a simpler way.

- **Compilation**: Qt requires a lot of disk space for compilation, mostly due to WebKit tool. That could be a big problem if the computer used for develop the app has the hard-disk full.

3.2.3 Conclusion

Qt library is an open-source and cross-platform user interface framework that provides an easy way to develop friendly user interface applications, helped by its integrated development environment QtCreator.

QtCreator provides a very good documentation of all the Qt classes, signals, slots, functions, etc. making easier and faster to the developer learn and uses Qt. Also has another good features, named above, as highlighted syntax and code completion.

The new way that Qt allows the communication between objects by signals and slots, also make easy to the developers the implementation of their applications. But this way of communication could be slightly slower than callbacks [13] (the way that others toolkits achieve this communication between objects), because emitting a signal that is connected to some slots if approximately ten times slower than calling the receivers directly. Although this way of communication is slower, the difference for real applications is insignificant.

As Qt provides a very good GUI designer to make user-friendly applications, an easy way to port the programs to other OS, a lot of multimedia libraries to reproduce, modify, create, etc. multimedia files and programs, an easy installation of the libraries and the QtCreator, we decided to proceed with this program to made the video-monitorization application for DriS (objective of this thesis).

4 Video-monitorization system

To fulfil the needs requested by DriS researches to analyse each one of the experiences made in the driving simulator, was decided to implement a video-monitorization system, which consist in a video application and a text reader.

Four video screens and a text reader compose the application. The objective of the video screens is reproduce the four videos recorded in each experience (the road, the driver, the pedals and the controls). The application allows reproducing each video in one screen, to see them at the same time.

Figure 20: DriS video application

The text reader objective, in the other side, is to display in the screen and read a text file that contains some important data of the experience as could be the time, the X, Y, Z position, etc.

All the video screens and the file data are time-synchronized and could move them forward and backward using a time slider. The application also includes an "Open Files" button to load the videos and the text file into the video-monitorization system. For this, all the files should be in the same folder, and the system will open all the files in the folder instantly, without having to load one by one the files.

4.1 Experiences with different software tools

Before start to implement the video-monitorization system, we take a time to decide in which programming language will be implemented the video-monitorization system. The language that has chosen was C++, due to its good performance in multimedia applications.

Then, were defined the layout and the functions that the video-monitorization system may have. Basically, it consists in an application that open and reproduce at the same time four videos and one file text recorded in driving experience inside the DriS. A

time slider handles this videos and the file text, allowing the user to move them forward or backward.

After, as explained in the chapter before, were considered two powerful tools to help us make the video-application: OpenCV and Qt (both written in C++). After some experiences explained below, and after comparing advantages and disadvantages of both of them, we choose to use the Qt library.

4.1.1 Experiences with OpenCV

At the first time, was considered to use OpenCV to implement the app, based in all the good features that it provides in computer vision application (see the chapter 2 to read about it).

The first step was went to the OpenCV web page (http://opencv.org/) and download the OpenCV version 2.4.8 for iOS, since the computer used for develop the application was a MacBook Pro. Was used this version of OpenCV for being the most recent when the implementation starts, although this version does not provide mayor changes that affects our implementation.

After this first step, were installed the OpenCV in order to works with Xcode. This was the most difficult task before programming, due to some problems that arises, and that will be explained after in this section. To install the OpenCV in an easy way to works with Xcode, first, is needed to install MacPorts [14], or Homebrew [15] or CMake [16]. The three of them are software package managers that simplify the task of compiling and installing open-source software on Mac OS. Depends on which of them the user chooses, the OpenCV installation way changes [17], [18]. At the beginning, was try to use MacPorts but when compiling a program, an error appears and were not allowed to compile the program. Then, was try with Homebrew obtaining the same result, so after some researches and reports read, the decision was to install CMake, and when try again to compile the file, there was no error and the program compilation was fine.

Once that OpenCV have worked fine with Xcode, was started to write some code to make the video application. These first lines of code are irrelevant for this paper, because the development using OpenCV was almost zero. One relevant thing that was found using OpenCV, was that the GUI using by it was very poor, and for the video-monitorization system was defined a very complete GUI as a must to have. Also, this system is not conceived to use real-time videos, which are one of the strengths of OpenCV.

4.1.2 Experiences with QT 5.2.1 & Qt Creator

As was decided not to use OpenCV, and the application that will be implemented must have a very good and friendly user interface, was decided to use Qt library with its integrated development environment, Qt Creator.

Was downloaded from the Qt project web page (http://qt-project.org/downloads) the Qt 5.2.1 and the Qt Creator 3.0.1. As the installer package has an installation helper, the install of these tools was very quick and easy. This version of the Qt has some new features than older versions does not have, and were important for the development of the app. Some of them are:

- **User interface**: in this version, the developers could develop applications with intuitive user interfaces for multiple targets, faster than before.

- **Multimedia**: the new library QtMultimedia, provides a rich sets of QML types and C++ classes to handle multimedia content. This module it is part of the set of essential modules with support on all major platforms.

- **Modularized Qt libraries**: now, the Qt libraries are split into domain specific libraries and modules, allowing to each application to choose which libraries it requires and only compile, use and deploy those.

- **Widgets**: the Qt Widgets are in this version separated into their own module, the Widgets module, which is part of the essential modules.

After installing the Qt tools (Qt Creator and Qt 5.2.1), was made a big research work to meet as many Qt modules and libraries as possible, in order to make easier and faster the implementation of the application. The main modules are the following:

- **QtCore**: core of non-graphical classes used by other modules. [19]
- **QtGui**: base classes for graphical user interface (GUI) components, including OpenGL. [20]
- **QtMultimedia**: classes for audio, video, radio and camera functionality. [21]
- **QtMultimediaWidgets**: widget-based classes for implementing multimedia functionality. [22]
- **QtWidgets**: classes to extend Qt GUI with C++ widgets. [23]
- **QtNetwork**: classes to make network programming easier and more portable. [24]
- **Qt QML**: Classes for QML and JavaScript. [25]
- **QtQuick**: framework for building highly dynamic applications with custom user interfaces. [26]
- **QtQuickControls**: reusable QtQuick based user interface controls to create desktop-style user interfaces. [27]
- **QtQuickDialogs**: types for creating and interacting with system dialogs from a QtQuick application. [28]
- **QtSQL**: classes for database integration using SQL. [29]
- **QtTest**: classes for unit testing Qt applications and libraries. [30]

For the development of the video-monitorization system, were only used some of these modules, as explained in the section below, and not all of them.

4.2 Video-Monitorization System development

Before going deep into the classes and the code that was developed to make the DriS application, will be explained the basis of how to make a Qt project, having as an example the DriS application.

To make a new app in Qt, the efficient and faster way to do it is using QtCreator, as explained before. In this development environment, a wizard helps the users step by step to make a new application. On the first step, is needed to choose which kind of application will be created.

Qt allows choosing between many types of templates, from desktop to Android, iOS and Blackberry, to develop the applications. The DriS application was thought to be a desktop one, having to choose between one of the following templates shown in the image below:

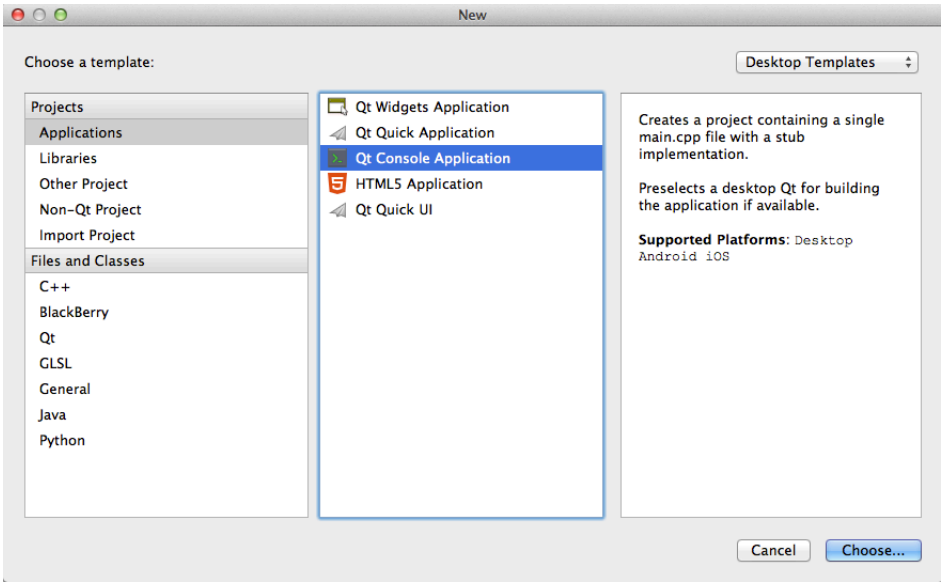


Figure 21: Qt Creator Desktop Templates

- **QtWidgets Application:** uses Qt Designer forms to design a Qt widget based user interface for the desktop and C++ as programming language.

- **Qt Quick Application:** creates an application that contains both QML and C++ code. The project includes a QDeclarativeView or a QQuickView. This type of project allows building the application and deploying it to desktop and mobile target platforms.
- **Qt Console Application:** uses a single main.cpp file. DriS application is of this type of application.
- **HTML5 Application:** uses a QtWebKit view to design the user interface and HTML5 and C++ code to implement the application logic.
- **Qt Quick UI:** uses a single QML file that contains the main view. This kind of projects could be reviewed in a preview tool and is not mandatory to build them. Also, is possible not to have the development environment installed in the computer to create and run this type of projects.

Once that the type of application is chosen, the wizard guide the users through few more steps where is defined the location for the project and specified settings for it. When all the steps are completed, QtCreator automatically generates the project with required headers, source files, user interface descriptions and project files, as defined by the wizard. In DriS application case, was created a *main.cpp* file and a *Dris.pro* file.

This *.pro* files stores the different classes and files that will be in the project, the libraries and modules that will use the project and the way in which the project will compile (the name of the project). The *DriS.pro*, for example has the following architecture:

```
QT          += widgets \
               multimedia \
               multimediawidgets \
               core \

TARGET = Dris
TEMPLATE = app


SOURCES += main.cpp\
           videowidget.cpp \
           player.cpp \
           controls.cpp

HEADERS  += videowidget.h \
            player.h \
            controls.h
```

Figure 22: DriS.pro file structure

- **QT section**: contains the modules that the whole application will use. With the function +=, modules are added. To remove them is possible to use the function -=, or directly remove it from the code. These modules could be added during the development of the application. DriS application only uses four modules: QtCore, QtWidgets, QtMultimedia and QtMultimediaWidgets.
- **TARGET**: where is defined the name of the application.
- **TEMPLATE**: defines the type of the application.
- **SOURCES**: all the *.cpp* classes that the application contains. The sources are added automatically to *.pro* file when created.
- **HEADERS**: all the *.h* headers that the application contains. As the sources, the headers are added automatically to *.pro* file when created.

In a QtConsole Application, at the beginning, only this *.pro* file and the *main.cpp* are created, and when the developer needs, the other classes are created. In DriS application, only *.cpp* classes were created. Create a *.cpp* class in Qt Creator is very simple and the same wizard than before (Figure 22) could help the developers, after following this flow in the toolbar: *File > New File or Project*. Choosing the C++ class option, will appear a C++ Class Wizard, that allows to create a C++ header and source

file for the new class, with only specify the class name, base name (optional), base class and header and source files for it.

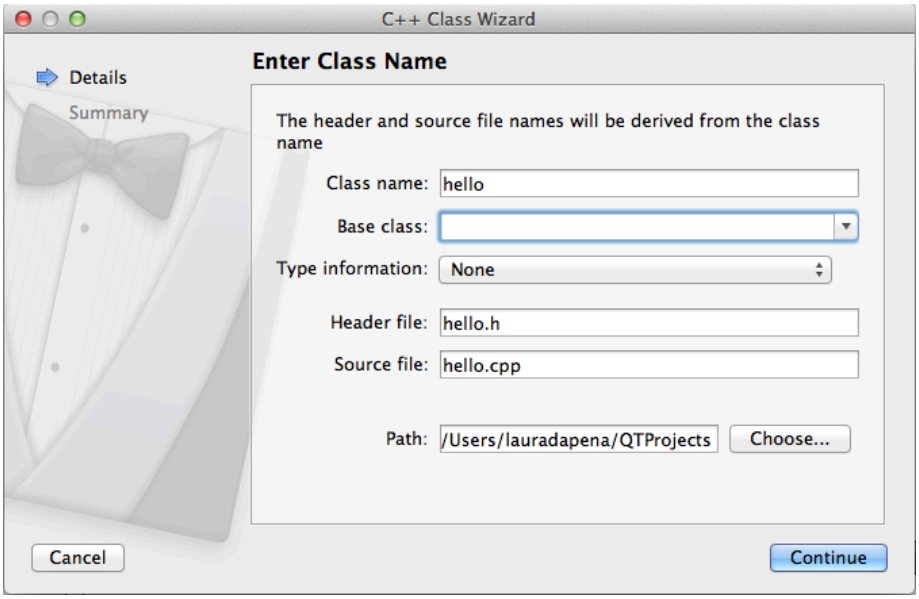


Figure 23: C++ Class Wizard

After explaining how to add classes to a project in Qt Creator, it can particularize this process for DriS application.

One main class and three other classes, with its respective headers, compose DriS application. These classes are: *VideoWidget* class, *Controls* class and *Player* class. All of them are C++ classes, as expected, and its functionality and structure will be explained just below.

4.2.1 VideoWidget class

This class creates a QVideoWidget object. The QVideoWidget class, provide a Widget, which presents video produced by a media object, and inherits the QWidget and QMediaBindableInterface classes. It belongs to the QtMultimedia module, new in 5.0 Qt versions.

Attaching a QVideoWidget to a QMediaObject, allows it to display the video of image output of that media object. A QVideoWidget is attached to media object by passing a pointer to the QMediaObject in its constructor, and detached by destroying

the QVideoWidget. Only a single display output can be attached to a media object at one time.

Once that the QVideoWidget object is created, was used the class QPalette, to create a palette object which set to black the window that is entire occupied by the QVideoWidget object. The result could be seen below:

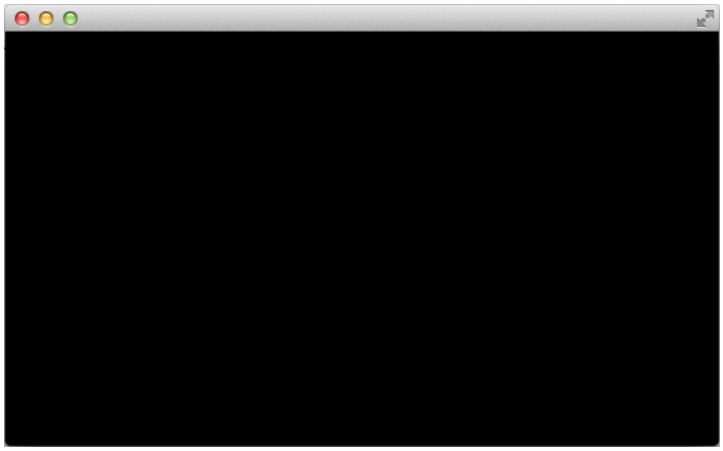


Figure 24: QVideoWidget object

Basically, a black screen is appeared in the screen. For the purpose of DriS, was pretended to have four of this black screens, each one reproducing one different video, but this process will be explained better in the Player class section.

4.2.2 Controls class

As this thesis consists in create a video application to monitor the experiences made by the drivers in a driving simulator, and usually the video applications has some common buttons to hold the video reproduce, was made a class that keeps all the video application controls together.

This class generates a QWidget object, which will include three QAbstractButton buttons: play, stop and mute. For this version of DriS application were only consider these three buttons. The play and stop buttons are essential in any basic video application and the mute button was added to allow the users of the application not to hear the sound of the four videos at the same time.

The QAbstractButton class is the base class of button widgets, providing common functionality to buttons. As this class implements an abstract button, subclasses of it, handle user actions and specify the style of the button. QAbstract Button allows support for checkable buttons (QRadioButton and QCheckBox) and for push buttons (QPushButton and QToolButton).

Our three buttons are push buttons, from QToolButton class. This kind of buttons provides quick-access to specific commands or options, and as opposed to a normal command button, a tool button usually shows an icon instead shows a text label.

Figure 25: Different types of buttons

The buttons will be positioned in the wigdget, creating a QBoxLayout layout. QBoxLayout class lines up child widgets horizontally (our case, using QHBoxLayout) or vertically (using QVBoxLayout).

As this buttons are made to control a QMediaPlayer object (explained after in the Player class), the QMediaPlayer should be part of the Controls class.

The functionality that QMediaPlayer has in this class is to allow inherit some signals and slots to allow the communication between objects. Also, was used its function *state()*, to know the current state of the QMediaPlayer object and implements the buttons functionality depends on it. Only three states are defined for a media player:

- **Stop**: The media player is not playing content. Playback will begin from the start of the current track.
- **Play**: The media player is currently playing content.
- **Pause**: The media player has paused the reproduction of the track. Playback of the current track will resume from the position the player was paused at.

The QMediaPlayer, also provides some slots, signals and functions to control the mute button.

To control the behaviour of this three buttons, some methods were implemented and signals and slots were connected. Just below, could be found the explanation of each button's behaviour.

- **Play button (play mode)**: This is the play button default mode. Clicking in this button, the media player will start the reproduction of the media, and the play icon will change to the pause icon as well as the stop icon will be enabled. Also, if the media player is stopped or paused, after clicking in play button the media will be reproduced again. Summarizing, only in two situations the play button (play mode) can be clicked: to start the reproduction (new one or stopped one) or to continue the reproduction after paused.

- **Play button (pause mode)**: When the media player is currently playing the track, the play button is on pause mode. When clicked this button, the reproduction of the track will stop and the pause icon will change to player icon as well as the stop icon will be enabled. This button is only clickable when the media player is currently reproducing a track.

- **Stop button**: The stop button only is enabled when the media player is reproducing a track or if it is paused. Otherwise, the stop button stays unable . Clicking in this button, the signal *stop()* is triggered, stopping the playback of the track and forcing the reproduction from the start of the current track. Always, when this button is clicked, automatically the player button changes to player mode , regardless of its previous state (play or pause).

- **Mute button**: When mute button is clicked, the sound of the reproduction stopped. Also, the icon changes from this icon when the sound is enabled to this icon when the sound is unabled. The default state for this button is to be unable (the sound is enable).

All of this buttons do not make sense without a media player object, which will be described in the section below.

4.2.3 Player class

The player class is, with the main class, the most important class for this application. In this class, all the needed objects are created and is defined their behaviour too, at the same time that the final layout is composed. The headers for *Videowidget* class and *Controls* class are included in the *Player* class, in order to use video widgets objects and the buttons defined in the *Controls* class to construct the application.

The *Player* class will create a QWidget object, which allows the reproduction of DriS videos and the file text with the results of the experiences, at the same time. For that, the layout of the widget was divided in 5 main parts: four video widgets objects in which reproduce the videos and one QTextEdit object in which shows the text file.

At the first point, will be explained how was created the players to reproduce the videos. Then, how these videos were open, loaded, reproduced and synchronized and after that, the missing part related with the reading and synchronization of the text file.

Video players

Were created four QMediaPlayer objects, and four QVideoWidget objects to achieve the reproduction of the four DriS videos.

The QMediaPlayer class inherits the QMediaObject class and allows the playing of a media source. This class is a high-level media playback class that can be used to reproduce different content as songs, movies, etc. The content to playback is specified as a QMediaContent object, which can be thought of as a main or canonical URL with additional information attached. In this project, we also use the QMediaPlayer with QVideoWidgets with the purpose of video rendering.

As just said, the created media player objects (*player1, player2, player3 and player4*) and the created video widget objects (*videoWidget1, videoWidget2, videoWidget3 and videoWidget4*) are completely related. Was used the QMediaPlayer function *setVideoOutpu(QVideoWidget *output)* to attach each video widget output to the related media player (player1 -> videoWidget1, player2 -> videoWidget2, and so on). If the media player has already a video output attached, will be replaced with a new one.

Once that the media players and the video widgets were connected, was created a layout to present them well organized in the player. For this, were defined two horizontal layouts (QHBoxLayout) and a vertical one (QVBoxLayout). In the horizontal ones, were presented in one layout the videoWidget1 and the videoWidget2, and in the other, the videoWidget3 and videoWidget4. The vertical layout reorganizes these two horizontal layouts, being the videoWidget1 and videoWidget2 in the top part of the screen. Follows is shown, an image of the layout at this part of the development.

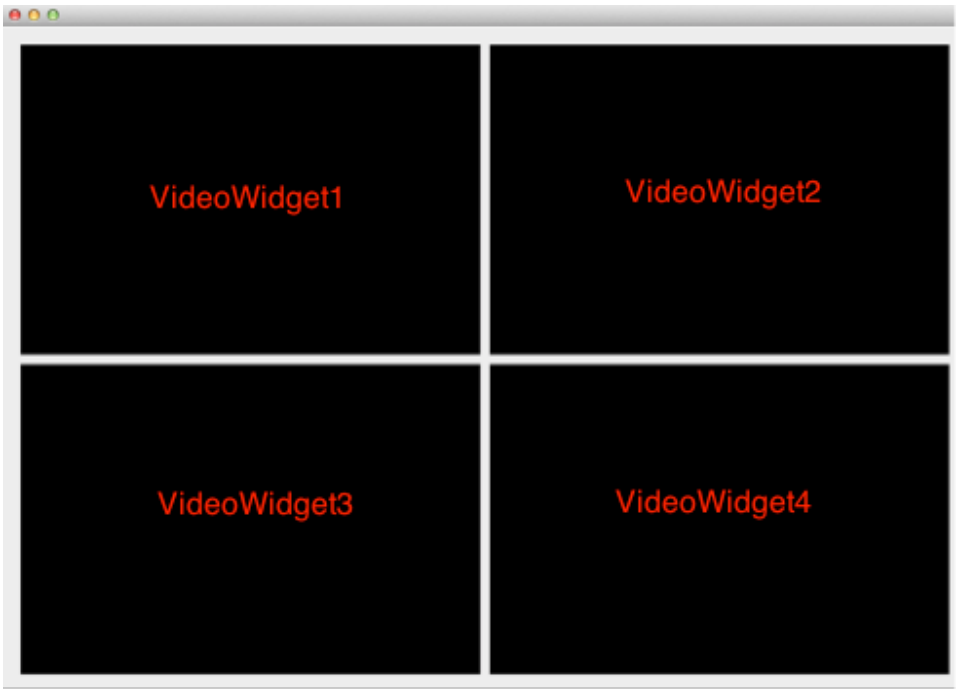


Figure 26: Video widgets layout

Open and loading videos

To allow the application to reproduce the videos, first was implemented a function which allow the user to choose which videos wants to reproduce. For this, was defined an "Open Files" button and the method *open()* to allow open the files.

The "Open Files" button is a QPushButton, that when is clicked, it calls *open()* method. Basically, a QPushButton is a command button (Figure 26) . That means when clicks this button, allows commanding the computer to perform some action or to answer a question. Usually, this buttons are rectangular and displays a text label describing its action (Open Files in our case). The push button emits the signal *clicked()* when it is activated by the mouse (i.e., pressed down then released while the mouse cursor is inside the button) and then execute *open()* method, which will be explained few lines below. This button was added to the general layout too, and the result can be seen in the Figure 28.

As said before, after clicking the "Open Files" button, the *open()* method is calling. This method allows the user to choose which files will be first loading and then reproduced in the application. To facilitate the users choose the files, was establish the following condition: the five files could be saved in any folder, and what the application open will be that directory, avoiding the user have to open the five files one by one. For this, is mandatory that the videos be saved with the following names: Camera1.avi, Camera2.avi, Camera3.avi and Camera4.avi. Also, the file text must be named FileText.txt. If the files have other name, the application will not load and open the videos, and if in the folder there are more files, the application will open only the ones with the names explaining above.

The *open()* function uses the QFileDialog to allow choose a directory and indicate which one should be opened. The QFileDialog class provides a dialog that allows users to select files or directories (Figure 28).

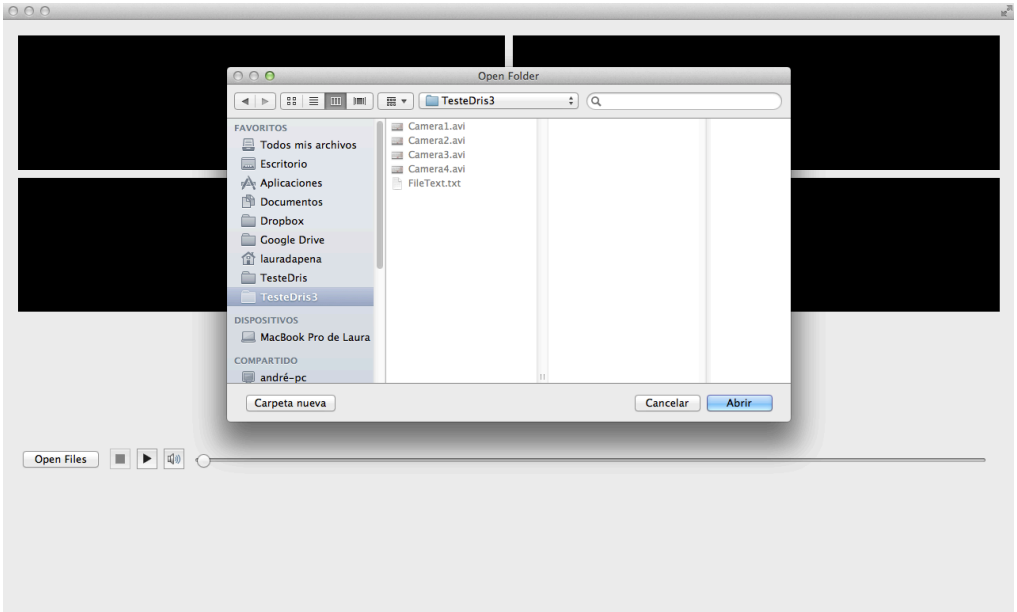


Figure 27: Dialog that allows choosing a directory

In our case, the static function getExistingDirectory(QWidget* parent=0, const QString & caption = QString(), const QString & dir = QString(), Options options = ShowDirsOnly) will return an existing directory selected by the user, after the native OS file dialog is open. This function creates a modal file dialog with the given parent widget. For DriS application, the values of these arguments are:

Arguments	Value
QWidget * parent = 0 (In which widget the file dialog will be opened)	this (i.e.: player)
const QString & caption = QString() (Defines the name of the file dialog)	tr("Open Folder")
const QString & dir = QString() (Predefines the directory in which the file dialog will open)	"/home"
Options options = ShowDirsOnly (Different options about how to run the file dialog)	QFileDialog::ShowDirsOnly

Once that the application allows the user to choose which directory wants to open, should allow him to load the videos in each video widget of the application. For this, we proceed in the same way for the four videos:

1. Was concatenated the directory and the names of the video files, composing a QString. Example: folderName + "/CameraX.avi". Where X can be 1, 2, 3 or 4.

2. Then, each one of the players set the media that will be reproduced in each video widget, as we explained in the beginning of this section. The *setMedia(const QMediaContent & media, QIODevice * stream = 0)* function holds the active media source being user by the player object. This player object will use the QMediaContent for select the content to play. In DriS application, this function constructs media content with *url* providing a reference to the QString explained in 1.

Is worth to mention that the video with the name Camera1.avi, will be loaded and reproduced in the videowidget1, the one with the name Camera2.avi, will be loaded and reproduced in the videowidget2, and the same with the other two videos (Camera3.avi → videoWidget3 and Camera4.avi → VideoWidget4).

Reproduce and synchronize videos

Once that the videos are already loaded in each video widget, was needed to make them reproduce together and at the same time. For this, was necessary to import the *Controls* class and connect the controls with the players and video widgets and to add some other widgets and functionalities.

To connect the controls with the players were used the signals and slots (as to connect the other objects). In this case, was necessary to repeat for the four players the same concept, based in the current status of one of them (the four videos should have the same state). In the following chart, can be seen how the controls are connected with the players and video widgets.

Player status	Controls	Player & Video Widget
Paused or stopped	If clicked in play	All players will start to reproduce
Playing	If clicked in pause	All players will pause the reproduction
	If clicked in stop	All players will stop the reproduction
		All video widgets will be updated to the start of the videos.
Sound enable	If clicked in mute button	Sound will be disabling.
Sound disable	If clicked in mute button	Sound will be enabling.

Also to allow the application users to navigate through the videos was added a horizontal slider with a range from zero to the duration of the videos. This slider is connected with the media players in the following way: when the slider is dragged the signal *sliderMoved(int)* is triggered and in consequence, the media player calls to the *seek(int)* function, which will be explained few lines above. Was assumed that videos recorded in the same experience have the same duration. Also, was added besides the end of the slider a label to indicate the current time position of the video and the final duration (Figure 29).

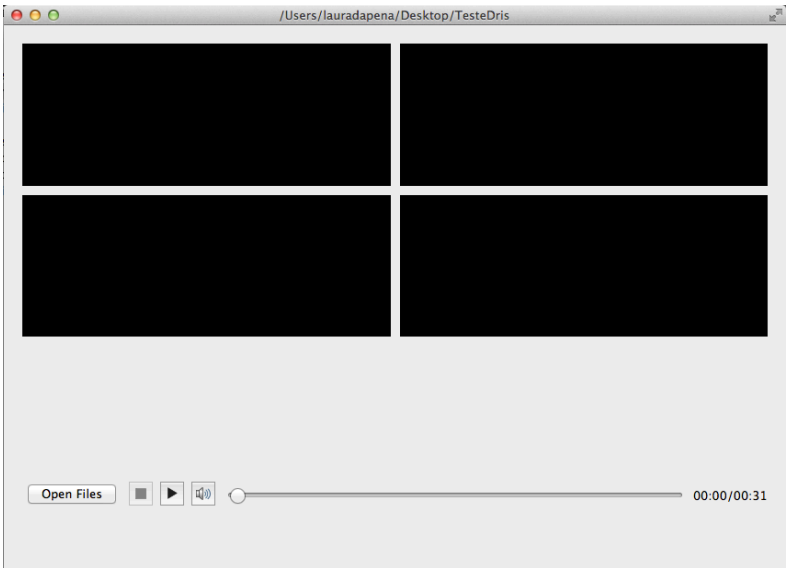


Figure 28: DriS application with slider layout

Moreover, were defined some functions to ensure the videos, the slider and the duration label working fine together. These functions are:

- ***updateDurationInfo(qint64 currentInfo)***: The function of this method is to update the duration information of the duration label, as well as give it the format: *hh:mm:ss/hh:mm:ss* (Figure 29). The first time represents the current time information while the video is being played and the second time represents the duration information of the whole video. For one set of videos, the duration of the whole video should be static. Besides, the current time information will depends on the position of the slider.

- ***durationChanged(qint64 duration)***: This function allows the application to update the duration of the videos (when new videos are loaded), both in the slider to set the maximum and in the *updateDurationInfo(qint64 currentInfo)* to update the time with the whole duration of the videos. This signal is connected with the media players through the player1.

- ***positionChanged(qint64 progress)***: This method is a very important one. Using the argument *progress*, will update the value of the slider when the slider is down. For this reason, the user only can move through the videos when the slider is clicked (or down). This method also update the current

time info in *updateDurationInfo(qint64 currentInfo)*. Is connected with the media players through the player 1, as the above function.

- ***seek(int seconds)***: This function sets the position of each one of the media players when the slider is moved. To make this, was used the function *setPosition(qint64 position)* of QMediaPlayer class. The *setPosition* function holds the playback position of the current media. The parameter *qint64 position* is the current playback position expressed in milliseconds since the beginning of the media.

- ***statusChanged(QMediaPlayer::MediaStatus status)***: Define the different status in which the media player's current media can be. It only can be in one status each time. And this function is called by the media player when the media player 1 triggered the signal *mediaStatusChanged(QMediaPlayer::MediaStatus)*, i.e. when the status of the media changes. The possible status of a media player and its explanation can be found in [31].

- ***setStatusInfo(const QString &info)***: This function set the title of the window in function of in which state is the media player.

Text reader

At this point, the only missing element to introduce in the application is the text reader. For this function, was created a QTextEdit widget and was integrated in both, the layout and the functionality of DriS application.

The QTextEdit class provides a widget that is used to edit and display both plain and rich text. Also, is an advanced WYSIWYG viewer/editor optimized to handle large documents and to respond quickly to user input. It works on paragraphs and characters. A paragraph is a formatted string, which is word-wrapped to fit into the width of the widget, and by default, when reading plain text (DriS app case), one new line signifies a paragraph. If the text to show is too large to view within the text edit's viewport, a scroll bar will appear.

To open the document, was added in the *open()* function a part for the text edit. Proceeding like with the video files, was created a QString concatenating the name of the directory and the txt file. (Example: folderName + "/FileText.txt"). For this, and as said when *open()* was explained, the text file must be named FileText.txt in the directory where the videos and the text file are. In other case, this file will never be open in the application.

For DriS application, was only needed the text widget to display and read the text, not to edit it. For this, was set the option *ReadOnly*, which allows the user only navigate through the text and select text, not being possible modify the text.

To display the .txt in the text widget was created a QFile object with the QString concatenated before. The QFile class is an I/O device that provides an interface for reading from and writing to files. The text file is opened using the *open()* function of the QFile class and reading using the *readAll()* function of the same class. After, was set some options to allow display the text file in the text widget:

1. ***setPlainText()****:* this property gets and sets the text editor's contents as plain text.

2. ***setLineWrapMode()****:* this function is used to specify the kind of line wrap wanted. It exists four types: *NoWrap, WidgetWidth, FixedPixelWidth and FixedColumWidth.* To implement this application, was chosen the *FixedPixedWidth* mode.

3. ***setLineWrapColumnOrWidth()****:* this property holds the position (in pixels or columns depending on the wrap mode) where the text will be wrapped. For DriS application, was set the *FixedPixelWidth* mode (as said above), so the value will be the number of pixels from the left edge of the text edit at which text should be wrapped.

After all this implementation, the application has the following aspect:

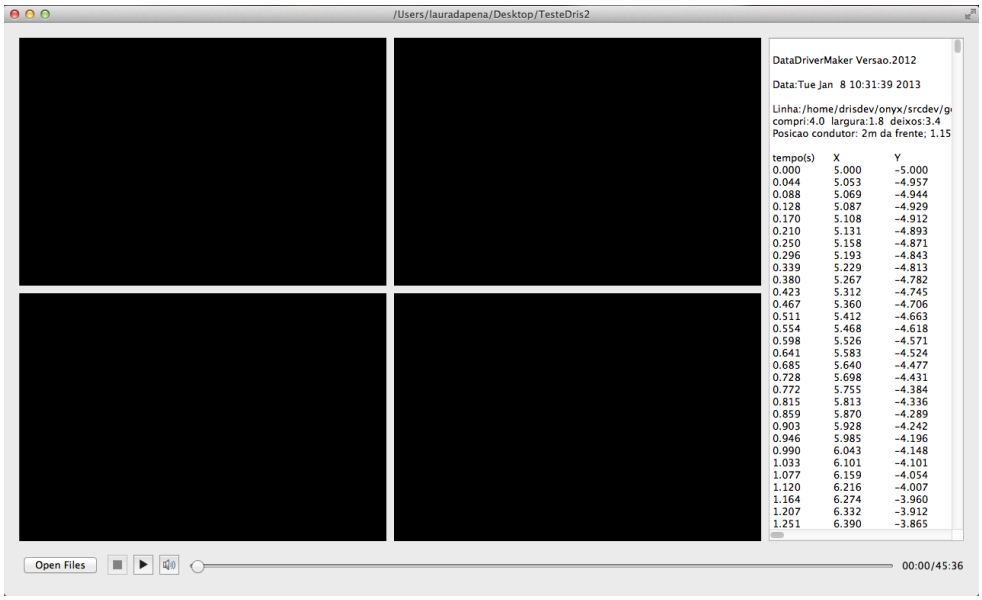


Figure 29: DriS application with video widgets and text reader

To finish the implementation of the video-monitorization system, only is left the part that allow to synchronize the reading of the text file with the reproduction of the videos and with the position of the slider. For this, was created the function *updateTextScroll()*, which is called from the *positionChanged(qint64 progress)* method to synchronized the text when the slider is changing its position.

This method considers two states: when the video already starts its reproduction (player or pause state) and when the video is waiting to start the reproduction. To show the text file conscious with the videos, were used the QTextCursor class.

As almost all the people knows, the text cursors are objects used to access and modify the contents and underlying structure of text documents via a programming interface that reproduce the behaviour of a cursor in a text editor. This class offers an API to access and modify text documents, and contains information about the cursor's position inside the document and any selection that it has made. QTextCursor is modelled on the way a text cursor behaves in a text editor, providing a programmatic means of performing standard actions through the user interface.

- **Case 1: The video already starts its reproduction**: The first thing that was done was to look for the word *"tempo(s)"* in the document, which represent

the title of the column in where the time data is presented. All the words that appear in the document above this word are showed when the videos have not started the reproduction yet. Once that known this word exists on the document, the first position of the cursor is defined to be behind the first character of the first line of the document (i.e.: behind the first time number). Then, each time that appears in the text file is converted to a *float* in order to allow comparing this number with the position of the player. If this time in the text document is bigger or equals than the current time position of the player, was created a QTextCursor, which moves down one row each time and approximately 20 rows per second. To be easier to analyse, what was be done was to highlight the current row that corresponds with the duration of the player, as can be seen in the following picture (Figure 31).

Figure 30: DriS reproduction mode

- **<u>Case 2: The video is waiting to start the reproduction</u>**: In this case, was created a *QTextCursor* too, which will set its position to zero, since the video has not started to reproduce and the text file has not start to read. In this case, no line is highlighted in the text widget, as can be seen in the Figure 30.

4.2.4 Main class

As its name says, the main class is the most important class for the application. In our case, it inherits the other three classes described above, and also the QApplication class, since our program will be an application.

In this class, a Player object is created and executed, leading to the final result of this thesis: an application that represents a video-monitorization system to a realistic driving simulator. (Figure 31).

5 Conclusion and future work

The driving simulators world is always in constant evolution being marked by the rhythm that dictates the technological advances in computer graphics, mechanics, electronics and in the automotive industry. The use of these simulators could be almost infinite, being used by universities, laboratories and car manufacturers to research and study the data collected in them for a variety of purposes.

The simulator used and studied in this thesis was the DriS driving simulator, and in the second point of the first chapter, could be seen its structure. In it, is possible to read that the DriS simulator is placed in a laboratory that is divided in three rooms, one where the car is for make the experiences, and two control rooms. To monitor the driver experience, three cameras placed inside the car and one outside it, record all the movements of the driver, being principal the ones that concerns the controls, the pedals

and the driver body. The camera placed outside records the driving environment, i.e.: the road and the environment of it (landscape, people, buildings, etc.). Is important to mention that this driving environment is made by the specialists who create the experiences and vary depends on the purpose of the driving experience.

The main objective of this thesis was to create a video-monitorization system for DriS realistic driving simulator, in order study and analyse the data recorded, both in video and text format, in a singular driving experience. To achieve this objective, was proposed to make an application and for this, were studied some software tools to do it, finally choosing the Qt library.

Choosing Qt to use it with its integrated development environment was a very good decision, due to all the possibilities that this library offers and all the documentation that its provides to help and take full advantage of this tool. Also, the facility of Qt to port an app between platforms, made easy that DriS app works in Windows, Linux and MacOs.

The created application consists in a basic video application with a very simple user interface. It covers all the features specified before the thesis starts: it reproduces at the same time four videos and reads one text file. All of them are time-synchronized and with a slider the application users could move these files forward or backward. Also and as a video application were implemented some basic controls as play/pause button and stop button and a mute button that allow to hear the sound of the four videos or mute them.

This application can be used for multiple research topics based in a driving simulator, and will bring to the researches a big number of advantages. The main advantage is the fact to see all the videos and the data that the system records at the same time. This allow the researches to study at the same time the reaction of the drivers at different obstacles, changes in the pavement, curves, etc., allowing them to have a more realistic vision of the drivers experience in the simulator.

Some of the research topics in which DriS application can be applied could be:

- Evaluate and improve current driving learning methods.
- Study the different parts of the pavement to improve it and reduce the number of accidents.
- Study the reaction of the different parts of the driver's body when an unexpected object arises in the road.
- And many more, where the behaviour of the driver can affects.

Basically, the result of the thesis is good, being the behaviour the pretended when the thesis was proposed. Besides that, in the future, some improvements can be done.

5.1. Future work

Although the good result obtained in the development of this thesis, some improvements can be done. Below, can be found a list of some of them, and a short explanation of why they should be done.

- **Layout improvement**: right now, the layout of the app is very basic. In the future app layout could be improved and make it more user friendly, adding a tool bar with some options, or more buttons (forward and backward for example), etc.

- **Desktop app icon**: for the app being easier to find in the computer and being more user friendly (again), would be good if it has an icon that represents it.

- **Change dynamically the number of video widgets**: maybe in the present future some cameras can be added to DriS car to record some other useful data for new studies. In this case, DriS can allow reproducing all these videos without made changes in the app code. Usually a maximum number of videos should be defined, to not have performance problems.

- **Change the video widgets size dynamically**: depends on how many videos are being reproduced at the same time, the video widgets size could change. *Example*: *if only one video is being reproduced, this video widget could occupy the whole screen. If two videos are being reproduced, the video widgets can occupy each one the half of the screen, etc.*

- **Error messages**: when something unexpected happens in the application, the users like to have feedback on what is the problem. One good improvement could be adding error messages in the app when something unexpected (wrong video name, wrong video format, etc.) arises.

- **Improve slider movement**: right now, the movement through the slider is only allowed when the slider cursor is pressed down. Ideally and as happens in very known video players (*Youtube, MediaPlayer*, *VLC*, etc) the app can allow the forward and backward movement only with click in the slider and not in slider's cursor.

- **Improve the display of the data collected during the experiences:** right now, the data collected during the experiences is displaying in the app as a text file. A good to have in the app is representing this data as graphics instead as a file text, making easy for the professionals analysing the experiences. If this proposal is impossible to implement, would be good to have a checkbox with different possibilities which allow to show in the app the information that is needed in this moment and that is inside the text file (for example, if the user only want to see the speed, the time and the X,Y,Z position.)

- **Include maps to improve the localization of the cart during the ride:** sometimes a video can be less important, and what really matters is the location of the cart during the ride. For this, is important to keep in mind that would be great to have other widget in the app in which is presented in a 2D map the ride and the exactly location where the cart is in real time.

References

[1] Univerisity of Iowa. *The National Advanced Driving Simulator*. From http://www.nads-sc.uiowa.edu/

[2] Toyota. *Pursuit for Vehicle Safety*. From Toyota: http://www.toyota-global.com/innovation/safety_technology/safety_measurements/driving_simulator.html

[3] Blana, E. (1996). A Survey of Driving Research Simulators Around the World. *Institute of Transport Studies, University of Leeds.* Leeds, UK.

[4] University of Minnesota. *Human First* . From Driving Simulation: http://www.humanfirst.umn.edu/capabilities/simulation/

[5] Vti. *Research Areas*. From Driving Simulator: http://www.vti.se/en/research-areas/vehicle-technology/driving-simulation/

[6] Transport Research Laboratories.*TRL*. From http://www.trl.co.uk/

[7] M. Leitão, J. Santos, A. Sousa, & N. Ferreira (1999). *Evaluation of Driving Education Methods in a Driving Simulator.* Coimbra.

[8] J. Miguel Leitão, A. Augusto Sousa, & F. Nunes Ferreira. (1999). *An Image Generation Sub-system for a Realistic Video Simulator.* Las Vegas.

[9] Silicon Graphics International. *OpenGL Performer*. From sgi: http://oss.sgi.com/projects/performer/

[10] Itseez. *OpenCV*. From: http://opencv.org/about.html

[11] QtProject. From Qt Project: http://qt-project.org/

[12] Gary Bradsky, Adrian Kaheler. (2008). *Learning OpenCV.* O'Reilly.

[13] Wikipedia. *Callback (Computer programming)*. From http://en.wikipedia.org/wiki/Callback_(computer_programming)

[14] MacPorts. From http://www.macports.org/

[15] Max Howell & Rémi Prévost. *Homebrew*. From http://brew.sh/

[16] CMake. From http://www.cmake.org/

[17] Jeffrey Thompson. *UPDATE: Installing OpenCV on Mac Mountain Lion/Mavericks*. From Jeff Thompson + Blog: http://www.jeffreythompson.org/blog/2013/08/22/update-installing-opencv-on-mac-mountain-lion/

[18] Sadeep. *How to install OpenCV on Mac OS X Lion to work with XCode*. From Sadeep's Tech Blog: http://sadeepj.blogspot.pt/2012/03/installing-and-configuring-opencv-to.html
[19] *Qt Core module*. From http://qt-project.org/doc/qt-4.8/qtcore.html
[20] *Qt GUI module*. From http://qt-project.org/doc/qt-4.8/qtgui.html
[21] *Qt Multimedia module*. From http://qt-project.org/doc/qt-5/qtmultimedia-index.html
[22] *Qt Multimedia Widgets module*. From http://qt-project.org/doc/qt-5/qtmultimediawidgets-index.html
[23] *Qt Widgets module*. From http://qt-project.org/doc/qt-5/qtwidgets-index.html
[24] *Qt Network module*. From http://qt-project.org/doc/qt-5/qtnetwork-index.html
[25] *Qt QML module*. From http://qt-project.org/doc/qt-5/qtqml-index.html
[26] *Qt Quick module*. From http://qt-project.org/doc/qt-5/qtquick-index.html
[27] *Qt Quick Controls module*. From http://qt-project.org/doc/qt-5/qtquick-index.html
[28] *Qt Quick Dialogs module*. From http://qt-project.org/doc/qt-5/qtquickdialogs-ind
[29] *Qt SQL module*. From http://qt-project.org/doc/qt-5/qtsql-index.html
ex.html
[30] *Qt Test module*. From http://qt-project.org/doc/qt-5/qttest-index.html
[31] *Qt Possible status of a media player*. From http://qt-project.org/doc/qt-5/qmediaplayer.html#MediaStatus-enum

Annex: User manual

Requisites

Before start using DriS application, the users have to take care of some requirements that the files they want to load should complain:

1. All the files related to the same experience should be stored in the same directory.

2. The names of the files may be: Camera1.avi, Camera2.avi, Camera3.avi, Camera4.avi and FileText.txt. (Figure 31).

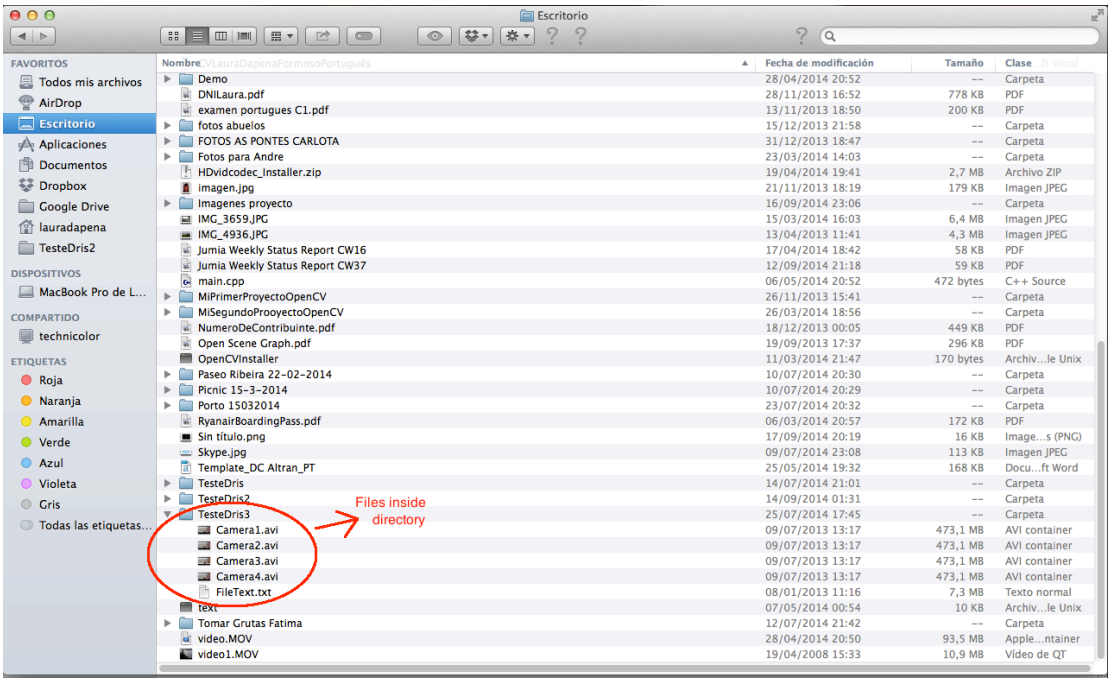


Figure 31: Files correct stored in the directory

3. All the other files stored in this directory, will not be loaded nor opened.

4. To ensure a good app behaviour, is recommended the videos has the same duration, and the text file be the one correspondent to the same driving experience as the videos.

Open and loading the files

After meeting the requirements, the user can open the application. The first screen that will appear will be the following one.

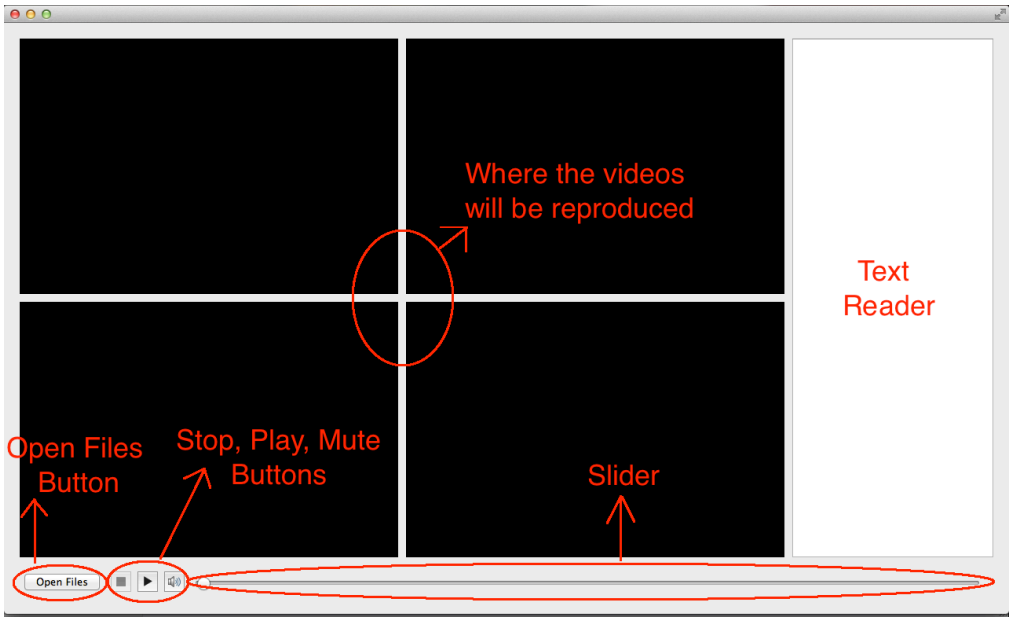


Figure 32: DriS app without any files loaded

In this screen (Figure 32), no files were loaded, so if the user clicks in the buttons, nothing will happen. Only if he clicks in the "Open Files" button, the application will open a files dialog, where the user can choose the directory wanted and open it and load it in the application clicking in open (Figure 33).

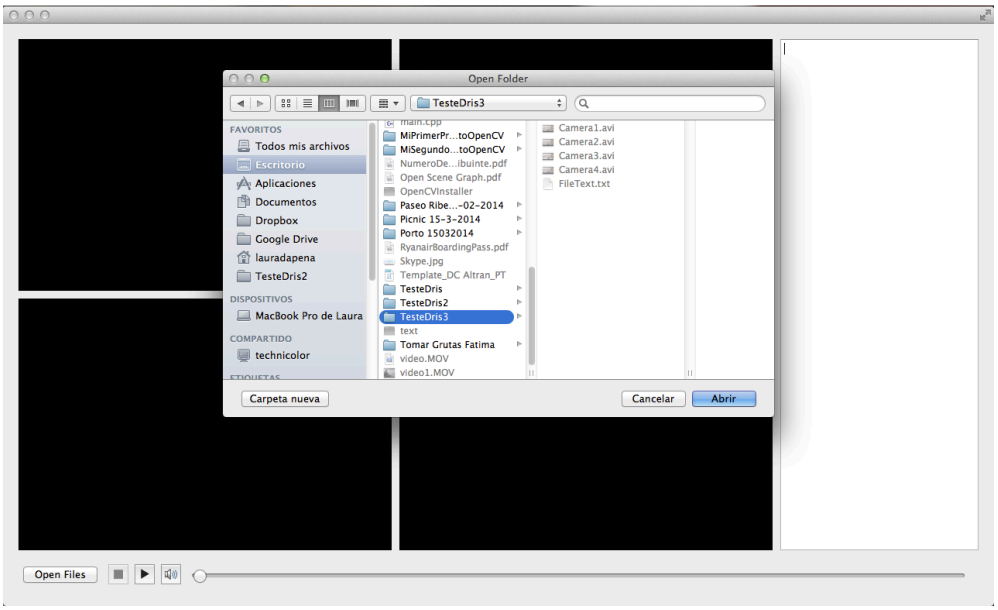


Figure 33: Open files screen

Once the videos are loading (each one in one of the black screens), the application looks like as showed in the (Figure 34).

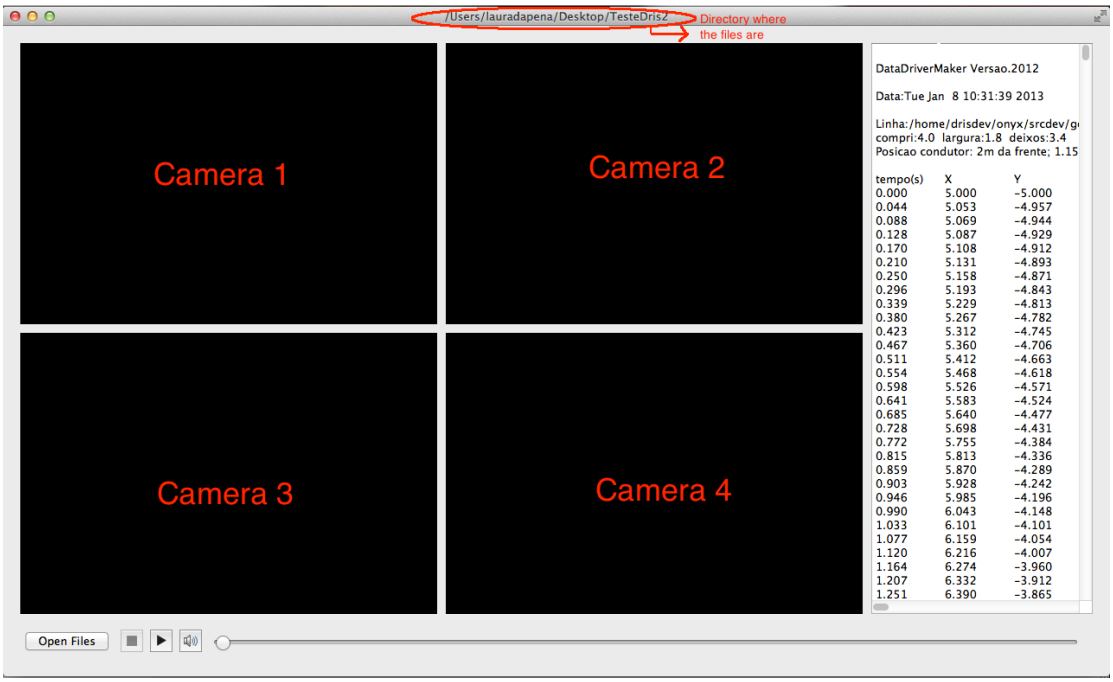


Figure 34: DriS application with the files already loaded

Until the user does not click in the play button, the videos will not appear in the screen, although they are loaded.

Buttons functions

When the videos are already loaded, clicking in play button will start their reproduction (Figure 35).

Figure 35: DriS in reproduction mode

Once the application is in reproduction mode, the slider cursor starts to move, the stop button is enabled and the play button change its icon to pause icon. In this status the following cases could happen:

1. **Clicks in the pause button**: the pause icon will change to play icon again, the stop button will be enabled also, the slider cursor will stop and the text reader will stop also (Figure 36).

Figure 36: DriS in pause mode

2. **<u>Clicks in stop button</u>**: the stop icon will be unable, the pause/play button will have the play icon, the screens will be black again, the slider will go to the beginning as same as the text reader. Basically, clicking in stop will stop the reproduction and if clicks in play again will be start the reproduction from the beginning.

Figure 37: DriS in stop mode

3. **Clicks in mute button**: when clicks in mute button, the icon of the button will change and the DriS will be in mute mode, i.e.: will not reproduce the sounds of the videos.

Figure 38: DriS in mute mode

4. **Slider cursor**: clicking in the slider cursor, the application allows to move through the slider forward and backward, moving the videos and the text reader forward or backward too.

5. **Text reader**: the text reader is synchronized with the videos and slider. When the videos start to play, the text reader starts to read, highlighting in blue the current time of the reproduction. Is composed by several columns, that the user can see if moves the text slider from right.