INGENIERÍA DE TELECOMUNICACIÓN - INGENIERÍA TÉCNICA EN INFORMÁTICA DE SISTEMAS

Escuela Técnica Superior de Ingeniería de Telecomunicación

Academic Year 2014-2015

Final Year Project

3D Augmented Reality System using JdeRobot

Author: Yazmin Lucy Cumberbirch
Tutor: José María Cañas Plaza

Victory belongs to the most persevering.

Napoleon Bonaparte

Acknowledgements

I would like to thank all the people who have helped me on this long and fascinating journey called University.

First of all, to the tutor of this project, José María Cañas Plaza, whose hard work along my side, always having a nice word to say and a motivational speech have helped me so much along the way, as well as made me learn so much. It has been a pleasure to work with you. Thank you!

I would also like to thank everyone involved in the Robotics department of the URJC, who always have an answer for our development problems through our Mail List, and are always willing to give a hand when needed.

Next, to my family, who have helped me be where I am today, as none of this would be so without them. Grandma, you have been an absolute inspiration for me during all my life, and always shown me to move forward and follow my dreams, regardless of how hard these may seem at the beginning. Mum, Charlie and Matthew, thank you for always being supporting me and offering all the help needed during these years. I have missed you all dearly, and can only appreciate the constant support you have given me. To the rest of my family, thank you all for your support and help given during these last years, regarding both university and life in general. And to Dani, for being by my side, regardless of the day or time, and providing unconditional support during the development of this project, as well as with everything else in life.

Last, but not least, I must thank all the friends I have made during these years at university. I must say, It has been an absolute pleasure to work (and not work, depending on the day) with all of you.

Thanks!

Summary

One of the most attractive fields within computer vision is that of visual localization. This opens the door to applications that can be developed for mobile phones or *tablets*, and can also be combined with augmented reality, to turn our applications into very visual and entertaining ones.

The project we present in this document is immersed in both these technologies, and aims to design and develop a self localization algorithm that is based on visual markers, as well as an augmented reality application that is based on it. The particular augmented reality will be a film that is continuously played in a virtual screen drawn over the real scene, in the same 3D area regardless of the location and orientation from which the camera observes the scene. This algorithm has been validated experimentally for several scenes, with the markers located in various different positions and numbers, studying the error in the varying axis.

For the development of this project we have used JdeRobot 5.2 software platform. The development has been done in C++, and we have used ICE for the communication interfaces between the different components. In order to perform the graphics rendering we have used OpenGL. OpenCV has been used to process images, and April Tags and ARuCo to interact with the markers and allow an easy detection.

Contents

List of Figures

Chapter 1

Introduction

Sight isn't just one of the most important senses for the human being, but it is also one of the most interesting ones. All the tasks that are related with vision need a great amount of brain activity, and they allow us to obtain about 80% of the sensations and information that surround us.

Not all the information we can capture through our eyes is real though. In some occasions, our brain can fool us into believing as real, certain situations that are actually just optical illusions caused by perspective of tricky illumination. These situations that we can consider confusing, or a limitation in our human condition, can also be taken advantage of, and thought of an open door towards imagination and illusions. To represent these ideas, it is important to mention very traditional examples, such as optical illusions that are designed with that aim, as well as more modern ones, such as video-games, which each year surprise us more with the realism of the graphics. Also, video-games are starting to go for new immersive tendencies, in which they make use of either virtual reality, in order to build the sensation of being inside the video-game world, which is all simulated in a virtual environment, or augmented reality, in which they introduce virtual objects to our real world, giving us a whole new perception of the gaming world.

Apart from the obvious uses this may have in entertainment, we also have to mention that there are several other fields that are also investigating into these new technologies, in order to put them in their advantage. Mobile applications and marketing are two of the most popular ones, between many others. It is thanks to the great futuristic aura they have, as well as the fascination that is created when playing with illusions that have built the interest in the development of this project.

In this first chapter, we would like to give a general idea of the context that surrounds this project, which can be placed in between computer vision, visual auto-localization and augmented reality. We shall also give an insight into the present of these technologies, as well as examples of their use.

1.1 Computer Vision

We can understand computer vision under two different points of view. One of them, is as the study of the human sense of vision, in order to obtain a computer model that can reproduce it. The other, the design of autonomous systems that can perform some of the activities that have traditionally been job only for the human eye.

The origin of this field of study can be placed in 1960, when Larry Roberts [10] wrote his doctoral thesis in the MIT, and performed a study on the possibility of extracting 3D information from standard two dimensional images taken from different perspectives. Using this as a starting point, several other investigation projects were performed on computer vision, applied to a world created with geometric blocks. After some investigation, they got to the conclusion that a certain image preprocessing was needed in order to be able to obtain valuable results.

The cheap hardware needed, and the exponential growth of the computers processing have been two key points in the evolution of this technology, yielding in various fields that have developed applications with it:

- **Industry:** In industrial plants, inspections and quality control are very important in order to be successful. Using computer vision systems we can manage to comply with the highest standards without affecting the work rhythm, thanks to the great speed of processing that can be fulfilled. In Figure 1.1 we can see an example of a bottle inspection system, as well as an automatic packeting robot.

- **Security:** Be it in big events, with people identification systems, to a more personal perspective, with home security, accident prevention or the monitoring of an elderly person, computer vision has given the security world a totally new perspective.

- **Identification:** With the world being more digital as the time goes by, biometric recognition (be it face, fingerprints or eyes, between others), allows us to not have to carry so many keys or ID cards around, as well as substituting traditional passwords.

(a) Quality Control System

(b) ABB Robot: Automatic Packeting

Figure 1.1: Computer Vision Industrial Applications

- **Sports:** Even though most people know the *Hawk-Eye* system that has been used in tennis to determine whether a ball has bounced in or out of the pitch using the information from several cameras, we can also find other similar applications in other sports that aren't so well known. The repetition of a baseball run with several different vision angles is another very interesting one, as well as the new system introduced in the last football world cup, which sets several cameras around the stadium in order to determine whether the ball crossed the line or not.

(a) HawkEye System

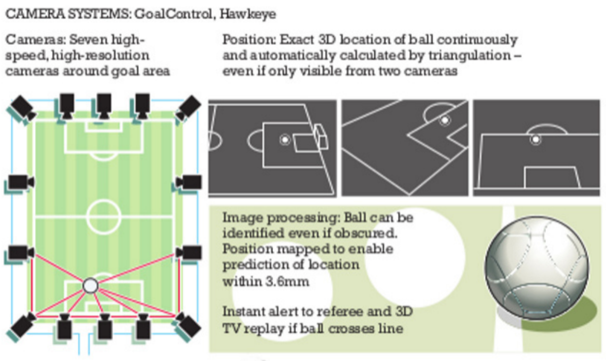
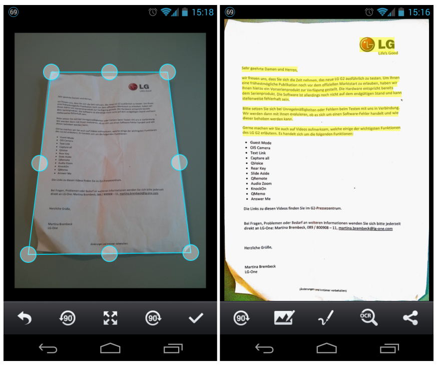

CAMERA SYSTEMS: GoalControl, Hawkeye
Cameras: Seven high-speed, high-resolution cameras around goal area
Position: Exact 3D location of ball continuously and automatically calculated by triangulation – even if only visible from two cameras
Image processing: Ball can be identified even if obscured. Position mapped to enable prediction of location within 3.6mm
Instant alert to referee and 3D TV replay if ball crosses line
(b) Goal Control 4D System

Figure 1.2: Computer Vision Sports Applications

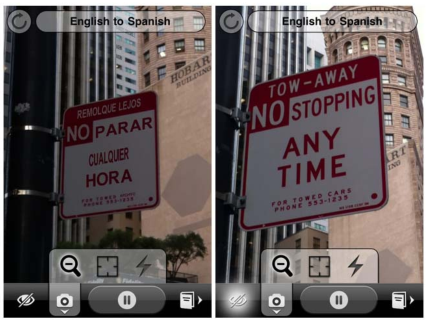
- **Robotics:** When developing the intelligence for a robot, one of the most complicated tasks we can find is that of managing for the robot to understand and interact with its surroundings.

- **Entertainment:** In the videogames world, computer vision can provide a whole new way to interact with our games, using our body as the remote control, or with augmented reality to give a new realism to games that wasn't available before.

- **Mobile Applications:** One of the handiest fields with computer vision applications, is those developed for your mobile phone. Thanks to C*amScanner* (Fig. 1.3a), you can have a scanner in your pocket, or with *Word Lens*(Fig. 1.3b), a international dictionary that works with a picture of what you would like to translate. Not understanding traffic signs when you travel will no longer be a problem, thanks to the great advances that are being done in this field.



(a) CamScanner



(b) Word Lens

Figure 1.3: Computer Vision Mobile Phone Applications

- **Medicine:** Computer vision is of great importance in the medical field, specially in terms of image analysis in order to obtain more accurate diagnoses. These images can provide from X-rays, ultrasounds or by a magnetic resonance, and their correct analysis is very important for the lives of many people.

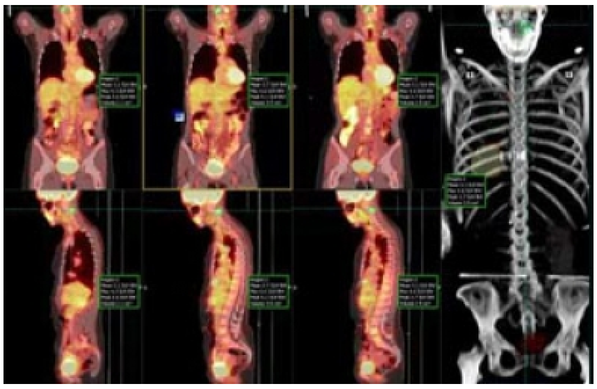


Figure 1.4: Miranda Medical

In this field we must make a mention to *Miranda Medical* [1], a company that is focused on software development and diagnosis tools for medical images. One of their best known software products is *XD3*, which we can see in Figure 1.4, and is aimed at the diagnosis of oncologic illnesses. It is being used by several centres around the world to evaluate the response patients are having to their oncological treatment, and the advance of the illness.

1.2 Visual Auto-Localization

Computer vision can be divided into several different sub-fields. One of the most interesting ones, and a base for the present project, is *visual auto-localization*. This can be described as the ability of an algorithm to estimate the position of a camera, using as input only the images that are taken by this camera. This means that, with a series of RGB images taken by a camera, we should be able to calculate the absolute position and orientation of this camera, taking a point in the world as reference.

As we have mentioned, the only information that a visual auto-localization algorithm will receive is that of an RGB camera, not having any depth information, such as that we could have if using a Kinect Camera (RGBD), nor laser sensor information. This has its fall backs, obviously, but we can also mention several advantages of only using RGB information. For one, the cost involved in hardware is minimal, as we are talking about any normal camera, and can be found in a wide variety of devices, be they mobile phone, *wearables* or web cams, between others. Apart from the cost, one of the fall-backs of using RGBD information is that this is provided using infra-red lights. This means it cannot be used outdoors, as it would combine with the sun's infra-red lights.

When talking about visual localization, there are several different techniques, depending on the way this localization is performed. We shall now mention the three most important ones:

1.2.1 Visual Odometry

We can talk about *visual odometry* as a method to calculate the relative movement between the camera positions when taking two separate pictures. Therefore, if we have two

[1]http://www.mirada-medical.com/

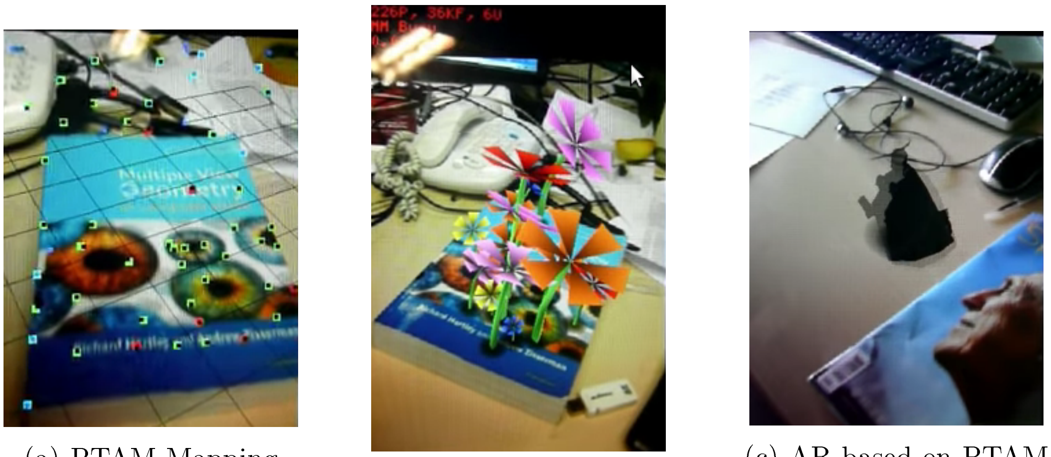
sequential images, a visual odometry system would calculate the movement the camera has made to go from the place it took the first image, to that where it took the second.

The most commonly extended method to perform visual odometry, is based on the extraction of characteristic points of an image, that allows us to compare these points between both images to find matching points, and calculate the movement that must have been performed in order to have this change.

1.2.2 Visual SLAM

Visual SLAM (Simultaneous Localization and Mapping) is a technique that allows us to perform both the mapping of an area that we are observing and the localization of the camera in that world. This method is also based on characteristic points, which we must obtain through a two dimensional image processing algorithm (*FAST*, for example), and then provide these to a 3D processor.

From the work on *monoSLAM*, by Davidson[2] [2] is where the first ideas on how to perform such actions simultaneously. The idea proposed is to use an extended Kalman Filter to calculate the camera's estimated position and orientation, as well as that of a series of 3D points. Before this idea, the only possible way we could perform this action was with an initial known position of at least three 3D points. From this point, we no longer needed to have any information of the map we are using previous to the processing.



(a) PTAM Mapping (b) AR based on PTAM (c) AR based on PTAM

Figure 1.5: PTAM Augmented Reality Application

Also, it is very important to mention the work proposed by George Klein[3] [7] regarding

[2]http://www.doc.ic.ac.uk/ ajd/
[3]http://www.robots.ox.ac.uk/ gk/

PTAM (Parallel Tracking and Mapping). One of the main fall-backs all the algorithms based on *monoSLAM* have is that the computing time increases exponentially when increasing the number of 3D points we wish to use for our estimations. This so due to the fact that, for each iteration, the algorithm performs both the mapping and the localizing with all these points. This problem was dealt with by Klein, who suggested the mapping and localization should be separated, with the condition that the localization must be performed in real time, when the mapping can be done in an asynchronous mode, in order to optimize the timing. An example of an augmented reality application that makes use of PTAM can be found in Figure 1.5, where we can see the mapping performed with certain points in 1.5a, and the AR based on this mapping and camera location on 1.5b and 1.5c.

1.2.3 Localization based on Markers

The last localization method we are going to mention is that based on markers. This method is based on a known map, which is indicated to the camera via a series of markers which we are able to detect easily, and that have a known position for us. With these markers, when the camera manages to detect one of them, it can calculate its distance from the tag, and seen as the tag has a known position, it can also calculate it from the world's reference.

The markers that can be used are plenty, with the only limitation being that the camera must be able to recognize them easily and without error, in order for the estimation to be accurate. Some of the most common markers used are bi-dimensional bar codes, similar to QR codes. These are easily detectable by a camera that knows what it is looking for, and seen as we can use so many different ones, there shouldn't be much confusion between their detections. April Tags or ARuCo are two auto-localization libraries based on markers that have their own families of markers, compatible with their software.

1.2.4 Relevant Examples

Keeping in the field of visual localization, we can mention several examples that make use of this technique in order to provide the information that is required from them. A few of the most important are the following:

- **Drones Localization:** Given how very modern most of these techniques are, it isn't strange for the applications that make use of them to also be very modern. Drones are the new *toy* everyone wants to play with, and therefore, there is also a great amount of people that develop new applications for them. In Figure 1.6 we can see a visual odometry system[11] that is running with the images that a drone is capturing during its flight. If these methods evolve, we could soon be talking about drones that go from A to B without any human assistance.

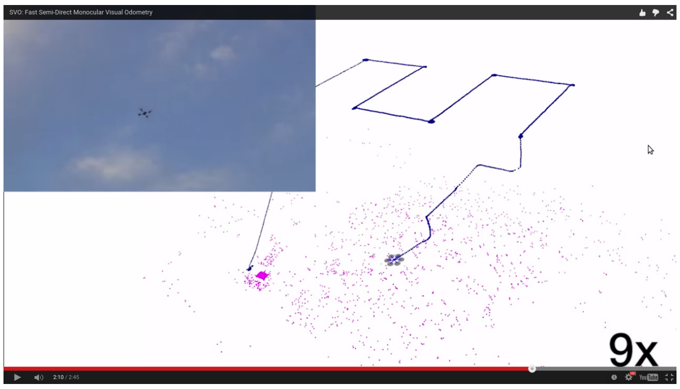


Figure 1.6: Drones Visual Auto-localization

- **Dyson Hoover**[4]**:** With a camera lens that features 360^o vision, *Dyson 360 Eye* captures 30 images a second in order to build a picture of the room, and uses this information along with a mapping software, to plot and follow the optimal minimal route around the house. In figure 1.7 we can see the hoover on its self-charging station, cleaning and moving between different surfaces, and the mobile application that we can use to interact with it.

- **Augmented Reality:** As we will mention in the following section, auto-localization is very important for augmented reality applications, as in order to obtain a certain realism that will give our application a good user experience, it's very important to have the camera located at all times, to project the virtual objects that will construct our augmented reality in the correct positions.

1.3 Augmented Reality

Augmented reality consists of, having an image flow provided by a camera, performing a virtual projection of a series of objects above this image flow, giving the end user the idea

[4]https://www.dyson360eye.com/

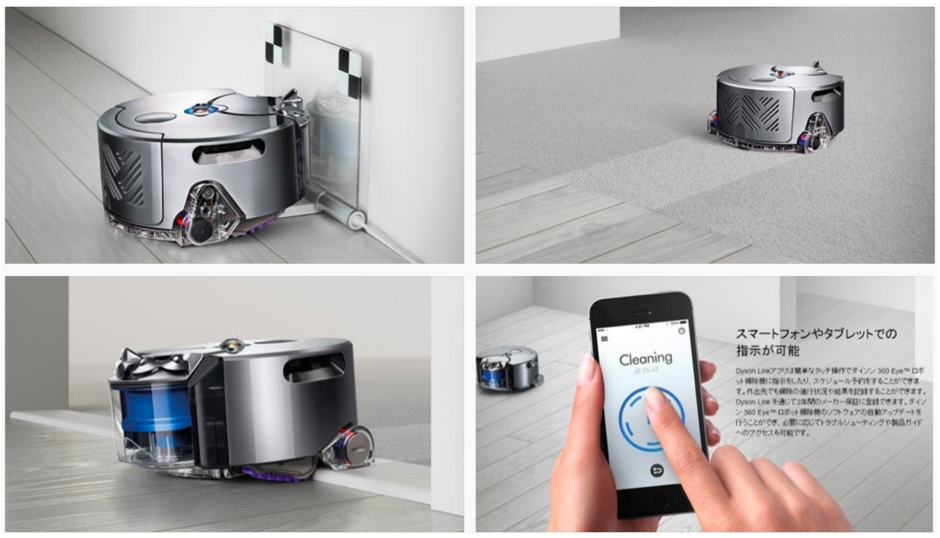


Figure 1.7: Dyson 360 Eye

that he is watching a real time video, but it has something new on it, something that isn't quite real.

In order to perform a realistic augmented reality application we must link a series of operations and algorithms, for the final result to be the best possible. First, if the camera we are using to capture this image flow is moving, it is very important to have a visual localization algorithm running, in order for this camera to have full knowledge of its position at all times, and be able to project the virtual objects in the correct place. *ARuCo*, which we have already mentioned, or *ARToolKit* are good libraries that can provide us with the minimal tools necesary for these applications, as we can see in Figure 1.8. Following this, we need a graphics engine that provides us with the rendering of the virtual object over our video flow. *Ogre* or *Open GL* could be examples for this step, as they are very extended libraries that give us the required functionalities. The last point we can introduce to our augmented reality application has to do with the realism that we wish to give it. In this line, we can differentiate between simple graphic AR, which wouldn't have much more to be added that what we already have, or physical ones. These applications try to get the virtual objects to interact with the real world, giving them physical properties such as weight. Using a physics engine such as *Bullet*, we can add realism to this application.

Augmented reality is also a very modern application, and is being used in a great variety of fields to develop applications. These are obviously very visual, and manages to easily capture the end users attention. When talking about applications, we can separate these in different fields:

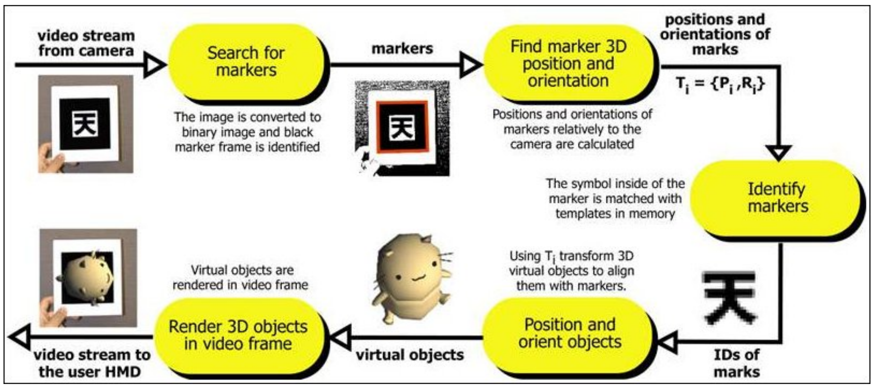

video stream from camera
Search for markers
The image is converted to binary image and black marker frame is identified
markers
Find marker 3D position and orientation
Positions and orientations of markers relatively to the camera are calculated
positions and orientations of marks
$T_i = \{P_i, R_i\}$
Identify markers
The symbol inside of the marker is matched with templates in memory
IDs of marks
Position and orient objects
Using T_i transform 3D virtual objects to align them with markers.
virtual objects
Render 3D objects in video frame
Virtual objects are rendered in video frame
video stream to the user HMD

Figure 1.8: ARToolKit Phases

- **Entertainment.** When talking about entertainment, the first idea to come to mind, and one of the first fields to develop AR applications is the videogames world. One of the earliest AR games to be available is *Invizimals*, a game launched in 2009 by Sony for their PSP. With the advances that are taking place in the smartphone world, we can now find several augmented reality games for *Android* or *iOS*, such as *AR Defender*. We can find examples of both of these in Figure 1.9.

(a) Invizimals

(b) AR Defender

Figure 1.9: Augmented Reality Video Games

Another modern way of AR entertainment is the one that *Parrot's* AR Drone 2.0 offers us. It has available several different AR games, which we can see in figure 1.10 and gives us more reasons to acquire one of these modern toys.

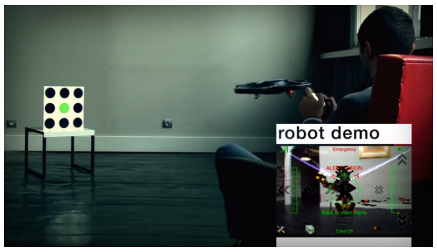
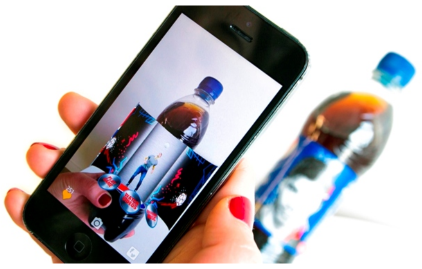


Figure 1.10: Game modes with Parrot's AR Drone 2.0

- **Advertisement.** Regarding this world, AR is perfect to capture the potential buyers attention and make them remember the product you are trying to announce. One of the options available that is most appealing is that of *scanning* the inside of a product, which will then provide the user with an hidden advertisement or some information regarding the product, as we can see in Figure 1.11. With these techniques, not only do we have more effective marketing campaigns, but also we can receive much more *feedback* regarding the end users experience.


(a) Scanning a drink bottle with AR

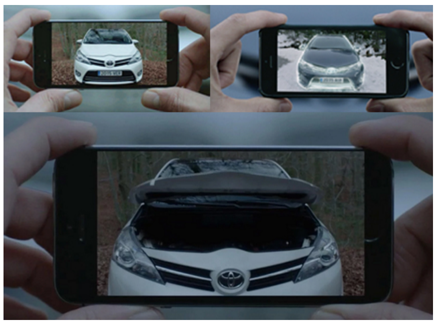
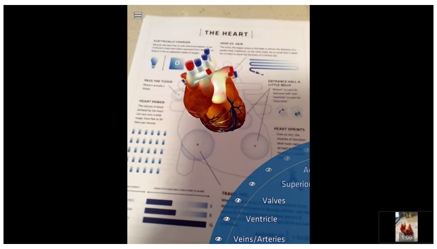

(b) Toyota Campaign based on AR

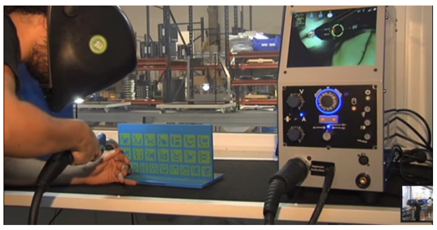
Figure 1.11: Augmented Reality examples in advertisement

Another set of examples related to advertisement are those that we see on the television between sports retransmissions. On these, we can often find the camera watching the football pitch, and suddenly see virtual people or objects appearing in scene and interacting with it.

- **Education.** Another application field with great potential is the educational one. Helping children learn in a new, modern and entertaining way will always have better results that boring standard classes, as well as providing a new point of view regarding the concepts learnt. AR can also help cheapen the cost of certain professional practical activities that need be learn, but usually involve a high cost in material and equipment. In Figure 1.12b we can see *Soldamatic*, an augmented reality welding system that has revolutionized a welders professional course, giving them the possibility to perform very realistic practical activities without risk and cost that these usually carry. Another great example is found in the smartphone application *Anatomy 4D*, with which we can analyse all the different parts of a human heart, watching how this beats, and being able to view the exact parts that we wish to visualize. This application is a great help in medical learning, as it gives the student a new way of visualizing what they learn in the books. We can see an example of this application in Figure 1.12a



(a) Anatomy 4D



(b) Seaberry Soldamatic

Figure 1.12: AR examples in Education

- **Tourism.** AR has also arrived to the tourism world, giving the travellers knew ways to discover the city they are travelling to. These provide us with new ways to discover the cities history and events, as well as information regarding restaurants and museums. In this section, the company *Layar* [5] has started creating personalised augmented reality application for different places that wish to have them, in order to offer them to their clients and increase their status as a tourist attraction. One of the first users of this system was the Segovian town hall, which has a *layar* application for their tourists to discover the city.

- **Medicine:** With the great potential that is in augmented reality, it isn't strange that the medical world has also gone into it. A good example of use in this field is the projection of a 3D reconstruction of the internal organs over a body that one is operating on. Another way to use augmented reality in the medical world is in the treatment of phobias, creating virtual projections of the elements that create the phobia in order to treat them in a controlled environment.

1.4 Visual Localization and Augmented Reality in the Robotics Group of the URJC

As we can see, **Augmented Reality** and **Computer Vision** applications are present in a very extensive number of fields, and these are growing day by day. Within the *Robotics Group* or the *URJC*, we can mention several previous projects and thesis where the fundamentals of these have been exposed.

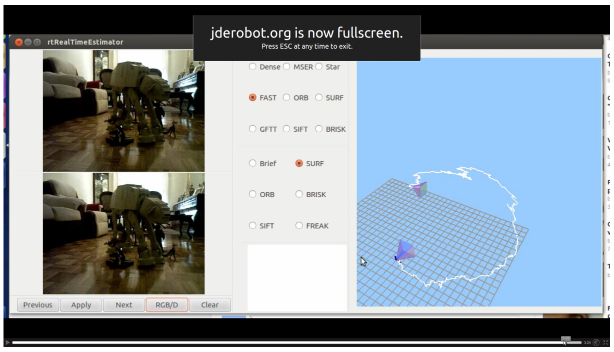
First, we would like to mention some projects done regarding camera calibration, which are very important for visual localization algorithms. To know the intrinsic parameters of

[5] www.layar.com

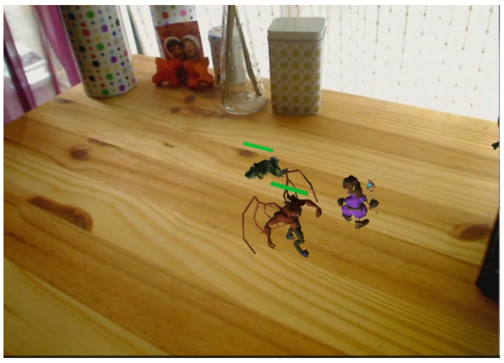
a camera is a fundamental point when having to analyse an image, as it is important to know the physics behind the camera that captures the image we are analysing to perform a better job. In this field, we can mention the projects performed by Redouane Kachach [6] y Agustín Gallardo[3].

There have also been projects related to localization making use of RGBD cameras, such as the one on visual odometry by Daniel Martín [8], who developed a system that integrates camera position and route estimation, based not only on RGB images, but also depth information provided by a Asus Xtion (RGBD). Daniel performs an estimation of the relative movement the camera has performed between two consecutive image frames using ***SVD****(Singular value decomposition)* and ***RANDSAC****(Random sample consensus)* algorithms, of which we can see the final application in Figure 1.13a

Visual self-localization with RGB cameras was the main point of the master thesis written by Alejandro Hernández [5], in which he performed implementations and tests of different visual auto-localization algorithms, as well as an Augmented Reality video game that makes use of them, as we can see in Figure 1.13b. Alex performed his own ***PTAM*** implementation, based on the original, but making use only of OpenCV and Eigen libraries.


jderobot.org is now fullscreen.
Press ESC at any time to exit.
rtRealTimeEstimator
Dense MSER Star
FAST ORB SURF
GFTT SIFT BRISK
Brief SURF
ORB BRISK
SIFT FREAK
Previous Apply Next RGB/D Clear

(a) RGBD Visual Odometry by Daniel Martín



(b) AR Video Game by Alejandro Hernández

Figure 1.13: Computer Vision Projects in the URJC

In this project, we wish to get into the world of visual localization and its appliances in augmented reality. We would like to develop a marker based visual auto-localization algorithm, making use of several different technologies that we must manage to blend in together, in order to obtain the desired results. With this algorithm in place, we would like to integrate it with an augmented reality application in order to determine the cameras position, and be able to project in consequence. Last, we would also like to port this algorithm and augmented reality application to an *Android* device, in order to verify the complexity of such migration, as well as test the software in a different environment.

This document has been divided into 7 chapters, of which the current is the first. In the second chapter, we will talk about the requisites and objectives that we have established for our applications. In chapter three, we shall go through all the hardware elements and software infrastructure that has helped in the developments. In the fourth, a description of the mathematical fundamentals that are behind great part of this development. In the fifth chapter, we shall provide a detailed explanation of the development performed for all the different parts of the project. The next chapter will analyse the results after performing a series of experiments to the application, in order to verify it quality. Last, chapter seven will provide the conclusions we have obtained during this development, along with future improvements that could be performed.

Chapter 2

Objectives

In this chapter, we are going to describe the present project's main objectives, as well as the methodology followed to obtain the final results.

2.1 Problem Description

This project is immersed in the computer vision world, and its objectives are all very tightly related to this topic. We aim to learn and understand different technologies related to the field. The main idea of the project is the development of a computer application that estimates a mobile camera's position, and uses this information to create Augmented Reality, in the form of a film projection.

We can clearly identify three main points in the development:

1. To *investigate visual localization techniques*, and develop a marker-based self-localization algorithm. This algorithm will estimate the camera's position in a known world, using as only input the images provided by a RGB camera and the position of the markers that describe our world. This algorithm will also be tested in a simulated world, in order to evaluate its accuracy, and then move on to a real scene using a RGB web cam to perform the image capturing.

2. *Development of an Augmented Reality application for PC*, using as a base the previously calculated camera position. This application will represent an augmented reality cinema, in which we will project a virtual film screen between certain points of our 3D world. This projection will take into account the perspective from which the camera is viewing the world, and the mobile nature of the camera.

3. *Development of an Android version of the previous application*, making use of libraries that allow us to reuse as much code as possible. The localization module of this application will be reused from the PC version, in order to compare the performance on different platforms, but the user interface module, along with the general application structure, will be redone in order to be able to perform the required actions on the Android platform.

2.2 Requirements

As well as the previously mentioned investigation and development, this project must also satisfy a series of requirements, which we are going to describe in this section.

- **Modular Software**. The developed software must be able to integrate with JdeRobot platform, as well as its components, using C++ programming language.
- **Platform**. The software must be executable on a Linux run PC, as well as on a mobile telephone running Android operating system.
- **Camera**. The mentioned system must work with any RGB camera that is connected to either the PC or the Android run smartphone, with the condition that the mentioned camera must be previously calibrated in order for the system to work correctly.
- **Scene**. This project will work in a real scenario with visual markers, being exportable to any world which contains the mentioned markers. The camera position estimation will be performed parting from the known location of the tags used for such estimation, and the area in which we project the augmented reality will also be defined previously.
- **Real Time**. The developed algorithm must execute in real time, in order for the film view to be realistic. It must analyse the images given by a camera and provide the data needed for the representation of the augmented reality, in such a way that there is a minimal delay between both.

2.3 Methodology

The cycle life model used during the development of this project will be a spiral model based on prototypes. This model will allow us to work on the desired software incrementally, progressively increasing the complexity of the project, and providing working simplified prototypes in each phase. The use of this model is very helpful, due to the fact that it allows the developer to have a simplified working version of his work, on which to perform tests and perfect, before moving on to higher goals in the development.

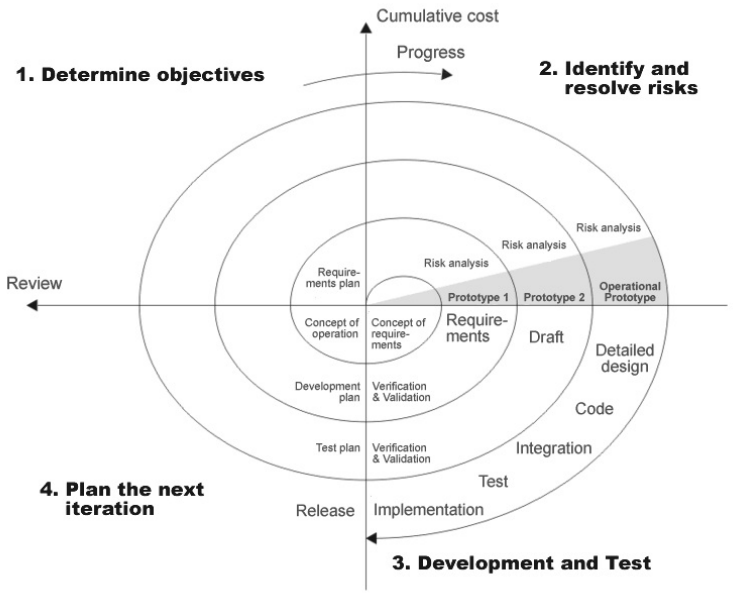


Figure 2.1: Spiral Model

The spiral model is developed in cycles, and in each cycle we can differentiate four main parts. When we have gone through all four parts of our cycle, we can move on to the next iteration of the project. In Image 2.1 we can distinguish the 4 main parts of each development cycle:

- **Determine Objectives.** We find what objectives must be satisfied after each iteration, taking in to account the desired final result.
- **Identify Risks and Alternatives.** Study of the processes that can be followed to fulfil the determined objectives, and an analysis of the pros and cons that each one holds, such as performance or time consumption. In this phase we also determine the risks that will be present in this cycle, and possible mitigation actions against them.

- **Development and Test.** After deciding the process and tools that will be used, we develop the desired product. Once the development is finished, we perform a series of tests to assure its compliance with the objectives we had for this phase.
- **Plan the Next Iteration.** After seeing the results of this phase, we start to plan the following phase taking into account the problems that we have had in the present cycle, and trying to avoid them in the following.

The cycles that we have managed for the project here described are detailed in the following section. In order to perform them accurately and decide future actions, my tutor and I shall have weekly meetings to supervise and orientate the work in the right direction, and to evaluate possible alternatives in the development.

You can find a diary on the work performed in mediawiki [1], in order to keep a temporal reference to the work done, accompanied by images and videos to analyse the results.

2.4 Work Plan

During the development of this project, the work to be done is divided in several different stages to be worked on progressively:

- **JdeRobot and image processing initiation**. The main objective of this initial phase is the familiarization with JdeRobot and the Robotics group of the URJC's general work, as well as an insight to the different projects they are working on. We will also take our first steps into image processing, and getting familiar with OpenCV library to do so.
- **Visual markers detection**. Using the image processing studied previously, an early version of this project will developed, using different colours as the tags between which to project the film. This version will then improved introducing April Tags visual markers, in order to prove a more robust detection pattern. The camera position estimation won't be involved in this stage.
- **3D Camera pose estimation**. With the detection dominated, we will start working on the camera pose estimation, combining April Tags' detection and ARuCo's pose estimation to determine the distance that the camera holds from a single tag.

[1]http://jderobot.org/Ycumberbirch-pfc

Following this, we will use the tags known pose to calculate the camera's estimated position in reference to the world's coordinates. When we considere our camera pose estimation to be good, we shall move on to the use of several tags, and the combination of the pose estimation obtained from all to obtain a more robust estimation.

- **AR Cinema in PC**. After developing a robust auto-localization algorithm, we will have obtained the first working version of our demo application. The augmented reality for this project will be developed using OpenGL API's texture rendering to represent the image obtained from our RGB camera in a background, and to render on top of this, a virtual film screen, which receives an image feed of a film saved on the PC.

- **AR Cinema in Android**. With a final PC version, we shall try to perform an integration to an Android run smartphone. This will start by familiarising ourselves with Android development, and the libraries that may be needed to obtain the desired result, paying special attention to JNI (Java Native Interface) and OpenGL ES. To develop this Android application, the core of the auto-localization will be modified as least as possible, in order to run the same algorithm in both systems, and the main changes performed will be regarding the representation.

Chapter 3

Infrastructure

In this chapter, I am going to describe the different external elements that have been used in order to give this thesis shape. I shall go through hardware devices, PC programs and third party libraries that take an important role in this project.

3.1 Hardware: Mobile Phone and Webcam

This project can be divided in two parts, mainly differentiated by the hardware used in both. In the first part of this project, an external web cam was used to provide the images over which the Augmented Reality would be placed, and a PC to do the processing. In the second, an Android run mobile telephone with it's integrated camera does all the work.

(a) BQ Aquaris E5 HD

(b) Logitech WebCam Pro 9000

Figure 3.1: Hardware used during the development of this project

The web cam used is a *Logitech WebCam Pro 9000*, which is connected to the PC through a USB drive. Its usual frame rate is 30 fps, and can support several different camera and video resolutions (320x240, 640x480, 800x600, from others).

The PC used for the development and testing of the software is a *Intel (R) Core (TM) i7 3537U @ 2.00 Ghz.* The operating system used during the development was *Ubuntu 12.04 LTS (Precise Pangolin)*, an extensively used GNU/Linux distribution, available for several computer architectures.

The Android run mobile telephone used for the second part of this project is a *Quad Core Cortex A7 @ 1.3 GHz, BQ Aquaris E5 HD*, running *Android 4.4.2. (Kit Kat).* It includes two integrated cameras for the image capturing.

3.2 JdeRobot Framework

JdeRobot [1] is a robotics and computer vision development platform, which was originally developed thanks to a doctoral thesis [1]. JdeRobot is a middle-ware that provides support to several different sensors and actuators. It is mainly developed in C++ and gives the user all the necessary tools for the communication between different components, that do not necessarily have to be developed in the same programming language nor run under the same architecture. This middleware is supported and continuously under development by the members of the *Robotics Group* in the *Rey Juan Carlos University (URJC).*

JdeRobot uses ICE for the communication between its different components with TCP/IP connexions. The mentioned components can extract information from different robot sensors, or can provide this information for other components to use. In JdeRobot framework, a robotic application is composed of several interacting components, and the communication between these is done thought *ICE.*

Within JdeRobot, there have been plenty of drivers developed to support several physical sensors and adapters, as well as the gazebo simulator. These are very useful when working and processing information provided by sensors in your new JdeRobot component. The encapsulation level that JdeRobot uses allows users to interact with several different components at the same time, modularize the tasks to perform and organize your code in a very easy manner.

For this project, I have used version 5.2 of JdeRobot. In the next subsections, I am going to describe the components and libraries that have been used for the development of this project.

[1]http://JdeRobot.org

3.2.1 CameraServer Component

CameraServer is a *JdeRobot* component that, as its own name says, "serves" a camera flow for other JdeRobot components to receive and use the images. It uses ICE interfaces to provide the image flow on a certain IP and Port, for others to listen and capture the images, so that these can be used by others.

One of the facilities that Camera server provides us, is a unified interface for the reception of images in our application, as well as the use of ICE, which allows the developer to forget about low level connections between components.

In this concrete project, we use two instances of cameraserver, one used to serve the camera images that we are processing, and one serving the film frames to project in our Augmented Reality. This allows a great simplification in our code, as the reception and saving of the frames for both flows is identical and can be reused.

3.2.2 Progeo Library

Progeo (Projective Geometry) is a JdeRobot library that bases on Richard Hartley and Andrew Zisserman's [4] projective geometry calculations to provide the users with an *easy-to-use* set of functions to resolve projective geometry problems. Within these, we must stand out both the *project* and *backproject* functions. With the *project* function, we can project a 3D point in space within a 2D image, knowing the parameters, both intrinsic and extrinsic, of the camera that will be capturing the image. On the other hand, the *backproject* function cannot determine the exact 3D point that corresponds to a position on an image, but can give us a 3D point which, if we join it with the camera focus, will give us a ray that contains the 3D point represented in the image. This is very useful to obtain the 3 dimensional absolute coordinates of a certain point detected in an image, or to project to an image a point in our real world.

In order to use this library, we must previously calibrate the camera to obtain the exact intrinsic camera parameters to take in to account when projecting. For more reference, please consult Chapter 4, as there is an detailed explanation of the Pin Hole Camera model and its parameters.

For our augmented reality component, *Progeo* has been used to project the 3D points within which the augmented reality should be in spatial world to the image shown on the screen, using both the camera's extrinsic parameters (calculated estimating the camera pose) and intrinsic (given with the camera calibration).

3.3 ICE

ICE (The **Internet Communications Engine)**[2] is a modern object-oriented toolkit that enables you to build distributed applications with minimal effort. Ice takes care of all interactions with low-level network programming interfaces, such as opening network connections, serializing and de-serializing data for transmission or retrying failed connection attempts.

Within JdeRobot environment, all communications between components are done via ICE, which helps them to be more portable and interchangeable. Being a JdeRobot component, this project uses ICE version 3.4 to communicate with other JdeRobot components used, such as *CameraServer*, described previously, or *Gazebo* plugins to help the interaction with our simulated world.

3.4 Gazebo Simulator

Gazebo [3] is a multi-robot simulator for both outdoor and indoor environments. It is able to simulate several robots with their sensors, in a 3D world. This world can be personalized by the user, thanks to the tools provided for the integration with 3D models designed with SDF, and the models that *Gazebo* includes (*TurtleBot* or *Pioneer2*, from others).

This simulator in under constant development, under the Open Source Robotics Foundation. It is also able to simulate physical interactions between objects in the 3D world, thanks to its integration with physics libraries, such as Bullet or ODE.

Gazebo also provides us with an extensive API, which is very well documented, to allow the users to develop plugins for their personalized robots and sensors. It also guarantees high quality graphics in the simulations, thanks to the use of *OGRE* for the texture rendering.

The version of Gazebo used in this project is 1.8.1., and has been used to simulate a RGB camera in a virtual world, for the first tests of the developed component. It was

[2]https://zeroc.com/6
[3]http:://gazebosim.org

also used for a plugin development to move this camera within the virtual world with a teleoperating component.

3.5 OpenGL

OpenGL (Open Graphics Library) [4]is a standard specification that defines a multi-platform API for the development of 2D and 3D graphics. This API is used as a communication interface between your application and the graphics processing unit, in order to optimize the graphic rendering. It can be used to build very complex 3D scenes using, as a base, simple geometric shapes, such as points and lines. Thanks to this, OpenGL is used very commonly in Computed Aided Design (CAD), Virtual Reality, Flight Simulation and Video Games.

(a) Rage (b) Minecraft

Figure 3.2: Video Games with OpenGL Graphics Rendering

Basically, OpenGL is an interface, a document that describes a group of functions and the exact behaviour that these should have. Parting from this, each hardware developer can create their own implementations, libraries of functions that adjust to the given API. In order to be able to use the official OpenGL logo, and qualify the library as *OpenGL friendly*, these must go through certain tests that certify the compliance with the original development. OpenGL is supported on every major operating system and works with most major windowing systems.

In this concrete project, OpenGL is used to render both the image captured by a camera, and to project over that image a film in order to provide the Augmented Reality.

[4]https://www.opengl.org/

3.6 OpenCV

OpenCV (Open Source Computer Vision Library) [5]is an BSD-licensed open source computer vision library, originally developed by *Intel Russia research centre*, and now supported by *Willow Garage* and *Itseez*.

The library includes more than 2500 optimized algorithms that can be used to detect and recognize faces, track camera movements and extract 3D models of objects, between others. The library is used extensively in companies, research groups and by governmental bodies.

It has C++, C, Python, Java and MATLAB interfaces and supports Windows, Linux, Android and Mac OS. In this project, OpenCV 2.4.10 is used in the capture and processing of the camera images, prior to the tag detection, and is also used to convert rotation vectors to matrix's for the camera pose estimation.

3.7 AprilTags

AprilTags [6]is a visual fiducial system, useful for a wide variety of tasks including augmented reality, robotics, and camera calibration. AprilTags are conceptually similar to QR Codes, in that they are a type of two-dimensional bar code. However, they are designed to encode far smaller data payloads (between 4 and 12 bits), allowing them to be detected more robustly and from longer ranges.

The AprilTag detection software detects any April tags in a given image, providing the unique ID of the tag as well as its location in the image. If the camera is calibrated and the physical size of the tag known, also provides the relative transformation between tag and camera. Implementations are available in Java, as well as in C.

For this project, AprilTags has only been used for the detection of tags in an image, obtaining the tag ID and the location (height and width pixel) in the given image. We have not used April Tags' functionality to estimate the camera position in respect from the absolute tag position. We have used another library that gives us this functionality, and that we will mention next.

[5]http://opencv.org/
[6]http://april.eecs.umich.edu/wiki/index.php/AprilTags

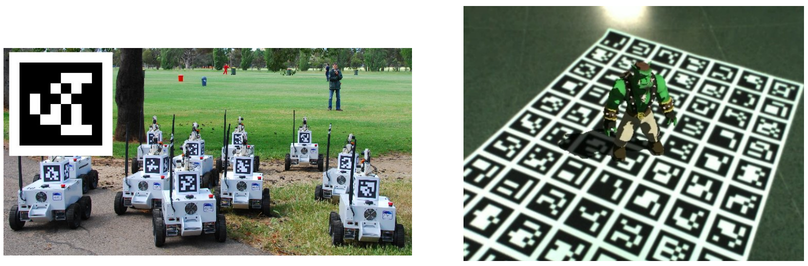

(a) April Tags Robots (b) ARuCo Tag used for Augmented Reality

Figure 3.3: Different 2D Tags: April Tags and ARuCo

3.8 Aruco

ArUco(Augmented Reality Universidad de Cordoba) [7]is a minimal library for Augmented Reality applications based on *OpenCV*. Similar to AprilTags, it relies on black and white 2D markers with codes that are detected by the library.

It is a fast, reliable and cross-platform library developed by the *University of Cordoba*, with approximately 1024 different tags to be used for detection. If the camera used for the detection has been calibrated, and we know the tags size, we can also estimate the camera position from the tag, the same as happens with *April Tags*.

In this project, we have used ArUco 1.2.5 for the camera pose estimation, using as a base the tag detection from AprilTags, explained previously.

3.9 Android

Android[8] is a mobile operating system, originally developed by a company named *Android Inc.*, which was bought by *Google*, its current maintainers. Android is based on a long term support Linux kernel. Currently, the version of this kernel is either 3.4 or 3.10, but has used several different ones since they started with version 2.6.35 in Android 1.0.

Android's Linux kernel has been slightly modified, in order to allow the OS to interact with the difficulties derived from use in mobile terminals. From others, it includes several new *Out Of Memory (OOM)* handlers and a *Logging* class.

The wide range of smartphones that run Android and the fact that is it open code, make this platform a great one to develop applications for, and distribute them, as well as

[7]http://www.uco.es/investiga/grupos/ava/node/26
[8]https://www.android.com/

the extensive documentation and tools available.

We are now going to describe the main components that are present in Android operating system:

- **Application.** Android includes plenty of applications by default with its system, such as contact list, messaging or a web browser. We can also get hold of more applications via Google Play, Android's application store. All android applications are written in Java.
- **Application Framework.** Android developers have access to the same tools from the Android framework used by the base applications. These are also designed in order to simplify the re-usability of the components, therefore any android application can announce its capabilities, for others to make use of them.
- **Libraries.** Android includes a set of C and C++ libraries that are used in several system components. The functions included in these are available to developers via the application framework. Some of the most important ones are the System C library, an implementation on the standard C library; media and graphics libraries or SQLite. Android also provides a good base of libraries that allow the developer to access most of the functions available in the standard Java libraries
- **Android Runtime.** Each Android application runs its own process, with its instance of *Dalkiv* Java virtual machine. *Dalkiv* has been designed in order for it to be run in several virtual machines efficiently. It executes files with its own executable format (.dex), which is optimized to use as little memory as possible.

3.9.1 Android Application Fundamentals

When developing an Android Application, we have several different components that can be developed, depending on the task we wish to perform. We can generally differentiate four main components:

- **Activity.** An *Activity* is a component that represents a single user interface screen, as well as handling the user interactions with the terminals screen. This component is the most used within Android, seen as it is necessary for the user to be able to make use of the application.

If we take the messaging application as an example, we can clearly differentiate two activities. The first, lists all the messages received, and a second activity that is called when the user wishes to read one of the messages, which would give all the message information. When we wish to navigate between different activities, the one we leave behind is left in a paused state, and will not return from this state until the activity is called again.

In the developed Android application, we have used 2 activities, the first for the user to configure some details needed for the projection, and a second that takes care of the camera and film projection.

- **Service.** A *Service* is an Android component that runs in the background, and as such, has no user interface. These components can run in the background of our terminal for an undefined amount of time, and have the possibility to call an activity providing it with information when certain conditions occur. For an example, we can take a service that is constantly checking a certain news feed, and informs the user when there is a new article regarding his city. Meanwhile, the user can keep using his smartphone as usual.

  In our application, we make use of a service that processes the image given from the camera, and estimates the 4 points between which to project the film.

- **BroadCast Receiver.** A Broadcast Receiver's function is to respond to broadcast messages originating from another application, or even from the telephone's system. We could use a broadcast receiver to act within our application and release memory when the systems sends a message informing that the terminal is running low on memory.

  In our case, we use this functionality to receive the information given by our image analysing service.

- **Content Provider.** This component is used to supply data to another applications that requests it. They encapsulate the data, and provide mechanisms for defining data security. Android's system includes content providers to access certain data such as audio, video or contact information.

  Our Android application uses a content provider to obtain the list of videos saved in the telephone's SD card, and allows the user to choose which film he wishes to project.

3.9.2 Java Native Interface / Native Development Kit

JNI (Java Native Interface)[9] is a development environment that provides our Android application, running within its Java Virtual Machine, to call functions developed in other programming languages, such as native C/C++.

JNI is used in the scenarios when we need to perform certain activities in our application that need native C/C++ code to be able to fulfil them. There are several classes in Java's standard API that make use of JNI in order to provide the developer the required tools, such as the access to file systems. It is also used when high performance arithmetic calculations are required, due to the fact that native code is generally faster than the code run in JVM.

JNI has been very important for the development of our Android application, as it allowed us to reuse the camera localization algorithm developed for our PC application, with no need to rewrite this in Java. It also enables us to use exactly the same code used on PC in Android, which allows us to be able to compare the performance in a fair way.

3.9.3 OpenGL ES

OpenGL ES (OpenGL for Embedded Systems) [10] is a cross-platform simplified version of the OpenGL API to provide 2D and 3D graphics on embedded systems, such as consoles or phones. It consists of well-defined subsets of desktop OpenGL, creating a flexible and powerful low-level interface between software and graphics acceleration.

This standard 3D graphics API makes it easy to offer a variety of advanced 3D graphics across most major mobile and embedded platforms. Since it is based on OpenGL, no new technologies are required, and this ensures a simple migration to and from you desktop OpenGL application.

We have used OpenGL ES 2.0 to perform the film projection over the camera preview on our application.

[9]http://developer.android.com/training/articles/perf-jni.html
[10]https://www.khronos.org/opengles/

Chapter 4

Theoretical Fundamentals

In this chapter, we describe some of the fundamental mathematical procedures and techniques that have been used during the development of this project. More in detail, we shall explain some projective geometry notions, as well as different ways to represent an objects orientation in a 3D world.

4.1 Pin Hole Camera Model

Great part of the work on this project has been related to camera models and their projections. In order to estimate the position of a camera, parting from an object with a known position and its representation on an image from the mentioned camera, we must have a very clear idea on the geometric model the camera uses to capture the image.

The *Pin Hole Camera Model* bases itself on a modelling where the projection is conical, meaning that all the light rays, at some point, go past one unique point, the camera focus. With this theory, the light from a scene passes through this pin hole and projects an inverted version of the scene on the other side of this pin hole.

For this model to be applicable, the used cameras should not be able to have a lens, as this would invalidate the supposition, but modern cameras allow us to use this model regardless of the lens. In figure 4.1 we can see a simplified version of the Pin Hole Camera model, which includes the image plane in front of the focus, instead of behind as would be the real case. This model is very useful thanks to its simplicity and precision in camera modelling.

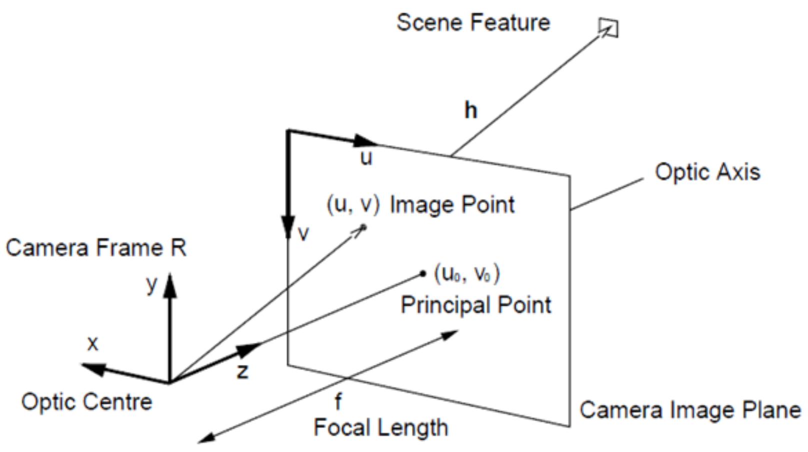


Figure 4.1: Pin Hole Camera Model

4.1.1 Intrinsic Parameters

An ideal camera with its focus located on the origin of coordinates, and orientated in the positive Z axis, would project 3D points to an image plane following the next equation, where f represents the focal distance (distance from the focus to the image plane):

$$\begin{pmatrix} u \\ v \end{pmatrix} = f \begin{pmatrix} -\frac{x}{z} \\ -\frac{y}{z} \end{pmatrix} \tag{4.1}$$

The origin of coordinates of images that are saved on computer systems is usually located on the top left corner of the image, so if the image has dimensions $m \times n$ the equation would be as follows.

$$\begin{pmatrix} u \\ v \end{pmatrix} = \begin{pmatrix} \frac{m}{2} \\ \frac{n}{2} \end{pmatrix} + f \begin{pmatrix} -\frac{x}{z} \\ -\frac{y}{z} \end{pmatrix} \tag{4.2}$$

This equation should be considered as our ideal pin hole model. In a more realistic model, the focal length is generally different for the x axis then for y, and even in the case of an ideal lens, unless this is perfectly aligned with the origin of coordinates, our optical centre could never be $\begin{pmatrix} \frac{m}{2} \\ \frac{n}{2} \end{pmatrix}$. We would have to use a generic point $\begin{pmatrix} u_0 \\ v_0 \end{pmatrix}$ as our optical centre. Including the mentioned parameters to our model, the expression would be the following:

$$\begin{pmatrix} u \\ v \end{pmatrix} = \begin{pmatrix} u_0 \\ v_0 \end{pmatrix} + \begin{pmatrix} f_x & 0 \\ 0 & f_y \end{pmatrix} \begin{pmatrix} -\frac{x}{z} \\ -\frac{y}{z} \end{pmatrix} \tag{4.3}$$

It is a good practice to express these geometric expressions in homogeneous coordinates, which enables us to represent points at infinity using standard notation for finite coordinates. If we do so for the mentioned transformation, expression 4.3 would result in the following:

$$\lambda \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & 0 & 0 & u_0 \\ 0 & f_y & 0 & v_0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ H \end{bmatrix}_{cam} = K \begin{bmatrix} X \\ Y \\ Z \\ H \end{bmatrix}_{cam} = KP_{cam} \tag{4.4}$$

In expression 4.4 we can differentiate the following factors:

- **λ:** This term is present due to the fact that we are using equivalence classes and homogeneous coordinates. It is a scaling factor.
- **K:** The projection matrix, that contains the cameras focal distance and optical centre, which are the main intrinsic parameters. We have not included any distortion parameters nor skew, as modern cameras don't tend to introduce deformations.

4.1.2 Extrinsic Parameters

When working with cameras, we cannot necessarily use the parting point that it will always be located in the same place, nor orientated in the same direction. When modelling a camera we must take into account the position and orientation it has, as the image taken from it can change drastically from one point to another.

In the last expression we defined, our 3D point P_{cam} is expressed in universal coordinates, which aren't equal to the cameras. In the previous suppositions, we were taking the cameras position equal to our universal systems centre of coordinates. If we want to take into account the cameras mobile position, we would have to express the three dimensional point in the camera coordinates, so that we can use the previously calculated expressions.

To express our 3D point in the camera coordinates, we will have to perform a generic rotation and translation. These transformations can be expressed in a matrix form, called Rotation and Translation Matrices. If using homogeneous coordinates, as the ones explained previously, our RT matrix would have the following shape:

$$RT = \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{4.5}$$

With this, we can define the relation between the 3D point in both coordinates:

$$\begin{bmatrix} X \\ Y \\ Z \\ H \end{bmatrix}_{cam} = \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ H \end{bmatrix}_{w} \tag{4.6}$$

or expressed in another way,

$$P_{cam} = \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} P_w \tag{4.7}$$

If we combine the expressions calculated for both intrinsic and extrinsic parameters, we can obtain a general equation that relates any 3D point in our world to the corresponding image plane of the camera used to capture it. Obviously, there will be points that do not reflect on to the image plane, as they are out of the cameras view.

The general expression calculated is the following:

$$P_{im} = K \cdot RT \cdot P_w \tag{4.8}$$

$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix}_{im} = \begin{bmatrix} f_x & 0 & u_0 & 0 \\ 0 & f_y & v_0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}_{w} \tag{4.9}$$

In the current project, we part from the premise that the cameras are calibrated, which means that the K matrix is known, and we will calculate the camera position (RT).

4.2 Quaternions

One of the problems we have faced during the development of this project, has been the decision on the different orientation representations, and which is more optimal for the augmented reality application to be developed. After a great amount of searching and analysing, I came to the conclusion that none of these representations are perfect, and that they all have their pros and cons.

Before going in to details on the chosen model, lets give a quick description of some of the other choices available:

- **Axis Angle Representation**

  One of the most simple ways to represent the orientation of an object in a three dimensional world is using an axis that unites our position with where we are looking at, and an angle that represents the inclination that we have around such axis, following the right hand rule, as seen in Figure 4.2.

With this representation, we need 3 coordinates (x, y, z) to for a vector that represents in what direction our object is looking (ê), and another to clarify the inclination that it has(θ). This means, that with 4 values, we can fully represent our objects orientation.

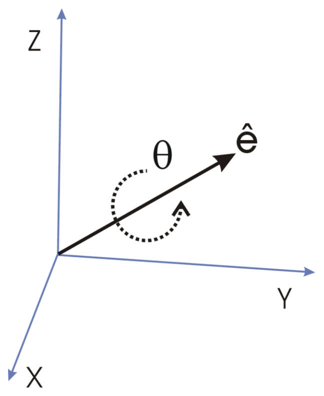


Figure 4.2: FOA and angle rotation representation

On the down side, the interaction with this kind on representation is quite complex, as when we need to move the object, we need several mathematical operations to calculate the objects new orientation, and this is not trivial

- **Roll, Pitch and Yaw**

  Another way to represent an objects orientation is using three angles, the ones corresponding to rotations around the axis that are present in a three dimensional world: Roll represents the rotation around X axis, Pitch around Y axis and Yaw around Z axis.

  With this approximation, we have a full orientation representation with 3 angle values. It was originated in aircraft physics, thanks to the need to express the movements an air-plane must perform to obtain the required route.

  In the negative points, we must mention that the order in which these 3 rotations are performed can change the final orientation result, which gives an extra complexity to the arithmetic operations. Using this representation of rotations, we are also in risk of getting a *Gimbal Lock*, which is the situation we get when, in certain rotations, we lose one degree of rotation.

- **Rotation Matrix**

  Finally, the other orientation representation that we evaluated is using a NxN matrix, being N the number of dimensions we are working with. As we are currently managing a three dimensional world, we will limit the explanation to 3D. In section 4.3 we shall see a more extensive explanation of all the parameters involved in rotation matrices.

  A three dimensional rotation can be represented by 9 values in a matrix form, that gives us a detailed view regarding the orientation of our object.

  The main fall-back for this kind of representation is the memory use for it, as not only does the rotation use 9 values, but operations with them are also quite computationally expensive.

After this quick insight to orientation representations, we must talk about the chosen option: quaternions. A **quaternion** is generally thought of as a complex number, with the peculiarity that it has three complex parts, or better said, an *hypercomplex* number, only realizable in a 4D world. This idea was first thought of by W.R. Hamilton (1805–1865) in 1843, and gives us the first example of a hypercomplex system.

The significance of these 4 terms is very similar to that of the axis angle representation of rotations. The three complex numbers give us a reference to the axis around which we shall be rotating our object, and the real part denotes the rotation we will apply around the given axis.

The basic representation of a quaternion is as follows:

$$Q = q_0 + q_1 i + q_2 j + q_3 k \tag{4.10}$$

In this representation, q_0, q_1, q_2 and q_3 can all denote any real quantities, and i, j and k represent three imaginary quantities, regarding the three complex parts of our quaternion, which must obey the following identity:

$$i^2 = j^2 = k^2 = ijk = -1 \tag{4.11}$$

The main advantage of using quaternions, in stead of the other mentioned representation, is the simplicity they provide in terms of rotation calculations. With just a few simple calculations we can calculate an objects new orientation, after having been rotated, and eliminating the risk of suffering from *gimbal lock*. Further into this chapter, we will go into more detail on how these operations are done, in order to calculate an objects orientation. Another important factor to take into account is the computational load that these operations might have in our system. As a quaternion is a set of 4 real numbers, the memory use to save them is not very high, and the operations performed (sums and multiplications) will not be a problem in a modern day computer. Quaternion representation also has ways to transform them into other representation, which helps the re-usability of code and the integration in other applications, giving the developer the option to convert to and from quaternions, depending on the other representations the application may need.

4.2.1 Properties

The quaternion set is a four dimensional vector space, based on real numbers ($H = R^4$). We can define three basic operations for the set that defines quaternions: addition, scalar multiplication, and quaternion multiplication.

The addition of several elements in H is defined as the sum of the elements that compose this set. This means:

$$Q+Q' = (q_0+q_1i+q_2j+q_3k)+(q'_0+q'_1i+q'_2j+q'_3k) = (q_0+q'_0)+(q_1+q'_1)i+(q_2+q'_2)j+(q_3+q'_3)k \tag{4.12}$$

As we can see, the addition of quaternions can be done in the same manner as the sum in complex numbers or vectors.

In the same way, the multiplication between a quaternion and a scalar number is defined as the multiplication of the scalar and each element of the quaternion:

$$Q \cdot N = (q_0 + q_1i + q_2j + q_3k) \cdot N = q_0 \cdot N + q_1i \cdot N + q_2j \cdot N + q_3k \cdot N \tag{4.13}$$

Yet again, the scalar multiplication for quaternions has no difference from the same operation applied to a standard vector, or to a complex number.

When we start to talk about the quaternion multiplication, is when start to note some differences. Thanks to the basic quaternion identities stated in equation 4.11, we can easily define the pairwise cross-product between the three unit vectors:

$$\begin{aligned} ij &= -ji = k \\ jk &= -kj = i \\ ki &= -ik = j \end{aligned} \tag{4.14}$$

As can be noted, the multiplication of these unit vectors is not commutative, and therefore, the multiplication of two quaternions won't be either. The method to do this multiplication is very similar to a general one, but we have to take into account the order of the factors in order to obtain the correct result:

$$\begin{aligned}&Q \times Q' = (q_0 + q_1 i + q_2 j + q_3 k) \times (q'_0 + q'_1 i + q'_2 j + q'_3 k) = \\ &q_0 q'_0 + q_0 q'_1 i + q_0 q'_2 j + q_0 q'_3 k + q_1 i q'_0 + q_1 i q'_1 i + q_1 i q'_2 j + q_1 i q'_3 k + \\ &q_2 j q'_0 + q_2 j q'_1 i + q_2 j q'_2 j + q_2 j q'_3 k + q_3 k q'_0 + q_3 k q'_1 i + q_3 k q'_2 j + q_3 k q'_3 k\end{aligned} \tag{4.15}$$

If we apply the product of the unit vectors defined in equation 4.14 we obtain the following result:

$$\begin{aligned}&(q_0 + q_1 i + q_2 j + q_3 k) \times (q'_0 + q'_1 i + q'_2 j + q'_3 k) = \\ &q_0 q'_0 + q_0 q'_1 i + q_0 q'_2 j + q_0 q'_3 k + q_1 q'_0 i - q_1 q'_1 + q_1 q'_2 k - q_1 q'_3 j + \\ &q_2 q'_0 j - q_2 q'_1 k - q_2 q'_2 + q_2 q'_3 i + q_3 q'_0 k + q_3 q'_1 j - q_3 q'_2 i - q_3 q'_3 = \\ &(q_0 q'_0 - q_1 q'_1 - q_2 q'_2 - q_3 q'_3) + (q_0 q'_1 + q_1 q'_0 + q_2 q'_3 - q_3 q'_2) i + \\ &(q_0 q'_2 - q_1 q'_3 + q_2 q'_0 j + q_3 q'_1) j + (q_0 q'_3 + q_1 q'_2 - q_2 q'_1 + q_3 q'_0) k\end{aligned} \tag{4.16}$$

Apart from these main operations, there are others that can be defined based on these. The most important ones to be mentioned are the following:

- **Conjugate**

  The concept of conjugate for quaternions is equal to the one applied to complex numbers. The main difference with the complex plain, is that the conjugate of a quaternion can be entirely expressed with a combination of additions and multiplications.

  $$q* = -\frac{1}{2}[q + iqi + jqj + kqk] \tag{4.17}$$

  If we expand this equation, yet again using the expressions in equation 4.14 we obtain the well known expression for a conjugate, applied to quaternions:

  $$q* = q_0 - q_1 i - q_2 j - q_3 k \tag{4.18}$$

- **Norm**

  The norm of a quaternion is what, in other scopes would be the square root of the product of a quaternion with its conjugate. This yields a non negative number, which is calculated in the same way as the norm applied to a vector:

  $$||q|| = \sqrt{qq*} = \sqrt{q * q} = \sqrt{q_0^2 + q_1^2 + q_2^2 + q_3^2} \tag{4.19}$$

A quaternion's norm is mainly used to calculate *unit quaternions*, which are quaternions with unitary norm. They are calculated in the following way:

$$U_q = \frac{q}{||q||} \tag{4.20}$$

Unit quaternions are important in the representation of angles and rotations, as they allow us to use the general addition and multiplication methods that we apply within quaternions, and use these results to obtain rotations, having to just do one extra calculation in order to do these calculations.

4.2.2 Representing Rotations with Quaternions

As we have mentioned previously, a quaternion is a way to represent rotations which is composed by three complex numbers and a real part, totalling 4 numbers. We have also mentioned that unit quaternions provide the user with a very useful mathematical notation for representing the orientation and rotation of an object in a three-dimensional world. Because of these reasons, quaternions are widely used in computer graphics, robotics, navigation and flight dynamics, between other fields.

If we call Θ the rotation around the axis, and a_x, a_y, a_z the vector representing the axis, our members of the quaternion have the following definition:

$$\begin{aligned} q_0 &= cos\left(\frac{\Theta}{2}\right) \\ q_1 &= a_x \cdot sin\left(\frac{\Theta}{2}\right) \\ q_2 &= a_y \cdot sin\left(\frac{\Theta}{2}\right) \\ q_3 &= a_z \cdot sin\left(\frac{\Theta}{2}\right) \end{aligned} \tag{4.21}$$

With this definition, we can now start to use quaternions to perform rotations. If we have a quaternion with the absolute rotation of an object, defined in the same way as the previous, and a quaternion with the rotation we wish to perform on our objects rotation, we should use the following formula: $newOrientation = rotation * orientation$, using the quaternion multiplication description in 4.16. Using matrices, we obtain the following:

$$\begin{bmatrix} cos\left(\frac{\Theta^{new}}{2}\right) \\ a_x^{new} \cdot sin\left(\frac{\Theta^{new}}{2}\right) \\ a_y^{new} \cdot sin\left(\frac{\Theta^{new}}{2}\right) \\ a_z^{new} \cdot sin\left(\frac{\Theta^{new}}{2}\right) \end{bmatrix} = \begin{bmatrix} cos\left(\frac{\Theta^{rot}}{2}\right) \\ a_x^{rot} \cdot sin\left(\frac{\Theta^{rot}}{2}\right) \\ a_y^{rot} \cdot sin\left(\frac{\Theta^{rot}}{2}\right) \\ a_z^{rot} \cdot sin\left(\frac{\Theta^{rot}}{2}\right) \end{bmatrix} \begin{bmatrix} cos\left(\frac{\Theta^{pos}}{2}\right) \\ a_x^{pos} \cdot sin\left(\frac{\Theta^{pos}}{2}\right) \\ a_y^{pos} \cdot sin\left(\frac{\Theta^{pos}}{2}\right) \\ a_z^{pos} \cdot sin\left(\frac{\Theta^{pos}}{2}\right) \end{bmatrix} \tag{4.22}$$

As we can see, thanks to quaternions, in a relatively simple way we have performed a complex operation such as rotating from one orientation to another in 3 different axis.

Obviously, quaternions are not the most simple representation, from the users point of view. It would be quite complicated to ask an application user to describe the orientation of an object in quaternions. In order to solve this, we are going to describe the conversions between roll, pitch and yaw, and quaternions.

Quaternions and Roll, Pitch and Yaw

As mentioned, we can convert quaternions to and from yaw, pitch and roll format, in case it is needed during our application. These equations can be very useful, as it means we can obtain our rotation data in a simple way, such as yaw, pitch and roll, which are concepts whose values can be interpreted easier, and use a conversion to quaternions to do the complicated operations that would be very costly if we wished to do them without conversion.

To obtain the 4 quaternion factors from yaw, pitch and roll, we would have to apply the following equations:

$$\begin{aligned} q_0 &= cos\left(\frac{roll}{2}\right) cos\left(\frac{pitch}{2}\right) cos\left(\frac{yaw}{2}\right) + sin\left(\frac{roll}{2}\right) sin\left(\frac{pitch}{2}\right) sin\left(\frac{yaw}{2}\right) \\ q_1 &= sin\left(\frac{roll}{2}\right) cos\left(\frac{pitch}{2}\right) cos\left(\frac{yaw}{2}\right) - cos\left(\frac{roll}{2}\right) sin\left(\frac{pitch}{2}\right) sin\left(\frac{yaw}{2}\right) \\ q_2 &= cos\left(\frac{roll}{2}\right) sin\left(\frac{pitch}{2}\right) cos\left(\frac{yaw}{2}\right) + sin\left(\frac{roll}{2}\right) cos\left(\frac{pitch}{2}\right) sin\left(\frac{yaw}{2}\right) \\ q_3 &= cos\left(\frac{roll}{2}\right) cos\left(\frac{pitch}{2}\right) sin\left(\frac{yaw}{2}\right) - sin\left(\frac{roll}{2}\right) sin\left(\frac{pitch}{2}\right) cos\left(\frac{yaw}{2}\right) \end{aligned} \tag{4.23}$$

On the other hand, we can also represent our rotation in yaw, pitch and roll format after doing the desired operations with quaternions, converting from the quaternion we have obtained to our original angle format. To do so, we have the following operations:

$$\begin{aligned} roll &= atan\left(\frac{2(q_0q_1 + q_2q_3)}{q_0^2 - q_1^2 - q_2^2 + q_3^2}\right) \\ pitch &= asin\left(2(q_1q_3 - q_0q_2)\right) \\ yaw &= -atan\left(\frac{2(q_0q_3 + q_1q_2)}{q_0^2 + q_1^2 - q_2^2 - q_3^2}\right) \end{aligned} \tag{4.24}$$

4.3 Rotation-Translation Matrices

In this project we have already introduced RT matrices, when talking about the pin hole camera model. In that case, we used then to describe the position and orientation the camera has, in order to be able to project objects in our world towards an image plane. This manner to represent the position and orientation of an object is not only used for cameras, but is very extended in order to express an objects absolute position and orientation. We have seen how we can use quaternions to express and calculate an objects orientation, but when we use both position and orientation, we need more data to represent all of this, that quaternions cannot give us.

We tend to talk about rotation translation matrices as just one thing, but they are actually two separate ones, a rotation matrix, such as the ones described at the beginning of this chapter, and a translation matrix, that gives us the position of the objects. They are both defined as follows, using homogeneous expressions:

$$R = \begin{bmatrix} r_{11} & r_{12} & r_{13} & 0 \\ r_{21} & r_{22} & r_{23} & 0 \\ r_{31} & r_{32} & r_{33} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{4.25}$$

$$T = \begin{bmatrix} 1 & 0 & 0 & X \\ 0 & 1 & 0 & Y \\ 0 & 0 & 1 & Z \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{4.26}$$

If both these expressions are multiplied, we will obtain the formula described in equation 4.5, with the following equivalences:

$$\begin{aligned} t_x &= r_{11}X + r_{12}Y + r_{13}Z \\ t_y &= r_{21}X + r_{22}Y + r_{23}Z \\ t_z &= r_{31}X + r_{32}Y + r_{33}Z \end{aligned} \tag{4.27}$$

In these expressions, t_x, t_y and t_z give us the position of an object from a reference point given, and with the rotation values, we describe the orientation of the object from an initial rotation. To understand what the values of the rotation matrix mean, we have to go a little deeper. We will start by 2D rotations following the right hand rule, where we can describe a rotation matrix in the following way:

$$R(\theta) = \begin{bmatrix} cos(\theta) & -sin(\theta) \\ sin(\theta) & cos(\theta) \end{bmatrix} \tag{4.28}$$

In the case we wish to perform this rotation following the left hand rule, we would just have to change the sign to all the $sin(\theta)$ elements.

If we wish to move a 2D point x, y according to this rotation, we can represent it with the following relation of matrices:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} cos(\theta) & -sin(\theta) \\ sin(\theta) & cos(\theta) \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \tag{4.29}$$

With this, our resulting point would be described as:

$$\begin{aligned} x' &= xcos(\theta) - ysin(\theta) \\ y' &= xsin(\theta) + ycos(\theta) \end{aligned} \tag{4.30}$$

If we wish to do this same operation on a three dimensional space, we have different equation, one for the rotation on each axis. If we have rotations around more than one axis, we would have to perform each rotation, and then multiply these to obtain the final rotation, always taking into account that, the order in which we do the mentioned multiplication can change our final result.

The three simple rotations in a three dimensional space, have an equal representation as that we have just described for a two dimensional space, but taking into account the axis that are involved:

$$R_x(\theta) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & cos(\theta) & -sin(\theta) \\ 0 & sin(\theta) & cos(\theta) \end{bmatrix} \tag{4.31}$$

$$R_y(\theta) = \begin{bmatrix} cos(\theta) & 0 & sin(\theta) \\ 0 & 1 & 0 \\ -sin(\theta) & 0 & cos(\theta) \end{bmatrix} \tag{4.32}$$

$$R_z(\theta) = \begin{bmatrix} cos(\theta) & -sin(\theta) & 0 \\ sin(\theta) & cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{4.33}$$

With these three rotations, we can combine them to obtain our full 3D rotation. As I mentioned previously, the order in which we perform this multiplication is relevant to the final solution we obtain. The general convention regarding this order, is to apply them in the same order that an air-plane would. This is, first we rotate around Z (before the aircraft has taken off), then we rotate around Y in order to leave the ground, and finally around X to orientate the plane in the correct direction. Applying this convention, we would obtain the following values for each member of the equation 4.25:

$$\begin{aligned}
r_{11} &= cos(\alpha)cos(\beta) \\
r_{12} &= -sin(\alpha)cos(\gamma) + cos(\alpha)sin(\beta)sin(\gamma) \\
r_{13} &= sin(\alpha)sin(\gamma) + cos(\alpha)sin(\beta)cos(\gamma) \\
r_{21} &= sen(\alpha)cos(\beta) \\
r_{22} &= cos(\alpha)cos(\gamma) + sin(\alpha)sin(\beta)sin(\gamma) \\
r_{23} &= -cos(\alpha)sen(\gamma) + sin(\alpha)sin(\beta)cos(\gamma) \\
r_{31} &= -sen(\beta) \\
r_{32} &= cos(\beta)sin(\gamma) \\
r_{33} &= cos(\beta)cos(\gamma)
\end{aligned} \tag{4.34}$$

With RT matrices we can represent the movement an object, or the camera itself, has done from the origin of the world it is in. Parting from the origin of the world begin at position $[0, 0, 0]$, and orientated looking towards the positive X axis, an RT matrix fully describes the movement an object has done to go from this origin position to its new localization.

This functionality can be used to obtain the position of an object in a coordinate system different to the one his RT is represent from, knowing the conversion from one another. In order to use an example related to this project, we are going to use a robot that has a camera located on a certain part of this robot. The RT of the camera is expressed from the origin of its coordinates system, which would be the robots feet, for instance. If we know the position the robot has from the world it is walking around, and the position of the camera from the robot, we can use RT's product to calculate the position of the camera from the world, and use this RT in the camera's pin hole camera model.

$$RT_{cam-world} = RT_{robot-world} * RT_{camera-robot}$$

Chapter 5

Software Development

In this chapter, we will give a detailed explanation on the developed augmented reality JdeRobot component. This application, named AR Cinema, includes both augmented reality and auto-localization, in order to provide a good user experience and a realistic representation. We will start the chapter with a general design of the component, and the different data exchanges done, to then go into detail on each element of the application. In addition, we shall also detail the steps taken to build an Android version of this application, trying to reuse as much code as possible, in order to be able to compare the performance on different frameworks.

5.1 Design Overview

The development of the application AR Cinema has several parts worth mentioning. We have to describe the inputs and outputs of the whole application, as well as detailing the different modules inside the component.

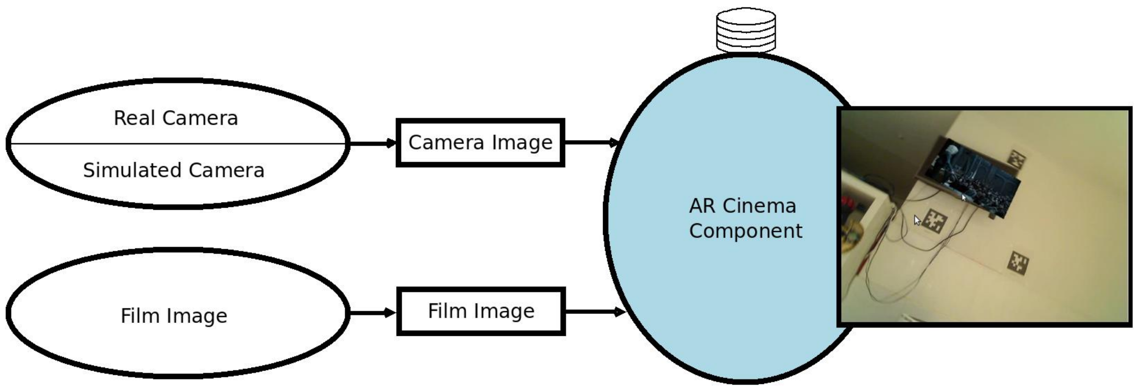


Figure 5.1: Global Schema

In Figure 5.1 we can see a diagram with the inputs and outputs of AR Cinema. As we can see, the application receives two image flows, one that comes from the camera that is capturing images, and will the one to be analysed, and another that provides a film feed, which will be used to project our virtual film over the camera image, as well as a series of data values that will describe the position of the tags, and between which points the film projection must be done. Both the images and the initial scene data is provided to the application via ICE.

If we go into more detail, we can see the application has two main parts that must be very well differentiated in order to have a full understanding of the objectives of this application. First, we have the visual auto-localization module, that determines the position of our mobile camera in the 3D world and in real time, and provides us with the position of the 4 corners between which we wish to project our virtual cinema. And second, and AR component that is in charge of projecting the virtual cinema. As well as these main parts, we also have a module which saves shared information for both of these to access, as can be see in Figure 5.2

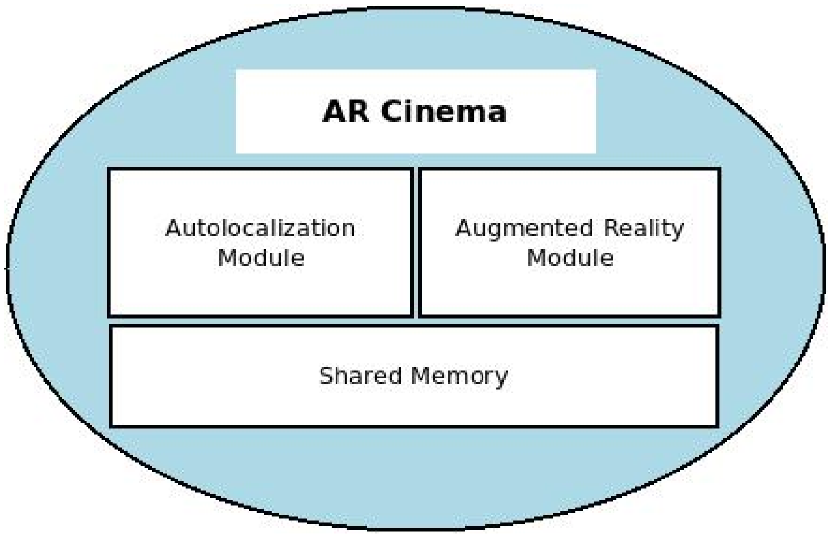


Figure 5.2: Design of AR Cinema

During this chapter, we will describe both the modules developed, going into a detailed explanation on all the main steps taken during the development, accompanied by images and references. To end, we will also describe the steps taken to migrate the developed modules to Android platform.

5.2 Visual Auto-Localization

As its name well says, the main function of this module is to determine a mobile camera's position and orientation in a 3D world, using as only input to determine this position the information it has on the disposition of the world, and the images that the camera captures. These images are analysed in real time to determine the cameras position, which means our module must be light enough to be able to execute and obtain real time results.

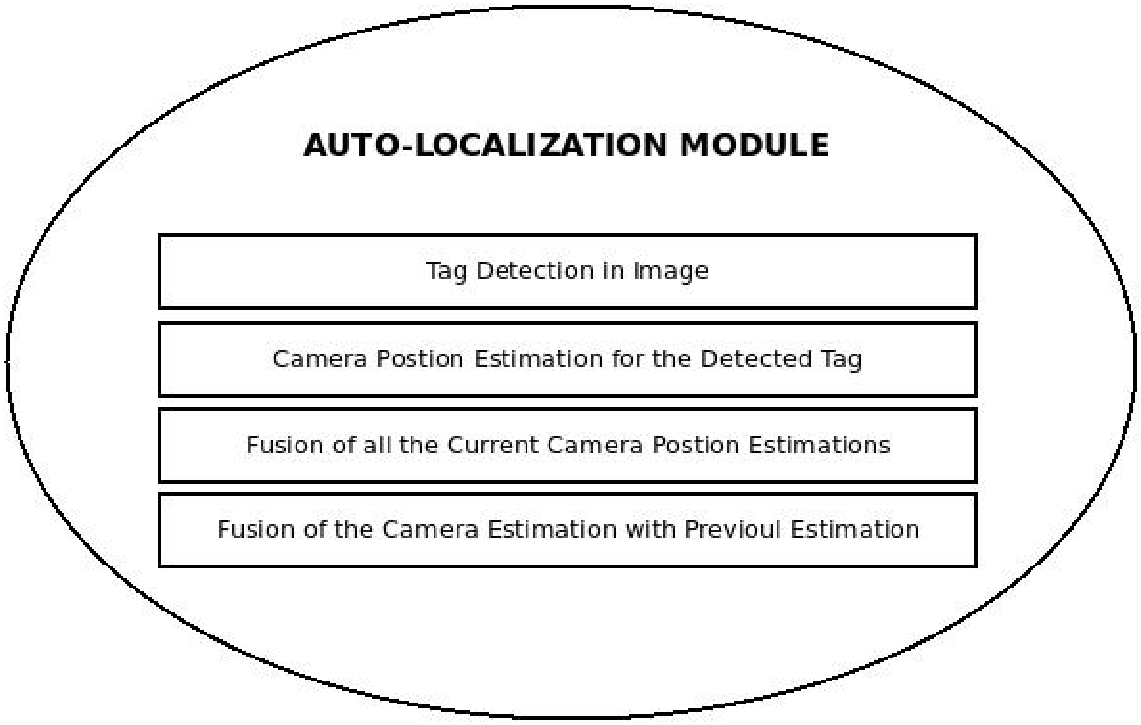

AUTO-LOCALIZATION MODULE
Tag Detection in Image
Camera Postion Estimation for the Detected Tag
Fusion of all the Current Camera Postion Estimations
Fusion of the Camera Estimation with Previoul Estimation

Figure 5.3: Auto-Localization Module Diagram

The general structure of this module is illustrated in Figure 5.3, with the main functional blocks that divide the module:

1. **Marker Detection**. Using April Tags library, we detect the tags present in the image provided by the camera, and the image position of these.
2. **Estimate Camera Position**. With the localization of each tag on the image, ARUCO allows us to estimate the position and orientation of the camera, using as a base the tag position. As we also know the position of each tag in our world, we calculate the absolute camera position.

3. **Fusion of Several Estimations**. After calculating the estimated camera position provided by each tag in the current image, we perform a weighted fusion of the all the camera positions estimated, in order to obtain a more robust estimation.

4. **Temporal Fusion of Estimations**. In order to avoid sudden jumps in the camera position, we perform a fusion of the estimated position with the previously calculated one.

5.2.1 Marker Detection

To perform the visual marker detection in our auto-localization module, April Tags were chosen due to the great advantages that these provide us with [9]. Contrary to QR codes, which require the camera to be very close to the code in order to be able to detect and extract the information from it, April Tags is based on a visual fiducial system which encodes much less payload in the 2D codes, giving more importance to the long range detection than to the information to be saved in the codes. This allows us to detect a relatively small tag in a large image with a lot of other information, as well as being less sensitive to light and orientation than QR codes, as we can see in Image 5.4.

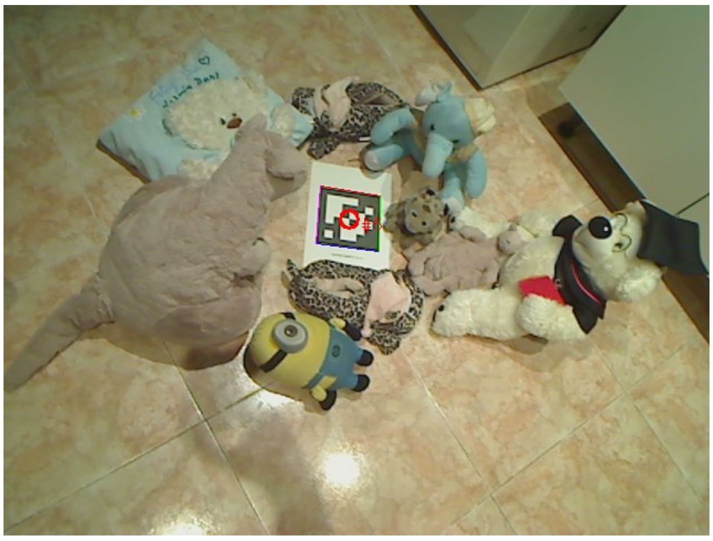
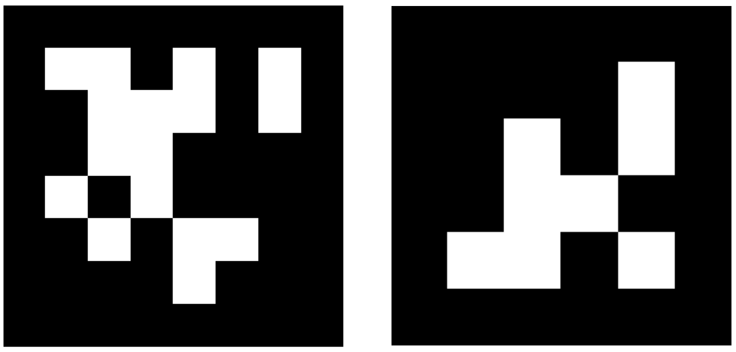


Figure 5.4: April Tags Detection

In this example, the camera is located about 1 metre away from the tag, and we can see how it marks both the centre of the tag as the perimeter with great precision.

April Tags provides us with several families of Tags, with more or less data saved, depending on the chosen family. Obviously, the more information the tags have in them, the more complicated the detection can result, but it will also be much more robust and give us less false positives, which will give our augmented reality application a better user experience.



(a) Tag Family 36h11 (b) Tag Family 16h5

Figure 5.5: April Tags Different Families

April Tags library provides us with the necessary functions in order to analyse a given image and return the information regarding the tags detected.

The first step to use April Tags library in one's code is to initialize a detector, telling it the tag family it must search for:

```
//Instanciate TagCodes and TagDetector
AprilTags::TagCodes m_tagCodes = AprilTags::tagCodes36h9;
AprilTags::TagDetector*m_tagDetector = new AprilTags::TagDetector(m_tagCodes);
```

When it detects a series of tags in an images, it will return the developer with an array of *detections*, of which the most important information is the following:

- **id**: Integer with the tag id within the family we are working.
- **p**: Array with four pairs of integers, which represent the position in the image of each corner of the tag .
- **cxy**: Pair of integers that give us the position of the centre of the detected tag.

A very basic function that prints the tag information after detecting them would be the following:

```
void processImage(cv::Mat& image){
    cv::Mat& image_gray;
    //Converto to grayscale for detection
    cv::cvtColor(image, image_gray, CV_BGR2GRAY);
    //Call detector
    vector<AprilTags::TagDetection> detections = m_tagDetector->extractTags(image_gray);
    //Print information
    cout << detections.size() << " tags detected:" << endl;
    for (int i=0; i<detections.size(); i++) {
        print_detection(detections[i]);
    }
}
```

In the code above, we can see that it is necessary to convert the given image to grey-scale before we can pass it to the April Tags detector. To do so, we make use of OpenCV and the functions it includes to perform such operations. After the detection, we can see that we have a series of detections, over which we can iterate in order to analyse each one. In our case, after the detection, we would perform the camera pose estimation for each tag.

5.2.2 Estimate Camera Position

In order to calculate the camera's position in a 3D world which has several tags to help us locate the camera, we can differentiate 2 main steps. First, we have to locate the position of the camera from the detected tag, and as we know the tag's position from the origin of coordinates of the world, we can then calculate the position of the camera from such origin, as is represented in Image 5.6.

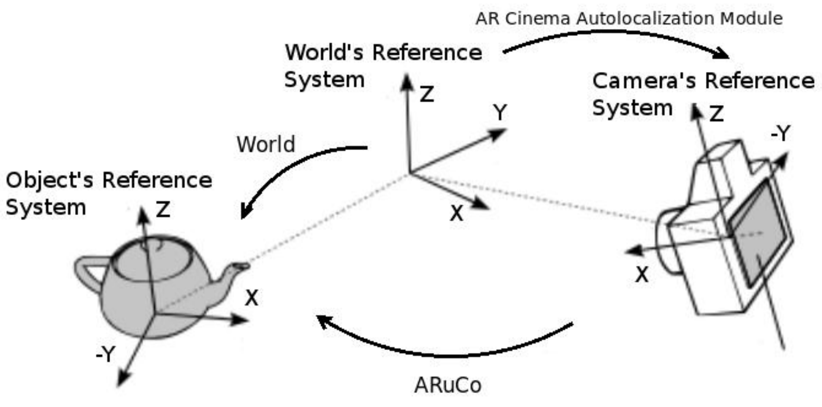

AR Cinema Autolocalization Module
World's Reference System
Z
Y
X
Camera's Reference System
Z
-Y
X
World
Object's Reference System
Z
X
-Y
ARuCo

Figure 5.6: Relation between the different objects reference systems

Camera-Tag Position

The first step to calculate the camera's global position is to obtain it using as a reference the tag detected's coordinates system. To do so, we can make use of several frameworks that allow us to determine the position of the tag, using the centre of the camera as a reference. With this information, we would only need to invert the obtained RT in order to have fulfilled our first objective.

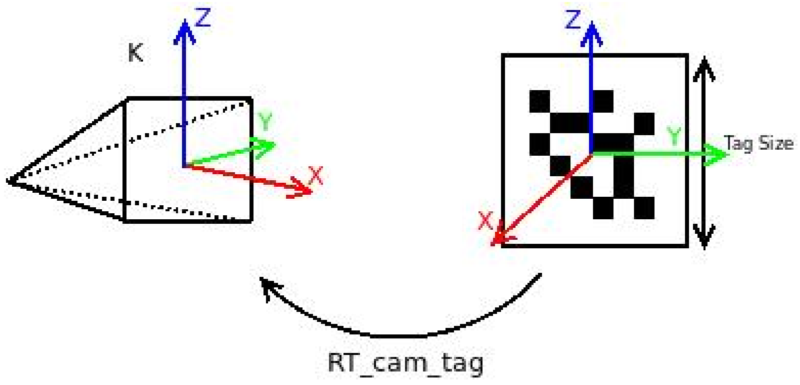
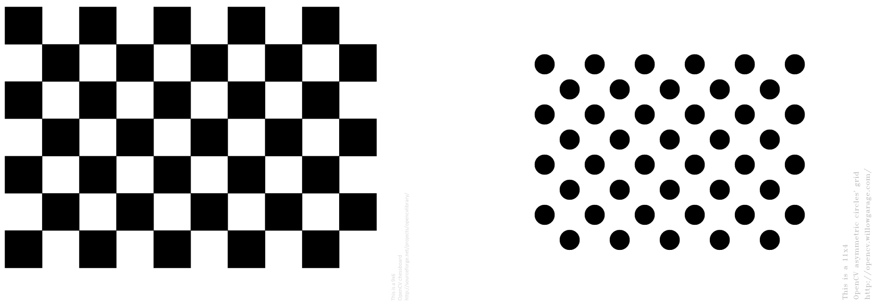


Figure 5.7: RT to calculate the camera's position, using the tag's as a reference

To calculate the position our camera has from the tag detected in the image it is capturing, we could have used April Tags, as it has the functionality to perform the opposite operation, providing the library with the tag size and certain information regarding the camera's intrinsic parameters, but we finally decided to perform these operations making use of another library, in order to test the integration between both of these.

ARuCo is a library very similar to April Tags, which has it's own families of tags and also includes a tag detector, to perform the tag detection in an image. After performing the detection of a tag in ARuCo, we also have the possibility to give the library some extra information, such as the tag size and the camera's K matrix, for it to perform a calculation of the transformation necessary to know the tag's position using the camera as reference. With this information, we would just have to invert it to obtain our first camera position.

In order to perform any position estimation, we first need to provide the application with the intrinsic parameters of the camera used. The procedure followed to determine this information is generally denoted as camera calibration, and uses a *board* with known size and pattern, such as those you can see in Image 5.8, to determine the cameras focal distance and optical centre. OpenCV has an implementation of this functionality, which can be executed both on PC and Android, each with their adapted boards. Several projects have also been done in the URJC to provide other camera calibration methods, which are also compatible with JdeRobot [3].


This is a 11x4
OpenCV asymmetric circles grid
http://opencv.willowgarage.com/

(a) PC (b) Android

Figure 5.8: OpenCV Calibration Pattern Boards

Having the camera matrix (K), we can proceed to calculate the position of the marker, taking our camera as a reference point. To do so combining ARuCo and April Tags, we have to feed ARuCo with the position of the 4 corners of the tag in the image, as calculated with April Tags. We mentioned previously that April Tags returns the developer with both the tags centre position and the 4 positions of the corners, for him to choose which information he needs. In this case, we have to provide ARuCo's *Marker* class with a vector that contains the 4 pairs of coordinates that represent our tag perimeter. With this information, along with the camera matrix and tag size, ARuCo will calculate two vectors, R_{vec} and T_{vec}, that represent the movements that the camera would have to perform in order to position itself with the same location and orientation as the tag has.

```
std::vector<cv::Point2f> realCorners;
realCorners.push_back(cv::Point2f(detection.p[3].first, detection.p[3].second));
realCorners.push_back(cv::Point2f(detection.p[2].first, detection.p[2].second));
realCorners.push_back(cv::Point2f(detection.p[1].first, detection.p[1].second));
realCorners.push_back(cv::Point2f(detection.p[0].first, detection.p[0].second));
Marker m(realCorners, detection.id);
m.calculateExtrinsics(realTag->getSideLength(), camMatrix, distCoeffs, false);
//m.Rvec contains vector with rotations
//m.Tvec contains vector with translation
```

Within this code, we must make a special mention to the ARuCo function *calculateExtrinsics*, which is in charge of calculating the tag's RT matrix, with camera's optical centre as reference. To do so, one would have to resolve the function defined in equation 4.8, which establishes a relation between a 3D point and it's projection on a 2D plane (image), using the cameras intrinsic and extrinsic parameters. In this function, we need to determine the RT, as we already have the rest of the information. In order to calculate this RT, ARuCo uses an OpenCV function named *cv::solvePnP*, which performs this exact operation.

Thanks to ARuCo, we have calculated the rotation and translation necessary to move our camera towards the detected marker. Unfortunately, what we need to do is the opposite: calculate the movement the marker has to do to place itself in the same position as the camera, in order to be able to calculate the transformation the camera has suffered if the origin of the world was the tag's centre. To do so, we need to transform the vectors given by ARuCo to a generic RT matrix, to be able to calculate the inverse of such matrix.

This problem must be tackled in two phases, first, to set the data corresponding to the *translation* and the *rotation*, and then perform the inverse of such matrix. In reference to the translation, the T vector provided by ARuCo has 3 positions, each corresponding to the movement performed in X, Y and Z. In our RT matrix, the column corresponding with the translation data would just have to be replaced with these values. When trying to convert the R vector to a matrix form, we have a few more problems. The values contained in *Rvec* correspond to the *Roll*, *Pitch* and *Yaw* that must be performed to rotate the tag's centre to the same orientation as the camera. As we have already mentioned, the order in which one executes these rotations can modify the final result, as well as having a performance problem with the great amount of computer memory needed to perform them. OpenCV, being a vision library has already got several implementations of problems similar to these. In this case, the function *cv::Rodrigues* is of great help, as it allows us to provide a rotation vector which it will transform into a 3x3 Matrix, or vice-versa. With the values contained in this 3x3 matrix, and the data in Tvec, we can obtain the RT matrix corresponding with the data that ARuCo provides. Calculating the inverse of this matrix, we can obtain our new RT which indicates the movements needed to move the point corresponding to the centre of the marker detected to the camera capturing the image. In Figure 5.9 we can see an overview of the full process followed and described.

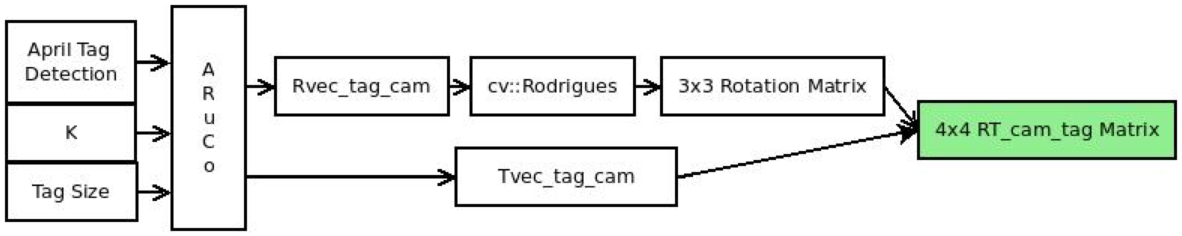


Figure 5.9: RT to calculate the cameras position, using the tag's as a reference

Camera-World Position

With the RT calculated in the previous step, we only have half of the required work done. We now have to add to the calculated RT, the transformations required to move from the origin of our world to the tag's centre. Seen as the world we are working with is known and we have full knowledge of the position and orientation of each marker in our world, we would just need to calculate the RT transformation from the centre of the world to the tag's centre, in order to add this to our previously calculated camera RT, to obtain the final result.

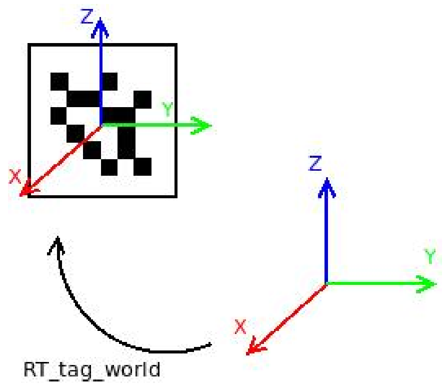


Figure 5.10: RT to calculate the tag's position, using the world reference

Each tag located in out scene has a known position and orientation, which is passed to our application via ICE when this starts. The information given to the program is the position (X, Y, Z) of the tag, and the orientation, expressed in Roll, Pitch and Yaw. Making use of *cv::Rodrigues* again, we can obtain the rotation matrix corresponding to the given values, and construct the required RT as in the previous step.

Having calculated both RT's, we can join them in order to calculate the absolute position of the camera, this is, the position of the camera using the worlds reference. To do so, we would have to apply the following formula:

$$RT_{cam-world} = RT_{tag-world} * RT_{cam-tag}$$

5.2.3 Fusion of Several Estimations

With these procedures, we have managed to obtain the cameras absolute position, taking into account the information provided for one of the tags located in our world. In order to have a more robust estimation, we have decided to include several tags in the scene used, and combine the information we obtain from them.

The fusion of several estimations is a complex problem. If calculating the mean of the different camera positions isn't much of a problem, to do so with rotations makes things a lot harder. Rotations, at the end of the day, are angles, which are circular quantities. With circular quantities, one cannot apply the general methods to calculate the mean value, as the discontinuous values would lead us to incorrect results.

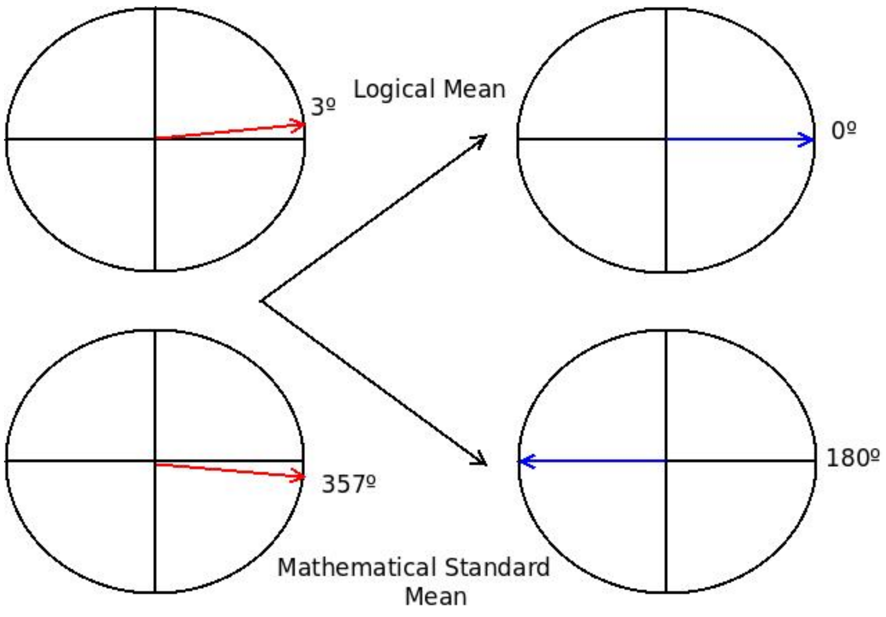


Figure 5.11: Mean of Circular Measures

When one is working with angles, they have the peculiarity that 0^o and 360^o, even being such different numbers, represent exactly the same quantity. This fact creates a discontinuity in the values of the measure, which leads to problems performing weighing operations. When calculating the mean value between 0^o and 360^o, if one understands the values that they are working with, they will know that the mean value would be 0^o (or 360^o, seen as they are thesame), but it we just apply the usual mathematical expression, we would obtain 180^o, which in this case, is the exact opposite of the value we wish to calculate. As we can see in Image 5.11, to calculate the mean value between 2 angles can be very tricky.

During this project, we faced the challenge of performing a weighing operation between all the tags detected in our image, in order to obtain a more robust estimation. To do so, the first step was to determine the weight we would give each tag, and under what conditions this weight could improve. After this, we went into the adventure of calculating the mean value of N angles, each with their own weight.

Weighing Criteria

As mentioned previously, one of the first steps taken into calculating a weighed average camera pose estimation, was determining the weight we would give each tag. The main aspects taken into account for this choice were related to te position of the tag from the camera. For one, at shorter distances we have a more robust camera pose estimation, and then, when the camera and tag had similar inclinations in all angles, our camera pose estimation was also more accurate. All these ideas have been experimentally tested, in order to determine the weight criteria we must follow for each tag, as we will see in chapter 6. In table 5.1 we can see a detailed description of these choices taken in order to provide each tag with a weight according to the importance we believe it should have, after experimentally validating different options.

Distance	**Weight**
$<$ 0.5m	5
$<$ 1m	4
$<$ 2m	3
$<$ 3m	2
$<$ 4m	1
$>$ 4m	0

Table 5.1: Weighting criteria for each tag

Mean Position and Rotation

Having determined the weight that each tag must have in our fusion algorithm, we must start to weigh the camera positions provided by each tag. This operation must be done both with the camera position as with its rotation, in order to obtain the desired results.

First, the position. To calculate a weighed mean between several positions, one can apply the general standard on calculating mean values. As so, each position would be multiplied by the weight it was given previously, and then divided by the total weight of all tags in order to obtain its portion in the final result.

With this equation applied to all tags, the only thing left in order to obtain the camera position would be to perform the sum of all these to obtain the final camera pose estimation:

$$[x, y, z] = \sum \left[\frac{[x_i, y_i, z_i] * weight_i}{\sum weight_j} \right] \tag{5.1}$$

Second, we move on to calculating the mean value for the orientation of the camera. As we have mentioned, this operation is not trivial, and needs some thinking over, before obtaining a working method.

In our case, the first approach we took was to perform the mean value of the quaternion that represents our orientation. When one performs the mean value of each component of a quaternion, and then normalizes it, theoretically, we should obtain the mean value of the rotation. After testing this alternative, we realized that such operation worked in all cases except the critical case we mentioned previously. When performing the mean value in the discontinuity area, we yet again obtained the opposite result to that we wished to obtain.

After this failed attempt, we finally managed to obtain an algorithm to calculate the mean value of two angles, without having problems in the critical parts. The main idea of this approach is based on the fact that, even though angles are discontinuous, some of their trigonometric functions are not, such as sin and cosine.We can make use of these to calculate the mean value of two angles, and a relation between the two (arctangent), in order to obtain more accuracy. Therefore, we can express the average value between two angles as:

$$Mean(\alpha, \beta) = atg \left(\frac{(sin(\alpha)ratio + sin(\beta)(1 - ratio)}{(cos(\alpha)ratio + cos(\beta)(1 - ratio))} \right) \tag{5.2}$$

When applying this function with our weight values for each estimation, we obtain an average value of all the tags detected, giving more importance to the ones we consider should be more accurate than others. In equation 5.3, we can see how this would be performed. In the mentioned equation, we have taken $weight_i$ and $weight_j$ as already being divided by the total amount of weights, in order to simplify the function slightly.

$$\alpha = atg \left(\frac{\sum(sin(\alpha_i)weight_i))}{\sum(cos(\alpha_j)weight_j))} \right) \tag{5.3}$$

5.2.4 Temporal Fusion of Estimations

With all the previous steps completed, we now have a estimation of the camera position in real time, taking into account the image provided by the camera in a precise moment.This estimation will be more or less accurate, depending on certain circumstances, such as lighting, distance from the marker or rotation from the marker, between others. We believe it was a good idea to perform, apart from the calculation of the current position using several estimations with different tags, a weighted mean calculation with the previous camera position, using as a base the fact that, in the time it takes the camera to capture another frame, the camera position cannot have changed very radically. We can use this as an advantage, in order to filter bad estimations.

When performing this mean calculation, we yet again had to decide the weighting values we would give for each tag, and then, perform the mean calculation in the same manner as the previous one.

In this case, the weight values given to the camera position depended on its similarity to the previous camera pose estimation. In addition to this, we also took into account the number of estimations we have already had, in order to determine whether the previous estimation can be considered accurate or not. The first step taken, was to perform a 50% mean between the current camera pose estimation and the previous, for the first 20 estimations, in order to obtain a reliable base estimation. From there, we tested experimentally different weight criteria, and opted finally to give more importance to the current estimation as opposed to the previous one, when the similarity between them is high, and progressively decrease the weight of the new estimation when the similarity decreases, as can be seen in table 5.2

Condition	**Position Diff.**	**Rotation Diff.**	**Weight New Pose**
All Axis and All Rotations	<0.1	<0.05	0.90
All Axis or All Rotations	<0.15	<0.1	0.80
All Axis or All Rotations	<0.15	<0.1	0.50
All Axis or All Rotations	<0.3	<0.15	0.25
Other	>0.3	>0.15	0

Table 5.2: Weighting criteria for Temporal Fusion

Having determined the weight for current and previous camera estimation, we proceed to calculate the fusion of the two, following the same procedure as in equation 5.2

5.3 Augmented Reality

The second main module in the AR Cinema application is the representation of the virtual cinema screen, between the four known points in our 3D world.

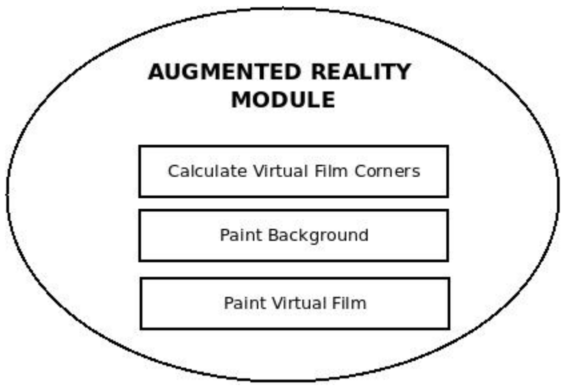


Figure 5.12: Augmented-Reality Module Diagram

The structure of this module is described in Figure 5.12, with the main blocks that compose it being the following:

- **Cinema Corners Calculation**. The first step performed by this module is the use of Progeo library to calculate the position of the corners of the virtual film, taking the position of the camera calculated previously.
- **Draw Background.** The second step would be to paint, in our OpenGL GUI window, the image from the camera as the background, in order to be able to paint over it.
- **Draw Virtual Film.** Having the background in place, we shall then continue to paint the virtual film between the corresponding pixels of the image.

5.3.1 Cinema Corners Calculation

After fulfilling all the previous objectives, we have an estimation of the camera position which we believe is reliable and valid in order to perform an augmented reality application with such camera position. The next step to perform, is to identify between which pixels of the image provided by the camera we wish to perform the augmented reality.

As we have mentioned, the final application to be developed in this project is a virtual cinema screen, which will be projected between certain fixed points of our world. Depending on the position the camera holds, these pixels will vary, and therefore must be calculated each time we have a new camera position. In order to perform this calculation, we have made use of ProGeo library described in chapter 3.2.2. This library gives us all the necessary data types and functions to perform the projection of a 3D point in our scene towards an image captured by a camera in such world.

In this project, we have developed a personalized wrapper for the ProGeo library, in order to have a easier interaction with this library, and make use of some data types that were already in use by our project. Some of the main parts to be mentioned about this wrapper are the following:

- **Camera Intrinsic Parameters.** Possibility to give ProGeo the camera's K matrix, both using a file that contains the information as giving it the data via parameters.
- **Camera Extrinsic Parameters.** Possibility to give ProGeo the cameras RT matrix, passing the position data as X, Y and Z, and the rotation information either as a quaternion, as used in the rest of our project, or as the FOA (focus of attention) and a roll angle, which is the format used internally by ProGeo.
- **Project Function.** Wrapper around ProGeo's original *project* function, initializing the data types used by ProGeo with the information passed to the function, in order to keep a high level of abstraction.
- **Backproject Function.** As in the previous case, we also developed a wrapper around ProGeo's original *backproject* function, with the same characteristics and objectives as the previous.

With our progeo wrapper class, we can easily interact with the library to obtain the pixel corresponding to the 3D points between which we wish to project our virtual film. ProGeo is based on the pin hole camera model, and uses a ray projection theory in order to calculate these pixels. Seen as the 3D points between which we shall be projecting the film are known, we simply have to pass these points to our *project* function in order for this to calculate and return the pixel corresponding to such positions.

```
//Init progeo with predefined file
progeoWrapper* prog = api->getProgeoCamera();
//Init camera RT
Pose3D* poseEstim = api->getPoseEstim();
prog->setCamRT(poseEstim);
//Variables that will contain 3D point
float X3d, Y3d, Z3d;
getData3D(X3d, Y3d, Z3d);
//Variables that will contain corresponding pixel data
float X2d, Y2d;
```

```
//Project
prog->project(tl3dx, tl3dy, tl3dz, &tl2dx, &tl2dy);
```

5.3.2 Draw Background

In order to draw an image flow as a background in an OpenGL window, the first thing one must do is to understand the different projection options we can use in OpenGL, in order to understand how to proceed with our application. Having understood this, we will then proceed to loading the camera image to an OpenGL texture, which we will then paint on the screen. Let's describe these three parts separately.

Understanding and Choosing Projections

One of the many difficulties we have experienced with application was to manage to draw an image on a full background. This is due to the fact that, by default, OpenGL uses a perspective projection, which is more similar to a camera, but also makes the task of drawing an image that takes up the whole background very difficult. We had to change the projection used to an orthographic one, in order to be able to draw such background.

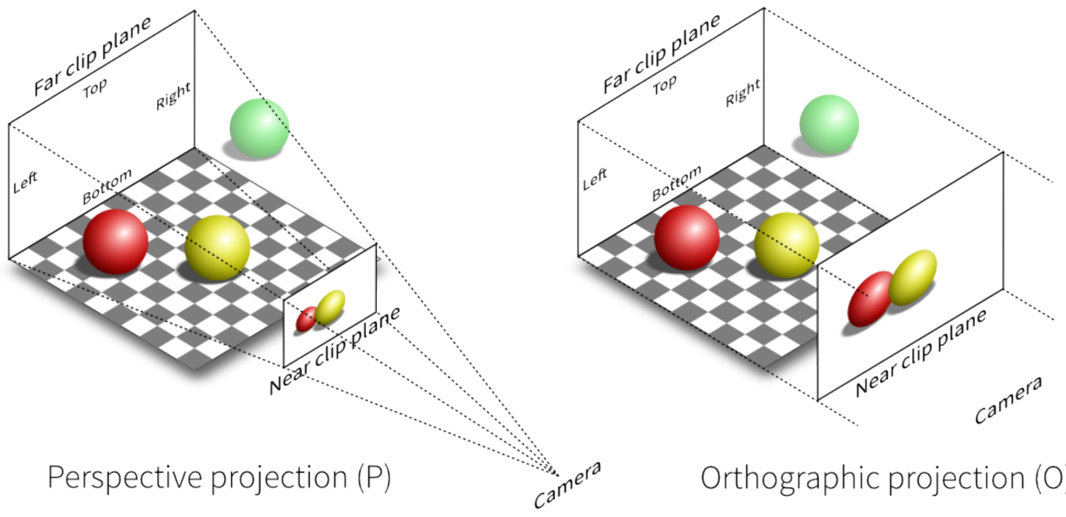


Figure 5.13: Different projection options in OpenGL

In Figure 5.13 we can see the main differences between the two. As mentioned, a perspective projection is similar to a pin hole camera, in the sense that it uses ray theory to project the points it can see to an image that will contain the objects it captures, all of these being captured using the perspective of the scene: if an image is further away, it will appear smaller in the image, and vice-versa. An orthographic projection, is totally opposing to this idea. It bases on a camera that captures all the image without projection. We could say that this camera is as big as the scene, and for each pixel it only captures what is straight in front of that pixel. We would be reducing our 3D world to a flat one, that conserves the objects sizes.

In our application, we have used an orthographic projection to be able to draw the whole camera image as the background of the image. To do so, we need to know the height and width of the world we are managing, to know where each object must be positioned. We have taken the size of the camera image as height and width, in order to keep the same measure system.

```
glOrtho(0,w,0,h);
```

Loading Textures

The next step in order to draw our background it to load the current camera image in an OpenGL texture. Textures in OpenGL are used to give the objects you are drawing a pattern that comes from an image or a camera. When drawing an object, one can choose the colour you want to give this object, but when you want a more realistic representation of the object, you may need to use an image to give it a different pattern.

In our case, we have an image saved in the shared memory of our application, and we wish to use this image to feed the background of the OpenGL world. To do so, we first have to generate and bind a texture in the application, which is saved as a *GLuint*, OpenGL's own integer type. With this, we then have to give OpenGL information on how the image is saved, in order for it to be able to extract the information correctly. In this case, our image is saved in OpenCV's type Mat. If it were to be saved in another image type, we would have to give OpenGL the correct information. With this, the only thing left would be to load the image data in OpenGl, as we can see in the following code snippet.

```
//Texture and Image Variables
GLuint textureId;
Cv::Matm_GLFrame = api->getBackgroundImage();
//Generate Texture
glGenTextures(1, textureID);
// Bind Texture
glBindTexture(GL_TEXTURE_2D, *textureID);
//Give information on the how the data is saved
```

```
glPixelStorei (GL_UNPACK_ALIGNMENT, (m_GLFrame.step & 3) ? 1 : 4);
glPixelStorei (GL_UNPACK_ROW_LENGTH, m_GLFrame.step/m_GLFrame.elemSize());
//Save Image Data to the Texture
glTexImage2D(GL_TEXTURE_2D,
0,
3,
m_GLFrame.cols,
m_GLFrame.rows,
0,
GL_RGB,
GL_UNSIGNED_BYTE,
m_GLFrame.data);
```

Draw Background

Having loaded the image we wish to draw in a texture, the last step is to draw this texture on the background of our OpenGL world. As we mentioned previously, textures are used to give an object a pattern, they cannot be drawn if it is not using an object as base. Therefore, to draw the background, we will have to draw a rectangle, the size of the OpenGL world, and they give this rectangle the camera image as a pattern.

A rectangle in OpenGL is noted as a *QUAD*, so we will have to draw this QUAD in order to have our background. We have several ways to perform this object rendering in OpenGL. In this project, we have adopted the way that uses less memory, and the one that is most similar to the drawing in OpenGL ES in Android, to be able to make use of some of the code in a latter phase. This method to draw an object is based on vectors, saving the corners of the quad in one vector, and those of the texture in another. After this, we must enable the function to draw using vectors, and proceed to draw the image, with it's corresponding texture.

```
//Load Texture Generated
loadTexture(textureId, 1);
//Enable Texture Rendering
glEnable(GL_TEXTURE_2D);
//Bind the Texture Generated to this Rendering
glBindTexture(GL_TEXTURE_2D, this->textureId);
//QUAD vertices vector
GLfloat vertices [] = {api->w,api->h,
    0,api->h,
    0,0,
    api->w,0};
//Texture vertices vector
GLfloat texVertices [] = {1,0, 0,0, 0,1, 1,1};
//Enable drawing through vectors
```

```
glEnableClientState(GL_VERTEX_ARRAY);
glEnableClientState (GL_TEXTURE_COORD_ARRAY_EXT);
// Draw the image.
glVertexPointer(2, GL_FLOAT, 0, vertices);
glTexCoordPointer(2, GL_FLOAT, 0, texVertices);
glDrawArrays(GL_QUADS, 0, 4);
//Disable Texture
glDisable(GL_TEXTURE_2D);
//Disable drawing through vectors
glDisableClientState(GL_VERTEX_ARRAY);
glDisableClientState (GL_TEXTURE_COORD_ARRAY_EXT);
```

5.3.3 Draw Virtual Film

With our background already drawn, the last step in order for our Augmented Reality to be finished would be to project our virtual cinema screen between the four pre-established points in the world, and over which we have already performed calculations to obtain the pixels between which the film will be represented. Therefore, in our shared memory, we have the position of the 4 pixels that will limit the cinema virtual screen, and we will have to perform an operation very similar to the one to draw the background of the image.

As we have mentioned, the OpenGL world within which we are working has been given the same size as the image captured by the camera. Therefore, in order to determine the position of the *film screen*, we will have to perform a transformation between the camera pixels that represent the corners of the virtual screen, and the corresponding points in OpenGL. OpenGL performs the homography between the texture frame and the four corner pixels provided as image placeholder. In Figure 5.14 we can see the difference between both systems. To perform such transformation, we can see that the width value will remain unchanged, but the height is taken from different points in each system, and must be recalculated.

The procedure followed to draw the virtual film screen, as we said, has been very similar to the one to draw the background. In this case, we don't need to perform any changes on the world's projection, as this is constant for the whole OpenGL world. Therefore, to draw the virtual screen, we will have to load the film image as a texture, and draw a quad between the four mentioned corners, rendering the texture in order to give it the image we need.

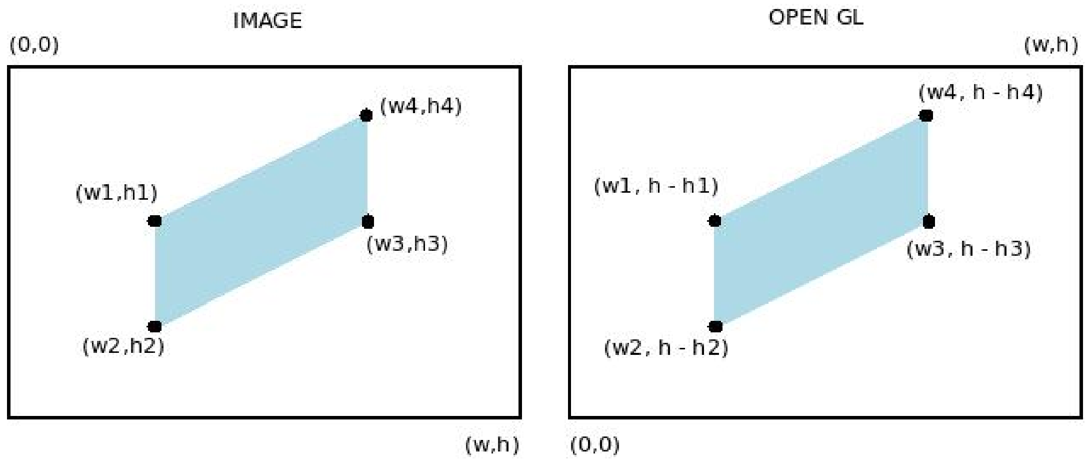


Figure 5.14: Relation between Image pixels and Points in OpenGL World

5.4 AR Cinema for Android

With a working PC prototype of our component, the next challenge was to export this application to an Android running mobile device. This application would try to reuse as much of the already developed code as possible, and perform its own representation of the film using a light weight OpenGL (OpenGL ES). As we can see in Figure 5.15, the general schema of the application is very similar to that of the PC component, with the main difference being in that, the camera images and the film frames are not provided externally via ICE, but taken from inside the app.

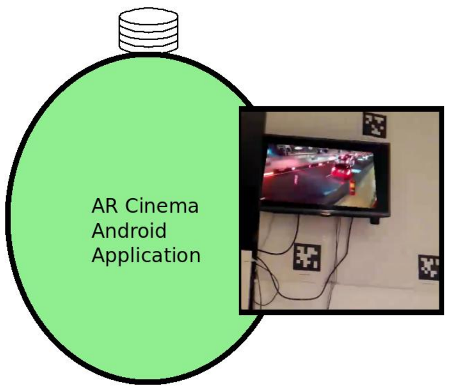


Figure 5.15: Android Global Schema

Going more into detail, in Figure 5.16 we can see that we will have another two modules inside our application, which are those related to the extraction of the film frames from a local video file in order for these to be projected, and that of the local Android camera.

These were not necessary in the PC component, as JdeRobot provided the images via ICE.

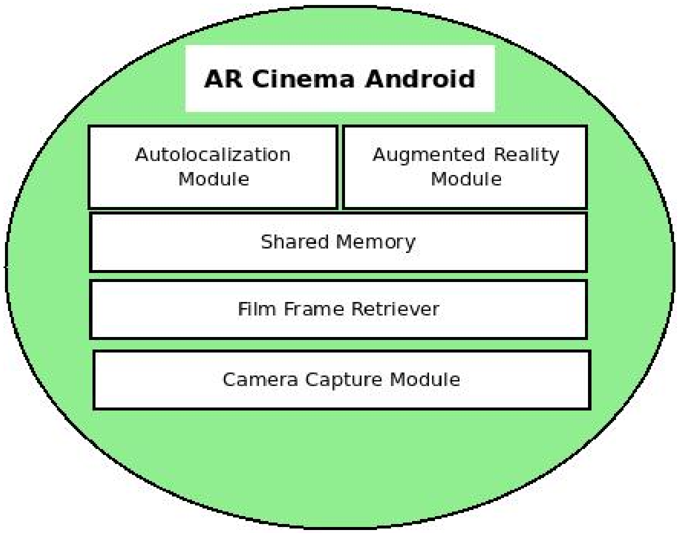


Figure 5.16: Design of Android AR Cinema

5.4.1 Android Manifest

In Android, the developer must indicate what permissions of the operating system, or the physical telephone we need to give the application, in order to give this information to the end user for him to decide whether they wish to install the application or not. To do so, we have to indicate these permissions in the *Android Manifest*, as follows:

```
<uses-permission android:name="android.permission.CAMERA" />
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
```

If we interpret this information, we can see that we are requesting permission to use the phone camera, as well as to access the external memory of the application. The first permission is obvious, seen as an Augmented Reality application needs to capture the camera image to be able to *augment* the reality. The second permission is provided for the application to be able to access the videos that are saved on the telephone, for the user to choose which film he would like projected.

5.4.2 Camera Capture Module

As mentioned, the developed Android application is not dependant on any external JdeRobot components. Therefore we do not use *cameraServer* to provide the camera images for the application, and we perform this action internally.

To obtain the camera images, we make use of OpenCV's Android library, that has a great wrapper between the code and the interaction with Android's camera. To do so, we have to implement *CameraBridgeViewBase.CvCameraViewListener2*, which has a method (*onCameraFrame*) that is called each time we have a new frame from the camera. With this, we have access to the image provided by Android's camera in a *cv::Mat* format, for us to work with. As we will explain in 5.4.5, thanks to this listener we will also provide the image to be drawn on the screen's background.

In order to use *OpenCV* in Android we must initialize it with a *BaseLoaderCallback*. We perform this action asynchronously in order to not block the user interface, as it's quite a costly operation.

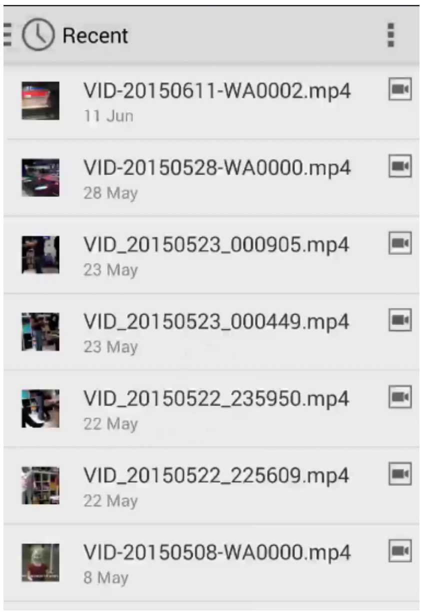
```
private BaseLoaderCallback mLoaderCallback = new BaseLoaderCallback(this) {
    @Override
    public void onManagerConnected(int status) {
        switch (status) {
            case LoaderCallbackInterface.SUCCESS:
            {
                Log.i(TAG, "OpenCV loaded successfully");
                mOpenCvCameraView.enableView();
            } break;
            default:
            {
                super.onManagerConnected(status);
            } break;
        }
    }
};

@Override
protected void onResume() {
    super.onResume();
    OpenCVLoader.initAsync(OpenCVLoader.OPENCV_VERSION_2_4_3,
         this, mLoaderCallback);
}
```

5.4.3 Film Frame Retriever

When trying to perform this application, seen as one can use ICE in Java also, we tried to provide the film frames via ICE, to avoid interacting with Android's file system. This gave us several problems: first, that both PC and Telephone had to be connected to the same Wi-Fi connection for it to work, and secondly, that sending the frames through ICE in Android was very slow, and didn't give us a good user experience regarding the virtual film. Finally, it was decided to let the end user choose a video saved on the telephone, and extract the tags of this film as OpenGL textures to be able to render them.

Choose Film to be Rendered

Of the main components of an Android application that we described in chapter 3, to choose a file from android's external file system, we would have to use a *Content Provider*. With this, we can perform a call to the telephones gallery to be able to choose a file. In this application, we indicated that the files we wish to be given to choose from where video files, as we can see in Figure 5.17, reducing the amount of files that it provided us with. When chosen, the Content Provider returned us with the full file path, in order for us to be able to access the file in another part of the application.



VID-20150611-WA0002.mp4
11 Jun

VID-20150528-WA0000.mp4
28 May

VID_20150523_000905.mp4
23 May

VID_20150523_000449.mp4
23 May

VID_20150522_235950.mp4
22 May

VID_20150522_225609.mp4
22 May

VID-20150508-WA0000.mp4
8 May

Figure 5.17: Content Provider to Choose Video File

Extract Film Frames

When having the path to the video file we wish to project, we made use of a class named *MediaPlayer*, that allows us to feed it with the data source we wish to play, and an OpenGL *Surface* on which we wish to dump the data. When we wish to start the media player, it will proceed to *play* the film, sending each frame to a *SurfaceTexture* that is associated to the *Surface* indicated to the *MediaPlayer*. This action is done automatically on another thread, in order to not block the thread from which we perform the call. On a application with a user interface, this action is very important, as if we block the actual user interface, it will give the user a bad user experience.

```
player = new MediaPlayer();
try
{
    File file = new File(this.filePath);
    FileInputStream fis = new FileInputStream(file);
    player.setDataSource(fis.getFD());
    player.setSurface(new Surface(renderer.getVideoTexture()));
    player.setLooping(true);
    player.prepare();
    player.start();
}catch (IOException e){
    returnConfig(getString(R.string.FormatNotSupported));
}
```

5.4.4 Porting Camera Localization

As we have mentioned several times in this document, we wanted to perform an Android application that reuses as much native code as possible, in order to be able to compare the performance and results of the application in both platforms in the fairest way possible. The part of the component that could be reused in both platforms corresponds to the auto-localization algorithm, thanks to the use of JNI library.

JNI (Java Native Interface) is an Android library that allows us to use native C++ code in our Android application, performing the compilation of such code specifically for Android architecture, and creating a dynamic library that can be called from our Java Code. Therefore, in order to use the auto-localization module developed in our Android platform, we could reuse the code (with certain small adjustments regarding the way the data is saved), and we only have to perform a wrapper between the C++ and the Java code, which JNI helps us create.

The wrapper created only has one function, as it was developed as an interface between Java and C++ code. This function is in charge of receiving the information provided by the Java part of the application, and then performing the calls needed to perform the camera pose estimation.

```
JNIEXPORT jfloatArray JNICALL
Java_org_jderobot_gui3_NativeWrapper_doProcessNative
(JNIEnv * jEnv, jobject jThis, jlong jImgAddr){
        //Get Img
        cv::Mat& img = *(cv::Mat*)jImgAddr;
        //Process Image
        jfloatArray result = doProcessing(img);
        //Return Data to Java
        return result;
    }
```

On the Java part of the wrapper, we have to create a function with the indicator *native*, and after so, loading the library that will include the C++ files.

```
public native float[] doProcessNative(long imgAddr);

/** Load the native library where the native method
* is stored.
*/
static {
System.loadLibrary("autoloc_lib");
}
```

In order to compile our C++ source files for Android architecture, we have to create two specific files in our Android Project, *Android.mk* and *Application.mk*, that will allow us to indicate the location of the source files and libraries needed (Android.mk), as well as the native resources that the application will use (Application.mk).

For the *Application.mk* file, the main fields are the following:

- APP_STL. Implementation of standard C++ library used.
- APP_CPPFLAGS. Flags that we must give the C++ compiler.
- APP_ABI. Type of architecture we are compiling the files for.

```
APP_STL := gnustl_static
APP_CPPFLAGS := -frtti -fexceptions
APP_ABI := armeabi
APP_PLATFORM := android-16
```

For the *Android.mk* file, we can mention the following fields:

- LOCAL_MODULE. Name we wish to give the generated dynamic library.
- LOCAL_SRC_FILES. Location of the C++ native files.
- LOCAL_C_INCLUDES. Location of the C++ header files.
- LOCAL_LDLIBS. Location of any auxiliary libraries we may need to use.

5.4.5 Augmented Reality

The last part of the Android application developed is the actual Augmented Reality part, this is, to paint the camera image as a background of the application, and then project a film between 4 predefined points, as we do on the PC version. To do so, the method used is quite different to the previous version. In this case, we do not use OpenGL to render two textures, one with the background image and another with the film frame texture, but only one with the film, and use a native view that contains the camera images as the background.

Draw Background

In order to represent the camera image, we have used the Java version of *OpenCV* to capture the image from the camera, and then represent it in a *JavaCameraView*. This kind of view is part of *OpenCV*, and inherits *CameraBridgeViewBase*, which provides us a listener for such a class in order to be able to manipulate the image provided by the camera.

```
setContentView( R.layout.activity_ar );
mOpenCvCameraView = (CameraBridgeViewBase) findViewById(R.id.java_cam_view);
mOpenCvCameraView.setVisibility(SurfaceView.VISIBLE);
mOpenCvCameraView.setCvCameraViewListener(this);
```

The listener provided has 3 functions, which are called when the camera view starts, stops and has a new frame. Out of these, the most important function is *onCameraFrame*, as it allows us to access the camera image before it begin represented on the screen. This is the point where, in our application, we create another thread that performs the image processing, as it is the only point where we can access the camera image in a OpenCV friendly format.

```
    @Override
    public void onCameraViewStarted(int width, int height) {}
    @Override
    public void onCameraViewStopped() {}
    @Override
    public Mat onCameraFrame(CameraBridgeViewBase.CvCameraViewFrame inputFrame){}
```

Draw Virtual Film

Having the camera image as a background, we need to create another view that is compatible with OpenGL in order to be able to project the film texture we have between the required points. The view used for this purpose is a *TextureView*.

```
surface = new TextureView(this);
surface.setSurfaceTextureListener(this);
addContentView(surface, new ViewGroup.LayoutParams
    (ViewGroup.LayoutParams.WRAP_CONTENT,
     ViewGroup.LayoutParams.WRAP_CONTENT ) );
```

Every TextureView has a SurfaceTexture associated to it, which can be provided to a SurfaceViewRenderer in order to perform the rendering of graphics on the Surface Texture.

For this application, the *SurfaceViewRenderer* has been named *VideoTextureRenderer*, and between others, we have declared a attribute named *videoTextureHolder* which corresponds to the *SurfaceTexture* that contains the film frame mentioned previously, and which is directly related to a native OpenGL texture attribute. From our code, if we call the method *updateTex()* from videoTextureHolder, it will dump the texture it contains to the native texture, in order for us to be able to use it for other purposes. We have assigned this attribute with yet another listener, that will inform us when we have some new information in the SurfaceTexture, and therefore when we need to send it to the native texture for this to render it.

When performing the actual rendering, there are two points to be taken into account. For one, that the background of the *TextureView* must be transparent in order to be able to see the camera image below, and secondly, the position of the film projection.

The first point is simple and is resolved in the following code:

```
//RGB and ALPHA VALUE (corresponds to transparency)
GLES20.glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
//Apply the colour above
GLES20.glClear(GLES20.GL_COLOR_BUFFER_BIT);
```

For the second point, as in the PC version of this application, we need to perform a transformation between the image pixels and the representation of these that OpenGL will apply. In this case, the OpenGL world does not have the same measures as the image, and we cannot set them to be like so. This is because the variety of devices that Android is orientated to is very high, and each of these can have different size screens. If OpenGL allowed the mapping of points to the screen be personalized, it would be very difficult to provide support to all the available devices.

In order to solve this problem, OpenGL assumes a square uniform coordinates system for the developer to use, and then internally will scale this image, as is represented in Figure 5.18.

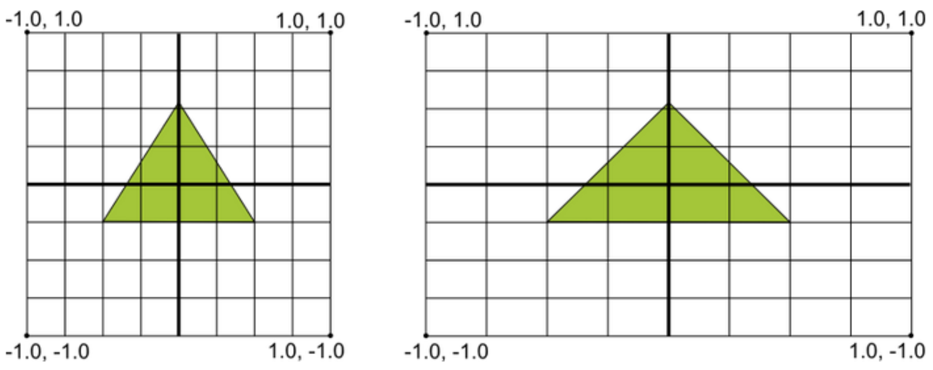


Figure 5.18: Relation between Image pixels and Points in Android OpenGL World

With this information and the pixels between which we wish to project, we can calculate their correspondence in Android OpenGL ES format:

$$h_{android} = \frac{2h_{img}}{h} - 1$$
$$w_{android} = \frac{2w_{img}}{w} - 1$$

Having these points, we can then proceed to perform the virtual film projection. To do so, we need to have clear the way one gives OpenGL ES the data to be drawn. We must use vectors to indicate both the QUAD and the texture positions.

```
//Define ByteBuffer
ByteBuffer texturebb = ByteBuffer.allocateDirect(textureCoords.length * 4);
texturebb.order(ByteOrder.nativeOrder());
//Put the data in
textureBuffer = texturebb.asFloatBuffer();
textureBuffer.put(textureCoords);
//Reset to the begining to start reading from it
textureBuffer.position(0);
```

After defining the required vectors, we can proceed to perform the drawing of the QUAD and assign it a texture.

```
//Enable Position Vector
GLES20.glEnableVertexAttribArray(positionHandle);
GLES20.glVertexAttribPointer(positionHandle, 3,
    GLES20.GL_FLOAT, false, 4 * 3, vertexBuffer);
//Bind and Activate Texture
GLES20.glBindTexture(GLES20.GL_TEXTURE0, textures[0]);
GLES20.glActiveTexture(GLES20.GL_TEXTURE0);
GLES20.glUniform1i(textureParamHandle, 0);
//Enable Texture Vector
GLES20.glEnableVertexAttribArray(textureCoordinateHandle);
```

```
GLES20.glVertexAttribPointer(textureCoordinateHandle,
    4, GLES20.GL_FLOAT, false, 0, textureBuffer);
//Draw
GLES20.glDrawElements(GLES20.GL_TRIANGLES, drawOrder.length,
    GLES20.GL_UNSIGNED_SHORT, drawListBuffer);
//Disable Vectors
GLES20.glDisableVertexAttribArray(positionHandle);
GLES20.glDisableVertexAttribArray(textureCoordinateHandle);
```

Chapter 6

Experiments

After describing the procedure we have followed to develop the AR Cinema application, we must now describe the tests we have performed on it, in order to verify the results obtained. The first tests performed have been done in the three different platforms we have been using during the development of this project: real scene, Gazebo and Android. Apart from these, we have also performed accuracy tests to the algorithm in Gazebo, in order to calculate the error that our algorithm introduces to the system.

6.1 PC Prototype AR Cinema with Real Camera

To start of this chapter, we would like to provide some images of the working prototype of the standard version of the JdeRobot component that has been designed.

In Figure 6.1a we can find an image of the scene we have used to perform this demo. This image has been remarked in Figure 6.1b, where we can see the three tags present in the world, all located on the same plane, marked in red. The television located between them (and marked in blue) indicates the points we wish to use as the cinema screen.

As we have mentioned, this component receives both the camera images and the film frames through ICE interfaces, using as a provider the JdeRobot component *cameraServer*. It also receives a configuration file that will inform the application of the real three dimensional position of the tags that are visible in our world, as well as the position of the four points between which we wish to project the virtual film.

Using the camera image provided, AR Cinema performs an image processing, extracting the pixels that correspond to each tag and using this information, along with the real tags position, to calculate an estimated camera position and orientation. With this, we will

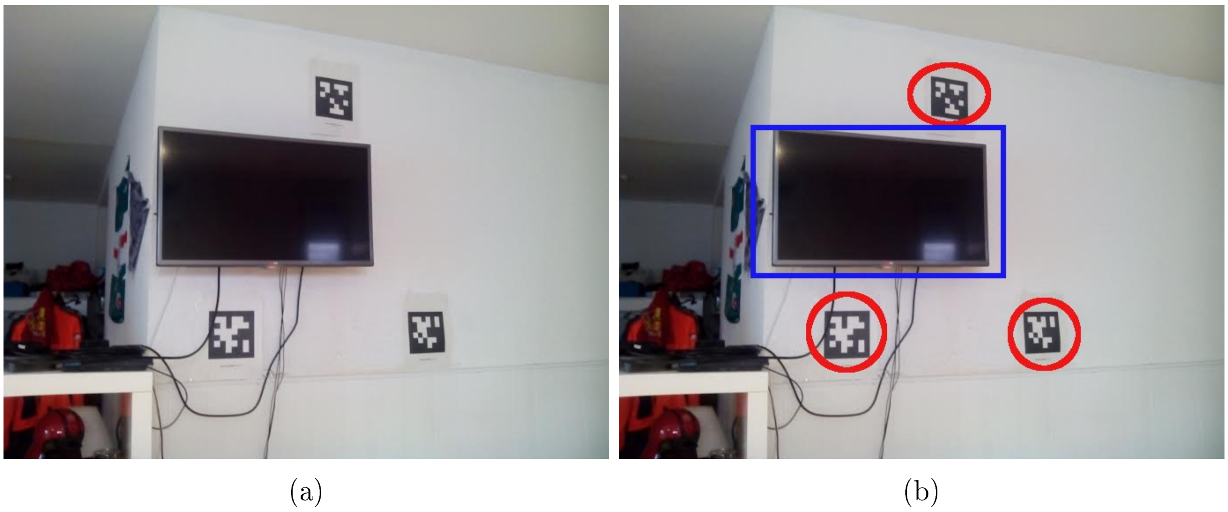


Figure 6.1: Real Camera Scene for AR Cinema Demo

then perform a projection of the *cinema corners* to calculate the pixels that correspond to these positions.

Last, we shall perform a projection of the film frames between those calculated pixels, giving the effect of a virtual cinema film, that will be playing in Augmented Reality.

The demo we shall see in the images has been executed on a *Intel (R) Core (TM) i7 3537U @ 2.00 Ghz* running *Ubuntu 12.04*. As the algorithm performs a great amount of calculations, the computational cost is quite high. When trying to execute this same demo on a *Intel (R) Dual Core (TM) @ 1.4 Ghz*, we noted a serious performance flaw. The images are provided by *CameraServer* at 30fps, which is the speed the component works at.

We can see some images of this component working in a real scenario in Figure 6.2, where we can see the execution from different angles. As we can see, the film projection is performed between the TV area, and it remains between it during the different rotation and position changes, being consistent with the movements the camera performs.

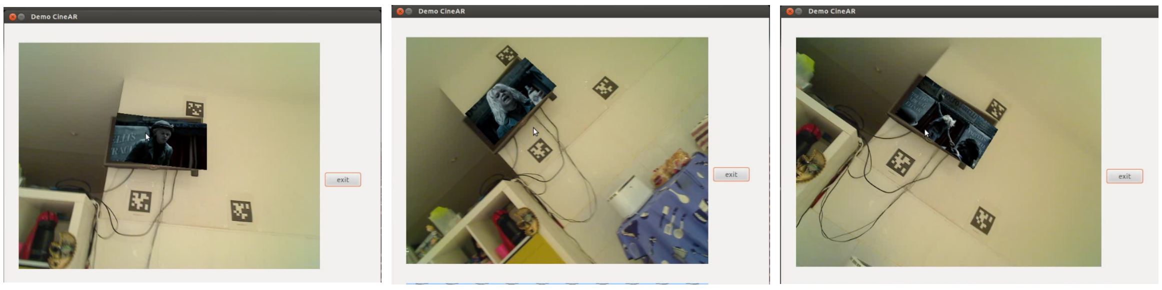


Figure 6.2: Working Prototype AR Cinema

6.2 PC Prototype AR Cinema with Gazebo Simulator

As we have mentioned during this document, the video source that is analysed, and therefore augmented is irrelevant to the algorithm, needing only to have known tags around the world. In order to demonstrate this, we have performed the execution of the component using the Gazebo simulator, which has been our test environment during the development of this project, and where we have tested all the algorithms developed before using a real scene.

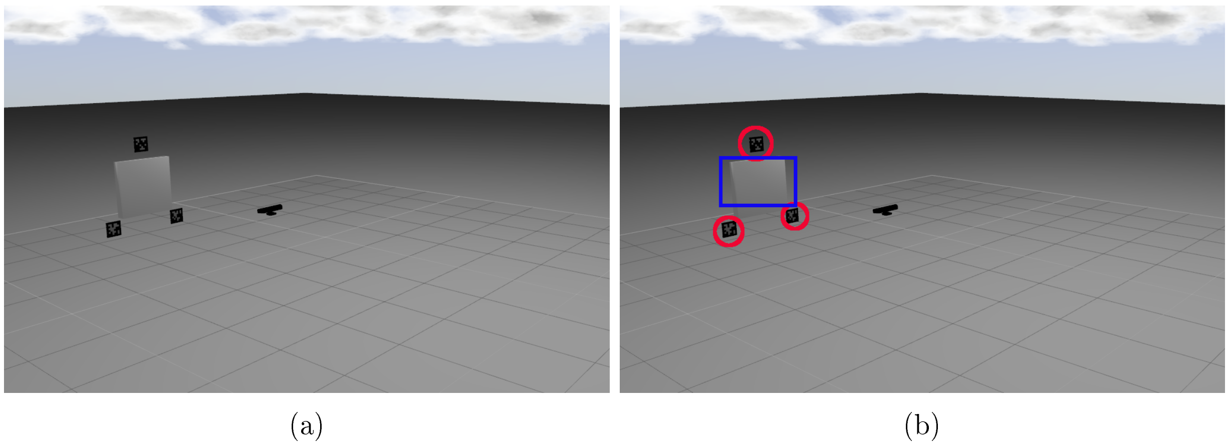

(a) (b)

Figure 6.3: Gazebo Simulator Scene for AR Cinema Demo

In Figure 6.3a we can see the virtual scene we have designed in Gazebo, which has all the tags located in the same position as they have in the real world. In this case, the cinema screen is represented by the grey box that is located between the 3 markers, which are in the same position as in the real world. We can see these relevant features of the scene marked in Figure 6.3b, with the tags in red and the projection screen in blue. In Figure 6.4 we can see images from the execution in Gazebo.

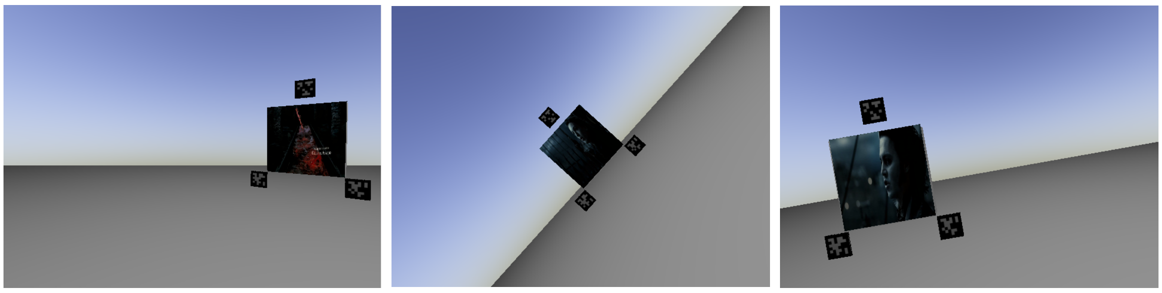


Figure 6.4: Working Prototype AR Cinema in Gazebo

As happened in the real scene, in these images we can appreciate different camera movements, both in position and in orientation, and how the film projection remains between the predefined points for it. In this case, we can appreciate changes in the distance to the screen, as well as rotations in two different axis, so as to test several movement options around the designed scene.

6.3 Android Prototype AR Cinema

To finish the set of working scenes this application has been tested on, we shall describe the execution on an Android platform. The Android device used for this test has been a *Quad Core Cortex A7 @ 1.3 GHz*. If we mentioned previously that the execution of the algorithm on a PC with similar specifications to this telephone was quite slow, we can assume the execution will also worsen on this platform. We tried to execute this application at 30 fps on this telephone, having quite poor performance results. When lowering the camera flow's image rate (20 fps), we obtained better results, but these are still far from those obtained on the PC.

The scene used for the Android demo is the same as that used during the PC demo with a real camera, and which we can see in Figure 6.1a. In Figure 6.5 we can see a successful execution on the Android device, with two different angles.

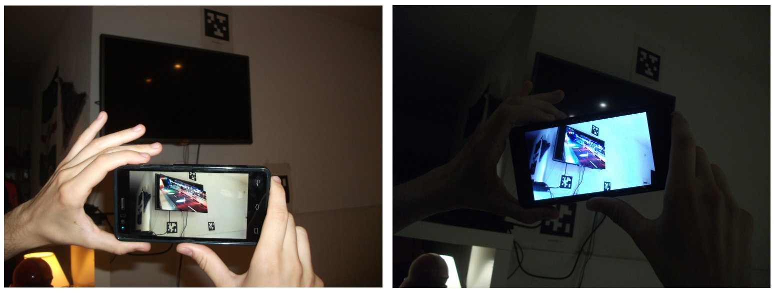


Figure 6.5: Working Android Prototype AR Cinema

6.4 Auxiliary Components for Gazebo

The first task to be performed in order to be able to verify our results, was the development of certain JdeRobot components and Gazebo plugins that were necessary to interact with the simulated Gazebo world. More specifically, we needed to have a camera *remote control*, to be able to move it in the Gazebo 3D world and know the camera's absolute position. This required component can be described as a Gazebo camera *teleoperator*, and was developed in a collaborative way between the JdeRobot robotics group of the URJC. This component was named *moveCamera*, and would have to be able to move the camera in the Gazebo world, giving it the exact points where it should place itself, and also receive a feed of the image from the camera and the real position of the camera, to help the user know where the it is placed at all times.

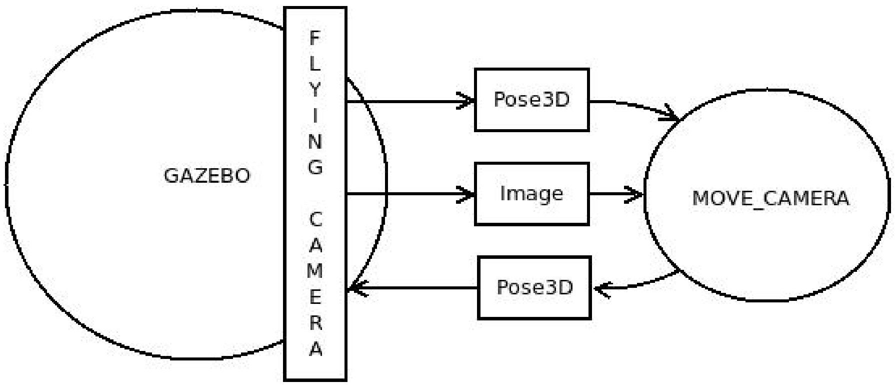


Figure 6.6: Relations moveCamera component - flyingCamera plugin

Associated to this component, we also have a Gazebo plugin that can interact with it and move the simulated camera. This plugin will be in charge of receiving the commanded camera position from the teleoperator and setting the cameras position to that required, as well as providing the application with the images that the camera is capturing, along with the real camera pose.

The communication between these modules is done via ICE interfaces. We have three interfaces, one that starts from the plugin, and provides the camera position, another that also starts on the plugin and provides the camera images, and the last one, which starts on the component and sends the plugin the new desired position for the camera.

In Figure 6.6 we can see a relation between the different interfaces that connect Gazebo's plugin and the developed component, and in Figure 6.7, an image of the component in action.

6.5 Camera Localization Accuracy Study

The tests performed on the auto-localization module of our application were performed in the Gazebo simulator. This is because, being a controlled environment, we can have the real camera position information along with the estimated one, and therefore we can quantify the error that our module is introducing.

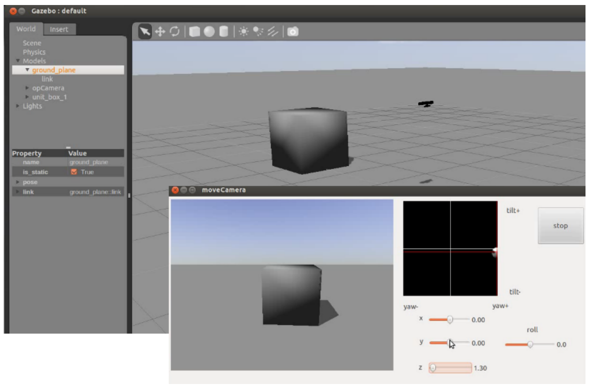
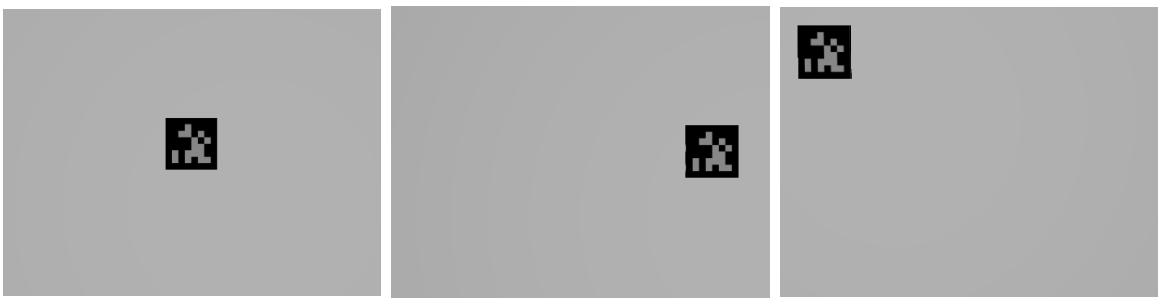


Figure 6.7: moveCamera Component

Thanks to the previously described component, we have the ability to move the camera in our Gazebo world and receive the images that it sees, along with the real camera position. Using the same auto-localization algorithm as described in chapter 5, we can use a combination of this algorithm and the auxiliary component to perform an error analysis of the algorithm, in order to verify the results. To do so, we have performed a series of experiments in order to validate the error that is introduced by our algorithm in several different scenarios.

6.5.1 Eccentricity

To start with, the first experiment performed is regarding a single tags radius from the camera centre. To do so, we have placed the tag in three different positions of the Gazebo world, each with different radius values and positions within the image, as we can see in Figure 6.8, and analysed the error in each axis (x, y, z), as well as the rotation (roll, pitch, yaw).



(a) Scene 1 (b) Scene 2 (c) Scene 3

Figure 6.8: Eccentricity Experiment Scenes

After performing the camera pose estimation for each of theses scenarios, leaving the camera height as a constant value, we can see in the Table 6.1 the results we have obtained. In this table, we have the three mentioned scenarios, along with the error in each variable we have obtained, being in meters the error measures for distances, and in radians for the case of the angles.

Scene	**Z Error**	**XY Error**	**Roll Error**	**Pitch Error**	**Yaw Error**
1	0.124515	0.00676985	0.18925	0.00423333	0.1983
2	0.136663	0.0292605	0.2519	0.0282087	0.25338
3	0.105571	0.00282806	0.242208	0.00591452	0.244711

Table 6.1: Eccentricity Experiment Results

If we analyse this information, we must say, it is not very intuitive. The fact that the scene with the tag in the centre has a low error in all 3 position axis is reasonable, but for what we have not managed to get an explanation is that, in scene number 3, where the radius between camera and tag is highest, the error is very similar to that in scene number one, where we would expect to have the best results.

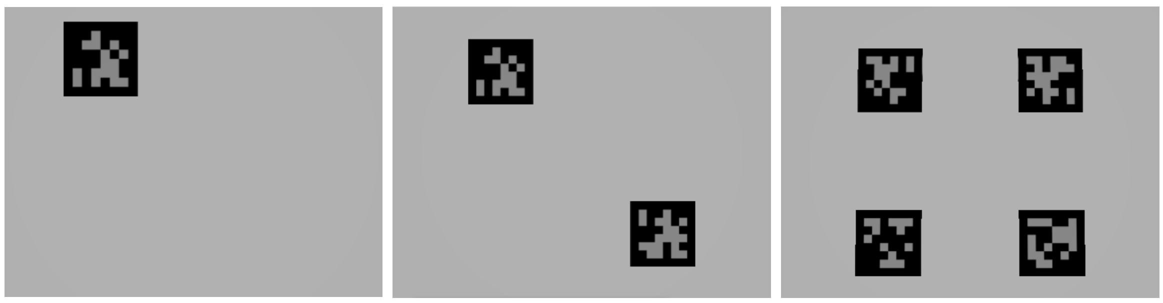
Regarding the angles, we have the same problem. In this case, it isn't so obvious, seen as the error in scene 3 isn't so similar to that in scene 1, but it's still very similar to the second scene, which we would expect to be lower.

The only minimal explanation we have found for these results is that, even if the radius is higher in scene number three, the fact that the tag is moved in two different axis helps compensate between them the error caused by the high radius. In scene two, we have a lower radius, but the fact we only have one movement axis turns the scene to having a high error that isn't compensated, and therefore is dragged all the way to the end of the experiment.

6.5.2 Number of Tags

With the results we have obtained in our previous experiment, we will now proceed to analyse the impact that the number of tags has on the obtained results. Seen as the previous results indicate quite a large difference between one radius and another, we will perform this experiment having all the tags around the same radius, in order for the analysis to be as fair as possible.

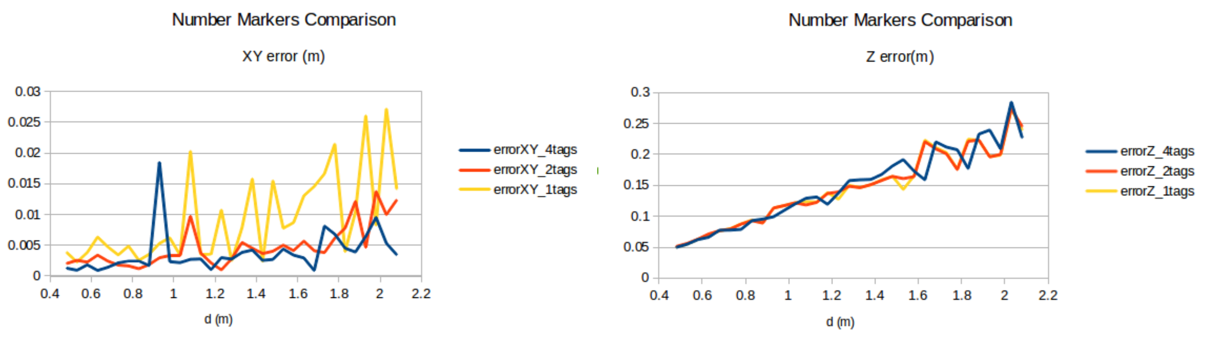
In this case, we have also considered 3 different scenes, each with one, two and four tags, in order to decide the best option to be used, as we can see in Figure 6.9.



(a) One Tag (b) Two Tag (c) Four Tag

Figure 6.9: Number Markers Experiment Scene

In order to be able to represent this information in a graph, we decided to maintain the camera's XY position fixed for this experiment, and increase the height of the camera for each case. Like that, we can have a comparison on how the error changes by varying the camera's Z value, as well as the impact the number of tags in the image has in such error.

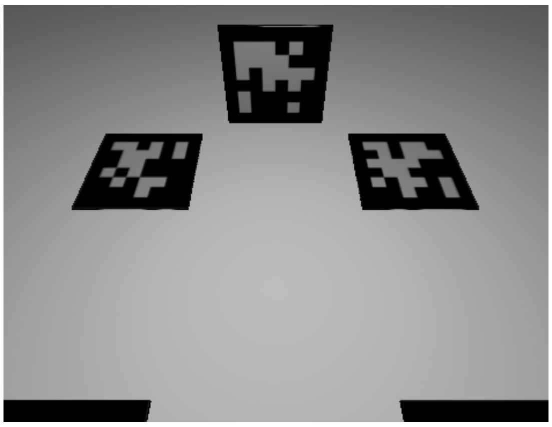


(a) Error in XY (b) Error in Z

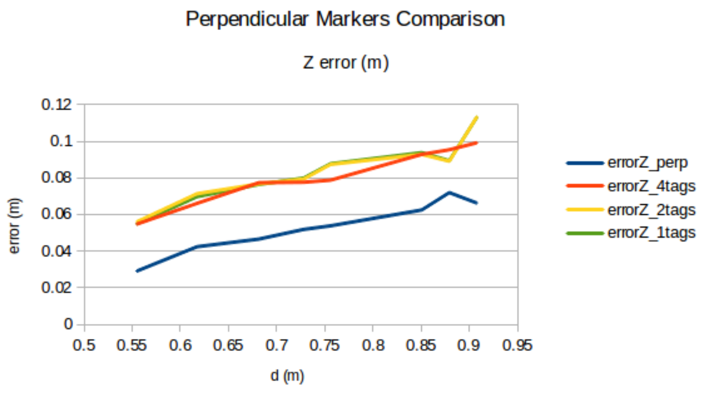
Figure 6.10: Number Markers Experiment Results

As we can see in Figure 6.10, the error in the XY axis reduces as we increase the number of tags in our image, as well as does it increase with the distance between tags and camera. On the opposite side, we can see that the error introduced in the Z axis remains practically the same regardless of the number of tags we have. This result can be explained as all the tags that have been used are coplanar, and therefore should the same value in Z. To perform the mean value of a group that will be very similar, will result in yet another similar result.

In order to verify this, we will perform another experiment similar to the previous, but apart from the four coplanar tags, we are going to add a fifth tag perpendicular to the previous, as can be seen in Figure 6.11a. With this, we hoped to find a way to reduce the error in the Z axis as much as possible.



(a) Perpendicular Experiment Scene



(b) Perpendicular Experiment Error in Z

Figure 6.11: Experiment and Results for Perpendicular Tags

In Figure 6.11b we can see that our supposition was correct, and that introducing markers on different planes will reduce the error in Z axis. This result is very interesting in order to apply it in the final scene that will be used during the demo application we prepare.

6.5.3 Translation

With the previously gathered information, we shall proceed to calculating the error when the camera is moved within the X and Y axis. To do so, we will part from the 4 tag scene, and perform movements in X and Y separately, and then in both, to see the different error values that we obtain.In Figure 6.12 we can see an example of one of the test scenes.

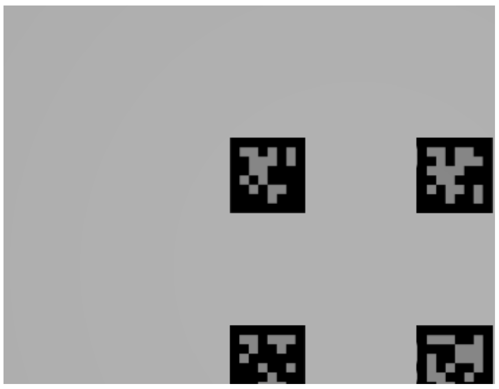
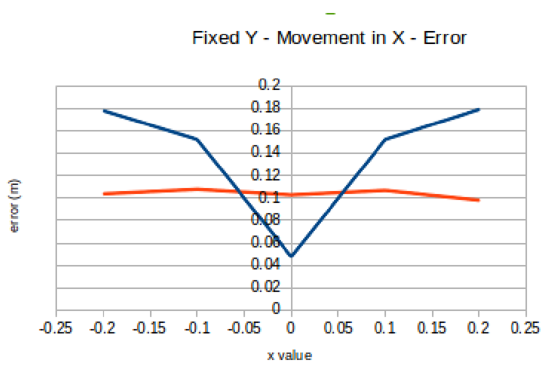
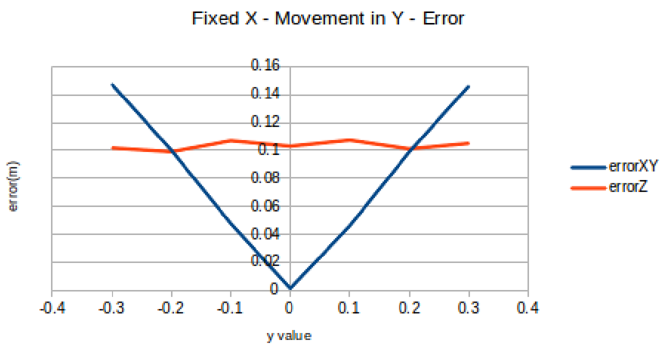


Figure 6.12: Translation Experiment Scene

In order to represent the error we have obtained for this experiment, we are going to do so in three different graphs. The first of these we can find in 6.13a, and represents the error caused by moving the camera linearly in the X axis, represented in two different lots: one that represents the XY error, and another that represents the Z error. In the graph, we can see how the error that is introduced in the Z axis is constant, while that of the XY axis increases with the movement. In 6.13b we can see how this effect also occurs when the single translation is done in Y.



(a) X Translation Results



(b) Y Translation Results

Figure 6.13: Translation Experiment Results for movement in one axis

Having characterized the error in each axis separately, we could conclude that a movement in both axis would have a similar result: the error would increase with the distance between the tag and camera centres. In Figure 6.14 we can see a graph that indicates that this effect indeed happens. We can see that, for the error in the XY axis, it increases with the distance. The error in Z maintains a constant value, which is coherent with the rest of the information we have received during these experiments.

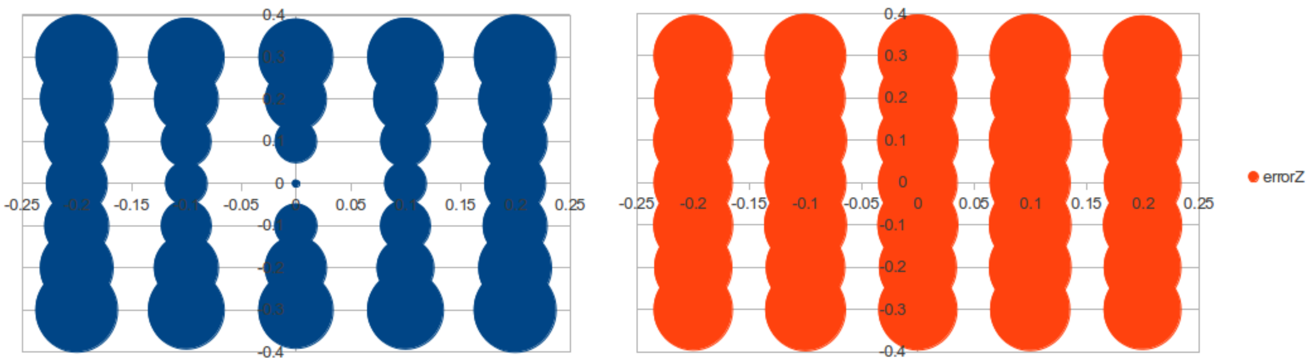
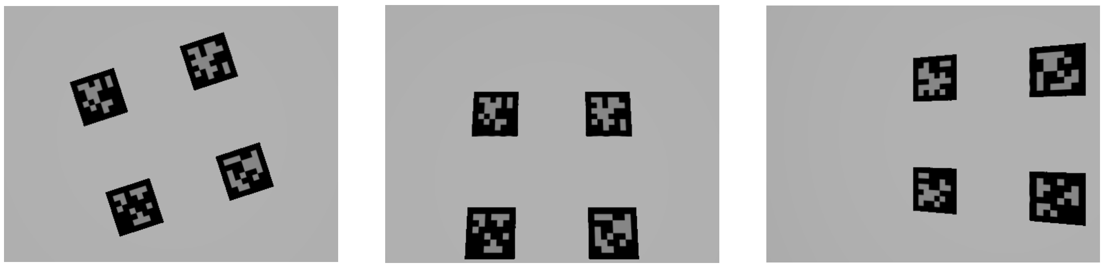


Figure 6.14: Translation Experiment Results for movement in two axis

6.5.4 Rotation

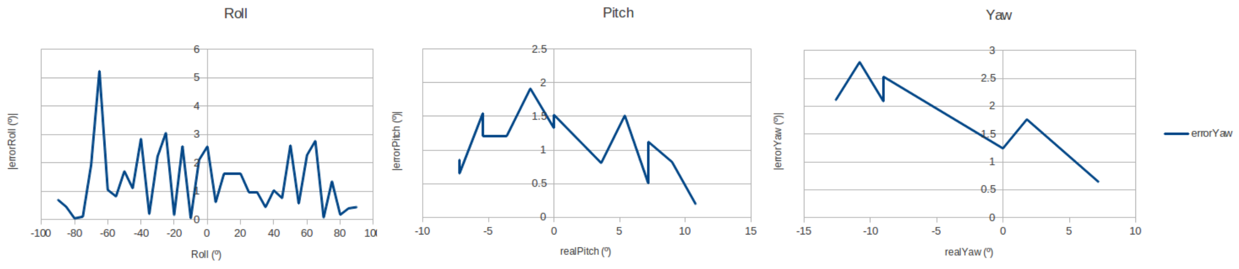
The last point of our algorithm we wish to perform experiments with, is how is behaves when performing a rotation around a single axis. With this, we will have completed a full analysis of the algorithm, with movements in all the directions that are available to the camera, and therefore verified it's validity.



(a) Roll Rotation (b) Pitch Rotation (c) Yaw Rotation

Figure 6.15: Rotation Experiment Scenes

In this case, we have also performed three tests, one performing a rotation in each direction, yaw (around Z), pitch (around Y) and roll (around X), as we can see in Figure 6.15 with the three different rotations.



(a) Error in Roll (b) Error in Pitch (c) Error in Yaw

Figure 6.16: Rotation Experiment Results

In Figure 6.16 we can see the results of this experiment. As we can see, the error remains relatively constant during the performance of the rotations, and this error is below 3^o in most cases. The interpretation we can make of this is that the rotation does not introduce much error in the camera pose estimation. If we compare it to the translations performed by the camera, we can conclude that this last one has much more impact on our algorithm than the rotation does.

From these results, we can conclude that the algorithm developed has is flaws, as most do, but even so, the performance it gives is quite high, and when applied to an AR application, this executes fluidly, without many sudden jumps.

Chapter 7

Conclusions

During the different chapters of this document we have described the building process of the visual localization system developed, along with the augmented reality application we have built using it, starting from a description of the technologies and systems used until the experimental validation of the systems. Now, in this final chapter, we are going to go through the objectives we had at the beginning of this document, and see in what measure these have been fulfilled. Also, we shall comment a series of future actions that could be taken in order to improve the current project.

7.1 Conclusions

We have designed and developed a marker based auto-localization algorithm, making use of a very wide range of different technologies, which had to be tamed, not only in order to understand how they work, but how to combine them between themselves in order for them to all work together towards the final results. Based on this algorithm, we have also developed an augmented reality application, and this has been prototyped not only for a Linux running PC, but also on an Android framework. To perform this, the first developed the PC version, and then performed an analysis on how this could be ported to an Android device with a minimum number of changes.

If we go back to chapter 2, we can remember a series of objectives we established for our application. Going back to them, and performing a posterior analysis of these, we can conclude the following points:

- The first objective we established was the investigation into self-localization techniques, and the following development of a marker based one. In chapter 5 we have performed an in depth analysis of the algorithm that has been developed, and in the sixth one, an experimental validation of its accuracy. The developed algorithm is based on the pin hole camera model, that has been described in chapter 4, and can be divided in three main steps. The first, the detection of markers in the image, in order to use these to perform the localization of the camera. The next step is to actually perform this localization, using the information of the tags position in the world, and an estimation we perform of the camera's position from the tag. The last part of the algorithm is the fusion of several of the previously calculated camera positions, seen as we have one for each tag. In order to perform this algorithm, we have made use of several vision libraries, the main ones being April Tags, ARuCo and OpenCV, that have worked together to obtain the final results. Finally, and in order to verify the validity of our algorithm, we validated it in chapter 6, performing a series of experiments designed to test the algorithm in different limit cases, and obtaining valuable information that could be used in a future application.

- The following objective we gave ourselves was the development of an augmented reality application that made use of the developed localization algorithm. In order to accomplish this point, we took note of the validation results in chapter 6 in order to use these to design a good scene for the final application. With the scene designed, we integrated the self-localization algorithm in a JdeRobot component that uses this camera position to create an OpenGL augmented reality application. This application consists of a virtual cinema screen that projects a film between 4 pre-established corners. Therefore, our application does not only have to perform the augmented reality, but based on the camera position it must also calculate the image pixels between which the projection must be done, making use of the pin hole camera model (chapter 4).

- The last objective given was that of porting the previous application to an Android platform, reusing the same localization algorithm as the PC application. When we say using the same algorithm, we do not only mean the same principles, but trying to reuse as much code as possible, in order to verify the differences between both executions fairly. Thanks to the use of *Java Native Interface*, we can use C++ code in an Android application. *JNI* will compile this code in Android's architecture, and provide us with a dynamic library so that we can execute this same code on Android. As we know, this application doesn't only have the algorithm part, but also a graphics representation. This GUI cannot be reused from the PC version of the application, and has been developed directly for Android thanks to OpenCV Android library, to represent the camera image, and OpenGL ES to project the virtual objects.

Apart from these main objectives that we have just described, in chapter 2 we also established a series of requirements that the project has to comply. These were the following:

The developed components in this project have been developed under JdeRobot framework, in C++. Using JdeRobot has given us a great number of advantages, as it offers us a series of ICE interfaces to communicate with different components that we may need to use, as well as also providing information for others to reuse in their own applications. Apart from JdeRobot, we have also taken advantage of several other libraries and software that have given us a series of tools that were very important during the development. OpenCV during the image processing, April Tags and ARuCo with the algorithm development, and OpenGL during the augmented reality are the main ones, but there have also been other smaller libraries that have helped during this development.

The algorithm we have developed has been written in C++, and executed both on a Linux running PC as on an Android mobile terminal. In these cases, we have obviously used different cameras in each system, and therefore performed test with several calibrated cameras. We can see then, that the application is not dependant on a concrete camera, but on a decent camera calibration method.

All the described operations are performed in real time, as the image flow we are working with provides us with approximately 30 fps, and there are processed and analysed on the moment in order to provide our augmented reality application with the best user experience possible.

In the experimental analysis we have seen how the application works in a series of different scenes. Obviously, the algorithm will work better with a scene that has been prepared with that objective, such as the ones we described in chapter 6. When using a higher number of tags we can obtain a more robust estimation, as we have more information on which to base the estimations. Also, if we distribute these tags in different planes we shall obtain better estimations as we have several tags that provide information from radically different positions. Obviously, being a marker dependant algorithm, when none of these markers are visible from the camera it will loose it's pose estimation, and therefore no augmented reality will be projected.

Regarding the knowledge acquired during this project, most of this has been regarding computer vision and auto-localization techniques. I believe I have now got quite a good base of information regarding camera models, image analysis, three dimensional camera localization, which are all very modern technologies.

7.2 Future Works

As well as performing a summary of the objectives that we have fulfilled, we would like to end this document by describing a series a future works that we believe could be performed in order to improve the developments performed.

In first place, the developed augmented reality application could replace the current self-localization algorithm by another more robust one, or one that isn't dependant on a known scene for this to work. Introducing visual SLAM or PTAM to our AR application could give this a better user experience, as it wouldn't be dependant on the markers.

Another improvement that would be very interesting is a optimization of the code, in order to improve the performance on mobile terminals. We have seen that the application has a correct execution on an Android terminal, but this is not as fluid as on a PC, and therefore, should be improved in order to have a fully working and useful mobile application.

Finally, the last improvement we can mention for our project is the development of a final application that has more commercial objectives. The application developed is perfect to demonstrate the integration between self localization and augmented reality, but a commercial application that can be sold is always a good idea.

Bibliography

[1] José María Cañas Plaza. Jerarquía dinámica de esquemas para la generación de comportamiento autónomo. *Doctor's Thesis, Universidad Politécnica de Madrid*, 2003.

[2] Andrew J. Davison, Ian D. Reid, Nicholas D. Molton, and Olivier Stasse. Monoslam: Real-time single camera slam. *IEEE Trans. Pattern Analysis and Machine Intelligence*, 29:2007, 2007.

[3] Agustín Gallardo Díaz. Herramienta de calibración de cámaras en jderobot. *Final Year Project, ETSII, Universidad Rey Juan Carlos*, 2013.

[4] Richard Hartley and Andrew Zisserman. *Multiple View Geometry in Computer Vision, Second Edition.* Cambridge University Press, 2004.

[5] Alejandro Hernandez Cordero. Autolocalización visual aplicada a la realidad aumentada. *Master's Thesis, ETSIT, Universidad Rey Juan Carlos*, 2014.

[6] Redouane Kachach. Calibración automática de cámaras en la plataforma jdec. *Final Year Project, ETSII, Universidad Rey Juan Carlos*, 2008.

[7] G. Klein and D. Murray. Parallel tracking and mapping for small ar workspaces. In *Mixed and Augmented Reality, 2007. ISMAR 2007. 6th IEEE and ACM International Symposium on*, pages 225–234, 2007.

[8] Daniel Martín Organista. Odometría visual con sensores rgbd. *Final Year Project, ETSIT, Universidad Rey Juan Carlos*, 2014.

[9] Edwin Olson. AprilTag: A robust and flexible visual fiducial system. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, pages 3400–3407. IEEE, May 2011.

[10] Lawrence G. Roberts. Machine perception of three-dimensional solids. *Thesis (Ph. D.)–Massachusetts Institute of Technology, Dept. of Electrical Engineering*, 1963.

[11] Davide Scaramuzza. Fast semi-direct monocular visual odometry. *Proc. IEEE International Conference on Robotics and Automation (ICRA)*, 2014.