Real-Time Camera Localization in Pre-Existing 3D Point Cloud Maps on Mobile Devices via Monocular SLAM and Continuous Registration

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Software and Information Engineering

by

Florian Miklautsch
Registration Number 12023974

to the Faculty of Informatics

at the TU Wien

Advisor: Univ.Prof. Dipl.-Ing. Dipl.-Ing. Dr.techn. Michael Wimmer
Assistance: Projektass. Mag.rer.soc.oec. Stephan Ohrhallinger, PhD

Vienna, December 31, 2025

Florian Miklautsch　　Michael Wimmer

Declaration of Authorship

Florian Miklautsch

I hereby declare that I have written this thesis independently, that I have completely specified the utilized sources and resources and that I have definitely marked all parts of the work – including tables, maps and figures – which belong to other works or to the internet, literally or extracted, by referencing the source as borrowed.
I further declare that I have used generative AI tools only as an aid, and that my own intellectual and creative efforts predominate in this work. In the appendix "Overview of Generative AI Tools Used" I have listed all generative AI tools that were used in the creation of this work, and indicated where in the work they were used. If whole passages of text were used without substantial changes, I have indicated the input (prompts) I formulated and the IT application used with its product name and version number/date.

Vienna, December 31, 2025

Florian Miklautsch

This thesis addresses the challenge of performing real-time camera localization within pre-existing 3D point cloud maps on mobile devices, while also visualizing them in augmented reality. By enabling the use of arbitrary external point clouds on standard smartphones, this work overcomes the limitations of current mobile AR frameworks. Furthermore, a novel two-stage pipeline is introduced to address the high computational requirements of state-of-the-art image-to-point cloud registration methods. This solution combines the established ORB-SLAM3 system for monocular camera tracking with a continuous background process that leverages existing algorithms from the Small-GICP library for rigid (SE(3)) registration against a reference map. To address the inherent scale ambiguity of monocular SLAM, this is extended with a custom similarity transformation (Sim(3)) estimation. The evaluation confirms that the system achieves real-time performance on modern smartphones. However, the final localization accuracy is highly dependent on the characteristics of the SLAM session, such as the environment at startup and the sensor trajectory, as well as the modality of the input data. Root Mean Square Errors (RMSE) ranged from 0.3m to 13m in ideal same-device scenarios, but increased to between 18m and 230m for challenging cross-modality registrations against LiDAR scans.

Contents

CHAPTER 1

Introduction

Point clouds have become a ubiquitous form of representing three-dimensional data in various applications, including computer vision, robotics, and augmented reality, spanning diverse fields such as cultural heritage preservation, urban planning, and gaming. Created through different techniques such as LiDAR scanning, photogrammetry, and depth sensing, these collections of coordinates provide accurate and useful digital representations of real-world environments and have become increasingly available for the likes of buildings and infrastructure. With this gain in popularity, there also arises a need to better utilize them in the field, like the fundamental challenge of accurately determining the position and orientation of a camera within one of these pre-existing maps. This capability is crucial for applications that require the overlay of digital information onto the physical world, or where precise spatial awareness is needed. A few examples include the positioning of autonomous hardware within a known environment, educational augmented reality experiences to overlay historical reconstructions, or contextual information onto real-world sites. Furthermore, indoor navigation systems can benefit from accurate camera localization to provide users with real-time directions and information about their surroundings.

In these scenarios, it is evident why traditional means of localization, like GPS or Wi-Fi triangulation, are not sufficient. This is due to their limited information and accuracy, especially in indoor or densely built-up areas, and an approach that leverages the rich spatial data contained within the 3D point clouds is needed. Approaches that utilize this form of data include feature-based methods that extract distinctive features from the camera images and match them to corresponding features in the point cloud. This process of image-to-point cloud registration, however, is often computationally intensive and is not feasible for real-time applications on resource-constrained devices. Alternatively, proprietary solutions and mobile AR frameworks exist, but they often suffer from being closed-source and expensive, limiting their accessibility. A significant limitation of these systems is their frequent restriction to point clouds generated within their own specific ecosystem, which precludes the use of arbitrary, pre-existing maps. Finally, simultaneous localization and mapping (SLAM) techniques can be employed to track the camera's position while building a map. While highly effective for tracking, standard implementations rely on internal mapping and do not inherently utilize pre-existing point clouds, inhibiting their use in scenarios where such data is already available.

This thesis aims to solve these problems by providing a solution that is able to perform real-time camera localization within pre-existing 3D point cloud maps on mobile devices. The focus lies on developing a system that can run efficiently on smartphones, leveraging their built-in cameras and sensors to achieve accurate localization and visualization in augmented reality (AR) applications and provide a more accessible solution than existing methods. This is achieved by building on top of established technologies in the fields of monocular SLAM and point cloud registration and combining them into a novel two-stage pipeline: First, the monocular SLAM system provides real-time camera tracking and a sparse 3D map of the environment. Second, a continuous registration step aligns this generated map to the pre-existing reference point cloud, allowing for localization within the known environment. This separation allows the SLAM system to run at the camera's frame rate, while the computationally heavier registration is executed at a lower, fixed interval. This approach is further extended with a custom similarity transformation (Sim(3)) estimation to address the scale ambiguity inherent in monocular SLAM systems.

The implementation uses the Flutter framework to provide ease of development for mobile platforms, while also using Dart's foreign function interface (FFI) to integrate native industry standard libraries, which also provide the necessary performance needed. Namely, ORB-SLAM3 is used for monocular SLAM, while the point cloud registration is performed using different iterative closest point (ICP) algorithms from the Small-GICP library. In order to visualize the point clouds in AR and provide rudimentary user feedback, a basic custom rendering engine that is also written natively in Dart is used.

The developed system is evaluated in various real-world scenarios, demonstrating its effectiveness and limitations. The results show that the application provides good responsiveness and usability on modern smartphones, while displaying mixed results that are very dependent on the environment and quality of the pre-existing point cloud when it comes to the localization accuracy. The findings highlight the challenges associated with monocular SLAM and point cloud registration, such as scale ambiguity and sensitivity to environmental conditions. Based on this, potential future improvements are discussed, including the integration of additional sensors, like inertial measurement units (IMUs), to enhance robustness and accuracy.

This thesis is structured to first provide a comprehensive overview of the state of the art, followed by the background and methodology. Afterwards, the system design and implementation details are presented. Finally, the evaluation results are discussed, followed by conclusions and suggestions for future research directions.

CHAPTER 2

State of the Art

This chapter provides a brief initial overview of existing approaches to the problem of localizing a camera within a pre-existing 3D point cloud map. Various proprietary solutions and image-to-point cloud registration techniques are discussed, highlighting their advantages and limitations in the context of real-time applications on mobile devices.

2.1 Platform-Specific Mobile AR Frameworks and Proprietary Solutions

When looking at mobile hardware, two major AR frameworks that allow for some form of point cloud visualization and localization capabilities, while providing ease of development, appear frequently. Apple's ARKit is one of them, providing a high-level tool for the company's platforms. Their `ARWorldMap` feature allows developers to create a scan of a location and save it for later use [Inc24]. This saved map can then be reloaded in future sessions to localize the device within this previously scanned environment, solving the formulated problem to some extent. The second framework is Google's ARCore, which offers similar functionality for Android devices called `Cloud Anchors` [Goo24]. Both of them lack, however, the ability to import and utilize pre-existing point clouds for localization, limiting their use cases to environments that have been previously scanned using the respective framework. When looking beyond both of these mobile operating system giants, there also exist third-party proprietary solutions for mobile devices. One example is Immersal, which provides a cloud-based AR platform that allows for localization within pre-scanned environments [Imm24]. Immersal supports the use of point clouds for localization, but being a proprietary solution, it also comes with limitations regarding accessibility and cost for research and development purposes.

2.2 Proprietary Hardware

Proprietary solutions also extend into hardware. Magic Leap, a company specializing in augmented reality headsets, offers devices that are equipped with advanced sensors and cameras capable of capturing detailed 3D maps of the environment. Their headsets implement `Spaces` that let the HMD create a scan of the surrounding area, which can then be used for localization down the line [Mag24]. Microsoft HoloLens 2 and others also implement similar solutions [Mic23]. However, like the previously mentioned software solutions, these hardware-based approaches are often expensive and closed-source, limiting their accessibility for broader research and development efforts [VRC24].

2.3 Image-to-Point Cloud Registration

The most apparent scientific approach to localize a camera within a pre-existing point cloud is to directly register the current camera image to the map. This process, known as image-to-point cloud registration, involves extracting features from the 2D image and finding corresponding points in the 3D point cloud. Once these correspondences are established, algorithms such as Perspective-n-Point (PnP) can be used to estimate the camera's 6-DoF pose. Several research works have explored this approach, utilizing various feature detection and matching techniques to improve accuracy and robustness. One notable example of this is DeepI2P, which uses frustum classification [LL21]. Here, a neural network is trained to predict whether points are inside or outside the camera frustum, allowing for pose estimation using RANSAC on EPnP. Corrl2P is another solution for this challenge, which establishes 2D-3D correspondences between image features and point cloud points using a shared feature space to find shared regions [RZHC22]. The final pose is then calculated by using standard geometric solvers (like EPnP). CoFiI2P is one of the most performant methods for this task, using a coarse-to-fine strategy to first estimate a rough pose and then refine it using iterative optimization techniques [KLL$^+$23]. Even though these methods show promising results and achieve state-of-the-art accuracy, they are not fit for this exact issue. They have the ability to perform in real time, but only achieve this on high-end desktop hardware with powerful GPUs. For example, CoFiI2P specifies an NVIDIA RTX 4090 GPU and Intel Core i9-13900K CPU were used for their benchmarks in order to reach around 15FPS, which is far beyond the capabilities of current smartphones.

CHAPTER 3

Background and Methodology

This chapter details the fundamental concepts and techniques that form the basis of the proposed method. It provides an in-depth analysis of simultaneous localization and mapping (SLAM), specifically focusing on the monocular visual variant, and point cloud registration algorithms. As these technologies are discussed, the rationale for their selection is presented, while also outlining how they are integrated into the overall system and what modifications were necessary to adapt them to the specific requirements of this work. The chapter further summarizes this methodology, demonstrating how the individual parts are combined into a unified pipeline, and concludes with the rationale for the chosen mobile development framework.

3.1 Visual SLAM for Real-Time Camera Tracking

Since all of the solutions for the formulated problem discussed in the previous chapter have various limitations, a different approach is needed. SLAM is a well-researched field that does not specifically solve the problem at hand, but can provide a good basis for a solution. The following sections provide an overview of the SLAM problem, its visual variant, and the specific reasons for choosing monocular SLAM for this work. Finally, the selected ORB-SLAM3 system is presented in detail.

3.1.1 The SLAM Problem

Simultaneous Localization and Mapping (SLAM) solves the problem of estimating a sensor's trajectory and pose while simultaneously reconstructing the surrounding environment [DWB06]. It is a fundamental problem in robotics and computer vision and has been extensively studied over the past few decades. The main challenge of these systems lies in the fact that both of these tasks are interdependent and rely on the accuracy of the other. This circular dependency is resolved using iterative approaches, where both the map and the trajectory are refined over time as more data becomes available, in order to minimize the accumulated reprojection errors [CCC$^+$16].

Visual SLAM is a variant that relies on visual data from cameras to perform localization and mapping. It takes a sequence of timestamped images as input and calculates the 6-DoF pose and improved map for each frame based on visual features extracted from the images [SF11]. After key points are determined for a frame, correspondences are established with previously processed data, allowing for the estimation of three-dimensional positions. To decrease the error accumulation over time, loop closures (revisiting previously seen locations) are detected, and methods like bundle adjustment (a nonlinear least-squares optimization technique) are used [TMHF99].

3.1.2 Visual SLAM Variants

Visual SLAM systems differ in their respective sensor configuration used and how this data is being processed [SF11]. Different sensor setups include monocular SLAM, which utilizes a single camera, but therefore suffers from scale ambiguity and needs motion to function properly [MAMT15]; stereo SLAM, which is similar but employs a more advanced visual system consisting of two cameras to calculate depth values directly [ESC15]; and RGB-D SLAM, which uses measurement devices such as time-of-flight sensors in addition to the visual data to provide direct depth measurements, simplifying the mapping process and improving accuracy [EHS$^+$14]. Furthermore, visual-inertial SLAM systems combine visual data with inertial measurements from IMUs to enhance robustness and accuracy, especially in dynamic or fast-paced scenarios [LLB$^+$15].

Processing methods can be categorized into feature-based and direct methods. Feature-based methods such as ORB-SLAM rely on detecting and matching distinctive features across frames to estimate motion and build the map [MAMT15]. They are generally more robust to changes in lighting and texture, but can struggle in feature-poor environments. Performance is often better, as only a subset of the image data is used for processing. Direct methods such as LSD-SLAM, on the other hand, optimize the camera pose by minimizing the photometric error across all pixels, allowing them to utilize more information from the images. Because of this, they can leverage more information each frame, but suffer from motion blur issues and require higher frame rate and computational power [ESC14].

3.1.3 Why Monocular SLAM for This Application?

The requirements of mobile deployment with real-time performance and acceptable device compatibility already dictate the choice of system that can be used for this work as a base for camera tracking. Monocular SLAM is the best option for various reasons, with the biggest one being the availability of a standard RGB camera on virtually all modern smartphones. Stereo SLAM requires synchronized dual cameras, which are only present on higher-end devices. Here, an additional problem arises, as the physical distance between the cameras (the baseline) is fixed and often quite small, limiting depth accuracy and range [FHASAM16]. Also, most mobile devices that have multiple cameras mostly use them for zoom capabilities, meaning that the lenses have different focal lengths, complicating stereo matching [DXO21]. RGB-D SLAM demands depth sensors, which are also exclusive to high-end devices, and even then, not widespread. Visual-inertial SLAM could be a viable option, as almost all smartphones have built-in IMUs. However, this would add additional complexity to the system and could always be added later, if needed.

Monocular SLAM introduces two fundamental challenges. The already mentioned scale ambiguity means that the absolute scale of the environment cannot be determined from monocular images alone [SF11]. The trajectory and constructed map are still correct, but can be arbitrarily scaled. A camera moving one meter may be indistinguishable from one moving two meters, if there are no other points of reference. The second challenge is the need for motion to initialize and maintain tracking. Monocular SLAM systems require parallax to triangulate depth and build a 3D map [MAMT15] [CDM08]. If the camera remains static or moves too slowly, tracking can be lost. This, of course, would not be the case with other SLAM variants, which provide direct depth measurements. Both of these issues are, in theory, mostly mitigated for the use case presented in this work through the continuous registration step, which is discussed in Section 3.2. In a nutshell, registering the generated map to a pre-existing point cloud, with known metrics, should provide the necessary scale information. This is discussed further in Section 3.2.5. Also, when the tracking is lost due to the lack of motion, the registration will automatically jump in once enough parallax is present again. One issue that remains is that the user experience might suffer from these jumps in tracking, as the camera pose will suddenly change when re-localization occurs. This, however, is an acceptable trade-off, given the advantages of monocular SLAM for this application.

Feature-based monocular SLAM further suits this work, as it provides a good balance between performance and robustness. The generated maps of this method have the characteristic that they consist of sparse three-dimensional points, which would make the registration process less computationally demanding [MAT17].

3.1.4 ORB-SLAM3 System Overview

Based on the previously discussed criteria and other practical considerations, ORB-SLAM3 was selected as the feature-based monocular SLAM system for this work [CER+21]. As an evolution of the well-known ORB-SLAM2 with thousands of citations, it introduces several improvements and new features that enhance its usability [MAT17].

The ORB-SLAM3 system architecture consists of three concurrent threads, handling tracking, local mapping, and loop closing, respectively. The tracking thread first extracts ORB (Oriented FAST and Rotated BRIEF) feature descriptors from the incoming camera frames, which provide a good balance between computational efficiency and robustness to changes in scale and rotation [RRKB11].

It then establishes correspondences with the local map points to estimate the camera pose through solving the Perspective-n-Point problem using RANSAC and motion-only bundle adjustment for refinement. If tracking is successful, the local mapping thread is responsible for adding new key frames and map points to the map, as well as performing local bundle adjustment to refine these poses. The map structure is further optimized by selecting new key frames if the current frame has moved sufficiently far from existing key frames. If tracking is lost at any point, the Atlas system provides multimap features, like initializing a new map or re-localizing in the old one.

The loop closing thread detects loop closures by recognizing previously visited locations using a bag-of-words approach and performs pose graph optimization to correct drift in the trajectory and improve the overall map consistency.

For this work, two specific outputs are needed from ORB-SLAM3. First, the real-time camera poses as a 4×4 homogeneous transformation matrix $T_{\text{cam}}^{\text{SLAM}} \in SE(3)$ representing the SLAM coordinate system to the camera coordinate system in the OpenCV convention (x-right, y-down, z-forward). Second, the sparse 3D map points $\{p_i^{\text{SLAM}}\}$ generated by the SLAM system in the same world frame, which are periodically requested to perform the continuous registration step described in Section 3.2.

3.2 Point Cloud Registration for Coordinate Alignment

The map, which is generated by the monocular SLAM system, exists in its own arbitrary coordinate system, which does not correspond to any real-world reference frame. In order to localize the camera within a pre-existing point cloud map, a transformation between both coordinate systems must be established. This is achieved through another fundamental computer vision problem known as point cloud registration. The following sections provide an overview of this problem, its necessity for this work, and the specific algorithms used to solve it.

3.2.1 The Registration Problem

Point cloud registration is the process of calculating the transformation that aligns two sets of three-dimensional points. For rigid registration, this transformation consists of a rotation $\mathbf{R} \in SO(3)$ and a translation $\mathbf{t} \in \mathbb{R}^3$, that minimize the distance between corresponding points in the two point clouds and, which can be represented as a homogeneous transformation matrix $T \in SE(3)$ [Ume91, BM92]. When point clouds additionally differ in scale, however, the problem becomes one of similarity registration $Sim(3)$, where a uniform scaling factor $s \in \mathbb{R}^+$ is also estimated alongside rotation and translation. Here, all of these parameters have to be considered when minimizing the distance between corresponding points. This increases the complexity of the problem and reduces accuracy and robustness, as an additional degree of freedom is introduced.

In this work, the challenge of registering two point clouds distinguishes itself from standard registration tasks to some extent. Usually, both point clouds stem from similar sources, like two consecutive LiDAR scans or photogrammetric reconstruction, where they share similar densities and distributions of points. Here, however, one point cloud is always generated by the monocular SLAM system (sparse), while the other can be any arbitrary pre-existing point cloud (potentially dense). The asymmetric nature of this registration problem requires special consideration when selecting appropriate algorithms and techniques, and might limit the achievable accuracy. This also further motivates the continuous registration approach, with periodic map updates, as a single registration might not be sufficient to achieve the desired accuracy. The monocular approach also means that scale ambiguity has to be considered, as discussed in Section 3.2.5.

3.2.2 Why Registration is Necessary

Visual SLAM establishes a local coordinate system that is arbitrary in terms of scale, orientation, and position at initialization [SF11]. This means that the camera pose $T_{\text{cam}}^{\text{SLAM}}$ and map points $\{p_i^{\text{SLAM}}\}$ generated by the SLAM system are defined relative to this local frame, which does not correspond to any real-world metric or orientation. ORB-SLAM3 sets its world origin based on the initial camera pose [CER$^+$21]. This remains consistent throughout the session, but does not provide any absolute reference.

The pre-existing point cloud map, which acts as the reference, exists in its own coordinate system. This reference frame is entirely independent of the one created by the SLAM system. In some scenarios, this reference map might be a metrically accurate scan of the real world, for instance, from a LiDAR scanner or a photogrammetry pipeline. In other cases, it could be a map generated during a previous SLAM session, which would also have its own arbitrary origin (and possibly scale). Regardless of the source, to align the camera view with the reference map, a transformation $T_{\text{SLAM}}^{\text{ref}} \in \text{Sim}(3)$ (or $SE(3)$ if the scales are identical) between the two coordinate systems must be established. With this, the camera pose output by the SLAM system can be transformed into the point cloud's reference frame through $T_{\text{cam}}^{\text{ref}} = T_{\text{SLAM}}^{\text{ref}} \cdot T_{\text{cam}}^{\text{SLAM}}$. This transformation enables the use of accurate AR by placing the camera and the reference point cloud into a unified coordinate system, allowing for a correct projection.

3.2.3 Iterative Closest Point (ICP) Algorithm

The Iterative Closest Point (ICP) algorithm is a widely used method for rigid point cloud registration [BM92]. The algorithm works by iteratively refining an initial estimate of the transformation between the source and target point clouds. There are four main steps that are iteratively repeated until convergence is reached:

1. Correspondence Estimation: For each point in the source point cloud $p_i \in P_{\text{source}}$, the closest point in the target point cloud $q_{\text{nn}(i)} \in P_{\text{target}}$ (nearest neighbor) is found, establishing correspondences between the two sets of points.

2. Transformation Estimation: Using the established correspondences, the optimal rigid transformation $(\mathbf{R}, \mathbf{t})$ that minimizes the distance between the corresponding

points is computed using Singular Value Decomposition (SVD) or other approaches [AHB87].

3. Apply Transformation: The computed transformation is applied to the source point cloud.

4. Convergence Check: The error is checked to see if it is below a predefined threshold or if a maximum number of iterations has been reached. If not, it returns to step 1.

The performance of this algorithm depends strongly on the nearest-neighbor calculation method used. Basic implementations that rely on brute-force have a time complexity of $O(N \cdot M)$, where N and M are the number of points in the source and target point clouds, respectively. Using more advanced approaches like KD-trees can reduce the time complexity to $O(N \log M)$, significantly speeding up the correspondence estimation step [RL01]. Another important characteristic of ICP is its strong convergence to local minima [BM92, RL01]. This means that the error always decreases, but it can get stuck in suboptimal solutions. Lastly, the algorithm is also sensitive to outliers and noise in the data, which can lead to incorrect correspondences and poor registration results.

3.2.4 GICP, VGICP and PLANE ICP Improvements

Generalized ICP (GICP) is an extension of the standard ICP algorithm with improved robustness and accuracy [SHT09]. In simple terms, instead of looking at each point individually, GICP considers the local surface structure around each point by estimating local covariance matrices. This allows GICP to better handle noise and outliers, as well as improve convergence properties. The algorithm modifies the error metric to account for the local surface geometry, leading to more accurate registration results, especially in cases where the point clouds have different densities or distributions.

Voxelized GICP (VGICP) further builds upon GICP by introducing a voxel grid structure to partition the point clouds into smaller, manageable regions [KOY21]. This voxelization allows for more efficient nearest-neighbor searches and reduces computational complexity. VGICP can therefore achieve faster convergence and improved performance, particularly for large point clouds. The voxel grid also helps in reducing the impact of noise and outliers by averaging points within each voxel, leading to more stable registration results.

Plane ICP (or Plane-to-Plane ICP) is another variant of the standard ICP algorithm that improves convergence and accuracy by modifying the error metric [Low04]. Unlike the original Point-to-Point ICP, which minimized the distance between corresponding points, Plane ICP minimizes the distance between each source point and the tangent plane defined by its corresponding target point. This is achieved by estimating surface normals for each of the target points and then minimizing the dot product between these normals and the point-to-point vectors.

To integrate these algorithms into this work, the Small-GICP library is used, which provides all the discussed ICP variants, while also being optimized for multithreaded CPU-oriented performance and ease of use [Ish23]. Benchmarks show that the implementation is even faster than the one provided in the popular Point Cloud Library (PCL), while also offering the flexibility to choose between different ICP variants based on the specific requirements of the registration task at hand.

3.2.5 The Scaling Problem

ICP-based registration algorithms provide a stable and robust solution when both point clouds are in the same scale. This can be the case when the same SLAM system and sensor configuration that are now used for localization have been used to create the pre-existing point cloud map. However, since monocular SLAM suffers from scale ambiguity and the pre-scanned map might also have an arbitrary scale, this cannot be assumed. To therefore make the approach presented in this work more generalizable, the possibility of scale differences between the two point clouds has to be considered. This means, that instead of estimating a rigid transformation $T^{\text{ref}}_{\text{SLAM}} \in SE(3)$, a similarity transformation $T^{\text{ref}}_{\text{SLAM}} \in Sim(3)$ also including a uniform scaling factor has to be calculated increasing the DoF from 6 to 7. Further increases of complexity, like anisotropic scaling or affine transformations, are not considered, as they lead to even more unstable results and are not strictly necessary for this application. Another characteristic of this work, which aids in solving the scale problem, is visual feedback to the user, which allows for manual adjustments of the scale if the automatic estimation fails.

In this work, there are two make-shifted approaches to automatically try to solve this problem, which both rely on a hierarchical coarse-to-fine multiscale pyramid of different voxel resolutions. This is aimed at improving robustness by first looking at more global structures and then refining the results at finer levels. They also share the approach of taking an initial scale estimate of the user into account, and when not provided, using a bounding box-based heuristic to determine a rough initial scale. This is done by calculating the diagonal lengths of the bounding boxes of both point clouds and using their ratio as an initial scale estimate: $s_0 = |\mathbf{d}\text{target}|/|\mathbf{d}\text{source}|$, where $\mathbf{d} = \mathbf{p}\text{max} - \mathbf{p}\text{min}$ represents the bounding box diagonal vector. Furthermore, the coarse estimate is clamped to avoid extreme scale values.

The first algorithm builds on top of Small-GICP's registration algorithms [SHT09]. This method decouples the correspondence calculations and scale estimations by passing the current scale estimate to the rigid ICP variant at each level. This is a high-level overview of the proposed algorithm:

1. Estimate initial clamped scale s_0 using bounding box heuristic or user input.
2. For each level in the multiscale pyramid (from coarse to fine) l with voxel resolution v_l:
    a) Downsample both point clouds P_{source} and P_{target} to voxel resolution v_l. This merges points within each voxel, reducing noise, computational load, and

enhancing the bigger structures. The resulting point clouds are $P_{\text{source},l}$ and $P_{\text{target},l}$. Also build KD-trees for fast nearest-neighbor searches later on.

b) Build a transformation matrix T_l with the current scale estimate s_{l-1} integrated.

$$T_l = \begin{bmatrix} \mathbf{R} \cdot s_{l-1} & \mathbf{t} \\ \mathbf{0} & 1 \end{bmatrix} \tag{3.1}$$

c) This initial transformation guess T_l is then passed to the Small-GICP library with the selected ICP variant to perform rigid registration on the current point clouds at level l at the scale s_{l-1}.

d) The resulting refined transformation T'_l now delivers the optimal (for the settings and approach) rotation $\mathbf{R}'$ and translation $\mathbf{t}'$ at the estimated scale s_{l-1}.

e) Now the voxelized source point cloud $_{\text{source},l}$ is considered again. Correspondences are established between this and the target point cloud by now transforming every point p_i using the refined transformation T'_l and then finding the nearest neighbor $q_{\text{nn}(i)}$ in the target point cloud.

f) Now, utilizing these correspondences, the Umeyama method [Ume91] is used to estimate a new similarity transformation $T_{\text{sim},l}$ between the two point clouds. If the resulting scale s_l is within acceptable bounds and everything else looks good (number of matches), the current scale estimate is updated to this new value. If not, the previous scale estimate is retained.

3. After all levels have been processed, one last refinement step is performed using Small-GICP at the finest voxel resolution with the final scale estimate s_L to obtain the final transformation $T^{\text{ref}}_{\text{SLAM}}$.

This hybrid approach intends to leverage both the robust registration capabilities of Small-GICP and the explicit scale estimation of the Umeyama method to achieve accurate similarity registration. In future sections, it will be referred to as the *Sim*(3) *External* algorithm, since the scale estimation is performed outside of the ICP loop.

The second algorithm is simpler and does not use Small-GICP's registration methods. Instead, it implements a custom ICP loop based on Open3D [ZPK18], where scale is directly estimated using Umeyama's least-squares similarity transformation within a multiscale framework [PZK17] [Ume91]. The high-level steps are as follows:

1. Estimate initial scale clamped s_0 using bounding box heuristic or user input.

2. For each level in the multiscale pyramid (from coarse to fine) l with voxel resolution v_l:

   a) Run an ICP loop until convergence or maximum iterations reached:

i. Apply current transformation estimate T_l^{iter} to the source point cloud $P_{\text{source},l}$. This is then used to establish correspondences with the target point cloud $P_{\text{target},l}$ by finding nearest neighbors. If the number of correspondences is too low, the process is aborted.
ii. Using the established correspondences, the Umeyama method is employed to estimate a new optimal similarity transformation $T_{\text{sim},l}^{\text{iter}}$ that includes rotation, translation, and scale. If the resulting scale s_l^{iter} is not within acceptable bounds, the previous scale estimate is retained.
iii. Convergence is checked by looking at the change in scale and transformation. If below a certain threshold or oscillating, the ICP loop is exited.

This approach directly integrates scale estimation into the ICP loop, allowing for simultaneous refinement of rotation, translation, and scale at each iteration. It is expected to be more straightforward but potentially less robust than the previous method, as it does not leverage the advanced registration techniques of Small-GICP. In future sections, it will be referred to as the *Sim*(3) *Internal* algorithm, since the scale estimation is performed within the ICP loop.

How well these two approaches perform in practice is evaluated in Chapter 6.

3.3 Coordinate Systems and Transformations

In order to combine the different components of the proposed system, it is important to clearly define the various coordinate systems involved and how they relate to each other. The following coordinate systems are used throughout this work:

- **SLAM Coordinate System:** The local coordinate system established by the monocular SLAM system (ORB-SLAM3) on initialization, but stays consistent throughout the session. The camera pose $T_{\text{cam}}^{\text{SLAM}}$ and map points $\{p_i^{\text{SLAM}}\}$ are defined in this frame.
- **Pre-existing System:** The presumed real-world coordinate system of the pre-existing point cloud map, with reference scale and orientation. $\{p_i^{\text{ref}}\}$ is defined here.
- **Camera Coordinate System:** The coordinate system of the camera, defined by its intrinsic parameters and image plane. The camera pose $T_{\text{cam}}^{\text{ref}}$ is eventually needed in this frame for AR rendering.

3.4 Methodology Overview

Based on the theoretical foundations and algorithmic components analyzed in the previous sections, the methodology for this work is summarized as a two-stage pipeline consisting of real-time monocular SLAM, followed by a continuous point cloud registration step. This effectively combines a high-frequency, locally consistent tracking method with a global alignment method that anchors the trajectory in a pre-existing map, thus achieving real-time 6-DoF localization within an arbitrary point cloud.

3.4.1 Method at a Glance

The proposed method employs a continuous loop that couples tracking and alignment by executing both stages repeatedly, however, at the respective frequencies.

First, the monocular SLAM system estimates the camera pose $T_{\text{cam}}^{\text{SLAM}}$ ideally at the frame rate of the camera, while also incrementally building a sparse 3D map $\{p_i^{\text{SLAM}}\}$ of the environment in its own local coordinate system. This ensures that the system remains responsive and can handle fast camera motions, while also being robust to temporary tracking failures through relocalization capabilities.

The second stage is the point cloud registration step, which operates at a lower frequency in a fixed interval. Here, the current sparse map points $\{p_i^{\text{SLAM}}\}$ generated by the SLAM system are extracted and used as the source point cloud for registration against the pre-existing reference point cloud $\{p_i^{\text{ref}}\}$. This process estimates the transformation $T_{\text{SLAM}}^{\text{ref}}$ that relates the SLAM coordinate system to the fixed reference frame of the pre-existing map, employing either existing rigid registration algorithms or the custom similarity (Sim(3)) estimation approach. By continuously updating this alignment as the SLAM map evolves, the system can correct for drift and refine the global position over time.

Once an estimate of this alignment is available, any locally tracked camera pose $T_{\text{cam}}^{\text{SLAM}}$ can be expressed in the global reference frame through composition:

$$T_{\text{cam}}^{\text{ref}} = T_{\text{SLAM}}^{\text{ref}} \cdot T_{\text{cam}}^{\text{SLAM}}.$$

Finally, this global pose is used to render the reference point cloud from the current virtual viewpoint, overlaying it onto the live camera feed to create the augmented reality experience. This separation ensures that the AR visualization remains responsive to user motion, while its alignment to the real world is robustly maintained by the background registration stage.

3.5 Mobile Development with Flutter

Realizing the proposed system on mobile devices requires a development approach that balances rapid iteration with high-performance computing. This is achieved by combining a modern application framework for the user interface with direct integration of native libraries for computationally intensive tasks.

3.5.1 Why Flutter?

Flutter was selected as the mobile framework for the application in this work due to its modern architecture, expressive UI capabilities, and ease of development. Flutter's reactive framework and widget-based approach allow for rapid prototyping and iteration, which makes the development process, especially of the user interface, more efficient. The hot-reload feature, which uses Dart's Just-In-Time (JIT) compilation, enables quick testing and debugging, significantly speeding up the development cycle. Furthermore, Flutter's performance is comparable to native applications, as it compiles to native ARM code using Ahead-Of-Time (AOT) compilation for release builds. The package ecosystem is also quite mature, providing a wide range of plugins and libraries that can be leveraged, while also providing good documentation and community support. The multi-platform capabilities of Flutter also deserve a mention, as they keep the possibility open to extend the application to other platforms.

3.5.2 Foreign Function Interface (FFI)

Although Dart and Flutter provide a good level of performance, there are still certain computationally intensive tasks that are better suited for native implementations. Especially for the SLAM and point cloud registration components that are needed in this work, every millisecond counts to achieve real-time performance on mobile devices. Furthermore, the existing libraries that are used, like ORB-SLAM3 and Small-GICP, are only implemented in C++ and would be difficult to replicate in Dart. To bridge this gap, Dart's Foreign Function Interface (FFI) is utilized, which allows for the integration of native C/C++ code and also provides some developer tools, like automatically generated bindings.

CHAPTER 4

System Design

This chapter presents the architectural design and implementation strategies of the proposed system. It provides an overview of the software components and their layered organization and also details the data processing pipeline that enables real-time operation. Finally, the threading model is discussed, highlighting the concurrency mechanisms employed to ensure responsiveness on mobile hardware.

4.1 Architecture Overview

The system is designed to follow a layered architecture that starts at the Flutter-based interface and application logic and goes down to the native C++ implementations of the SLAM and point cloud registration components. It also follows a modular approach, where each component is encapsulated and is responsible for a specific task, allowing for some separation of concerns and future extensibility.

The main application (`flutter_3d_localization`) implements the overall logic and multipage user interface using Flutter and Dart. It is responsible for managing different workflows and interactions between the user and the underlying subsystems, also orchestrating state and data flow. The application is structured into several key pages, each serving a specific purpose:

- The main page includes the camera widget, with the AR Overlay and a bottom sheet for quick settings and status information
- The gallery page allows for management and selection of pre-existing point clouds
- The registration settings page is used to configure the registration algorithms and parameters. There is also a benchmark section provided that allows for performance evaluation of different settings.

- The ORB-SLAM3 configuration page provides access to the SLAM system settings, like camera intrinsics and other parameters, while also offering exporting and analytics capabilities.
- The AR settings page provides options for customizing the AR rendering, like point size, colors, and downsampling thresholds.
- The about page provides information about the application, licenses, and credits.

The `orb_slam3_wrapper` package serves mainly as a Dart FFI wrapper around a native, slightly modified ORB-SLAM3 library. It exposes necessary functions to initialize the SLAM system, process camera frames, and retrieve the camera pose and map points, while also having lifecycle management capabilities. This also includes importing the vocabulary file needed for place recognition. The package abstracts away the complexities of the native code, providing a simple Dart interface for the main application to interact with. Here, all the dependencies of ORB-SLAM3, like OpenCV and Eigen, are also managed.

The `point_cloud_registration` package similarly acts as a Dart FFI wrapper around the Small-GICP library. This makes the use of the different ICP variants possible from Dart code. On top of this, additional logic is implemented to handle the multiscale registration approaches, manage point clouds, and also provide benchmarking capabilities to evaluate the performance of different algorithms and settings.

The `point_cloud_renderer` implements a simple rendering engine that is purely written in Dart. This design choice allows for better integration with the Flutter framework and simplifies the overall architecture. The subsystem provides a widget for rendering point clouds using a pinhole camera model and basic OpenGL-like rendering techniques, while also offering an AR overlay version.

4.2 System Pipeline

The processing pipeline of the proposed system can be broken down into several key stages, which are executed in a continuous loop to provide real-time camera tracking and localization within the pre-existing point cloud map. The main stages of this pipeline are as follows:

4.2.1 Frame acquisition and preprocessing

The pipeline initiates with the camera widget, which provides a continuous stream of frames from the selected sensor at specific settings once started. While there are multiple resolution presets available, the chosen image format is always YUV420. The reason for this lies in the fact that ORB-SLAM3 requires grayscale images for feature extraction and tracking, and since YUV420 contains a dedicated luminance channel, it can be easily converted to grayscale by extracting this channel directly, avoiding unnecessary color

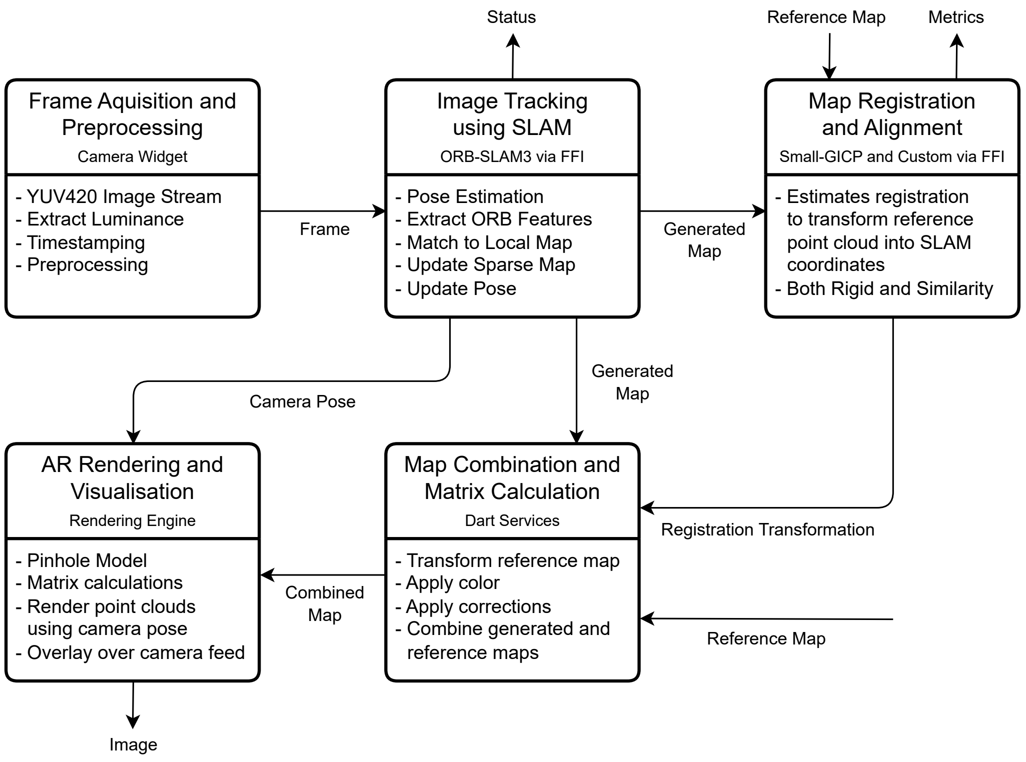

Status
Reference Map
Metrics
Frame Aquisition and Preprocessing
Camera Widget
- YUV420 Image Stream
- Extract Luminance
- Timestamping
- Preprocessing
Frame
Image Tracking using SLAM
ORB-SLAM3 via FFI
- Pose Estimation
- Extract ORB Features
- Match to Local Map
- Update Sparse Map
- Update Pose
Generated Map
Map Registration and Alignment
Small-GICP and Custom via FFI
- Estimates registration to transform reference point cloud into SLAM coordinates
- Both Rigid and Similarity
Camera Pose
Generated Map
AR Rendering and Visualisation
Rendering Engine
- Pinhole Model
- Matrix calculations
- Render point clouds using camera pose
- Overlay over camera feed
Combined Map
Map Combination and Matrix Calculation
Dart Services
- Transform reference map
- Apply color
- Apply corrections
- Combine generated and reference maps
Registration Transformation
Reference Map
Image

Figure 4.1: The high-level processing pipeline of the proposed system.

space conversions and reducing computational overhead. Each frame is then further timestamped in order to provide temporal consistency for the SLAM system. Lastly, the camera dimensions are captured for later use with the rendering engine.

4.2.2 Image Tracking and SLAM Integration

The preprocessed frames are then passed to the ORB-SLAM3 subsystem via the FFI wrapper. Here, the ORB features of each frame are extracted and matched against the local map of 3D points to estimate the camera pose $T_{\text{cam}}^{\text{SLAM}}$. The SLAM system also maintains and updates a sparse 3D map of the environment $\{p_i^{\text{SLAM}}\}$, which is periodically requested for the registration step and AR rendering. This process is described in more detail in Section 3.1.4.

For each frame, the current tracking status is also calculated. If the system has not been initialized yet, the tracking status is set to `NOT_INITIALIZED` and a few frames with enough parallax are needed to bootstrap the process. Once initialized, the system enters the `OK` state, where tracking is successful, and the map and pose are being updated normally. If tracking is lost due to insufficient features or rapid motion, the status changes to `LOST`, and the system will attempt to re-localize using the existing map points. The current tracking status is communicated back to the main application for user feedback and to manage the registration process accordingly.

Furthermore, the Atlas system of ORB-SLAM3 is utilized for basic analytics. Everything else of this data structure is, however, abstracted away, since only the currently active map is needed for this work.

4.2.3 Point Cloud Extraction and Registration

Once a reference point cloud has been chosen in the gallery and automatic registration is enabled after SLAM is initialized, the registration subsystem is activated. Here, the periodically extracted sparse 3D map points $\{p_i^{\text{SLAM}}\}$ from the SLAM step are continuously registered with the chosen pre-existing point cloud $\{p_i^{\text{ref}}\}$ in a set interval. Depending on the selected algorithm and settings, either rigid or similarity registration is performed to estimate the transformation $T_{\text{SLAM}}^{\text{ref}}$ between the two coordinate systems. This transformation is then used in the AR rendering step to view the reference point cloud from the correct camera pose. For user feedback, the error metrics and calculated changes are also communicated back to the main application.

4.2.4 Point Cloud Combination

In the next stage, the reference point cloud, which has been transformed into the SLAM coordinate system using the estimated transformation, $T_{\text{SLAM}}^{\text{ref}}$, is prepared for rendering. Based on the user selection, nothing, only the SLAM map points, only the reference point cloud, or both point clouds combined will be shown. Before both point clouds are merged, a uniform RGBA is applied to each point cloud, so that they can be visually distinguished. The combined point cloud is then downsampled below at a specific threshold using a voxel grid filter to ensure real-time rendering performance on mobile devices.

4.2.5 Augmented Reality Rendering

The final stage of the pipeline focuses on rendering the selected point clouds. This is handled by the rendering engine subsystem, which introduces a widget that is drawn above the camera feed from earlier. Rendering is performed using a simple pinhole model with the camera intrinsic parameters (focal lengths and principal point), which are set in the ORB-SLAM3 configuration. For each point, the process involves:

1. World-to-camera transformation: By using the inverse of the estimated camera pose $T_{\text{cam}}^{\text{ref}}$, each point in the world coordinate system can be transformed into the camera coordinate system: $p_i^{\text{cam}} = (T_{\text{cam}}^{\text{ref}})^{-1} \cdot p_i^{\text{ref}}$.
2. Projection onto image plane: In the next step, now every point in the camera coordinate system can then be projected onto the 2D image plane using the pinhole

camera model: $u_i = f_x \cdot \frac{x_i^{\text{cam}}}{z_i^{\text{cam}}} + c_x$, $v_i = f_y \cdot \frac{y_i^{\text{cam}}}{z_i^{\text{cam}}} + c_y$, where (f_x, f_y) are the focal lengths and (c_x, c_y) is the principal point.

3. Intrinsic transformation: Using the camera intrinsic, the projected points are then further transformed into pixel coordinates on the screen: $x_i^{\text{pixel}} = u_i$, $y_i^{\text{pixel}} = v_i$.
4. Viewport mapping: Lastly, the pixel coordinates are mapped to the viewport of the Flutter widget, taking into account aspect ratio and scaling.

There are also multiple correction settings implemented in the Dart services that allow for zoom, rotation, and vertical/horizontal offsets of the rendered point cloud to better align it with the camera feed.

4.3 Threading Model

The application, which is presented in this work, relies on multiple subsystems that have different performance and responsiveness requirements. To ensure smooth operation and real-time capabilities, a threading model is implemented to separate concerns and manage concurrency effectively:

4.3.1 Main Isolate Thread

The application runs primarily on the main Dart isolate thread, which is responsible for the user interface and the overall application logic. This includes all navigation, state management, and widgets. Asynchronous calls are handled using Futures and Streams, which allow for I/O operations without blocking, among other things, while not having to deal with thread management directly.

4.3.2 Frame Processing

The camera is also handled on the main thread, since it is part of the UI. The frames, however, are received from platform channels and are passed to the SLAM subsystem asynchronously to avoid blocking the UI using a Microtask. Since SLAM tracking is computationally intensive and therefore slower than the frame acquisition, a flag is employed to not process new frames while the previous one is still being handled, thus preventing a backlog of frames.

4.3.3 Concurrency in the FFI

When data is fed into the native subsystems via FFI, the execution will not run on the main Dart thread, but rather on a separate native thread. Dart, however, waits for the native call to finish before continuing execution on the main thread. Because of this, both SLAM and registration have to be performed asynchronously separately, even though they both run on native threads.

4.3.4 Native Layer Threading

When looking at the native C++ libraries, both ORB-SLAM3 and Small-GICP implement their own multithreading capabilities. ORB-SLAM3 specifically uses three concurrent threads for tracking, local mapping, and loop closing. The wrapper package around this library also introduces a mutex protection flag for thread-safe access to the current estimated camera pose.

4.3.5 Registration Thread

Point cloud registration is the heaviest task, which is part of the application and happens at a set interval. To ensure that no blocking of the main thread occurs, this is handled inside a dedicated Dart isolate that starts up its own native thread for executing the registration algorithms. This requires some additional data serialization and transfer overhead, but keeps the main application responsive. The prevention of multiple concurrent registration attempts is also achieved using flags. Inside the isolate, Small-GICP further accepts a thread count, which additionally parallelizes certain parts of the registration process, further improving performance.

4.3.6 AR Rendering

The rendering engine is implemented purely in Dart and runs on the main isolate thread, utilizing Flutter's CustomPainter class for drawing. Point clouds are fetched asynchronously, but the actual projection and drawing are performed synchronously.

CHAPTER 5

Implementation

This chapter gives a high-level overview of the implementation details and challenges encountered when developing this application. It covers the integration of the various subsystems, the design choices made, and how the overall architecture was realized in practice. The complete implementation is available as open-source software in the project's git repository [Mik25].

5.1 Native Library Integration

5.1.1 FFI Bridge

As previously discussed in Chapter 4, Dart's Foreign Function Interface (FFI) is used to expose native C++ libraries and functions to the Flutter application, thereby achieving the necessary performance for real-time operation. Both FFI plugin packages (`orb_slam3_wrapper` and `point_cloud_registration`) follow a consistent pattern in order to achieve this: a C-compatible header of the necessary functions is created with `extern "C"`, which is then annotated with `FFI_PLUGIN_EXPORT` to ensure proper symbol visibility. `ffigen` is then utilized to automatically generate the Dart bindings, which are then wrapped in higher-level Dart classes to provide a more idiomatic interface for the main application. This also requires memory management considerations, as data passed between Dart and C++ needs to be allocated and freed appropriately to avoid memory leaks.

5.1.2 Build System Integration

Native code is compiled directly within the Flutter build pipeline through the use of CMake and platform-specific build scripts. For Android, the `CMakeLists.txt` files are configured to include the necessary source files, dependencies, and compiler flags for ORB-SLAM3 and Small-GICP, respectively. The correct architecture has to be considered here (ARMv8, x86_64 are supported in this work) to ensure compatibility with the target devices.

5.2 ORB-SLAM3 Wrapper Package

From an integration standpoint, ORB-SLAM3 is the most complex component of the system. Here, many dependencies (like OpenCV, Eigen, DBoW2, g2o) are required in order for it to function. The correct binaries for the specific architecture have to be built and linked properly. Furthermore, some modifications to the original codebase were necessary in order to get it to run on mobile devices and to expose the needed functionality via FFI. This includes, for example, removing the native visualization components that are not needed in this work.

Another important aspect is the loading of the ORB vocabulary file, which is essential for place recognition and loop closure. Since this file is quite large (around 150MB), it is bundled as an asset in the Flutter application and then copied to the device's file system at runtime, where it can be accessed by the native code. The configuration parameters are also handled using this approach, by providing a YAML file that is parsed by ORB-SLAM3 during initialization.

For the map extraction to work properly, the native code iterates over the current map points in the active Atlas map and serializes them into a contiguous array that can then be passed back to Dart via FFI. Pose estimation follows a similar pattern, while additionally having to be converted from row-major to column-major format, as expected by the `vector_math` Dart library.

5.3 Point Cloud Registration Package

The integration of Small-GICP is far simpler. Here, there are fewer dependencies, and the library is also header-only, meaning that no separate compilation step is necessary. The main implementation work lies in implementing the custom similarity registration algorithms, which are not provided by the library out of the box. Apart from this, only a few extra considerations have to be made, like proper memory management of the point clouds passed via FFI and ensuring thread-safety during concurrent registration calls.

5.4 Point Cloud Renderer

Unlike the other packages, the renderer is written in pure Dart. This favors better integration, portability, and ease of development. Furthermore, an approach is chosen that does not rely on any dedicated rendering libraries or APIs. The main reason for this is the lack of mature and well-maintained 3D rendering options for Flutter at the time of writing. For example, `flutter_gl` has not been updated in multiple years. Furthermore, `flutter_gpu`, which is an experimental rendering API, still has a lot of rough edges and is not production-ready. The math follows a standard pinhole camera model approach, which is implemented using the `vector_math` Dart library. The actual rendering is performed using Flutter's `CustomPainter` class, which provides a canvas for drawing 2D graphics. Here, each point is drawn as a simple circle at its projected 2D coordinates.

5.5 Application Services

The rest of the application logic is implemented directly in Dart inside the main application package. To keep the code organized and modular, different services are created to encapsulate specific functionalities.

5.5.1 ORB Registration Service

One of the key services is the `OrbRegistrationService`, which manages the continuous registration process of the SLAM map points to the pre-existing point cloud. Here, the different registration modes are chosen based on the user settings, and the necessary parameters are prepared and passed to the subsystem at the specified interval. If requested, this is also the service where inversion of the registration direction is handled, by swapping source and target and then calculating the inverse transformation. After the process is complete, the resulting transformation and error metrics are stored and made available for the rest of the application.

5.5.2 Point Cloud File Management

Another important service is the `PointCloudFileService`, which handles loading and saving of point clouds inside a dedicated application directory. Besides CRUD and import functionality using `file_picker`, parsing of PLY files is implemented. This service is also responsible for creating the benchmark isolate, where the current point cloud is transformed and registered onto the original again to measure performance.

5.5.3 Settings Services

To manage the various settings and configurations of the application, multiple settings services are created. The `PCRegistrationSettingsService` and `ARRenderSettingsService` are quite similar in their implementation, as they simply use the `shared_preferences` package to persist user preferences across sessions. They also provide default values and validation logic to ensure that the settings are within acceptable ranges. Similarly, the `OverlayModeService` manages the different overlay modes for the AR rendering. The `OrbSlam3ConfigService` is a bit more complex, as ORB-SLAM3 takes a YAML configuration file for initialization and configuration. Here, a template file is stored as a string that is then modified at runtime based on the user settings and written to the device's file system for ORB-SLAM3 to access. There are also some values present that will not have any effect on the SLAM system (because of the removal of certain library components), but are required for proper parsing of the file.

5.6 Major widgets

The user interface of the application is built using Flutter's widget system. Several major widgets are created to encapsulate specific functionalities and provide a cohesive user experience.

5.6.1 Camera Widget

`camera` is responsible for handling the camera lifecycle, frame acquisition, and providing a preview of the camera feed with the AR overlay on top of the main page. It utilizes the `camera` package to access the device's camera hardware and manage different resolutions and settings.

5.6.2 Bottom Sheet

The `main_page_bottom_sheet` widget provides an expandable bottom sheet on the main page that displays quick settings, status information, and controls for starting/stopping the SLAM and registration processes. It also shows real-time metrics like registration error and tracking status, while serving as a central hub for all other functionalities.

5.7 Pages

The application is structured into multiple pages, which are mostly intended for configuration and management of the different subsystems and visualization options. Each page is implemented as a separate widget that encapsulates its own layout and logic:

- `main_page`: The main interface of the application, which combines the camera widget, bottom sheet, and AR overlay.
- `gallery_page`: A page for managing and selecting pre-existing point clouds, with options for importing, deleting, and viewing details.
- `registration_settings_page`: A configuration page for the point cloud registration subsystem, allowing users to select algorithms, set parameters, and run benchmarks.
- `orb_slam3_config_page`: A page for configuring ORB-SLAM3 settings, including camera intrinsics, vocabulary loading, exporting capabilities, while also providing analytics.

- `ar_settings_page`: A page for customizing AR rendering options, such as point size, colors, and downsampling thresholds.
- `about_page`: An informational page providing details about the application, licenses, and credits.

CHAPTER 6

Evaluation

This chapter looks at the performance of the proposed system in terms of accuracy, robustness, and real-time capabilities in real-world scenarios. Based on different test scenarios, the focus lies in answering the following research questions:

1. Can this system effectively register the SLAM-generated point cloud to the pre-existing map, therefore providing accurate camera poses in the reference coordinate system, making localization viable for AR applications? (see Section 6.3.1)
2. Does ORB-SLAM3 provide sufficient tracking accuracy and map quality on mobile devices to support reliable registration and localization? (see Section 6.3.2)
3. How do the different point cloud registration approaches compare? (see Section 6.3.3)
4. What are the practical limitations of this system and the chosen approaches when deployed on real mobile hardware? (see Section 6.3.4)

In the following sections, testing methodology, experimental setup, datasets used, and metrics for assessing performance are described. The results of the experiments are then presented, followed by a discussion of the findings in relation to the research questions outlined above.

6.1 Testing Methodology and Experimental Setup

The evaluation of the proposed system employs three test scenarios that cover different kinds of pre-existing point clouds and, therefore, registration scenarios for different environments and use cases. The scenarios increase in complexity and difficulty and focus on realistic conditions that would be encountered in practical applications.

6.1.1 Scenario 1: Rigid Registration (Same Device)

This scenario examines the ability of the system to handle the localization inside a point cloud that has been constructed using the same monocular SLAM setup and device. This represents the ideal case, where both point clouds share similar characteristics. The environment is first scanned using the application itself, generating a map, which is then exported and later re-imported for localization. Since the map that will be created in the localisation pass shares the same scale, the focus lies on the rigid registration capabilities of the system. Concretely, a Samsung Galaxy S22 Ultra is deployed in two distinct locations at Technische Universität Wien: a seminar room with only sparse features, including tables and chairs, and a community room of the student representation, which offers more visual clutter, like plants, furniture, and other objects.

6.1.2 Scenario 2: Similarity Registration (Cross-Device Simulation)

To evaluate this more difficult scenario of compatibility between different hardware, the map is still generated first using the SLAM system, to then be exported later. The difference lies in the fact that this first mapping pass cloud is then scaled up by a factor of 2.0, before re-importing. This simulates the case where the reference map was created using a different device with different camera intrinsics than the one used for localisation, leading to these scale differences. Therefore, the focus here lies on the similarity registration capabilities of the system. The same community room environment and device as in Scenario 1 are used for this purpose.

6.1.3 Scenario 3: Cross-Modality Registration

This final and most challenging scenario evaluates the system's performance when localizing inside a pre-existing point cloud that was created using a completely different modality. Here, an outdoor LiDAR scan of a street corner in Vienna, Austria, is used as the reference map. Instead of using the same device as before, an Oppo Find X3 smartphone is deployed for this purpose for availability reasons. This, however, should not have a significant impact on the results, as the focus lies on the cross-modality registration capabilities of the system and its ability to handle large discrepancies in point cloud density, noise characteristics, and coverage, no matter the device. The similarity registration algorithms are again evaluated in this scenario, since the scale difference is also unknown here.

6.1.4 Configuration Parameters and Metrics

Across all scenarios, the ORB-SLAM3 configuration was always initialized with the parameters that were calculated for the specific device and sensors using the provided guide in the code's documentation. The registration parameters were tuned to fit the specific scenario. This was done by either analyzing the exported SLAM-generated point cloud or LiDAR scan, respectively, before the final localization pass and picking sensible values for downsampling resolution and maximum correspondence distance, among others.

The exact values are described in Table **??**. Every parameter that is not mentioned there was kept at the default value of the application.

Table 6.1: Experimental Parameters and System Configuration across Test Scenarios

Parameter	**Scenario 1**	**Scenario 2**	**Scenario 3**
Registration Task	**Rigid** $SE(3)$	**Similarity** $Sim(3)$	**Similarity** $Sim(3)$
Algorithms / Modes	PLANE ICP / GICP	**Internal:** Custom Loop (Custom ICP) **External:** `small_gicp` (PLANE ICP / GICP)	
Device & Intrinsics			
Device Model	Samsung S22 Ultra	Samsung S22 Ultra	Oppo Find X3
Focal Length (f_x, f_y)	$355.0, 533.0$	$355.0, 533.0$	$294.0, 442.0$
Principal Point (c_x, c_y)	$240.0, 360.0$	$240.0, 360.0$	$240.0, 360.0$
Resolution	480×720	480×720	480×720
ICP Configuration			
Downsampling Res.	0.01m	0.01m	1.0m
Max Corr. Distance	1.5m	1.5m	50.0m
Max Iterations	100	100	20
Conv. Threshold	$1e-6$	$1e-6$	0.001

The primary evaluation metric is the registration error, which is calculated as the Root Mean Square Error (RMSE) between corresponding points after applying the estimated transformation. Additionally, the visual quality of the registration is assessed, while also considering the smoothness and responsiveness of the application during real-time operation on the mobile device. Since monocular SLAM tracking is also prone to interruptions, re-initialization sometimes forces the registration process to restart on a new map. This often produces significant variance in registration outcomes between these separate localization sessions.

6.2 Results

Based on the defined scenarios and configurations, the results of the experiments are presented in this section. This includes a benchmark of the registration algorithms in isolation, followed by the findings from each scenario.

6.2.1 Registration Evaluation: Synthetic Benchmark

In order to provide a better analysis for the full real-time system in the following sections, a synthetic benchmark is first performed separately to evaluate the custom $Sim(3)$ registration algorithms in isolation. Here, the SLAM-generated point cloud from Scenario 2 is taken along with the corresponding settings defined in Table **??**. Three different test cases are then defined, where the reference point cloud is transformed using known random transformations with varying degrees of difficulty: a baseline case (12° rotation, 1.3 scale factor), a noisy case with 10% Gaussian noise and 30% point removal, and a stress test with a significant misalignment (53° rotation, 2.0 scale factor).

Table 6.2: Comparison of External and Internal $Sim(3)$ Registration Performance

Metric	**Case 1: Baseline** (Input Scale 1.3)		**Case 2: Noisy** (Input Scale 1.3)		**Case 3: Stress Test** (Input Scale 2.0)	
	Ext.	**Int.**	**Ext.**	**Int.**	**Ext.**	**Int.**
Final Error (RMSE) [m]	0.002	**0.0005**	0.034	**0.008**	1.362	0.154
Rotation Est. [deg]	11.99°	12.00°	11.97°	11.99°	49.60°	21.84°
Scale Est. (Target S^{-1})	0.769	0.769	0.769	0.769	0.377	0.200
Target Value	*(0.769)*		*(0.769)*		*(0.500)*	
Iterations	**3**	21	**3**	23	32	21
Converged	Yes	Yes	Yes	Yes	**No**	**No**

In the baseline and noisy cases, both $Sim(3)Internal$ and $Sim(3)External$ perform quite well and were able to recover the correct transformation. The External mode was computationally faster, taking only 3 iterations, while the Internal mode required 23. Conversely, the internal mode achieved a better accuracy, with an RMSE below 0.01m, even when noise was present, while the external mode had an RMSE of around 0.034m in both cases.

In the stress tests, however, both algorithms struggled to converge to the correct solution. The External mode estimated the rotation slightly better at 49.6°, with the Internal mode only reaching 21.8°. Both scale and translation were approximated incorrectly, leading to high RMSE values of 1.36m and 1.54m, respectively. Unlike the sub-centimeter accuracy of the baseline case, these values highlight that without a coarse initial alignment, the algorithms are prone to getting stuck in local minima, resulting in large errors even in synthetic environments. These results confirm that while both algorithms are effective under moderate conditions, they face challenges when dealing with significant misalignments and noise, a factor that directly influences the performance in the following real-world scenarios.

6.2.2 Scenario 1: Same Device Registration

In this baseline scenario, the system showed very good real-time throughput across all stages. ORB-SLAM3 took around 3-5 seconds to initialize and was very stable and responsive thereafter. The configured settings provided good tracking results, with only minor losses in challenging conditions (fast motion, low texture). The registration step was also near instantaneous and could therefore be performed at tighter intervals (around 1 second) and more demanding settings (seen above).

The tracking accuracy of ORB-SLAM3 was also quite good, providing high-quality data for AR rendering that was visually well-aligned with the camera feed. The registration result, however, showed mixed results based on the environment and conditions. In the seminar room, the values ranged widely from an RMSE of around 3.1m to 15.4m, depending on the viewpoint of the localisation scan, oscillating mostly between consecutive intervals in the same pass. In the community room, the results were a bit more accurate, with RMSE values between 0.3m and 13.5m, but still inconsistent. Here, when not creating a new SLAM map, the accuracy decreased initially, but oscillated afterwards. The rendered overlay of the registration also ranged from mostly good to visibly misaligned. The choice of registration algorithm did not influence these values in any apparent way.

The main reason for the observed inaccuracies in the results is presumably that the SLAM-generated point cloud relies heavily on visual features. This introduces the possibility of inconsistent densities and coverages between maps, depending on the area where the system was initialized and how the subsequent tracking was performed. This then leads to difficulties for the registration algorithms to find good correspondences between the two point clouds. The variability in the results is likely further exacerbated by the lack of an initial pose guess from the SLAM system. Without an initial guess to guide the registration, the algorithm's success is influenced by whether the random initialization point is close enough and oriented in a similar direction for convergence to occur.

6.2.3 Scenario 2: Cross-Device Registration

In the second scenario focused on similarity capabilities, the system throughput remained similar to the previous section. ORB-SLAM3 performed well, and the registration step still happened near-instantaneously, no matter which algorithm was chosen.

When looking at the accuracy results, however, the quality of the results decreased further. The RMSE values now ranged even more widely, from around 0.6m all the way to 26.2m. Since the exported point cloud was scaled in a controlled manner, the scale estimation of the registration algorithm could also be evaluated. Since the map was scaled by a factor of 2, the registration had to predict a value close to 0.5 to compensate for this. The results here showed somewhat better accuracy and consistency than the RMSE, with scale being estimated between 0.39 and 0.58. Visually, the registration also did not provide good results, with the overlay being misaligned in most cases. Both registration algorithms performed comparably, while providing mostly oscillating results within a single localization session.

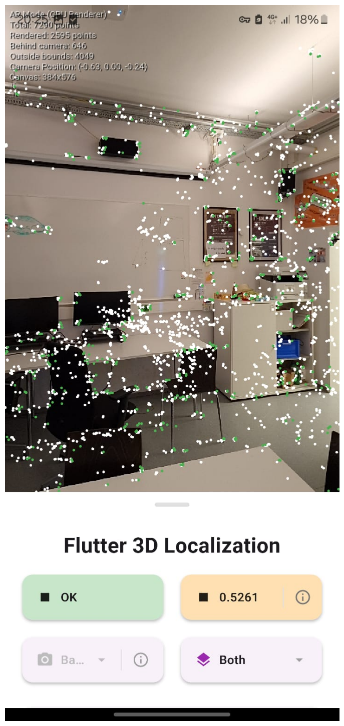
These findings can be explained by the same reasons as in Scenario 1, but now further exacerbated by the added complexity of scale estimation. The SLAM-generated point cloud still suffered from inconsistent coverage and features, making it difficult for the registration algorithm to find good correspondences and estimate an accurate transformation.

6.2.4 Scenario 3: Cross-Modality Registration

In this final scenario, the system throughput decreased noticeably. ORB-SLAM3 still performed adequately, but the registration step now took up to 30 seconds to complete (in both External and Internal modes), causing subsequent registrations to wait for the previous one to finish and effectively increasing the registration interval beyond the configured value, thereby harming the user experience.

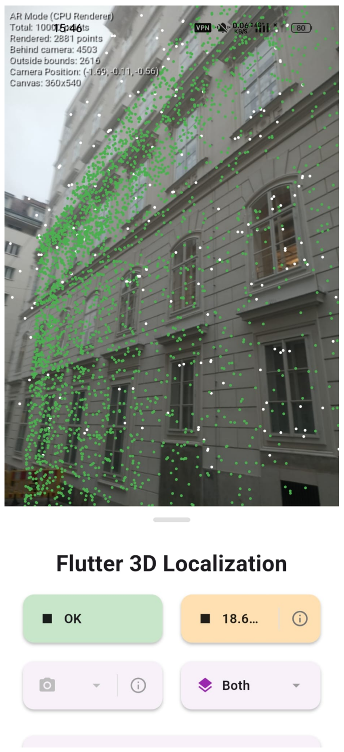
Here, the quality of the results further decreased. The RMSE values now ranged from around 18.2 m all the way to 230.4 m. These results appeared largely stochastic, exhibiting significant oscillation both between consecutive updates within a single continuous tracking session and across different localization passes when SLAM lost tracking and re-initialized. The scale estimation also suffered visually. This, however, could not be confirmed by numbers, since the pre-existing LiDAR scan did not provide scale metrics.

This drop in performance can be explained by the significant differences in point cloud characteristics between the SLAM-generated map and the LiDAR scan. The density, distribution, and noise profiles of the two point clouds differ greatly, making it challenging for the registration algorithms to find reliable correspondences and estimate an accurate transformation.



```
AR Mode (CPU Renderer)
Total: 7290 points
Rendered: 2595 points
Behind camera: 646
Outside bounds: 4049
Camera Position: (-0.63, 0.00, -0.24)
Canvas: 384x576
```

(a) Scenario 1: Successful alignment in the Community Room (Same Device).



```
AR Mode (CPU Renderer)
Total: 1000[illegible]ts
Rendered: 2881 points
Behind camera: 4503
Outside bounds: 2616
Camera Position: (-1.69, -0.11, -0.56)
Canvas: 360x540
```

(b) Scenario 3: Registration overlay against the LiDAR street scan (Cross-Modality).

Figure 6.1: Qualitative results of the application in operation. SLAM points are drawn in white and reference map points in green. The value on the bottom sheet shows the RMSE.

Table 6.3: Summary of Experimental Results across Test Scenarios

Scenario	Task	Env. / Input	RMSE [m]	Scale	Visual Assessment
1: Same Dev.	Rigid	Seminar Room	3.1 – 15.4	–	High variance
		Community Room	0.3 – 13.5	–	Occasional good align.
2: Cross-Dev.	Sim.	Comm. (Scaled)	0.6 – 26.2	≈ 0.5	Poor alignment
3: Cross-Mod.	Sim.	Outdoor (LiDAR)	18.2 – 230.4	Failed	Diverged; high latency

6.3 Discussion

The results from the evaluation provide the necessary insights to answer the research questions outlined at the beginning of this chapter, creating a broader understanding of the system's capabilities and limitations.

6.3.1 Effectiveness of the System for the Formulated Task

The evaluation results across the three scenarios indicate that while the proposed system is capable of performing point cloud registration and localization in real-time on mobile devices, its effectiveness is highly dependent on the characteristics of the pre-existing point cloud map and the conditions under which the SLAM-generated map is created. In the least difficult case of same-device registration, the system can achieve reasonable accuracy when other factors also align well. When even small changes in coverage, path, and viewpoint are introduced, the registration accuracy degrades significantly, leading to a low overall reliability and robustness of the system. This trend continues and worsens in the more complex scenario of cross-device registration, where scale differences further complicate the task by introducing an additional degree of freedom. The most difficult scenario of cross-modality registration highlights the limitations of the current approach: while sometimes delivering somewhat approximate results, it more often than not breaks down completely due to significant differences in point cloud characteristics between SLAM-generated maps and LiDAR scans.

Overall, the real-time aspect of the system is a strong point, with ORB-SLAM3 and the registration algorithms performing adequately on mobile hardware. However, the accuracy and robustness of the registration results are not sufficient for reliable AR localization in practical applications. Different settings also make the use of this system quite difficult, since there is no clear guidance on how to choose them for a specific use case. Therefore, while the system demonstrates the feasibility of real-time point cloud registration on mobile devices, further improvements are needed to enhance its effectiveness for AR localization tasks.

6.3.2 Suitability of ORB-SLAM3 for Mobile AR Localization

ORB-SLAM3 proves to be a capable SLAM system for mobile AR localization tasks, offering real-time performance and robust tracking capabilities on mobile hardware. The monocular configuration, which was used in this work, offers the advantage of simplicity and ease of deployment, as it only requires a single camera, which is readily available on most mobile devices. This, however, comes at the cost of the quality of the generated point cloud map, which is inherently sparse and relies heavily on visual features. This not only creates challenges when the image quality is poor, but also leads to inconsistent coverage and density in the resulting point cloud, depending on the motion and path taken during the mapping phase. This negatively impacts the subsequent registration step, as reliable correspondences are harder to find, resulting in reduced accuracy and robustness.

6.3.3 Comparison of Point Cloud Registration Approaches

When looking at the synthetic benchmark results in Section 6.2.1, both the custom internal similarity registration algorithm and the external Small-GICP-based approach demonstrate promising performance under controlled conditions, with low RMSE values and successful convergence in the baseline and noisy test cases. The result of the stress test, however, highlights the limitations of both algorithms when faced with significant misalignments, a factor that directly impacts their suitability for real-world applications.

When integrated into the real-world system, these limitations became evident. The rigid registration algorithms struggled to find reliable correspondences in the SLAM-generated point clouds, leading to high RMSE values and visually misaligned overlays. The similarity registration approaches faced additional challenges due to scale differences and the inherent variability in point cloud characteristics, resulting in even higher errors and inconsistent scale estimations. Overall, while both registration approaches show promising estimations under controlled conditions, their effectiveness and, more specifically, robustness in this concrete application are limited.

6.3.4 Practical Limitations on Mobile Hardware

The deployment of the proposed system on mobile devices introduces multiple limitations. Performance and the unavailability of advanced sensor configurations are the main issues. While the overall application runs well and can provide real-time capabilities in most cases, the accuracy of the results is hindered by these characteristics. The monocular SLAM setup limits the quality of the generated point cloud maps, which in turn affects the registration accuracy. The computational resources of mobile devices also constrain the complexity of the algorithms that can be implemented, so lighter-weight registration methods had to be chosen over more advanced alternatives, resulting in inherent trade-offs between achievable accuracy and processing speed. Furthermore, the variability in mobile device hardware and camera quality can lead to inconsistent performance across different platforms.

CHAPTER 7

Conclusion and Future Work

This chapter concludes the thesis by summarizing the key contributions and findings regarding the formulated problem. It reflects on the achieved results in relation to the initial research objectives and discusses the limitations encountered. Finally, potential directions for future research are outlined to address the identified challenges.

7.1 Summary of Contributions and Findings

Overall, this thesis set out to address the challenge of real-time localization in pre-existing point cloud maps on mobile hardware, overcoming the limitations of proprietary ecosystems and the computational constraints of direct image-to-point cloud registration. The presented solution is a fully functional mobile application that integrates established components, specifically ORB-SLAM3 and Small-GICP, into a novel two-stage pipeline, while also introducing custom similarity registration algorithms to handle scale ambiguities. The evaluation demonstrated the feasibility of the concept, while also highlighting both the strengths and significant hurdles regarding accuracy and robustness, particularly when bridging the domain gap between sparse SLAM maps and dense LiDAR reference data.

The core hypothesis, that sparse SLAM-generated point clouds can be effectively registered to pre-existing maps for localization purposes, was tested across multiple scenarios of increasing complexity and resulted in the following key findings:

- Real-Time Feasibility: The system successfully demonstrated real-time capabilities on mobile devices, with ORB-SLAM3 and the registration algorithms performing adequately within the constraints of mobile hardware.

- Registration Accuracy: While the system could achieve reasonable accuracy in controlled scenarios, the registration results were highly variable and often insufficient for reliable AR localization in practical applications.
- Scale Ambiguity: The similarity registration approaches further exposed weaknesses of the system, although also performing well under controlled conditions.
- Gap in Modality: The most difficult challenge that was identified is the inherent gap between SLAM-generated point clouds and those created using other modalities like LiDAR. This discrepancy in characteristics poses a major hurdle for effective registration.

Beyond the conceptual framework, the thesis also provides a robust implementation basis for future research, specifically through the integration of ORB-SLAM3 into Flutter and the development of a registration library extending Small-GICP with custom similarity estimation algorithms.

7.2 Future Work

In order to bridge the identified gaps and improve the overall system, the two major parts of the pipeline could be further investigated and enhanced.

7.2.1 Enhancing Map Quality and Scale

Since the monocular setup proved to be a limiting factor in terms of point cloud quality, investing in other sensor configurations could yield better results. For example, stereo or RGB-D SLAM systems could provide denser and more reliable point clouds, while also solving the scale ambiguity inherent in monocular setups. This, however, could limit the applicability of the system, since not all mobile devices are equipped with the necessary hardware, namely multiple cameras or time-of-flight sensors. As a sweet spot, IMU-augmented monocular SLAM could be investigated, but this approach might also suffer from similar issues as pure monocular SLAM.

7.2.2 Advanced Registration Strategies

The standard geometric registration approaches also restricted the overall system performance. This could be improved by exploring machine learning-based approaches that can learn robust feature representations and correspondences from data, potentially improving registration accuracy and robustness. Additionally, more robust similarity registration algorithms could be employed that can better handle scale differences and outliers. This, however, might also require more performant hardware.

Overview of Generative AI Tools Used

The use of AI tools in this work was mainly limited to supportive tasks and did not replace core findings or contributions. For the implementation and application part, GitHub Copilot was used to assist with code generation and suggestions, primarily for refactoring, boilerplate code, debugging, and visual enhancements, as well as some basic logic implementations. It was also responsible for documentation and comments. Concerning the writing part of this work, Perplexity AI was employed as a supportive tool for literature search, receiving general writing advice and occasionally generating or rephrasing small text snippets. All AI-generated content was critically evaluated, verified, and edited to ensure accuracy, coherence, and alignment with the original ideas and findings.

Bibliography

[AHB87] K. Somani Arun, Thomas S. Huang, and Steven D. Blostein. Least-squares fitting of two 3-d point sets. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, (5):698–700, 1987.

[BM92] Paul J. Besl and Neil D. McKay. A method for registration of 3-d shapes. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 14(2):239–256, 1992.

[CCC+16] Cesar Cadena, Luca Carlone, Henry Carrillo, Yasir Latif, Davide Scaramuzza, José Neira, Ian Reid, and John J Leonard. Past, present, and future of simultaneous localization and mapping. *IEEE Transactions on Robotics*, 32(6):1309–1332, 2016.

[CDM08] Javier Civera, Andrew J Davison, and JM Martinez Montiel. Inverse depth parametrization for monocular slam. *IEEE Transactions on Robotics*, 24(5):932–945, 2008.

[CER+21] Carlos Campos, Richard Elvira, Juan J. Gómez Rodríguez, J. M. M. Montiel, and Juan D. Tardós. Orb-slam3: An accurate open-source library for visual, visual–inertial, and multi-map slam. *IEEE Transactions on Robotics*, 37(6):1874–1890, 2021.

[DWB06] Hugh Durrant-Whyte and Tim Bailey. Simultaneous localization and mapping: part i. *IEEE Robotics & Automation Magazine*, 13(2):99–110, 2006.

[DXO21] DXOMARK. Multi-camera smartphones: Benefits and challenges. `https://www.dxomark.com/multi-camera-smartphones-benefits-and-challenges/`, 2021. Accessed: 2025-12-15.

[EHS+14] Felix Endres, Jürgen Hess, Jürgen Sturm, Daniel Cremers, and Wolfram Burgard. 3-d mapping with an rgb-d camera. *IEEE Transactions on Robotics*, 30(1):177–187, 2014.

[ESC14] Jakob Engel, Thomas Schöps, and Daniel Cremers. Lsd-slam: Large-scale direct monocular slam. In *European Conference on Computer Vision (ECCV)*, pages 834–849. Springer, 2014.

[ESC15] Jakob Engel, Jörg Stückler, and Daniel Cremers. Large-scale direct slam with stereo cameras. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 1935–1942, 2015.

[FHASAM16] Mohammed Faisal, Ramdane Hedjar, Mansour Al-Sulaiman, and Khalid Al-Mhedib. Multi-sensors multi-baseline mapping system for mobile robot. *Advances in Mechanical Engineering*, 8(6), 2016.

[Goo24] Google. ARCore Cloud Anchors. `https://developers.google.com/ar/develop/cloud-anchors`, 2024. Accessed: 2025-12-15.

[Imm24] Immersal Ltd. Immersal SDK Documentation: Visual Positioning System. `https://immersal.com/developers`, 2024. Accessed: 2025-12-15.

[Inc24] Apple Inc. ARKit Documentation: Saving and Loading World Data. `https://developer.apple.com/documentation/arkit/arworldmap`, 2024. Accessed: 2025-12-15.

[Ish23] Yohsuke Ishida. small_gicp: A header-only, cpu-optimized, and flexible gicp implementation. `https://github.com/SMRT-AIST/small_gicp`, 2023. Accessed: 2025-12-15.

[KLL$^+$23] Shuhao Kang, Youqi Liao, Jianping Li, Fuxun Liang, Yuhao Li, Xianghong Zou, Fangning Li, Xieyuanli Chen, Zhen Dong, and Bisheng Yang. Cofii2p: Coarse-to-fine correspondences-based image-to-point cloud registration. *arXiv preprint arXiv:2309.14660*, 2023.

[KOY21] Kenji Koide, Shuji Oishi, and Kei Yokoi. Voxelized gicp for fast and accurate 3d point cloud registration. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 11130–11136. IEEE, 2021.

[LL21] Jiaxin Li and Gim Hee Lee. Deepi2p: Image-to-point cloud registration via deep classification. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 15960–15969, 2021.

[LLB$^+$15] Stefan Leutenegger, Simon Lynen, Michael Bosse, Roland Siegwart, and Paul Furgale. Keyframe-based visual-inertial odometry using nonlinear optimization. *The International Journal of Robotics Research*, 34(3):314–334, 2015.

[Low04] Kok-Lim Low. Linear least-squares optimization for point-to-plane icp surface registration. Technical Report TR04-004, University of North Carolina, Chapel Hill, 2004.

[Mag24] Magic Leap, Inc. Magic Leap 2 Developer Documentation: Spatial Mapping & Localization. https://developer-docs.magicleap.cloud/docs/guides/features/spaces/spaces-tool/#, 2024. Accessed: 2025-12-15.

[MAMT15] Raúl Mur-Artal, J. M. M. Montiel, and Juan D. Tardós. Orb-slam: a versatile and accurate monocular slam system. *IEEE Transactions on Robotics*, 31(5):1147–1163, 2015.

[MAT17] Raúl Mur-Artal and Juan D Tardós. Orb-slam2: An open-source slam system for monocular, stereo, and rgb-d cameras. *IEEE Transactions on Robotics*, 33(5):1255–1262, 2017.

[Mic23] Microsoft. Spatial mapping in DirectX - Mixed Reality. https://learn.microsoft.com/en-us/windows/mixed-reality/develop/native/spatial-mapping-in-directx, 2023. Accessed: 2025-12-15.

[Mik25] Florian Miklautsch. Flutter 3d localization: Real-time camera localization in point cloud maps. https://gitlab.cg.tuwien.ac.at/fmiklautsch/flutter_3d_localization, 2025. Accessed: 2025-12-22.

[PZK17] Jaesik Park, Qian-Yi Zhou, and Vladlen Koltun. Colored point cloud registration revisited. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 143–152, 2017.

[RL01] Szymon Rusinkiewicz and Marc Levoy. Efficient variants of the icp algorithm. In *Proceedings of the Third International Conference on 3-D Digital Imaging and Modeling (3DIM)*, pages 145–152. IEEE, 2001.

[RRKB11] Ethan Rublee, Vincent Rabaud, Kurt Konolige, and Gary Bradski. Orb: An efficient alternative to sift or surf. In *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, pages 2564–2571, 2011.

[RZHC22] Siyu Ren, Yiming Zeng, Junhui Hou, and Xiaodong Chen. Corri2p: Deep image-to-point cloud registration via dense correspondence. *IEEE Transactions on Circuits and Systems for Video Technology*, 32(12):8747–8760, 2022.

[SF11] Davide Scaramuzza and Friedrich Fraundorfer. Visual odometry: part i: The first 30 years and fundamentals. *IEEE Robotics & Automation Magazine*, 18(4):80–92, 2011.

[SHT09] Aleksandr Segal, Dirk Haehnel, and Sebastian Thrun. Generalized-icp. In *Robotics: Science and Systems*, volume 2, page 435, 2009.

[TMHF99] Bill Triggs, Philip F McLauchlan, Richard I Hartley, and Andrew W Fitzgibbon. Bundle adjustment—a modern synthesis. In *Proceedings of the International Workshop on Vision Algorithms*, pages 298–372. Springer, 1999.

[Ume91] Shinji Umeyama. Least-squares estimation of transformation parameters between two point patterns. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 13(4):376–380, 1991.

[VRC24] VRCompare. Comparison of Magic Leap 2 vs Microsoft HoloLens 2. `https://vr-compare.com/compare?h1=mt3AEYJu5&h2=EkSDYv0cW`, 2024. Accessed: 2025-12-15.

[ZPK18] Qian-Yi Zhou, Jaesik Park, and Vladlen Koltun. Open3D: A modern library for 3D data processing. *arXiv preprint arXiv:1801.09847*, 2018.