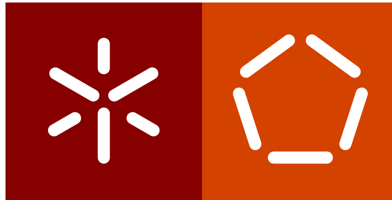
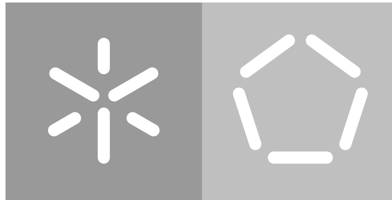


Universidade do Minho
Escola de Engenharia
Departamento de Informática

Pedro Nicolau Machado Carvalho

Mobile AR for large 3D Scenes using Markers

December 2017



Universidade do Minho
Escola de Engenharia
Departamento de Informática

Pedro Nicolau Machado Carvalho

Mobile AR for large 3D Scenes using Markers

Master dissertation
Master Degree in Computer Science

Dissertation supervised by
António José Borba Ramires Fernandes

December 2017

Acknowledgements

There was some fundamental help in creating this dissertation which I would like to express my unconditional gratitude.

To my supervisor Professor António Ramires, I would like to thank for the guidance, availability, commitment and patience given during the work of this thesis.

To my family and friends, I would like to thank for their support, help and advice along my academic career.

ABSTRACT

Augmented Reality (AR) relies on overlaying 3D objects over a real environment on a screen display. Recently this technology has begun to gain a lot of attention by the common consumer because it is starting to be used in devices available to them, one of those devices being mobile smartphones. Hence, it is relevant to seek the limits and explore the potential that these devices are capable of, in order to build a balanced AR experience.

The purpose of this work was to study the limits of developing a localization technique for an Android smartphone which was accurate and fast. This technique was to be used in an AR application which allowed the user to view large 3D environments related to the real scene (e.g a room could be turned into an aquarium).

The research was initially targeted at finding the right technique to localize the device, by assessing the accuracy of algorithms like Inertial Navigation Systems, Monocular Visual Odometry and SLAM on a hardware limited device - a smartphone. The results obtained with these algorithms did not meet the precision requirements to avoid misalignments between virtual and real entities.

The solution found was to use 2D markers around the real scene with the help of a computer tool which comprised a novel feature which aided the user in positioning the markers with visual cues painted on the 3D model's surface.

Markers detected by the smartphone's camera allowed the computation of the position of the device. These markers are detected using Vuforia's SDK Image Target Recognition. Furthermore, some improvements were implemented to provide more robust tracking for large 3D Models.

A framework was devised which consists of the android AR application and the computer tool which, aside from helping the user position the markers, can build AR setups and deploy them to the mobile app.

Finally, some tests were made which assess the accuracy of the implementation. Example AR scenes were built to examine how the application fares in very different environments, one being a room and the other a public square and streets in a village.

RESUMO

Realidade Aumentada consiste na sobreposição de objetos 3D sobre um ambiente real. A transição para *smartphones* desta tecnologia tem aumentado a sua popularidade sobre consumidores comuns. Desta forma, torna-se relevante explorar e definir o potencial destes dispositivos, de forma a construir uma experiência equilibrada de Realidade Aumentada.

O objetivo desta dissertação é analisar os limites de desenvolver uma técnica de localização para um *smartphone* Android que seria rápida e precisa. Esta técnica posteriormente foi usada numa aplicação cujo propósito seria permitir ao utilizador a visualização de grandes modelos 3D, que estariam relacionados com o ambiente real (e.g uma sala poderia ser transformada num aquário).

O estudo científico focou-se inicialmente em encontrar a técnica mais correta para localizar o dispositivo, ao avaliar a precisão de algoritmos como *Inertial Navigation Systems*, *Monocular Visual Odometry* e *SLAM* num dispositivo limitado em termos de *hardware* - um *smartphone*. Os resultados obtidos com estes algoritmos não cumpriram requisitos de precisão de forma a evitar desalinhamentos entre entidades virtuais e reais.

A solução formulada consistiu em usar marcadores 2D à volta da cena real com a ajuda de uma ferramenta para computador que suporta uma funcionalidade que ajuda o utilizador a posicionar os marcadores usando cores pintadas nas superfícies do modelo 3D.

Marcadores detetados pela câmara do *smartphone* permitem a computação da posição do dispositivo. Estes marcadores são detetados pelo *Image Target Recognition* do SDK da *Vuforia*. Foram ainda efectuadas algumas modificações ao algoritmo base de *tracking* de forma a fornecer resultados mais robustos para grandes modelos 3D.

Foi desenvolvida uma arquitetura que consiste na aplicação Android e na ferramenta para computador, que, além de ajudar o utilizador no posicionamento dos marcadores, também é capaz de construir cenas de Realidade Aumentada e de carregar essas cenas para a aplicação móvel.

Alguns testes foram feitos de forma a analisar a robustez da implementação. Algumas cenas exemplo de Realidade Aumentada foram construídas de forma a investigar qualidade da aplicação em diferentes cenários, nomeadamente uma sala e uma praça e ruas adjacentes de uma vila.

CONTENTS

1 INTRODUCTION 1
- 1.1 Context 1
- 1.2 Motivation 2
- 1.3 Objectives 3
- 1.4 Document Structure 3

2 STATE OF THE ART AND RELATED TECHNOLOGIES 5
- 2.1 Android Sensors 6
  - 2.1.1 Gyroscope 7
  - 2.1.2 Accelerometer 8
  - 2.1.3 Magnetometer 9
  - 2.1.4 GPS 10
  - 2.1.5 Filters for Sensor Data 11
    - 2.1.5.1 Low Pass 11
    - 2.1.5.2 High Pass 12
    - 2.1.5.3 Kalman Filter 12
  - 2.1.6 Sensor Fusion 14
- 2.2 Image Localization Techniques 16
  - 2.2.1 Feature Detection, Description and Matching 16
    - 2.2.1.1 An Introduction 16
    - 2.2.1.2 Realtime Algorithms 18
  - 2.2.2 Monocular Visual Odometry 21
    - 2.2.2.1 Motion Estimation 22
    - 2.2.2.2 Implementations 23
  - 2.2.3 SLAM 24
- 2.3 Augmented Reality Standard Development Kits 25
  - 2.3.1 ARToolkit 26
  - 2.3.2 Wikitude 28
  - 2.3.3 Vuforia 29
  - 2.3.4 Conclusions 30
- 2.4 Summary 31

3 PROPOSAL 32
- 3.1 Navigation System 32
  - 3.1.1 Marker-Based Navigation 32
  - 3.1.2 Inertial Navigation System (INS) 34

3.1.3 Monocular Visual Odometry (MVO) 37
3.1.4 Conclusions 37
3.2 Architecture Overview 38
3.2.1 Data Exchange Format 38
3.3 AR Assistant 40
3.3.1 User Interface 41
3.3.2 Marker and Camera Creation 43
3.3.3 Object Picking 45
3.3.4 Camera Simulation 46
3.4 AR Viewer 53
3.4.1 User Interface 54
3.4.2 Application Workflow 55
3.4.3 Marker Placement 56
3.4.4 Sensor Rotation Estimation 57
3.4.5 Jitter Smoothing 57
4 RESULTS AND TEST SCENES 60
4.1 Device 60
4.1.1 Computing Power 60
4.2 Navigation System on Android Results 61
4.2.1 INS Test 61
4.2.2 MVO Test 64
4.3 Sensor Rotation Test 64
4.4 Marker Detection and Tracking Results 65
4.5 AR Projects 71
4.5.1 Natural Markers for AR Scenes 71
4.5.2 Room 76
4.5.3 Ponte de Lima 78
4.6 Performance Analysis 82
5 CONCLUSIONS AND FUTURE WORK 83
5.1 Future Work 84

A SUPPORT MATERIAL 90
A.1 Sensor Precision 90
A.2 Vuforia Developer Portal Guide 91
A.2.1 Creating a License Key 91
A.2.2 Creating Markers 92
A.2.3 Downloading Database 93
A.3 Computer App Guide 94

A.3.1 Key Bindings 94
A.3.2 Creating Camera 94
A.3.3 Creating Marker 95
A.4 Android App Guide 95
A.5 Repositories 95

LIST OF FIGURES

Figure 1 AR using see-through head-worn display (Höllerer et al. (1999)) 1
Figure 2 Global Augmented Reality (AR) Market, 2015-2021 (USD Billion) (Zion Market Research (2016)) 2
Figure 3 The requirement for special equipment for AR has changed over the years 2
Figure 4 Gyro Data 7
Figure 5 Single Integration of Gyroscope data for Rotation values 8
Figure 6 Magnetometer Data 10
Figure 7 Moving Average Filter 11
Figure 8 High pass filter 12
Figure 9 Kalman Filter 13
Figure 10 Degrees of Freedom 14
Figure 11 What are good Features? (OpenCV Docs) 16
Figure 12 Homography 18
Figure 13 FAST - Testing Interesting Points (Rosten and Drummond (2006)) 19
Figure 14 Lucas–Kanade method using Shi-Tomasi corner detection (OpenCV Docs) 21
Figure 15 Perpective Projection Model (Scaramuzza and Fraundorfer (2011)) 22
Figure 16 Epipolar Geometry (Zhang (1998)) 23
Figure 17 Kudan's SLAM tracking 25
Figure 18 Marker Tracking 26
Figure 19 ARToolkit Marker & Marker Tracking Example 27
Figure 20 Wikitude Image Target Tracking Example (Marker from Wikitude) 28
Figure 21 Extended Tracking keeps tracking when Marker is lost 29
Figure 22 User Defined Marker on Vuforia 30
Figure 23 Real Scale Scene using Vuforia With Default Marker 33
Figure 24 Natural Features as Markers 33
Figure 25 Multiple Markers 34
Figure 26 Discrete Signal integration causes errors (Seifert and Camacho (2007)) 35
Figure 27 Trapezoid Method (Seifert and Camacho (2007)) 35
Figure 28 Architecture Pipeline 38
Figure 29 Vuforia XML Structure 39
Figure 30 Ponte de Lima 3D Models (PL3D) 39

Figure 31 Computer application Modules 40
Figure 32 Computer App UI (Marker Menu) 41
Figure 33 Computer App UI (Camera Menu) 42
Figure 34 Computer App UI (Save Project Menu) 43
Figure 35 Marker Surface Alignment 44
Figure 36 Render Passes for Marker Alignment 44
Figure 37 Object Picking - Different Entity Types are saved in different Color Channels (Red for markers and Green for cameras) 45
Figure 38 ShadowMap (OpenGL-Tutorial) 46
Figure 39 Cubemap Rendering (Acko) 47
Figure 40 Camera Simulation Example 48
Figure 41 Surface normals and the surface-to-camera vector are compared and the pixel is assigned a color based on their similarity. 49
Figure 42 Shadow Acne (before and after bias) 51
Figure 43 Dealing with Aliasing 52
Figure 44 Android application Modules 53
Figure 45 Android App UI 54
Figure 46 Rendering Pipeline 56
Figure 47 Position Estimation using IMU (Environment 1) in Meters 62
Figure 48 Position Estimation using IMU (Environment 2) in Meters 63
Figure 49 Monocular Visual Odometry Trajectory Test 64
Figure 50 Rotation Estimation (Gyroscope and Rotation Vector) 65
Figure 51 Test Marker (taken from Vuforia (a)) 66
Figure 52 Different Light Environments 67
Figure 53 Different Light Environment Trajectories 68
Figure 54 Partial Occluded Marker 69
Figure 55 Markers with different Ratings 70
Figure 56 Room Trajectory 71
Figure 57 Natural Markers (3D Surface) 73
Figure 58 Natural Markers (Reflective Material) 75
Figure 59 Room Markers 76
Figure 60 Camera Simulation on Room Project 77
Figure 61 Room Project Models (Same Perspective) 77
Figure 62 Room Trajectory 78
Figure 63 Same Facade Marker at different points in time 79
Figure 64 Ponte de Lima Project (Satellite images taken from Google) 80
Figure 65 Ponte de Lima - Virtual Model at positions **A**,**B** and **C** from Figure 64 81

Figure 66 Ponte de Lima - Rotation Error 82
Figure 67 Adding a License Key 91
Figure 68 Viewing a Vuforia Key 92
Figure 69 Adding a database 92
Figure 70 Marker Creation 93
Figure 71 Viewing Marker Properties 93
Figure 72 Database Download 94

LIST OF TABLES

Table 1 BQ Aquaris E5 FHD Specs 60
Table 2 Monocular Visual Odometry Test Errors 64
Table 3 Angle Errors at A & B positions from Figure 50 for the Gyroscope and the Rotation Vector 65
Table 4 Maximum Detection/Tracking Distance using Vuforia in different lighting conditions 67
Table 5 Android App Performace 82
Table 6 Mean, Median, Variance and Standard Deviation of multiple sensors in a resting position for 30 seconds 90

1

INTRODUCTION

1.1 CONTEXT

Augmented Reality (AR) is a field in which 3D virtual objects are integrated into a 3D real environment in real time (Azuma (1997)). Figure 1 shows an early (1999) implementation of the technology, where locations in the real world are marked with virtual flags and their designations.

Figure 1.: AR using see-through head-worn display (Höllerer et al. (1999))

In recent years, improvements in computing power made the technology more robust and feasible. This development marks a huge step in technology and user interaction with gadgets and the real world. With these advancements, AR is now also viewed as a big investment potential, as the revenue from this market is predicted to grow very quickly in the coming years, as shown in Figure 2.

Figure 2.: Global Augmented Reality (AR) Market, 2015-2021 (USD Billion) (Zion Market Research (2016))

As the revenue and investment potential is very big for AR, many developers are starting to build and launch products for end consumers. But as the technology is still in its early phases, and there is no widely adopted model, not much as been done yet for it to have big impact on the population.

1.2 MOTIVATION

The recent surge of AR technology still needs to be able to reach to a normal consumer without the requirement of additional equipment. Success is ultimately determined by the achievement of this requirement. A simple example shown in Figure 3 is the comparison of the implementation from Höllerer et al. (1999) (Figure 1), which uses as equipment the MARS backpack, to Pokemon GO, a smartphone videogame which reached cult status for its AR components, having been downloaded 100 million times (Google Play Store).

(a) MARS backpack (Höllerer et al. (1999))

(b) Smartphone playing Pokemon GO (International Business Times)

Figure 3.: The requirement for special equipment for AR has changed over the years

The difference in equipment required is remarkable, and may be the deciding factor of making this technology desirable for everyday use. By using simple devices like smartphones, AR can thrive by really augmenting the everyday reality of its user, by minimizing the intrusiveness that additional equipment may require. The use cases behind this technology vary immensely, from videogames to teaching/working tools (Liarokapis et al. (2004)) or even tourism (Kounavis et al. (2012)).

But, as powerful as the smartphone is, it was not built specifically for Augmented Reality. The localization precision of an Android smartphone is essential for Augmented reality purposes, and it is crucial that its limits are very well defined, as will be detailed in this thesis.

The motivation behind this project relies in exploring the possibilities of localization of a widely available device, the smartphone, to reach the best technique which allows for a fast and precise localization of a smartphone in an area. This technique would use the tools offered by most smartphones, the IMU (sensors) and the camera.

This technique would be used in an app to view large 3D structures like monuments or fantastic environments. The user would be able to walk around while viewing the AR scene, and it would move with him, always having the 3D content perfectly align with the real environment.

1.3 Objectives

The objective of this dissertation is to build a framework which is capable of creating and visualizing AR scenes of large 3D structures. The architecture built should be modular and simple to use by a normal user. This simplicity of use would also require some tools which would facilitate the user's job, like providing helpful input when creating AR scenes.

One important focus of this dissertation is that the AR app is able to localize the device precisely and quickly in the AR scene.

Finally, it is intended to test the implementation with some test cases, one of them would be to use the available 3D models by the PL3D project of the XIV century wall around the village of Ponte de Lima, in Portugal, for the AR application.

1.4 Document structure

The current chapter, Chapter 1, is about exposing the theme of the dissertation, the motivation behind it and also aims to introduce important concepts.

Chapter 2 presents the state of the art, analyzing important concepts and related work connected to the smartphone's sensors and localization techniques.

Chapter 3 details the framework, describing the devised architecture and discussing the structure of the applications that were developed.

Chapter 4 is about demonstrating accuracy tests of the several algorithms that were implemented, from unsuccessful approaches to the final implementation. Two example AR scenes are also demonstrated in order to compare the implementation's feasibility in different environments.

Chapter 5 concludes the dissertation by summarizing the work, exposing some final assessments and also describing future tasks that could be implemented.

2

STATE OF THE ART AND RELATED TECHNOLOGIES

In this chapter an analysis will be done on key concepts related to localization techniques and the Android smartphone. As the smartphone is a limited device, choosing the right localization technique was the most challenging step of this dissertation, and thus required considerable investigation. This extensive research was made in order to pick the best technique for the AR application which provides the most balanced localization algorithm, meaning an algorithm which can identify the position of an Android smartphone in a certain area but that also accounts for some restrictions:

1. **Low Pre-Processing:** The algorithm should not require a lot of processing of the real environment prior to its use;
2. **No Additional Hardware needed:** Only the hardware that a normal consumer has should be required (an Android phone, and if needed for the construction of an AR scene, a computer);
3. **Versatile:** The technique should work for multiple environments, and not solely built around a specific location and its particularities;
4. **Precise:** As Augmented Reality merges the reality as seen by the device's camera with 3D elements, a high precision localization technique is required in order to avoid a misalignment between the reality and the virtual content;
5. **Highly Responsive:** Being a 3D Real Time application, it is required that the algorithm is capable of updating, with precision, several times per second;

There are few implementations that discuss the use of a smartphone for precise localization purposes. In the scope of related work, several localization technologies have been developed with different pros and cons. Wireless Networks have been used for localization purposes (Kulaib et al. (2011)), but this approach would require additional equipment and pre-processing. Other implementations require the building of a 3D cloud map of the real environment (Royer et al. (2005))(Lategahn (2014))(Leonard and Durrant-Whyte (1991)), which also requires large amounts of pre-processing and additional hardware.

One important step in finding the correct technique is to layout the tools at disposal for this effect. They would have to undergo a scrutiny to identify the best possible outcome. These tools are an android smatphone's Sensors (Gyroscope, Accelerometer, Magnetometer, GPS) and its monocular Camera, which are the main utensils that measure and record information about the environment that surrounds the device. Caron et al. (2006) discusses a technique which merges the Android's sensors and uses Kalman filtering for localization purposes. They also assert that the GPS suffers from a lack of credibility and that an INS (Inertial Navigation System) suffers from drift, which uses the accelerometer, magnetometer and gyroscope. In computer vision, Singh and Venkatesh (2015) describes a Visual Odometry implementation, in a monocular context (uses only one camera), which, from consecutive frames of a video feed and sensor data, can, with relative precision, calculate the location of a driving car. Burkard (2015) discusses the same algorithm implemented for Android, and suggests that the technique is flawed in certain scenarios, like moving in a curvilinear fashion and not aligning the smartphone with the horizon. Finally, still in a computer vision context, Garach (2013) discusses the use of Qualcomm's Augmented Reality SDK (hereafter referred to as Vuforia) for a mobile device. This SDK is capable of recognizing pre-determined 2D patterns, like a painting of a specific wall pattern. It was used to merge 3D virtual content with reality, namely water pipes and electric cables for engineering purposes.

In order to understand all these concepts and implementations, it is crucial to fully understand the characteristics of the tools that the smartphone provides. Thus, in this chapter, the features of Android sensors will be analyzed, and also techniques which provide the extraction of better data from them. Then, Computer Vision algorithms will be discussed, with a brief introduction to Feature Detection and Tracking (and some related algorithms), and the explanation of localization techniques like SLAM and Visual Odometry. Finally, some AR SDK's will be evaluated, assessing their overall performance and robustness.

2.1 ANDROID SENSORS

Most Android smartphones nowadays have built-in sensors which provide data about the device's motion, position, orientation and more. They are very helpful tools when referring to Augmented Reality as they provide input to estimate the pose of the device. These sensors and related algorithms will be discussed in this section.

2.1.1 *Gyroscope*

Gyroscopes are used to measure orientation, more precisely the rate of rotation. Their data on Android devices are given as radians per second around the device's own axis. It measures angular momentum, being a helpful tool in indicating orientation.

As a mechanical force has to measured into a digital environment, Microelectromechanical systems (MEMS) are used. MEMS are microscopic devices, which although are composed as circuits, are mechanical in nature. MEMS gyroscopes use the Coriolis effect to measure angle rate.

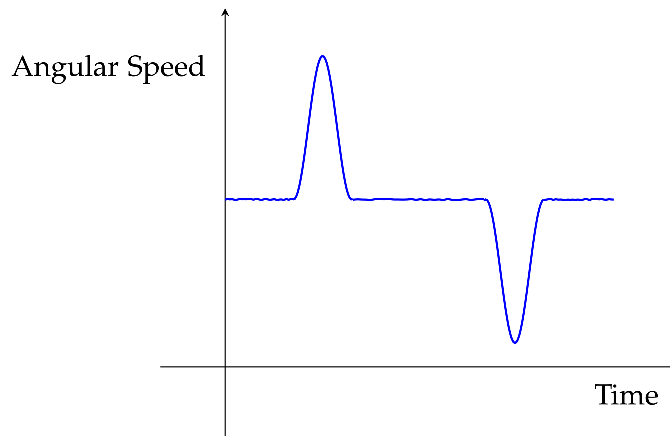


Figure 4.: Gyro Data

In an Android smartphone, gyroscope data (Figure 4) is accurate although prone to some noise. To calculate the angle of orientation from the gyroscope's angular velocity, single integration is needed. As the gyroscope measures degrees per second, angular velocity ($\dot{\theta}$) can be defined as:

$$\dot{\theta} = \frac{d\theta}{dt} \tag{2.1.1.1}$$

So, assuming that at $t = 0$, then $\theta = 0$, we can find the angular rotation θ at any given time t:

$$\theta(t) = \int_0^t \dot{\theta}(t)dt \approx \sum_0^t \dot{\theta}(t) \tag{2.1.1.2}$$

As the absolute orientation of the device is calculated by the sum of all the previous angle increments, small errors are introduced in each iteration. This concept is called drift, and increases over time.

Integration of a signal can be defined by Equation 2.1.1.3. So, after integration, if the frequency (f) of a signal is high, which is characteristic of noise, then the $\frac{1}{f}$ from the integration will turn that noise into drift, by greatly reducing the output of the signal.

$$\int cos(2\pi ft)dt = \frac{sin(2\pi ft)}{2\pi f} \tag{2.1.1.3}$$

Integrating the data from Figure 4, which represents a 180 degree rotation in two seperate 90 degree motions, we get the rotation from Figure 5. The noise from the original gyro data creates noticeable drift.

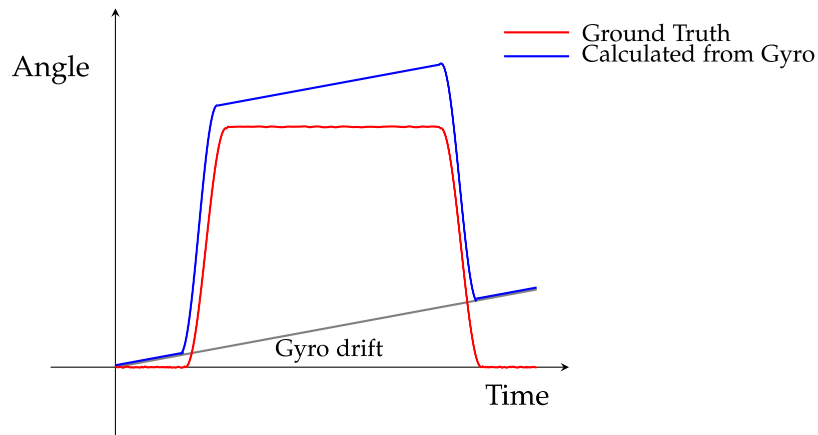


Figure 5.: Single Integration of Gyroscope data for Rotation values

Although orientation data from the gyroscope can drift, it still is useful, and, as will be seen later on, this drift can be somewhat eliminated.

2.1.2 *Accelerometer*

The accelerometer sensor always outputs the value of gravity except in free fall. It is very sensitive and prone to noise. Due to the fact that gravity is almost always a factor, it can be hard to separate it completely from other forces to calculate linear acceleration (acceleration without the force of gravity). Errors in calculating the direction of the force of gravity and linear acceleration can result in inconsistent and useless data.

An Inertial Navigation System (INS) is a system that uses accelerometers and gyroscopes to track a device's position by taking into account an initial starting point using a Dead Reckoning algorithm. A Dead Reckoning algorithm calculates the device's position by using a previous known position and estimated displacements over time. To calculate this displacement the approach would be to integrate the accelerometer data twice. But

first the gravity needs to be extracted from the data, resulting in the linear acceleration (a →acceleration,g →gravity):

$$a_{linear} = a_{original} - g \tag{2.1.2.1}$$

then, to calculate position:

$$v = \int a_{linear} dt \tag{2.1.2.2}$$

$$x = \int v dt \tag{2.1.2.3}$$

But, if a single integration on noisy input creates drift as seen in Figure 5, a second integration exponentiates that drift quadratically, resulting in even more unreliable data. Also, take into account that accelerometers are usually more noisier than gyroscopes, so the drift is more accentuated.

Implementing a navigation tracker on Android using solely an accelerometer is then completely unreliable due to the amount of drift. This is due to the fact that Android integrated sensors are consumer level and inexpensive. Reliable INS is only possible using expensive and very accurate sensors, or in controlled/limited environments. Although the accelerometer should not be discarded as an input to localize the device, relying solely on it would result in nonsensical outputs.

2.1.3 *Magnetometer*

The Magnetometer (also called magnetic sensor or compass) indicates the orientation of magnetic north in relation to the device's 3 axis. The sensor is prone to some errors since it collects data from every nearby magnetic field, even fields originated from within the android device. It is measured with micro Tesla units (μT) (magnetic flux). Figure 6 shows an example of magnetometer data. The rotation represented is the same as Figure 4 and 5.

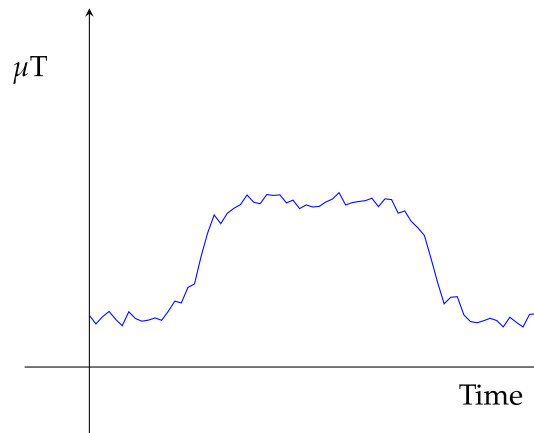


Figure 6.: Magnetometer Data

When comparing this to single integration of the gyroscope (Figure 5) to extract rotation values, some differences can be noticed. As this signal is not integrated, and because it is subject to a lot of input from nearby magnetic fields in the environment, the data is by definition very noisy, but it doesn't drift. One possible solution to provide rotation values is then to fuse gyroscope and magnetometer data.

2.1.4 *GPS*

The Global Positioning System provides geolocation to a receiver using satellites and is widely used as a location tool for a multitude of applications. The device's location is iteratively recalculated without the need for cellphone reception or Internet connection.

In indoor areas, GPS is mostly useless. This is because the device does not have a clear line of sight to the sky. The signal then distorts and weakens as it travels from the satellite to the device, resulting in inaccurate results.

Also, it has some problems when approaching the topic of precision and accuracy. Even in outdoor areas, the precision of location is not entirely accurate. In best case scenarios, it averages to about 5 to 10 meters (Zandbergen and Barbeau (2011)).

There have been implementations that aim to improve the signal GPS provides. One of the ways is to use GLONASS, the "russian GPS" together with GPS to increase the number of visible satellites to the device, increasing position accuracy and response time. When using this double ended location provider, it is said that precision increases to within 2 meters (Beebom). Although it improves the results for cluttered areas, it still has problems for indoor locations and other adverse conditions. A-GPS (Assisted GPS) is another system which aims to improve GPS performance by using cellphone towers, either by supplying orbital data (and precise time) of the GPS satellites to the receiver, which enables the receiver to find the satellites faster, or by capturing the GPS signal, and computing the positions of

receivers. Although A-GPS is an improvement to the many faults of the GPS, like location acquisition time and location acquisition in non-optimal environments, it does not fix those problems, and only offers a more optimized solution.

For the context of this thesis, GPS does not offer the precision nor reliability needed.

2.1.5 *Filters for Sensor Data*

Filters are often applied to signals in order to remove certain components of a signal. These filters are used to increase the viability of the results. Three main filter categories will be discussed: the low pass, the high pass and the kalman filter.

2.1.5.1 *Low Pass*

A low pass filter eliminates frequencies above a certain cutoff frequency and passes frequencies below that cutoff frequency. This smooths the data and ideally removes noise, making it a more clear signal and reducing unexpected behaviours. There are several types of low pass filters, in here the focus is on the moving average algorithm.

Moving average is done by calculating averages of multiple subsets of data. A simple implementation of the algorithm could be defined as:

$$MA = \frac{1}{n} \sum_{i=0}^{n-1} x_(i) \tag{2.1.5.1}$$

In figure 7 a moving average algorithm is applied to noisy data.

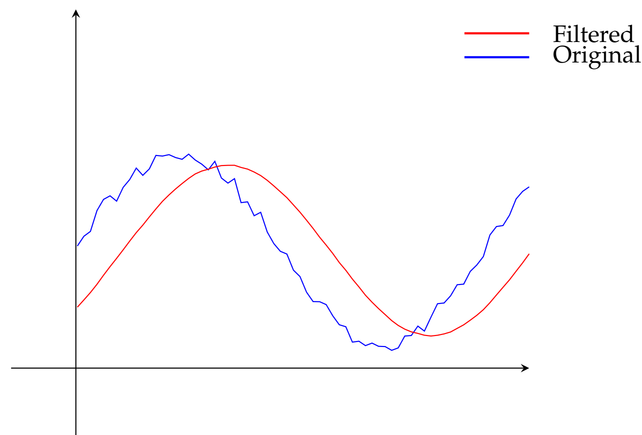


Figure 7.: Moving Average Filter

There are also versions of this algorithm that are cumulative, meaning that the average is iteratively accumulated as new data arrives, and also algorithms that use weights, for example, attributing a larger weight to newer data points.

In the context of collecting and analyzing smartphone sensor data, these filters could be used on noisy data from the accelerometer or magnetometer to smoothen the signal. Although removing unwanted components, these filters can also cause a laggy and low dynamic response, as they tend to smoothen the signal, reacting poorly and slowly to data spikes (like a sudden movement or change in rotation).

2.1.5.2 *High Pass*

A high pass filter (a counterpart to the low pass filter) is a filter that passes signals with a frequency higher than a certain cutoff frequency and attenuates signals with frequencies lower than the cutoff frequency. A simple algorithm for an high pass filter would be:

$$HP_x = \alpha \times (HP_{x-1} + x - (x - 1)) \tag{2.1.5.2}$$

In order to measure the real acceleration of the device, the contribution of the force of gravity must be eliminated. This can be achieved by applying a high-pass filter. So, while the low pass filter is used to remove small variations (like noise) in the signal, an high pass filter is used to remove large influences, like a constant bias the sensor has, or the force of gravity for example. An example of an high pass filter applied to data can be seen in Figure 8. The differences with the low pass filter can be seen by comparing it to Figure 7.

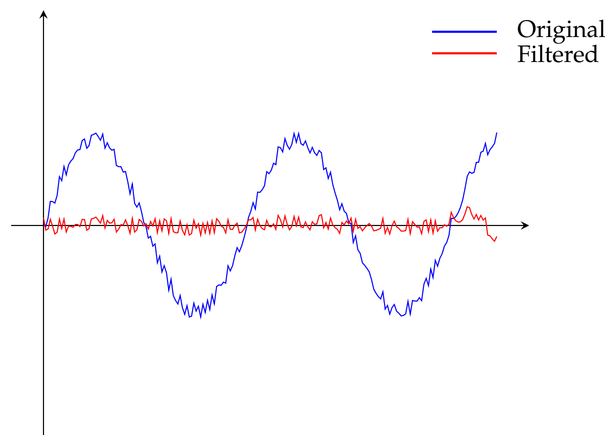


Figure 8.: High pass filter

2.1.5.3 *Kalman Filter*

Kalman filter is an algorithm that estimates the value of a variable based on measurements over time, which contain some noise. It is used to remove unwanted components of a signal, and is widely used to predict values. The algorithm is separated in three major calculations: the calculation of the Kalman gain, the calculation of the current estimate and the calculation of estimate error. New estimates are recursively calculated as new

measurements arrive. An example can be seen in Figure 9, where a kalman filter is applied to noisy data from measurements. After several samples the kalman filter aproximates very precisely to the true value.

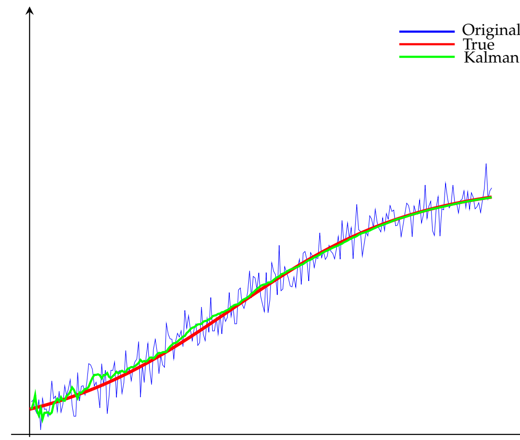


Figure 9.: Kalman Filter

The Kalman Gain is used to determine the ratio of new measurements to use in a new estimate. It is calculated by:

$$KG = \frac{E_{EST}}{E_{EST} + E_{MEA}} \tag{2.1.5.3}$$

$$0 \leq KG \leq 1 \tag{2.1.5.4}$$

E_{EST} is the error covariance, denominated as the error in the estimate and E_{MEA} is the standard deviation of noise expected from the data, here called the error in the measurement. Based on the equation, gain can be described as directly proportional to the accuracy of measurements, but inversily proportional to the accuracy of the estimates. This will impact the weight of the measurement in the following estimate calculation.

The estimate calculation is used to predict the behaviour of the observed entity. To calculate the current estimate:

$$EST_t = KG * MEA + (1 - KG) * EST_{t-1} \tag{2.1.5.5}$$

EST being the estimate and MEA the measurement. So, if the gain is large (close to one), then the measurement will have a bigger impact on the current estimate than if it was small (close to zero). This can be explained by what was previously said about the gain being proportional to the accuracy of measurements but inversily proportional to the accuracy of estimates. If the measurement is accurate and the estimate is not, then the measurement should have a big impact on the estimate calculation. On the other hand, if the previous

estimate is accurate and the measurement is not, then the measurement should not have a significant impact on the current estimate (only small corrections should be applied). As a final step in the iterative process, the error in the estimate can be calculated:

$$E_{EST_t} = \frac{E_{MEA}E_{EST_{t-1}}}{E_{MEA} + E_{EST_{t-1}}} \Rightarrow E_{EST_t} = (1 - KG)E_{EST_{t-1}} \tag{2.1.5.6}$$

2.1.6 *Sensor Fusion*

Each sensor has its limitations. Sensor fusion is the concept of fusing multiple sensor data in order to compensate for each's weakness. It is widely used in smartphone applications.

In an Augmented Reality context, this could be used to solve problems associated with Deegrees of Freedom (DOF). DOF defines the number of parameters in which the device can be tracked. So, 3DOF could refer to the rotation in Three Dimensional Space and 6DOF could refer to that 3DOF plus translation motion in 3D space. This can be visualized in Figure 10.

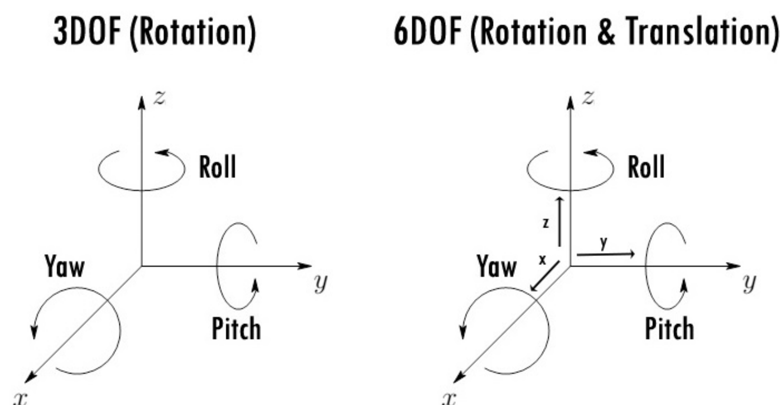


Figure 10.: Degrees of Freedom

To implement 3DOF as Figure 10, sensors that measure orientation could be used. But as we've seen before, although the gyroscope, magnetometer and even accelerometer fit that category, each suffers some problems, the gyro's weakness being drift over time, and the magnetometer's and accelerometer's being noise. This is where sensor fusion comes into play. By using all of these sensors together, we can somewhat deal with both drift and noise, albeit not entirely. The approach is to use the magnetometer to indicate the magnetic North, the accelerometer would indicate the gravity vector (down vector of the device in relation to earth), and the gyro would provide quick response. One approach is to apply low pass filtering to magnetometer and accelerometer data (after calculating the vector of gravity),

offering a slow but reliable response and use the gyroscope for quick but unreliable data over long lengths of time.

So, large state changes should be monitored by the magnetometer and accelerometer, while the gyroscope should account for changes in small time intervals.

Another example of sensor fusion would be to implement a 6DOF (Rotation & Translation) algorithm by using the previously discussed 3DOF algorithm and also a GPS fused with an Accelerometer to track motion of a device. One approach for ALV's (Autonomous Land Vehicles) was to use Kalman Filtering because GPS/INS fusion was not satisfactory by itself (Caron et al. (2006)). The main problem in using this approach on a smartphone is achieving reliable positioning data from the IMU. This is due to the nature of a smartphone when compared to a robot or vehicle. Besides the accuracy level of the sensors, which are more precise in ALV's, robots and vehicles have a much more consistent and stable motion, like having a mounted camera which doesn't shake, limited movement (only back and forth), and mostly stable velocities and accelerations, which produces less jittery sensor data. The smartphone is usually handheld. This is much more unstable, as hands shake and a person's movement is jerky, which ends up producing more error.

2.2 Image Localization Techniques

In this section several approaches that could be taken to track a device's movement, as precisely as possible with the hardware available on most Android smartphones will be described, meaning that algorithms that use depth sensors, laser sensors, and stereo camera setups will not be regarded. These are Computer Vision algorithms, so first their building blocks will be described, like Feature Detection and Tracking, and several techniques will be analyzed. Then the navigation algorithms themselves will be explained: Simultaneous Localization and Tracking (SLAM) and Visual Odometry, specifically Monocular Visual Odometry (MVO).

2.2.1 *Feature Detection, Description and Matching*

2.2.1.1 *An Introduction*

The concept of identifying an image or an object or even a scenery is something that seems very simple for a human being, but when we think about implementing this concept in a machine, there are some questions that arise. What makes an image identifiable? What makes an object or scenery identifiable even from different viewpoints and in different lighting conditions? The answer is to abstract image information in the form of features. Features in computer vision are usually described with the very vague definition of being the "interesting" part of an image. Over the years, many algorithms appeared that implement their own idea of what a feature really is. There are specific characteristics that a good feature must have. Take a look at Figure 11:

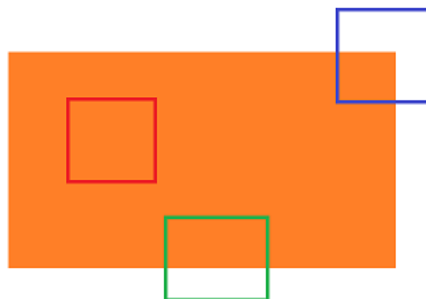


Figure 11.: What are good Features? (OpenCV Docs)

Let's assume that there are three possible features, all surrounded by the coloured squares. The feature from the red square would be an example of a bad feature. Wherever you would move the square inside the larger orange square, the feature would remain unchanged. So, good features need to be distinguishable from the surrounding image. In the case of the green square, it is located at an edge. This creates a problem wherein moving the square

sideways along the gradient would not change its content. This means that good features need to be localizable. The blue square represents a good feature. Corners are distinguishable and localized, and in this example, they are different from everywhere else. So, most Feature Detection algorithms focus on locating corners.

One of the first algorithms to try to achieve this was the Moravec Corner Detection algorithm (Moravec (1980)). The algorithm worked on small windows of an image, and calculated the difference in intensity along the vertical and horizontal axis. If the difference in both axis is larger than a threshold then a corner was present. If it was large on only one axis, then an edge was present. Still, it had some problems like rotation variance, meaning that a rotation of an image or corner would produce different results. The Harris Corner Detector (Harris and Stephens (1988)) fixed that by using eigenvalues. Later there were some improvements to the corner scoring algorithm which improved tracking results (Shi). The problem with this later algorithm is that it is not scale invariant, as a change in scale of an image would produce different results.

Then came SIFT (Scale Invariant Feature Transform) (Lowe (2004)). This algorithm took into account scale invariation and also allowed for very versatile feature detection. The algorithm is separated in four major steps: Scale-space Extrema Detection, Keypoint Localization, Orientation Assignment, and Keypoint Descriptor. The keypoint detection algorithm uses a cascade filtering method and the first stage of the algorithm is finding locations which are invariant to scale, which means finding stable features across all possible scales. Gaussian filters are convolved with the image with different σ, blurring the image at different magnitudes. A DoG (difference of Gaussians) is calculated between two nearby scales, and finally these results are searched for local extrema by comparing each pixel to its neighbours in the current image and the pixels in the neighbouring scales. If it is a local extrema, then it is considered a candidate keypoint. The second step, Keypoint Localization, determines and discards bad candidates such as low contrast keypoints using a quadratic Taylor expansion of the scale space function and edge keypoints using a similar scoring algorithm to the harris corner detector. Orientation assignment achieves invariance to rotation and the final step, keypoint description transforms the keypoint into a representation that allows for significant shape distortion and change in illumination. SIFT was a breakthrough in Feature Detection, but it was slow. SURF (Speeded-Up Robust Features) (Bay et al. (2006)) aimed to do what SIFT did, but faster. Results showed it was three times faster than SIFT, but wasn't as good in viewpoint changes or illumination changes. Also, it still was not fast enough for realtime applications, and in the context of this dissertation, there is a necessity for a feature detector that fills this requirement.

FAST or Features from Accelerated Segment Test was proposed to solve efficiency problems in realtime applications with limited computational power (Rosten and Drummond (2006)). Feature descriptions act as a firgerprint for a feature. They can be used to compare

features. SIFT and SURF implemented their own feature descriptors, but both suffer limitations as SIFT descriptors take about 512 bytes, and SURF's take a minimum of 256 bytes. In the context of using for example, one million features, that would take about 500 Mb and 250 Mb respectively. There are several methods that are used to compress it using binary strings (Ke and Sukthankar (2004)), which also improves efficiency since CPU's work faster with binary computations. So, BRIEF (Binary Robust Independent Elementary Features) appeared that could compute these binary strings directly without the need of a default descriptor (Calonder et al. (2010)).

Feature Matching means to match features of the same object/scene, in different images. There are many algorithms of feature matching, such as Brute Force, which takes the descriptor of one feature, and matches the closest feature in a set. There are also nearest neighbour algorithms which have increased efficiency for large datasets. One of the use cases of feature matching is to calculate a perspective transform between two images, or homography. This concept is represented in Figure 12.

Feature detection and matching is used in a multitude of algorithms and serves multiple purposes. It may be used to stitch images to create panoramic shots, to facial recognition, and also may be used to estimate movement in a video. This last concept is called optical flow, and it consists in estimating the 2D vector of movement of a certain feature between two frames.

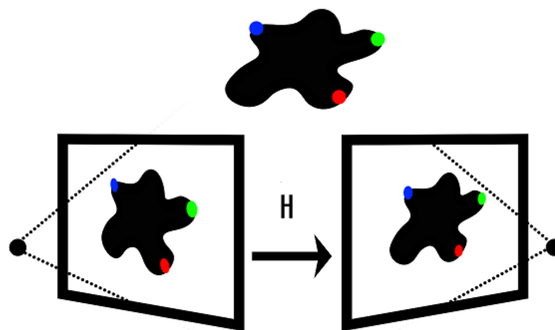


Figure 12.: Homography

2.2.1.2 *Realtime Algorithms*

In the purpose of this dissertation it would only make sense to choose realtime algorithms, so I will detail three algorithms: FAST, ORB and Lucas–Kanade Optical Flow.

FAST

As mentioned before, FAST solved efficiency problems that algorithms like SIFT had. The algorithm works as follows:

1. Select pixel p;
2. Extract pixel intensity I_p;
3. Select Threshold t;
4. Consider 16 pixels around p (Figure 13);
5. Select N number of sample pixels (authors considered 12);
6. If N contiguous pixels around p have an intensity above $I_p + t$ or below $I_p - t$ then p is an interest point;
7. Repeat for all image pixels;

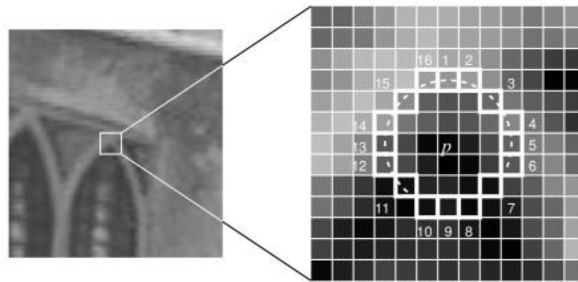


Figure 13.: FAST - Testing Interesting Points (Rosten and Drummond (2006))

The algorithm of comparing each pixel intensity to its surrounding pixels, works in a way to maximize efficiency. First, pixels 1, 5, 9 and 13 (Figure 13) are compared with I_p. If 3 of those are neither below or above the threshold criteria, then p cannot be an interest point. Else, check all pixels for N contiguous pixels that fit the criteria. As told by the authors, the algorithm has a few limitations. If $N < 12$, the number of detected points is very high, and the order in which the surrounding pixels are tested majorly influences efficiency. So, a machine learning approach was implemented in order to deal with these problems. This approach works in the following way:

1. Take a set of images to train;
2. Run the FAST algorithm on each image;
3. For each pixel p, store the pixels surrounding it in a vector;
4. Each of the vector pixels, can have three states, d (darker than p), s (similar to p), and l, lighter than p, which are defined by comparing each pixel with the threshold criteria of p. Then, we will have three subsets of the vector, V_d, V_s,V_l;

5. Determine K_p using FAST's feature detection algorithm, which is true if p is an interest point, and false if not;

6. Using a decision three qualifier (ID3 algorithm), query each of the subsets with K_p multiple times, in order to minimize the number of queries that FAST needs to make for each pixel.

A final problem of the algorithm was the detection of multiple interest points close to each other. Non-maximum suppression is applied to deal with this. The algorithm works in the following way:

1. For each interest point compute V as the sum of absolute differences between the interest point and the surrounding 16 pixels;

2. Compare two adjacent interest points

3. discard the one with the lower V

The scoring method can be changed, as long as it is replaced by another method which compares two adjacent points and removes the less significant one.

ORB

Oriented FAST and Rotated BRIEF or ORB was proposed as a fusion of FAST and BRIEF, with some modifications, enhancing overall performance (Rublee et al. (2011)). One of the unsolved problems of FAST was that it was not rotation invariant. To fix this, corner orientation was implemented (orientation provides rotation invariance) using an intensity centroid approach. This approach assumes that each corner has a radius r which is defined by the magnitude of the orientation vector, providing thusly rotation invariance. BRIEF also has a problem with rotation, it performs poorly with it. So the authors (Rublee et al. (2011)) contributed with rBRIEF, which computes descriptors efficiently with rotation.

Lucas Kanade Optical Flow

When referring to consecutive images caused by the movement of a camera (a video for example), optical flow can be very efficient in tracking features, assuming that the displacement between frames is not very large. Firstly we extract features of an image using a feature detector (like FAST) , and in the consequent image, the Lucas Kanade algorithm tracks those features (Lucas et al. (1981)). It assumes that the displacement between two frames is small and constant between a neighbourhood of the point p.

Figure 14.: Lucas–Kanade method using Shi-Tomasi corner detection (OpenCV Docs)

2.2.2 *Monocular Visual Odometry*

Visual Odometry is a computer vision technique that is used mostly in robot navigation applications. It consists in estimating the ego-motion, the motion estimation of a camera system, of a vehicle. It appeared in the 1980's with Moravec's work (Moravec (1980). There were several early iterations, all based on the problem of tracking motion of planetary rovers (in the context of the NASA Mars exploration program), where there were problems in using wheel rotation estimations due to wheel slippage in sandy terrain. Most research was done using stereo cameras, two cameras which together can estimate depth information of the environment. Still, there is a global interest in monocular algorithms, since the stereo case degenerates to the monocular case when the distance to interest points is very large when compared to the distance between the cameras (stereo baseline). In the context of this dissertation it only makes sense approaching single camera algorithms because most smartphones only have one. The main problem with monocular VO, is that 3D motion has to be computed from 2D visual data. Since from 2D images absolute scale can't be inferred (i.e from a 2D picture of a building it is impossible to calculate its size without a frame of reference), an alternative method must be used to retrieve this data.

The most successful approaches, in terms of performance and robustness have been feature-based, with one of the first real time implementations of MVO using RANSAC and 3D-to-2D pose estimation (Nistér (2005)). RANSAC is able to remove outliers efficiently, producing better results when calculating the essential matrix (which relates corresponding points in different images). Similar approaches have been taken using perspective (Nistér et al. (2006)) and omnidirectional (Corke et al. (2004)) cameras, although omnidirectional cameras are not present in everyday smartphones, so that option must be discarded. It is assumed the perspective camera is a pinhole projection system (Figure 15), meaning that each pixel of an image corresponds to the intersection of light rays from the object through

the center of the lens (center of projection) with the focal plane (view plane). The direction of the vector of the ray is then affected by the (u,v) image coordinates of p and the camera's focal length.

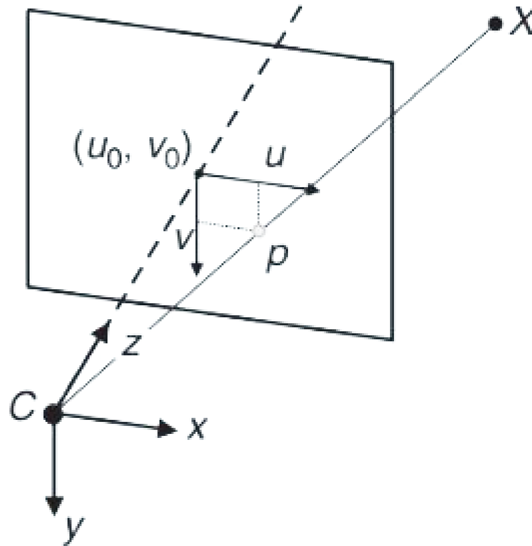


Figure 15.: Perpective Projection Model (Scaramuzza and Fraundorfer (2011))

On a Visual Odometry context, since the camera position is calculated based on the previous one, and as small errors are introduced in each iteration, drift builds up to considerable amounts in large trajectories, varying on the hardware quality and algorithm implementation.

2.2.2.1 *Motion Estimation*

Motion Estimation of MVO can be calculated using a set of consecutive images I_i and I_{i-1}, which are feature-matched:

1. New Image I_i
2. Detect and match features with I_{i-1};
3. Compute Essential Matrix & Pose to extract relative Rotation and Translation;

   The Essential matrix is used to estimate the camera's relative pose, which describes how the camera is situated in 3D relative space. It is computed from 2D-to-2D features matches. The method of estimating where a feature, which is matched in two images, is located in 3D space is called epipolar constraint, its concept can seen in Figure 16. m and m' correspond to the same feature and M is the respective point in 3D Space

joined by lines which represent the rays using the perspective projection model. The motion estimation is then calculated using a set of feature correspondences and their epipolar constraints (like Nister's five-point algorithm [Nistér (2004)]).

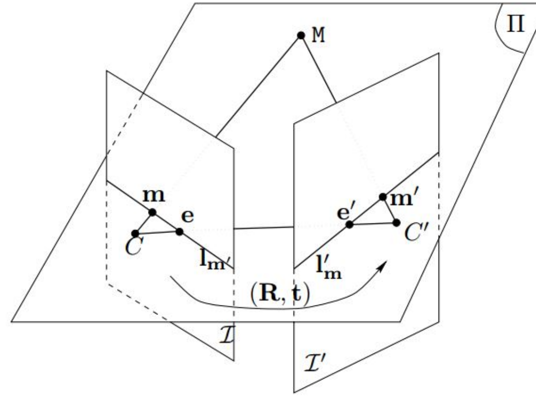


Figure 16.: Epipolar Geometry (Zhang (1998))

After calculating the Essential Matrix, the relative camera's pose is calculated so R and T (rotation and translation) can be extracted.

4. Compute relative and absolute scale and concatenate transformation to previous position;

   To concatenate transformations, relative scale between transformations must first be calculated. One way to do this is to triangulate 3D points from consecutive images. Then, distance between features f_i and f_{i-1} from images I_k and I_{k-1} can be calculated between corresponding 3D points:

$$d = \frac{\|f_{k-1,i} - f_{k-1,j}\|}{\|f_{k,i} - f_{k,j}\|} \tag{2.2.2.1}$$

   Absolute scale can also be calculated using external available data. On a smartphone context, IMU data from accelerometers can be used as a scaling factor to input the magnitude of motion that took place.

   Finally, the transformation must be appended to the previous transformations, resulting in a set of motions through 3D space over time.

2.2.2.2 *Implementations*

There are libraries available such as libviso2 which implement monocular and stereo visual odometry. For the monocular case this library uses an 8-point algorithm for essential matrix estimation and is still very limited. Work done by Singh and Venkatesh (2015) uses the

Lucas-Kanade method with FAST feature detection to compute 2D-to-2D motion estimation. Burkard (2015) uses a similar method using a smartphone, confirming the algorithm's capability of being computed on a Android device, although with some precision issues.

2.2.3 *SLAM*

Simultaneous Localization and Mapping (SLAM) is an algorithm concerned with building an environment by a device (mostly used for robots) while it simultaneously localizes the entity in that environment. It was developed for mobile robot navigation and consisted of tracking geometric beacons (naturally occurring features) (Leonard and Durrant-Whyte (1991)). It can be implemented using different hardware just like Visual Odometry, such as a stereo camera setup, laser sensors and depth sensors.

The algorithm consists in several steps: landmark extraction, data association, state update and landmark update. The idea is to extract features from the environment, and, when the device moves, the algorithm tracks the movement of those features, most commonly using an Extended Kalman Filter (EKF) to estimate (with some uncertainty) the position of the device. Each iteration, landmarks (features) are extracted and added to the Kalman filter. As in Visual Odometry, the problem in a monocular context is assessing depth and scale of the features.

SLAM is a hardware heavy algorithm and precise results in a monocular, hardware-limited device like most smartphones is practically impossible. Still, there are some developers which have used heuristic approaches to develop workable solutions. Kudan's SDK is an example of that as it uses a SLAM algorithm for its markerless tracking feature. But a simple test using their demo app, leaves no doubt that it can't be used in the context of this thesis, as perspective changes produce large errors. In the example of Figure 17, a shuttle is placed on a table for markerless tracking, and the device is moved around the table, always pointed to its center. The large amount of perspective error in a small amount of time is very considerable. Although the quality of the algorithm is excellent considering the hardware it is being processed on, the precision is not sufficient for the requirements of this dissertation.

(a) Kudan intitialization (b) Kudan after perspective changes

Figure 17.: Kudan's SLAM tracking

2.3 AUGMENTED REALITY STANDARD DEVELOPMENT KITS

Many Augmented reality SDK's provide localization tools with visual markers using feature detection and tracking. This concept consists in pointing the device's camera at a 2D image or marker (printed figure on flat paper for example), such that the device recognizes the marker, and consequently calculates the camera's pose. By knowing the camera's position and rotation relative to the marker, it is then possible to draw 3D entities in the device's screen which change accordingly to the camera's pose relative to the marker. An example is shown in Figure 18.

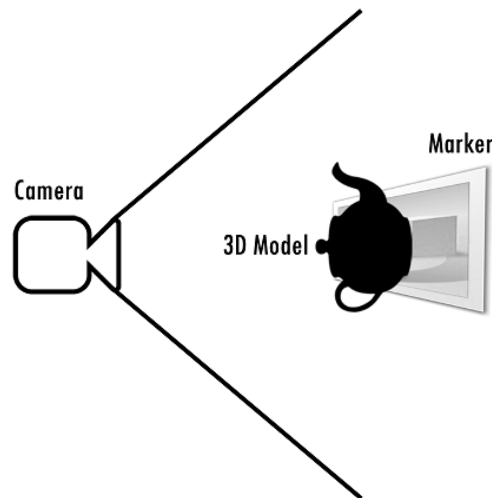


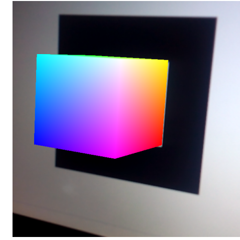
Figure 18.: Marker Tracking

There are several AR SDK's available, and to choose one tests were made with several SDK's to understand each's advantages and disadvantages in order to make the right choice. Every SDK chosen, shared at least one mutual feature: 2-D image detection and tracking. Three SDK's were tested. The criteria for choosing them were the overall popularity and positive feedback seen on the web. From those three, only one was chosen. The decision was based on multiple factors, which will be explained further on.

2.3.1 *ARToolkit*

Referred to as the most widely used tracking library for augmented reality (ARToolkit), ARToolkit is definitely the most recognizable of the bunch. The project has been open source since 2001, which explains its reach. It is available for Windows, Mac OS, Linux, iOS and Android. The 2d image tracking is usually done using Traditional Template Square Markers (marker system of ARToolkit), characterized by square markers with white padding, black border, and a pattern. An example of the marker and an use case can be seen in Figure 19.

(a) ARToolkit Marker



(b) ARToolkit Tracking Example

Figure 19.: ARToolkit Marker & Marker Tracking Example

One very important problem to solve, in this dissertation's context, is calculating how far, in real scale, the device is from the marker. The Triangle Similarity approach is usually used for this purpose, the algorithm (Rosebrock) is as follows: If we have a certain Marker with W width, at D distance from the camera, and capture one frame of the marker, which will have P width in pixels in the photo, one can calculate the Focal Length F of the camera:

$$F = \frac{(P \times D)}{W} \tag{2.3.1.1}$$

Based on the computed Focal Length, any consequent distance can be calculated by:

$$D' = \frac{(W \times F)}{P} \tag{2.3.1.2}$$

Unfortunately, ARToolkit's API doesn't provide tools to discover the width of the marker in pixels, so another approach must be taken. This approach is described by converting the virtual distance to the marker in real distance. What ARToolkit does provide is a transformation matrix from the Camera to the Marker. Also, the ARToolkit Marker width corresponds to a certain number of virtual units (40 virtual units). If we measure the real width RW and the virtual width VW of the marker, and extract the Translation Vector TV from the Matrix, we can then calculate the real distance vector RD by:

$$RD = \frac{RW}{VW} \times TV \tag{2.3.1.3}$$

The problem with ARToolkit are its limitations. For it to recognize 2D images which are not based on its marker template, there are several steps that need to be taken, like training ARToolkit to recognize the image. On Android, camera calibration needs to be made for it to work properly, and even so, the camera calibration application (made by ARToolkit) tends to "uncalibrate" the camera even more. Also, it lacks some features that modern AR SDK's have, like Extended Tracking (explained later on), object tracking and 3D tracking.

2.3.2 *Wikitude*

Wikitude is a fairly new AR SDK, launched in 2012. It was founded in 2008 and focused on a location based AR app until changing its aim to developing a SDK. It is available for Android and iOS and its features include 2D image recognition and tracking, extended tracking and 3D Recognition. The 2D images can be of any kind, and are not limited like the ARToolkit markers. An example can be seen in Figure 20.

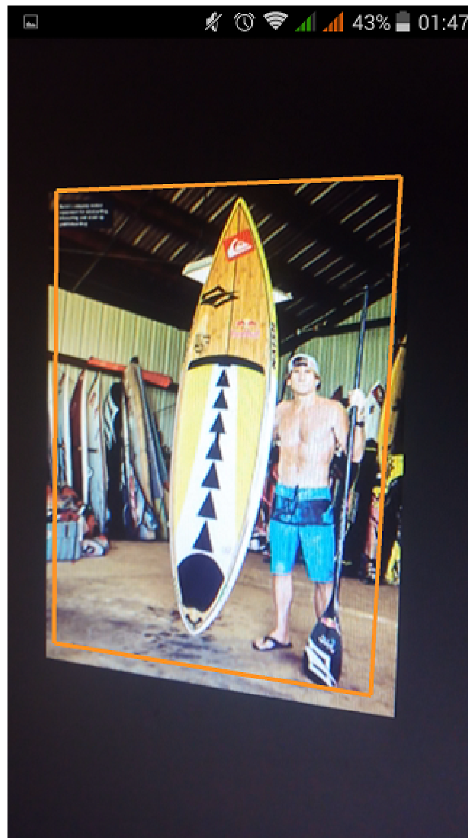
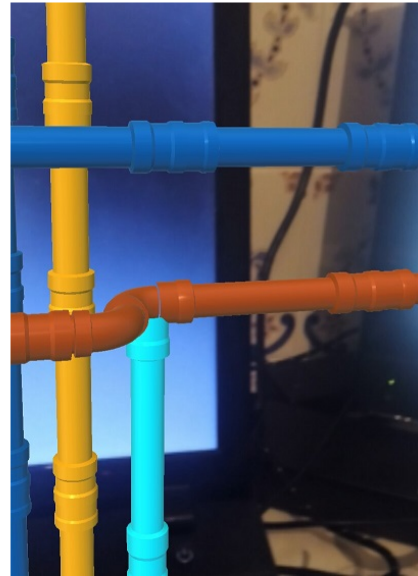


Figure 20.: Wikitude Image Target Tracking Example (Marker from Wikitude)

Wikitude also allows for multiple marker tracking at the same time.

Extended Tracking keeps calculating the camera's position and orientation when the marker has been lost by the algorithm, using clues from the surrounding environment. A visualization of the feature can be seen in Figure 21.

(a) Extended Tracking - Marker is Visible



(b) Extended Tracking - Marker is not Visible

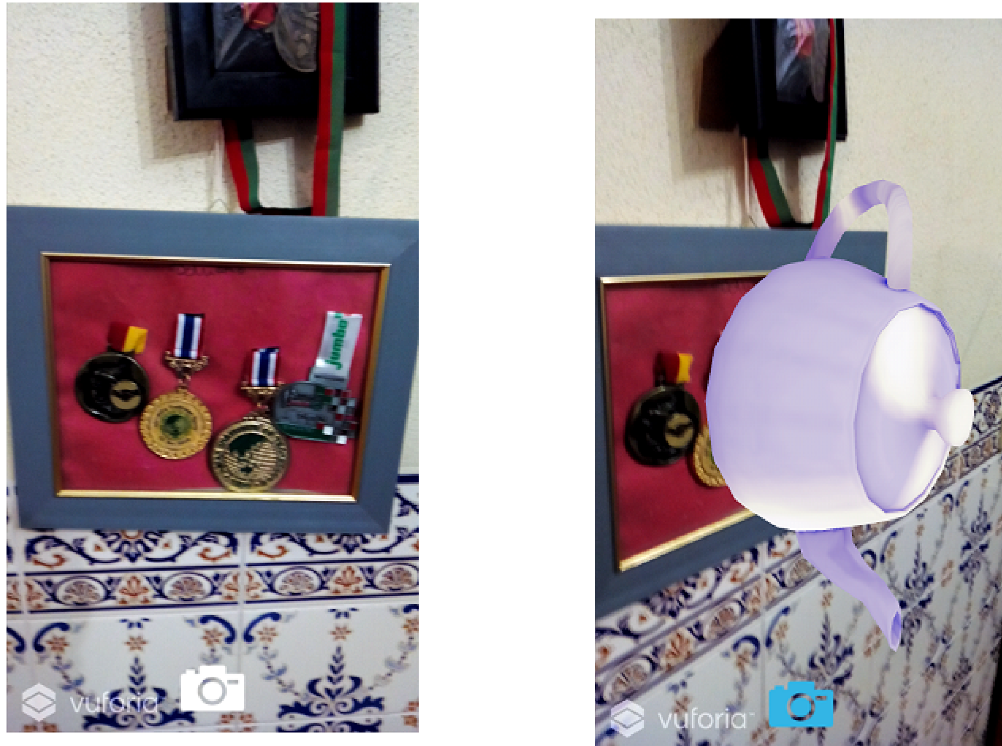
Figure 21.: Extended Tracking keeps tracking when Marker is lost

3D recognition means that the SDK can track the device's movement through a predefined and mapped space.

Tests made with the SDK concluded that although very promising, the SDK still has alot to accomplish, since most of its features are still in very early phases, and aren't robust enough for reliable use, like 3D tracking and Extended Tracking.

2.3.3 *Vuforia*

Vuforia's SDK (available for Android, iOS and Windows Platforms) provides many features including 2D Image Tracking, Multiple Image Tracking and Extended tracking as Wikitude, and also Object Recognition, User Defined Markers, Text Recognition, and more. User defined Markers allows for a runtime definition of a marker by the user of the device, which allows for many use cases. One example could be using a unique pattern on a wall as a marker while in runtime. An example can be seen in Figure 22.



(a) Creating a User Defined Marker while in Runtime

(b) Tracking of User defined Marker

Figure 22.: User Defined Marker on Vuforia

Vuforia also includes an online Database (to create markers) and Unity integration, making it easier for the developer to create AR aplications.

When developing real scale virtual worlds, one has to take into account that the ratio between the virtual marker size, V_{marker}, and real marker size, R_{marker}, and the ratio between the virtual world size, $Vworld$, and the real world size, $Rworld$, has to be the same.

$$\frac{V_{marker}}{R_{marker}} = \frac{V_{world}}{R_{world}} \tag{2.3.3.1}$$

2.3.4 *Conclusions*

Although the many different AR SDK's provide multiple features which seem useful for the context of this dissertation, there is a clear lack in precision and stability in features like Extended Tracking across all SDK's.

After brief experiments, it can be inferred that the only feature which provides reasonable precision and stability, for this context, is Image Tracking. And along all the sample applications, Vuforia's Image Tracking seemed to be the most robust of all the AR SDK'S. For this reason, Vuforia's AR SDK was chosen for the Android app development.

2.4 SUMMARY

This chapter exposed multiple concepts that relate to identifying how a smartphone is situated in 3D space, from IMU sensors, which can provide information about the rotation and motion of a device, to computer vision algorithms which can identify and calculate the motion of a device from two consecutive camera frames, and finally Augmented Reality SDK's which provide detection and tracking algorithms of 2D images.

The next chapter will gather all this information and describe the best achieved architecture which merges the discussed topics.

3

PROPOSAL

In this chapter the developed solutions are exposed and described, from their architecture to implementation details, as the multiple concepts approached in the State of the Art are merged as core pieces for the Android Application, and a complementary Computer Application is described which serves as a tool to build AR scenes. Section 3.1 presents an exploration of three techniques that could localize a smartphone in an area: a marker-based technique, a INS technique and MVO. These two last algorithms are ultimately discarded based on precision data from tests. Section 3.2 presents the architecture of the implementation and the file format devised which contains all data of an AR scene. Section 3.3 describes the AR Assistant implementation (computer app) and Section 3.4 describes the AR Viewer implementation (Android app).

3.1 NAVIGATION SYSTEM

In this section three different navigation systems are discussed which are able to localize a smartphone in a certain space by recurring to the device's camera and/or sensors. A marker-based technique is described and two additional algorithms which aim to assist it are described and tested.

3.1.1 *Marker-Based Navigation*

As a fast and precise localization technique was needed, the most robust option was using Vuforia's 2D Image Tracking Software to quickly localize a device in a certain space, while the smartphone rendered the desired 3D Model in the correct location. This idea can be illustrated in Figure 23.

(a) Camera View (Marker is marked in green)

(b) 3D Render View

Figure 23.: Real Scale Scene using Vuforia With Default Marker

Although Figure 23 illustrates the purpose of using Markers as localization tools, it displays the intrusion of the user in the real environment, by requiring the placement of an artificial Marker in it (marked in green). It is desirable that this intrusion does not occur, and that instead, natural occurring features of the environment are used, like shown in Figure 24 (marker also marked in green).

(a) Natural Marker example

(b) Extracted Features from Natural Marker

(c) Camera View (Marker is marked in green)

(d) 3D Render View

Figure 24.: Natural Features as Markers

The implementation is based on multiple markers, meaning that an AR scene could contain multiple natural markers across its real environment, serving as anchor points for the Android Application (Figure 25).

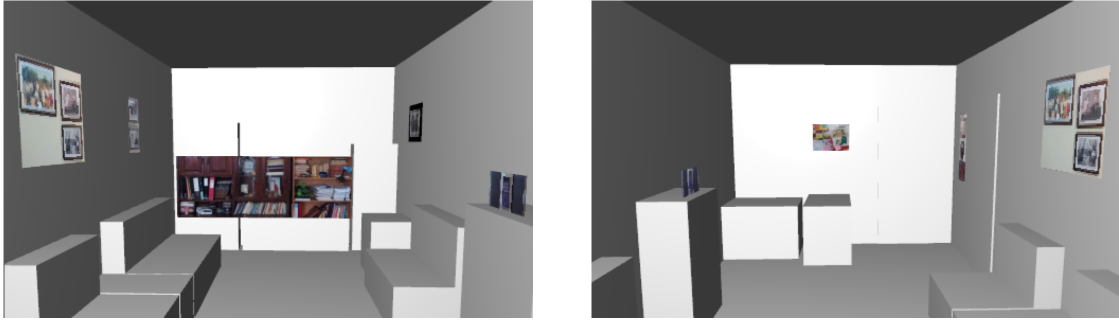


Figure 25.: Multiple Markers

This allows for a continuous updated position and rotation of the device while a marker is being tracked. However, if none of the markers are visible it is no longer able to properly perform these updates. Thus, there was a need for the existence of a navigation algorithm using the Android's sensors and/or camera when no marker could be tracked. Two options were explored to solve this problem, an Inertial Navigation System algorithm and Monocular Vision Odometry.

3.1.2 *Inertial Navigation System (INS)*

As approached in the State of the Art chapter, an INS provides navigation input using sensors like accelerometers and gyroscopes, which can be converted to rotation and translation parameters. In this section a positioning algorithm using the Android's accelerometer is presented.

As described in the previous chapter, to calculate the position of a smartphone device using sensors, double integration needs to be applied to accelerometer values, which can be a daunting task, producing large amounts of errors. To reduce this, integration was done using the trapezoid method. This is because assuming a signal f, its integral is the area below its curve, and as the magnitude of f arrives in intervals and not as a continuous signal, small areas are created between samples. This creates errors in each sequence which accumulates over time. This is illustrated in Figure 26.

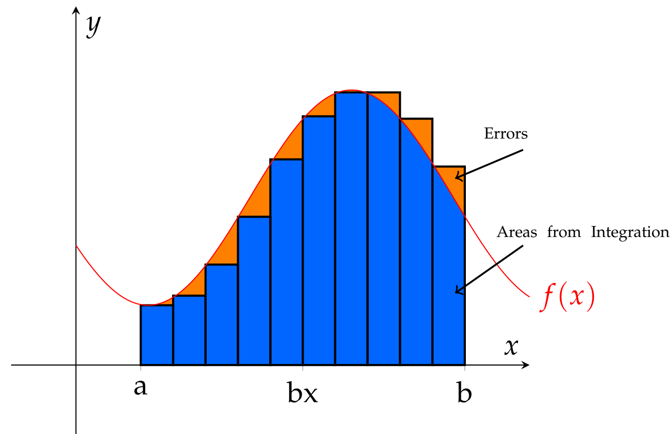


Figure 26.: Discrete Signal integration causes errors (Seifert and Camacho (2007))

To reduce the sampling losses, the trapezoid method assumes that the area is not a rectangle, it's a trapezoid. This results in what can be seen in Figure 27.

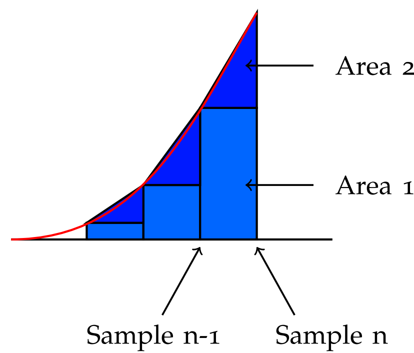


Figure 27.: Trapezoid Method (Seifert and Camacho (2007))

Although this does not completely remove errors in integration, and does not handle errors in the data itself, it is still a very simple method which is widely used in this scenario. The formula which defines it is:

$$A_n = S_{n-1} + \frac{|S_n - S_{n-1}|}{2} \times t \tag{3.1.2.1}$$

So, to calculate position and velocity with acceleration values:

$$v_n = v_{n-1} + a_{n-1} + \frac{a_n - a_{n-1}}{2} \times \Delta t \tag{3.1.2.2}$$

$$p_n = p_{n-1} + v_{n-1} + \frac{v_n - v_{n-1}}{2} \times \Delta t \tag{3.1.2.3}$$

Different increments of the positioning algorithm were created to assess the impact that small changes made. The first one is described in Algorithm 1.

```
while a= accelerometerValue() do
    v= previousV + previousA + (a - previousA)/2 * dt ;
    p= previousP + previousA + (v - previousV)/2 * dt ;
end
```

Algorithm 1: Raw Position

In the second algorithm an initial estimation of the accelerometer bias is done. This can be achieved in several ways. Recorded data from the device's resting position was used to calculate the bias by determining the average of all its samples. That is the best available approximation to the bias of the sensor. An average is also applied for each N samples (N=3 was used). The algorithm is described in Algorithm 2.

```
biasOffset=retrieveOffset();
while a= accelerometerValue() do
    vector.removeFirstElement();
    vector.push(a- biasOffset);
    finalAcc= mean(vector);
    v= previousV + previousA + (a - previousA)/2 * dt ;
    p= previousP + previousA + (v - previousV)/2 * dt ;
end
```

Algorithm 2: Position with Moving Average and Bias Removal

Finally, a window of discrimination is applied. This removes small noise from the sensor and allows for somewhat stable and less jittery results.

```
biasOffset=retrieveOffset();
i=0;
while a= accelerometerValue() do
    vector.removeFirstElement();
    vector.push(a- biasOffset);
    finalAcc= mean(vector);
    if abs(finalAcc) >window then
        v= previousV + previousA + (a - previousA)/2 * dt ;
        p= previousP + previousA + (v - previousV)/2 * dt ;
    end
end
```

Algorithm 3: Position with Bias Removal and Window of Discrimination

The results of these algorithms are described and analyzed in section 4.2.1.

As explained in the mentioned section, this approach of using solely an IMU based approach is effectively proven to very problematic due to its accuracy and must be discarded.

3.1.3 *Monocular Visual Odometry (MVO)*

To implement this algorithm, Singh and Venkatesh (2015) implementation was used, ported to Android. It uses OpenCV and Native code (C++). Below is a description of each iteration of the algorithm:

1. Capture frame from camera;
2. Extract Features using FAST;
3. Track Features based on previous frame;
4. Calculate Essential Matrix;
5. Recover Relative Pose based on Essential Matrix;
6. Extract Translation and Rotation from Pose;
7. Apply scaling factor to translation using Smartphone's IMU;
8. Append Transformation;

Singh and Venkatesh (2015) implementation had a different application. It was tested on a KITTI dataset, which is a visual odometry benchmark of a driving car. The dataset has accelerometer values and camera frames. This scenario is optimal due to dominant forward movement, and also altitude is not a factor, so only two dimensions are involved. Whereas on a smartphone, a small tilt on the device completely changes the direction of movement in 3D space.

The matrices recovered by the algorithm are in a unit format, so no real scale can be inferred from them. This means that the algorithm always assumes that the translation that occurred between frames is unitary. So, the algorithm from 3.1.2 was used to apply a scaling factor to the matrices.

A test was made using the algorithm which is discussed in section 4.2.2. Reviewing the results, the precision acquired is not sufficient to render 3D objects with faithful accuracy over a real environment.

3.1.4 *Conclusions*

The marker-based navigation is optimal to quickly position the device in a scene, but suffers some faults by not being able to calculate the position of the device when Vuforia fails to detect a Marker. Implementations were built and tested of an Inertial Navigation System and Monocular Visual Odometry in order to assess the feasibility of these techniques to

aid the marker-based navigation. The tests proved that the techniques did not meet the precision requirements. It becomes clear that a high precision dead reckoning solution on a smartphone with mono-camera setup is simply not possible, as of yet. So, the application has to rely almost exclusively on markers to recover the device's pose.

3.2 ARCHITECTURE OVERVIEW

By relying on markers, their position becomes very crucial in reaching good tracking results. And as the best marker positions change between scenes, a marker positioning tool would be a great asset to this concept. One was developed which allows the user to position markers in virtual 3D scenes, and also has a feature which tells the user the optimal places for a marker, based on the scene's geometry and other markers.

The architecture was built with modulation and simplicity in mind. To connect the PC application with the Android app, there was a need to create a simple structure which comprised the 3D models and the transformations of the 3D scene associated with each marker, so a file format was devised which contained all the information about a specific AR scene and its marker layouts.

The architecture follows a streamlined process which can be seen in Figure 28. The computer app reads 3D models, and after a user places markers, the environment can be deployed as an ARData file, a format constructed in this thesis for the sole purpose of saving 3D and marker data. This type of file can then be read and edited by the computer app or read and viewed in Augmented Reality by the mobile app.

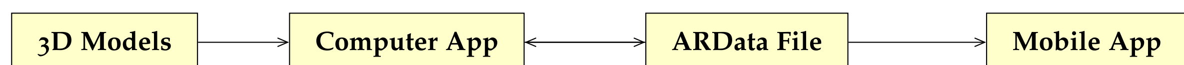


Figure 28.: Architecture Pipeline

3.2.1 *Data Exchange Format*

A file format was devised to exchange data between the AR assistant (computer app) and the AR Viewer (Android app) name as the ARData format. An ARData File is a file with .ardata format which in essence can be read as a zip file. It contains information about a specific AR scene, and contains the following data:

- **Vuforia binary** - A **.dat** file which contains the marker's features to be read by Vuforia's API.

- **Vuforia XML** - A **.xml** which contains the names (ID's) of the markers and their respective 2D virtual size. The structure of the XML is illustrated in Figure 29.

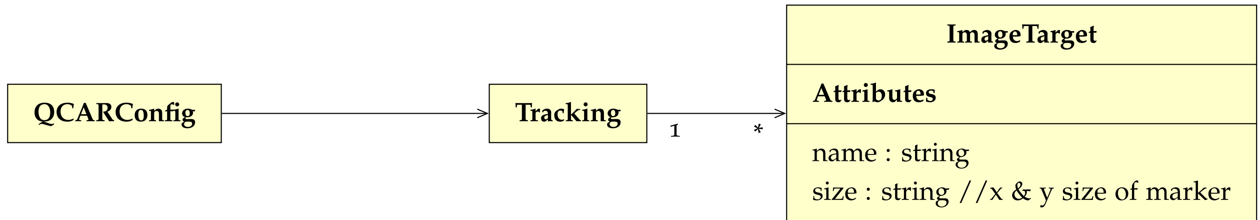


Figure 29.: Vuforia XML Structure

- **3D Models** - Two 3D models, one for the Virtual Scene and another for the Real Scene.. The Real 3D Model should replicate, as best as possible, the real environment. It serves as a helpful mean to place markers, as the user should easily be able to layout the markers if a faithful portrayal of the real environment is shown. This 3D model should be as small in size as possible, in order to minimize its space in the project file. The real 3D model is included in the file for debug purposes in the Android app and so it can be edited in the computer app. The 3D Virtual Model, on the other hand, is the virtual environment the user desires to view in Augmented Reality. Taking the example of Ponte de Lima, Figure 30 illustrates the concepts of Real 3D Model and Virtual 3D Model:

(a) Real 3D Model

(b) Virtual 3D Model

Figure 30.: Ponte de Lima 3D Models (PL3D)

- **Dataset Info** - A **.json** file which contains information about the structure of the file and the positions and rotations of the markers.

```
{
key : string,  //License Vuforia API key
modelFolder : string, //Folder contained in .ardata which contains
    the 3D Models
real : string, //filename of the real 3D model
virtual : string, //filename of the virtual 3D model
xml : string, //xml filename from Vuforia's database
markers: [{
    name:string, //name of Marker which matches ID from Vuforia's
        binaries
    rotation:[double,double,double,double],
    translation:[double,double,double],
}]
}
```

Listing 3.1: Structure of the *dataset.json* for an ARData project

3.3 AR ASSISTANT

The idea for a Computer app came from the unreliability of navigation algorithms like the INS and MVO (Subsections 3.1.2 and 3.1.3), making a focus of this thesis the markers, and their positions across the scene. The application is divided in three major components as can be seen in Figure 31.

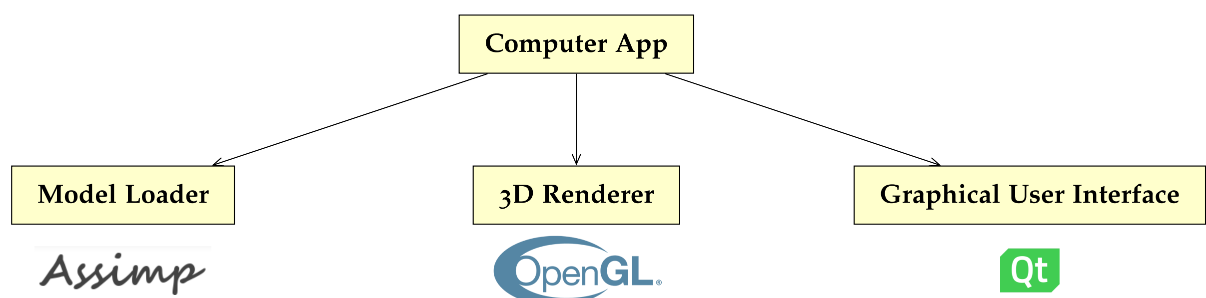


Figure 31.: Computer application Modules

As the application would work similarly to 3D Editors, the idea of using already existing 3D Editors with plugin support like Unity was taken into account, but was ultimately discarded. This was because they are ultimately too "bloated" for this task. Aside from some essential components, like the ability to create and save projects and also load 3D models, the application is composed of the following features:

1. **Creating Markers:** An intuitive and easy way to arrange markers must be created, similarly to the arrangement of 3D objects in other 3D editors or building video games. Also editing properties of rotation and translation for these entities, ID assignment and texture image display.

2. **Creating Cameras and Running simulations:** The application needs a way to test the arrangement of markers based on their visibility, size and orientation in relation to possible positions of where a person might be using the AR app. The creation of cameras in the 3D scene correspond to possible user positions, and a feature is available in which a simulation is ran, painting the 3D landscape with colors representing the quality of the current marker arrangement. This feature would also allow the user to define the camera's parameters, like vertical and horizontal angle of view, and also height.

3.3.1 *User Interface*

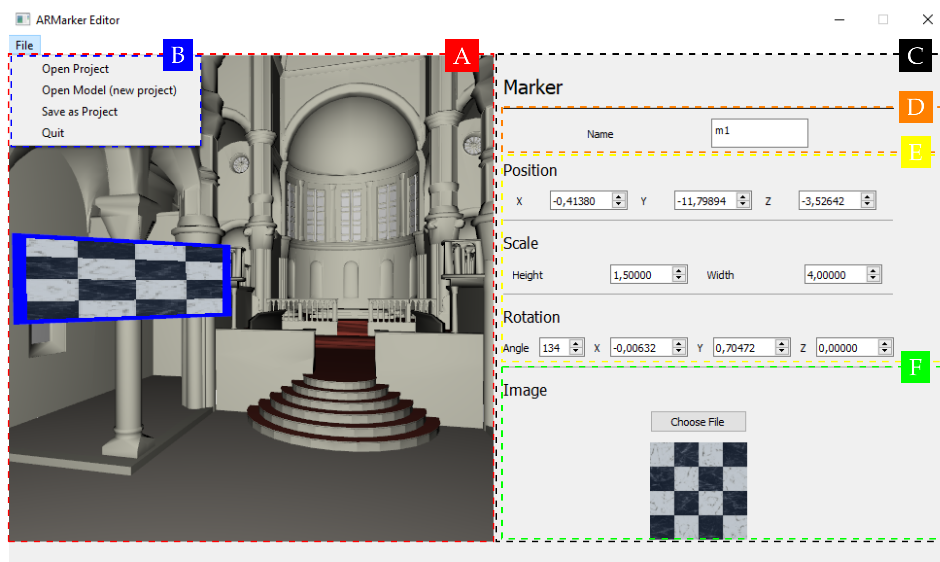


Figure 32.: Computer App UI (Marker Menu)

A - 3D EDITOR: Screen which provides a real time rendering of the 3D Models to be used, and also allows the user to easily create markers and cameras through hotkeys (Annex A.3.1). The creation of these entities is made easier by dynamically interacting with the 3D models surfaces. It also provides other features, like moving around the scene in a FPS-style camera, the removal of entities and switching between 3D models (real and virtual).

B - TOOLBAR: The toolbar provides options to open .ardata files, otherwise known as projects, also allows the creation of projects by opening two 3D models, and saving the current program state to a .ardata file.

C - OBJECT EDITOR: This screen provides a multitude of parameters which are context dependent: If a marker is selected (Marker Editor), if a camera is selected (Camera Editor), or if the user has decided to save the project (Saving Project).

- Marker Editor

  D - NAME: The marker's name is its identifier for Vuforia. Before using the AR assistant, the image markers should be added to the Vuforia Target Manager. When added, the user has to define a name. That name must be the same as the one defined here. So, it is a very crucial parameter since this name is what defines the set of features of this target.

  E - TRANSFORMATIONS: The transformations section defines the translation, rotation and size of the marker in the 3D Model.

  F - DISPLAY TEXTURE: The display texture is for improving the user experience by having assigned a custom image to the marker on the 3D Editor region. This image does not define the set of features of the image target, that task is for the Name parameter.

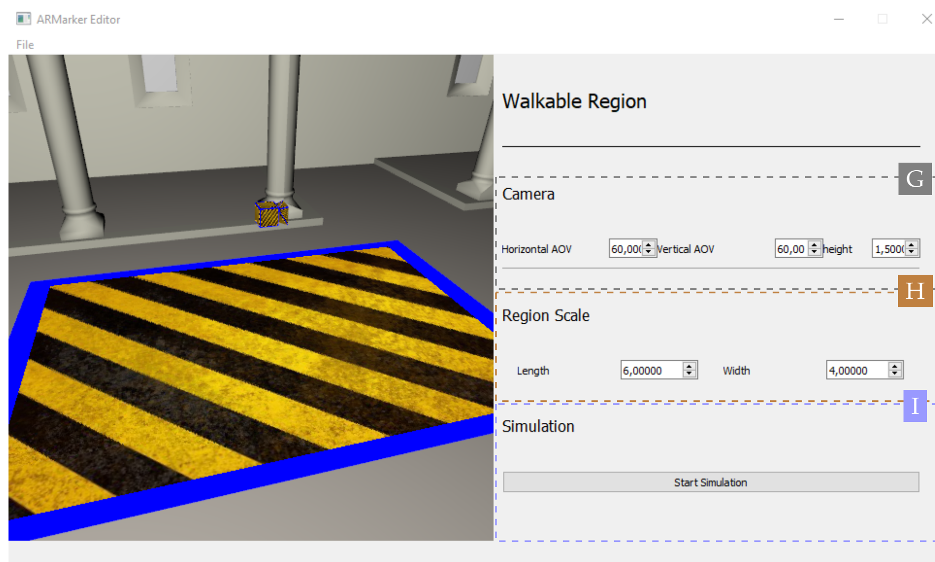


Figure 33.: Computer App UI (Camera Menu)

- Camera Editor

  G - CAMERA PARAMETERS: In this section, the vertical and horizontal angle of view, and the height of a virtual camera can be defined. The AOV parameters should match

the smartphone camera that will be used. The height should be the average height of the camera when using the Android AR Viewer.

H - SIZE: This section defines the region that the virtual camera resides in. This should match the area in which the user will use the android app, in the real environment.

I - SIMULATION: The simulation provides a colorful visualization painted over the 3D model of the overall quality of marker placement, and also good areas to place markers.

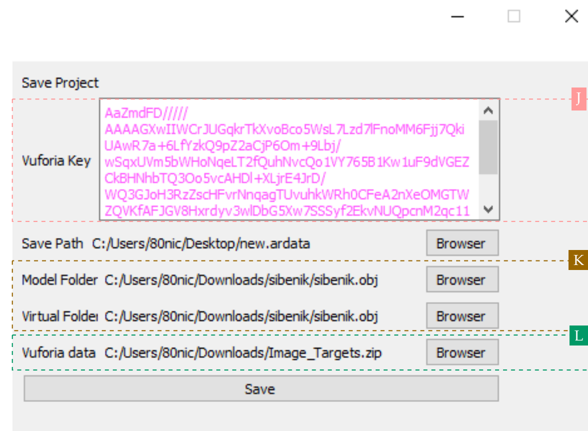


Figure 34.: Computer App UI (Save Project Menu)

- Saving Project

J - VUFORIA KEY: This license key is provided by Vuforia on its Vuforia Developer Portal, available online. A user creating a new project should register and request a key.

K - 3D MODELS: Both the virtual and real 3D models can be assigned here. The textures of these models must be contained in this folder or in a child folder.

L - VUFORIA BINARIES: In Vuforia's Developer portal, after adding the markers, a user can acquire binaries by requesting to download the database, specifically the one for Android Studio. The .zip file contains the names and the respective set of features for each marker contained in the database. The binaries are associated with a project here.

3.3.2 *Marker and Camera Creation*

As mentioned before, Marker and Camera Creation in the application had to be intuitive, which meant that when creating these entities, the user could position and align them

along the 3D Model surface. For example, if a user wanted to create a marker, he could move his mouse along the surface of the 3D Model, and the marker would position itself automatically and align itself in a parallel manner with the Model's surface in the location of the mouse pointer. A group of surface aligned markers can be seen in Figure 35.

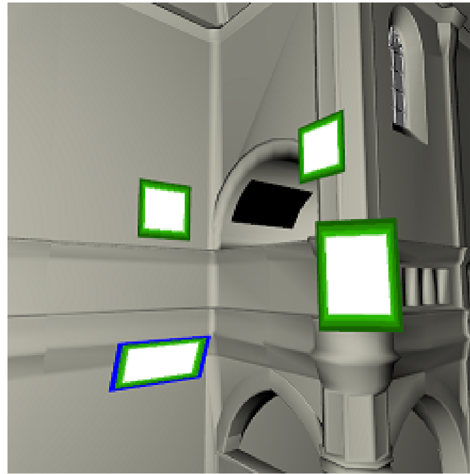
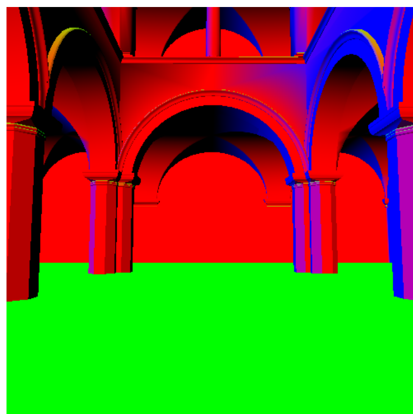
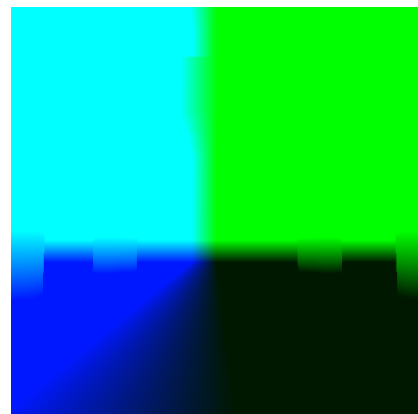


Figure 35.: Marker Surface Alignment

To accomplish this goal, certain components of the surface of the 3D Model, specifically of the pixel where the mouse pointer is located on, must be extracted. To achieve this, two render passes are made to Frame Buffer Objects, and later assigned to two textures, which encode the position and normal information of the surface (Figure 36). When the program is in a context of creating a marker, it can retrieve position and normal information from these textures in the specific pixel the mouse pointer is pointing to. An example of the aforementioned textures can be seen in figure 36. The information about the normals and position is encoded in the color channels, so specific pixels can be queried when desired.



(a) Normals



(b) Position

Figure 36.: Render Passes for Marker Alignment

To align the marker in a parallel manner to the surface, a conversion of a pixel encoded normal vector must be converted to an axis-angle representation. A vector which works as an axis of rotation for the marker rotation can be simply a conversion of the normal vector, which is perpendicular to the surface, to a vector which is tangent to the surface. To do this, the cross product is used with a vector which rotates the normal vector by 90 degrees.

$$cross = \begin{pmatrix} 0 \\ 0 \\ -1 \end{pmatrix} \times Normal \tag{3.3.2.1}$$

The angle is calculated using the arc cosine of the dot product:

$$angle = cos^{-1}\left(\begin{pmatrix} 0 \\ 0 \\ -1 \end{pmatrix} \cdot Normal \right) \tag{3.3.2.2}$$

3.3.3 *Object Picking*

The application also requires an entity selecting feature, meaning that you can select markers and cameras in order to change their properties. This can be done using an Object Picking algorithm. To achieve this, a render pass is done to a FBO which encodes the ID of an entity (camera or marker) in the color channels (Red for marker and Blue for Marker) as seen in Figure 37. This ID is simply the index of the rendered entity in the structure it is saved in. When a double click is recorded on the screen, the texture is queried to check if an entity was selected. If it was, its info is retrieved and shown to the user.

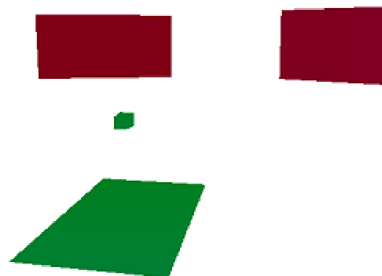


Figure 37.: Object Picking - Different Entity Types are saved in different Color Channels (Red for markers and Green for cameras)

3.3.4 *Camera Simulation*

When developing the idea of the implementation of the Android application, it became apparent that the placement of markers would greatly affect the final outcome of the seamless tracking between markers and overall experience of the AR experience. So, a simulation was created in the computer application which visually shows the user an appreciation of the placement of markers.

This simulation is based on choosing a location in the real model where a user should be using the AR app in the real environment. The algorithm then processes the surrounding area, surfaces and markers, and colors the 3D Model with representative colors of qualitative appreciation.

The process followed is similar to creating shadowmaps in computer graphics. The basic shadowmap algorithm consists in two passes. Firstly, the 3D world is rendered from the perspective of the light, saving to a texture the distance to a surface for every pixel. Then, the world is rendered normally, and, for every surface, we test the distance to the light. If at a location, the distance to the light is greater than what is saved in the shadowmap texture, at the same point, the area is in shadow, because another surface is blocking the direct pathway between that point and the light. An illustration of this idea can be seen in Figure 38.

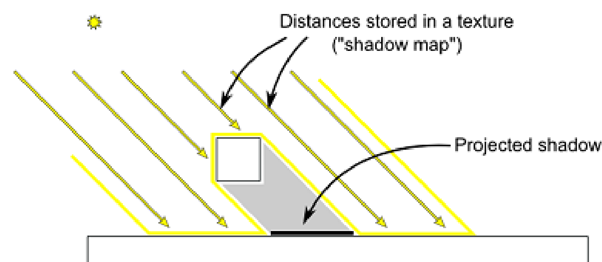


Figure 38.: ShadowMap (OpenGL-Tutorial)

In this context, the light is replaced by the location where the user placed a camera, and a few steps are adapted to fit this feature. The rendering from the placed camera's perspective must be of the complete area around it. This means that we are rendering a cubemap, which requires rendering six times, two times for each axis, and with an horizontal and vertical field of view of exactly 90 degrees, so the edges of each face of the cube matches their neighbours. Figure 39 illustrates the concept of cubemap rendering.

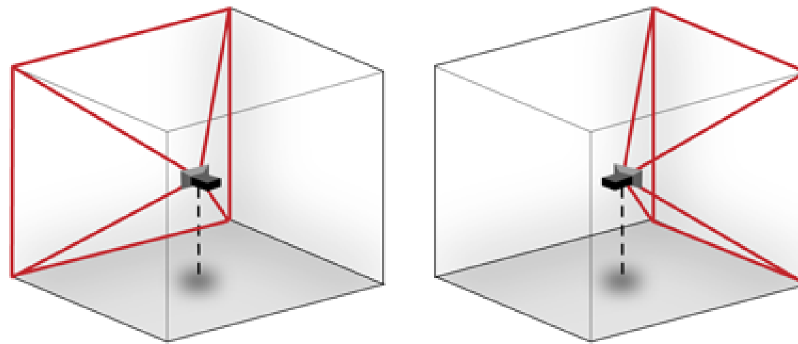


Figure 39.: Cubemap Rendering (Acko)

Instead of doing the traditional one pass from the light's (in this context, the placed camera) perspective, in this case there are two. Firstly, the world and markers are rendered but only the markers are encoded with color, so we know their location. The 3D World is rendered so obstructed markers are overwriten, because since they're not seen by the placed camera, they are not deemed important. The distance to each marker is also color encoded.

In the second pass, the main algorithm is processed in which surface normals of the 3D model and the placement of markers are evaluated. After evaluation, each pixel is assigned a color and saved in the RGB channel, while the depth which is later used to calculate the obstruction of object is saved in the alpha channel. There are four colors which represent the following:

- **Green, Orange and Red**: Represents the quality of the surface normals in relation to the placed camera's position, from good to bad. This means that a surface that is being viewed from the side by the proposed camera, is considered a bad surface (red color) and a surface that is viewed from the front is considered a good surface (green color). These colors also signify that there are no markers nearby.

- **Blue**: Signifies that the current surface has a marker nearby, meaning that another marker is not required at that location.

The result looks like Figure 40.

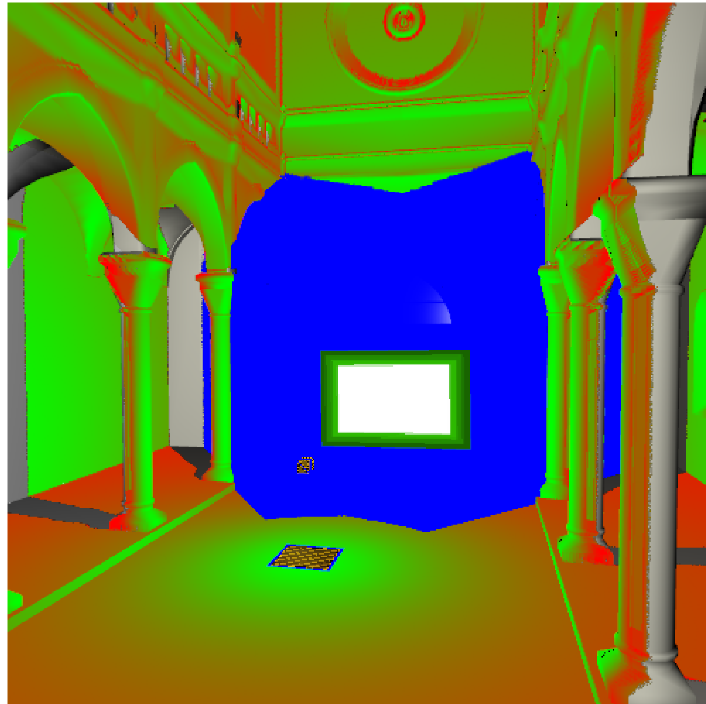


Figure 40.: Camera Simulation Example

The calculation of the quality of normals (for the green,orange or red color assignment) is done by first determining the world position and normals for each vertex:

```
1 interpolatedNormal =  vertexNormal ;
2 interpolatedPosition = vec3( M * vec4( vertexPosition , 1.0 ) );
```

Listing 3.2: Calculating World Vertex Position and Normals

Then, for each fragment, the dot product is calculated between the direction of the camera ray and the normal vector of the surface. The dot product can be used to infer the parallelism of two vectors. The more similar these vectors are, the better the normals are considered.

```
1 vec3 RayVector = normalize( interpolatedPosition-CameraPos);
2 float perpendicular =dot(-RayVector, normalVector );
```

Listing 3.3: Calculating Ray and Normal Vectors Parallellism

The color of the surface uses the dot product as a weight for the green and red channels, making the surface which normals are considered good, green, and the ones which are considered bad, red.

```
1 normalColor+= vec3(1-dot,dot,0);
```

Listing 3.4: Surface Color Based on Normals

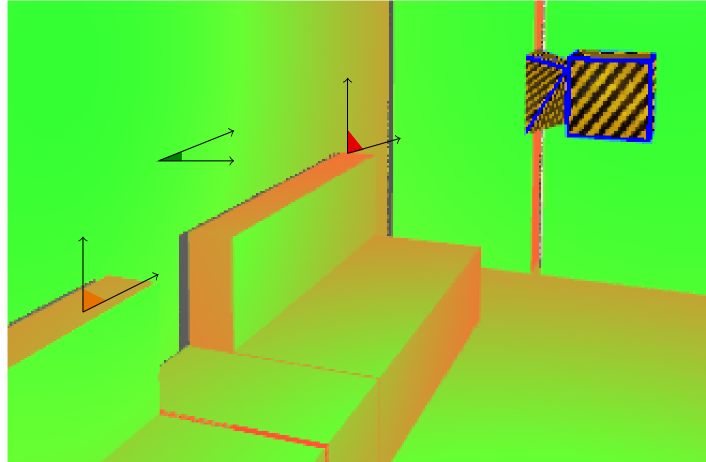


Figure 41.: Surface normals and the surface-to-camera vector are compared and the pixel is assigned a color based on their similarity.

For the blue color, the distance between a marker and the current surface is calculated. The distance threshold is related to the horizontal and vertical angle of vision of the android camera when using the app. So, the user may define these parameters to match the specific smartphone that is intended to be used.

In each fragment, the marker-only texture of the previous pass is probed to infer if there is a marker nearby to the current pixel, using the defined horizontal and vertical angle of vision as delimiters. This is done by taking the ray pixel vector and rotating it horizontally and vertically along the area contained in the specified angle of view.

It is coded in two *for* loops where the outside loop increments the horizontal rotation and the nested loop increments vertical rotation. The angle variables are denoted as **xAngle** and **yAngle**.

To rotate the camera horizontally, the Up vector of the ray vector needs to be calculated, which it can be by first calculating the right vector using the cross product of the ray vector with the default OpenGL up vector:

$$right = v \times \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} \tag{3.3.4.1}$$

$$up = right \times v \tag{3.3.4.2}$$

Then, it is possible to rotate the vector horizontally using the calculated up vector as the axis of rotation. A vector tangent to the surface and the XZ plane is also calculated as it will be the axis for vertical rotation.

```
vec3 xRot =rotate(lightSourceVector,cameraUp,xAngle);
vec3 tangent = vec3(xRot.z,0.0,-xRot.x);
```

Listing 3.5: Calculating Horizontal Rotation

The vertical rotation can be calculated and the cubemap can be sampled to check if a marker is present:

```
            vec3 yRot =rotate(xRot,tangent,yAngle);
            vec4 mdColor= texture(markerCubemap,yRot);
            if(mdColor.b!=0.0 && mdColor.r==0.0){
                color= vec3(0,0,1);
                break;
            }
```

Listing 3.6: Vertical Rotation and Cubemap Sampling

After the color assignment steps previously described, the passes in the user position are complete, and the normal rendering of the 3D model and markers can be done. Six textures are passed to the rendering pipeline with the qualitative colors and depth. These six textures represent each face of the cube of the previous pass. Six corresponding Model View Projection matrices are also passed. Six textures are passed instead of a cubemap due to Qt limitations of creating cubemaps from floating point textures. The interpolated position is calculated using each MVP matrix and multiplying with the vertex position. In each fragment, we calculate the projection coordinates in order to pick the correct texture. Te correct texture is defined by being the cube face in which the surface point is contained.

```
projCoords = currentShadow.xyz/ currentShadow.w;
projCoords = projCoords * 0.5 + 0.5;
```

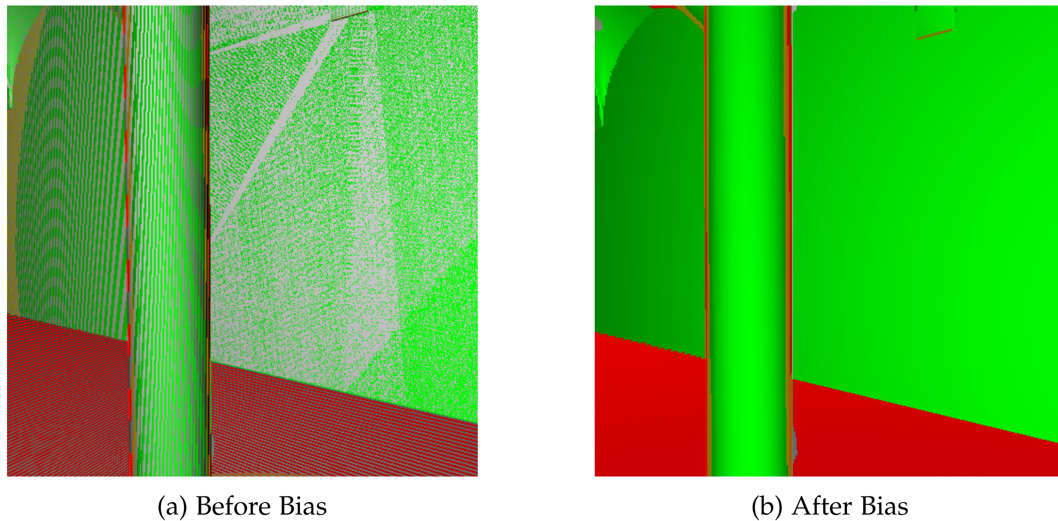
Listing 3.7: Projection Coordinates Calculation

Then, the texture is sampled.

```
if ( texture2D( simTexture, projCoords.xy).a >= projCoords.z - bias){
                color =vec3(texture2D( simTexture, projCoords.xy).rgb;
            }
```

Listing 3.8: Sampling Simulation Texture

The bias variable is used to prevent the commonly known effect of shadow acne. This effect is represented in Figure 42 and is due to the texture not being able to represent with complete precision the distance from the user position to the surface.



(a) Before Bias (b) After Bias

Figure 42.: Shadow Acne (before and after bias)

Another effect in a shadowmap context, is peter panning, which happens when the shading looks displaced relative to the objects. This can simply be avoided by not using thin geometry.

Still, aliasing is very noticeable as can be seen in Figure 43a. The solution is to sample the texture not once, but multiple times at different positions for the same pixel, thus creating an average which smoothes the edges of the surface. It is desirable that the indexes are random but close to original pixel. One way to calculate indexes is to use Poisson-disk sampling which uses a constant array of positions that are added to the original pixel:

```
vec2 poissonDisk[4] = vec2[](
  vec2( -0.94201624, -0.39906216 ),
  vec2( 0.94558609, -0.76890725 ),
  vec2( -0.094184101, -0.92938870 ),
  vec2( 0.34495938, 0.29387760 )
);
for (int i=0;i<4;i++){
  if ( texture( shadowMap, ShadowCoord.xy + poissonDisk[i]/700.0 ).z  <
      ShadowCoord.z-bias ){
    ...
  }
}
poisson
```

Listing 3.9: Sampling Simulation Texture using Poisson Disk

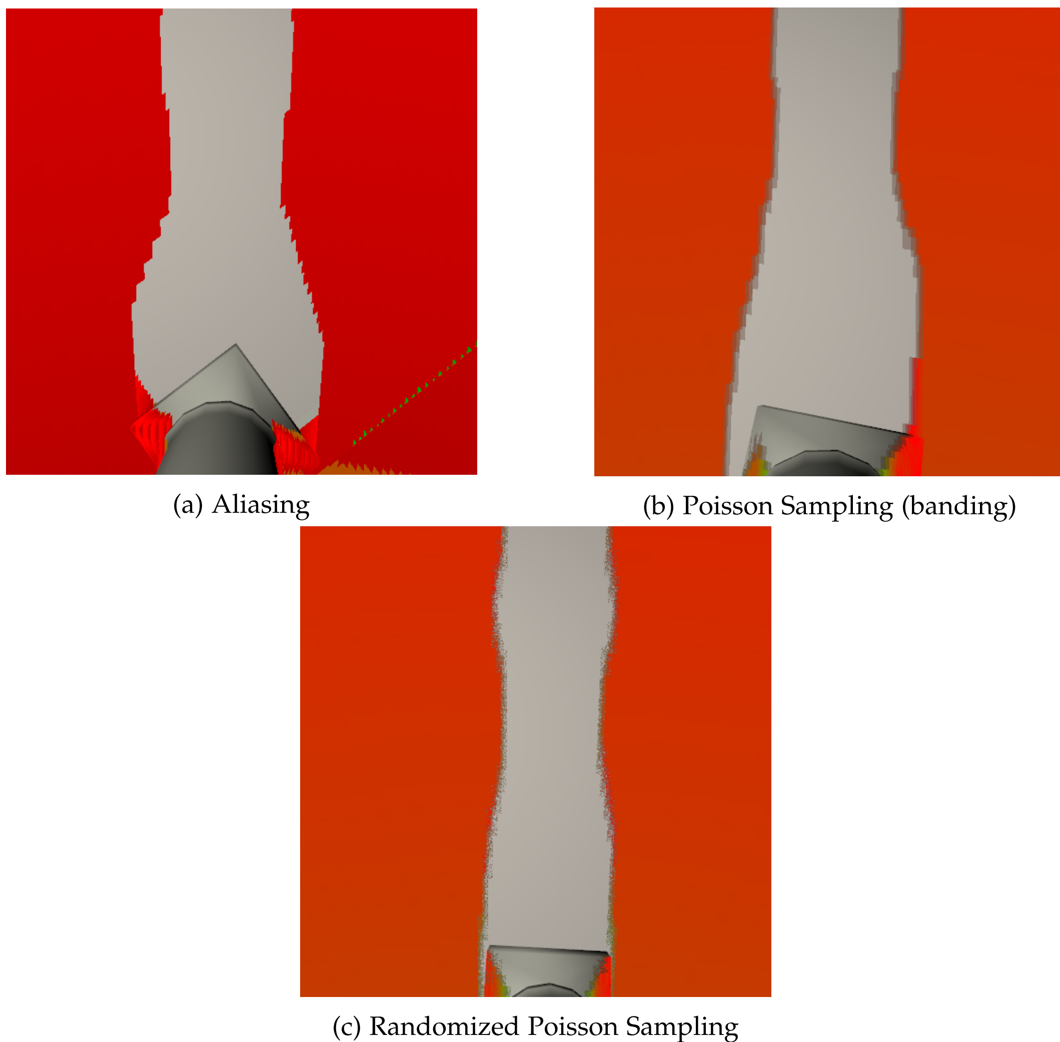
But this can lead to banding as seen in Figure 43b. To prevent this, the poisson disk can be filled with more positions, 16 in this case, and the index calculated when sampling is a

random number between 0 and 15. This way, four random positions from sixteen total are sampled, instead of four fixed positions.

```
for (int i=0;i<4;i++){
        int index =int(16.0*random(floor(interpolated_position*1000.0), i
            ))%16;
        if ( texture( shadowMap, ShadowCoord.xy + poissonDisk[index
            ]/700.0 ).z  <  ShadowCoord.z-bias ){
    ...
    }
}
```

Listing 3.10: Sampling Simulation Texture using Poisson Disk and Random Index

The final sampling result can be seen in Figure 43c.



(a) Aliasing

(b) Poisson Sampling (banding)

(c) Randomized Poisson Sampling

Figure 43.: Dealing with Aliasing

To summarize, the algorithm is divided in three major steps:

1. **Marker Depth:** A cubemap is built which encodes the marker positions around the scene from the perspective of a placed camera by the user;
2. **Color Assignment:** From the same position as before, another cubemap is built which encodes a certain color, based on if a Marker is near the pixel or based on the similarity between the surface normal vector and the vector from the surface to the placed camera;
3. **Surface Painting:** When the application is in the regular render pass, meaning the pass which will render to the computer screen, the surfaces which are potentially visible by the user placed camera will be painted with the colors assigned previously;

3.4 AR VIEWER

Some thought was put into choosing the building blocks of the software application, as Vuforia offers the SDK for Unity, Java and C++ development. C++ was the chosen route, as it offers an API which allows for more freedom and also because the other API's used (like Assimp) were written for C++. The Android application is composed of four major components that can be seen in Figure 44.

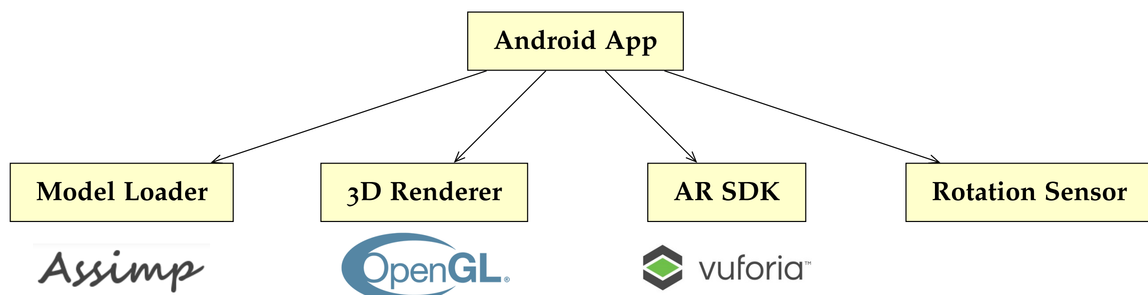
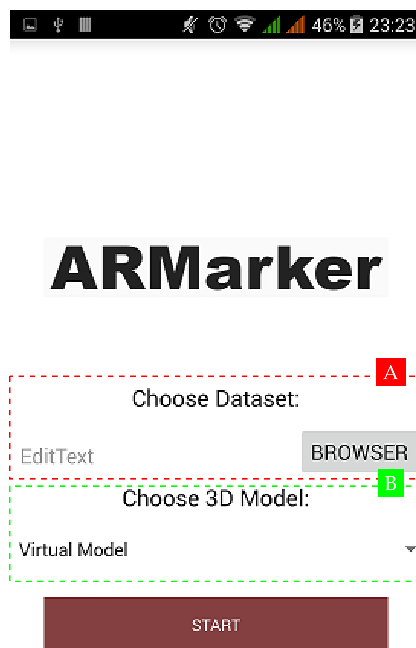


Figure 44.: Android application Modules

Aside these four modules, there are other less important components, like the one which extracts and reads the information of an ARData project or the performance and trajectory tracker, which tracks the FPS of the application, and saves the poses (positions and rotations) of the device to a .txt file for debug purposes, respectively.

3.4.1 *User Interface*

The user interface is as minimalist as possible, with a menu to select the AR scene, and after one has been loaded, only the rendered model and camera feed are shown. The UI is shown in Figure 45.



(a) Android app Menu

(b) Android Model Render

Figure 45.: Android App UI

A - PROJECT BROWSER: Browser for the smartphone's filesystem to select ARData projects.

B - 3D MODEL CHOOSER: In this picker, the user can pick between starting the AR experience with the virtual model, or the real model. The real model can be used mostly for debug purposes, to check if the markers are properly placed by confirming the alignment of the virtual and real environment.

C - CAMERA FEED: While running the app, the camera feed is rendered in the background.

D - RENDERED MODEL: A 3D model overlay is rendered over the camera feed, merging the two components.

3.4.2 *Application Workflow*

The Android OS limits some features to Java Code, so, the initial Menu, Vuforia's initialization routine and the Rotation sensor component are written in Java, while the main chunk of the program (the rendering/marker tracking loop) is written in C++. The application follows the following workflow:

1. **Load ARData Project:** In order to display an AR scene, a file browser is displayed so the user can choose an ARData File from the android's filesystem. The user may pick between rendering the Real 3D Model or the Virtual 3D Model. Then, all the data specifying 3D Models and marker data is extracted from that file.
2. **Wait for first marker detection:** While the system has yet to detect the first marker, only the camera feed is shown, and no 3D model is rendered, as there is no info on the device's position.
3. **Detection of Marker:** After the detection of the first marker, the camera's position and rotation can be inferred.
   a) **Extraction of pose:** By extracting the camera's pose, the correct transformation matrix that matches the virtual and real position of the camera is calculated.
   b) **Render 3D Model:** The 3D model is then rendered over the camera's frame with the correct transformation, merging virtual and 3D environments.
4. **Process next camera frame:** In subsequent frames, there are two approaches that can be taken, whether a marker was detected or not.
   - **Marker Detected:** If a marker has been detected, repeat from step 3.
   - **No Marker Detected:** Go to step 5.
5. **Merge Sensor Rotation with a Recorded Pose:** Using the Android's Sensors, the device's rotation is combined with a recorded transformation matrix of when a marker

was still detected. This way, the 3D model is able to render with some persistence even when no marker is detected. Although this keeps the 3D Model in the right position and rotation when the smartphone is rotated, it does not account for smartphone movement.

6. **Repeat from step 4.**

3.4.3 *Marker Placement*

Matrices are very important in an OpenGL Context, as they are used to transform objects in 3D space, and are a crucial part in the OpenGL rendering pipeline.

Transformation matrices are 4x4 matrices which are capable of representing translation, scale and rotation.

The rendering pipeline usually follows the process seen on Figure 46.

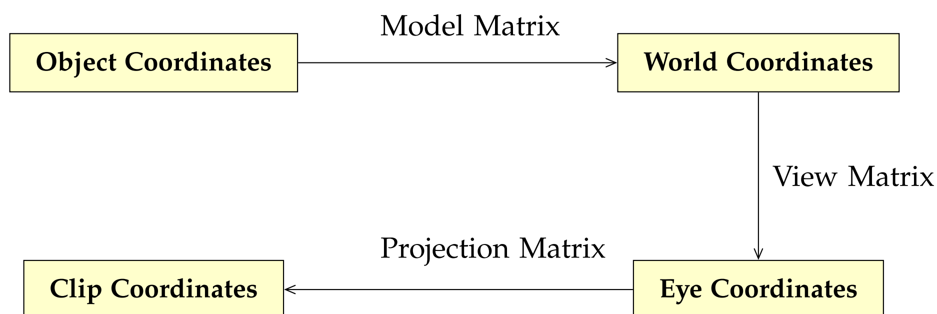


Figure 46.: Rendering Pipeline

- **Model Matrix:** It is used to transform a 3D model. This converts the coordinates from Object to World coordinates;
- **View Matrix:** In OpenGL, the camera does not move, it is the world that moves around it. So, a matrix is needed to move the 3d Objects from World to Camera coordinates, thus creating the View Matrix;
- **Projection Matrix:** The projection coordinates converts the position where an object should be in relation to the camera, to where an object should be in relation to the screen;

After the detection of a marker, it is possible to calculate the Model View matrix (the multiplication of the Model Matrix with the View Matrix) with Vuforia's API. This Model View matrix assumes that the marker is positioned on the origin in World coordinates,

pointing towards the negative Z axis. A Vuforia Marker is also associated with an ID, meaning that when a set of features is identified by Vuforia, the developer can discover the specific marker detected. In this context, that marker is associated with a translation and rotation in the AR scene, so, we can merge those transformations with the extracted model view matrix.

$$MV_{final} = MV_{Vuforia} \times R_{marker} \times T_{marker} \quad (3.4.3.1)$$

If no marker is found, then a previous recorded pose transformation is merged with a rotation matrix from the Android's Sensor rotation.

$$MV_{final} = MV_{Vuforia} \times R_{marker} \times R_{sensor} \times T_{marker} \quad (3.4.3.2)$$

In the current state of the implementation, only one marker is tracked at a time. This means that Vuforia gives priority to an already detected Marker, or if no marker is detected, Vuforia's SDK chooses one. A multi-concurrent target tracking system would be an improvement over the current state, although Vuforia's performance would decrease while in a multi-target tracking context (Vuforia (b)).

3.4.4 *Sensor Rotation Estimation*

When no marker is detected, sensor rotation is merged with a previous recorded pose. These rotation values are retrieved from the rotation vector sensor available by the Android's OS. It is a result of sensor fusion using the accelerometer, the magnetometer and the gyroscope. In section 4.3, a comparison is made between the precision of the gyroscope and the rotation vector. From this simple test, it can be inferred that the rotation vector sensor is more precise and stable, and is the best choice to recover rotation information using the smartphone's sensors.

3.4.5 *Jitter Smoothing*

One recurring problem when testing the implementation was the jitter of the 3D Model. This happens when consecutive frames have a disparity in the extracted Model View Matrix, due to small pixel differences between frames, or when markers with low feature density are used. This is usually not noticeable when using Vuforia's demo applications because small 3D Models are used, so the jitter is insignificant. But in this application there is a need to use large 3D models, so the jitter of the rendered objects can be very significant.

In this case, jitter occurs differently in two separate situations:

1. **Looking directly at marker:** When looking directly at a marker, with its borders all contained in the camera frame, a small jitter occurs due to pixel inconsistencies between frames.

2. **Looking partially at marker:** When only part of a marker is available, for example, when in the process of turning away from a marker, the jitter becomes very significant due to the fact that only a smaller set of features are detected.

The approach used to deal with jitter was a combination of different methodologies: a low pass filter applied to the values of the given Model View matrix and an algorithm which decides if a Vuforia's calculated Camera Pose should be discarded by comparing the rotation given by its transformation matrix with the rotation given by the device's sensor.

To use a low pass filter on rotation and translation parameters, we must first convert the Model View Matrix from camera coordinates to world coordinates, by inverting it. This is possible because the Model matrix in Vuforia is an identity matrix. We can then extract the rotation component, by dividing it in three vectors, the Right, Up, and Direction vectors. The MV^{-1} also provides the Translation vector. With these four vectors, the low pass filter can be applied to each vector, and then used to build a new Model View Matrix.

$$MV^{-1} = \begin{pmatrix} m_0 & m_4 & m_8 & m_{12} \\ m_1 & m_5 & m_9 & m_{13} \\ m_2 & m_6 & m_{10} & m_{14} \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{3.4.5.1}$$

The Low Pass Filter used is displayed in equation:

$$V_{smooth} = V_n \times \alpha + (V_{smooth-1} \times (1 - \alpha)); \tag{3.4.5.2}$$

Although the α value can be changed depending on the amount of jitter, a value of 0.3 was found to be acceptable in most cases.

After smoothing all the vectors, the original matrix is then inverted again, which results in a Model View Matrix with low passed translation and rotation components.

Although this algorithm resolves the jitter present when looking at a marker directly, it does not hold when looking partially at a marker. In this situation, vuforia produces heavily distorted Model View matrices, in which the amount of error can be huge, that even the low passed Model View Matrices have large discrepancies. To tackle this problem we compare the rotation given by vuforia with the rotation given by the Android's rotation vector sensor. Firstly there is an extraction of two rotation matrices, one being vuforia's rotation between frames, and another from the rotation given by the Android rotation sensor between frames. Then we invert one of the resulting matrices and multiply them.

The result is a matrix which measures the rotation difference between Vuforia's tracking and the Android's rotation sensor.

$$DiffRotation_{Vuforia} = LastRotation^{-1}_{Vuforia} CurrRotation_{Vuforia} \quad (3.4.5.3)$$

$$DiffRotation_{Sensor} = LastRotation^{-1}_{Sensor} CurrRotation_{Sensor} \quad (3.4.5.4)$$

$$DiffRotation = DiffRotation^{-1}_{Vuforia} DiffRotation_{Sensor} \quad (3.4.5.5)$$

A calculation of the difference between vuforia's and the sensors can now be made. This error is proportional to the amount of rotation in this resulting matrix. By converting this matrix to an axis angle representation (represented by an unit vector **e** as the direction of axis of rotation and an angle θ as the magnitude), the θ angle can be used as a measure of the amount of error present. As the calculation of **e** is redundant as we do not need to know the axis of rotation, the extraction of θ in radians from the rotation matrix can be made by:

$$\theta = \arccos\left(\frac{Tr(R) - 1}{2}\right) \quad (3.4.5.6)$$

A simple threshold of 15 degrees was used to determine if the discrepancy between vuforia's rotation and the sensors rotation is to great. This could be interpreted as a large threshold but this approach is used to deal with large errors. So a 15 degree threshold is acceptable because the low pass filter somewhat deals with lower amounts of error.

If a Vuforia matrix is proved to have to an unreliable rotation, it is discarded, and instead, vuforia's pose of the previous frame is used, combined with the rotation recorded by the Android sensors since the last frame. This way, although the current Model View Matrix was discarded, the previous one is used with the rotation vector sensor, thus not discarding real rotation that could happen.

4

RESULTS AND TEST SCENES

4.1 DEVICE

The device used in this dissertation was a BQ Aquaris E5 FHD Android smartphone which has the specifications in Table 1:

Specification	Description
Operating System	KitKat (4.4.2)
Screen	1920x1080 (5′)
CPU	Octa-Core ARM Cortex-A7 @ 2.00 GHz
GPU	Mali-450MP @ 700MHz
Camera (Back)	13 MP with AutoFocus/Flash
Sensors (Precision Results in Anex A.1)	Accelerometer Gyroscope Magnetometer Proximity & Light Sensor

Table 1.: BQ Aquaris E5 FHD Specs

Android also provides other sensors at a software level, like the gravity sensor, the linear acceleration sensors and the rotation vector sensor.

4.1.1 *Computing Power*

To compare the smartphone's performance with other modern Android smartphones, **AnTuTu** benchmark was used. The result ranked the device as a middle/low end device, with a score of 25142. A similar smartphone is the Samsung Galaxy J5 which has a score of 21597. High end devices are much more powerful. For example, the OnePlus 5, which is ranked number one by **AnTuTu** has a score of 181042 (August 2017).

4.2 Navigation System on Android Results

In this section the results of algorithms described in the previous chapter, namely the INS and MVO algorithms, are tested and analyzed.

As these results are based on the effective precision of the smartphones sensors, some important sensor information from the device used is presented in Annex A.1, collected from a 30 second resting position.

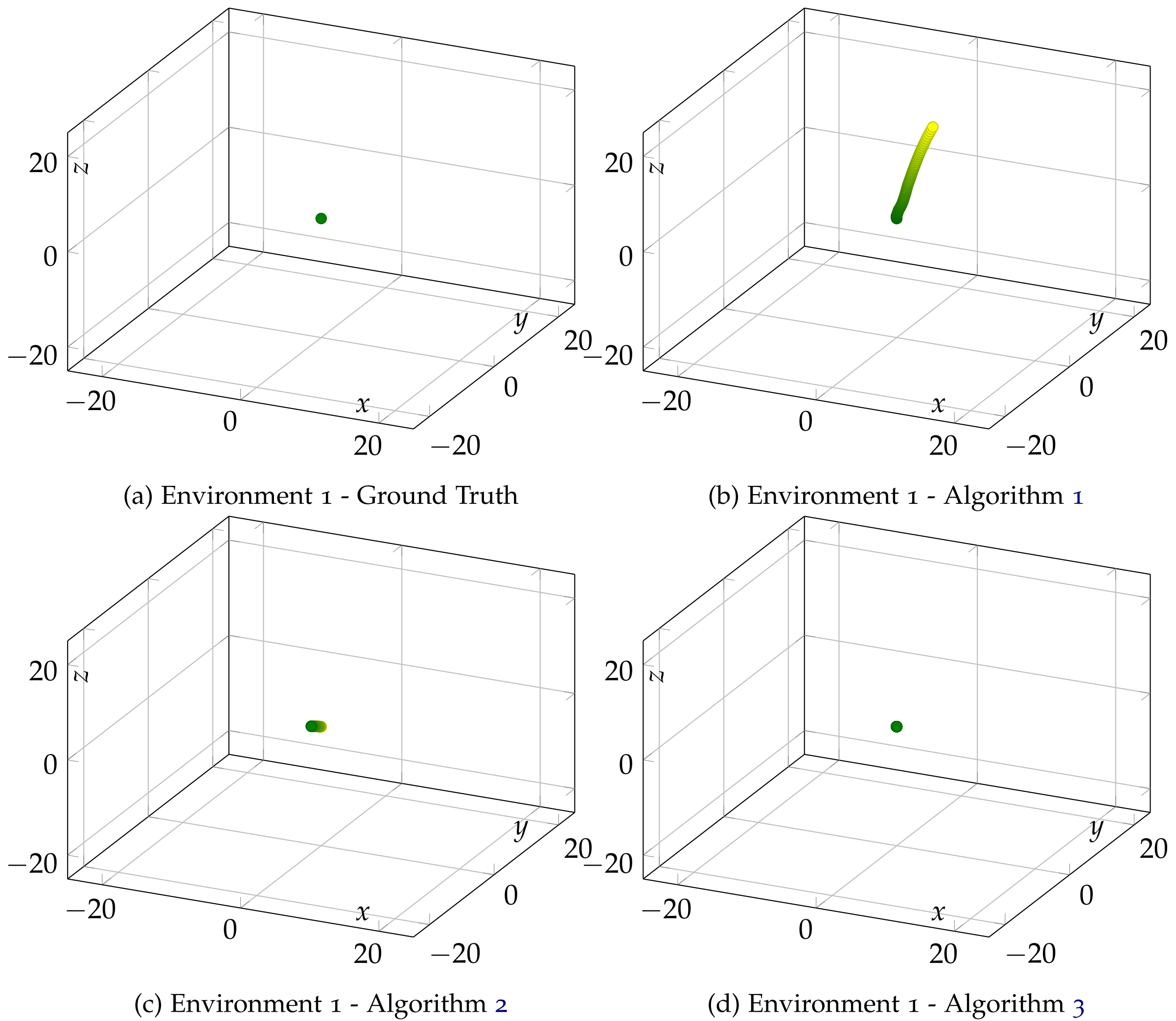
4.2.1 *INS Test*

In this section, the algorithm presented in 3.1.2 is tested in different environments, assessing its feasibility by analyzing its accuracy and precision.

Two different testing environments were devised to test the positioning algorithm:

1. **Environment 1**: Device lies resting on top of a table for 30 seconds;
2. **Environment 2**: Device is pushed back and forth while lying down over a meter line for 60 seconds in a straight line;

The results of position estimation for Environment 1 using Algorithms 1 (simple positioning algorithm), 2 (positioning algorithm with bias removal) and 3 (positioning algorithm with bias removal and window of discrimination) (algorithms described in section 3.1.2), and also the Ground Truth, which is the true trajectory of the device, are displayed in Figure 47.



(a) Environment 1 - Ground Truth

(b) Environment 1 - Algorithm 1

(c) Environment 1 - Algorithm 2

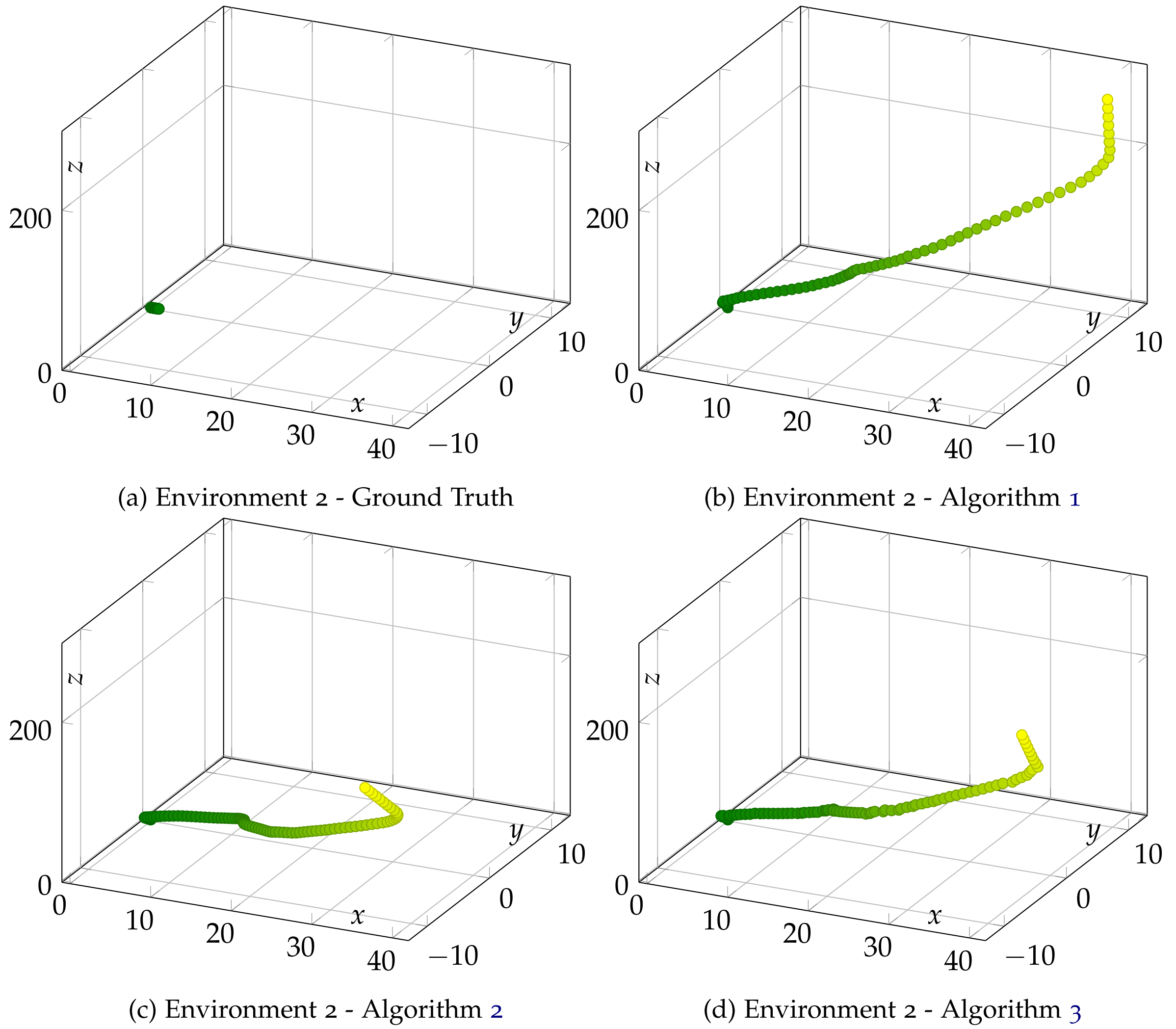
(d) Environment 1 - Algorithm 3

Figure 47.: Position Estimation using IMU (Environment 1) in Meters

Using Algorithm 1 the position drifted 20 meters in 30 seconds on the z axis, in Algorithm 2 there was a drift of about 30 centimeters also in the z axis, and in Algorithm 3 there was only a drift of 8 centimeters in the x axis. The difference in the algorithms is noticeable, but the changes made between algorithms may remove important properties of the signal. For example, a high window of discrimination may remove real recorded movement.

The properties of these changes are sensor dependent, as each sensor has its own bias and has its own noise magnitude, so, each algorithm needs to be tune for a specific Android accelerometer. Also, as we're applying an average in N iterations, that may increase the integration error. This is because, the lower the sample rate (interval time between samples), the better the result of integration will be. Still, the average method serves to eliminate the noise present in the data, and is advantageous to use in this context.

Figure 48 presents the position estimation results for Environment 2.



(a) Environment 2 - Ground Truth

(b) Environment 2 - Algorithm 1

(c) Environment 2 - Algorithm 2

(d) Environment 2 - Algorithm 3

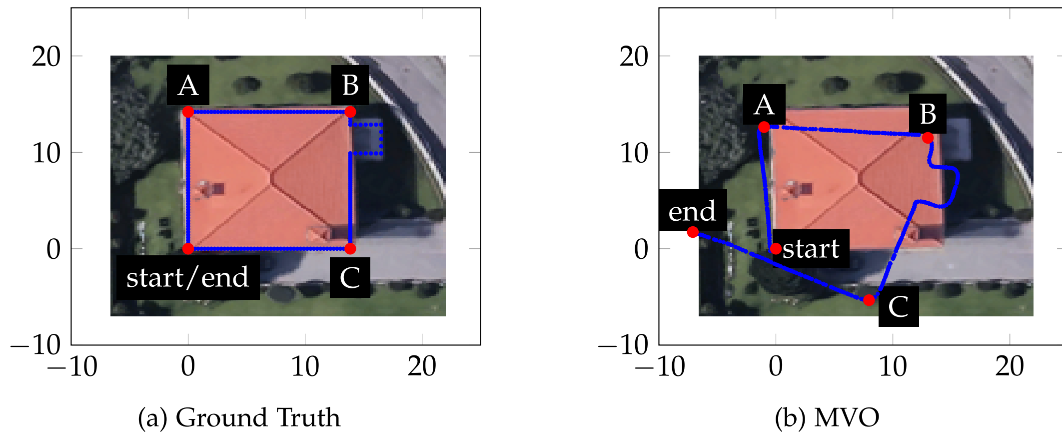
Figure 48.: Position Estimation using IMU (Environment 2) in Meters

The motion which is supposed to be back and forth on the x axis, is seen as an enormous drift mainly on the z axis. Although a zig-zag motion can be seen, the error is too large to recognize similarities between the results and the ground truth.

This drift is due to the fact that the position and velocity calculation is based on the previous positions and velocities. So, the error accumulates, and as a double integration is present, it accumulates at a very large rate. Also, when the Android's firmware removes the gravity component from the accelerometer, small errors are sometimes made, which prove catastrophic in this context.

4.2.2 *MVO Test*

To test the MVO implementation (described in 3.1.3), a trajectory was devised around a house, to compare the ground truth with the MVO algorithm. The user would travel the trajectory holding the smartphone perpendicularly to the floor, pointing in the moving direction, as steadily as possible. Figure 49 illustrates the devised trajectory (Ground Truth) and the results obtained.



(a) Ground Truth (b) MVO

Figure 49.: Monocular Visual Odometry Trajectory Test

In Table 2 the displacement is calculated between the Ground Truth and the MVO test at several flagged positions from Figure 49.

Position	Ground Truth	MVO	Distance Error
A	0,14.20	-1.01,12.62	1.87
B	13.86,14.20	12.99,-11.5	2.83
C	13.86,0	7.96,-5.34	7.95
End	0,0	-7.1,1.728	7.31

Table 2.: Monocular Visual Odometry Test Errors

The error builds quickly and reaches a large magnitude. In the context of this dissertation, this amount of error is not acceptable, as the 3D objects rendered would have a noticeable displacement.

4.3 SENSOR ROTATION TEST

The following section analyzes the comparison between the precision obtained with the gyroscope versus the rotation vector sensor (the implementation is described in section

3.4.4). In Figure 50, the sine of the angle recorded from the gyroscope and the rotation vector is illustrated.The device was rotated 360 degrees in two separate 180 degree increments.

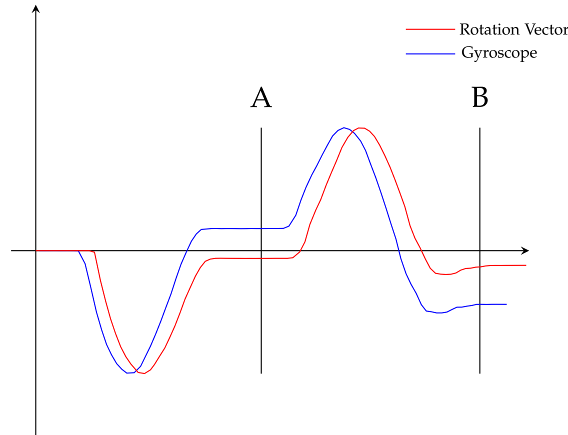


Figure 50.: Rotation Estimation (Gyroscope and Rotation Vector)

In Table 3 the angle error is measured for both sensors at the end of each 180 degree turn, named position A and B, where the sine of the angle should be zero.

Position	Gyroscope Error	Rotation Vector Error
A	10.34°	3.56°
B	25.86°	6.84°

Table 3.: Angle Errors at A & B positions from Figure 50 for the Gyroscope and the Rotation Vector

This test hints that the rotation vector provides better precision than the gyroscope. This, although, does not mean that its results are perfect, as some mild error is noticeable. One large improvement with the rotation vector sensor is that drift over time is dealt with significantly because the magnetometer and the accelerometer do not drift, while a gyroscope-only implementation will always drift over time, as there is no data validation from other sensors.

4.4 MARKER DETECTION AND TRACKING RESULTS

One important aspect to analyze was the robustness and efficiency of the implementation in detecting and tracking markers. The tests in this section are done using the final implementation, meaning that the marker tracking is slightly different than Vuforia's default tracking, due to the Jitter Smoothing algorithm discussed in section 3.4.5. Different factors were important to evaluate:

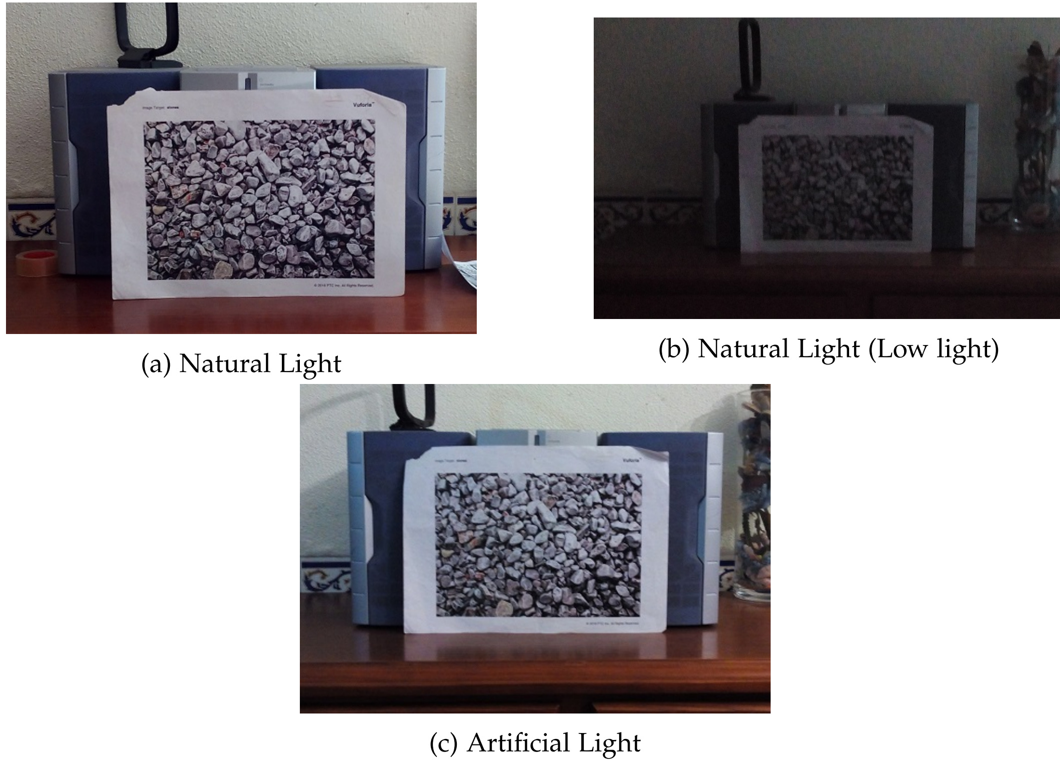
1. The effective range (distance) in Marker detection and tracking;

2. The effect lighting has in Marker detection and tracking;
3. How accurate the tracking of a Marker is;
4. The effects of tracking a partially occluded Marker;
5. The difference in tracking markers of different quality;

The setup for tests was made using Vuforia's own "Stones" example marker, which is a feature-rich 2D image seen in Figure 51. The marker was printed on A4 paper with dimensions 23cm x 16.1cm.

Figure 51.: Test Marker (taken from Vuforia (a))

To test different lighting conditions, three different environments were created which are illustrated in Figure 52. The low light environment is the lowest light environment where the tracking and detection is still feasible. A Marker placed in a less luminous setting is very unpredictable (sometimes it is detected, other times it is not) and most of the times impossible to detect and track. The artificial light environment is supposed to investigate the difference in detection and tracking when markers are illuminated in different ways. The artificial light used emanates white light, while natural light has a more orange tone affecting the diffuse color emanated by materials.



(a) Natural Light

(b) Natural Light (Low light)

(c) Artificial Light

Figure 52.: Different Light Environments

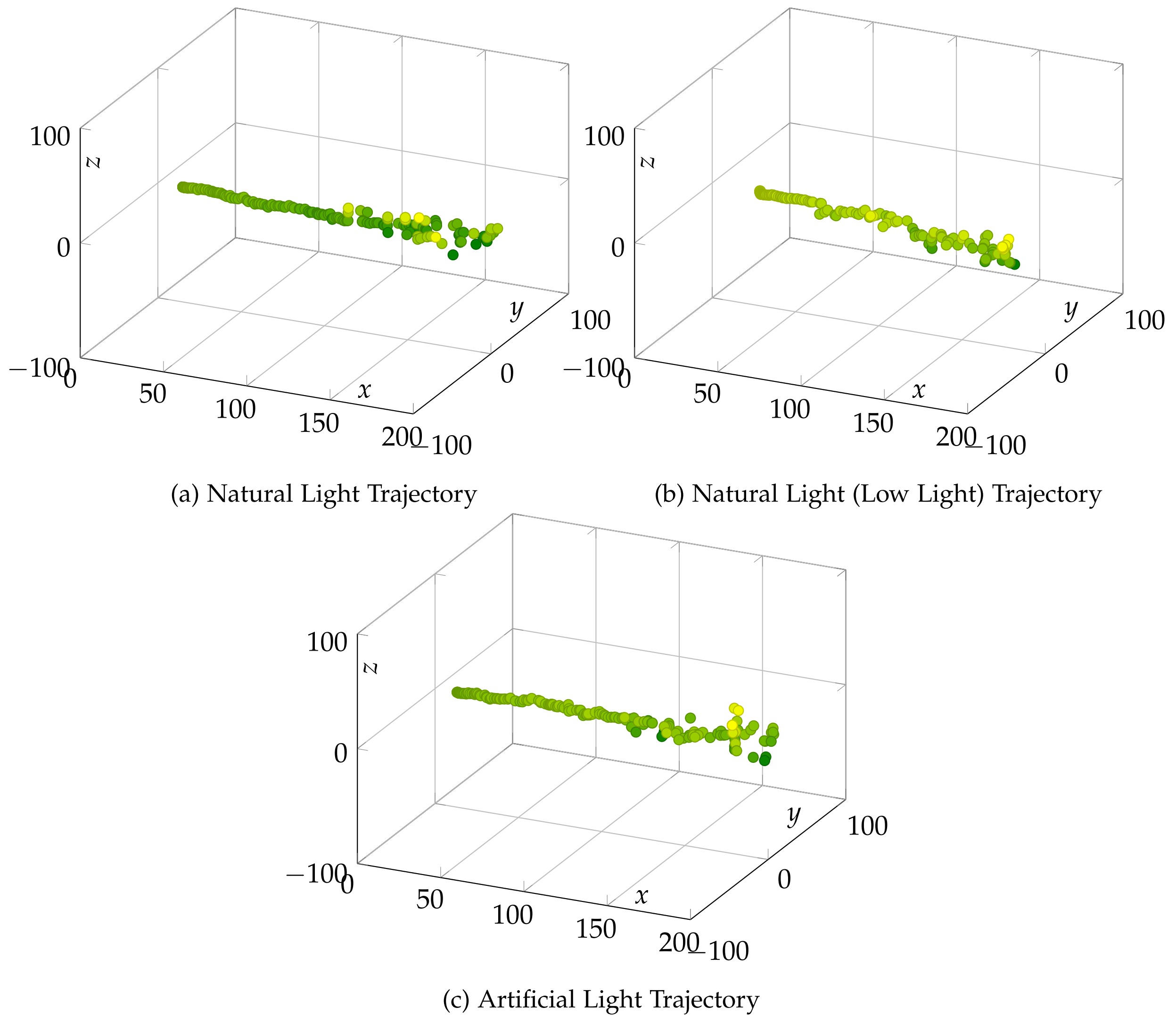
In Table 4, results are presented for the maximum distance that the test marker was able to be detected, and the maximum distance that the marker was able to be tracked by the app.

Illumination	Maximum Detection Distance	Maximum Tracking Distance
Natural Light	130 cm	265 cm
Natural Light (Low Light)	110 cm	220 cm
Artificial Light	130 cm	265 cm

Table 4.: Maximum Detection/Tracking Distance using Vuforia in different lighting conditions

From the data in Table 4 we can conclude that lighting does somewhat affect detection and tracking. As expected, it works better for lighter environments. However, there seems to be no notable difference between using artificial and natural light, which means that the markers can have some changes in color tones without affecting detection and tracking distances. This is mostly because the detection algorithm works on grayscale images, as can be seen in the Vuforia Developer Portal, when viewing the extracted features from each Marker.

In Figure 53 trajectory tests are illustrated along a 2 meter line (results are presented in centimeters). The marker was positioned as seen in Figure 52 while the tester walked backwards pointing the device's camera to the marker, holding the smartphone steadily.



(a) Natural Light Trajectory

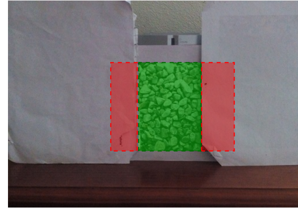
(b) Natural Light (Low Light) Trajectory

(c) Artificial Light Trajectory

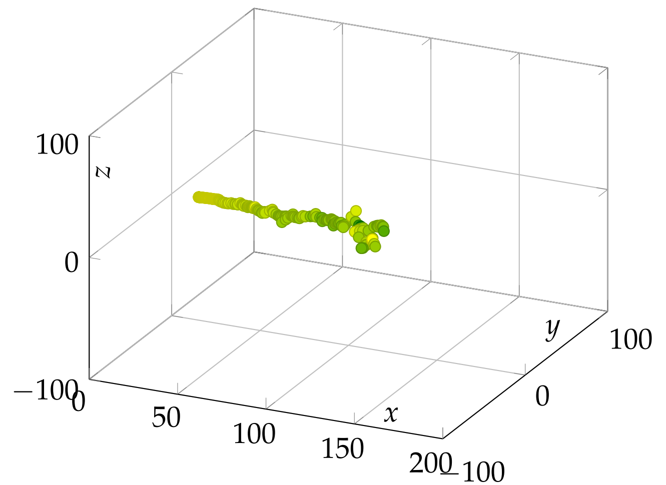
Figure 53.: Different Light Environment Trajectories

The test demonstrates that the distance to the Marker greatly influences the tracking accuracy, as the smartphone moves away from the Marker, the calculation of the pose starts to be more inconstant. Tracking seems to be equivalent across all types of illumination, although the Low Light trajectory seems to start to jitter at a noticeable lower distance from the marker than the other two environments.

The effectiveness of detection and tracking on a partial occluded marker was tested using the same trajectory of the previous tests, shown in Figure 54b. About 50% of the surface of the Marker was covered (Figure 54a).



(a) Marker



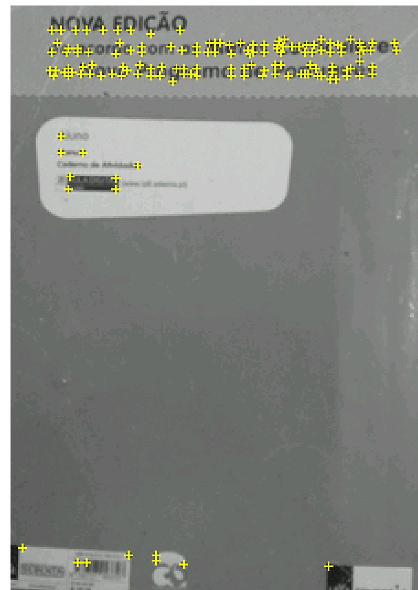
(b) Partially Occluded Marker Trajectory

Figure 54.: Partial Occluded Marker

The tracking was lost at 116 cm meaning that, in this case, that was the maximum tracking distance. Additionally, the maximum detection distance was 90 cm. Comparing it to the other measured cases, a partially occluded marker provides worse detection and tracking results as expected.

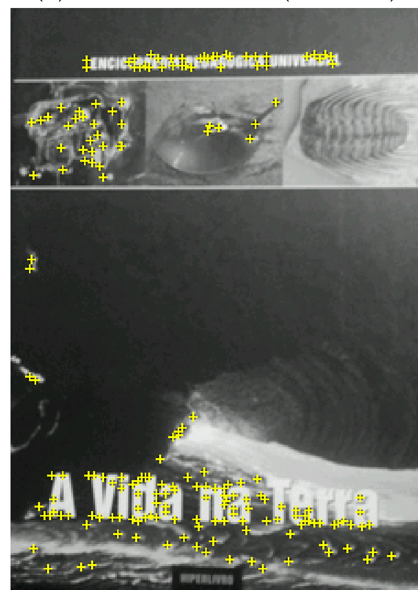
The feature properties which makes a good marker, are, as said previously, being rich in detail, contrast, and having a balanced allocation of features across its 2D image plane. Vuforia provides a rating tool (5 star rating) for its added markers which takes into account these previous properties. A marker with low rating will effectively be less detectable and trackable than a marker with high rating. Two markers with different ratings were compared in their tracking along the same trajectory from the previous tests (200 cm straight line). The markers used were book covers and their features are represented in Figure 55.

(a) Two Star Marker



(b) Two Star Marker (Features)

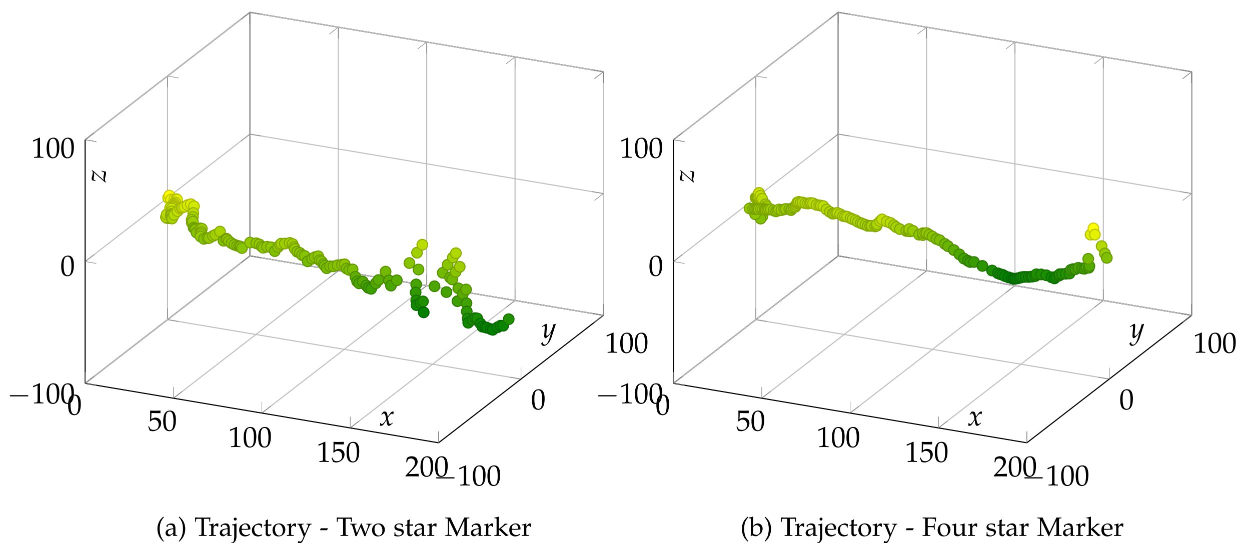
(c) Four Star Marker



(d) Four Star Marker (Features)

Figure 55.: Markers with different Ratings

The difference between the two starred marker and the four starred marker is demonstrated in Figure 56.



(a) Trajectory - Two star Marker

(b) Trajectory - Four star Marker

Figure 56.: Room Trajectory

The results somewhat vary between both markers. The four marker trajectory has a smoother trajectory while the two star marker starts to jitter heavily at the 150 cm mark. This is an indicator that the rating assigned by Vuforia to the marker is relevant to the tracking results.

4.5 AR PROJECTS

In this section two example AR Projects are demonstrated, one in an indoor small area and another one in a large open space. The creation of markers is a vital part of creating these AR Scenes (a guide to the creation of markers in the Vuforia Developer Portal is presented in Annex A.2) and thus, prior to the discussion of these two examples, an examination of natural markers, their properties and certain consequences will be presented.

4.5.1 *Natural Markers for AR Scenes*

There are some important aspects when picking a natural Marker for an AR project for this dissertation:

- **Placement:** The placement of markers is very important. They should not be cluttered in the same space and, on the flip side, should also not be spread to thinly around the scene. They also should obviously be visible and pointed towards possible user positions. As the user may have difficulty in figuring out if a certain collection of markers are placed optimally, the camera simulation feature of the computer app helps in giving the user feedback about Marker placement.

- **Size:** A Marker size is very relevant and is proportional to the distance to possible user positions, meaning that a marker that is farther away from the area that the user will be walking on, should be bigger to allow for satisfactory detection and tracking.

- **Feature Properties:** There are intrinsic properties of markers that can either make them viable, or completely useless.

  The results from Figure 56 comparing the tracking from two different markers proves that choosing the right markers heavily influences the outcome of the AR experience of a project.

  Still, there are other natural Marker properties that the Vuforia rating tool can't measure which greatly influences its quality. These circumstances that degrade the quality of a natural marker are multiple. They can be external environmental factors, the materials the markers are made of, or the way that the marker is seen when viewed from different angles.

  Seen in Figure 57 is an example of building a marker from a 3D surface. After viewing the marker from a different viewing angle, as the surface is not completely flat, the appearance of the features changes completely, becoming unrecognizable from the original point of view. The solution obviously relies in removing this unwanted property, which means to use mostly flat objects whenever possible. Although this is easily achievable in controlled space and for specific objects, such as the one used in Figure 57, it can be harder for open environments or different objects. For those situations, the use of non-flat markers should be avoided or used with precaution.

(a) Marker

(b) Deformed Natural Marker (Front)

(c) Deformed Natural Marker (Side)

(d) Corrected Natural Marker (Front)

(e) Corrected Natural Marker (Side)

Figure 57.: Natural Markers (3D Surface)

In Figure 58, another unwanted component is illustrated. Example features **1**,**2** and **3** demonstrate the distorted result that reflective materials impinge on markers. As the reflection is not a definite part of a material and changes with perspective changes, it damages the outcome of detection and tracking.

Mutability of natural markers can also become an issue when collecting markers in uncontrolled spaces. Mutability means that the marker changes over time. For example, if a marker is extracted from a construction area (like a temporary road sign), it is expected that the marker will disappear in a certain span of time. Another example is to extract a marker from the facade of a building, which may change over time (depending on the type of building). One day some windows might be open or may have the curtains closed while in other days the reverse may happen.

One final aspect to take into account are shadows, more specifically the penumbra, meaning the line which separates the shadow from the light, which can cross through sections of a natural marker at different times of the day in a natural light context. This completely affects the way the marker looks, not only because features are extracted from that penumbra line, but also because the camera, as it focuses on high luminance areas, blackens the shadowed sections of the markers.

To summarize, there are multiple properties which negatively affect the quality of a natural marker which an user should be aware of when collecting markers to create an AR project with the implementation discussed in this dissertation. Some were discussed here, like the volume of a surface, reflections, shadows and mutability.

(a) Reflections on Natural Marker (Front)

(b) Reflections on Natural Marker (Front) - Features

(c) Reflections on Natural Marker (Side)

(d) Reflections on Natural Marker (Side) - Features

(e) Reflections on Natural Marker (Curtains Closed)

(f) Reflections on Natural Marker (Curtains Closed) - Features

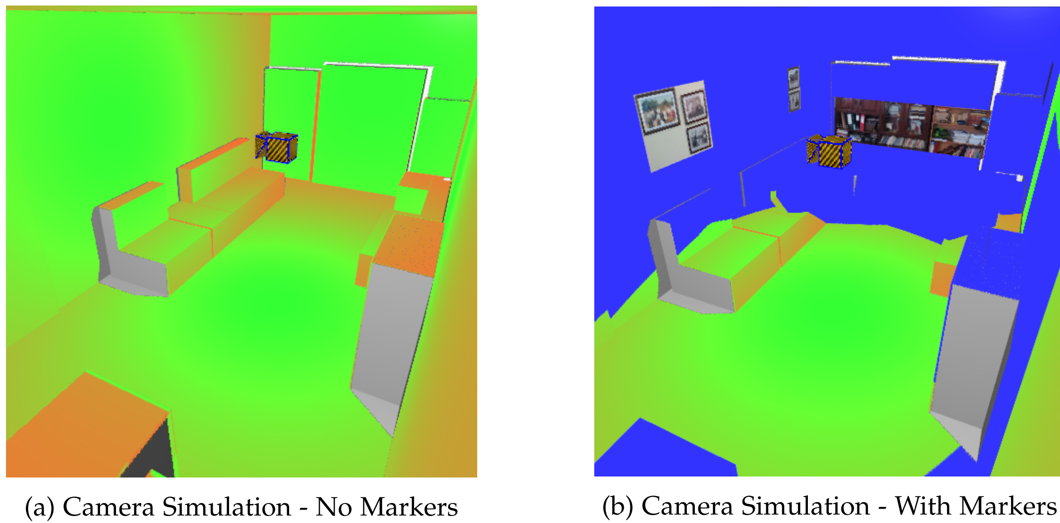
Figure 58.: Natural Markers (Reflective Material)

4.5.2 *Room*

The Room Project is an AR scene of a house division. The real environment and the markers used can be seen in Figure 59. It was made with the objective of analyzing the implementation in a more controlled, small space, and also to test the tracking of the implementation while the user moves around and uses different markers as anchor points.

Figure 59.: Room Markers

The collection of the markers took into account the aspects discussed in subsection 4.5.1, and in Figure 60 the result of the camera simulation running on the project is demonstrated. The result is optimal as most of the walls are covered in blue, meaning that the surface does not require additional markers.



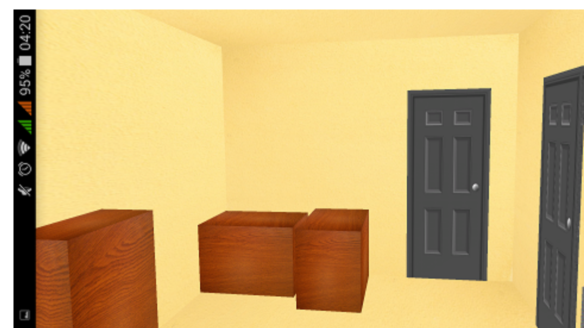
(a) Camera Simulation - No Markers (b) Camera Simulation - With Markers

Figure 60.: Camera Simulation on Room Project

In Figure 61 three screenshots of the same perspective using the app are demonstrated. These screenshots are of the camera and the 3D Models of the AR project.

(a) Camera

(b) Virtual Model



(c) Real Model

Figure 61.: Room Project Models (Same Perspective)

To measure the feasibility of the implementation in an AR scene with multiple markers, a test was made which consisted of walking around the scene, locking on to different markers, while following a defined trajectory of a 150x360 cm rectangle. The tester pointed

the camera at different markers along the trajectory. When switching between markers, the tester did not move. The result of the trajectory recorded from the android app can be seen (in centimeters) in Figure 62.

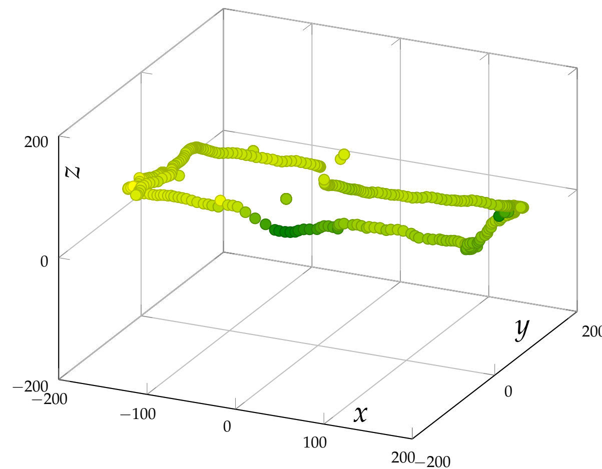


Figure 62.: Room Trajectory

Although some incongruousness is noticed, the trajectory does not strafe far from the devised path. This is an indication that the implementation does calculate the position of the smartphone with somewhat acceptable accuracy, and that the implementation can successfully work in a scene with multiple markers.

One aspect that is crucial for a good outcome in AR experience, is user knowledge. The user should know where to point the camera, and has to be aware that it could take a couple of seconds to detect a marker. Also, he has to be aware of some limitations, for example avoiding locking on to a marker which is small and far away from the device.These situations may stifle the interest of more casual users. But, as we'll see in the next section, the architecture devised in this dissertation allows for some versatility, thus being able to create AR scenes to be used in different ways.

4.5.3 *Ponte de Lima*

The initial idea of Ponte de Lima's project was to use a specific area, defined by the blue rectangle in Figure 64, as the AR environmnet. The user could walk along a catwalk (orange area from Figure 64) while pointing the camera at the building facades, where most of the markers would be. But this idea was later scrapped because the facades were deemed too unreliable for tracking, due to aspects referred to in 4.5.1. The reflections of the windows, problems with some 3D parts of the markers, and shadows which limited the use of the app to a time frame of the day. But the bigger problem was the mutability of the markers, from day-to-day small changes (curtains being closed, opened windows), to bigger changes

between weeks. These changes are demonstrated in Figure 63. In October the view to the facade is obstructed by a bandstand, and also the curtains are mostly closed. In July and September the curtains are open, although a banner can be seen in September which did not exist in July. There are also some visible problems with the marker, mostly in the unevenness of the surface. The small balconies and the lamp present are very unreliable in perspective changes. These continuous changes were also present in other facades which were collected to be markers.

(a) October 2017

(b) July 2017

(c) September 2017

Figure 63.: Same Facade Marker at different points in time

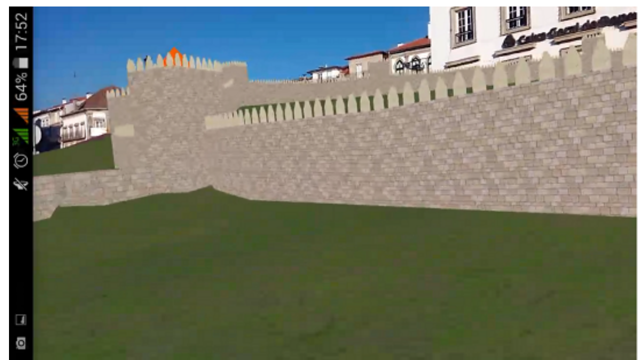
Due to the referred problems, the project pivoted to a different approach. This approach relied on creating more spaced markers, distributed more scarcely and widely across the village. This changes the AR environment from a small space with many markers, to a big area with less markers. The idea is for the user to travel to different places around the village to view the virtual environment. The layout of the markers across the village is illustrated in Figure 64.

Figure 64.: Ponte de Lima Project (Satellite images taken from Google)

This placement of markers creates several interest points around the village to use the AR project.

Figure 65 demonstrates the use of the Ponte de Lima project. The XIV wall from PL3D project (PL3D) can be seen rendered over the real environment.

(a) **A** - Camera Frame



(b) **A** - Virtual Model

(c) **B** - Camera Frame

(d) **B** - Virtual Model

(e) **C** - Camera Frame

(f) **C** - Virtual Model

Figure 65.: Ponte de Lima - Virtual Model at positions **A**,**B** and **C** from Figure 64

One thing that was noticed while using the project, is that as the environment and 3D model are very big, small errors can create noticeable errors in big distances. In Figure 66 a small rotation error is illustrated, which creates a noticeable difference between the real

environment and the 3D Model. This error is very difficult to overcome due to its small magnitude, which origins from Vuforia's pose retrieval. Vuforia's (and other AR SDK's) tracking algorithm is developed in a way to balance fast performance with robust tracking. Its use cases are also for smalller 3D models, where this kind of error is not noticeable. To fix this, the user can re-anchor to a marker until Vuforia provides a perfect pose.

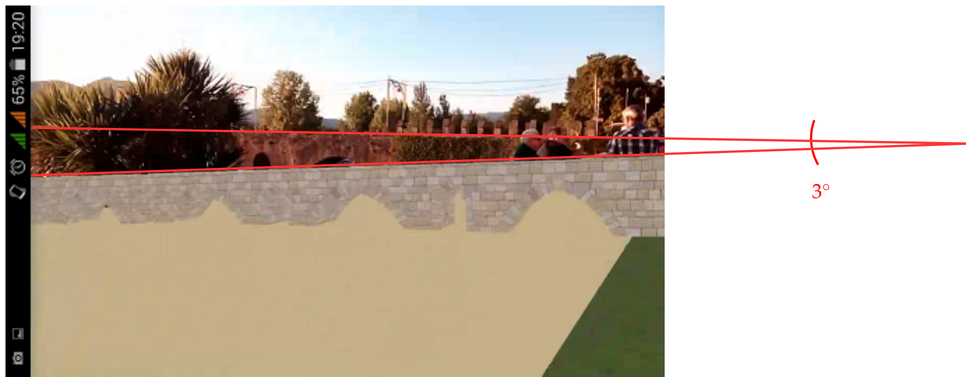


Figure 66.: Ponte de Lima - Rotation Error

4.6 PERFORMANCE ANALYSIS

The performance of the android app is mostly related to the amount of geometry of the 3D models being used, and the limit of frames per second of the camera feed of the smartphone. In Table 5 a simple performance analysis of the previously mentioned AR projects is demonstrated, using the device previously mentioned, which had a limit of 30 FPS in the camera feed. The loading time refers to the time which the application takes to extract and load the information (3D models, feature data, marker transformations) about a project.

ARDATA Project	Size	Loading Time	3D Model	Vertex Count	FPS
Room	16 MB	15 seconds	Virtual	2,622	Max (30)
			Real	136	Max (30)
Ponte de Lima	40 MB	50-70 seconds	Virtual	117,000	8
			Real	8,955	29

Table 5.: Android App Performace

5 CONCLUSIONS AND FUTURE WORK

Augmented Reality is an up and coming technology which aims to merge virtual and real content. Its use on a mobile device is a perfect combination due to the presence of smartphones in everyday life, making the AR experience seem like an extension of the daily routine.

The purpose of this work was to build an AR Android application which would be able to position the user precisely in a large 3D scene. Extensive research and tests were done on positioning techniques like SLAM, INS, and MVO, but ultimately they were proved too imprecise for this context.

The solution was built around the positioning of markers around the scene which the app was able to recognize using Vuforia's SDK through the mobile's camera. The default tracking technique was altered to provide a more robust result and also merged with Android's sensors in specific conditions.

A modular framework was built with easiness of use in mind, with an assistant app capable of creating AR scenes by allowing the user to place virtual markers and simulate the quality of the current marker disposition. The assistant also allowed projects to be deployed and consequently read by the mobile app.

The results obtained were satisfactory but not perfect. This was due to the smartphone's hardware limitations in certain scenarios. The use of the app in scenarios where the user walks around and looks multiple ways can result in some problems, but this can be avoided by shifting the project to a more static perspective. From a product scope, the implementation has potential for tourism endeavours in this way, by spreading the markers more scarcely around the scene. For instance, a city environment, enabling the user to view virtual scenes in certain checkpoints around the city. These virtual scenes could be of old monuments, structures, or even text descriptions of tourist attractions like bars or museums.

Although the implementation has its faults, mainly in the positioning of the android phone, it is still a reasonable approach taking into account the device's limitations. In the coming years, these limitations will decrease, with new smartphones being equipped with better sensors and dual camera setups (which allows for stereo visual odometry, which is

potentially more accurate than the monocular counterpart). With better hardware, the potential for more precise and unlimited localization techniques for mobile devices increases.

5.1 FUTURE WORK

In the context of future work, there are many improvements that could be done to both the android and computer applications.

After multiple tests with the android app, when looking to a specific marker, many times the algorithm detects other markers which are in the background of the camera frame, which many times are not in a prime position to be tracked, as they may be partially occluded or in an unfavourable angle or even too small for robust tracking. For this reason, the marker tracking algorithm only takes into account one marker at a time. Developing a multi-concurrent marker algorithm would require a system which discards markers in unfavourable conditions. This would obviously be an improvement to the current implementation, as it would rely on multiple markers for positioning information. Also, calculating how much space the marker occupies in the camera frame (and if part of it is beyond the camera's field of view) to measure how reliable the tracking information from that marker is (a small or partial marker provides worse tracking information). This could be calculated using the size of the marker and its Model View Matrix combined with the projection matrix. In a 3D context, the immersion of the rendered objects could be increased by using the real 3D model to calculate which parts of the virtual 3D objects should be visible. For example, if a user pointed the camera at a building facade in Ponte de Lima, only the part of the wall which should be in front of the building would be rendered, as everything that is behind it is considered to be occluded.

For the assistant app, the camera simulation feature could have some improvements. Although the marker's size does somewhat influence the outcome of the simulation, it is not a decisive factor.

BIBLIOGRAPHY

Acko. Making Worlds 2 - Scaling Heights. https://acko.net/blog/making-worlds-2-scaling-heights/. Accessed: 2017-09-24.

Mohammad OA Aqel, Mohammad H Marhaban, M Iqbal Saripan, and Napsiah Bt Ismail. Review of visual odometry: types, approaches, challenges, and applications. *SpringerPlus*, 5(1):1897, 2016.

ARToolkit. ARToolkit. https://archive.artoolkit.org/. Accessed: 2017-09-23.

Assimp. Assimp. http://assimp.sourceforge.net/. Accessed: 2017-09-23.

Ronald T Azuma. A survey of augmented reality. *Presence: Teleoperators and virtual environments*, 6(4):355–385, 1997.

Tim Bailey and Hugh Durrant-Whyte. Simultaneous localization and mapping (slam): Part ii. *IEEE Robotics & Automation Magazine*, 13(3):108–117, 2006.

Herbert Bay, Tinne Tuytelaars, and Luc Van Gool. Surf: Speeded up robust features. In *European conference on computer vision*, pages 404–417. Springer, 2006.

Beebom. What is GLONASS And How It Is Different From GPS. https://beebom.com/what-is-glonass-and-how-it-is-different-from-gps/. Accessed: 2017-10-24.

Simon Burkard. Visual odometry on mobile devices for virtual walkthroughs. Institute of Software Engineering and Theoretical Computer Science Quality and Usability Lab, Technische Universitat Berlin, 2015.

Michael Calonder, Vincent Lepetit, Christoph Strecha, and Pascal Fua. Brief: Binary robust independent elementary features. In *European conference on computer vision*, pages 778–792. Springer, 2010.

Francois Caron, Emmanuel Duflos, Denis Pomorski, and Philippe Vanheeghe. Gps/imu data fusion using multisensor kalman filtering: introduction of contextual aspects. *Information fusion*, 7(2):221–230, 2006.

Peter Corke, Dennis Strelow, and Sanjiv Singh. Omnidirectional visual odometry for a planetary rover. In *Intelligent Robots and Systems, 2004.(IROS 2004). Proceedings. 2004 IEEE/RSJ International Conference on*, volume 4, pages 4007–4012. IEEE, 2004.

Hugh Durrant-Whyte and Tim Bailey. Simultaneous localization and mapping: part i. *IEEE robotics & automation magazine*, 13(2):99–110, 2006.

Friedrich Fraundorfer and Davide Scaramuzza. Visual odometry: Part ii: Matching, robustness, optimization, and applications. *IEEE Robotics & Automation Magazine*, 19(2):78–90, 2012.

Namrata Ravindra Garach. *Fine grained location using mobile augmented reality*. PhD thesis, San Diego State University, 2013. URL=http://sdsu-dspace.calstate.edu/bitstream/handle/10211.10/3560/Garach_Namrata.pdf;sequence=1.

Google. Google Maps. https://www.google.pt/maps. Accessed: 2017-10-10.

Google Play Store. Pokémon GO. https://play.google.com/store/apps/details?id=com.nianticlabs.pokemongo&hl=pt_PT. Accessed: 2017-10-13.

Chris Harris and Mike Stephens. A combined corner and edge detector. In *Alvey vision conference*, volume 15, page 50. Citeseer, 1988.

Tobias Höllerer, Steven Feiner, Tachio Terauchi, Gus Rashid, and Drexel Hallaway. Exploring mars: developing indoor and outdoor user interfaces to a mobile augmented reality system. *Computers & Graphics*, 23(6):779–785, 1999.

Andrew Howard. Real-time stereo visual odometry for autonomous ground vehicles. In *2008 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 3946–3952. IEEE, 2008.

International Business Times. 'Pokémon Go' Japan Release Dates: Nintendo To Launch Game In Tie-Up With McDonald's; Shares Double To $42.5 Billion. http://www.ibtimes.com/. Accessed: 2017-10-11.

Yan Ke and Rahul Sukthankar. Pca-sift: A more distinctive representation for local image descriptors. In *Computer Vision and Pattern Recognition, 2004. CVPR 2004. Proceedings of the 2004 IEEE Computer Society Conference on*, volume 2, pages II–506. IEEE, 2004.

Chris D Kounavis, Anna E Kasimati, and Efpraxia D Zamani. Enhancing the tourism experience through mobile augmented reality: Challenges and prospects. *International Journal of Engineering Business Management*, 4:10, 2012.

AR Kulaib, RM Shubair, MA Al-Qutayri, and Jason WP Ng. An overview of localization techniques for wireless sensor networks. In *Innovations in Information Technology (IIT), 2011 International Conference on*, pages 167–172. IEEE, 2011.

Henning Lategahn. *Mapping and Localization in Urban Environments Using Cameras*, volume 28. KIT Scientific Publishing, 2014.

John J Leonard and Hugh F Durrant-Whyte. Mobile robot localization by tracking geometric beacons. *IEEE Transactions on robotics and Automation*, 7(3):376–382, 1991.

Fotis Liarokapis, Nikolaos Mourkoussis, Martin White, Joe Darcy, Maria Sifniotis, Panos Petridis, Anirban Basu, and Paul F Lister. Web3d and augmented reality to support engineering education. *World Transactions on Engineering and Technology Education*, 3(1): 11–14, 2004.

David G Lowe. Distinctive image features from scale-invariant keypoints. *International journal of computer vision*, 60(2):91–110, 2004.

Bruce D Lucas, Takeo Kanade, et al. An iterative image registration technique with an application to stereo vision. In *IJCAI*, volume 81, pages 674–679, 1981.

Hans Moravec. Obstacle avoidance and navigation in the real world by a seeing robot rover. In *tech. report CMU-RI-TR-80-03, Robotics Institute, Carnegie Mellon University & doctoral dissertation, Stanford University*, number CMU-RI-TR-80-03. September 1980.

David Nistér. An efficient solution to the five-point relative pose problem. *IEEE transactions on pattern analysis and machine intelligence*, 26(6):756–770, 2004.

David Nistér. Preemptive ransac for live structure and motion estimation. *Machine Vision and Applications*, 16(5):321–329, 2005.

David Nistér, Oleg Naroditsky, and James Bergen. Visual odometry for ground vehicle applications. *Journal of Field Robotics*, 23(1):3–20, 2006.

OpenCV Docs. Understanding Features. `https://docs.opencv.org/3.0-beta/doc/py_tutorials/py_feature2d/py_features_meaning/py_features_meaning.html#features-meaning`. Accessed: 2017-10-24.

OpenGL-Tutorial. Tutorial 16 : Shadow mapping. `http://www.opengl-tutorial.org/intermediate-tutorials/tutorial-16-shadow-mapping/`. Accessed: 2017-09-24.

PL3D. Ponte de Lima 3D. `http://www4.di.uminho.pt/pl3d/`. Accessed: 2017-09-23.

Adrian Rosebrock. Find distance from camera to object/marker using python and opencv. `http://www.pyimagesearch.com/2015/01/19/find-distance-camera-objectmarker-using-python-opencv/`. Accessed: 2017-01-20.

Edward Rosten and Tom Drummond. Machine learning for high-speed corner detection. In *European conference on computer vision*, pages 430–443. Springer, 2006.

Eric Royer, Maxime Lhuillier, Michel Dhome, and Thierry Chateau. Localization in urban environments: monocular vision compared to a differential gps sensor. In *Computer Vision and Pattern Recognition, 2005. CVPR 2005. IEEE Computer Society Conference on*, volume 2, pages 114–121. IEEE, 2005.

Ethan Rublee, Vincent Rabaud, Kurt Konolige, and Gary Bradski. Orb: An efficient alternative to sift or surf. In *2011 International conference on computer vision*, pages 2564–2571. IEEE, 2011.

David Sachs. Sensor fusion on android devices: A revolution in motion processing. Google Tech Talk, 2010. URL https://www.youtube.com/watch?v=C7JQ7Rpwn2k.

Davide Scaramuzza and Friedrich Fraundorfer. Visual odometry [tutorial]. *IEEE Robotics & Automation Magazine*, 18(4):80–92, 2011.

Kurt Seifert and Oscar Camacho. Implementing positioning algorithms using accelerometers. *Freescale Semiconductor*, pages 1–13, 2007.

Jianbo Shi. Good features to track jianbo shi carlo tomasi computer science department computer science department cornell university stanford university ithaca, ny 14853 stanford, ca 94305.

Avi Singh and KS Venkatesh. Monocular visual odometry. 2015. Undergraduate Project, IIT Kanpur, URL=http://avisingh599.github.io/assets/ugp2-report.pdf.

Vuforia. Optimizing Target Detection and Tracking Stability. https://library.vuforia.com/articles/Solution/Optimizing-Target-Detection-and-Tracking-Stability.html, a. Accessed: 2017-09-26.

Vuforia. Targets. https://library.vuforia.com/articles/Solution/Targets.html, b. Accessed: 2017-10-21.

Wikitude. Wikitude SDK Android. http://www.wikitude.com/external/doc/documentation/4.0/android/imagerecognition.html. Accessed: 2017-09-26.

Amir Roshan Zamir and Mubarak Shah. Accurate image localization based on google maps street view. In *European Conference on Computer Vision*, pages 255–268. Springer, 2010.

Paul A Zandbergen and Sean J Barbeau. Positional accuracy of assisted gps data from high-sensitivity gps-enabled mobile phones. *Journal of Navigation*, 64(03):381–399, 2011.

Zhengyou Zhang. Determining the epipolar geometry and its uncertainty: A review. *International journal of computer vision*, 27(2):161–195, 1998.

Zion Market Research. Augmented reality (ar) market (sensor, display and software) for aerospace & defense, industrial, consumer, commercial, e-commerce, retail and other applications: Global industry perspective, comprehensive analysis, size, share, growth, segment, trends and forecast, 2015 – 2021. 2016.

SUPPORT MATERIAL

A.1 SENSOR PRECISION

In the following table is represented each sensor's accuracy, precision and possible bias for the Android device used in this dissertation.

Sensor	Samples	Mean (x,y,z)	Median (x,y,z)	Var (x,y,z)	Std Dev (x,y,z)
Accelerometer (m/s^2)	143 (∼ 4.76 samp/sec)	(0.20728671 , -0.09430769 , 9.84739860)	(0.191, -0.095, 9.845)	(0.0023514313 , 0.0001583976 , 0.0002359456)	(0.04849156 , 0.01258561 , 0.01536052)
Gravity (m/s^2)	147 (∼ 4.9 samp/sec)	(0.20184422 , -0.09379048 , 9.80407075)	(0.1997 , -0.0936 , 9.8041)	(9.573454e-05, 1.259594e-05, 4.879601e-08)	(0.0097844028, 0.0035490754, 0.0002208982)
Linear Acceleration (m/s^2)	148 (∼ 4.93 samp/sec)	(0.01411487 , 0.00116698, 0.04434294)	(-0.0077740 , 0.0024685 , 0.0409345)	(0.0037800817 , 0.0001233236 , 0.0001041905)	(0.06148237 , 0.01110512 , 0.01020737)
Gyroscope (rads/sec)	146 (∼ 4.87 samp/sec)	(-0.0005037066 , -0.0005119192 , -0.0000219003)	(-0.0003996803 , -0.0005329070 , -0.0003996803)	(8.972977e-06, 7.934258e-06, 5.928051e-06)	(0.002995493 , 0.002816782 , 0.002434759)
Magnetometer (μT)	146 (∼ 4.87 samp/sec)	(-30.18163, 204.92194, -24.72133)	(-29.45557 , 205.12695, -23.16284)	(2.884676 , 0.559630 , 12.796353)	(1.6984335 , 0.7480842 , 3.5771990)
Rotation Vector (unitless)	147 (∼ 4.9 samp/sec)	(-0.007458497 , -0.008555299 , -0.278256918)	(-0.007430 , -0.008443, -0.279147)	(5.490242e-08, 2.538265e-07, 4.362553e-05)	(0.0002343126, 0.0005038119, 0.0066049625)

Table 6.: Mean, Median, Variance and Standard Deviation of multiple sensors in a resting position for 30 seconds

Some incongruences can be noticed, like the mean and median of the accelerometer not being zero for the x and y axis (only the force of gravity should be present). This noticeable bias is due to an uncalibrated sensor. This creates a snowball effect as the gravity, linear acceleration and rotation vector sensors all use the accelerometer. This can be treated, although difficult to completely remove, using bias removal algorithms (like removing an offset to every sample). When comparing the accelerometer with the gyroscope and the magnetometer, the magnetometer has the highest variance and standard deviation, followed by the accelerometer, which is followed by the gyroscope. Although the three

sensors are represented in different units, it can still be confirmed that the magnetometer and accelerometer are noisier sensors than the gyroscope.

A.2 VUFORIA DEVELOPER PORTAL GUIDE

To create ARData projects, using the Vuforia Portal is required. Following in this section are brief guides to using the portal. The Vuforia Developer Portal is hosted at `https://developer.vuforia.com/target-manager`.

A.2.1 *Creating a License Key*

1. In the Main Page, go to "License Manager" → "Add License Key"

Figure 67.: Adding a License Key

2. In the Main Page, go to "License Manager" → Click the added License Key

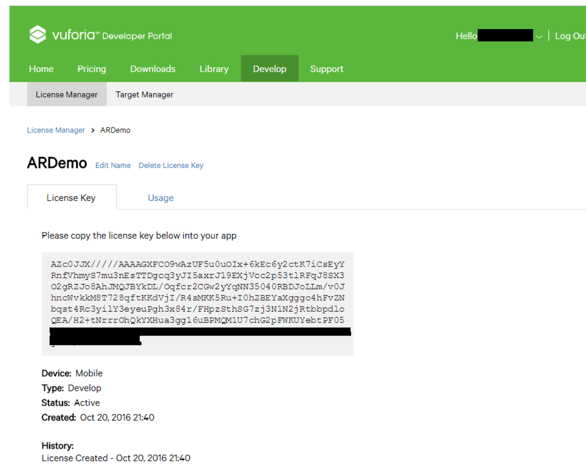


Figure 68.: Viewing a Vuforia Key

3. Use this Key when saving a project with the Computer Application. This will be used later when initializing Vuforia on the Android App.

A.2.2 *Creating Markers*

1. Before adding markers, the creation of a database is needed. The markers will be contained in it. In the Main Page, go to "Target Manager" → "Add Database"

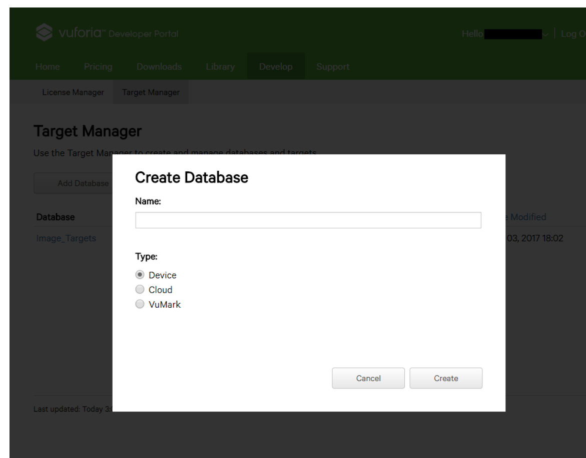


Figure 69.: Adding a database

2. Then, to create Markers, Click the added Database → "Add Target"

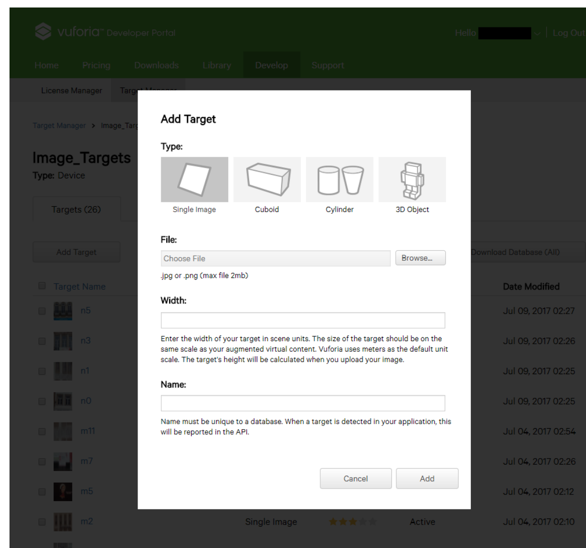


Figure 70.: Marker Creation

3. Check the properties of the added marker by selecting it.

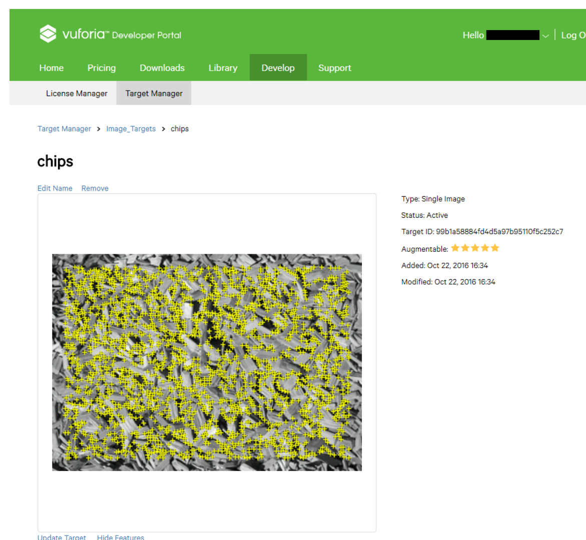


Figure 71.: Viewing Marker Properties

A.2.3 *Downloading Database*

1. To download the binaries of a database, click "Download Database" → "Android Studio"

Download Database

26 of 26 active targets will be downloaded

Name:
Image_Targets

Select a development platform:

Android Studio, Xcode or Visual Studio
Unity Editor

Cancel Download

Figure 72.: Database Download

A.3 COMPUTER APP GUIDE

A.3.1 *Key Bindings*

W: Walk Forward;

S: Walk Backwards;

SHIFT: Walking units per frame is increased;

LMB: Hold to rotate camera; Double click to select entity; Also used in the creation of entities;

RMB: Used in the creation of entities;

C: Begin Camera Creation Context;

M: Begin Marker Creation Context;

F: When a camera is selected, assumes that camera's position with the determined Android camera's parameters (Angle of Vision and Height);

E: Top-Down View;

P: Switch between Real 3D Model and Virtual 3D Model;

BACKSPACE: Delete entity;

A.3.2 *Creating Camera*

After pressing the Camera creation key, holding and dragging the Right Mouse Button selects the walkable region. Releasing the RMB will create the entity.

A.3.3 *Creating Marker*

When pressing the Marker creation key, holding and dragging the Right Mouse Button sticks the marker to the surface the pointer is on. Pressing LMB sets the marker at the apointed location. The middle mouse button can also be used to increase or decrease the distance between the surface and the Marker.

A.4 ANDROID APP GUIDE

To use the android app, select an ARData file in the menu. Click start and point the camera to a known marker.

A.5 REPOSITORIES

Android App: `https://github.com/Machinezero/ARMarker`

Computer App: `https://github.com/Machinezero/ARMarkerEditor`