Università degli Studi di Padova
Dipartimento di Ingegneria dell'Informazione

Corso di Laurea Magistrale in
Ingegneria Informatica

OPTAR: Automatic Coordinate Frame Registration between OpenPTrack and Google ARCore using Ambient Visual Features

Author:
Carlo Rizzardo

Supervisor:
Prof. Stefano Ghidoni
Advisor:
Prof. Jeff Burke

8 July 2019
Academic Year 2018/2019

Contents

List of Figures

Chapter 1

Introduction

Augmented Reality (AR) is a family of technologies that aim at providing experiences of the real world that are augmented with computer-generated information. The term can describe technologies involving different perceptual senses but the most successful approaches usually exploit vision. Traditional visual AR applications make use of a screen that displays the view of a camera, on top of this view, software overlays elements that are not present in the scene and keeps them in position within the real-world environment by tracking the movement of the device or other real-world elements.

Applications of this technology are countless, ranging from the medical field with technologies for supporting surgeons and nurses, to the video games, industrial design, education or entertainment. The video game industry, in particular, has recently brought Augmented Reality to the masses with AR games playable on smartphones (e.g. *Pokémon Go* [29]).

One important limitation of traditional AR applications is that the software can be aware only of what is inside the view of the camera. To tackle this limitation OPTAR aims at combining Augmented Reality with *OpenPTrack*, a system developed at *UCLA REMAP* (*Center for Research in Engineering Media And Performance*) that integrates people tracking, skeleton tracking, pose recognition and object tracking using a network of RGB-D cameras based on *ROS* (*Robot Operating System*).

The system aims to be a platform that allows to easily develop artistic and interactive installations, because of this it was decided to use *Android* devices as the user interface, exploiting *Unity3D* and *Google ARCore* for the phone-side application development.

The integration of these two independent systems requires the estimation of the registration between them. By knowing the transformation between the two

systems, OPTAR will be able to transfer the tracking information produced by *OpenPTrack* to the mobile device and at the same time inform *OpenPTrack* of the movements of the mobile device tracked by *Google ARCore*.

This thesis tackles the problem of estimating the registration by exploiting visual features seen in the environment by both the mobile camera and the fixed cameras of the *OpenPTrack* system. The system extracts the features from the images received from the different cameras and matches them to find common elements that are visible by both the fixed cameras and the smartphone.

The fixed cameras used by *OpenPTrack* are depth cameras, either *Kinect*, *Zed* or *Realsense*. Thanks to depth information it is possible to know the 3D position of the elements seen by the system. Using this information it is then possible to estimate the pose of the mobile camera using a *PnP* algorithm.

Once the phone pose is known in both the *OpenPTrack* and *ARCore* systems it is naturally possible to compute the registration between the two.

A crucial issue is that the registration between the two systems is not constant. The understanding of the world of *Google ARCore* changes and improves with time and, because of this, its frame of reference moves with respect to the fixed coordinate system of *OpenPTrack*.

This forces us to update the registration estimate continuously as time progresses.

Moreover, the *PnP* estimations of the phone pose are not perfect but naturally affected by noise. To get a more stable and reliable estimate of the phone position we employed an approach based on a *Kalman filter*. However, the rate at which the *PnP* estimations are computed is quite low, it is in the order of one estimation per second. This low frequency would make the estimations produced by the *Kalman filter* unreliable, to solve this issue we update the filter at a higher rate using the information produced by the *ARCore* tracking, effectively fusing the two estimation approaches.

An important characteristic of the proposed system is that it supports the use of multiple phones at the same time. An analysis of the number of devices the system is capable of handling has not been performed, but it has been tested with up to 4 devices without noticeable performance degradation.

To present the system, first of all, chapter 2 will describe the methods, tools, and techniques that have been used. The first two sections will introduce *OpenPTrack*, *Google ARCore*. Section 2.3 will discuss, in general terms, the integration of AR applications with fixed camera systems. Section 2.4 will provide a theoretical

description of the *Kalman Filter*. Section 2.5 will introduce the *PnP* problem and some approaches to its solution.

Chapter 3 will thoroughly describe the proposed method. Section 3.1 will give an overview of *OPTAR*'s high-level structure. Section 3.2 will detail the functionalities and the implementation of the *Unity Android* application. Section 3.3 will instead detail the internals of the *ROS* component of *OPTAR*.

Chapter 4 will describe the experimental analysis that has been performed. Section 4.1 will detail the setup of the quantitative evaluation that has been performed and present the results that were obtained. Section 4.2 instead describes and discusses a demonstrative application that has been developed, presenting some first practical conclusions.

Finally, chapter 5 first discusses the overall performance of the system. Then, section 5.1 presents some proposals for the improvement of *OPTAR*.

Chapter 2

Related Works

This chapter provides an outline of the systems used in this thesis and of some related works that tackle issues similar to those faced by OPTAR.

Sections 2.1 and 2.2 describe the two systems on which OPTAR is based: *OpenPTrack* and *Google ARCore*.

Section 2.3 explores previous approaches to the integration of *Augmented Reality* with fixed camera systems available in the literature.

Section 2.4 provides an introduction to the *Kalman Filter* and its application to pose tracking.

Section 2.5 describes the *PnP* problem and the approaches to its solution.

2.1 OpenPTrack

OpenPTrack is a multi-camera system for people tracking, object detection and pose tracking. The system has been developed as a collaboration between the *University of Padua*, *UCLA REMAP* (*Center for Research in Engineering, Media And Performance*)[2] and *Open Perception*[1] with the specific objective of supporting applications in education, art, and culture. OpenPTrack aims to support creative coders in the arts, culture, and educational sectors who wish to experiment with real-time person tracking as an input for their projects.

The system makes use of multiple RGB-D cameras connected through a ROS network. The original 2013 release supported the *Microsoft Kinect* and *Mesa Imaging Swissranger SR4500*, and provided only the people tracking functionality. Later versions added the support for the new *Kinect for Xbox One* (also known as *Kinect v2*) and implemented pose and object tracking. The pose tracking is based on *OpenPose* [7] [16] while the object tracking is based on *YOLO* [33].

To simplify the process of setting up a multi-camera system *OpenPTrack* provides a calibration pipeline for determining the relative poses of the fixed cameras and establishing a common reference frame. The procedure is based on the `calibration_toolkit` ROS package [5].

To perform the calibration the user must move a calibration checkerboard in front of the cameras. By combining the checkerboard pose estimates from the different cameras the system is capable of determining the relative poses of the sensors and performing the calibration.

The original central core of *OpenPTrack* is the person tracking. Differently from other components of the system the person tracking is based on a custom algorithm. As described in [27], the algorithm runs independently for each camera, then the multiple detections are fused together by a central tracking node. The detection algorithm first exploits the pointcloud to determine clusters of points that are candidates to be people detections, these detections are then accepted or rejected based on a HOG classifier applied to the RGB image. To further improve the results an *AdaBoost* classifier is applied to the depth image.

The tracking node, which fuses the different position detections, uses an *Unscented Kalman Filter* to perform the position tracking and exploits the *Mahalanobis* distance to associate new detections to the already established tracks.

The other components of *OpenPTrack*, namely the pose and object tracking, follow the same detection/tracker logic. The detections are obtained using either *OpenPose* or *YOLO* and then fused by a central tracking node.

Figures 2.1 and 2.2 show two examples of the position and pose tracking performed by *OpenPTrack*.

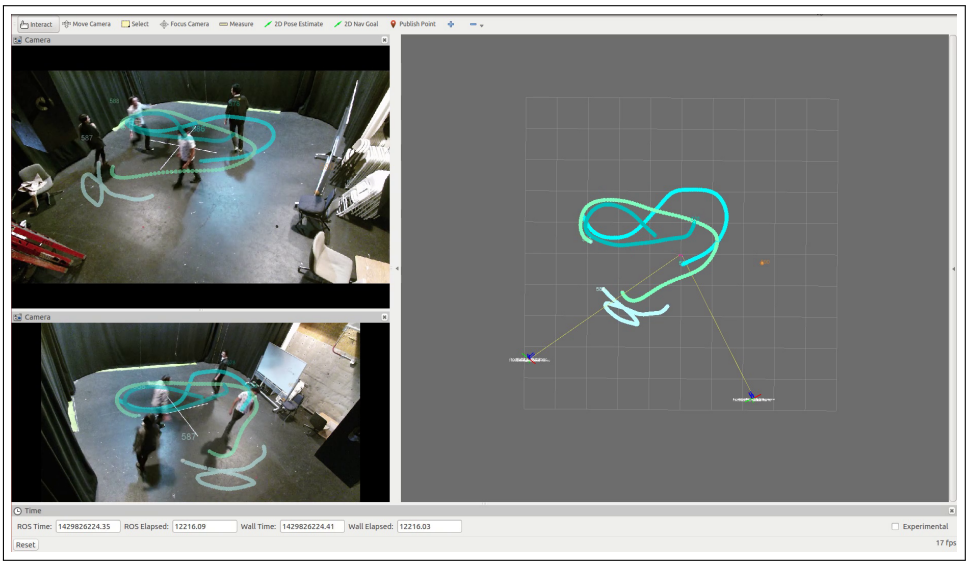


Figure 2.1: OpenPTrack position tracking
View of the OpenPTrack position tracking within the ROS RViz visualizer. This setup used two Kinect v2 sensors

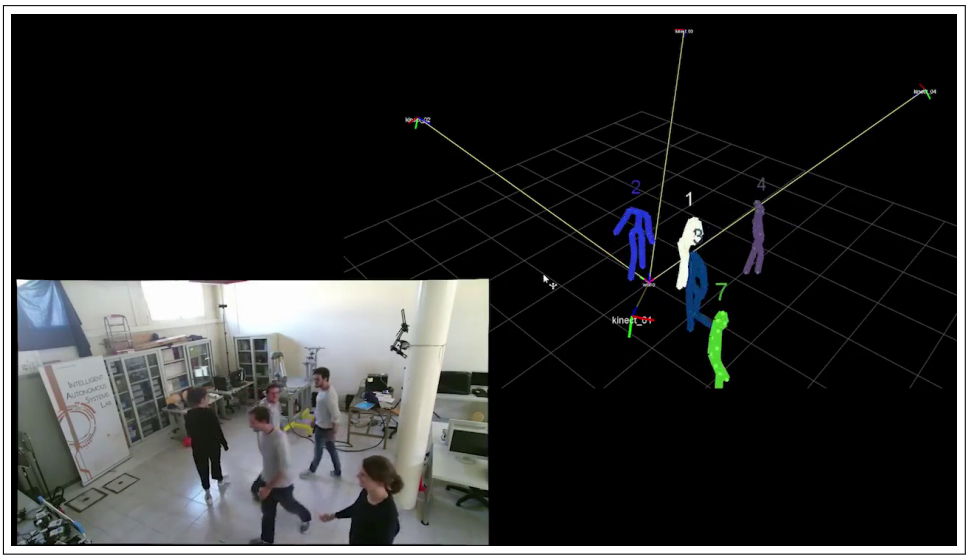


Figure 2.2: OpenPTrack pose tracking
View of the OpenPTrack pose tracking within the ROS RViz visualizer. This setup used three Kinect v2 sensors

2.2 Google ARCore

Google ARCore is a framework for the development of *Augmented Reality* applications on smartphones. The software requires *Android 7.0* or later to function and is currently officially supported only by a limited number of devices[15]. A version for *iOS* devices is also available, but it only provides an interface to *Apple ARKit* and does not implement *ARCore*'s main features.

On *Android* platforms, *ARCore* is distributed as a regular application, downloadable from the *Google Play Store*. *Google* then provides four different SDKs for the development of AR applications in four different environments: *Unity*, *Unreal Engine*, *Android NDK*, and the *Android* Java SDK. These *SDKs* are interfaces to the functionalities provided by the *ARCore* application, which acts as a library.

While the *ARCore* application is automatically updated via the regular *Google Play Store* system, the different SDKs are provided to the developer in different versions to be included in the AR application. This project uses the *Unity SDK*, version 1.7.0.

2.2.1 Google ARCore Fundamental Concepts

To provide *Augmented Reality* functionalities *ARCore* employs a *Simultaneous Localization And Mapping (SLAM)* approach [14]. The application continuously processes the camera feed to detect visually distinct features present the in scene. By tracking the position of these feature points within the camera images, and combining this knowledge with odometry estimations produced by the *Inertial Measurement Unit* (IMU) of the device, the software is capable of estimating the smartphone pose within the surrounding environment.[14]

The framework provides the developer the means to embed virtual 3D content within its internal representation of the world. From this, it is possible to generate an Augmented Reality view. By aligning the virtual-world camera with the real camera pose, provided by the SLAM system, it is possible to display the virtual world from the point of view of the real-world camera. By overlaying the rendering of the virtual image on the view obtained from the real camera the virtual content appears to be part of the real world.

A central aspect of *Google ARCore* is that its understanding of the geometry of the real world can change. This is due to the evolution of the map built by the underlying *SLAM* algorithm. When *ARCore* establishes that its internal representation of the real world can be enhanced it is free to move or rotate the origin of its coordinate frame, together with the location of the feature points it is tracking.

These movements will potentially change the correspondence between virtual-world poses and real-world poses, making the virtual objects move with respect to the pose they are intended to occupy in the real world.

To resolve this issue *Google ARCore* introduces the concepts of *Trackables* and *Anchors*[14]. *Trackables* are real-world features that *ARCore* is capable of tracking, the position of these entities can be tracked across mapping changes. In practice, *Trackables* can be either visually distinct feature points or planes.

Anchors are, instead, virtual world poses that are linked to the pose of a *Trackable*. As the pose estimate for a *Trackable* changes, the pose of the *Anchor* will also change. By attaching virtual world objects to *Anchors* it is possible to maintain the relative pose between a virtual object and a *Trackable*. This can, for example, allow placing a virtual object on a painting. The system can easily keep track of the painting pose, regardless of the changes in the mapping. By representing the painting as a *Trackable*, and anchoring the virtual object to it, the application will be able to maintain constant the pose of the object relative to the painting. This results in a very effective illusion.

2.2.2 OpenPTrack Integration Issues

The concept of *Trackables* and *Anchors* does not perfectly fit the use case of this project. The integration of the tracking data produced by *OpenPTrack* requires to represent in the virtual space objects that are not bounded to fixed feature points. There is not a fixed correspondence between the position of single persons in the scene and specific fixed features.

The different entities tracked by *OpenPTrack* are all in the same reference frame. This is why what the proposed system tries to estimate is the transformation between the fixed coordinate frame of *OpenPTrack* and the coordinate system of *ARCore*.

2.3 Integration of Augmented Reality and Fixed Camera Systems

The literature does not provide many examples of integration of Augmented Reality with fixed camera systems. For the most part, Augmented Reality systems have been implemented using just mobile devices performing an independent pose tracking.

An example of registration of a moving camera with a network of fixed cameras is presented in [18] by *Imre, Guillemaut, and Hilton*, one of the proposed applications is a scene augmentation demo. However, the setup is considerably different from that of this thesis. First of all, the work of *Imre et al.* proposes a complete integration of the mobile camera tracking with the fixed camera system, the camera pose is tracked directly in the coordinate frame of the fixed camera system. The approach proposed here for *OPTAR* instead aims at integrating, *Google ARCore* and *OpenPTrack*, two independent systems.

In the system by *Imre et al.* the mobile camera does not perform any mapping. The pose is estimated using only feature points that are visible from the fixed cameras. This implies that the mobile camera can only move in a very constrained area. This because the view of the mobile camera must always include feature points seen by the fixed cameras. Instead, in the approach proposed with *OPTAR*, the mobile camera can move more freely, as it does not need to constantly refer to the feature points seen by the fixed camera system.

Secondly, while *OPTAR* uses a network of RGB-D cameras, the work from *Imre et al.* uses regular cameras and relies on stereo triangulation to recover the 3D position of the feature points. This naturally requires the use of a greater number of cameras with respect to those used in this work. As an example, the experiments performed on *OPTAR* presented in this work (see section 4.2) only two fixed cameras, the setup presented by *Imre et al.* uses seven.

Moreover, the work of *Imre et al.* does not mention which hardware was used and what level of performance was achieved in terms of the rate of the pose estimation. Also, there is no mention of the network setup used to connect the mobile camera. This is important, as in this work a crucial limitation has been the rate of the exchange of image and feature information between the mobile camera and the fixed system, which had to be performed over WiFi.

Still, the fundamental approaches of the two methods to the estimation of the pose of the mobile camera make use of the same two fundamental techniques. In both cases, the camera pose is determined by solving a *Perspective-n-Point (PnP)*

problem and then the result is filtered exploiting a *Kalman Filter* (an introduction to *Kalman Filtering* and *PnP* is provided in the following sections). However, while *Imre et al.* use the camera pose estimate directly, in *OPTAR* it is used to compute the transformation between the static coordinate frame of *OpenPTrack* and the coordinate frame of *Google ARCore*.

2.4 The Kalman Filter

As already mentioned in chapter 1 a fundamental technique exploited by this work is the *Kalman Filter*. In particular, the filter is applied to the estimation of the smartphone pose, using the measurements provided by *Google ARCore* and those generated via *PnP* camera pose estimation.

In the literature there are numerous sources that discuss and explain *Kalman Filtering*, most a notably the paper from *Bishop & Welch*[40]. This section will provide an introduction to the filter, in particular, with a focus on the filtering of 3D poses.

2.4.1 The System Model

The *Kalman Filter* addresses the problem of estimating the internal state of a discrete linear system by observing a series of noisy measurements. The state is represented as a vector $x \in \mathbb{R}^n$ and its evolution is governed by the linear stochastic difference equation

$$x_k = A_k x_{k-1} + B_k u_{k-1} + w_{k-1} \tag{2.1}$$

With u_k being the control input and $w_k \in \mathbb{R}^n$ the process noise. The $n \times n$ matrix A controls the zero-input response of the system, it is called the *transition matrix*.

The measurement $z \in \mathbb{R}^m$ is instead defined as

$$z_k = H x_k + v_k \tag{2.2}$$

The vector v_k represents the measurement noise, and H is an $m \times n$ matrix, the *measurement matrix*.

The process and measurement noise w_k and v_k are assumed to be independent and Gaussian.

$$\begin{aligned} w_k &\sim N(0, Q_k) \\ v_k &\sim N(0, R_k) \end{aligned} \tag{2.3}$$

It is important to notice how the covariance matrices Q and R can change at each time step. In this thesis, this detail will be significant as it is exploited to fuse measurements generated by different sources.

2.4.2 The Discrete Kalman Filter Algorithm

The update algorithm of the *Kalman Filter* follows a two-steps process that resembles a feedback control approach. For each time step, the filter first predicts the new state of the system from the previous one, then, this estimate is corrected with the new measurement data.

The prediction step is performed by assuming the noise w_k to be null and applying equation 2.1, which becomes

$$\hat{x}_k^- = A_k \hat{x}_{k-1} + B_k u_{k-1} \tag{2.4}$$

Where $\hat{x}_{k-1}$ is the previous state estimate.

Afterwards, the correction step computes the difference between the actual measurement and the measurement that would have been obtained if the real state of the system was the predicted one. This difference is called *innovation*. The *innovation* value is then used to correct the predicted state. The entity of this correction is controlled by the *Kalman Gain* K_k.

$$\hat{x}_k = \hat{x}_k^- + K_k(z_k - H\hat{x}_k^-) \tag{2.5}$$

The *Kalman Gain* K_k is determined so to minimize the error covariance P_k of the *a posteriori* state estimate $\hat{x}_k$. The optimal K_k can be determined to be

$$K_k = P_k^- H^T (H P_k^- H^T + R)^{-1} \tag{2.6}$$

Where P_k^- is the error covariance of $\hat{x}_k^-$, the *a priori* error covariance.

As P_k is needed to perform the correction step, its value has to be estimated, along with the rest of the system state. This estimation is performed following the same prediction-correction logic of the state estimation, using the following formulas:

$$\begin{aligned} P_k^- &= A P_{k-1} A^T + Q \\ P_k &= (1 - K_k H) P_k^- \end{aligned} \tag{2.7}$$

By following the predict-update logic the filter can update the state estimate recursively, without looking back at past data at each step. This guarantees an extreme efficiency of the filtering process.

2.4.3 Tuning of Process and Measurement Noise Covariance

The Q and R matrices play a crucial role in controlling the filter behaviour. This because, by tuning the covariances values, it is possible to inform the filter of how reliable either the measurements or the predicted state are. In practice R and Q influence the computed value of the *Kalman Gain* K_k, amplifying or reducing the entity of the state correction.

The process noise represents all the dynamics of the state evolution that are not represented in the model defined by the prediction equation 2.1. Consequently, the process noise covariance indicates how frequent and extensive are the deviations of the real state from the predicted state.

The role of the process noise can be seen by taking as an example the estimation of the 1-dimensional position of a point. The state model could be defined as $x = (x, \dot{x})^T$, where $\dot{x}$ is the speed at which the point is moving. Equation 2.1 could then be adapted as

$$\hat{x}_k^- = \begin{bmatrix} 1 & T \\ 0 & 1 \end{bmatrix} \hat{x}_{k-1} + w_{k-1} \tag{2.8}$$

This model only represents the movement of the object at a constant velocity. Any change in velocity is a deviation from the base model. Changes in velocity are therefore represented by the process noise, which can represent any arbitrary change in the state.

The measurement noise, instead, represents the deviation of the measurements from the real value of the quantity they are observing.

Varying the values of Q and R changes the response of the filter to the innovation. Having a higher process covariance will increase the impact of the innovation as changes in the state will be deemed more plausible. A higher measurement covariance will instead decrease the impact of the innovation in the state correction.

Prediction equations (time update)
$\hat{x}_k^- = A_k \hat{x}_{k-1} + B_k u_{k-1}$
$P_k^- = A P_{k-1} A^T + Q$
Correction equations (measurement update)
$K_k = P_k^- H^T (H P_k^- H^T + R)^{-1}$
$\hat{x}_k = \hat{x}_k^- + K_k (z_k - H \hat{x}_k^-)$
$P_k = (1 - K_k H) P_k^-$

Table 2.1: Kalman Filter equations

Summary of the Kalman Filter predict and update equations

This because the variation in the measurements will be deemed to be likely due to observation noise.

As already mentioned, the R matrix does not need to be constant during the operation of the filter. It can be varied so to reflect the actual changes in the observation noise. In the use case of this project, this possibility will be exploited, enabling to feed into the filter measurements produced by different sources having very different characteristics.

2.4.4 Pose Filtering

Applying the *Kalman Filter* to the tracking of the 3D pose of an object requires the joint modeling of the evolution of position and orientation. This section will analyze the problem of applying *Kalman Filtering* to pose tracking. In the presented case only pose estimates are available as input and no direct measurement of velocity or acceleration is performed.

2.4.4.1 Position Filtering

The filtering of the position can be performed following the same logic of the example in the previous section, in a way similar to [35]. First of all, the state model has to be extended to represent the position in 3D space. Differently from the example in the previous section, this formulation will also include the acceleration component in the system state, similarly to the formulation in [12].

The state is defined as

$$x = (x, y, z, \dot{x}, \dot{y}, \dot{z}, \ddot{x}, \ddot{y}, \ddot{z})^T \tag{2.9}$$

The prediction equation is instead

$$\hat{x}_k^- = \begin{bmatrix} 1 & 0 & 0 & T & 0 & 0 & \frac{T^2}{2} & 0 & 0 \\ 0 & 1 & 0 & 0 & T & 0 & 0 & \frac{T^2}{2} & 0 \\ 0 & 0 & 1 & 0 & 0 & T & 0 & 0 & \frac{T^2}{2} \\ 0 & 0 & 0 & 1 & 0 & 0 & T & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & T & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & T \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \hat{x}_{k-1} \tag{2.10}$$

The *measurement matrix* is simply

$$H = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{2.11}$$

The prediction equation by itself only models an object moving with constant acceleration, changes in the acceleration are represented by the process noise, which has to be characterized appropriately.

In the model presented here, the process noise represents random changes in the acceleration as a *discrete Wiener Process*. A *Wiener Process* is the representation of the accumulation of uniform random independent increments of a value[4]. In the filter formulation this corresponds in random increments in the acceleration that occur between one timestep and the next one. In practice, this formulation represents the first two derivatives of the position within the system state, while the derivatives beyond the second order are represented as noise. The logic for this is that it would be meaningless to represent them within the state as they change too rapidly to be modeled.

$$\begin{aligned} w_k &= Gn_k \\ n_k &\sim N(0, \sigma_n^2) \\ G &= \begin{bmatrix} \frac{T^2}{2} \\ T \\ 1 \end{bmatrix} \end{aligned} \tag{2.12}$$

Within the *Kalman Filter* formulation, this modeling is expressed via the process noise covariance, which can be computed as

$$\begin{aligned} Q &= E[w_k w_k^T] \\ &= GE[n_k n_k^T]G^T \\ &= GG^T E[n_k n_k^T] \\ &= GG^T \sigma_n^2 \end{aligned} \tag{2.13}$$

2.4.4.2 Orientation Filtering

Applying the *Kalman Filter* to the orientation is more problematic. The most straightforward way of modeling the orientation is using the *Euler Angles* representation, which means representing an orientation with the *yaw, pitch* and *roll* angles along the object axes. By using this representation it is possible to use the same modeling devised for the position filtering, using yaw, pitch, and roll in place of x, y and z. However, this representation suffers from the issue of the gimbal lock, the loss of one degree of freedom due to the alignment of the rotation axes of the euler angles. In practice, this means that in specific configurations, which depend on the chosen *Euler Angles* convention, the represented pose is unable to rotate along one of its three major axes. This makes *Euler Angles* unsuitable for most applications.

To overcome the limitations of the *Euler Angles* the usual approach is to represent the orientation using unit quaternions. However, the update operations required by the quaternion representation are not linear and cannot be performed by the regular *Kalman Filter*.

Instead, the integration of a constant angular velocity for a Δt time is performed as:

$$q_t = e^{\frac{1}{2}\boldsymbol{\omega}\Delta t} q_0 \tag{2.14}$$

Where q_t is the orientation after time Δt, $\boldsymbol{\omega}$ is a vector representing the angular velocities along the x,y and z axes and q_0 is the initial orientation.

There are several examples in the literature that make use of the *Extended Kalman Filter* to tackle this issue [26][21][25]. Indeed, the *Extended Kalman Filter* allows to perform the prediction step using non-linear operations, however, this is not enough for the use case of this thesis. Differently from other works, the system developed in this thesis does not have access to direct measurements of angular velocity or acceleration. To include in the model the angular velocity or acceleration, the filter would need to represent them as process noise, in the way this is done for the position. However, the non-linear relationship of 2.14 cannot be represented with an *Additive Gaussian* noise as in 2.12.

For this reason, and because the occurrence of gimbal lock issues is limited in the case of handheld devices, this work will use the *Euler Angles* representation.

2.5 The Perspective-n-Point (PnP) Problem

The *Perspective-n-Point* (*PnP*) problem is the problem of estimating the pose of a camera from two sets of corresponding 3D and 2D points, representing respectively points in the scene and their projection on the camera image.

The solutions to this problem can be categorized in iterative approaches and linear approaches[38]. Iterative approaches solve the problem by minimizing an optimization variable, usually the reprojection error, using non-linear least squares optimization. Linear approaches usually follow the linear programming paradigm by representing the problem as a system of linear constraints.

While iterative methods provide the best quality estimates, they are inherently slower than their counterparts. Moreover, a limitation of the iterative methods is their need for an initialization value from which to start the optimization procedure. Also, depending on the initialization value, the optimization may converge to a local minimum or maximum and consequently settle on an incorrect solution.

Among the linear approaches, the most simple and generic is the *direct linear transform* (*DLT*)[37]. The algorithm does not only perform the pose estimation, but it also computes both the extrinsic and intrinsic camera calibration parameters.

The algorithm defines two equations for each corresponding pair of points, generating a system of $2n$ linear equations. The unknowns of the system are the 12 components p_{ij} of the camera matrix P, once the full calibration is known it is possible to decompose it in its intrinsic and extrinsic components, determining the pose of the camera.

For each pair (u, v) and (X_i, Y_i, Z_i) of an image point and a world-frame point the following two equations are defined:

$$\begin{aligned} u_i &= \frac{p_{00}X_i + p_{01}Y_i + p_{02}Z_i + p_{03}}{p_{20}X_i + p_{21}Y_i + p_{22}Z_i + p_{23}} \\ v_i &= \frac{p_{10}X_i + p_{11}Y_i + p_{12}Z_i + p_{13}}{p_{20}X_i + p_{21}Y_i + p_{22}Z_i + p_{23}} \end{aligned} \tag{2.15}$$

To estimate the 12 parameters the algorithm requires at least six pairs of points. To produce more accurate results it is possible to provide more pairs and solve the system with a least-squares iterative algorithm. However, as the algorithm does not make use of any preexisting knowledge of the intrinsics of the camera, the quality of its estimates results to be lower with respect to other methods.

Indeed, the intrinsic camera calibration is usually already known, as it will be in the system developed in this work. More specialized approaches are available

that make use of this information. Estimating the pose of an already calibrated camera requires a minimum of 4 3D-2D point correspondences, the problem is well studied and there are numerous approaches to its solution. Methods are available for estimating the pose using just 3 correspondences [13] (this is known as the *P3P* problem and returns up to 4 possible solutions), 4 correspondences [17] or 5 correspondences [39]. Nevertheless, methods that use a limited number of points are highly vulnerable to the influence of noisy measurements, methods that are capable of exploiting greater numbers of correspondences are more stable and reliable.

2.5.1 EPnP: Efficient Perspective-n-Point

A notable approach is *EPnP* [19], it is a non-iterative solution to the PnP problem. The algorithm, with an $O(n)$ computational complexity, is extremely more efficient than the other most notable solutions, which have complexities of $O(n^4)$ [32] or $O(n^8)$ [3]. Moreover, the accuracy of *EPnP*'s estimates is on the same level of iterative methods like [20], while being significantly faster and not requiring an initialization input pose.

Most *PnP* algorithms solve the problem through the computation of the distance of all the points from the camera. The *EPnP* algorithm instead simplifies the problem by representing all of the 3D points as linear combinations of 4 virtual *control points* $(\boldsymbol{c}_1^w, \boldsymbol{c}_2^w, \boldsymbol{c}_3^w, \boldsymbol{c}_4^w)$.

Following this formulation, the i-th 3D point $(X_i^w, Y_i^w, Z_i^w)^T$, represented in the world coordinate frame, can be defined to as

$$\begin{bmatrix} X_i^w \\ Y_i^w \\ Z_i^w \end{bmatrix} = \sum_{j=1}^{4} \alpha_{ij} \boldsymbol{c}_j^w \tag{2.16}$$

To reach the solution to the problem, the algorithm aims at determining the position of the control points in the camera reference frame. Once the control points coordinates are known in both the world and camera reference frames it is possible to easily determine the camera pose. Equation 2.16 can be converted to the camera reference frame as

$$\begin{bmatrix} X_i^c \\ Y_i^c \\ Z_i^c \end{bmatrix} = \sum_{j=1}^{4} \alpha_{ij} \boldsymbol{c}_j^c \tag{2.17}$$

From this equation, indicating the intrinsic calibration camera matrix as K, we can express the relation between each 2D image point and its corresponding 3D point in the camera coordinate frame as

$$w_i \begin{bmatrix} u_i \\ v_i \\ 1 \end{bmatrix} = K \begin{bmatrix} X_i^c \\ Y_i^c \\ Z_i^c \end{bmatrix} = \begin{bmatrix} f_u & 0 & u_c \\ 0 & f_v & v_c \\ 0 & 0 & 1 \end{bmatrix} \sum_{j=1}^{4} \alpha_{ij} \begin{bmatrix} c_{x,j}^c \\ c_{y,j}^c \\ c_{z,j}^c \end{bmatrix} \tag{2.18}$$

This relationship can be expressed with the two linear equations

$$\begin{aligned} \sum_{j=1}^{4} (\alpha_{ij} f_u c_{x,j}^c + \alpha_{ij}(u_c - u_i) c_{z,j}^c) = 0 \\ \sum_{j=1}^{4} (\alpha_{ij} f_v c_{y,j}^c + \alpha_{ij}(v_c - v_i) c_{z,j}^c) = 0 \end{aligned} \tag{2.19}$$

Having n pairs of 2D-3D points this corresponds to a system of $2n$ equations in the 12 unknowns c_{ij}^c. This equation can also be expressed in matrix form as

$$M\boldsymbol{x} = 0 \tag{2.20}$$

With $\boldsymbol{x} = (\boldsymbol{c}_1^{cT}, \boldsymbol{c}_2^{cT}, \boldsymbol{c}_3^{cT}, \boldsymbol{c}_4^{cT})^T$ and M being an $2n \times 12$ matrix.

As explained in [19], this particular system can be solved in $O(n)$ time. As also the other steps of the are at least $O(n)$ this results to be the overall complexity of the algorithm.

Chapter 3

The Proposed Method

This chapter will provide a detailed description of the proposed method. The first section provides a description of the high-level structure of the system. First by providing an overview of the communication network connecting the mobile AR devices and the computers, then, by introducing the distinction between the *Android* side of the system and the *ROS* side of the system.

The chapter follows by discussing separately the *Android Unity* application and the *ROS* side of the system.

Section 3.2 describes the smartphone application, defining the package structure and the implementation of the most important functionalities.

Section 3.3 details the implementation of the *ROS* side. First by defining the nodes and topics that compose the *ROS* network, and then by describing the implementation of the procedure for the smartphone pose estimation and the registration estimation.

3.1 The General Structure

Estimating the registration between *Google ARCore* and *OpenPTrack* consists in determining the geometric transformation between the coordinate frames used by the two systems. The method proposed by this thesis performs this estimation by matching the smartphone pose in one system with the smartphone pose in the other. The pose in the *ARCore* reference frame is provided by *ARCore* itself, while the pose in the *ROS OpenPTrack* coordinate frame has to be estimated independently.

This latter estimate is computed by exploiting common visual features seen by both the phone and the fixed cameras. From the correspondence between the phone pose in the two systems, it is naturally possible to compute the transformation between the *ROS* and *ARCore* coordinate frames, simply as:

$$A_r = P_r * P_a^{-1} \tag{3.1}$$

Where A_r is the transformation from the *ARCore* frame to the *ROS* one, P_r is the pose in the *ROS* frame, and P_a is the pose in the *ARCore* frame.

To determine the aforementioned common visual features the system extracts *ORB* descriptors [34] from the fixed and mobile cameras' images, these descriptors are then matched to find features that are present in both images.

Once the matches have been determined it is possible to estimate the phone pose by solving a *PnP* problem (an introduction to the *PnP* problem is provided in section 2.5). The 3D positions of the matches in the *ROS* coordinate frame are computed exploiting the depth information provided by the fixed cameras.

The *PnP* pose estimation of the smartphone is performed independently for each fixed camera. The estimates produced using the single fixed cameras are then aggregated and coupled with the corresponding *ARCore* poses to compute the transformation between the *ARCore* and *OpenPTrack* coordinate frames.

However, the pose estimates computed using *PnP* can be noisy. Because of this, the proposed method applies a *Kalman filter* to these phone pose estimates, allowing to reduce the effect of noisy measurements. The filter, however, requires the rate of its input to be high enough to represent the motion of the phone, and the rate at which the *PnP* pose estimates are produced is not sufficient. To provide an adequate input to the filter, additional pose estimates are generated, by using the last computed registration to transform the pose estimates produced by *ARCore* into the *OpenPTrack* coordinate frame.

Figure 3.1 represents the high-level structure of the system. Each smartphone is represented by a node and each fixed camera has a corresponding *"pose estimator"*

node. The *PnP* and *ARCore* pose estimates are collected by a single *"registration estimator"* node which computes the registration estimates for all the smartphones.

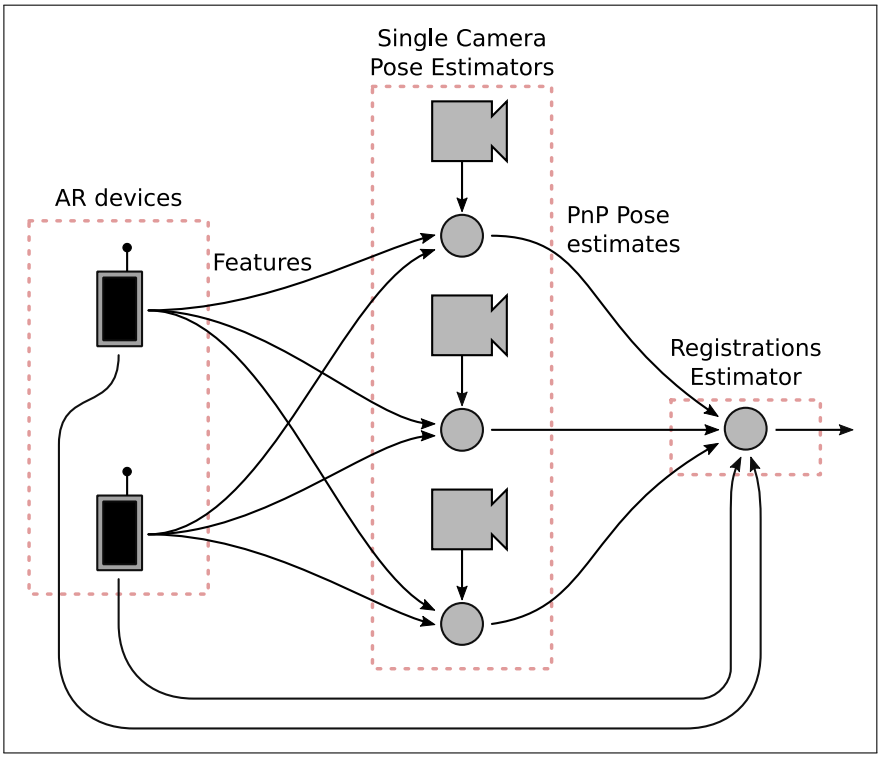


Figure 3.1: High-level system structure
Representation of the general system structure, with two AR devices and three fixed cameras

The *Android* application has been developed with *Unity3D* and *C#*, because of this it cannot interface directly with the *ROS* network. The communications are instead handled by *RosSharp*, which in turn uses the *WebSocket* [11] protocol to deliver messages to *ROS*. The *WebSocket* interface for *ROS* is provided by the *rosbridge_suite* package [9].

The feature extraction is performed on the smartphones, to reduce the latency involved with the network transmission, and to avoid situations of network saturation in the presence of multiple AR devices.

In support of the described system, there are some minor functionalities that had to be implemented.

The most notable is a simple *NTP* [24] client/server system, built upon the *ROS* messaging infrastructure. The need for it emerged from the synchronization issues between the smartphones and the *ROS* computers. Due to the difficulties of interfacing *Android* with existing local-network *NTP* services, the issue was

addressed by implementing a simple custom system composed of a *ROS* server node and a client class within the *Unity* application.

The following sections will describe in more detail the various components of the system. By addressing first the smartphone side and then the different *ROS* nodes.

3.2 The Android Unity Package

The *Android* component of OPTAR is tasked with extracting features from the smartphone camera feed and delivering them to the *ROS* side, which will compute the registration. Once the registration has been computed, the smartphone must be able to receive it, together with the tracking data produced by *OpenPTrack*.

These functionalities have been implemented as a *Unity* package, so to allow an easy development of different AR applications based on OPTAR. The code has actually been split into two packages, the `Optar` package contains the scripts that are solely related to the Augmented Reality functionalities, the `OpenPTrackCommons` package contains the scripts meant for interfacing with *ROS* and *OpenPTrack*.

The most important *Unity* scripts defined by the two packages are now introduced. A complete list of the scripts is instead provided in figures 3.2 and 3.3. An application wanting to use the functionalities offered by OPTAR will need to instantiate at least one of each of the following scripts as a *Unity MonoBehaviour*.

`OptarController` handles the initialization of the *ARCore* tracking and establishes the connection with the *ROS* network.

`CameraFeaturesPublisher` is tasked with extracting the *ORB* features from the camera feed and sending them to the *ROS* side of the system. More details are provided in section 3.2.1.

`ArcorePosePublisher` periodically publishes the current smartphone pose on a *ROS* topic specific to this device. By default, it will publish 30 times per second.

`HeartbeatPublisher` periodically publishes a message containing the device ID on a topic common for all devices. By default, it will publish once per second. The heartbeat topic is used to monitor the presence of active AR devices.

`PoseManager` keeps track, using the *ARCore* API, of the movements of the origin of the *ARCore* coordinate frame. This tracking is not always possible, so this does not solve the fundamental issue of the registration not being a constant transformation.

`NtpClient` establishes a connection with the *NTP* server node within the *ROS* network, and provides to the other scripts an interface for getting synchronized timestamps.

The *Optar* package provides a *Unity Prefab* with these scripts already defined and configured in it. By using the *Prefab* it is possible to easily set up the components needed to use OPTAR within a new application.

Once these scripts have been defined it is possible to use the `CentroidReceiver` and `SkeletonsReceiver` scripts from the `OpenPTrackCommons` package to receive the centroid and pose tracking data from *OpenPTrack*. To listen to the *ROS* transforms provided via `\tf` it is possible to use the `TfListener` script.

```
OpenPTrackCommons/scripts
├── messages
│   └── ..
├── CentroidReceiver.cs
├── CentroidsVisualizer.cs
├── FacesManager.cs
├── GenericPublisher.cs
├── GenericSubscriber.cs
├── NtpClient.cs
├── OptLogger.cs
├── SafeGenericSubscriber.cs
├── SetupCentroidParticleSystem.cs
├── SetupCentroidSphere.cs
├── SkeletonsReceiver.cs
├── SkeletonsVisualizer.cs
├── TfListener.cs
└── Utils.cs
```

Figure 3.2: The OpenPTrackCommons package
The folder structure of the OpenPTrackCommons Unity package

```
Optar/Scripts
├── messages
│   └── ..
├── ArcoreCameraPublisher.cs
├── ArcorePosePublisher.cs
├── CameraFeaturesPublisher.cs
├── HeartbeatPublisher.cs
├── MarkerTracker.cs
├── OptarController.cs
└── PoseManager.cs
```

Figure 3.3: The Optar package
The folder structure of the Optar Unity package

3.2.1 Feature Extraction and Publishing

The *ORB* features extraction is performed in the `CameraFeaturesPublisher` script. The script periodically retrieves the current camera image through the *ARCore* API, coupled with the current camera pose and the current *ROS* timestamp. Once this data is acquired it spawns a new thread to perform the data processing outside of the *Unity* UI thread.

Within this new thread, the *ORB* features are extracted using the *OpenCV* interface provided by the *OpenCV+Unity* package [31]. The provided *C#* interface matches almost exactly the original *C++* one.

The features are extracted by first detecting the best keypoints according to the *Harris* score, and then computing the corresponding *ORB* descriptors. The *OpenCV ORB* detector performs the search on multiple scales by scaling down the image. The number of extracted features, the number of scale levels and the scale factor are exposed as parameters of the *MonoBehaviour* object. As such, they can be tuned from the *Unity Inspector* and accessed by the application logic. By default, the number of features is set to 1000, the number of levels to 10 and the scale factor to 1.18.

Once the features have been extracted, a message containing the keypoints, the descriptors, the camera pose, the camera parameters, and the timestamp, is published to the ROS network. The message format also allows to attach an image, this is only used for debugging purposes, to send an extremely low-resolution camera image. During normal operation, the image field is left empty.

Not considering the camera image, the size of a message containing 1000 features is about 59KB. Of this, 60000 bytes are taken by the keypoints and the descriptors. By default, each device sends a message every second, giving a rate of 59KB/sec per phone, which can be safely handled by modern WiFi networks.

The thread tasked with the feature extraction and the message sending is protected by a mutex, so, in case the device is not able to execute the procedure within the required period of time, the next computation is delayed, making the publication rate drop safely.

3.3 The ROS Infrastructure

The *ROS* side of the system is tasked with receiving the features from the mobile devices, extracting the features from the fixed cameras and computing the registration for the different *ARCore* devices.

The *ROS* network is articulated in one node for each fixed camera, tasked with the estimation of the smartphone poses via *PnP*, one main node which computes the registration for all the AR devices and two other support nodes.

`ardevices_pose_estimator_single_camera_raw` is the node type used for estimating the poses via *PnP*. All of the nodes of this type will publish the estimates on a common topic.

`ardevices_registration_estimator` receives all of the *PnP* and *ARCore* pose estimates and computes the registration estimates for all the AR devices.

`ntp_server` is the server-side node for the simple *NTP* system used to synchronize the clocks of the mobile devices.

`ardevices_poses_republisher` receives the AR devices poses on their respective topics, aggregates them into an array and publishes them at a regular 30fps rate. At the same time, it publishes the *marker* topic for the *RViz* visualization of the phones

An example of a possible resulting network is depicted in figure 3.4.

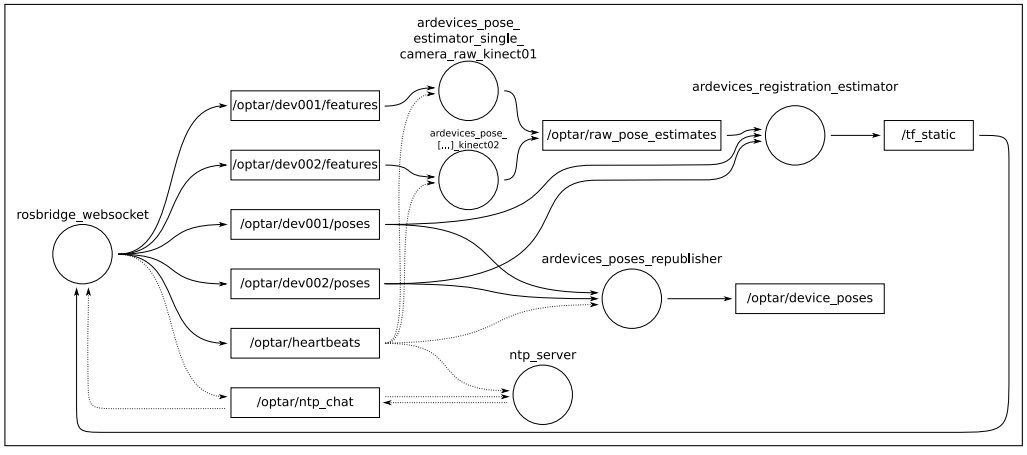


Figure 3.4: The ROS network
Representation of the Optar ROS node/topic network, with two mobile devices and two fixed cameras. The links not directly related to the registration estimation have been represented as dotted lines.

3.3.1 Single Camera PnP Pose Estimation

Each one of the `ardevices_pose_estimator_single_camera_raw` nodes generates *PnP*-based pose estimates for every running AR device using its corresponding fixed camera. Each node continuously receives as an input the greyscale and depth images from the fixed camera, together with the camera parameters. At the same time, it receives, from all the AR devices, their camera images and parameters coupled with the corresponding *ARCore* pose.

To handle multiple mobile devices the node listens to the `heartbeat` topic. When a new device is detected, the node constructs a handler that will listen to the topics specific to the device and perform the pose estimations. The handler will stop when no messages are received for a predefined timeout time (by default 5 seconds).

To synchronize the image messages coming from the fixed and mobile cameras the node takes advantage of the functionalities offered by the *message_filters* [10] *ROS* package, specifically of the `ApproximateTime` policy for the `Synchronizer` class.

For each new message received from a mobile device, the node executes the following steps.

1. The contents of the message are extracted, in particular, the ORB feature descriptors are converted from their raw representation to the *OpenCV* types `cv::KeyPoint` and `cv::Mat`.

2. The *ORB* features are extracted from the fixed camera image.

3. The fixed-camera features are matched to the mobile-camera ones.

4. The matches are filtered, imposing a threshold on the matched descriptors distance. Redundant matches are removed.

5. The 3D position of the matches is computed exploiting the depth image from the fixed camera.

6. If there are enough matches the mobile camera pose is estimated using the `solvePnPRansac()` method provided by *OpenCV*.

7. A number of different heuristics are applied to discard bad estimates.

A more detailed description of the most critical of these steps is provided in the following sections.

3.3.1.1 Feature Matching

The matching of the computed ORB features is performed by employing the `cv::BFMatcher` class from *OpenCV*, which performs a brute-force matching between the two feature sets.

The matches are then filtered by imposing a maximum threshold on the descriptor distance between each matched couple. The *OpenCV ORB* implementation employs a pattern of 256 point couples, resulting in a 256-bit descriptor. The distance between two descriptors is computed as the Hamming distance between the two. By default, the distance threshold has been set to 30.

The feature extraction is computed at multiple scales, consequently, multiple descriptors may be computed for the same image feature, both on the fixed camera and mobile camera sides. This implies there could be redundant matches, linking the same two image location more than once. At the same time, there could be contradicting matches, linking the same point in one image to different locations in the other one. Both of these details can compromise the quality of the smartphone pose estimation. The *PnP* computation requires the points to be unique and well distributed in the space. Moreover, contradicting matches obviously imply a contradictory input for the *PnP* algorithm.

To address these issues redundant matches are merged into a single one, and the contradictory ones are removed.

The procedure for this has been implemented as follows:

Algorithm 1: Removal of redundant and contradictory feature matches

Let M be the set of all the matches
Letting x be a match $x_{posArcore}$ is the position of x in the mobile camera image, x_{posRos} is the position in the fixed camera image
Let t be the maximum distance for two keypoints to be considered the same

---

foreach $m \in M$ **do**
  $M_{sameOrigin} = \{\ n \in M \mid dist_{eucl}(n_{posArcore}, m_{posArcore}) < t\}$;
  if $\forall\ m \in M_{sameOrigin}\ dist_{eucl}(n_{posRos}, m_{posRos}) < t$ **then**
    $m_{merged} = m$;
    $m_{merged}.dist = sum(M_{sameOrigin}.dist)/|M_{sameOrigin}|$;
    $M_{filtered}.push(m_{merged})$;
  end
  $M = M \setminus M_{sameOrigin}$
end
return $M_{filtered}$

3.3.1.2 Matches 3D Position Computation

An important issue arises in the computation of the 3D position of the matches. The position can be computed exploiting the depth information provided by the fixed camera, simply applying the following formulas:

$$\begin{aligned} X &= (x_I - c_x) * d(x_I, y_I)/f_x \\ Y &= (y_I - c_y) * d(x_I, y_I)/f_y \\ Z &= d(x_I, y_I) \end{aligned} \tag{3.2}$$

Where (X, Y, Z) is the 3D position, (x_I, y_I) is the feature position in the fixed camera image, (c_x, c_y) is the principal point position, $d(x_I, y_I)$ is the depth value, f_x and f_y are the focal lengths of the fixed camera. The regular and depth images received from the fixed camera are assumed to be already rectified and registered.

What makes this computation problematic is the fact that the depth information for a specific image position is not always available. The depth often cannot be determined along the borders of objects. Moreover, in the case of *Kinect* cameras, depth also cannot be determined on surfaces that don't reflect reliably the infrared light emitted by the sensor. An example of this behaviour is shown in figure 3.5.

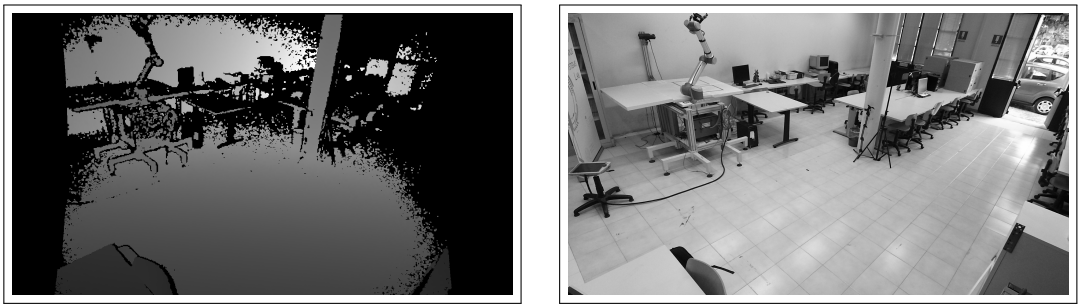


Figure 3.5: Example of Kinect image data
On the left a depth image obtained with a Kinect v2, on the right the corresponding registered conventional camera image.

The keypoints used for extracting the image features are mostly located on the edges and corners of objects, so it is quite frequent not to have the depth information for them. To solve this issue the system has to determine the depth by analyzing the area around the feature point.

Another issue is that, even if the depth information is available at the required position, it may not represent the correct depth of the feature. That is because if

the feature is located on the edge of an object, the depth that is needed is that of the foreground object, not of the background. Even a slight error in the keypoint positioning in the image could result in a large error in the associated depth.

To tackle both of the described issues, the system determines the depth for the feature points employing a custom approach. The procedure starts by searching the depth image for the non-zero pixel that is closest to the feature point. Then, it defines a circular ring that extends 10 pixels outward, starting from the non zero pixel it just found. Within this area, it then searches for the lowest-value non-zero pixel. The retrieved value will be used as the depth for the feature. This allows to first reach an area where the depth is correctly defined and then to search for a pixel representing the foreground object. A depiction of a possible search area for the depth a feature point is provided in figure 3.6.

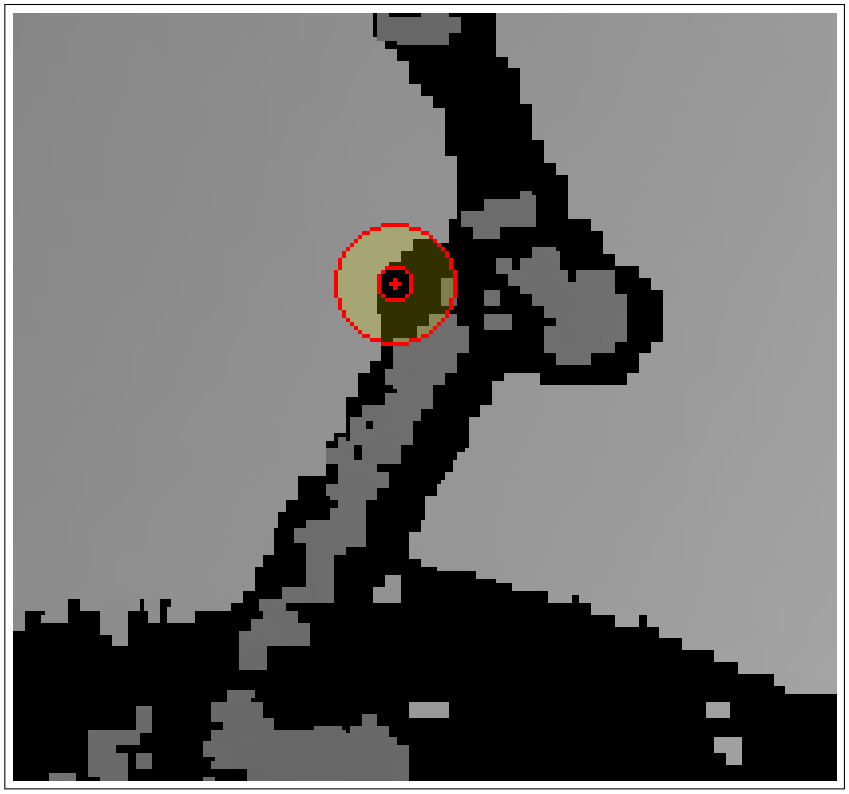


Figure 3.6: Feature point depth determination
Detail of the robotic manipulator in the depth image of figure 3.5, with a representation of the search area for the determination the depth of a feature point. The feature point is indicated by the red cross, and it is located in a zero-depth area. The depth selection algorithm would select the lowest non-zero pixel in the yellow ring-shaped area between the two red circles. This would effectively select the depth of the foreground object.

The limit of this approach is that if the depth of the foreground object can not be detected by the sensor, the algorithm will select the depth of the background. An example of this situation can be seen in figure 3.5, where the depth of the upper end of the robotic manipulator is not detected by the sensor. As the end effector is likely to be a feature point, this situation could result in the wrong estimation of the 3D pose of a point used for the *PnP* pose estimation. This could cause a wrong estimation of the *AR* device pose and affect the registration accuracy.

3.3.1.3 PnP Pose Estimation

By matching 2D positions of the features detected in the mobile camera image with the 3D positions of the features extracted from the fixed camera feed it is possible to estimate the pose of the smartphone using a *PnP* algorithm.

To perform this operation the proposed system makes use of the *solvePnPRansac* function provided by *OpenCV*. This function combines the *EPnP* algorithm described in section 2.5.1 with the RANSAC technique to perform a pose estimation that is robust, even in the presence of noisy measurements.

The pose estimation performed by *cv::solvePnPRansac* can still sometimes produce inaccurate results. This can be caused by the presence of false matches, by a wrong depth measurement or by an unfavorable spatial distribution of the matched points in the mobile camera image (as mentioned in section 2.5 the points should be well distributed in the image to obtain accurate results).

To mitigate the effects of these incorrect estimates a number of different heuristics have been employed.

- A minimum number of *RANSAC* inliers is imposed. The number of *RANSAC* inliers indicates the number of matches that confirm the current pose estimate. A minimum number of 4 points is required to obtain an estimate, by default the proposed algorithm requires at least 8 inliers.

- The mean re-projection error for the *RANSAC* inliers is computed. Estimates that exceed a maximum threshold are discarded. By default, this threshold is set to 2 pixels.

- Minimum and maximum thresholds are imposed on the height above the ground of the pose estimate. As the smartphone is assumed to be handheld its height from the ground can be bounded. Estimates indicating a height outside of these bounds are discarded. By default, the maximum height is set to 2.5 meters, the minimum to zero.

- Finally, the angle between the optical axes of the mobile and fixed cameras is computed. If the angle is greater than a specific threshold, the estimate is discarded. This because it is unlikely for estimates implying very different viewpoints between the two cameras to be correct. By default, the threshold is set to 45 degrees.

3.3.1.4 Configuration Parameters

All of the parameters previously mentioned are available to be tuned dynamically while the system is running, using the functionalities offered by the *ROS dynamic_reconfigure* [6] package. A view of the configuration window is presented in figure 3.7.

The most critical parameters are presented here:

- The `pnp_iterations`, `pnp_confidence`, `pnp_reprojection_error` parameters directly control the *PnP RANSAC* algorithm offered by *OpenCV* as `cv::solvePnPRansac()`.
- The `orb_max_points`, `orb_scale_factor`, and `orb_levels_number` parameters control the fixed camera features extraction performed by the *OpenCV* class `cv::ORB`.
- The `minimum_matches_number`, `reprojection_discard_threshold`, and `phone_orientation_diff_thresh` are the parameters are described in section 3.3.1.3.
- The `matching_threshold` parameter controls the matching of the *ORB* features as described in section 3.3.1.1.

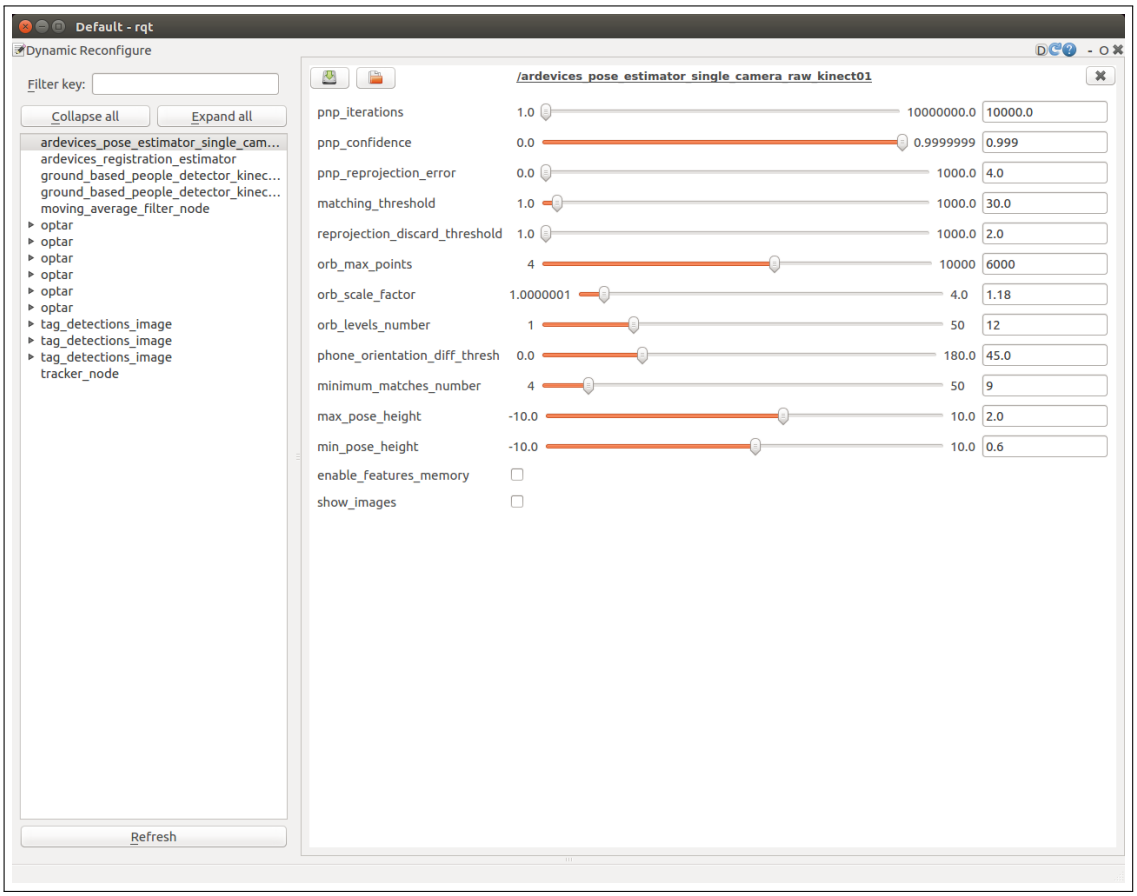


Figure 3.7: Single-camera pose estimation parameters
Tunable parameters for the ardevices_pose_estimator_single_camera_raw node as seen in the dynamic_reconfigure GUI.

3.3.2 Registration Estimator

The registration estimation for all the running AR devices is performed by the `ardevices_registration_estimator` node. It receives, from the single camera nodes, the *PnP*-based pose estimates, coupled with the corresponding *ARCore* poses. Already from this information, it is possible to compute the registration between the *ROS* and *ARCore* systems, but, to reduce the impact of the noise in the pose estimates, it is necessary to filter the received information.

A possible approach could be to filter the final registration estimate, which, in ideal conditions, should be constant. In practice the registration is not constant, it changes as the *ARCore* representation of the world evolves. As the movements of the origin of the *ARCore* coordinate frame are not bound by inertial constraints and do not follow a predictable trajectory, the estimation of its position results to be problematic. For this reason, the chosen approach is to apply a *Kalman Filter* to the smartphone pose.

An important issue with this approach is that the rate at which the *PnP* pose estimates are produced is too low and irregular to characterize the motion of the smartphone. For this reason, it is not possible to perform an effective filtering using only the information by the *PnP* estimation performed by the single camera nodes.

To solve this issue the *Kalman* filter is updated using the *ARCore* pose estimate, converted to the *ROS* frame using the last computed registration. Figure 3.8 outlines this filtering approach.

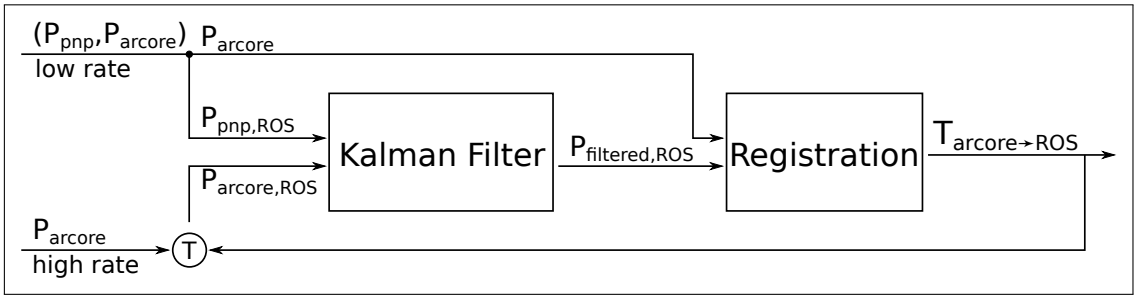


Figure 3.8: Pose Filtering
Representation of the filtering of the smartphone pose and the subsequent registration estimation

To handle the presence of multiple AR devices the node is structured similarly to the `ardevices_pose_estimator_single_camera_raw` node (described in section 3.3.1). The presence of AR devices is monitored via the `heartbeats` topic. If a

new device is detected the node constructs a handler object, which will listen to the topics specific to the device and update the pose and registration estimates. The handler is destroyed if no messages are received for a certain timeout time.

3.3.2.1 Kalman Filtering

To filter the smartphone pose the *Kalman Filter* has to handle both the position and the orientation of the device. This section will describe the details of the *Kalman Filter* that was implemented. More details on *Kalman Filtering* are provided in section 2.4.

Due to the difficulties involved with employing the *Kalman filter* with quaternions the orientation filtering has been performed using *Euler Angles*. In the case of *Euler Angles*, the orientation filter can maintain the same structure used for the position filtering. Indeed, the pose filter has not been implemented as a single *Kalman Filter* but as two simpler identical ones, one for the position and one for the orientation.

Following the approach described in 2.4.4 the state vector for each of the two filters has been defined as:

$$X_t = \begin{bmatrix} x_t & y_t & z_t & \dot{x}_t & \dot{y}_t & \dot{z}_t & \ddot{x}_t & \ddot{y}_t & \ddot{z}_t \end{bmatrix}^T \tag{3.3}$$

In the case of the position filter, x, y and z represent the position coordinates, In the case of the orientation filter, they instead represent the three components of the *Euler Angles* representation.

The transition matrix defined as:

$$A_t = \begin{pmatrix} 1 & 0 & 0 & \Delta t & 0 & 0 & \frac{\Delta t^2}{2} & 0 & 0 \\ 0 & 1 & 0 & 0 & \Delta t & 0 & 0 & \frac{\Delta t^2}{2} & 0 \\ 0 & 0 & 1 & 0 & 0 & \Delta t & 0 & 0 & \frac{\Delta t^2}{2} \\ 0 & 0 & 0 & 1 & 0 & 0 & \Delta t & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & \Delta t & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & \Delta t \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \tag{3.4}$$

Where Δt is the time since the last update. As Δt is not constant the matrix must be updated at each time step.

The measurement matrix is simply:

$$H = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \tag{3.5}$$

The process noise covariance matrix has been defined as:

$$Q_t = G_t G_t^T \sigma_p^2 \tag{3.6}$$

Where σ_p^2 is a tunable parameter and G_t is re-computed at each time step as:

$$G_t = \begin{pmatrix} \frac{\Delta t^2}{2} & 0 & 0 \\ \Delta t & 0 & 0 \\ 1 & 0 & 0 \\ 0 & \frac{\Delta t^2}{2} & 0 \\ 0 & \Delta t^2 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & \frac{\Delta t^2}{2} \\ 0 & 0 & \Delta t \\ 0 & 0 & 1 \end{pmatrix} \tag{3.7}$$

Finally, the measurement noise covariance matrix is defined as:

$$R = \begin{pmatrix} \sigma_m^2 & 0 & 0 \\ 0 & \sigma_m^2 & 0 \\ 0 & 0 & \sigma_m^2 \end{pmatrix} \tag{3.8}$$

With σ_m^2 also being a tunable parameter.

The filter has to accept measurements coming from two very different sources. Different both in their error distribution and in the rate of their output. To compensate for these differences the measurement covariance σ_m^2 parameter is kept different for the two sources. The covariance is kept considerably higher for the *ARCore* measurements, both to compensate for their higher rate and to give more weight to the *PnP* measurements.

The following default values were chosen for the position filter:

$$\begin{aligned} \sigma_p^2 &= 10^{-6} \\ \sigma_m^2 &= 10 \end{aligned} \tag{3.9}$$

While these were chosen for the orientation filter:

$$\begin{aligned} \sigma_p^2 &= 10^{-5} \\ \sigma_m^2 &= 10 \end{aligned} \tag{3.10}$$

The measurement covariances are kept different between the two sources by multiplying the σ_m^2 values by two different factors. By default, the *PnP* factor is set to 1, while the *ARCore* factor is 20. These parameters are available to be tuned during the system execution via the *ROS dynamic_reconfigure* package as shown in figure 3.9.

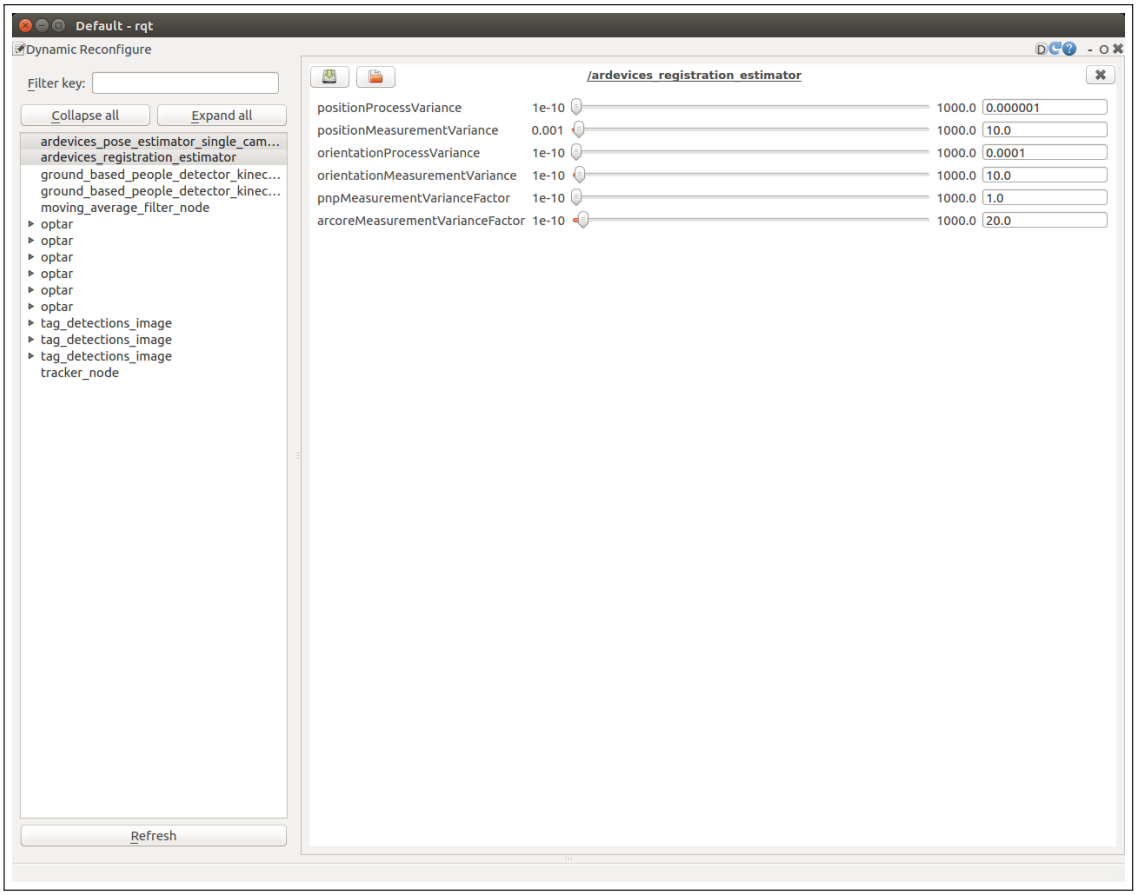


Figure 3.9: Registration estimation parameters
Tunable parameters for the ardevices_registration_estimator node as seen in the dynamic_reconfigure GUI.

Chapter 4

Experimental Evaluation

This chapter will present the results of the experiments that were performed to test the system performance.

Section 4.1 details the quantitative experiments that have been performed. First with a description of the test setup and then with an analysis of the collected data.

Section 4.2 describes a demo application that was developed to display the system capabilities.

4.1 Quantitative Analysis

The system has been tested on an *OpenPTrack* setup composed of two *Kinect v2* cameras, respectively connected to two desktop computers, one equipped with an *Intel i7-8700* CPU and an *NVIDIA GeForce GTX1070* graphics card, the other with an *Intel i7-7700* CPU and an *NVIDIA GeForce GTX1060* graphics card. A *Nokia 7 Plus* was used as the AR device.

The two *Kinect* cameras have been positioned about 4 meters apart, at about 2.3 meters from the ground, with their visuals directed at roughly 45 degrees between each other. Figure 4.1(a) depicts the test setup with the two *Kinect* cameras.

This setup has been deployed in the laboratory depicted in figure 4.1(b), an environment rich of visual features, but also containing some objects that are problematic for the depth measurement, as was described in section 3.3.1.2.

To monitor the system performance, the estimated smartphone pose in the *ROS* coordinate frame has been evaluated against a pose tracking produced using *AprilTag*[30] fiducial markers.

To do so, the smartphone was fixed to a rigid cardboard sheet having two 16x16 cm AprilTags on its two sides. The transformation between the *AprilTag* markers and the phone was manually measured.

The tracking of the *AprilTag* markers has been performed making use of the *apriltag_ros* package from *AprilRobotics*. Two tracking nodes have been used, one on each of the two computers, both the nodes tracked both the tags.

(a) (b)

Figure 4.1: The test setup
On the left an a photograph of the setup used for the testing, the two Kinect cameras have been highlighted in red. On the right the view of the Kinect that appears on the left in figure (a).

Three separate test runs have been performed using this setup, all three maintaining OPTAR's default parameter configuration. These three test runs were respectively 357, 624, and 813 seconds long, for a total duration of 1794 seconds, about half an hour. For each one of the tests, the tracking results have been saved to file, from these files the pose estimates produced by the two systems have been matched by their timestamps. As the rate of production of the *AprilTag* estimates is substantially lower compared to that of the estimates produced by *OPTAR* (about $5Hz$ for *AprilTag*, $30Hz$ for *OPTAR*), each *AprilTag* estimate has been matched to the temporally closest *OPTAR* estimate. A maximum time difference of $20ms$ has been imposed for a match to be accepted.

The error in the position estimate has been computed as the Euclidean distance of the *OPTAR* estimate from the ground truth provided by *AprilTag*. The error in the orientation estimate has instead been computed as the angle between the two orientations.

The metrics chosen to indicate the system performance are the mean error, which gives an intuitive representation of the accuracy, and the *Root Mean Square Error (RMSE)* which gives more weight to data points affected by a more significant error.

The system performance has been consistent among the different tests. Table 4.1 presents the aggregate results obtained by analyzing all of the collected tracking data. The mean position error that results from these tests is about half a meter, but it can be observed that this error can vary considerably during the operation of the system. The system is capable of obtaining a very accurate registration, but this is sometimes lost due to inaccurate estimates computed by the *PnP* algorithm or to movements of the *ARCore* coordinate frame.

This behaviour can be observed in figure 4.2, which shows the evolution of the position and orientation error in a section of one of the tests. The plot shows how the tracking error increases abruptly at $t = 151$ and then again at $t = 238$ to then return at a lower level between $t = 520$ and $t = 530$. Figure 4.3 shows the trajectory estimated by *OPTAR* and the one estimated with *AprilTag* in the final

testing time	1793s
data points	2833
position mean error	0.494m
position RMSE	0.690m
orientation mean error	0.05789rad
orientation RMSE	0.10649rad

Table 4.1: Aggregate results of the tests performed while moving the smartphone

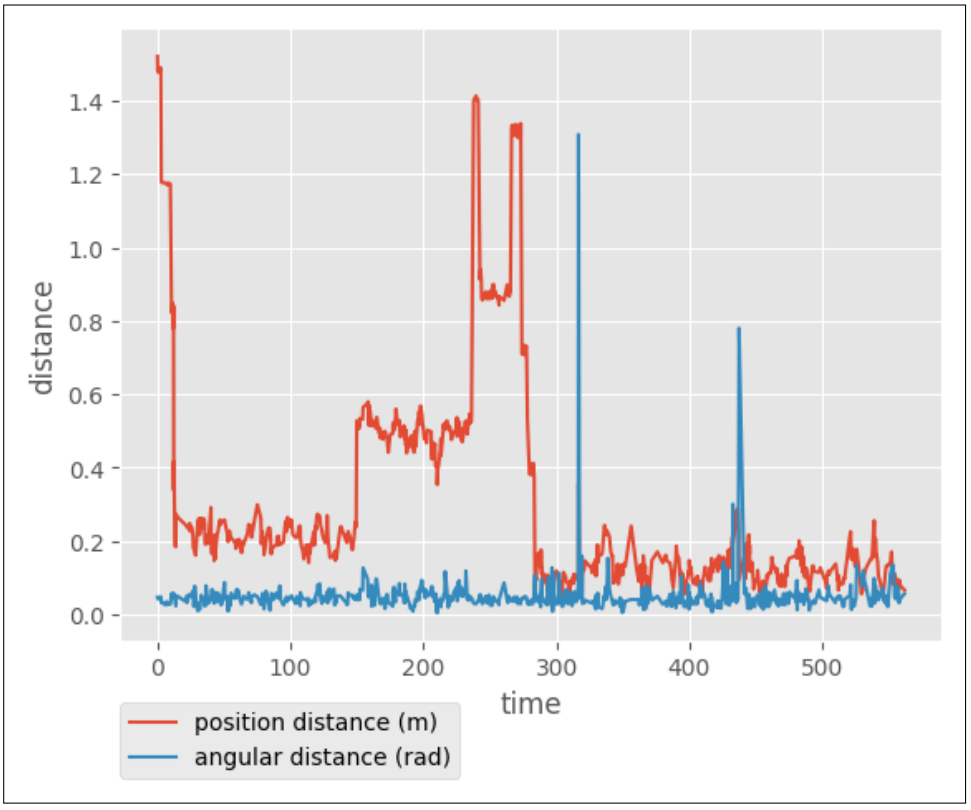


Figure 4.2: Pose estimation position and orientation error
Plot of the position and orientation error of the smartphone pose estimation, with the default parameter configuration, except for minimum and maximum height that were set respectively to 0.6m and 2.0m

part of figure 4.2 when the tracking has been accurate.

For what concerns the orientation, from table 4.1 and figure 4.2 it can be noted how, in this case, the tracking is particularly accurate and stable, except for the very narrow spikes in the error visible in the plot.

Finally, the system has also been tested while keeping the smartphone still. The results of this test can be seen in figure 4.4 and table 4.3.

As expected, the results are slightly better compared to those of table 4.2, in particular, the position estimate accuracy is considerably improved. Moreover, the sudden increases in the position estimation error seen in figure 4.2 are not present in these results.

This is an indication of how unfavorable viewpoints and the movements of the *ARCore* coordinate frame are the cause of the degradation of the system

performance. Indeed, in this static configuration, the understanding of the world of *Google ARCore* does not evolve, and the smartphone never encounters new, potentially unfavorable, viewpoints.

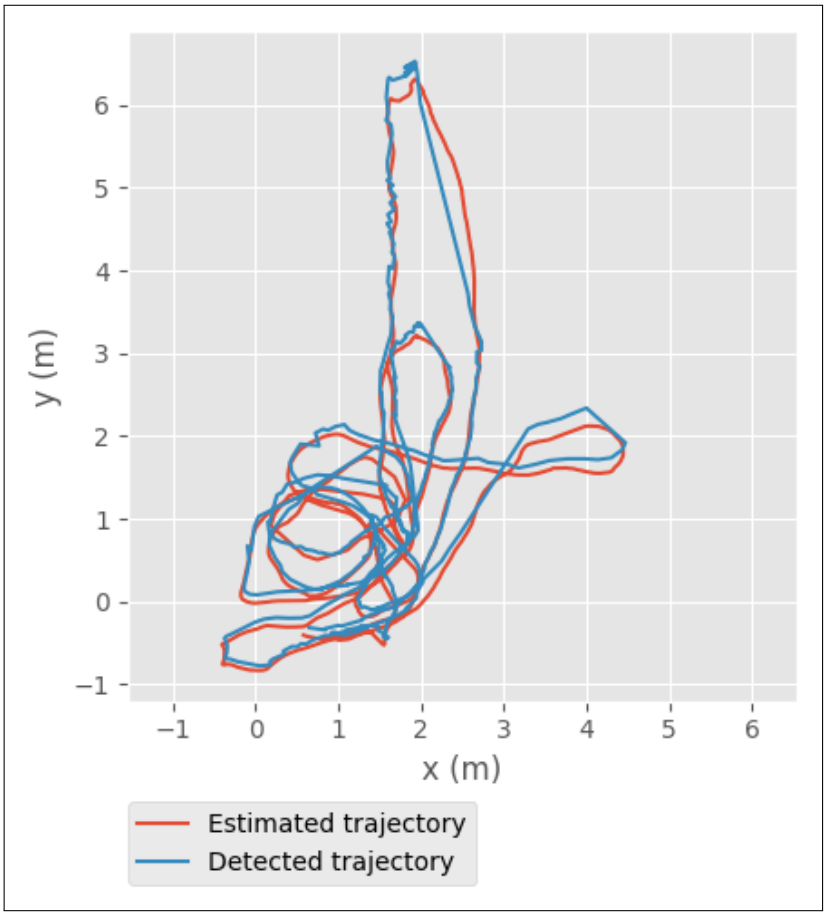


Figure 4.3: Estimated and detected trajectory
Plot of the trajectory followed by the AprilTag between second 350 and second 450 of figure 4.2. The plot depicts both the trajectory detected by apriltag_ros, and the one estimated via the proposed method.

testing time	561
data points	871
position mean error	0.318m
position RMSE	0.447m
orientation mean error	0.04802rad
orientation RMSE	0.07373rad

Table 4.2: *Results of the test in figure 4.2*

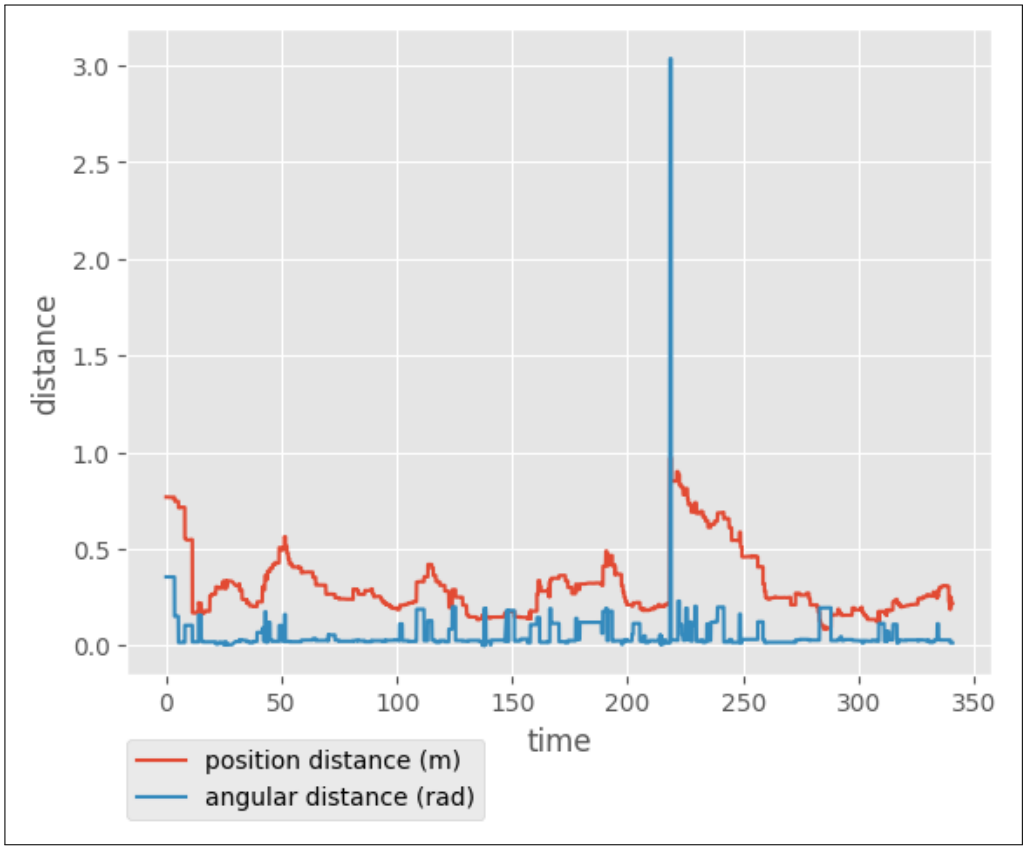


Figure 4.4: Position and orientation error in a stationary trial
In this experiment the smartphone was kept still, in a position in which it could get PnP pose estimates. The parameters were kept at their default values

testing time	340s
data points	9831
position mean error	0.314m
position RMSE	0.358m
orientation mean error	0.05633rad
orientation RMSE	0.14253rad

Table 4.3: Results of the test in figure 4.4

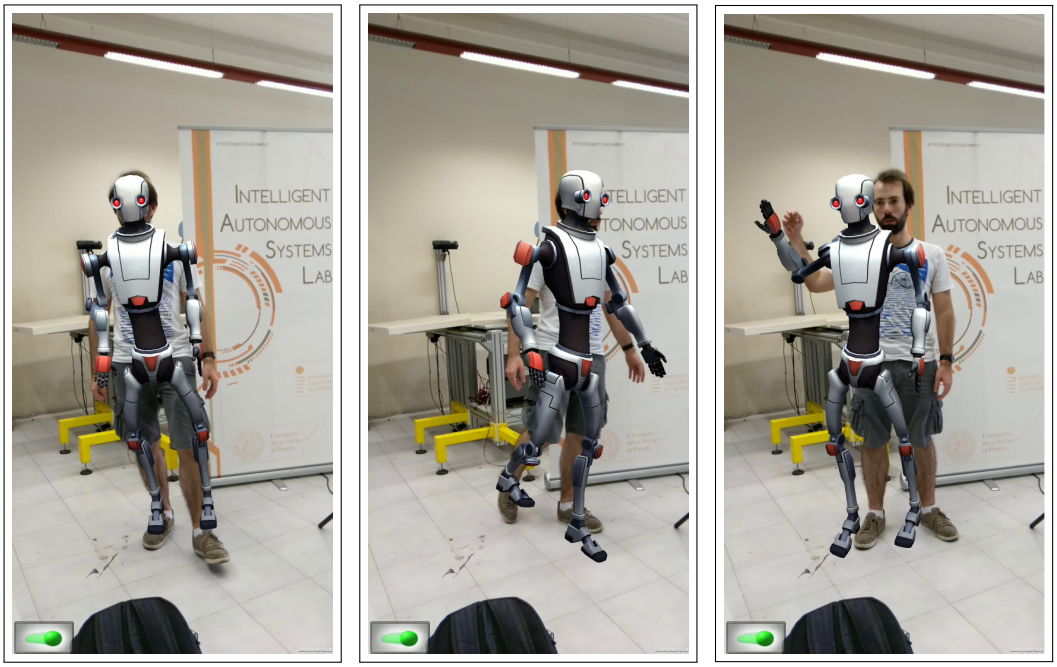
4.2 Qualitative Assessment

In parallel to the quantitative analysis detailed in the previous section, an Android demo application has been developed to observe the practical capabilities of the system.

The application uses the *Optar Unity* package to display the *OpenPTrack* centroid and skeleton tracking within *Google ARCore*.

The application was tested using the same configuration of section 4.1. From the experiments, it can be concluded that, even if the tracking of the smartphone position is relatively accurate, this by itself is often not enough to produce a convincing *Augmented Reality* experience.

The smartphone pose estimate has to be extremely accurate to obtain a convincing overlaying of the virtual objects on the real world. As shown in fig. 4.5(a) the superimposition can be accurate in certain moments, but even an offset of $10cm$ like in fig. 4.5(c) can completely disrupt the sense of realism.



(a) (b) (c)

Figure 4.5: Example of Skeleton tracking in AR
Three screenshots of the OpenPTrack skeleton tracking seen through the Unity ARCore smartphone application

However, what distinguishes this system from conventional AR applications is the capability to track entities that are outside of the handheld device visual. This tracking does not need to be as accurate as the one meant to provide the actual AR overlaying functionalities, the accuracy of the application is enough to provide this functionality satisfactorily.

Other than the registration accuracy, another factor that negatively impacts the performance of the application is the latency with which the tracking data is received. The observed latency is usually about one second, with peaks that can reach even ten seconds.

The cause for this seems to be a constant latency from the tracking performed by *OpenPTrack*, combined with communication issues in the *WebSocket* bridge between *ROS* and *Unity*. *WebSockets* communicate exclusively via *TCP* and the packets are text-based. This implies the packets have to be processed and converted to text and also that they are delivered in order. The result is that the communication is sometimes halted, supposedly to wait for lost packets. While this happens the messages are held in a queue and then delivered in rapid succession when the communication resumes. This behavior has been observed in practice.

From these observations, we can conclude that *OPTAR* is a useful tool to enhance AR systems, but an application cannot rely exclusively on it to generate the *Augmented Reality* rendering. Instead, an effective approach would be to combine the *OpenPTrack* data with detections obtained locally on the AR device. For example, the AR skeleton tracking could be improved using the *Augmented Faces* functionality provided by *Google ARCore* since version 1.7.0. By matching the face pose provided by *ARCore* with the skeleton tracking received from *OpenPTrack* it should be possible to obtain better results than those of this demo.

4.2.1 Multiple Smartphones

For what concerns the capabilities of the system in terms of the number of mobile devices used simultaneously, a separate test was performed with up to four smartphones.

The setup used two *Kinect v2* cameras and four *Galaxy S9+* smartphones.

In this test the *PnP* pose computation for the smartphones was performed on the computers directly connected to the fixed cameras. So each computer was running four *PnP* pose estimation nodes.

No noticeable degradation was noticed in the system performance. For what concerns the network, this is expected. As discussed in section 3.2.1 each smartphone requires a bandwidth of about 59KB/s. For four devices this amounts to roughly

240KB/s which is perfectly manageable by modern WiFi networks. Moreover, the computers were able to support the PnP computation and the fixed-cameras feature extraction without important slow-downs.

Chapter 5

Conclusions

This work introduced *OPTAR*, a system for integrating *Google ARCore Android Unity* applications with *OpenPTrack*. The system aims at estimating the registration between the two systems, which is the geometric transformation between their reference frames.

This estimation is performed by matching the AR device pose estimates provided by *Google ARCore* with pose estimates independently computed by OPTAR within the *OpenPTrack* coordinate frame. By knowing the device pose in both the reference frames it is possible to compute the registration.

The AR device pose estimate in the *OpenPTrack* reference frame is computed by exploiting visual features seen by both the smartphone and the fixed RGB-D cameras of the *OpenPTrack* system. From matching these features it is possible to compute the pose by solving a *PnP* problem.

The pose estimate for the smartphone produced by OPTAR frame has been filtered using a *Kalman Filter*, which, to operate reliably, fuses the pose measurements estimated via *PnP* with pose measurements produced by *Google ARCore*.

The system can support the operation of multiple AR devices simultaneously, which implies estimating a different registration for each device.

A demo *Android* application was developed to allow a qualitative evaluation of the system. The application uses the computed registration to visualize the skeleton tracking performed by *OpenPTrack* in Augmented Reality.

As discussed in section 4.1, a quantitative evaluation of the accuracy of the system has also been performed. The pose tracking for a single smartphone produced with the proposed system was compared with the tracking of the same pose obtained by using *AprilTag* fiducial markers attached to the smartphone.

From these tests, it can be concluded that *OPTAR* is capable of estimating the registration between the *OpenPTrack* and *Google ARCore* reference frames satis-

factorily, even if the pose estimate can, in certain situations, deviate considerably from the true value for short lengths of time.

It must be noted that the system accuracy depends heavily on the environment in which it is deployed. The best conditions for the system to work are well-textured environments, possibly with no objects that are not properly visible by the depth camera (see section 3.3.1.2). This because if a feature point is located on a foreground object of which the depth camera cannot measure the distance, the algorithm of section 3.3.1.2 may erroneously select the depth of the background in place of that of the object. This results in a wrong 3D position of the feature point and potentially in a wrong *PnP* estimate. Consequently, objects that are not visible by the depth cameras that are far from their background may have a high negative impact in the tracking accuracy, while if the object is close to its background the impact is potentially less significant.

An additional requirement is that the fixed cameras should not be too far from the scene and the handheld camera. The setup discussed in section 4.2 implies that the smartphone almost never gets farther than five meters from a fixed camera. The reasons for this constraint on the distance are that it is more difficult to determine matches among the features if their scale is too different and that the keypoint position accuracy is lower for more distant points.

Also, the *PnP* estimation cannot work if the angle between the fixed camera and the smartphone is too large. Because of this, the fixed cameras cannot be facing downward at a high angle.

Also the network performance is important. As discussed in section 4.2, the communication bridge between the smartphones and the *ROS* system is particularly sensitive to packet loss. This naturally becomes more and more critical as the network capacity gets lower and more smartphones are used simultaneously.

If these requirements are satisfied the system accuracy is sufficient for tracking the position of the smartphone, and, at the same time, for the smartphone to represent the tracking data from *OpenPTrack* in its reference frame. However, as shown in section 4.2 the accuracy is not enough to render the tracking data within the *Augmented Reality* view by overlaying. To obtain a convincing *AR* rendering it is necessary to integrate the information provided by *OpenPTrack* with local *AR* techniques operating directly on the camera feed.

Regarding the support for the usage of multiple smartphones simultaneously, the system has been proved to work with up to four devices at the same time.

The system was not tested with more than four devices, but, as described in section 3.2.1, in normal conditions each phone publishes a 60KB message per

second, requiring a bandwidth of 60KB/s. For this reason, it can be assumed that the system can support more than four devices simultaneously.

Eventually, increasing the number of devices, the network will saturate. Collisions between the packets will increase, leading to an overall degradation of the system performance. However, it will still be possible to operate the devices on separate WiFi channels, thus further increasing the number of devices usable at the same time.

The number of supported devices is also limited by the computational power of the computers performing the *PnP* pose estimation and the feature extraction for the fixed cameras. However, the system has been designed to be scalable, as the different *ROS* nodes of which it is composed can be executed on different computers.

5.1 Future Work

This section will present some proposals for the improvement of system performance and accuracy, and for reducing the reliance of the system on the requirements imposed on the setup and on the environment.

5.1.1 Feature Map

First of all, an enhancement to the system that may have a significant impact is the implementation of a "*Feature Map*". This would consist in using the features detected by the fixed cameras, and possibly also the handheld camera, to create a 3D map of the feature points. The *PnP* pose estimation of the smartphone pose would then be performed using the points from the 3D map instead of those extracted directly from the fixed camera feeds. A map like this one would resemble what is implemented in SLAM systems like, for example, ORB-SLAM [28].

This would offer various advantages with respect to the current system, improving its accuracy and reliability while, at the same time, reducing the requirements imposed on both the setup configuration and the environment.

The first of these advantages is that the matching of the mobile camera features would not be performed against one fixed camera at a time. Instead, it would be possible to compute a *PnP* pose estimate using features detected by different fixed cameras. As the number of points available for the *PnP* pose estimation is increased this would result in an increased system accuracy. As more 3D points are available it would also be possible to enforce more stringent requirements on the quality of the feature points, thus further increasing the system accuracy.

Moreover, by contributing to the map also the features detected by the AR device it could be possible to perform *PnP* pose estimates also from viewpoints that are very different from those of the fixed cameras. For example, every time a feature produced by the AR device is matched to a feature in the map, the descriptor from the AR device could be used to enrich that of the map's descriptor. This would progressively make the features represented in the map recognizable also from viewpoints that are farther and farther from those of the fixed cameras.

Also, as the map has been built, it would be available to any new AR device that is starting up. Because of this, the estimation of the pose of new devices would be both faster and easier, as the AR device viewpoint would not need to be similar to that of a fixed camera. It would also be possible to save the feature map built in a specific session to be used in future ones.

An issue that would need to be addressed would be how to differentiate static features from features located on moving objects. A first solution to this could

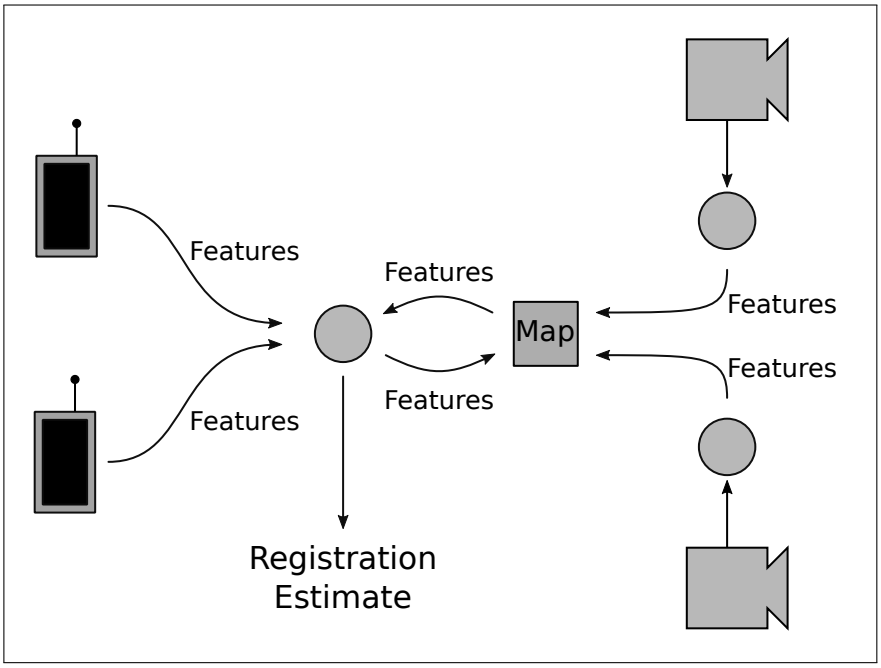


Figure 5.1: Proposed System Structure with Feature Map
High-level representation of the proposed system based on the feature map. The map receives directly the features produced by the fixed cameras on the right, which are integrated in the existing map. The features detected by the AR devices are first used to produce the pose estimates, then they are sent to be included in the feature map, with indications of the quality of the pose estimate they led to. The pose estimate for the AR devices is produced combining the features in the map with the features seen in the current view of the mobile device.

be to build the map while keeping the scene free of moving objects and then run the system disabling the updating of the map. A static map like this one could be complemented with features generated online as it is done in the current system implementation.

5.1.2 Pose Filtering Enhancements

Another component of the system that could be improved is the filtering of the AR devices' poses. Currently, the filtering is based on a linear *Kalman Filter* that uses *Euler Angles* to represent the orientation.

First of all, *Euler Angles* have well-known issues, in particular, singularity issues (gimbal lock). OPTAR does not suffer heavily from this, as the handheld devices poses rarely have high pitch angles, but the issue is still present.

Secondly, but not less importantly, the *Kalman Filter* assumes the measurement

noise to be Gaussian, and, from qualitative observations, this does not seem to be the case in this situation. In particular, the issue is that wildly wrong *PnP* estimates can move the filtered smartphone pose considerably far from its true value. While the impact of this problem is reduced by the heuristics described in section 3.3.1.3 it is not completely resolved. A possible approach to this problem could be using more sophisticated filtering methods, for example by following the approaches described by Masreliez [22] [23] or using different filtering techniques such as the *Particle Filter*.

5.1.3 Experimentation with Different RGB-D Cameras

The proposed system has only been tested with *Kinect v2* cameras as fixed sensors, but it has been designed to be device-agnostic. The only requirement is that the right *ROS* input topics are provided, delivering rectified greyscale camera images, rectified and registered depth images and the current camera intrinsic parameters.

This should allow testing the system with other RGB-D cameras, potentially reducing issues such as the missing depth information problem described in section 3.3.1.2. Using higher resolution cameras, would also improve the precision of the feature positioning and potentially allow the creation of setups in which the fixed cameras are positioned farther from the AR devices.

In particular, it would be fairly straightforward to test the system with stereo cameras such as *Zed* [36] and *RealSense* [8], which are supported by *OpenPTrack*.

5.1.4 Reducing ARCore Reference Frame Movement

Lastly, a component that could be improved is the *PoseManager* object within the *Unity* application, briefly mentioned in section 3.2.

By keeping track of multiple *Trackables* within *Google ARCore* it should be possible to further reduce the perceived movement of the *ARCore* reference frame.

Bibliography

[1] Openperception. URL: `http://www.openperception.org/`.

[2] Ucla center for research in engineering media and performance (remap). URL: `https://remap.ucla.edu/`.

[3] Adnan Ansar and Konstantinos Daniilidis. Linear pose estimation from points or lines. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 25(5):578–589, 2003.

[4] Yaakov Bar-Shalom, X Rong Li, and Thiagalingam Kirubarajan. *Estimation with applications to tracking and navigation: theory algorithms and software*. John Wiley & Sons, 2004.

[5] Filippo Basso, Riccardo Levorato, and Emanuele Menegatti. Online calibration for networks of cameras and depth sensors. In *OMNIVIS: The 12th Workshop on Non-classical Cameras, Camera Networks and Omnidirectional Vision-2014 IEEE International Conference on Robotics and Automation (ICRA 2014)*, 2014.

[6] Blaise Gassend, Michael Carroll. ROS dynamic_reconfigure package. URL: `https://wiki.ros.org/dynamic_reconfigure`.

[7] Zhe Cao, Gines Hidalgo, Tomas Simon, Shih-En Wei, and Yaser Sheikh. OpenPose: realtime multi-person 2D pose estimation using Part Affinity Fields. In *arXiv preprint arXiv:1812.08008*, 2018.

[8] © Intel Corporation. Realsense camera, 28-06-2019. URL: `https://www.intel.com/content/www/us/en/architecture-and-technology/realsense-overview.html`.

[9] Christopher Crick, Graylin Jay, Sarah Osentosiki, Benjamin Pitzer, Odest Chadwicke Jenkins, and Christopher Crick. Rosbridge: Ros for non-ros users. In *in Proceedings of the 15th International Symposium on Robotics Research (ISRR*, 2011.

[10] Dirk Thomas, Josh Faust, Vijay Pradeep. ROS message_filters package. URL: `https://wiki.ros.org/message_filters`.

[11] Ian Fette and Alexey Melnikov. The websocket protocol. Technical report, 2011.

[12] Sonja Gamse, Fereydoun Nobakht-Ersi, and Mohammad Sharifi. Statistical process control of a kalman filter model. *Sensors*, 14(10):18053–18074, 2014.

[13] Xiao-Shan Gao, Xiao-Rong Hou, Jianliang Tang, and Hang-Fei Cheng. Complete solution classification for the perspective-three-point problem. *IEEE transactions on pattern analysis and machine intelligence*, 25(8):930–943, 2003.

[14] Google. Google ARCore fundamental concepts. URL: `https://developers.google.com/ar/discover/concepts`.

[15] Google. Google ARCore supported devices. URL: `https://developers.google.com/ar/discover/supported-devices`.

[16] Mattia Guidolin, Marco Carraro, Stefano Ghidoni, and Emanuele Menegatti. A limb-based approach for body pose recognition using a predefined set of poses. 06 2018.

[17] Radu Horaud, Bernard Conio, Olivier Leboulleux, and Bernard Lacolle. An analytic solution for the perspective 4-point problem. In *Proceedings CVPR'89: IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pages 500–507. IEEE, 1989.

[18] HE Imre, Jean-Yves Guillemaut, and ADM Hilton. Moving camera registration for multiple camera setups in dynamic scenes. In *Proceedings of the 21st British Machine Vision Conference*, 2010.

[19] Vincent Lepetit, Francesc Moreno-Noguer, and Pascal Fua. Epnp: An accurate o (n) solution to the pnp problem. *International journal of computer vision*, 81(2):155, 2009.

[20] C-P Lu, Gregory D Hager, and Eric Mjolsness. Fast and globally convergent pose estimation from video images. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 22(6):610–622, 2000.

[21] João Luís Marins, Xiaoping Yun, Eric R Bachmann, Robert B McGhee, and Michael J Zyda. An extended kalman filter for quaternion-based orientation estimation using marg sensors. In *Proceedings 2001 IEEE/RSJ International Conference on Intelligent Robots and Systems. Expanding the Societal Role of*

Robotics in the the Next Millennium (Cat. No. 01CH37180), volume 4, pages 2003–2011. IEEE, 2001.

[22] C Masreliez. Approximate non-gaussian filtering with linear state and observation relations. *IEEE Transactions on Automatic Control*, 20(1):107–110, 1975.

[23] Cl Masreliez and R Martin. Robust bayesian estimation for the linear model and robustifying the kalman filter. *IEEE transactions on Automatic Control*, 22(3):361–371, 1977.

[24] David L Mills. Internet time synchronization: the network time protocol. *IEEE Transactions on communications*, 39(10):1482–1493, 1991.

[25] Faraz M Mirzaei and Stergios I Roumeliotis. A kalman filter-based algorithm for imu-camera calibration: Observability analysis and performance evaluation. *IEEE transactions on robotics*, 24(5):1143–1156, 2008.

[26] Anastasios I Mourikis and Stergios I Roumeliotis. A multi-state constraint kalman filter for vision-aided inertial navigation. In *Proceedings 2007 IEEE International Conference on Robotics and Automation*, pages 3565–3572. IEEE, 2007.

[27] Matteo Munaro, Alex Horn, Randy Illum, Jeff Burke, and Radu Bogdan Rusu. Openptrack: people tracking for heterogeneous networks of color-depth cameras. In *IAS-13 Workshop Proceedings: 1st Intl. Workshop on 3D Robot Perception with Point Cloud Library*, pages 235–247, 2014.

[28] Raul Mur-Artal, Jose Maria Martinez Montiel, and Juan D Tardos. Orb-slam: a versatile and accurate monocular slam system. *IEEE transactions on robotics*, 31(5):1147–1163, 2015.

[29] Niantic, Nintendo. Pokémon GO. URL: `https://www.pokemongo.com/`.

[30] Edwin Olson. Apriltag: A robust and flexible visual fiducial system. In *2011 IEEE International Conference on Robotics and Automation*, pages 3400–3407. IEEE, 2011.

[31] Paper Plane Tools. OpenCV plus Unity. URL: `https://assetstore.unity.com/packages/tools/integration/opencv-plus-unity-85928`.

[32] Long Quan and Zhongdan Lan. Linear n-point camera pose determination. *IEEE Transactions on pattern analysis and machine intelligence*, 21(8):774–780, 1999.

[33] Joseph Redmon and Ali Farhadi. Yolo9000: Better, faster, stronger. *arXiv preprint arXiv:1612.08242*, 2016.

[34] Ethan Rublee, Vincent Rabaud, Kurt Konolige, and Gary R Bradski. Orb: An efficient alternative to sift or surf. In *ICCV*, volume 11, page 2. Citeseer, 2011.

[35] Dan Simon. Kalman filtering. *Embedded systems programming*, 14(6):72–79, 2001.

[36] Stereolabs. Zed camera, 28-06-2019. URL: `https://www.stereolabs.com/`.

[37] Ivan E Sutherland. Three-dimensional data input by tablet. *Proceedings of the IEEE*, 62(4):453–461, 1974.

[38] Richard Szeliski. *Computer Vision: Algorithms and Applications*. Springer-Verlag, Berlin, Heidelberg, 1st edition, 2010.

[39] Bill Triggs. Camera pose and calibration from 4 or 5 known 3d points. In *Proceedings of the Seventh IEEE International Conference on Computer Vision*, volume 1, pages 278–284. IEEE, 1999.

[40] Greg Welch, Gary Bishop, et al. An introduction to the kalman filter. 1995.