HAW HAMBURG

Bachelor thesis

Patrick Hochstrasser

A Markerless Multi-Camera Motion Capture System

Faculty of Engineering and Computer Science
Department Computer Science

Patrick Hochstrasser

A Markerless Multi-Camera Motion Capture System

Bachelor thesis submitted for examination in Bachelor´s degree
in the study course *Bachelor of Science Angewandte Informatik*
at the Department Computer Science
at the Faculty of Engineering and Computer Science
at University of Applied Science Hamburg

Supervisor: Prof. Dr. Thomas Lehmann
Supervisor: Prof. Dr. Christian Lins

Submitted on: 13. December 2024

Patrick Hochstrasser

Thema der Arbeit

A Markerless Multi-Camera Motion Capture System

Stichworte

Motion Capture, Computer Vision, Smart Home, Multi-Kamera System, Pose Estimation

Kurzzusammenfassung

In dieser Bachelorarbeit wird ein Proof-of-Concept-System zur markerlosen Bewegungserfassung für Smart-Home-Überwachungsanwendungen entwickelt. Das System verarbeitet Videostreams von mehreren Kameras, um menschliche Bewegungen im 3D-Raum zu rekonstruieren, ohne dass spezielle Marker oder Geräte benötigt werden. Die Implementierung verwendet eine verteilte Service-Architektur, die Videostreamverarbeitung, Posenerkennung und 3D-Rekonstruktionskomponenten kombiniert. Das System erreicht eine Kamerasynchronisation innerhalb von 33 ms und arbeitet mit 9–12 Bildern pro Sekunde. Während die Machbarkeit grundlegender Bewegungsverfolgung mit Consumer-Hardware kombiniert mit Industriekameras demonstriert wird, zeigt die Evaluierung Herausforderungen in Bezug auf Echtzeitleistung und Umgebungsrobustheit auf, die für den praktischen Einsatz in Wohnumgebungen gelöst werden müssen.

Patrick Hochstrasser

Title of Thesis

A Markerless Multi-Camera Motion Capture System

Keywords

Motion Capture, Computer Vision, Smart Home, Multi-Camera System, Pose Estimation

Abstract

This bachelor thesis develops a proof-of-concept markerless motion capture system for smart home monitoring applications. The system processes video streams from multiple cameras to reconstruct human movement in 3D space without requiring special markers or equipment. The implementation uses a distributed service architecture combining video stream processing, pose detection, and 3D reconstruction components. The system achieves camera synchronization within 33 ms, operating at 9–12 frames per second. While demonstrating the feasibility of basic motion tracking with consumer-grade hardware combined with industrial cameras, the evaluation reveals challenges in real-time performance and environmental robustness that need to be addressed for practical deployment in residential settings.

Contents

List of Figures

List of Tables

Abbreviations

BHV Biovision Hierarchy.

CNN Convolutional Neural Network.

CPU Central Processing Unit.

CSV Comma-Separated Values.

DLT Direct Linear Transformation.

EPnP Efficient PnP.

FBX Filmbox.

FPS frames per second.

FR functional requirement.

GIL Global Interpreter Lock.

GPU Graphics Processing Unit.

HAW University of Applied Science.

IMU Inertial Measurement Unit.

mocap motion capture.

NFR non-functional requirement.

NTP Network Time Protocol.

P3P Perspective-3-Point.

PnP Perspective-n-Point.

RAII Resource acquisition is initialization.

RANSAC Random Sample Consensus.

RGB Red, Green, Blue.

RPC Remote Procedure Call.

RTCP Real-Time Transport Control Protocol.

RTP Real-time Transport Protocol.

SfM Structure from Motion.

SLAM Simultaneous Localization and Mapping.

UPnP Uncertainty-aware PnP.

VRAM Video Random Access Memory.

Symbols

α pixel aspect ratio.

B baseline between camera centers.

c_x x-coordinate of principal point.

c_y y-coordinate of principal point.

d distance function or reprojection error.

E essential matrix in epipolar geometry.

e epipole.

ϵ epipolar plane.

F fundamental matrix in epipolar geometry.

f focal length of a camera.

FPS frames per second, measure of video frame rate.

H homography matrix.

K camera intrinsic matrix.

k_1 first radial distortion coefficient.

k_2 second radial distortion coefficient.

k_3 third radial distortion coefficient.

l epipolar line.

λ scale factor in projection equations.

O camera optical center.

P 3D point in space or projection matrix (context dependent).

p_1 first tangential distortion coefficient.

p_2 second tangential distortion coefficient.

π projection function.

R rotation matrix in camera extrinsic parameters.

r radial distance from image center in distortion equations.

s skew coefficient in camera matrix.

t translation vector in camera extrinsic parameters.

θ angle between image rays in P3P algorithm.

1 Introduction

This chapter introduces the motivation and objectives of this bachelor thesis. The first section establishes the context and need for *smart home monitoring systems*, particularly in light of changing demographic trends and limitations of current solutions. The second section outlines the specific objectives of developing and evaluating a *markerless motion capture (mocap)* system for smart home environments. The chapter concludes with an overview of the thesis structure.

1.1 Motivation

The increasing number of single-person households and changing social structures have created new requirements for residential living environments. In Germany, single-person households have become the most common type of households, accounting for approximately 42% of all households in 2019. This proportion is projected to rise to about 44% by 2040 [19]. This trend emphasizes the growing need for technology-supported living spaces that help individuals maintain their quality of life and independence. While family members and healthcare providers traditionally fulfilled monitoring and support roles, changing social structures and limited healthcare resources require complementary technological approaches.

Smart homes have evolved beyond simple automation to become more complex environments equipped with sensors and actuators that can enhance residents' quality of life. These systems can support various monitoring applications, from early detection of health-related incidents and recognition of daily activity patterns to emergency response and automated health monitoring. The global smart home market reflects the increasing importance of these technologies, with projections indicating significant growth from USD 84.5 billion in 2024 to USD 116.4 billion by 2029, at a compound annual growth rate of 6.6% [49].

Present monitoring approaches often rely on wearable devices, multiple specialized sensors, or radar-based[1] [37] systems, which often have practical limitations in daily use. Wearable emergency systems, such as alert buttons, require active user participation; individuals must remember to wear and maintain these devices. If the device is not worn or is unaccessible during an emergency, it cannot provide assistance [52]. Similarly, fixed sensor installations throughout the residential space can be complex to implement and may offer limited information about the residents' activities [8], while radar systems often require expensive specialized hardware.

Markerless mocap systems offer a cost-effective solution to these challenges, leveraging widely available camera technology. By capturing human movement without requiring special equipment to be worn, these systems can provide continuous monitoring while maintaining privacy through *skeletal data*[2] processing rather than storing video footage. Research environments such as the Living Place[3] [35] laboratory at the *University of Applied Science (HAW) Hamburg* provide realistic environments for investigating and evaluating such monitoring technologies under real-world conditions.

1.2 Objectives of the Thesis

This bachelor thesis aims to develop and test a proof-of-concept *markerless motion capture* system for smart home environments. The work focuses on implementing a mocap solution that operates without requiring residents to wear markers or additional devices, considering practicality and user acceptance in daily life scenarios. The system processes skeletal data to monitor human movement within indoor environments, while attempting to safeguard the residents' privacy.

The thesis addresses several aspects: The implementation of the mocap system itself, including the processing of camera data and the extraction of skeletal information, and the technical evaluation of the system through basic performance metrics such as frame

[1] Radar-based systems typically use Frequency-Modulated Continuous Wave (FMCW) radar technology operating in frequency ranges like 24GHz or 60GHz to detect human presence and movement.

[2] Skeletal data consists only of geometric coordinates representing joint positions and connections, significantly reducing privacy concerns compared to raw video footage while retaining essential motion information.

[3] The Living Place laboratory is a 140 m^2 loft-style apartment equipped for ubiquitous computing research, featuring integrated systems for human-computer interaction studies. The lab provides a realistic single-room environment with dedicated areas for cooking, dining, sleeping, and working, along with monitoring capabilities through an adjoining control room.

rates and processing latency. This includes the examination of the system's behavior in different scenarios, including the identification and documentation of potential edge cases and limitations in the motion tracking process.

The thesis examines the suitability of the system as a basis for smart home monitoring applications, such as *fall detection*, *activity recognition*, and assisted living scenarios. The evaluation includes visual assessments of the triangulation stability and analysis of the detection accuracy. Through these technical assessments, the work provides insights into the practical aspects of implementing camera-based motion tracking in smart home settings, considering requirements like reliability and integration capabilities.

1.3 Thesis Structure

This thesis is organized into six chapters that describe the development and evaluation of the markerless mocap system.

The theoretical foundation begins with establishing background and related work for (markerless) mocap systems. This includes fundamental mocap concepts and progresses through technical aspects of camera models, calibration techniques, and pose estimation methods. The final section covers animation data generation, which provides the basis for the skeletal data processing.

System requirements, design, and architecture follow, starting with the definition of functional and non-functional requirements, including privacy considerations and practical constraints for daily use scenarios. The architecture is then described through multiple views—from context to component level—along with documentation of the architectural decisions that guided the implementation.

The implementation section focuses on the proof-of-concept system, outlining the selection of technologies and frameworks, presenting the code-level architecture, and describing the system components. This section demonstrates how the theoretical concepts are implemented in practice.

The evaluation examines the processing pipeline performance through metrics like frame rates and latency, pose detection results, and triangulation accuracy. It analyzes the system behavior in different scenarios and documents edge cases, concluding with an

analysis of how the implementation meets the defined requirements and its suitability for smart home monitoring applications.

The thesis concludes with a summary of the work and results, addressing current system limitations and presenting possibilities for future research and applications in motion tracking and analysis

To begin with the development of such a system, the following chapter introduces the theoretical foundations and current state of (markerless) mocap technology.

2 Theoretical Background and Related Work

This chapter establishes the theoretical foundations necessary for developing a mocap system. It begins with fundamental mocap concepts and technologies, followed by an examination of *camera models* and *calibration techniques* essential for *3D reconstruction*. The chapter then explores current approaches in *pose estimation* and *skeletal tracking*, concluding with methods for generating and processing *animation data*.

2.1 Introduction to Motion Capture

Mocap technology represents a bridge between physical movement and digital representation, encompassing various methods and approaches that have evolved over several decades. From its early applications in biomechanics research to modern uses in entertainment and healthcare monitoring, mocap continues to advance in both capability and accessibility.

2.1.1 Definition and Relevance

Mocap refers to the process of recording the movement of objects or individuals to translate real-world actions into digital models [17, pp. 10–13]. The technology captures motion by tracking specific points in space over time, reconstructing movement through spatial coordinates and temporal sequences. These points typically represent anatomical landmarks or joints, forming a *skeletal model* of the captured subject [63].

Mocap technology emerged from photogrammetry and biomechanics research in the 1970s and 1980s [23]. Early systems relied on multiple cameras and *manual digitization*

of points, evolving through several technological generations: from simple chronophotography to contemporary *computer vision* and *deep learning* approaches [54], [57], [58].

The fundamental technical process of mocap consists of several stages. The initial stage involves data acquisition through various sensing technologies—optical, inertial, or depth-based systems [101]. This raw data undergoes processing to identify *keypoints* or features of interest. In skeletal tracking, these points correspond to anatomical landmarks such as joints (e.g., elbow, knee, shoulder) that form a biomechanical model [90, p. 107–111]. The system then reconstructs these points in three-dimensional space, considering factors such as *occlusions*, *noise*, and *temporal consistency* [33, pp. 10–12], [80, pp. 568–576].

For indoor monitoring applications, mocap systems face specific technical challenges. These include varying lighting conditions, occlusions from furniture, multiple person tracking, and *real-time* processing requirements [87]. The accuracy of such systems depends on several factors: sensor resolution, processing algorithms, environmental conditions, and the complexity of tracked movements [72].

Recent advances in computer vision and machine learning have significantly influenced mocap technology. Deep learning approaches, particularly *Convolutional Neural Networks (CNNs)* [5], [14], [15], [48], [53], [62], [83] and transformer architectures [46], [91], [92], [97], [98], have improved the robustness of pose estimation. These developments have shifted the paradigm from traditional marker-based systems requiring multiple calibrated cameras to markerless solutions that can operate with standard *Red, Green, Blue (RGB)* cameras. However, this reduced physical hardware complexity comes at the cost of increased computational demands, as these deep learning models typically require significant *Graphics Processing Unit (GPU)* resources for real-time inference[1].

The technical requirements for mocap systems vary based on their application. While some applications demand sub-millimeter accuracy and high sampling rates, monitoring systems in smart homes prioritize reliability and continuous operation.

[1]For example, OpenPose [14], [15] requires a dedicated GPU for real-time inference on a single camera feed, typically achieving 10–15 *FPS* on mid-range GPUs with 4–6 GB *Video Random Access Memory (VRAM)*. Similarly, transformer-based approaches like VIBE [42] demand even more substantial computational resources for both training and inference. Performance generally scales with input resolution and desired frame rate.

2.1.2 Motion Capture Technologies and Approaches

Mocap systems can be categorized by their *tracking methodology* and sensor technology. Understanding these different approaches helps in selecting appropriate systems for specific applications.

Sensor Technologies

Various sensor technologies enable mocap, each with distinct characteristics:

Optical systems use cameras to track motion and can be further divided into passive and active marker systems. Passive marker systems track reflective markers using infrared cameras, as shown in Figure 2.1, while active marker systems use LED or similar self-illuminating markers [69]. These systems achieve high precision (sub-millimeter accuracy) and sampling rates (100+ Hz) but require controlled environments and marker attachment [29, pp. 419–423].

Figure 2.1: Optical motion capture (mocap)—actors wearing mocap suits with reflectors
Source: Achrekar, A. In [17, p. 12]. Courtesy of Centroid Motion Capture.

RGB camera-based systems use standard video cameras, processing color images to detect human poses. While more accessible and cost-effective, they typically offer lower precision than marker-based systems and are more susceptible to lighting conditions [74].

Depth sensors, such as structured light (e.g., first-generation `Microsoft Kinect` [56]) or *time-of-flight cameras* (e.g., *Azure Kinect* [55]), provide additional depth information. These systems operate effectively in indoor environments, but may have limitations in range and precision [96].

Inertial Measurement Units (IMUs) track motion using *accelerometers*, *gyroscopes*, and occasionally *magnetometers*. These body-worn sensors provide direct motion measurements without occlusion issues, but may suffer from drift and require regular calibration [73].

Tracking Approaches

Mocap can target different aspects of human movement:

Full-body tracking, which this thesis primarily addresses, captures the movement of major body joints and segments. Marker-based systems typically track 30–60 markers for full-body capture (as illustrated in Figure 2.1), while markerless systems estimate comparable joint positions through computer vision techniques [63].

Facial mocap requires higher resolution and more detailed tracking to capture subtle expressions. Specialized systems use either dense marker sets, detailed texture analysis, or high-resolution depth mapping [13], [88].

Hand mocap presents unique challenges due to the complexity of hand articulation and frequent self-occlusions. Solutions range from data gloves to camera-based systems with specific hand models [60], [75], [102].

Marker-based vs. Markerless Comparison

Marker-based systems achieve high precision through specialized hardware. Professional optical systems can track markers with sub-millimeter accuracy at high frame rates, making them the standard for applications requiring precise measurements [29, p. 429]. These systems demand careful marker placement on the subject and require a calibrated capture volume. The operational complexity extends to regular system maintenance and considerable setup time before each capture session. Additionally, the operation of such systems requires specialized knowledge and training.

Markerless systems represent a different approach to mocap, focusing on accessibility and ease of use rather than maximum precision. By eliminating the need for physical markers or specialized suits, these systems reduce setup complexity and enable more natural movement capture. The reduced hardware requirements and automated tracking capabilities make them particularly suitable for continuous operation scenarios.

However, markerless approaches face their own technical challenges. The tracking precision typically remains below that of marker-based systems [39], particularly when capturing complex poses or rapid movements. Environmental factors such as occlusions and varying lighting conditions can significantly impact tracking reliability. The technical trade-offs manifest most notably in scenarios requiring highly precise measurements, where marker-based systems maintain their advantage.

For smart home monitoring applications, markerless systems present a more practical solution despite these limitations. The ability to operate continuously without user intervention aligns with the requirements of residential monitoring scenarios, where system reliability and ease of use take precedence over maximum precision.

2.1.3 Markerless Motion Capture Workflows

Different markerless mocap systems employ varying approaches to achieve their goals. While some systems, like the Microsoft Kinect, use depth sensors to directly capture 3D information [76], others rely on multiple RGB cameras and *triangulation*. Despite these technological differences, most markerless systems follow a general processing pipeline [29, pp. 424–428].

Before the actual mocap can begin, the system requires *camera calibration* determining the parameters that relate 3D world coordinates to 2D image coordinates. Once calibrated, the mocap workflow consists of several key stages:

1. **Data Acquisition:** Capturing images or videos using one or more cameras.
2. **2D Pose Estimation:** Detecting and locating keypoints (e.g., joints) in the 2D images.
3. **3D Reconstruction:** Reconstructing 3D poses from the 2D keypoints using *triangulation methods*.
4. **Animation data generation:** Arranging the reconstructed poses into a sequence.

The following sections examine each of these components in detail, providing the theoretical foundation for implementing markerless mocap systems.

2.2 Camera Models and Image Formation

Camera geometry and modeling provide mathematical foundations for understanding how three-dimensional scenes are projected onto two-dimensional *image planes* [33, p. 153], [80, pp. 41–63, 545–552]. While physical cameras perform this projection optically, mathematical models offer a formal framework for an algorithmic understanding and manipulation of the image formation process in computer vision tasks. These theoretical foundations, combined with calibration, support various applications, including 3D reconstruction, by enabling the inference of spatial information from 2D images. The accuracy of both the geometric models and their calibration directly impacts the precision of 3D reconstruction.

2.2.1 Camera Models

The *pinhole camera model* serves as a foundational concept, upon which more complex representations build. Real cameras introduce optical effects, like lens distortions, that deviate from this basic model. Advanced models incorporate these effects and the geometrical relationships between multiple views of a scene. Accounting for these factors contributes to more accurate image interpretation and 3D scene understanding. The development and utilization of comprehensive models can enhance the accuracy and reliability of computer vision systems across various practical applications.

Pinhole camera model

The pinhole camera model represents (see Figure 2.2) an ideal camera mathematically [33, pp. 153–156] [31, pp. 13–26] [80, pp. 44–51], [22, pp. 9–11, 253–257, 443–448]. This model assumes that all light rays pass through a single point (the pinhole) to form an image on the image plane. Key elements of this model include:

Focal Length f The distance between the pinhole and the image plane.

Optical Axis The line perpendicular to the image plane passing through the pinhole.

Principal Point The point where the *optical axis* intersects the image plane.

While simplistic, this model provides a basis for understanding more complex camera behaviors and is widely used in computer vision and graphics applications.

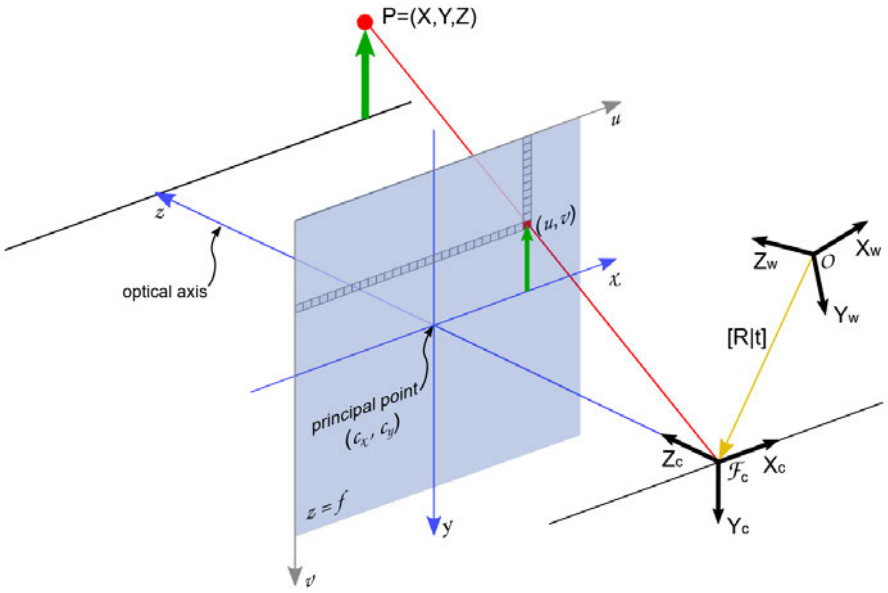


Figure 2.2: Pinhole Camera Model (Source: OpenCV Documentation [64]).

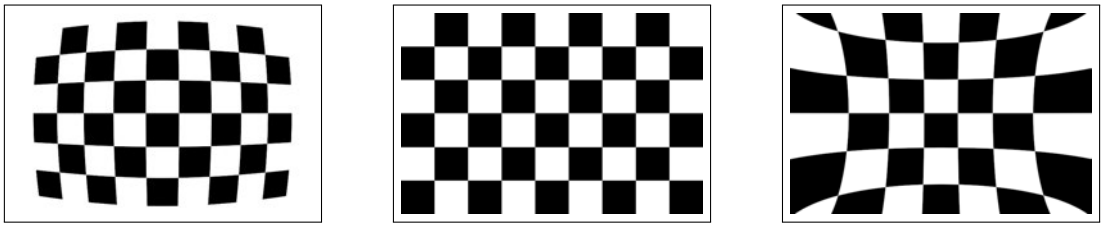
Lens distortion models

While the pinhole camera model provides a useful approximation, it does not account for the optical effects introduced by real camera lenses. Lens distortion models [36], [89] extend the basic pinhole model to more accurately represent image formation in physical cameras. The two primary types of lens distortion [80, pp. 51–53] are:

Radial distortion causes straight lines to appear curved in the image. It occurs due to the spherical nature of camera lenses and is more pronounced at the edges of the image.

Radial distortion can be modeled using polynomial functions [36], [89]:

$$\begin{aligned} x_{rad} &:= x(1 + k_1 r^2 + k_2 r^4 + k_3 r^6) \\ y_{rad} &:= y(1 + k_1 r^2 + k_2 r^4 + k_3 r^6) \end{aligned} \tag{2.1}$$



(a) Barrel distortion
positive radial displacement

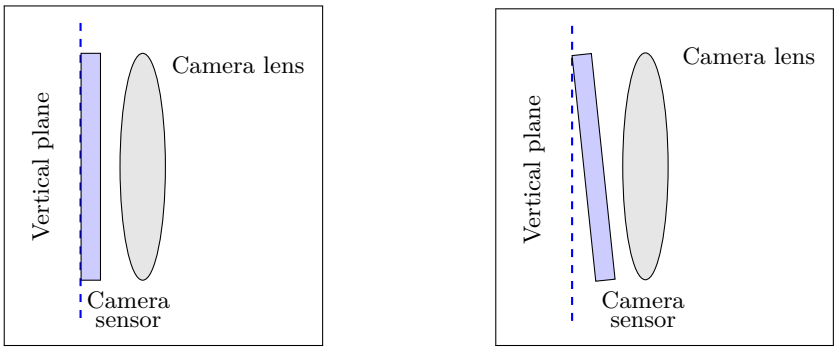
(b) No distortion

(c) Pincushion distortion
negative radial displacement

Figure 2.3: Examples of radial distortion.

where (x, y) are the undistorted coordinates, (x_{rad}, y_{rad}) are the distorted coordinates, r is the distance from the image center, and k_1, k_2, k_3 are the radial distortion coefficients.

Tangential distortion occurs when the lens is not perfectly parallel to the image plane. It causes some areas of the image to appear closer than expected.



Zero Tangential Distortion
Lens and sensor are parallel

Tangential Distortion
Lens and sensor are not parallel

Figure 2.4: Examples of tangential distortion in camera systems.

Tangential distortion can be modeled as [10], [94]:

$$\begin{aligned} x_{tang} &:= x + [2p_1xy + p_2(r^2 + 2x^2)] \\ y_{tang} &:= y + [p_1(r^2 + 2y^2) + 2p_2xy] \end{aligned} \tag{2.2}$$

where p_1 and p_2 are the tangential distortion coefficients.

Combined distortion model In practice, both radial and tangential distortions are often combined into a single model [36], [64], [94]:

$$\begin{bmatrix} x_{distorted} \\ y_{distorted} \end{bmatrix} := (1 + k_1 r^2 + k_2 r^4 + k_3 r^6) \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} 2p_1 xy + p_2(r^2 + 2x^2) \\ p_1(r^2 + 2y^2) + 2p_2 xy \end{bmatrix} \tag{2.3}$$

This model includes both radial (k_1, k_2, k_3) and tangential p_1, p_2 distortion parameters.

While lens distortion is crucial for accurate image formation modeling, the complete mapping from 3D world points to 2D image points requires additional parameters.

2.2.2 From 3D to 2D Image Space

This section presents the parameters required for modeling the projection from 3D space to a 2D space, such as a digital image. The discussion encompasses the various components involved in this mapping, including coordinate systems, camera parameters, and the projection process.

As mentioned earlier, a point P in 3D space can be mapped (or projected) into a 2D point P' in the image plane. This mapping is referred to as a *projective transformation*. This projection does not directly correspond to what appears in the actual image. First, points in the images are usually, in a different reference system than those in the image plane. Second, digital images are divided into pixels, whereas points in the image plane are continuous. Finally, the physical sensors, can as discussed before introduce non-linearity such as distortion to the mapping. To account for these differences, several additional transformations must be introduced that allow the mapping of any point from the 3D world to pixel coordinates [80, pp. 29–41].

Coordinate Systems

The understanding of different coordinate systems involved in the projection process is necessary before examining the camera matrix model [43, pp. 5–6]:

World Coordinate System The 3D coordinate system in which the scene and objects are defined.

Camera Coordinate System The 3D coordinate system with its origin at the camera's optical center.

Image Coordinate System The 2D coordinate system on the image plane, typically with the origin at the *principal point*.

Pixel Coordinate System The 2D discrete coordinate system of the digital image, with the origin usually in the top-left corner.

These coordinate systems each play a role in the image formation process. The *world* and *camera coordinate* systems enable representation of the 3D scene and the camera's position. The *image coordinate* system accounts for the projection of 3D points onto a 2D plane. Finally, the *pixel coordinate* system bridges the gap between the continuous mathematical model and the discrete nature of digital images. Understanding the transformations between these systems enables accurate modeling of how points in 3D space become pixels in the final image.

Homogeneous Coordinates

The process of projecting 3D points onto a 2D image plane involves non-linear operations, particularly divisions. These operations can complicate the mathematical model and make computations less efficient. *Homogeneous coordinates* offer a solution to this challenge [33, pp. 29–32], [22, pp. 195–242].

Homogeneous coordinates are a system of coordinates used in projective geometry to represent points and other geometric entities:

- A 3D point $(X, Y, Z)^T$ is represented as $(X, Y, Z, 1)^T$.
- A 2D point $(X, Y)^T$ is represented as $(X, Y, 1)^T$.

By adding this extra coordinate, homogeneous coordinates enable representation of projective transformations, including perspective projections, as linear matrix multiplications. This representation simplifies the mathematical model of the projection process and enables the use of more efficient linear algebra techniques [22, pp. 195–201], [80, pp. 728–741]. The actual projection from 3D to 2D is detailed in the projection process section, which demonstrates the practical application of these coordinates.

Camera Matrix Model

The camera matrix model [80, pp. 44–49], [33, pp. 178–193] describes a set of parameters that affect how a world point P is mapped to image coordinates P'. As the name suggests, these parameters will be represented in matrix form and are categorized into intrinsic and extrinsic.

Intrinsic Parameters: Describe the camera's internal characteristics affecting image formation.

The main *intrinsic parameters* include:

- **Focal length f:** The distance between the camera's lens and the image sensor
- **Principal point coordinates (c_x, c_y):** The point where the optical axis intersects the image plane. In an ideal camera, this would be at the center of the image, but in practice, it can deviate due to manufacturing imperfections or lens misalignment.
- **Pixel aspect ratio α:** The ratio of pixel width to height
- **Skew coefficient s:** Accounts for non-rectangular pixels (often assumed to be 0)

These parameters are represented in the camera intrinsic matrix K [86], [94]:

$$K := \begin{bmatrix} f_x & s & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \tag{2.4}$$

where $f_x = f\alpha$ and $f_y = f$.

The intrinsic matrix is used in camera calibration and 3D reconstruction tasks, which are discussed in subsequent sections.

Extrinsic Parameters: Define the camera's position and orientation relative to a world coordinate system. These parameters consist of:

- **Rotation matrix R (3×3):** Describes the camera's orientation
- **Translation vector t (3×1):** Represents the camera's position

These parameters facilitate the transformation between world and camera coordinate systems.

The *extrinsic parameters* are typically combined into a single 3×4 matrix $[R|t]$, known as the extrinsic matrix.

With the intrinsic and extrinsic parameters defined, the following sections explore how these components work together in the projection process.

Projection Process

The projection process describes how 3D world points are mapped onto the 2D image plane using both intrinsic and extrinsic parameters. This process involves several steps: The transformation of points from the world coordinate system to the camera coordinate system using the extrinsic parameters; the projection of these camera coordinates onto the image plane, resulting in normalized image coordinates; the application of any lens distortion effects; and finally, the conversion to *pixel coordinates* in the final image using the intrinsic parameters. Each step in this process can be represented mathematically, modeling the entire projection as a series of transformations [33, pp. 153–156], [80, pp. 29–52].

Projection Equations The projection process can be described as follows:

1. **World to Camera Coordinates:** The transformation of a point from world coordinates to camera coordinates using the extrinsic parameters: $\mathbf{P_c} = R\mathbf{P_w} + t$ where $\mathbf{P_w} = (X_w, Y_w, Z_w)^T$ is the point in world coordinates, $\mathbf{P_c} = (X_c, Y_c, Z_c)^T$ is the point in camera coordinates, R is the rotation matrix, and t is the translation vector.

2. **Projection to Normalized Image Coordinates:** The projection of the point onto the normalized image plane: $x_n = \frac{X_c}{Z_c}, \quad y_n = \frac{Y_c}{Z_c}$ where $(x_n, y_n)^T$ are the normalized image coordinates.

3. **Lens Distortion:** If needed, the application of lens distortion to the *normalized coordinates*: $(x_d, y_d)^T = \text{distortion}(x_n, y_n)$ The distortion function includes both radial and tangential components, as detailed in the final equation below.

4. **Conversion to Pixel Coordinates:** Finally, the conversion of distorted coordinates to pixel coordinates using the intrinsic parameters: $\begin{pmatrix} u, v, 1 \end{pmatrix}^T = K \begin{pmatrix} x_d, y_d, 1 \end{pmatrix}^T$ where $(u, v)^T$ are the final pixel coordinates and K is the intrinsic matrix.

The complete projection process, combining all these steps, can be expressed in homogeneous coordinates as:

$$\lambda \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = K \begin{bmatrix} x_n + ((k_1 r^2 + k_2 r^4 + k_3 r^6) x_n + 2 p_1 x_n y_n + p_2 (r^2 + 2 x_n^2)) \\ y_n + ((k_1 r^2 + k_2 r^4 + k_3 r^6) y_n + p_1 (r^2 + 2 y_n^2) + 2 p_2 x_n y_n) \\ 1 \end{bmatrix} \tag{2.5}$$

Where:

$$\begin{bmatrix} x_n \\ y_n \\ 1 \end{bmatrix} = \frac{1}{Z_c} [R|t] \begin{bmatrix} X_w \\ Y_w \\ Z_w \\ 1 \end{bmatrix} \tag{2.6}$$

where $\lambda = Z_c$ is the depth of the 3D point in camera coordinates and serves as the scale factor, (x_n, y_n) are the normalized image coordinates before distortion, $r^2 = x_n^2 + y_n^2$, and k_1, k_2, k_3, p_1, p_2 are the distortion coefficients as defined in equation 2.3.

This set of equations encapsulates the entire projection process, incorporating extrinsic parameters (camera position and orientation), intrinsic parameters (focal length, principal point), and lens distortion effects. It allows prediction of where any 3D world point will appear in the final image, accounting for all aspects of the camera model.

In practice, using these equations requires accurate knowledge of all the parameters involved. This is where camera calibration becomes crucial [85], [86], [94]. Calibration determines the specific values of these parameters for each individual camera, enabling precise mapping between 3D world coordinates and 2D image coordinates. Conversely, in applications like 3D reconstruction or markerless mocap, these equations are applied in reverse, inferring 3D information from 2D images using the calibrated parameters.

2.3 Camera Calibration Techniques

Camera calibration is a process in computer vision and image processing, aimed at determining the internal and external parameters of a camera [31, p. 1].

The discussion begins with an examination of traditional single camera calibration methods, including the widely used Zhang's [94] method. This leads to an analysis of the complexities of *multi-camera calibration*, which is particularly relevant for multi-camera mocap systems. The section concludes with calibration techniques that enable accurate pose estimation, providing the foundation for subsequent sections on 3D reconstruction and motion tracking.

Purpose of calibration

The primary purpose of camera calibration is to accurately estimate the parameters described in the camera model, compensating for the various imperfections in the imaging process [31, pp. 27–39] (see Figure 2.5). Calibration allows for the precise determination of a camera's intrinsic parameters (such as focal length, principal point, and distortion coefficients) and extrinsic parameters (such as the camera's position and orientation in space). By refining these parameters, calibration ensures that the camera's model accurately reflects the real-world imaging process, enabling precise measurements and reliable image-based analysis. It typically involves several key steps. First, a calibration pattern

Figure 2.5: Camera image before undistortion (left) and after (right). Source: Bradski and Kaehler, Learning OpenCV [9, p. 678]

with known geometry is captured in multiple images from various angles and distances. Next, feature points (such as corners or centroids) are detected in these images. The correspondence between these image points and their known 3D world coordinates is then used to estimate the camera's intrinsic and extrinsic parameters. This initial estimation is often followed by an optimization step to refine the parameters and minimize *reprojection errors*. Finally, the calibration results are validated to ensure their accuracy and reliability.

Calibration Methods

Early camera calibration techniques, such as Tsai's method [86], relied on precise 3D calibration objects. While effective, these methods were often cumbersome and required specialized equipment, limiting their practicality in many applications.

Zhang's Method [94], introduced in 2000, revolutionized camera calibration by using a planar pattern viewed from multiple orientations. This approach offers several advantages:

Flexibility: Uses a simple printed pattern instead of elaborate 3D objects.

Ease of use: Can be performed by non-experts.

Accuracy: Provides results comparable to traditional methods.

This method involves capturing multiple images of a planar checkerboard pattern, followed by detecting corner points in each image. The *homography* between the pattern and each image is then estimated, and these homographies are used to compute both intrinsic and extrinsic camera parameters. A homography, in this context, is a transformation that maps points from one plane (the checkerboard) to another (the image plane). It's represented by a 3×3 matrix and is required for relating the known geometry of the calibration pattern to its observed projections in the images. Finally, all parameters are refined through nonlinear optimization [33, pp. 325–343].

Zhang's method has become the de facto standard for camera calibration in computer vision applications due to its balance of simplicity and accuracy.

Modern Calibration Techniques Recent years have seen the development of various alternative calibration methods:

1D object-based calibration Uses a stick with markers, suitable for calibrating multiple cameras simultaneously [95].

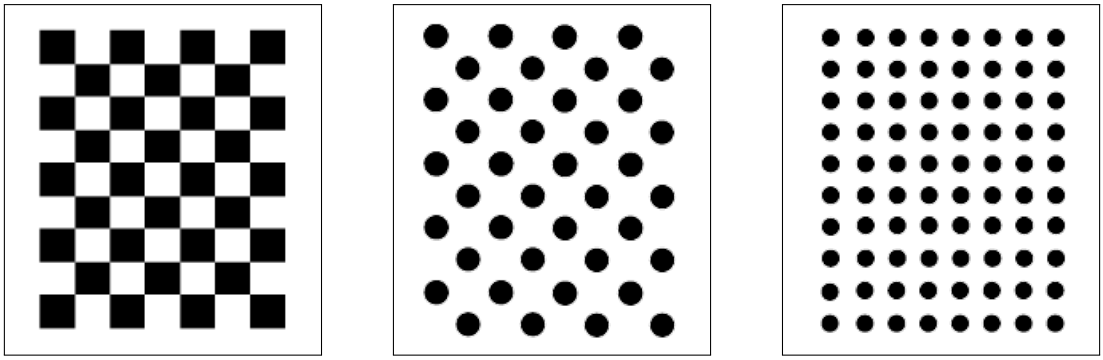
Self-calibration Estimates camera parameters without a known calibration object, useful for unconstrained environments [84].

Active calibration Employs controlled motion or structured light for better precision [100].

These methods offer solutions for specific scenarios where traditional approaches may be impractical, such as in robotics or large-scale multi-camera setups.

Calibration patterns

Calibration patterns, such as checkerboards [94], dot grids, or circles [36], are essential tools in the calibration process. These patterns provide known reference points that the camera system can use to estimate its parameters. During the calibration process, the camera captures multiple images of the pattern from different angles and distances. The known geometry of the pattern allows for the precise calculation of both intrinsic and extrinsic parameters by comparing the observed image points with their corresponding known world points. The choice of calibration pattern can impact the accuracy and ease of the calibration process, with checkerboards being the most commonly used due to their simple and highly detectable corners.



(a) Checkerboard
black-white squares

(b) Circular grid
Evenly-spaced black circles

(c) Dot pattern
Uniform black dots

Figure 2.6: Typical calibration pattern used for single and multi-camera calibration. (Source: MatLab Documentation [82])

Error Analysis and Evaluation

Understanding the accuracy and reliability of camera calibration is important for ensuring the quality of subsequent computer vision tasks [31, pp. 27–39]. Calibration errors can arise from various sources, including imperfections in the calibration pattern, inaccuracies in *feature detection*, and limitations of the calibration algorithm itself [89].

The primary metric for evaluating calibration quality is the reprojection error, which measures the distance between the observed image points and the projected points using the estimated camera parameters [94]. A low average reprojection error typically indicates a good calibration. However, it's important to consider both the mean and distribution of these errors across the image.

Validation techniques often involve using the calibrated camera to measure known distances or angles in a separate test scene. Cross-validation, where a subset of the calibration data is reserved for testing, can also provide insights into the calibration's generalization ability [36].

Robust calibration procedures should account for and minimize these errors to ensure reliable results in practical applications.

Fundamental Concepts in Multi-View Geometry

Before discussing multi-camera calibration, it's useful to understand two key concepts in multi-view geometry:

Homography A homography is a projective transformation that maps points from one plane to another. In the context of camera calibration, it describes the relationship between a planar object (like a calibration pattern) and its image [33, pp. 325–343].

Mathematically, a homography H is a 3×3 matrix that satisfies:

$$x' = Hx \tag{2.7}$$

where $\mathbf{x}$ and $\mathbf{x'}$ are homogeneous coordinates of corresponding points in two images. The homography matrix H has 8 degrees of freedom (despite having 9 elements) due to scale invariance—multiplying the entire matrix by a non-zero scalar results in the same transformation. These 8 degrees correspond to the possible geometric transformations: translation (2), rotation/shear (2), scaling (2), and perspective effects (2). This is why at least 4 point correspondences (each providing 2 constraints) are needed to estimate the homography.

Epipolar Geometry is the intrinsic projective geometry between two views. It is independent of scene structure and depends solely on the cameras' internal parameters and relative pose [33, pp. 239–244]. Figure 2.7 illustrates the key elements of *epipolar geometry*.

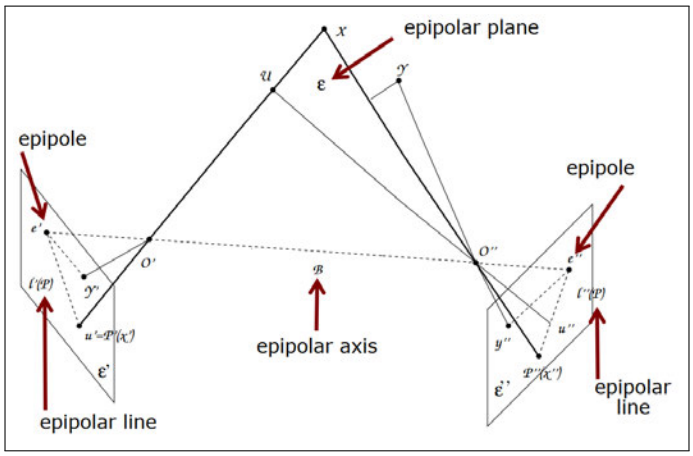


Figure 2.7: Key elements of epipolar geometry. Source (Förstner, Wolfgang [22, p. 563])

The main components of epipolar geometry are:

Projection centers O' and O'': The optical centers of two cameras.

Epipolar plane The plane ϵ containing a 3D point X and the optical centers of both cameras.

Epipolar lines The intersections $l'(P)$ and $l''(P)$ of the epipolar plane with the image planes.

Epipoles The points e' and e'' where the line joining the camera centers (baseline) intersects the image planes.

Baseline The line B connecting the two camera centers O' and O''.

For any 3D point X, its projections $u' = P'(x')$ and $u'' = P''(x'')$ in the two images must lie on the corresponding epipolar lines $l'(P)$ and $l''(P)$. This constraint is fundamental in stereo vision and helps in reducing the search space for corresponding points.

The fundamental matrix F encapsulates this geometry [33, pp. 279–280]. For corresponding points $\mathbf{u}'$ in the first image and $\mathbf{u}''$ in the second image:

$$u''^T F u' = 0 \tag{2.8}$$

This equation represents the epipolar constraint.

The essential matrix E is a special case of F for calibrated cameras [33, pp. 257–262]:

$$E = K'^T F K \tag{2.9}$$

where K and K' are the intrinsic parameter matrices of the two cameras.

It's important to note that while a point u' in one image determines the epipolar line $l''(P)$ in the other image, it does not uniquely determine the position of X along the projecting line. This ambiguity is resolved through triangulation when corresponding points are identified in both images. Understanding these concepts is important for tasks like stereo matching, 3D reconstruction, and multi-camera calibration.

Multi-camera calibration

Multi-camera calibration extends the principles of single camera calibration to systems involving multiple cameras. This process determines the relative positions and orientations of the cameras in addition to their intrinsic parameters [31, pp. 91–106]. It leverages the concepts of homography and epipolar geometry discussed previously to establish relationships between multiple camera views [22, pp. 550–567].

The calibration process typically involves calibrating each camera independently, then establishing correspondences between cameras to compute their relative poses. A global optimization step often refines all parameters simultaneously [94]. The result is a unified model that enables tasks like 3D reconstruction and depth estimation from multiple viewpoints, essential for applications such as mocap and virtual reality.

Camera Pose Estimation

Camera pose estimation [80, pp. 552–558] is the process of determining the position and orientation of a camera relative to a known coordinate system. Two important algorithms

for camera pose estimation are P3P (Perspective-3-Point) [24] and PnP (Perspective-n-Point) [21], [32].

Perspective-3-Point (P3P) The *P3P problem*[2], involves estimating a calibrated camera's pose from three 3D-2D point correspondences, as classified comprehensively by Gao et al. [24]. As the minimal case for pose estimation, it typically yields up to four solutions, necessitating a fourth point for unique determination [22, pp. 513-518].

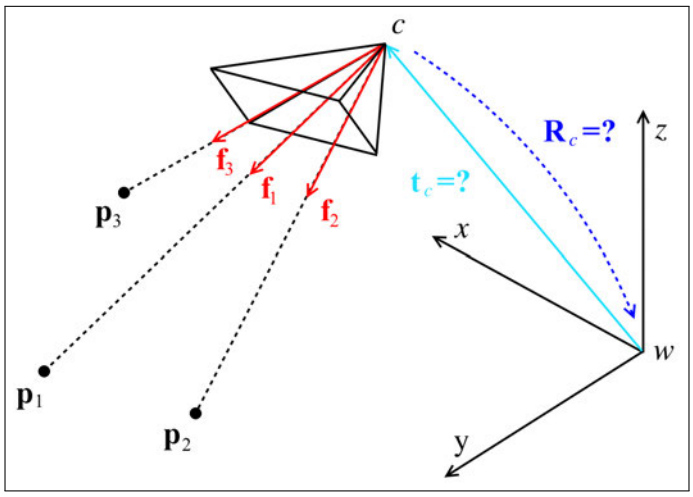


Figure 2.8: Synopsis of the P3P problem. Source (Kneip, Laurent [40])

[2]First formulated mathematically by Grunert in 1841 [28], though the practical implementation for computer vision wasn't realized until much later.

The P3P algorithm can be summarized in the following steps:

Algorithm 1: P3P Algorithm

1: **Input:** 3D world points (X_1, X_2, X_3) and their 2D image projections (x_1, x_2, x_3)
2: Calculate distances between world points: d_{12}, d_{23}, d_{31}
3: Form a system of equations:

$$s_1^2 + s_2^2 - 2s_1 s_2 \cos(\theta_{12}) = d_{12}^2$$
$$s_2^2 + s_3^2 - 2s_2 s_3 \cos(\theta_{23}) = d_{23}^2$$
$$s_3^2 + s_1^2 - 2s_3 s_1 \cos(\theta_{31}) = d_{31}^2$$

where s_i are distances from camera to points, θ_{ij} are angles between image rays
4: Solve the resulting fourth-degree polynomial equation
5: **for** *each real root* **do**
6: | Compute rotation (R) and translation (t)
7: **end**
8: Verify solutions using a fourth point if available
9: **Output:** Camera pose $[R, t]$ that best fits the data

Recent advancements in solving the P3P problem have been proposed in works such as [4], [61], [68], offering improved approaches to enhance its accuracy and efficiency.

Perspective-n-Point (PnP) is a generalization of P3P that uses $n \geq 3$ point correspondences to estimate the camera pose. It's more robust than P3P due to the use of more points. It can also handle cases with $n > 3$ points, improving accuracy and reliability. While traditional PnP algorithms had computational complexity of $O(n^2)$ or higher, modern variants have achieved significant improvements. Notable examples include *Efficient PnP (EPnP)* [44] and *Uncertainty-aware PnP (UPnP)* [41], which both provide a $O(n)$ solution to the PnP-Problem, making them much more practical for real-world applications with large numbers of points.

The general PnP algorithm can be formulated as an optimization problem [22, pp. 513-518]:

$$\text{minimize} \sum_{i=1}^{n} \|x_i - \pi(K[R|t]X_i)\|^2 \tag{2.10}$$

where π represents the non-linear parts of the projection function (including perspective division and distortion effects, as detailed in Equation 2.5), K is the intrinsic camera

matrix, R and t are the rotation and translation to be estimated, and (X_i, x_i) are the 3D-2D point correspondences.

Both P3P and PnP utilize the principles of perspective projection and epipolar geometry discussed earlier (see Section 2.3). They extend these concepts to the practical problem of determining camera pose from known 3D-2D correspondences. These algorithms are fundamental in computer vision and are often used as building blocks in more complex systems, such as *Simultaneous Localization and Mapping (SLAM)*[3] [18] or *Structure from Motion (SfM)*[4] [65].

In practice, robust implementations of these algorithms often incorporate *Random Sample Consensus (RANSAC)*[5] [21] to handle outliers and noise in the point correspondences [80, pp. 408–410].

For this mocap system, PnP might be more suitable due to its ability to handle multiple points, potentially leading to more accurate pose estimation of the camera relative to the captured subject.

Practical Considerations and Future Directions

While the principles of camera calibration are well-established, practical implementations often face challenges such as varying lighting conditions, lens distortions, and the need for frequent recalibration in dynamic environments [70]. Recent advancements in calibration techniques include the use of deep learning for more robust feature detection and the development of self-calibrating systems for autonomous robots [47].

Camera calibration, as discussed, provides the mapping between 3D world coordinates and 2D image coordinates. This mapping is necessary for the next step in mocap systems: pose estimation. With a calibrated camera system, the next step is determining the position and orientation of a subject's body parts in 3D space from 2D image data.

[3] Simultaneous Localization and Mapping (SLAM) is a technique where a system builds a map of an unknown environment while simultaneously keeping track of its location within it.

[4] Structure from Motion (SfM) refers to the process of estimating three-dimensional structures from two-dimensional image sequences that may be coupled with local motion signals.

[5] Random Sample Consensus (RANSAC) is an iterative method to estimate parameters of a mathematical model from a set of observed data that contains outliers.

2.4 Pose Estimation Techniques

Pose estimation in mocap systems relies heavily on the results of camera calibration. The intrinsic and extrinsic parameters obtained through calibration allow pose estimation algorithms to [80, pp. 552–558]:

1. Accurately project 3D body models onto 2D image planes
2. Triangulate 2D joint detections from multiple views into 3D space
3. Compensate for lens distortions when processing image data

These capabilities form the foundation for reconstructing 3D poses from 2D observations, a core task in markerless mocap systems. The geometric relationships established during calibration enable accurate and robust 3D pose reconstruction.

2.4.1 2D Pose Estimation

2D pose estimation is a basic technique in computer vision that aims to locate and identify keypoints or joints of a human body within a two-dimensional image [2]. This process forms the foundation for many advanced applications, including markerless mocap systems.

The primary goal of 2D pose estimation is to create a skeletal representation of a person's pose by identifying the locations of key anatomical landmarks, such as shoulders, elbows, hips, and knees. These landmarks are typically represented as a set of 2D coordinates within the image space [83].

There are two main approaches to 2D pose estimation:

Top-down These start by detecting and localizing entire persons in an image, then estimate the pose for each detected individual. This approach can be more accurate for individual pose estimation but may become computationally expensive as the number of people in the scene increases [93].

Bottom-up These first detect all potential body parts in an image, then associate them to form complete poses. This approach can be more efficient in scenes with many people, as its computational complexity scales better with the number of individuals in the scene [15].

Both typically involve two steps: Identifying the locations of individual body joints and connecting the detected joints to form a coherent skeleton structure.

Recent advancements in deep learning, particularly *CNNs*, have improved the accuracy and efficiency of 2D pose estimation [62]. These methods can learn to identify complex patterns and features in images, making them robust to variations in lighting, clothing, and body shapes.

However, 2D pose estimation still faces several challenges:

- Occlusions, where parts of the body are hidden from view
- Varying lighting conditions and complex backgrounds
- Real-time performance requirements for applications like mocap
- Accurate handling of unusual poses or activities

Despite these challenges, 2D pose estimation serves as a stepping stone towards 3D pose reconstruction in markerless mocap systems. By providing accurate 2D joint locations from multiple camera views, this enables the subsequent 3D *reconstruction process*, detailed in the following sections. Recent work has focused on improving the accuracy and efficiency of multi-person pose estimation in complex scenes [16].

2.4.2 3D Pose Estimation

While 2D pose estimation provides valuable information about human poses in images, markerless mocap systems require full three-dimensional representations of human movement. This is where 3D pose estimation comes into play, serving as a bridge between 2D image analysis and the creation of 3D motion data [74].

3D pose estimation aims to reconstruct the three-dimensional positions of body joints in real-world space, typically using input from multiple 2D views [66]. This process is central to markerless mocap, as it allows for the accurate representation of human movement in three dimensions without the need for physical markers or special suits [53].

From 2D to 3D: Bridging the Dimensional Gap

The transition from 2D to 3D pose estimation is not as simple as adding an extra coordinate to each joint position. It involves geometric reasoning and often requires input from multiple camera views to resolve ambiguities inherent in single-view 2D representations [1].

In 2D pose estimation, the locations of body joints are identified within the image plane. However, this representation lacks depth information—while a joint's location in the image is known, its distance from the camera remains undetermined. This limitation can lead to ambiguities, as different 3D poses can produce similar 2D projections when viewed from a single angle [51].

To overcome this, 3D pose estimation typically relies on multiple camera views. By observing the same pose from different angles simultaneously, the 3D positions of body joints can be triangulated [71]. This is where the importance of multiple camera views becomes evident.

Resolving Ambiguities Multiple views help resolve depth ambiguities that are inherent in single-view 2D representations [66].

Occlusion Handling When a body part is occluded in one view, it may be visible in another, allowing for more complete pose reconstruction [12].

Increased Accuracy The combination of information from multiple views enables more accurate 3D position estimates [79].

Triangulation is an important concept in this transition from 2D to 3D. In the context of 3D pose estimation, triangulation refers to the process of determining a point's 3D coordinates using its projections in two or more 2D images [33, pp. 310–324], [22, pp. 596–606]. The basic principle is as follows:

1. Identify corresponding points (body joints) in multiple 2D images.
2. Use the known camera parameters (obtained through camera calibration) to project rays from each camera through these points.
3. Calculate the 3D coordinates of the point where these rays intersect or come closest to intersecting.

This process, when applied to all identified body joints, enables reconstruction of the full 3D pose from multiple 2D observations.

The following sections examine the methods and challenges involved in 3D pose estimation, and their application in markerless mocap systems.

Multi-View 3D Reconstruction Methods

Multi-view 3D reconstruction is a technique in computer vision that aims to recover the three-dimensional structure of a scene or object from multiple two-dimensional images.

Role of Epipolar Geometry Epipolar geometry (see Section 2.3) provides the geometric constraints that relate corresponding points in different views. These constraints are elemental for efficient and accurate reconstruction of body joint positions across multiple camera feeds.

Triangulation Methods form the core of 3D reconstruction in mocap systems. They determine the 3D coordinates of a point using its projections in two or more 2D images. Common triangulation methods include [80, pp. 558–560]:

- **Direct Linear Transformation (DLT)**: Solves a system of linear equations formed from the projection equations of multiple views [33, pp. 88–91], [80, p. 552]. The *DLT* algorithm formulates the problem as $AX = 0$, where A is a matrix constructed from the camera matrices and image points, and X is the homogeneous 3D point to be determined. While computationally efficient, it can be sensitive to noise. Recent applications have refined this method for specific use cases [99].
- **Midpoint Method**: Computes the midpoint of the shortest line segment connecting the two projection rays. This method is based on the principle that the 3D point lies on the line that passes through the camera center and the image point. The 3D point is estimated as $P_{3D} = \frac{1}{2}(P_1 + P_2)$, where P_1 and P_2 are the closest points on the two projection rays. It's simple but also not optimal in the presence of noise.

- **Optimal Triangulation**: Aims to minimize the reprojection error (see Section 2.3). This method provides more accurate results at the cost of increased computational complexity [34, p. 318]. It seeks to find the 3D point that, when projected back onto the image planes, minimizes the distance to the observed 2D points.

Method	Accuracy	Computational Cost	Robustness to Noise
DLT	Medium	Low	Low
Midpoint	Low	Low	Medium
Optimal	High	High	High

Table 2.1: Comparison of Triangulation Methods

The choice of reconstruction method significantly impacts real-time performance. While optimal triangulation provides the highest accuracy, its computational cost may be prohibitive for real-time applications. In contrast, the DLT or midpoint methods offer faster processing at the expense of some accuracy, making them more suitable for real-time mocap systems.

Optimization Techniques To improve accuracy, especially in the presence of noise, optimization techniques are often employed [22, pp. 643–715]:

Bundle Adjustment This technique simultaneously refines the 3D structure and camera parameters by minimizing reprojection errors [85]. While computationally intensive, it can significantly improve the accuracy of joint position estimates. The optimization problem in *bundle adjustment* can be formulated as:

$$\min_{X,P} \sum_{i,j} d(x_{ij}, P_i X_j)^2$$

where X_j are the 3D points, P_i are the camera parameters, x_{ij} are the observed 2D points, and $d(\cdot,\cdot)$ is the reprojection error.

Levenberg-Marquardt Algorithm Often used in bundle adjustment, this algorithm is particularly effective for solving non-linear least squares problems in 3D reconstruction [45], [50], [59].

The choice of reconstruction method in a markerless mocap system depends on factors such as required accuracy and computational efficiency. Triangulation methods, often

combined with optimization techniques, are frequently employed due to their suitability for reconstructing sparse point sets (body joints) in real-time scenarios.

2.5 Animation Data Generation

Following Grünvogel [29, pp. 427–428], the process of generating animation data begins with 3D reconstruction of keypoints. By applying 3D pose estimation techniques to sequential frames, body joints can be tracked over time. This temporal data enables the generation of animation curves that represent each joint's position and rotation in 3D space throughout the captured sequence.

The conversion of mocap data to character animation presents several challenges. A fundamental issue is proportion mismatch between actor and virtual character—for example, applying motion data from a tall performer to a shorter character can result in unrealistic stride lengths. Another challenge lies in skeletal topology differences, as virtual characters may have different joint configurations than human performers.

To address these problems, Grünvogel describes a two-step conversion process: first converting the capture data to a standardized character format, followed by retargeting to the final virtual character. This approach enables a single mocap sequence to be adapted for multiple characters while preserving essential movement characteristics. The process often requires additional refinement, such as cleaning up foot skating artifacts and removing initial T-pose calibration data.

2.6 Summary

This chapter presented the core concepts and related work essential for understanding markerless mocap systems. It explored camera models and calibration techniques, emphasising their role in mapping 3D world coordinates to 2D image coordinates. The chapter then examined pose estimation methods, progressing from 2D to 3D techniques and highlighting the importance of multiple camera views and triangulation. Finally, it briefly touched on animation data generation, linking 3D pose estimation to movement description. This comprehensive overview establishes the theoretical foundation for the system design to be discussed in the following chapter.

3 Requirements, System Design and Architecture

Building upon the theoretical foundations of markerless mocap, this chapter translates these concepts into a practical system implementation. A requirements framework is first established, followed by the development of a distributed system architecture that addresses the challenges of real-time mocap. The chapter presents both the high-level system design and detailed architectural considerations, culminating in a complete system specification ready for implementation.

3.1 System Requirements

The requirements for the mocap system must be defined to establish a solid foundation. This chapter introduces two types of requirements: *functional requirements (FRs)* and *non-functional requirements (NFRs)*. These requirements stem from both the framework conditions set for this thesis and the specific objectives of the system to be developed [77, pp. 82–117].

ID	Requirement	Description	Verification Method
FR-1	Multi-Camera Setup	System must operate with synchronized stereo setup, extensible to N cameras	Experimental Testing
FR-2	Camera Calibration	Automated calibration process completing within 15 minutes, including intrinsic and extrinsic parameters	Technical Validation
FR-3	2D Pose Estimation	Real-time pose estimation for all major body joints with confidence scores	Quantitative Analysis
FR-4	3D Reconstruction	Accurate reconstruction of 3D joint positions with handling of occlusions	Experimental Testing
FR-5	Data Export	Support for common animation data formats and raw data export	Technical Validation
FR-6	Real-time Processing	Complete processing pipeline operating in real-time	Performance Testing

Table 3.1: Functional Requirements Overview

ID	Requirement	Description	Target Value
NFR-1	Processing Performance	Complete pipeline frame rate	$\geq$ 30 FPS
NFR-2	System Latency	End-to-end processing delay	$\leq 100\ ms$
NFR-3	Reconstruction Quality	Mean reprojection error for triangulated points	$\leq$ 1.0 pixel
NFR-4	Pose Confidence	Minimum confidence score for keypoint acceptance	$\geq$ 95%
NFR-5	Camera Calibration	Mean reprojection error for calibration	$\leq$ 1.0 pixel
NFR-6	Stream Sync	Maximum allowed frame timing delta between cameras	$\leq$ 33 ms

Table 3.2: Non-Functional Requirements Overview

3.1.1 Functional Requirements

Functional requirements define the specific capabilities and services the system must provide. Six core functional requirements (FR-1 to FR-6) have been identified for the mocap system, which are detailed in Table 3.1.

FR-1: Multi-Camera Setup

The system requires a minimum of two synchronized cameras. This stereo configuration serves as the basic setup, with the architecture supporting extension to N cameras while maintaining synchronization across all video streams. This requirement forms the foundation for accurate 3D reconstruction.

FR-2: Camera Calibration

The system implements an automated calibration process that must complete within 15 minutes. This process determines both intrinsic and extrinsic camera parameters [94]. The calibration data is stored persistently to enable quick system initialization in subsequent uses.

FR-3: 2D Pose Estimation

The system performs real-time pose estimation on the video feeds from all cameras simultaneously. Each frame processing yields the 2D coordinates of all major body joints along with their corresponding confidence scores. The system processes and outputs these results for all connected cameras in parallel.

FR-5: Data Export

The system provides functionality to export capture data in common animation formats. While industry standards like Biovision Hierarchy (BHV) [6] and Filmbox (FBX) [3] are commonly used in professional mocap pipelines, the current implementation focuses on raw positional data export in Comma-Separated Values (CSV) format. This format choice serves as a proof of concept while ensuring broad compatibility and easy data access for further analysis. The `CSV` implementation provides all necessary positional and temporal data, while the system's architecture supports future extension to additional animation formats like `BVH` and `FBX` as needed.

FR-6: Real-time Processing

The system must process and output mocap data continuously as it arrives, maintaining temporal consistency between input frames and output poses. The processing pipeline must operate synchronously from video input through to 3D pose output.

Implementation Scope

The system's architecture allows for future expansion, while currently focusing on single-person detection and tracking within the defined capture volume.

These functional requirements define the core capabilities needed for a real-time markerless mocap system. Table 3.1 summarizes the requirements and their verification methods.

3.1.2 Non-Functional Requirements

Non-functional requirements define system performance metrics and quality attributes. Table 3.2 specifies the target values for these requirements.

NFR-1: Processing Performance

The system must achieve a consistent frame rate of at least 30 frames per second (FPS), meaning each complete processing cycle must complete within approximately 33.30 ms. This performance requirement ensures smooth mocap and maintains the real-time capability defined in FR-6.

NFR-2: System Latency

The end-to-end processing delay between physical movement and digital representation must not exceed 100 ms. This latency requirement is required for maintaining temporal accuracy in the captured motion data.

NFR-3: Reconstruction Quality

The 3D reconstruction process must achieve a mean reprojection error of less than 1.0 pixel for triangulated points. This metric directly influences the accuracy of the final mocap data.

NFR-4: Pose Confidence

The system requires a minimum confidence score of 95% for *keypoint acceptance* in the 2D pose estimation stage. This threshold helps ensure the reliability of the detected joint positions.

NFR-5: Camera Calibration

The camera calibration process must achieve a mean reprojection error of less than 1.0 pixel. This accuracy in camera parameter estimation is required for precise 3D reconstruction.

NFR-6: Stream Synchronization

The maximum allowed frame timing delta between camera streams must not exceed 33 milliseconds. This synchronization requirement ensures consistent temporal alignment of the captured motion data.

3.2 System Overview

The mocap system uses a distributed, *service-oriented architecture* to implement the established requirements. The architecture addresses three main challenges: processing distribution across capture *nodes*, real-time performance constraints (NFR-1, NFR-2), and support for variable camera configurations (FR-1).

While the system architecture supports deployment across multiple processing nodes, this distributed design also enables flexible *deployment configurations* based on available infrastructure. The modular service architecture allows components to be distributed across separate nodes or consolidated as needed, while maintaining the logical separation and scalability benefits of the design.

The system implements *structural patterns* for component interactions and data flow. These patterns define approaches for inter-component communication and system *state management*, with particular consideration for the stream synchronization requirements specified in NFR-6.

The architectural documentation follows the C4 model methodology [11], examining the system through successive levels of detail: *system context*, *containers*, and *components*, a *deployment view* is included to address the physical distribution aspects of the system, though this view extends beyond the traditional C4 model. The following sections analyse these architectural levels and discuss the patterns and design decisions that form the system.

This structured decomposition shows how architectural decisions support both functional and non-functional requirements. The architecture focuses on meeting system latency targets (NFR-2), frame rate requirements (NFR-1), and processing pipeline performance (FR-6). Component boundaries are clearly defined while maintaining the interaction efficiency needed for real-time processing.

3.2.1 Context View

The system context defines the boundaries of the mocap system and maps its interactions with external actors and systems. This view establishes the system scope and identifies external dependencies and data flows.

The context includes two types of actors: human operators and hardware interfaces. The Mocap Operator handles system administration, including calibration procedures, pipeline configuration, and operational control. The Performer represents the mocap subject. These actors interact with the system through different interfaces: the Operator uses control functions while the Performer's movements generate the input data. The

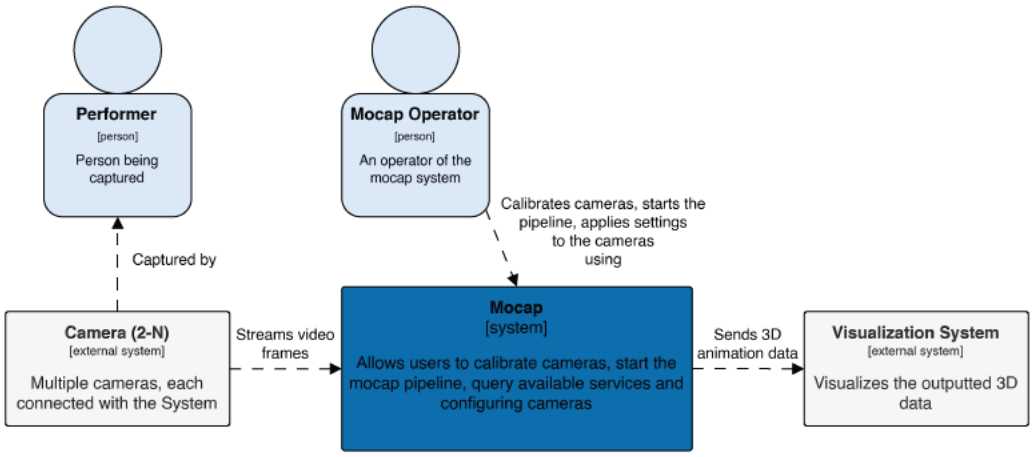


Figure 3.1: Context view of the mocap system.

hardware interface consists of multiple cameras, requiring a minimum of two units for stereo capture, as specified in FR-1. Each camera streams video frames to the mocap system. The system performs calibration according to FR-2, manages the processing pipeline (FR-6), and processes the captured data into motion information.

The system boundary ends at a data interface that provides the processed motion data in standard formats, as required by FR-5. This interface specification maintains interoperability between the mocap system and external applications that consume the data.

3.2.2 Container View

The container view breaks down the system into its main architectural units and defines their interactions. The system distributes functionality across three containers: a `Console Application` for system control and pipeline management, a `Discovery Service` for system configuration, and `Camera Services` for video capture and processing.

The `Console Application` serves as the system's control point, coordinating pipeline execution and system management. The `Discovery Service` maintains system topology through service registration and monitoring. Each `Camera Service` processes video data from its associated capture device independently.

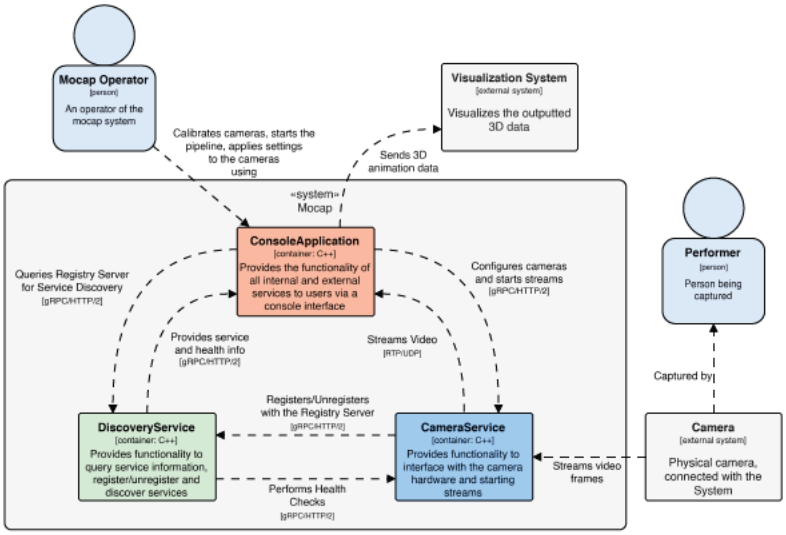


Figure 3.2: Container view of the mocap system.

The containers communicate through *service interfaces* that support the performance requirements specified in NFR-1 and NFR-2. Each `Camera Service` operates on hardware connected to its capture device, enabling parallel processing and reducing data transfer bottlenecks.

The `Discovery Service` runs as a separate container to handle system configuration state and service availability. This separation allows for dynamic service registration without impacting the capture pipeline's performance. The additional communication overhead stays within the latency requirements defined in NFR-2 when operating in local networks.

The system scales horizontally through `Camera Service` container addition. New capture nodes join through service registration, with the `Discovery Service` tracking the dynamic system topology. This design allows processing capacity to scale linearly with added capture nodes.

This *distributed architecture* requires specific state management and synchronization mechanisms to maintain consistency across containers while meeting real-time processing requirements (NFR-1, NFR-6). These requirements guide the component-level design decisions detailed in the following section.

3.2.3 Component View

The component view examines the internal architecture of each container and describes how components work together to fulfil the container responsibilities. Each container: `Camera Service`, `Console Application`, and `Discovery Service` implements specific aspects of the system's functionality.

Camera Service Architecture

The `Camera Service` architecture (Figure 3.3) consists of four main components, handling video capture and streaming. The `Communication Interface` defines the container's external boundary, managing service registration and request processing. `Device Control` operates the camera hardware and maintains device state. The `Configuration Manager` handles system parameters and their persistence, while the `Stream Manager` coordinates video data transmission. The components follow a *layered architecture* pattern. The `Communication Interface` separates external protocols from internal processing.

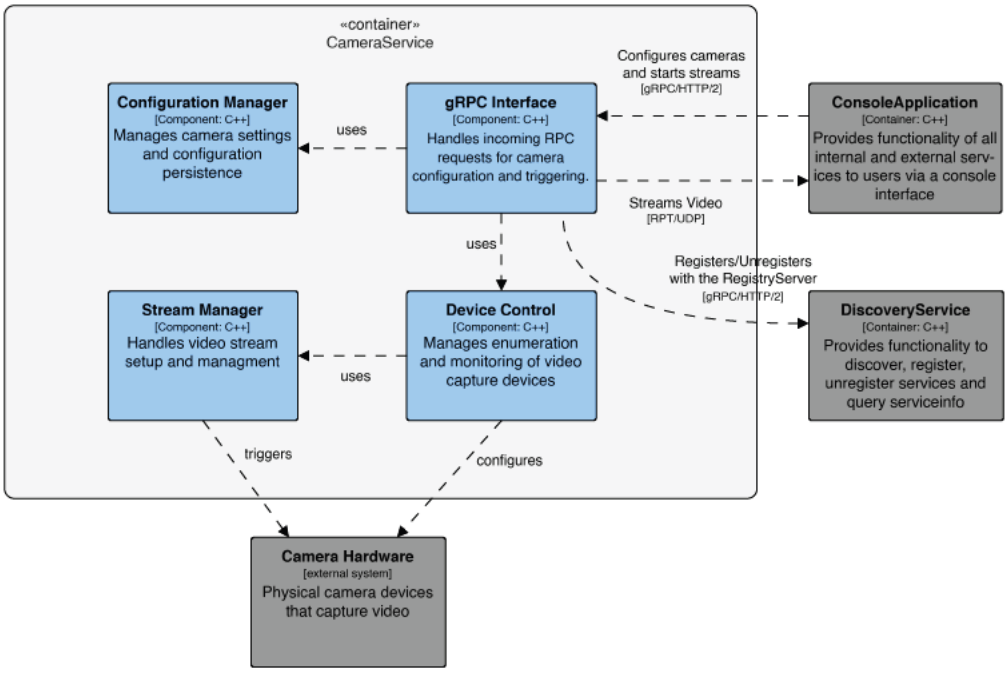


Figure 3.3: Component view of the camera service.

`Device Control` abstracts hardware operations, allowing the `Stream Manager` to handle data processing independently of the underlying hardware. This separation enables modifications to device handling or streaming implementations without affecting other components.

Console Application Architecture

The `Console Application` architecture (Figure 3.4) implements the system control and processing pipeline. The Console Interface provides the operator control point for system management. `Camera Calibration` executes the calibration process defined in FR-2, while the `Mocap Pipeline` handles mocap processing according to FR-3 and FR-4. The `Synchronization` component aligns multi-camera data streams to meet the timing requirements specified in NFR-6. The architecture defines data flows between components through dedicated interfaces. The `Console Interface` directs calibration and pipeline operations, while the `Synchronization` component provides timing services to both processes. This organization enables the system to meet the latency requirements (NFR-2) while maintaining processing performance (NFR-1).

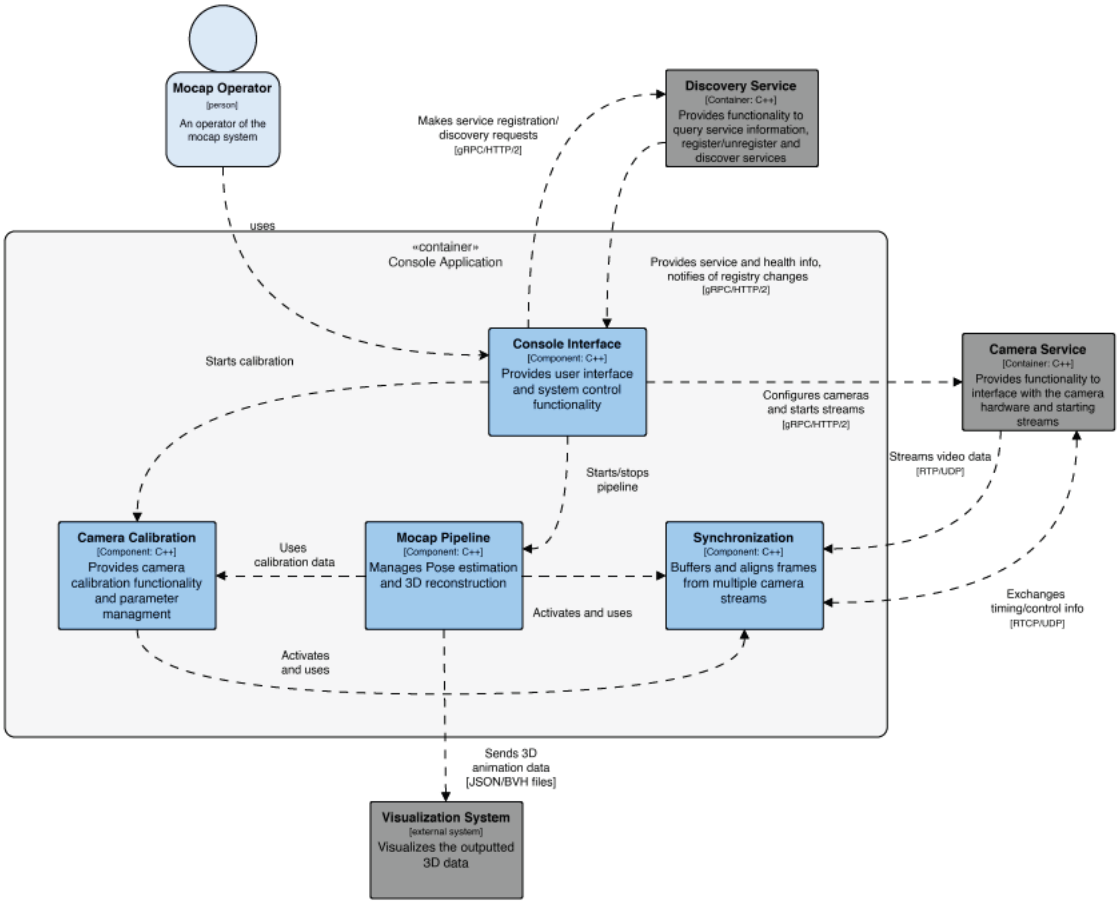


Figure 3.4: Component view of the console application.

Discovery Service Architecture

The `Discovery Service` (Figure 3.5) implements service registry management with two components. The `Communication Interface` combines service endpoint functions with registry operations, and the `Health Monitor` tracks service availability states. The minimal component structure reflects the focused role of this container in system coordination.

Component Integration

System state management occurs through defined communication patterns between components across containers. The `Camera Service` components report state changes through their `Communication Interface`, enabling the `Console Application` to track

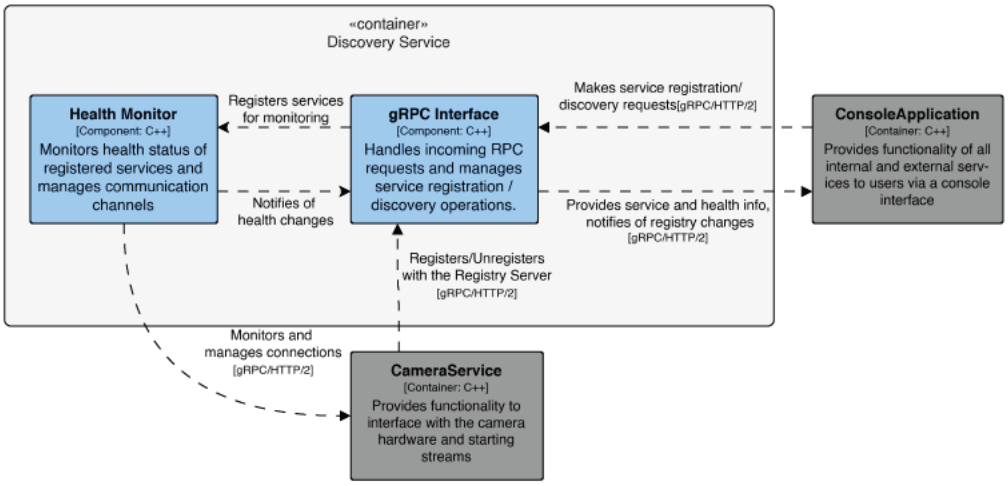


Figure 3.5: Component view of the discovery service.

system status. The `Discovery Service` propagates availability updates for system-wide adaptation to configuration changes.

The component organization supports distributed processing requirements while separating component responsibilities. Each component operates within defined boundaries and interacts through interfaces that handle both synchronous operations and asynchronous state updates.

3.2.4 Architectural Patterns and Design Details

The system architecture addresses distributed processing, real-time requirements, and component coordination through several key design approaches.

Service Communication Patterns

The system uses a *service-oriented architecture* for *inter-container communication* (Figure 3.6). This design establishes methods for service registration, discovery, and interaction. The interfaces separate control operations from data streaming, which allows independent scaling of these functions.

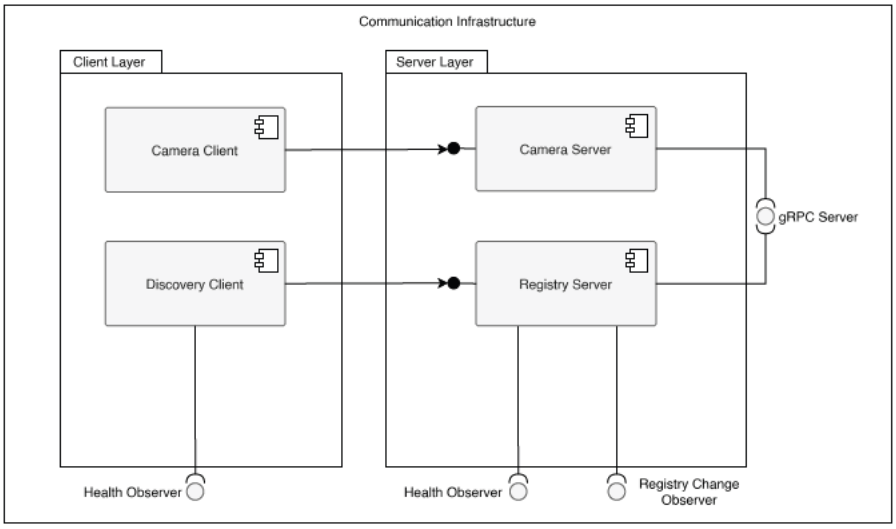


Figure 3.6: Communication infrastructure, showing the layered service design and interfaces for health monitoring and registry management.

Callback-Based State Management

The system manages state through a *callback-based observer* structure (Figure 3.7). Services maintain their local state and notify other components of changes through callbacks. This approach enables the synchronization required by NFR-6 while allowing services to operate independently.

Pipeline Structure

The mocap pipeline divides processing into distinct stages: frame acquisition, pose estimation, reconstruction, and data output (Figure 3.8). This separation helps meet FR-6 while maintaining the performance requirements of NFR-1. The output stage provides an interface for external systems to receive the processed motion data, fulfilling the data export requirement specified in FR-5.

The pipeline's performance can be expressed as:

$$FPS = \frac{1}{t_{\text{capture}} + t_{\text{2D\_pose}} + t_{\text{triangulation}} + t_{\text{output}}} \tag{3.1}$$

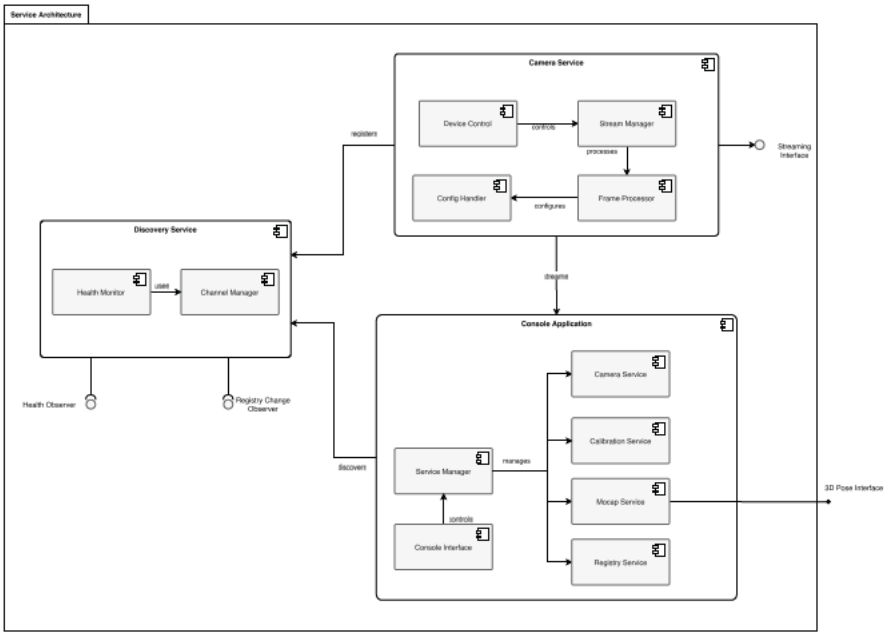


Figure 3.7: Service interaction structure, showing communication through health monitoring and registry mechanisms.

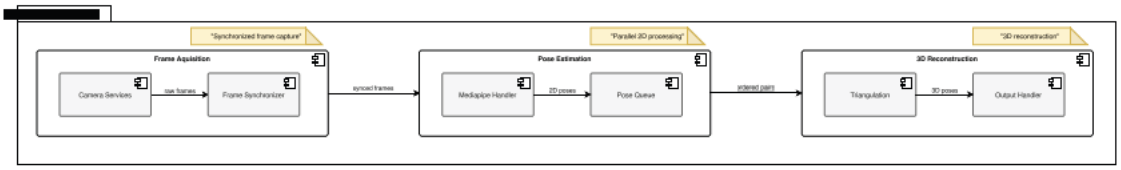


Figure 3.8: Mocap pipeline, showing processing stages and data flow between components, including the output interface for external systems.

Each term represents a stage's processing time, with the total bounded by NFR-1 and NFR-2. The t_{output} term accounts for the time needed to prepare and deliver data through the output interface.

Resource Management

Each service manages its local resources while participating in system-wide coordination. This distributed approach supports the scalability needs of FR-1 and the performance requirements of NFR-1.

3.2.5 Architectural Decisions

This section documents the main decisions that shaped the system architecture, explaining how each addresses specific requirements and technical constraints.

Service Distribution

The system distributes camera processing across independent services, as outlined in the container view. This decision reflects the physical setup where each camera connects to its own processing hardware. While this distribution improves system scalability through independent nodes, it requires additional coordination for synchronization (NFR-6) and state management.

Communication Architecture

The system requires a *communication architecture* that supports both *request-response* interactions and *continuous data streaming*. The chosen architecture must enable:

- Service discovery and registration
- Efficient video data transmission
- System state updates

These requirements influence the selection of specific protocols and implementations, detailed in Chapter 4.

Pipeline Structure

The pipeline structure, introduced in the previous section, separates processing stages based on data dependencies. This separation allows parallel execution where possible and enables stage-specific optimizations while maintaining the interfaces between stages. This approach directly supports the real-time processing requirement (FR-6) and performance targets (NFR-1).

Synchronization Approach

The system uses timestamp-based synchronization instead of hardware triggers. This decision prioritizes system flexibility and reduces hardware complexity, though it accepts lower precision than hardware synchronization. The approach allows adding capture nodes without hardware modifications, supporting the system's scalability goals.

State Distribution

The architecture implements a hybrid approach to state management that distributes state across services rather than employing centralized state management. For inter-container communication, this distribution introduces *eventual consistency* in system-wide configuration and health states, thereby eliminating single points of failure and supporting the performance requirements (NFR-1, NFR-2) through reduced communication overhead. Within containers, the system maintains *strong consistency* through callback-based mechanisms and synchronous operations, particularly important for real-time processing pipelines and stream synchronization requirements specified in NFR-6. This dual consistency model enables the system to balance reliability and performance demands, applying eventual consistency for system management functions while ensuring strict timing and synchronization requirements are met for the mocap processing pipeline.

3.2.6 Deployment View

The deployment view describes the physical distribution of architectural components across hardware infrastructure. While the container view established the logical service distribution, this view addresses the specific hardware and network requirements.

Hardware Topology

The system operates on a distributed infrastructure consisting of processing nodes and network components (Figure 3.9). Each camera requires a dedicated processing node with direct device connectivity. A local area network connects all nodes, supporting both control communication and video streaming.

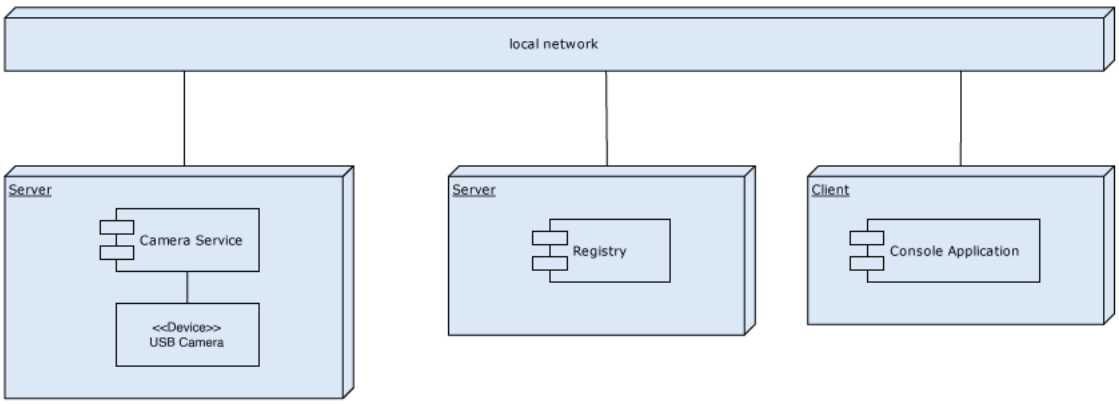


Figure 3.9: Physical deployment of system components showing hardware nodes, network connections, and service distribution.

Resource Requirements

Processing nodes must support real-time video processing while maintaining frame buffering capabilities. Network infrastructure must accommodate the bandwidth requirements of multiple video streams while maintaining latency characteristics within system constraints. Local storage on processing nodes maintains configuration and calibration data.

Network Configuration

The network topology follows a star configuration with a central switch. This configuration supports:

- System-wide Network Time Protocol (NTP) synchronization through multicast
- Direct communication paths between all nodes
- Service discovery within the network segment

The deployment topology directly reflects the service architecture while addressing physical constraints of camera connectivity and processing requirements.

3.3 Summary

This chapter defined the system requirements and architectural design of the mocap system. The requirements analysis specified functional requirements including multi-camera setup (FR-1), calibration processes (FR-2), and real-time processing capabilities (FR-6). The non-functional requirements established concrete performance targets for latency (NFR-2), processing rates (NFR-1), and reconstruction quality (NFR-3).

The architectural design implements these requirements through distributed services. The documentation followed the C4 model, starting from system context and progressing through containers to components. This systematic decomposition showed how each architectural layer implements specific requirements. The distributed nature of the system influenced several architectural decisions, particularly in state management and service communication.

Additionally, the deployment view detailed the physical distribution of components across hardware infrastructure, addressing the specific requirements of camera connectivity and network communication. The design establishes clear interfaces between components and defines their interactions across the distributed system. With the architecture and requirements defined, Chapter 4 presents their implementation, focusing on how the system achieves its real-time processing targets and maintains reliable operation.

4 Implementation

This chapter presents the implementation of the mocap system based on the architecture and requirements defined in Chapter 3. The implementation covers the complete processing pipeline from video capture through pose estimation to 3D reconstruction. The implementation approach and technology selections are presented first, followed by detailed examinations of each system component.

The implementation realizes a complete mocap processing pipeline. Key capabilities include multi-camera operation, calibration, real-time pose estimation, and 3D reconstruction. The system targets 30 FPS processing performance with under 100 ms latency, while maintaining calibration accuracy within 1.0 pixel reprojection error and 95% *pose confidence*.

4.1 Implementation Strategy and Technology Selection

Implementing a distributed mocap system with specific performance requirements (NFR-1, NFR-2) necessitates appropriate selection of frameworks and libraries. This section examines the technical decisions made to support both the functional requirements and performance targets established in Chapter 3.

4.1.1 Video Processing Framework

The implementation uses GStreamer [30] for video stream processing, selected after evaluating several frameworks. FFmpeg [20] provides simpler APIs but lacks the needed pipeline reconfiguration capabilities. OpenCV's [64] `VideoCapture` component offers direct integration with computer vision functions, but exhibits limitations in network streaming capabilities.

GStreamer's pipeline architecture enables multi-camera operation through configurable processing stages from network reception through decoding to frame delivery. The framework's native *Real-time Transport Protocol (RTP)/Real-Time Transport Control Protocol (RTCP)* implementation provides stream synchronization with quality-of-service monitoring. Additionally, GStreamer's plugin architecture supports hardware-accelerated video decoding, which helps meet the processing performance requirements (NFR-1) when handling multiple video streams.

4.1.2 Pose Estimation Technology

For 2D pose estimation, the implementation uses MediaPipe's `BlazePose` [5], [48], selected after evaluating several approaches. OpenPose [14] offers deep integration possibilities but requires higher computational resources. OpenCV's `Deep Neural Network (DNN)` module presents flexibility in model selection and potential integration with existing OpenCV functionality. However, this approach would necessitate extensive model integration and validation work beyond the scope of this thesis.

MediaPipe's `BlazePose` implementation provides real-time pose estimation with integrated confidence scoring for pose validation. This established solution enabled focusing development efforts on the core mocap functionality rather than pose estimation model development.

4.1.3 Development Framework Selection

The implementation uses C++ 20, leveraging modern language features including `concepts`[1], `coroutines`[2], and expanded `constexpr`[3] support. These features provide compile-time guarantees and abstractions for concurrent programming, supporting the system's real-time processing requirements.

Several established libraries form the implementation foundation. OpenCV 4.8 implements the camera calibration (FR-2) and 3D reconstruction (FR-4) functionality

[1] Concepts provide a way to specify constraints on template parameters, enabling better error messages and more explicit interface requirements for generic code.

[2] Coroutines provide language support for cooperative task execution and asynchronous programming, allowing functions to suspend and resume execution while maintaining their state.

[3] Constexpr enables compile-time computation and validation, allowing the compiler to evaluate expressions and functions at compile time rather than runtime, improving both performance and type safety.

through computer vision operations, image processing, and geometric computations. The Boost libraries [7] provide essential functionality across multiple system aspects. `Boost.Asio` handles asynchronous I/O operations, while `Boost.Interprocess` manages *shared memory* operations. Concurrent data structures utilize `Boost.Lockfree`, and additional components such as `Boost.UUID` and `Boost.Process` support service identification and process management, respectively.

Service communication employs gRPC [27], implementing type-safe *Remote Procedure Calls (RPCs)* between distributed system components. The use of protocol buffer definitions maintains clear interface boundaries while supporting the distributed processing requirements (FR-1). Protocol buffer [26] (`.proto`) files define service interfaces and message types, from which gRPC automatically generates type-safe client and server code[4].

This framework selection is designed to support target performance through a modular architecture. The combination of libraries enables a pipeline focused on real-time processing and accurate reconstruction while maintaining system maintainability through well-defined interfaces.

4.2 Code-Level Architecture

Following the C4 model progression established in Section 3.2, this section examines the code-level implementation of components through their classes, interfaces, and relationships. This examination connects the architectural design to its concrete implementation.

4.2.1 Common Architectural Elements

The implementation realizes the service-oriented architecture from Section 3.2 through several foundational elements supporting system operation.

[4] Generated code includes service interfaces, stub implementations, and serialization logic, ensuring type safety and consistency across system components.

Base Communication Infrastructure

The system implements distributed service communication through a template-based gRPC infrastructure, as shown in Figure 4.1. The `GrpcServer` template class establishes service endpoints through lifecycle management. Each server implementation provides health monitoring, request processing, and shutdown operations through inheritance hierarchies. The health monitoring system performs periodic status checks of service availability.

A complementary `GrpcClient` template implements both synchronous and asynchronous communication modes. The client infrastructure enables type-safe RPC method invocation with error handling and status propagation. Through *template specialization*[5] and *concept requirements* (`DerivedFromGrpcService`), the implementation ensures consistent behaviour across services, supporting service discovery and monitoring functions.

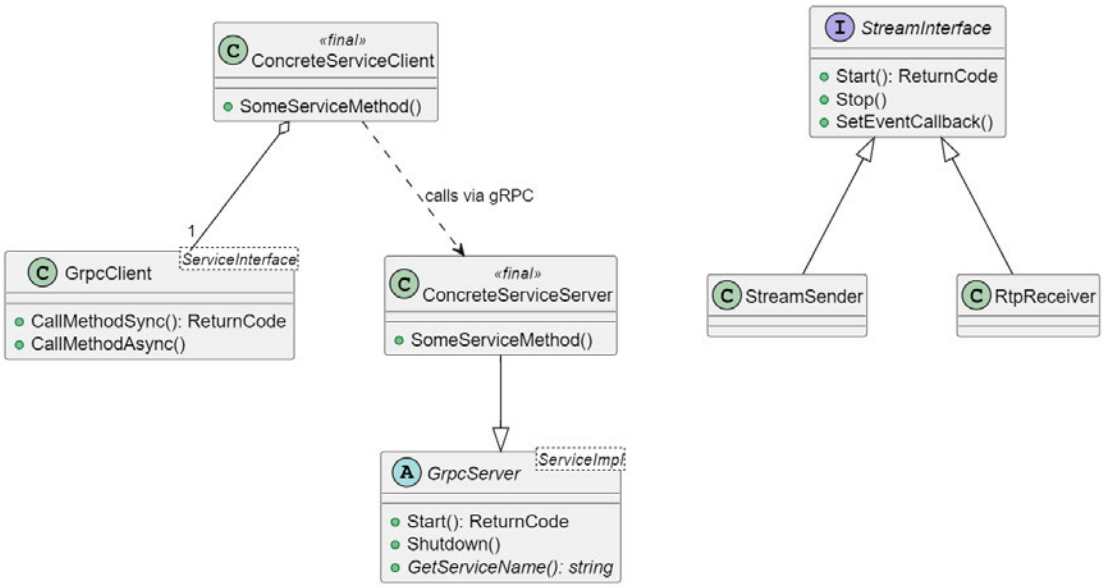


Figure 4.1: Core communication infrastructure, showing base gRPC templates and stream interface hierarchy. Templates enforce consistent service behaviour while allowing specialized implementations.

[5] Template specialization in C++ enables custom implementations of a template for specific types, while keeping the generic version for others. It allows both full specialization (all template parameters specified) and partial specialization (some parameters fixed), enabling optimized or type-specific behavior while maintaining template flexibility. Example: `std::vector<bool>` uses bit-level storage instead of bytes.

Stream Processing Infrastructure

Video stream handling uses a `StreamInterface` abstraction (Figure 4.1) that defines stream management operations. This interface specifies three operations: stream initialization with configuration parameters, stream termination, and event callback configuration for status monitoring.

Two classes implement the streaming infrastructure: `StreamSender` and `RtpReceiver`. The `StreamSender` class manages video streams using GStreamer [30] pipelines for video capture and RTP transmission. This implementation includes frame buffering and timestamp management with configurable stream parameters.

The `RtpReceiver` class handles incoming video streams, implementing video capture and RTP transmission with synchronized streams. The implementation includes monitoring systems for stream health and network statistics to maintain performance targets.

Both implementations use GStreamer pipelines with buffer management and timestamp synchronization to meet the real-time processing requirement. The implementation includes event callbacks for system-wide status monitoring.

Error Handling and Status Propagation

The error handling implementation uses a `ReturnCode` enumeration with categorized ranges. General operations occupy the range from 0 to 999, while communication-related errors use 2000 to 2999. Streaming operations utilize the range from 7000 to 7999, and hardware-specific errors are assigned values between 9000 and 9999. This structured categorization integrates with a logging system supporting multiple verbosity levels (error, warning, info, debug, and trace) and source location tracking. The implementation maintains consistent error propagation through `ErrorCategory` and Severity classifications.

Observer Pattern Implementation

The implementation uses two observer interfaces, as depicted in Figure 4.2: `HealthObserver` for service health monitoring and `RegistryChangeObserver` for service discovery

events. The `HealthObserver` interface monitors service health status through `OnServiceHealthy` and `OnServiceUnhealthy` callbacks, while `RegistryChangeObserver` tracks service registration changes.

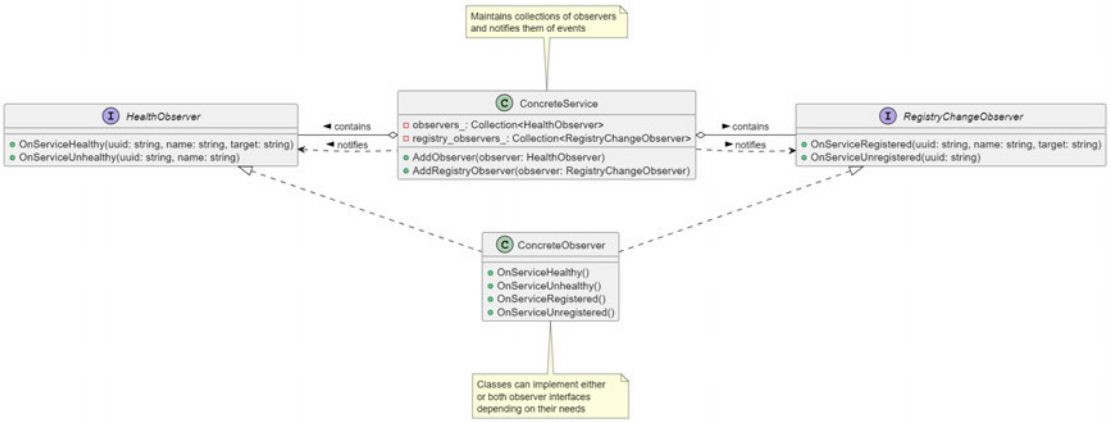


Figure 4.2: Observer pattern implementation, showing health monitoring and registry change notifications. The pattern enables loose coupling between services while maintaining system-wide state awareness.

The `HealthMonitor` class implements thread-safe observer management using periodic checking with configurable intervals. This implementation maintains stream synchronization through consistent health status monitoring.

Resource Management

The implementation employs C++ 20 resource management through *Resource acquisition is initialization (RAII)*[6] principles and *smart pointers*[7]. Unique ownership semantics for service implementations, channels, and monitors are enforced by `std::unique_ptr`. The implementation uses `std::shared_ptr` where shared ownership is needed, particularly for callbacks and shared resources.

[6] RAII (Resource Acquisition Is Initialization) is a C++ programming principle where resource management is tied to object lifetime. Resources (like memory, files, or locks) are acquired during object construction and automatically released in the destructor, ensuring proper cleanup even when exceptions occur.

[7] Smart pointers are C++ objects that act like regular pointers but automatically manage the memory of the objects they point to. `std::unique_ptr` enforces exclusive ownership and automatically deletes its object when going out of scope, while `std::shared_ptr` allows multiple owners and deletes the object when the last owner is gone.

This resource management extends to GStreamer pipeline elements and gRPC service instances, ensuring proper clean-up during error conditions. The implementation prevents resource leaks and maintains clear ownership semantics throughout the distributed architecture.

4.2.2 Service Structure Implementation

The service architecture implementation follows the distributed design outlined in Section 3.2, enabling multi-camera operation with 30 FPS processing performance. Each service implements specific domain functionality while following consistent communication patterns.

Camera Service Implementation

The Camera Service implementation comprises four components as discussed in Section 3.2.3: gRPC interface, device control, configuration management, and stream management. Figure 4.3 shows these components and their relationships.

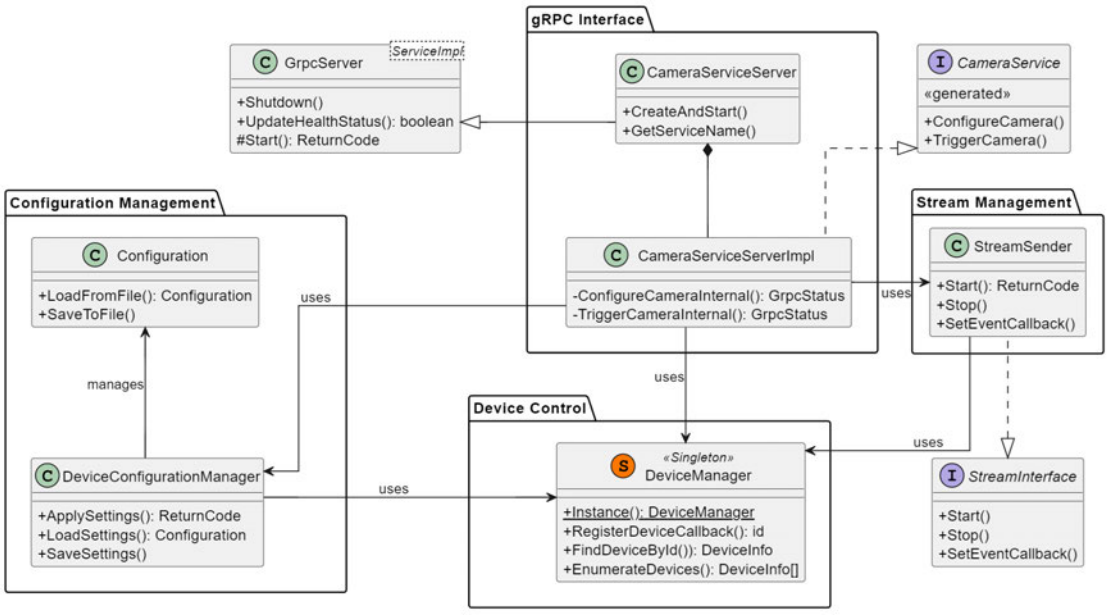


Figure 4.3: Camera Service implementation structure showing the relationships between core components. The `GrpcServer` template provides the foundation for the service implementation, while specialized components handle device control and streaming.

The `CameraServiceServer` inherits from the `GrpcServer` template to implement communication infrastructure, while `CameraServiceServerImpl` implements the generated service interface. This separation maintains distinct boundaries between communication and service-specific functionality.

A `DeviceManager` singleton implements device control, providing access to camera hardware through an abstraction layer. This encapsulation enables runtime device detection and management while isolating device-specific operations.

Video stream transmission uses `GStreamer's` pipeline architecture through the `StreamSender` class. The implementation captures video from industrial cameras via the *Video4Linux2 (V4L2)* interface, configuring streams according to device parameters. Figure 4.4 shows the pipeline architecture designed for low-latency streaming.

The pipeline begins with a `v4l2src` element for raw frame capture, using a `capsfilter` for format constraints. Format adaptation is handled by the `videoconvert` and `videoscale` elements, while the `videorate` element maintains frame timing consistency. H.264 compression[8] occurs through the `x264enc` element with *zero-latency* configuration, followed by `rtph264pay` for RTP packetization. The `udpsink` elements manage RTP video transmission and RTCP session control. The implementation captures timestamps at the moment of frame acquisition and synchronizes them with the hardware clock to minimize processing delays. A `rtpbin` element manages the RTP session, handling transmission timing and RTCP feedback.

Configuration and stream management interact through the observer pattern, where `StreamSender` monitors device states via `DeviceManager` callbacks. This supports both stream management and camera parameter reconfiguration.

Discovery Service Implementation

The `Discovery Service` implementation described in Section 3.2.3 implements service registration and health monitoring through a layered architecture, shown in Figure 4.5.

`RegistryServerImpl` implements both `HealthObserver` and `RegistryChangeObserver` interfaces to handle service health and registration signals. This implementation provides a single registration point while using observer interfaces for component decoupling.

[8] H.264/AVC (Advanced Video Coding) is a video compression standard that provides high-quality video at substantially lower bitrates than previous standards, making it suitable for real-time streaming applications.

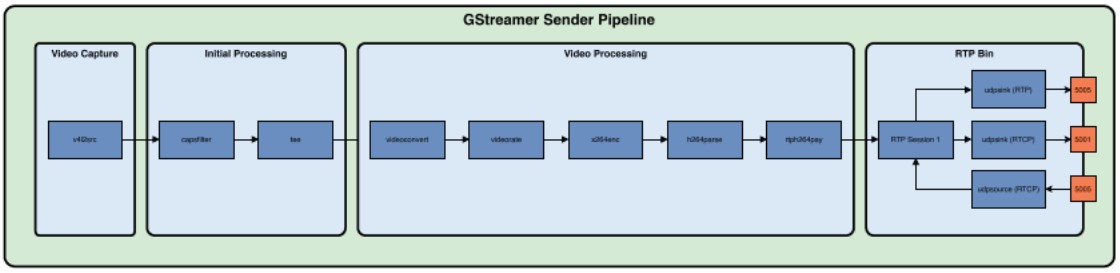


Figure 4.4: GStreamer pipeline structure for video streaming, showing element connections and data flow. The pipeline handles RTP video data transmission, RTCP control traffic and H.264 encoding

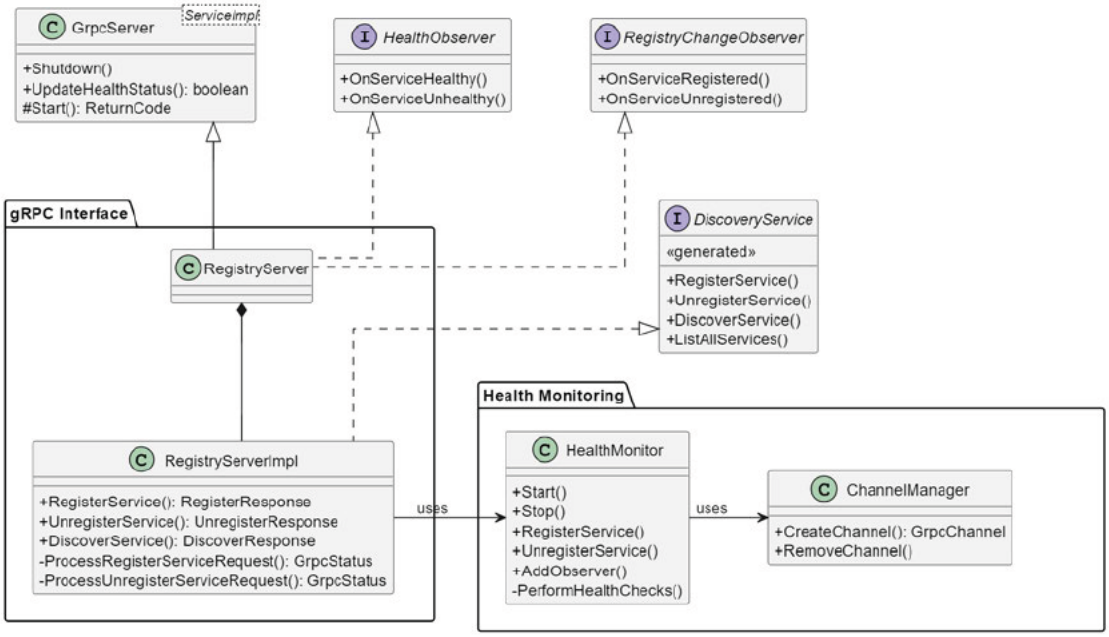


Figure 4.5: Discovery Service implementation, showing the integration of registry management and health monitoring components. The observer pattern enables consistent state propagation across the system.

Health monitoring uses the `HealthMonitor` component to implement periodic health checks, maintaining stream synchronization through consistent service state tracking.

Console Application Implementation

The console application implements system control functionality, addressing requirements for service management, command processing, and operation coordination. Following Section 3.2.3, the implementation comprises four components.

Console Interface Implementation The console interface implements command-line control through command processing and service management. Figure 4.6 shows the implementation structure based on the `CliRunner` class, which implements system control according to Section 3.1.2.

Command processing uses a three-phase parsing system. A regex-based parser extracts service names, commands, and parameter flags from input. This implementation supports both single-service commands and system-wide operations through a flag mechanism, allowing operation targeting at both instance and service type levels.

The `ServiceManager` class implements service management by extending the `RegistryChangeObserver` interface. A thread-safe registry tracks local and remote services using *UUID-based* identification. The implementation uses RAII principles and smart pointers for resource management.

The command execution process follows a three-stage validation approach. First, the system validates service existence and availability. Second, it verifies command support by the target service. Third, it checks parameter flag completeness and format. This separation enables specific error handling for each validation step.

A structured error handling system distinguishes between command syntax, service availability, and execution errors, providing specific responses for each case. The output system generates error-specific feedback for command correction, enabling quick error recovery. This combination of validation and feedback structures maintains real-time processing with synchronized streams.

Several control mechanisms ensure reliable system operation. All services use a consistent command syntax, supporting both instance-specific and broadcast execution modes. Continuous status monitoring works alongside service discovery to track system state, while a context-based help system adapts documentation to current service availability.

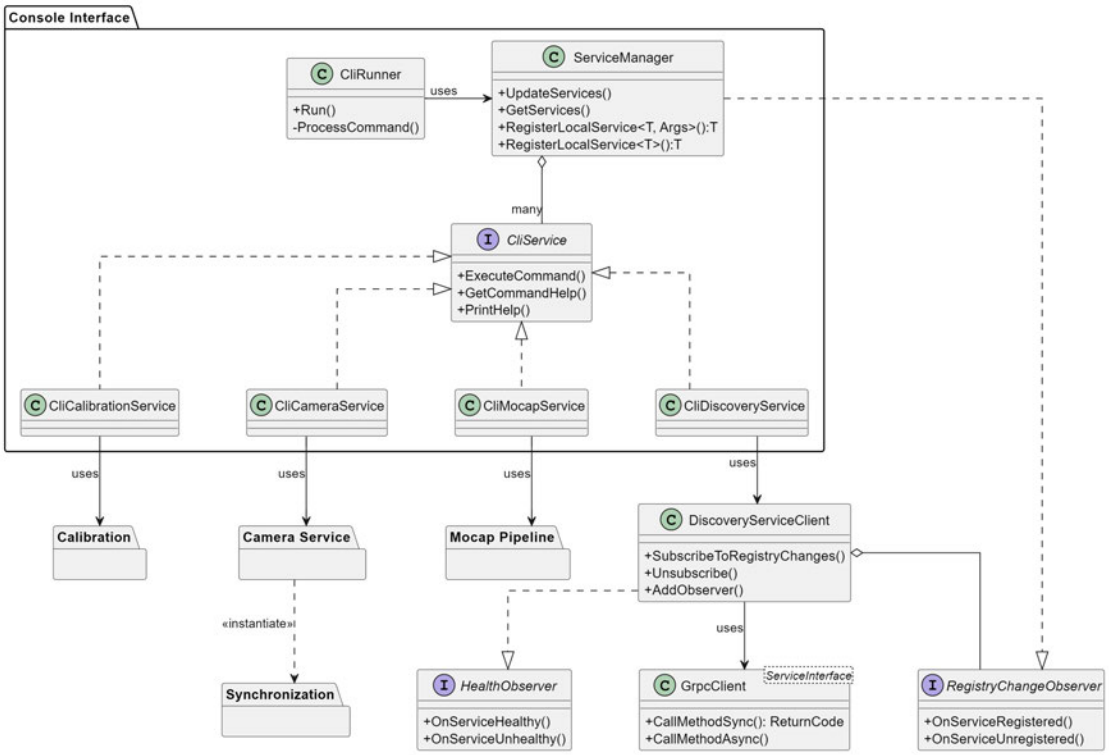


Figure 4.6: Console Application structure showing command processing and service management components. The `ServiceManager` maintains service connections, while specialized services handle specific system functionality.

This combination of validation and feedback structures supports the real-time processing requirement (FR-6) while maintaining the stream synchronization targets (NFR-6).

Synchronization Implementation Frame acquisition and temporal alignment across multiple video streams are handled by the synchronization component, which targets synchronization within 33 ms. Figure 4.7 shows the classes and relationships in this component. The `RtpReceiver` handles network video stream reception, while the `FrameSynchronizer` performs temporal alignment of frames across cameras.

Video Stream Reception Video stream reception uses GStreamer's pipeline architecture. The `RtpReceiver` class processes multiple video streams, each identified by RTP and RTCP port configurations. Each stream uses a dedicated pipeline for video data processing, as shown in Figure 4.8.

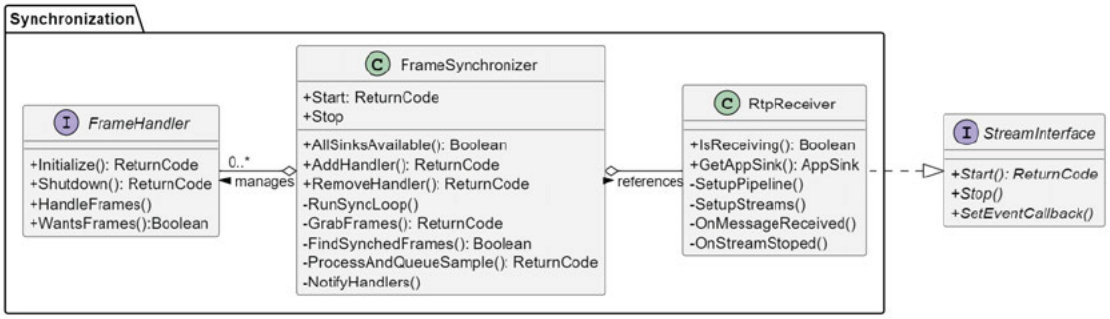


Figure 4.7: Class diagram of the synchronization component showing the relationships between `RtpReceiver`, `FrameSychronizer`, and frame handling interfaces. The diagram illustrates the core classes responsible for video stream management and temporal frame synchronization.

The pipeline configuration starts with a `udpsrc` element for RTP and RTCP packet reception, connected to `rtpbin` for session management. The video data flows through `rtph264depay` for RTP packet extraction and `avdec_h264` for H.264 decoding. A `videoconvert` element produces BGR format frames for `OpenCV` compatibility, with an `appsink` providing programmatic frame access.

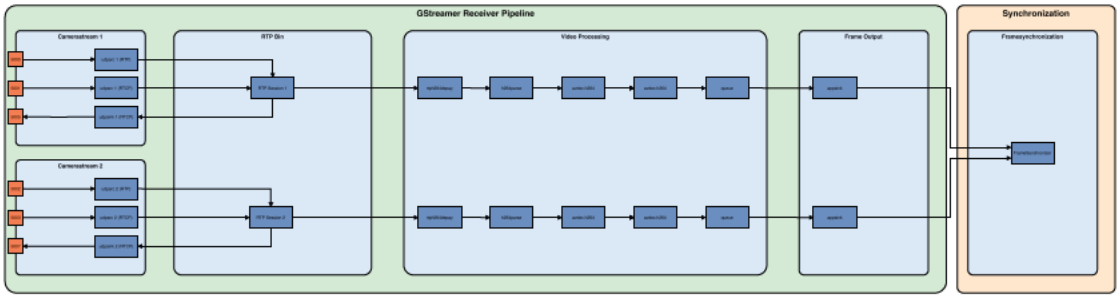


Figure 4.8: GStreamer pipeline structure for video reception, showing element connections and data flow. The pipeline handles RTP video data reception, RTCP control traffic, H.264 decoding, and frame conversion.

`RtpReceiver` instances use three UDP ports per stream: RTP video data reception, RTCP reception from sender, and RTCP report transmission. The RTCP implementation supports network monitoring and packet loss detection, though a GStreamer bug currently results in invalid package loss reporting (-1).

The `StreamState` structure manages stream state based on three criteria: pipeline playing state, network activity presence, and packet reception within two seconds. Pipeline

construction uses GStreamer's pad addition mechanism through the `OnPadAdded` callback as streams become available.

Frame Synchronization The `FrameSychronizer` employs a multithreaded architecture with dedicated grabber threads per video stream and a central synchronization thread for frame matching. The synchronization process uses timestamps with a 33 ms grace period, corresponding to the frame period at 30 FPS.

The synchronization process starts with the `ProcessAndQueueSample` method processing frames from `GStreamer's appsink` elements. This method extracts video data and timestamps, creating Frame objects containing raw image data in OpenCV's matrix format. Stream-specific queues, protected by individual mutexes, store these frames. The `FindSyncedFrames` method analyses timestamp values of frames at each queue's front, applying the 33 ms timing window for frame matching. Frames exceeding this temporal threshold are discarded, triggering new synchronization attempts.

Frame distribution uses the `FrameHandler` interface implementing the observer pattern. This allows calibration and pose estimation components to process synchronized frames independently. The implementation limits queue sizes to 300 frames (10 seconds at 30 FPS) to manage memory usage, removing older frames when this limit is reached.

Camera Calibration Implementation The camera calibration implementation employs a multithreaded workflow operating on synchronized video streams. Figure 4.9 illustrates the class structure implementing pattern detection, frame acquisition, and parameter estimation components.

Pattern Detection and Capture The implementation uses a 7×4 chessboard pattern with 50 mm squares for camera calibration. Operating at capture distances from 2 m to 4 m, the asymmetric pattern layout prevents rotational ambiguity during detection. The `ChessboardPatternDetector` processes calibration frames through `findChessboard-Corners`, applying adaptive thresholding and sub-pixel refinement to achieve reprojection errors below 1.0 pixel.

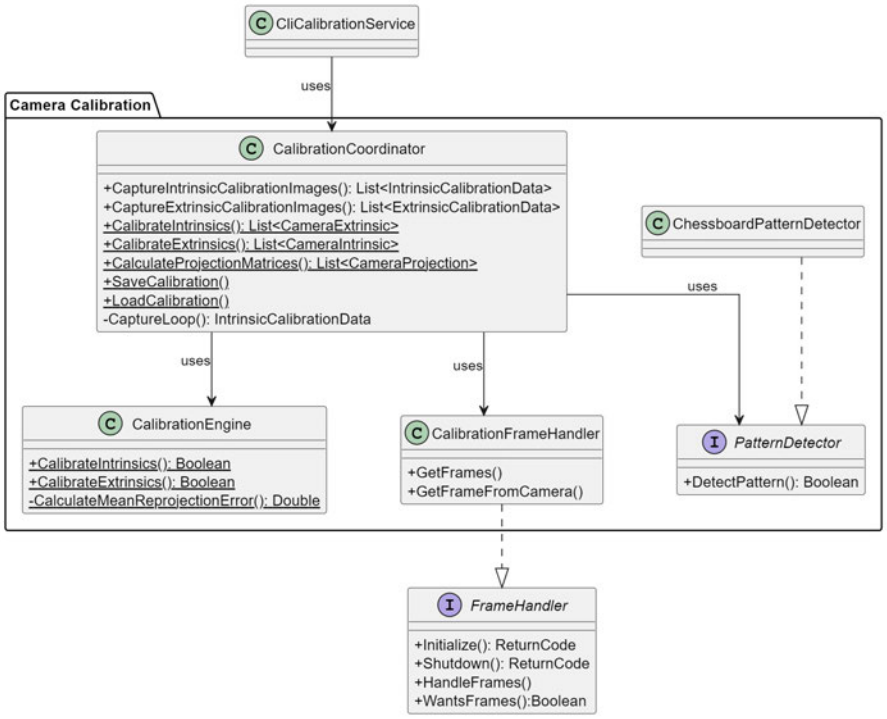


Figure 4.9: Class diagram showing `CalibrationCoordinator` interactions with pattern detection, frame management, and parameter estimation components.

Frame Acquisition through a *lock-free* single-producer-single-consumer queue in the `CalibrationFrameHandler`. Operating with a 300-frame capacity (10 sec at 30 FPS), the handler processes intrinsic calibration frames independently while maintaining synchronized queues for extrinsic calibration.

The pattern detection implementation includes a frame selection mechanism that operates on a newest-frame policy. This design choice prioritizes temporal relevance of calibration data by processing the most recent frames and discarding older ones when processing cannot keep up with the input frame rate.

The `CalibrationCoordinator` processes camera streams in parallel through the pattern detection interface. The implementation defaults to 20 images per camera for intrinsic calibration, configurable through the calibration parameters. This default value satisfies NFR-5's reprojection error constraint of 1.0 pixel. Extrinsic calibration establishes the world coordinate system through synchronized pattern capture, following `OpenCV's` right-handed coordinate convention.

The implementation determines camera poses through `solvePnP`, computing rotation and translation matrices **(R|t)** between world and camera coordinates, as discussed in Section 2.3. Among `cv::ITERATIVE`, `cv::P3P`, and `cv::EPNP`, the implementation utilizes `cv::ITERATIVE` with Levenberg-Marquardt optimization [45], [50], [59] for reprojection error minimization.

`cv::P3P` implements Gao's algorithm [24] for four-point scenarios, while `cv::EPNP` offers computational efficiency [44]. These methods did not achieve satisfactory accuracy in the given setup, potentially due to the planar nature of the chessboard pattern or image noise. The NFR-5 compliance is achieved using `cv::ITERATIVE`, which performs initial pose estimation via DLT or homography decomposition, followed by iterative refinement [9, p. 675].

Projection matrices **($\mathbf{P} = \mathbf{K}[\mathbf{R}|\mathbf{t}]$)** combine intrinsic and extrinsic parameters for each camera. These matrices transform world points to image coordinates, supporting the triangulation process detailed in Section 2.4.2.

Parameter Estimation and Validation The `CalibrationEngine` estimates camera parameters through OpenCV's `calibrateCamera` function. The implementation applies `CALIB_FIX_ASPECT_RATIO` and `CALIB_ZERO_TANGENT_DIST` flags [9, p. 674], reflecting fixed sensor aspect ratios and negligible tangential distortion in modern cameras.

The implementation enforces a 1.0 pixel reprojection error threshold before proceeding to subsequent stages. Camera parameters and validation metrics persist in *YAML* format, enabling rapid parameter validation during initialization.

Recovery from calibration failures operates at two levels: immediate user feedback for pattern detection errors and diagnostic logging for parameter estimation failures. The implementation maintains calibration data per camera, allowing independent retries of failed calibrations while preserving successful results.

Mocap Pipeline Implementation

The mocap pipeline operates through *interprocess communication* between C++ and Python components. Figure 4.10 illustrates the class structure connecting pose estimation with the synchronization infrastructure described previously.

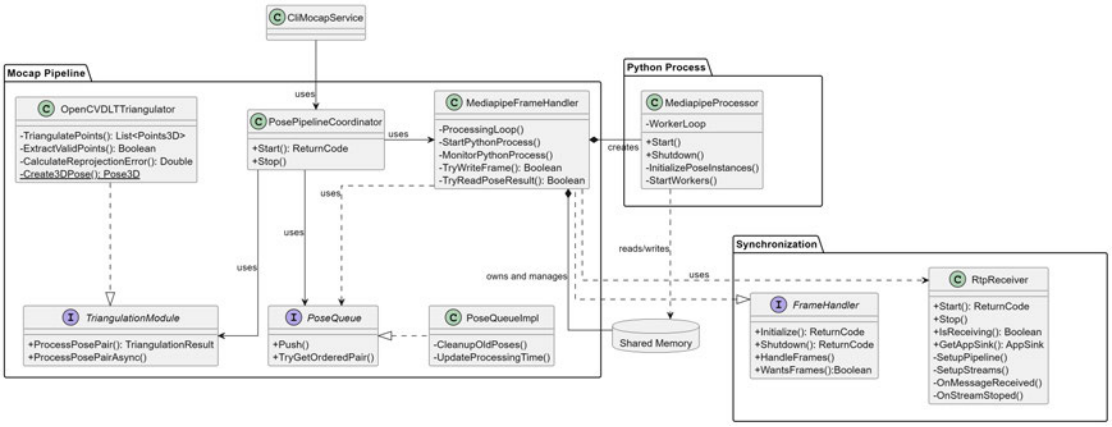


Figure 4.10: Class diagram of the mocap pipeline, implementation showing the relationships between its components.

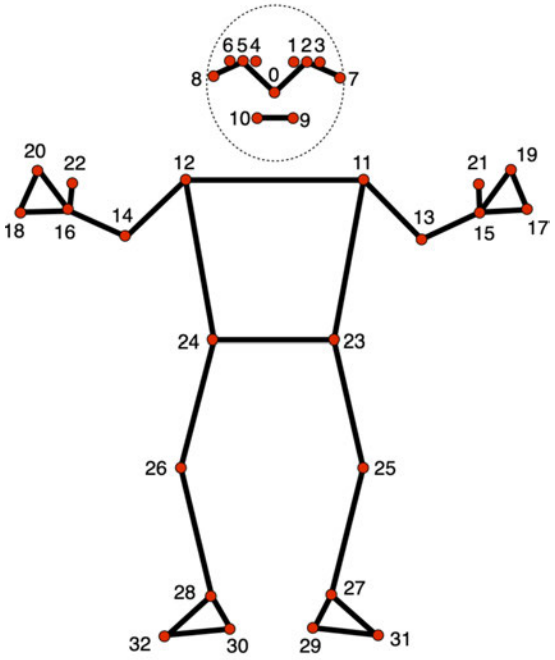
Pose Detection Model MediaPipe's landmark detection model processes 33 anatomical points shown in Figure 4.11. The implementation executes two sequential stages per frame: person detection and landmark localization. Each landmark output combines spatial coordinates with a confidence score, filtering results below the 95% confidence threshold. Frame processing is distributed across multiple worker threads to target 30 FPS performance.

Process Architecture and Communication The implementation uses shared memory for efficient data exchange between the C++ pipeline and Python-based pose estimation process. MediaPipe's framework architecture dictated the Python implementation decision. While MediaPipe's Python bindings provide a high-level API for pose detection integration, its C++ implementation requires managing the `Bazel`[9] build system, model dependencies, and custom API development. MediaPipe's C++ version provides example code rather than a framework, necessitating wrapper classes and interfaces for graph initialization, model loading, and result parsing.

The Python implementation accesses MediaPipe's pose detection models and configurations through pip-installable packages with an API that manages these complexities internally, allowing focus on mocap functionality implementation.

As illustrated in Figure 4.12, the shared memory structure implements a hierarchical organization with distinct sections for control, frame data, and pose results. The memory

[9] An open-source build and test tool similar to Make, Maven, and Gradle that uses a human-readable, high-level build language.



Face Landmarks

0	Nose		
1	Left eye (inner)	4	Right eye (inner)
2	Left eye	5	Right eye
3	Left eye (outer)	6	Right eye (outer)
7	Left ear	8	Right ear
9	Mouth (left)	10	Mouth (right)

Upper Body Landmarks

11	Left shoulder	12	Right shoulder
13	Left elbow	14	Right elbow
15	Left wrist	16	Right wrist
17	Left pinky	18	Right pinky
19	Left index	20	Right index
21	Left thumb	22	Right thumb

Lower Body Landmarks

23	Left hip	24	Right hip
25	Left knee	26	Right knee
27	Left ankle	28	Right ankle
29	Left heel	30	Right heel
31	Left foot index	32	Right foot index

Figure 4.11: MediaPipe pose landmark configuration showing the 33 tracked anatomical points and their connections, with landmarks grouped by body region. (Source: MediaPipe Documentation [25])

layout begins with a header section containing control structures and synchronization primitives. Each camera stream maintains independent frame and pose buffers, with 64-byte alignment to prevent *false sharing* in multithreaded access patterns. The implementation sets a buffer capacity of 300 frames per camera, accommodating 10 s of video at 30 FPS.

Stream synchronization uses *atomic sequence counters* for frame reading and writing operations. Each stream maintains separate counters for frame and pose data, enabling independent tracking of frame processing and pose result generation. This design allows the pose estimation process to work asynchronously while maintaining temporal consistency through frame timestamps.

The shared memory implementation includes validation mechanisms that verify structure alignment, buffer sizing, and memory layout integrity during initialization. This validation ensures consistent memory interpretation between C++ and Python processes, preventing data corruption and misalignment issues.

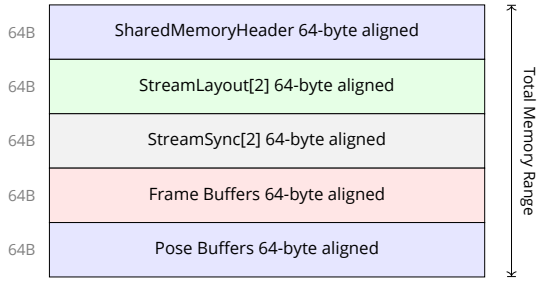


Figure 4.12: Shared memory layout showing the hierarchical organization of header, stream management, and data buffer sections. All sections are 64-byte aligned to optimize cache utilization and prevent false sharing.

Frame Processing Implementation The frame processing implementation centers on the `MediaPipeFrameHandler`, managing the interface between the synchronization system and pose estimation process. The handler implements a multithreaded architecture with dedicated queues per camera stream, allowing independent frame processing while maintaining temporal synchronization.

Frame handling occurs through a processing loop monitoring frame availability from RTP receivers. When frames arrive, the handler implements a lock-free queue system with 600 frames capacity per camera. This queue size handles processing rate variations while preventing unbounded memory growth. The implementation uses separate read and write indices per camera stream, enabling efficient frame management without explicit locking mechanisms.

The Python process initialization incorporates a validation phase verifying system capabilities and configuring GPU acceleration when available. The implementation creates separate worker threads per camera, each maintaining its own MediaPipe pose estimation instance. This approach enables parallel processing of frames from different cameras while avoiding resource contention.

Pose Estimation Pipeline The pose estimation process implements a *worker pool* with defaulting to four threads per camera. Each worker thread maintains an independent MediaPipe pose detector instance configured for real-time performance. Following MediaPipe's pose landmark model specifications [25], the detector uses model complexity 2 (offering 33 pose landmarks) and a detection confidence threshold of 0.7. These parameters balance detection accuracy with processing speed, while the confidence threshold minimizes false positives [5], [48]. Frame processing follows a staged approach:

1. Frame conversion for MediaPipe compatibility
2. Pose estimation through MediaPipe's detection pipeline
3. Keypoint extraction and confidence calculation
4. Result synchronization back to the C++ process

The implementation monitors processing performance through timing metrics at each stage. When processing delays exceed the frame interval, the system applies *frame skipping* to maintain real-time operation while avoiding queue *overflow*.

3D Reconstruction Implementation The 3D reconstruction component implements triangulation of 2D pose detections into 3D coordinates through the `OpenCVDLTTriangulator` class. The implementation uses DLT for triangulation while managing pose pair synchronization and result validation.

Pose Pair Management The implementation uses the `PoseQueueImpl` to manage temporal synchronization of 2D poses from different cameras. The queue maintains poses with a configurable age limit, defaulting to 100 ms to match latency requirements, handling processing delays while preventing memory accumulation. When poses from corresponding frames arrive, the queue forms `OrderedPosePair` structures for triangulation.

Triangulation Process To ensure accurate 3D reconstruction, the triangulation implementation processes pose pairs through several stages. The input keypoints from MediaPipe, provided in normalized image coordinates [0, 1], first undergo coordinate transformation. These points are converted to pixel coordinates using the image dimensions, then transformed to camera coordinates using the inverse of the camera matrix and undistortion parameters when applicable.

Point validation examines each keypoint pair through confidence thresholds and visibility checks. The implementation requires detected points to exceed the 95% confidence threshold (configurable, default matching NFR-4), while maintaining a minimum number of valid corresponding points between views for triangulation (configurable, default 33 points[10]).

[10] Matching MediaPipe's tracked landmarks

The core triangulation computation employs OpenCV's `triangulatePoints` function using the calibrated projection matrices and the transformed camera coordinates. This DLT approach produces homogeneous coordinates in world space (in millimeters), which the implementation then normalizes to obtain Euclidean 3D point representations.

Quality assessment concludes the triangulation process through reprojection error analysis. The implementation projects the computed 3D points back to each camera's view and measures the deviation from the original detections. Reconstructions exceeding the configured maximum reprojection error threshold (default 1.0 pixel as per NFR-3) are rejected, ensuring only valid 3D poses progress through the pipeline.

Data Export Implementation The implementation realizes FR-5 through the `PoseOutputInterface` callback interface. A concrete `CSVPoseWriter` implementation provides buffered CSV output containing frame timestamps, 3D coordinates, confidence scores, and reprojection errors for each detected landmark. The implementation maintains the 30 FPS processing requirement through configurable buffer sizes, reducing I/O operations while handling continuous data streams.

4.3 Summary

The implementation establishes a distributed mocap system through a modular service architecture encompassing video capture, pose estimation, and 3D reconstruction. The system integrates GStreamer for stream processing, MediaPipe for pose detection, and OpenCV for computer vision operations. Key capabilities include multi-camera operation with automated calibration, real-time pose estimation, and 3D reconstruction. The implementation employs both C++ and Python components, utilizing shared memory for efficient *inter-process* communication.

The following chapter presents a systematic evaluation of this implementation, examining performance metrics and accuracy measures under controlled laboratory conditions to assess fulfilment of the specified requirements.

5 Evaluation and Discussion

This chapter presents the evaluation of the implemented mocap system in terms of both functional capabilities and overall performance. Given the challenges of obtaining ground truth data without specialized equipment, the evaluation focuses on quantifiable performance metrics and acknowledges the inherent limitations in absolute accuracy assessment. The tests were conducted under controlled laboratory conditions, using two DMK 32BUR0521 [78] industrial cameras at 1080p and 30 FPS. While the system was designed for distributed operation, the evaluation employed a single development machine running multiple distributed components to validate core functionality and performance.

5.1 Evaluation Methodology

The evaluation employed a systematic approach across three areas: environment configuration, data collection, and performance metrics. The test setup utilized consumer-grade hardware combined with industrial cameras to validate the system's practicality in real-world scenarios.

5.1.1 Test Environment Setup

The evaluation environment included an AMD Ryzen 7 3700X *Central Processing Unit (CPU)* and an NVIDIA GeForce RTX 2080 GPU. Two DMK 32BUR0521 cameras provided synchronized, high-resolution video input. This setup is representative of the intended deployment scenario, allowing a realistic assessment of system performance and accuracy.

5.1.2 Data Collection Approach

Instrumentation combined direct timing measurements in Python-based components with `Tracy` profiling [81] for the C++ modules. The Python pose estimation component, identified as a potential bottleneck, was instrumented for frame-level timing. The C++ pipeline, responsible for frame streaming, synchronization, and 3D reconstruction, was profiled using Tracy to gain detailed insights into execution times and resource usage. Detailed profiling data is provided in Appendix B. This combined approach ensured a comprehensive performance analysis.

5.1.3 Evaluation Metrics

The evaluation criteria include the following key metrics:

Processing Pipeline Performance: End-to-end frame processing time from capture to 3D pose output. The target was 30 FPS, implying a maximum of 33 ms per frame.

Calibration Accuracy: Measured by reprojection error. A threshold of 1.0 pixel was set to ensure sufficient calibration quality.

Pose Detection Quality: Confidence scores from MediaPipe's pose estimation. A 95% keypoint confidence threshold was established for reliable detection.

Stream Synchronization: Verified by comparing timestamps to ensure frame alignment within the 33 ms interval.

3D Reconstruction Accuracy: Evaluated through reprojection checks and qualitative comparisons against measured reference points. While no absolute ground truth was available, the system's reconstruction fidelity is gauged through consistency and plausibility checks.

5.2 System Performance Analysis

Performance evaluation examined two distinct architectural components: the Python-based pose estimation pipeline and the C++ streaming/reconstruction pipeline. The analysis focused on identifying processing characteristics, resource utilization, and potential bottlenecks in each component.

5.2.1 Python Pose Estimation Component

Tests were conducted with single-, dual-, and quad-worker configurations per camera to assess parallelization effects. The single-worker setup achieved approximately 5FPS, while the dual-worker configuration initially reached about 14 FPS before diminishing over time. Surprisingly, the quad-worker configuration degraded performance to 4–5 FPS. None of these configurations met the 30 FPS target.

During testing, measured MediaPipe pose estimation times remained relatively stable (approximately 19 ms for camera 0 and 24 ms for camera 1), irrespective of thread configuration. This consistency indicates a fundamental processing limit dominated by the MediaPipe component. Although MediaPipe offers GPU acceleration options in certain configurations, the version integrated with the Python binding in this system was CPU-bound. Consequently, increased threading did not significantly reduce processing times, as CPU resources were already heavily utilized. This CPU-bound pose estimation step emerged as the principal bottleneck in achieving the target frame rate.

Frame reading remained at 4–5 ms. Queue management, however, showed significant Frame backlog[1] growth, especially in the single- and quad-worker setups, leading to substantial processing delays and reduced throughput.

In summary, the Python-based pose estimation component remained CPU-bound, with MediaPipe processing times showing little sensitivity to threading configurations. The absence of effective GPU acceleration and the constraints imposed by Python's *Global Interpreter Lock (GIL)*[2] prevented any meaningful parallelization gains. As a result, pose estimation performance stabilized at a level insufficient to meet the 30 FPS target, ultimately limiting the overall throughput of the system's processing pipeline.

5.2.2 Streaming and Reconstruction Pipeline

Profiling of the C++ modules showed efficient operation, with most tasks remaining well below the 33 ms per frame budget. Frame synchronization averaged 11.07 ms, and

[1] A frame backlog occurs when incoming video frames arrive faster than they can be processed, causing them to accumulate in a queue. This leads to increasing delays between when a frame is captured and when its results are available, making real-time operation difficult or impossible.

[2] The Global Interpreter Lock (GIL) is a mutex that protects access to Python objects, preventing multiple native threads from executing Python bytecode simultaneously. While this simplifies Python's memory management, it can significantly limit performance in CPU-bound multithreaded applications by preventing true parallel execution.

triangulation took about 1.54 ms per pose pair. Inter-process communication with the Python component (writing frames and reading pose results) was efficient, with pose result retrieval completing in microseconds.

To provide an overview, Table 5.1 summarizes key average processing times for major pipeline components: Combining the processing times from the Python component

Component/Stage	**Avg. Processing Time**
Python frame read / pose write	4–5 ms (read), 0.65 ms (write)
Python Pose Estimation (per frame)	43 ms
C++ Frame Synchronization	11.07 ms
C++ Frame Grabbing (critical path)	3.81 ms
C++ MediaPipe Handling (frame write/ pose read)	1.49 ms (write), 1 µs (read)
C++ Triangulation (incl. Extract/Process)	1.54 ms
C++ Pose Writing (batch every 100)	1.9 ms

Table 5.1: Timings of specific pipeline components

(53–79 ms) with the total average processing time for the C++ components (27.31 ms), yields end-to-end times between 80.31 and 106.31 ms, corresponding to approximately 9–12 FPS. This does not meet the target of 30 FPS (NFR-1). The primary bottleneck lies in the Python pose estimation step, which dominates total latency.

Key Takeaways for Performance:

- MediaPipe-driven pose estimation in Python is the main performance bottleneck.
- C++ components, including synchronization and triangulation, perform efficiently and remain within acceptable time budgets.
- The end-to-end rate of 9–12 FPS is significantly below the 30 FPS target.

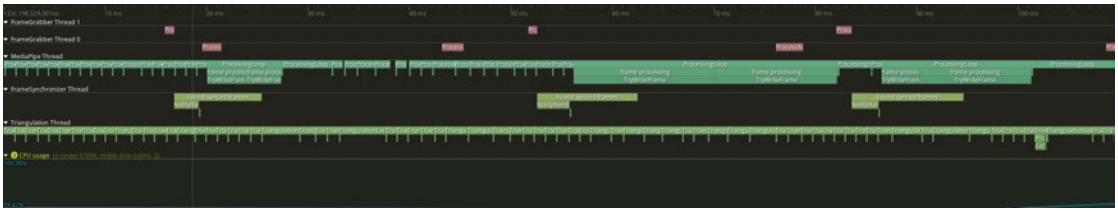


Figure 5.1: Tracy timeline showing thread activity for frame grabbing, synchronization, MediaPipe processing, and triangulation in the 3D reconstruction pipeline.

5.2.3 Latency Analysis

End-to-end latency exceeds the target 33 ms per frame due to the Python processing stage. The faster acquisition rate (30 FPS input) compared to lower processing throughput results in queue build-up. Over time, this leads to growing frame delays.

5.2.4 Resource Utilization

CPU utilization reached 100%, reflecting the high computational demands of the system. Despite Python's GIL limiting single-process thread parallelism, the multiprocess architecture enabled full CPU utilization. GPU utilization remained moderate, highlighting potential untapped acceleration opportunities. Memory usage was stable, with no significant leaks detected. Inter-process communication operated efficiently without introducing major latency.

5.2.5 Multi-Camera Synchronization

Despite the performance constraints, multi-camera synchronization proved robust. Frame timestamps remained aligned within the desired 33 ms interval. The synchronization overhead of approximately 18–19 ms contributes to the total latency but was manageable. Differences in frame grabbing times between cameras (3.81 ms vs. 1.7 ms) did not compromise synchronization quality. Temporal coherence was preserved, enabling consistent 3D reconstruction despite the reduced overall frame rate.

5.3 Accuracy Assessment

The system's accuracy was evaluated across four dimensions: calibration precision, pose detection reliability, reconstruction accuracy, and environmental adaptability. Each dimension was assessed using quantitative metrics where possible, supplemented by qualitative analysis where necessary.

5.3.1 Calibration Accuracy

Camera calibration consistently yielded reprojection errors below 0.3 pixels, surpassing the 1.0 pixel target. This indicates good intrinsic and extrinsic parameter estimation, supporting accurate 3D reconstruction.

5.3.2 Pose Detection Quality

MediaPipe pose estimation, tested under favourable lighting and minimal background interference, yielded an average overall confidence of about 0.82. Core landmarks (e.g., hips and shoulders) consistently scored above 0.91, while extremities such as heels (0.66–0.72) and elbows (0.67–0.70) were less reliable. Although these results were obtained under controlled conditions, pose detection quality is known to degrade in more complex environments. This variation in confidence, combined with environmental sensitivities, prevented the system from meeting the 95% threshold defined by NFR-4. As a result, the current level of pose detection reliability limits the system's effectiveness in diverse, real-world scenarios.

5.3.3 3D Reconstruction Accuracy

The accuracy of the 3D reconstruction pipeline, which uses OpenCV's `triangulatePoints` function, was evaluated by comparing computer-reconstructed coordinates against physical measurements. The experimental setup consisted of two cameras positioned 4 meters from the subject, with a calibration checkerboard establishing the world coordinate system at 2 meters distance. To assess reconstruction accuracy, keypoints in the scene were selected and their triangulated coordinates were compared with direct physical measurements obtained using a measuring tape. For example, one reference point was reconstructed at (-4.3 cm, 158.13 cm, -90.34 cm) through the vision system, while physical measurements placed the same point at (-8.0 cm, 167.5 cm, -98.5 cm). The discrepancies between these measurements—3.7 cm in X, 9.37 cm in Y, and 8.16 cm in Z—suggest centimeter-level accuracy, though systematic biases may be present. However, the validation approach has notable limitations: the physical measurement process introduces its own uncertainties, particularly for points that were difficult to access, and

the sample size of measured points was limited. A more comprehensive accuracy assessment would benefit from automated ground truth acquisition using precision motion stages or optical tracking systems.

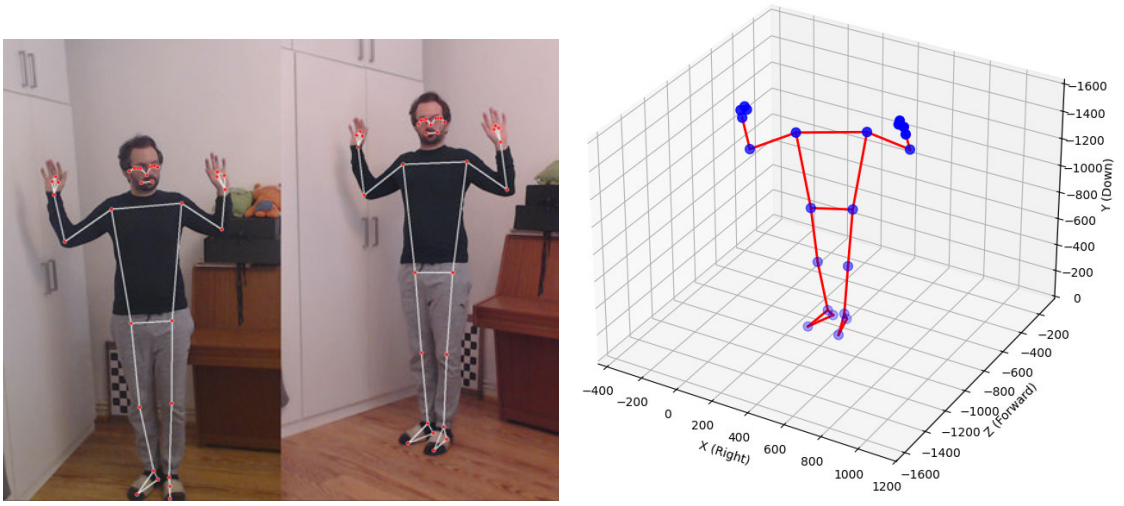


Figure 5.2: Multi-view human pose estimation results: (left) 2D pose detection from first camera view, (middle) 2D pose detection from second camera view, and (right) the resulting triangulated 3D pose reconstruction.

5.3.4 Environmental Impact Analysis

Testing under variable lighting (e.g., natural daylight in winter) showed that insufficient illumination prevented effective operation. Pose detection also suffered in complex indoor environments with common household objects, as illustrated in Figure 5.3. This sensitivity to environmental conditions indicated that reliable operation depended heavily on adequate lighting, a controlled background, and careful camera placement. Such constraints limit the system's robustness in typical indoor scenarios.

5.4 Requirements Fulfilment

This section analyses how well the implemented system met its defined functional and non-functional requirements, followed by a discussion of key limitations identified during testing.

Figure 5.3: Checkerboard pattern with visualized world coordinate system axes (left), and (right) an example of MediaPipe's pose detection failure case where the algorithm incorrectly detected human pose keypoints on a drawer instead of the actual person in the scene.

5.4.1 Functional Requirements Analysis

Table 5.2 presents the evaluation results for each functional requirement, indicating the degree of fulfilment and providing supporting evidence from system testing.

5.4.2 Non-Functional Requirements Analysis

Table 5.3 summarizes the evaluation results for non-functional requirements, comparing achieved performance against target specifications.

5.4.3 System Limitations

The evaluation revealed several limitations across performance, environmental, and technical domains. From a performance perspective, the Python-based pose estimation creates substantial processing overhead, with MediaPipe integration limiting achievable frame rates. This is compounded by queue management complexities in multithreaded configurations and inter-process communication overhead.

Environmental constraints impact system reliability. The implementation shows high sensitivity to ambient lighting conditions and background complexity, requiring carefully controlled capture environments. Camera placement requirements further restrict the capture volume and system flexibility.

Requirement	Status	Evaluation Results
FR-1 Multi-Camera Setup	Fulfilled	✓ Successful operation with stereo setup ❍ N-camera support implemented, but limited to two cameras due to pipeline performance ✓ Reliable synchronization within 33 ms interval
FR-2 Camera Calibration	Fulfilled	✓ Automated process complete within 15 minutes ✓ Reprojection errors < 0.3 pixels ✓ Consistent performance across sessions
FR-3 2D Pose Estimation	Partially Fulfilled	✗ Real-time pose estimation via MediaPipe ✓ Confidence scores for all keypoints ✗ Detection reliability varies significantly with environmental conditions ✗ Falls below 95% confidence threshold in challenging conditions
FR-4 3D Reconstruction	Partially Fulfilled	✓ Successful implementation of triangulation ❍ Limited occlusion handling ❍ Reasonable accuracy in controlled conditions
FR-5 Data Export	Fulfilled	✓ CSV export successfully implemented ✓ Raw coordinate data fully accessible ✓ Architecture supports format extensibility
FR-6 Real-time Processing	Partially Fulfilled	✗ Operates at 9.41–12.45 FPS ✗ Below target 30 FPS performance ✓ Maintains temporal coherence

Table 5.2: Functional Requirements Fulfilment Analysis

Requirement	Target	Achieved	Status
NFR-1: Processing Performance	$\geq$ 30 FPS	9.41–12.45 FPS	Not Fulfilled
NFR-2: System Latency	$\leq$ 100 ms	80.31–106.31 ms	Partially Fulfilled
NFR-3: Reconstruction Quality	$\leq$ 1.0 pixel	$<$ 0.3 pixels	Fulfilled
NFR-4: Pose Confidence	$\geq$ 95%	$\leq$ 82%	Not Fulfilled
NFR-5: Camera Calibration	$\leq$ 1.0 pixel	$<$ 0.3 pixels	Fulfilled
NFR-6: Stream Sync	$\leq$ 33 ms	$\leq$ 33 ms	Fulfilled

Table 5.3: Non-Functional Requirements Fulfilment Analysis

Technical limitations include inadequate occlusion handling, restricting usage when subject visibility is partially blocked. While the system supports CSV data export, the absence of industry-standard formats like BVH and FBX limits integration with professional mocap workflows. Limited validation capabilities for absolute accuracy complicate quality assessment, and Python's GIL prevents true parallel execution, leading to inefficient resource utilization in multithreaded scenarios.

These limitations, particularly in processing performance and real-time capabilities, impact the system's practical applications and provide clear direction for future development efforts in performance optimization, environmental robustness, and format support.

5.5 Discussion

The evaluation results highlight tradeoffs in system architecture, performance, and technical viability for markerless mocap, while suggesting clear paths for future development.

5.5.1 Architecture Considerations

The distributed architecture offered modularity but added complexity in synchronization and inter-process communication. Using Python for pose estimation accelerated initial development but introduced performance bottlenecks through GIL restrictions, with the multithreaded approach showing diminishing returns beyond two workers. Future iterations would need to either implement everything in C++ , potentially replacing MediaPipe

with a more efficient pose detection model, or at minimum use `MediaPipe`'s native APIs and a more suitable *threading model* like a *task system* for the whole Pipeline.

5.5.2 Performance vs Usability Balance

The achieved 9–12 FPS and average pose confidence of 82% fall short of real-time mocap requirements. While MediaPipe provided usable pose detection under ideal conditions, its sensitivity to background objects and furniture limits practical indoor use. The system architecture caused additional overhead through queue management and inter-process communication. Improvements should focus on three areas: replacing Python components with C++ implementations, evaluating alternative pose detection approaches with better environmental robustness, and restructuring the processing pipeline to use a task-based system instead of dedicated threads to reduce *context switching* and idle-wait overhead.

5.5.3 Technical Viability Assessment

The system demonstrated basic markerless mocap functionality using industrial cameras and standard computing hardware, achieving sub-centimeter calibration accuracy and stable camera synchronization. While the current frame rate and environmental constraints limit practical applications, the implementation proves that the concept works: accurate 3D pose reconstruction is possible using industrial cameras and open-source components. The modular pipeline design provides a foundation for systematic improvements, particularly in pose detection and processing efficiency, to reach commercial performance targets.

Beyond these technical considerations, several questions remain about broader applicability and future directions. The following chapter examines these aspects in detail, discussing the project's key contributions, remaining challenges, and opportunities for further development in both research and practical applications.

6 Conclusion and Future Work

6.1 Summary of Contributions

This thesis presented the development and implementation of a markerless mocap system for smart home monitoring applications. The work focused on creating a distributed system architecture that enables multi-camera operation using industrial cameras and standard computing hardware The implementation separates video capture, pose estimation, and 3D reconstruction into independent services, allowing flexible deployment across processing nodes while maintaining temporal synchronization between camera streams.

The system implementation achieved several technical objectives. The camera synchronization remained within the 33 ms target interval, while the calibration process consistently produced reprojection errors below 0.3 pixels, meeting the specified accuracy requirements. Integration of MediaPipe's pose detection with the custom 3D reconstruction pipeline enabled basic mocap functionality, though performance remained below the targeted frame rate.

The evaluation process examined both technical performance and practical limitations through systematic testing. Detailed timing analysis identified processing bottlenecks and resource utilization patterns, providing quantitative data on system behaviour. This analysis revealed specific constraints in the current implementation while highlighting areas where future optimization could improve performance.

6.2 Limitations of the Current System

The implementation demonstrated several limitations during evaluation. In terms of processing performance, the system operates at 9–12 frames per second, not meeting the targeted 30 FPS requirement. This limitation stems primarily from the computational

demands of the MediaPipe pose estimation component and the overhead introduced by Python-based inter-process communication.

The system shows significant sensitivity to environmental conditions. Even under optimal testing conditions—with clear backgrounds, no furniture, and consistent lighting—the pose detection component achieved only 82% confidence, falling short of the specified 95% threshold. This performance was achieved in a controlled laboratory setting that does not reflect typical residential environments. Preliminary testing in more realistic conditions with furniture and variable lighting indicated substantially lower detection reliability, particularly for tracking extremities like ankles and wrists.

Additional technical constraints affect system operation. The current implementation provides only basic occlusion handling, which proves insufficient when furniture or room layout obstruct camera views. The data export functionality supports only CSV format output, lacking integration with established mocap formats like BVH or FBX.

While the distributed architecture enables system scalability, it introduces complexity in deployment and operation. The inter-process communication and synchronization mechanisms create overhead that, though acceptable in testing, could affect long-term stability.

6.3 Future Research Directions

Based on the experiences throughout this thesis, several directions for future development emerge. A fundamental enhancement would be implementing the pose estimation directly in C++ . This architectural change would not only eliminate the current inter-process communication overhead, but also enable full GPU acceleration and native MediaPipe optimization that are currently limited by the Python bindings. By removing these artificial constraints, the system could better utilize available processing resources to approach the target frame rate of 30 FPS. Direct C++ implementation would also allow for deeper integration with MediaPipe's native features and more efficient memory management, potentially improving both performance and accuracy.

The system's environmental sensitivity, observed during both development and testing, aligns with known challenges in vision-based pose estimation systems [74]. While traditional RGB-based approaches face inherent limitations in varying lighting conditions and cluttered environments, the integration of depth sensors has been shown to improve

robustness [76], [96]. Depth information can significantly enhance pose estimation accuracy by providing explicit 3D measurements [38], though this would need to be balanced against the goal of using standard hardware.

The development process revealed particular challenges in occlusion handling that require attention for practical deployment. The current implementation would benefit from tracking algorithms that can maintain pose estimates when body parts are temporarily hidden from view [67]. As demonstrated in recent research [98], this could incorporate temporal movement prediction to bridge gaps in direct observation, leveraging the inherent patterns in human motion

The distributed processing architecture, while proving effective for basic system operation, presents several opportunities for optimization. Stream synchronization and data distribution methods could be refined to reduce overall system latency. Additionally, the current GPU capabilities remain largely unused, suggesting potential performance gains through hardware acceleration of specific processing steps.

6.4 Potential Applications and Impact

The system demonstrated both the potential and current limitations of markerless mocap for smart home monitoring applications. The system successfully achieved core technical objectives like sub-0.3 pixel calibration accuracy and reliable camera synchronization within 33 ms intervals, establishing a foundation for movement tracking in residential settings. The ability to process skeletal data without storing video footage proved the feasibility of privacy-preserving monitoring, addressing a key requirement for smart home applications.

However, the achieved performance metrics—9–12 FPS processing rate and 82% pose confidence—fell short of the requirements for reliable real-time monitoring applications like fall detection and activity recognition. The system's sensitivity to furniture and variable lighting conditions, typical challenges in residential environments, highlighted the gap between laboratory testing and practical deployment scenarios.

The distributed architecture demonstrated that basic markerless mocap functionality is achievable without specialized equipment. From an implementation perspective, this thesis demonstrated the feasibility of integrating existing technologies like MediaPipe

with industrial cameras into a basic motion tracking system for smart home environments.

From an implementation perspective, this thesis demonstrated the feasibility of integrating existing technologies like MediaPipe with industrial cameras into a basic motion tracking system for smart home environments. While performance and environmental robustness need significant improvement, the distributed system architecture provides useful insights for future development in this area.

This proof-of-concept achieved its primary goal of exploring the practical challenges of markerless mocap in residential settings. The systematic evaluation of both technical capabilities and limitations provides a clear foundation for understanding what improvements would be needed for practical deployment, while demonstrating the complexity and requirements of implementing mocap systems in smart home environments.

Bibliography

[1] S. Amin, M. Andriluka, M. Rohrbach, and B. Schiele, "Multi-view pictorial structures for 3d human pose estimation," in *British Machine Vision Conference, BMVC 2013, Bristol, UK, September 9-13, 2013*, T. Burghardt, D. Damen, W. W. Mayol-Cuevas, and M. Mirmehdi, Eds., BMVA Press, 2013. DOI: 10.5244/C.27.45. [Online]. Available: https://doi.org/10.5244/C.27.45.

[2] M. Andriluka, L. Pishchulin, P. Gehler, and B. Schiele, "2d human pose estimation: New benchmark and state of the art analysis," in *2014 IEEE Conference on Computer Vision and Pattern Recognition*, 2014, pp. 3686–3693. DOI: 10.1109/CVPR.2014.471.

[3] Autodesk, *Fbx overview*, https://www.autodesk.com/products/fbx/overview, Accessed: 2024-10-08, 2024.

[4] A. Banno, "A p3p problem solver representing all parameters as a linear combination," *Image and Vision Computing*, vol. 70, pp. 55–62, 2018, ISSN: 0262-8856. DOI: https://doi.org/10.1016/j.imavis.2018.01.001. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0262885618300027.

[5] V. Bazarevsky, I. Grishchenko, K. Raveendran, T. Zhu, F. Zhang, and M. Grundmann, "Blazepose: On-device real-time body pose tracking," *arXiv preprint arXiv: 2006.10204*, 2020.

[6] Biovision, *Biovision hierarchical (bvh) file format specification*, Biovision Motion Capture Company, Accessed via various online documentation, e.g., https://research.cs.wisc.edu/graphics/Courses/cs-838-1999/Jeff/BVH.html Accessed: 2024-10-08, 1992.

[7] *Boost C++ libraries*, Accessed: 2024-06-11, 2024. [Online]. Available: https://www.boost.org/.

[8] A. Bourke, K. O'Donovan, and A. M. Clifford, "Wearable and mobile technology for fall prevention and balance improvement in older adults: A systematic review," *Healthcare*, vol. 9, no. 10, p. 1329, 2021. DOI: 10.3390/healthcare9101329. [Online]. Available: https://www.mdpi.com/2227-9032/9/10/1329.

[9] G. Bradski and A. Kaehler, *Learning OpenCV: Computer Vision with the OpenCV Library*. Sebastopol, CA: O'Reilly Media, Inc., 2008, ISBN: 9780596516130.

[10] D. C. Brown, "Close-range camera calibration," *Photogrammetric Engineering*, vol. 37, no. 8, pp. 855–866, 1971.

[11] S. Brown, *The C4 model for visualising software architecture*, Accessed: 2024-09-28, 2024. [Online]. Available: https://c4model.com.

[12] M. Burenius, J. Sullivan, and S. Carlsson, "3d pictorial structures for multiple view articulated pose estimation," in *Proceedings of the 2013 IEEE Conference on Computer Vision and Pattern Recognition*, ser. CVPR '13, USA: IEEE Computer Society, 2013, pp. 3618–3625, ISBN: 9780769549897. DOI: 10.1109/CVPR.2013.464. [Online]. Available: https://doi.org/10.1109/CVPR.2013.464.

[13] C. Cao, Y. Weng, S. Zhou, Y. Tong, and K. Zhou, "Facewarehouse: A 3d facial expression database for visual computing," *IEEE Transactions on Visualization and Computer Graphics*, vol. 20, no. 3, pp. 413–425, 2014. DOI: 10.1109/TVCG.2013.249.

[14] Z. Cao, G. Hidalgo, T. Simon, S.-E. Wei, and Y. Sheikh, "OpenPose: Realtime multi-person 2D pose estimation using part affinity fields," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 43, no. 1, pp. 172–186, 2021. DOI: 10.1109/TPAMI.2019.2929257.

[15] Z. Cao, T. Simon, S.-E. Wei, and Y. Sheikh, "Realtime multi-person 2d pose estimation using part affinity fields," in *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017, pp. 1302–1310. DOI: 10.1109/CVPR.2017.143.

[16] Y. Chen, Z. Wang, Y. Peng, Z. Zhang, G. Yu, and J. Sun, "Cascaded pyramid network for multi-person pose estimation," in *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2018, pp. 7103–7112. DOI: 10.1109/CVPR.2018.00742.

[17] J. Dower and P. Langdale, *Performing for Motion Capture: A Guide for Practitioners*. London: Bloomsbury Publishing, 2022, ISBN: 9781350211254.

[18] H. Durrant-Whyte and T. Bailey, "Simultaneous localization and mapping: Part i," *IEEE Robotics & Automation Magazine*, vol. 13, no. 2, pp. 99–110, 2006. DOI: 10.1109/MRA.2006.1638022.

[19] Federal Institute for Research on Building, Urban Affairs and Spatial Development, *Household forecast 2020-2040*, Accessed: 2024-11-10, 2021. [Online]. Available: https://www.bbsr.bund.de/BBSR/EN/home/topnews/houshold-forecast.html.

[20] *FFmpeg*, Accessed: 2024-06-11, 2024. [Online]. Available: https://www.ffmpeg.org/.

[21] M. A. Fischler and R. C. Bolles, "Random sample consensus: A paradigm for model fitting with applications to image analysis and automated cartography," *Communications of the ACM*, vol. 24, no. 6, pp. 381–395, Jun. 1981, ISSN: 0001-0782. DOI: 10.1145/358669.358692.

[22] W. Förstner and B. P. Wrobel, *Photogrammetric Computer Vision, Statistics, Geometry, Orientation and Reconstruction* (Geometry and Computing), 1st ed. Springer Cham, 2016, vol. 11, pp. XVII, 816, ISBN: 978-3-319-11550-4. DOI: 10.1007/978-3-319-11550-4. [Online]. Available: https://doi.org/10.1007/978-3-319-11550-4.

[23] M. Furniss, "Motion capture: An overview," *Animation Journal*, vol. 8, no. 2, pp. 68–82, 2000.

[24] X.-S. Gao, X.-R. Hou, J. Tang, and H.-F. Cheng, "Complete solution classification for the perspective-three-point problem," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 25, no. 8, pp. 930–943, 2003. DOI: 10.1109/TPAMI.2003.1217599.

[25] Google, *MediaPipe solutions: Pose landmarker*, Accessed: 2024-06-11, 2024. [Online]. Available: https://ai.google.dev/edge/mediapipe/solutions/vision/pose_landmarker/python.

[26] Google, *Protocol buffers*, Accessed: 2024-10-12, 2024. [Online]. Available: https://protobuf.dev/.

[27] gRPC Authors, *Grpc*, https://grpc.io, Accessed: 2024-09-29, Cloud Native Computing Foundation, 2024.

[28] J. A. Grunert, "Das Pothenotische Problem in erweiterter Gestalt nebst über seine Anwendungen in der Geodäsie," *Grunerts Archiv für Mathematik und Physik*, vol. 1, pp. 238–248, 1841.

[29] S. M. Grünvogel, *Einführung in die Computeranimation: Methoden, Algorithmen, Grundlagen*, German. Wiesbaden: Springer Vieweg, 2024, p. 623, ISBN: 978-3-658-41988-2.

[30] GStreamer, *Gstreamer: Open source multimedia framework*, https://gstreamer.freedesktop.org/, Accessed: 2024-06-20, 2024.

[31] T. Hanning, *High Precision Camera Calibration*, 1st ed. Wiesbaden: Vieweg+Teubner Verlag, 2011, p. 212, ISBN: 978-3-8348-1413-5. DOI: 10.1007/978-3-8348-9830-2.

[32] R. Haralick, H. Joo, C. Lee, X. Zhuang, V. Vaidya, and M. Kim, "Pose estimation from corresponding point data," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 19, no. 6, pp. 1426–1446, 1989. DOI: 10.1109/21.44063.

[33] R. Hartley and A. Zisserman, *Multiple View Geometry in Computer Vision*, 2nd ed. Cambridge: Cambridge University Press, 2004, ISBN: 9780511811685. DOI: 10.1017/CBO9780511811685.

[34] R. I. Hartley and P. Sturm, "Triangulation," *Computer Vision and Image Understanding*, vol. 68, no. 2, pp. 146–157, 1997, ISSN: 1077-3142. DOI: https://doi.org/10.1006/cviu.1997.0547. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1077314297905476.

[35] HAW Hamburg, *Living place laboratory - exploring human-technology interaction*, Accessed: 2024-11-10, 2024. [Online]. Available: https://livingplace.haw-hamburg.de/.

[36] J. Heikkila, "Geometric camera calibration using circular control points," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 22, no. 10, pp. 1066–1077, 2000. DOI: 10.1109/34.879788.

[37] S. Hu, S. Cao, N. Toosizadeh, J. Barton, M. G. Hector, and M. J. Fain, "Radar-based fall detection: A survey [survey]," *IEEE Robotics & Automation Magazine*, vol. 31, no. 3, pp. 170–185, Sep. 2024, ISSN: 1558-223X. DOI: 10.1109/mra.2024.3352851. [Online]. Available: http://dx.doi.org/10.1109/MRA.2024.3352851.

[38] K. Iskakov, E. Burkov, V. Lempitsky, and Y. Malkov, "Learnable triangulation of human pose," in *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, 2019.

[39] R. Kanko, E. Laende, E. Davis, W. Selbie, and K. Deluzio, "Concurrent assessment of gait kinematics using marker-based and markerless motion capture," *Journal of Biomechanics*, vol. 127, p. 110 665, 2021. DOI: 10.1101/2020.12.10.420075.

[40] L. Kneip, "Real-time scalable structure from motion: From fundamental geometric vision to collaborative mapping," Ph.D. dissertation, ETH Zurich, 2013.

[41] L. Kneip, H. Li, and Y. Seo, "Upnp: An optimal o(n) solution to the absolute pose problem with universal applicability," in *Computer Vision – ECCV 2014*, D. Fleet, T. Pajdla, B. Schiele, and T. Tuytelaars, Eds., Cham: Springer International Publishing, 2014, pp. 127–142, ISBN: 978-3-319-10590-1.

[42] M. Kocabas, N. Athanasiou, and M. J. Black, "Vibe: Video inference for human body pose and shape estimation," in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2020.

[43] E. Lengyel, *Mathematics for 3D Game Programming and Computer Graphics*, 3rd. Boston, MA: Cengage Learning, Inc, 2011, p. 545, ISBN: 978-1435458864.

[44] V. Lepetit, F. Moreno-Noguer, and P. Fua, "EPnP: An Accurate O(n) Solution to the PnP Problem," *International Journal of Computer Vision*, vol. 81, no. 2, pp. 155–166, Feb. 2009. DOI: 10.1007/s11263-008-0152-6.

[45] K. Levenberg, "A method for the solution of certain non-linear problems in least squares," *Quarterly of Applied Mathematics*, vol. 2, no. 2, pp. 164–168, 1944.

[46] Y. Li *et al.*, "Tokenpose: Learning keypoint tokens for human pose estimation," in *IEEE/CVF International Conference on Computer Vision (ICCV)*, 2021.

[47] M. Lopez, R. Mari, P. Gargallo, Y. Kuang, J. Gonzalez-Jimenez, and G. Haro, "Deep single image camera calibration with radial distortion," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, Jun. 2019.

[48] C. Lugaresi *et al.*, "MediaPipe: A framework for building perception pipelines," *arXiv preprint arXiv:1906.08172*, 2019.

[49] Markets and Markets, *Smart home market by product, software & services, sales channel, and region - global forecast to 2029*, Accessed: 2024-11-10, 2024. [Online]. Available: https://www.marketsandmarkets.com/Market-Reports/smart-homes-and-assisted-living-advanced-technologie-and-global-market-121.html.

[50] D. W. Marquardt, "An algorithm for least-squares estimation of nonlinear parameters," *SIAM Journal on Applied Mathematics*, vol. 11, no. 2, pp. 431–441, 1963.

[51] J. Martinez, R. Hossain, J. Romero, and J. J. Little, "A simple yet effective baseline for 3d human pose estimation," in *2017 IEEE International Conference on Computer Vision (ICCV)*, 2017, pp. 2659–2668. DOI: 10.1109/ICCV.2017.288.

[52] Medical Alert Advice, *Risks and limitations of medical alert systems*, Accessed: 2024-11-10, 2023. [Online]. Available: https://www.medicalalertadvice.com/articles/risks-limitations-of-medical-alert-systems/.

[53] D. Mehta *et al.*, "Vnect: Real-time 3d human pose estimation with a single rgb camera," *ACM Trans. Graph.*, vol. 36, no. 4, Jul. 2017, ISSN: 0730-0301. DOI: 10.1145/3072959.3073596. [Online]. Available: https://doi.org/10.1145/3072959.3073596.

[54] A. Menache, *Understanding Motion Capture for Computer Animation* (The Morgan Kaufmann Series in Computer Graphics), Second. Elsevier: Morgan Kaufmann, 2011, Copyright © 2011 Elsevier Inc. All rights reserved, ISBN: 978-0-12-381496-8. [Online]. Available: https://doi.org/10.1016/C2009-0-62989-5.

[55] Microsoft Corporation, *Azure kinect developer kit*, https://news.microsoft.com, Accessed: 2024-11-10, 2019. [Online]. Available: https://news.microsoft.com/wp-content/uploads/prod/2019/02/FACT-SHEET-_Azure-Kinect-DK.pdf.

[56] Microsoft Corporation, *Kinect for xbox 360*, https://www.microsoft.com/, Accessed: 2024-11-10, 2010. [Online]. Available: https://www.microsoft.com/.

[57] T. B. Moeslund and E. Granum, "A survey of computer vision-based human motion capture," *Computer Vision and Image Understanding*, vol. 81, no. 3, pp. 231–268, 2001, ISSN: 1077-3142. DOI: https://doi.org/10.1006/cviu.2000.0897. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S107731420090897X.

[58] T. B. Moeslund, A. Hilton, and V. Krüger, "A survey of advances in vision-based human motion capture and analysis," *Computer Vision and Image Understanding*, vol. 104, no. 2, pp. 90–126, 2006, ISSN: 1077-3142. DOI: https://doi.org/10.1016/j.cviu.2006.08.002. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1077314206001263.

[59] J. J. Moré, "The levenberg-marquardt algorithm: Implementation and theory," in *Numerical Analysis*, G. A. Watson, Ed., Berlin, Heidelberg, 1978, pp. 105–116, ISBN: 978-3-540-35972-2.

[60] F. Mueller, D. Mehta, O. Sotnychenko, S. Sridhar, D. Casas, and C. Theobalt, "Real-time hand tracking under occlusion from an egocentric rgb-d sensor," in *Proceedings of International Conference on Computer Vision (ICCV)*, 2017. [Online]. Available: https://handtracker.mpi-inf.mpg.de/projects/OccludedHands/.

[61] G. Nakano, "A simple direct solution to the perspective-three-point problem," in *Proceedings of the British Machine Vision Conference (BMVC)*, Sep. 2019.

[62] A. Newell, K. Yang, and J. Deng, "Stacked hourglass networks for human pose estimation," in *Computer Vision – ECCV 2016*, B. Leibe, J. Matas, N. Sebe, and M. Welling, Eds., Cham: Springer International Publishing, 2016, pp. 483–499, ISBN: 978-3-319-46484-8. DOI: 10.1007/978-3-319-46484-8_29.

[63] J. O'Brien, R. Bodenheimer, G. Brostow, and J. Hodgins, "Automatic joint parameter estimation from magnetic motion capture data.," in *Proceedings of Graphics Interface 2000*, Jan. 2000, pp. 53–60.

[64] OpenCV Contributors, *Opencv: Open source computer vision library*, Accessed: 2024-07-10, 2023. [Online]. Available: https://docs.opencv.org/4.x/index.html.

[65] O. Ozyesil, V. Voroninski, R. Basri, and A. Singer, "A survey on structure from motion," *Acta Numerica*, vol. 26, Jan. 2017. DOI: 10.1017/S096249291700006X.

[66] G. Pavlakos, X. Zhou, K. G. Derpanis, and K. Daniilidis, "Coarse-to-fine volumetric prediction for single-image 3d human pose," in *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017, pp. 1263–1272. DOI: 10.1109/CVPR.2017.139.

[67] D. Pavllo, C. Feichtenhofer, D. Grangier, and M. Auli, "3d human pose estimation in video with temporal convolutions and semi-supervised training," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2019, pp. 7753–7762.

[68] M. Persson and K. Nordberg, "Lambda twist: An accurate fast robust perspective three point (p3p) solver," in *Computer Vision - ECCV 2018: 15th European Conference, Munich, Germany, September 8-14, 2018, Proceedings, Part*

IV, Berlin, Heidelberg: Springer-Verlag, 2018, pp. 334–349, ISBN: 978-3-030-01224-3. DOI: 10.1007/978-3-030-01225-0_20. [Online]. Available: https://doi.org/10.1007/978-3-030-01225-0_20.

[69] PhaseSpace Inc., *Phasespace x2e motion capture system*, https://www.phasespace.com/x2e-motion-capture/, Accessed: 2024-11-10, 2024.

[70] F. Remondino and S. El-Hakim, "Image-based 3d modelling: A review," *The Photogrammetric Record*, vol. 21, no. 115, pp. 269–291, 2006. DOI: https://doi.org/10.1111/j.1477-9730.2006.00383.x. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/j.1477-9730.2006.00383.x. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1111/j.1477-9730.2006.00383.x.

[71] H. Rhodin *et al.*, "Learning monocular 3d human pose estimation from multi-view images," *Proceedings / CVPR, IEEE Computer Society Conference on Computer Vision and Pattern Recognition. IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, vol. 2018, pp. 8437–8446, Jun. 2018. DOI: 10.1109/CVPR.2018.00880.

[72] J. G. Richards, "The measurement of human motion: A comparison of commercially available systems," *Human Movement Science*, vol. 18, no. 5, pp. 589–602, 1999, ISSN: 0167-9457. DOI: https://doi.org/10.1016/S0167-9457(99)00023-8. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167945799000238.

[73] D. Roetenberg, H. Luinge, and P. Slycke, "Xsens mvn: Full 6dof human motion tracking using miniature inertial sensors," *Xsens Motion Technol. BV Tech. Rep.*, vol. 3, Jan. 2009.

[74] N. Sarafianos, B. Boteanu, B. Ionescu, and I. A. Kakadiaris, "3d human pose estimation: A review of the literature and analysis of covariates," *Computer Vision and Image Understanding*, vol. 152, pp. 1–20, 2016, ISSN: 1077-3142. DOI: https://doi.org/10.1016/j.cviu.2016.09.002. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1077314216301369.

[75] T. Sharp *et al.*, "Accurate, robust, and flexible real-time hand tracking," in *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*, ser. CHI '15, Seoul, Republic of Korea: Association for Computing Machinery, 2015, pp. 3633–3642, ISBN: 9781450331456. DOI: 10.1145/2702123.2702179. [Online]. Available: https://doi.org/10.1145/2702123.2702179.

[76] J. Shotton *et al.*, "Efficient human pose estimation from single depth images," *IEEE transactions on pattern analysis and machine intelligence*, vol. 35, pp. 2821–40, Dec. 2013. DOI: 10.1109/TPAMI.2012.241.

[77] I. Sommerville, *Software Engineering, Global Edition*, 10th ed. Harlow, England: Pearson, 2015, p. 816, ISBN: 978-1-292-09613-1.

[78] T. I. Source, *Dmk 32bu-r0521*, https://www.theimagingsource.com/de-de/product/industrial/32u/dmk32bur0521/, Accessed: 2024-3-12, 2024.

[79] D. Sun, S. Wang, H. Xia, C. Zhang, J. Gao, and M. Mao, "Human pose estimation based on cross-view feature fusion," *The Visual Computer*, vol. 40, pp. 6581–6597, Sep. 2024. DOI: 10.1007/s00371-023-03184-3. [Online]. Available: https://doi.org/10.1007/s00371-023-03184-3.

[80] R. Szeliski, *Computer Vision: Algorithms and Applications* (Texts in Computer Science), 2nd ed. Cham: Springer, 2022, pp. XXII, 925, ISBN: 978-3-030-34371-2. DOI: 10.1007/978-3-030-34372-9.

[81] B. Taudul, *Tracy profiler*, Accessed: 2024-2-12, 2024. [Online]. Available: https://github.com/wolfpld/tracy.

[82] The MathWorks, Inc., *Matlab documentation*, Accessed: 2024-07-10, 2023. [Online]. Available: https://de.mathworks.com/help/matlab/.

[83] A. Toshev and C. Szegedy, "Deeppose: Human pose estimation via deep neural networks," in *2014 IEEE Conference on Computer Vision and Pattern Recognition*, 2014, pp. 1653–1660. DOI: 10.1109/CVPR.2014.214.

[84] B. Triggs, "Autocalibration from planar scenes," in *Proceedings of the 5th European Conference on Computer Vision-Volume I - Volume I*, ser. ECCV '98, Berlin, Heidelberg: Springer-Verlag, 1998, pp. 89–105, ISBN: 3540645691.

[85] B. Triggs, P. F. McLauchlan, R. I. Hartley, and A. W. Fitzgibbon, "Bundle adjustment — a modern synthesis," in *Vision Algorithms: Theory and Practice*, B. Triggs, A. Zisserman, and R. Szeliski, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 298–372, ISBN: 978-3-540-44480-0.

[86] R. Tsai, "A versatile camera calibration technique for high-accuracy 3d machine vision metrology using off-the-shelf tv cameras and lenses," *IEEE Journal on Robotics and Automation*, vol. 3, no. 4, pp. 323–344, 1987. DOI: 10.1109/JRA.1987.1087109.

[87] Y. Wang and G. Mori, "Multiple tree models for occlusion and spatial constraints in human pose estimation," in *Proceedings of the 10th European Conference on Computer Vision: Part III*, ser. ECCV '08, Marseille, France: Springer-Verlag, 2008, pp. 710–724, ISBN: 9783540886891. DOI: 10.1007/978-3-540-88690-7_53. [Online]. Available: https://doi.org/10.1007/978-3-540-88690-7_53.

[88] T. Weise, S. Bouaziz, H. Li, and M. Pauly, "Realtime performance-based facial animation," *ACM Trans. Graph.*, vol. 30, no. 4, Jul. 2011, ISSN: 0730-0301. DOI: 10.1145/2010324.1964972. [Online]. Available: https://doi.org/10.1145/2010324.1964972.

[89] J. Weng, P. Cohen, and M. Herniou, "Camera calibration with distortion models and accuracy evaluation," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 14, no. 10, pp. 965–980, 1992. DOI: 10.1109/34.159901.

[90] D. A. Winter, *Biomechanics and Motor Control of Human Movement*. Wiley Online Library: John Wiley & Sons, Inc., 2009, ISBN: 9780470398180. DOI: 10.1002/9780470549148.

[91] Y. Xu, J. Zhang, Q. Zhang, and D. Tao, "ViTPose: Simple vision transformer baselines for human pose estimation," in *Advances in Neural Information Processing Systems*, 2022.

[92] Y. Yuan *et al.*, *Hrformer: High-resolution transformer for dense prediction*, 2021.

[93] W. Zhang, J. Fang, X. Wang, and W. Liu, "Efficientpose: Efficient human pose estimation with neural architecture search," *Computational Visual Media*, vol. 7, no. 3, pp. 335–347, Sep. 2021. DOI: 10.1007/s41095-021-0214-z. [Online]. Available: https://doi.org/10.1007/s41095-021-0214-z.

[94] Z. Zhang, "A flexible new technique for camera calibration," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 22, no. 11, pp. 1330–1334, 2000. DOI: 10.1109/34.888718.

[95] Z. Zhang, "Camera calibration with one-dimensional objects," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 26, no. 7, pp. 892–899, 2004. DOI: 10.1109/TPAMI.2004.21.

[96] Z. Zhang, "Microsoft kinect sensor and its effect," *IEEE MultiMedia*, vol. 19, no. 2, pp. 4–10, 2012. DOI: 10.1109/MMUL.2012.24.

[97] Q. Zhao, C. Zheng, M. Liu, P. Wang, and C. Chen, "Poseformerv2: Exploring frequency domain for efficient and robust 3d human pose estimation," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2023, pp. 8877–8886.

[98] C. Zheng, S. Zhu, M. Mendieta, T. Yang, C. Chen, and Z. Ding, "3d human pose estimation with spatial and temporal transformers," *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, 2021.

[99] Z. Zhenqing, D. Ye, X. Zhang, G. Chen, and B. Zhang, "Improved direct linear transformation for parameter decoupling in camera calibration," *Algorithms*, vol. 9, p. 31, Apr. 2016. DOI: 10.3390/a9020031.

[100] F. Zhou and G. Zhang, "Complete calibration of a structured light stripe vision sensor through planar target of unknown orientations," *Image and Vision Computing*, vol. 23, no. 1, pp. 59–67, 2005, ISSN: 0262-8856. DOI: https://doi.org/10.1016/j.imavis.2004.07.006. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S026288560400188X.

[101] H. Zhou and H. Hu, "Human motion tracking for rehabilitation–a survey," *Biomedical Signal Processing and Control*, vol. 3, pp. 1–18, Jan. 2008. DOI: 10.1016/j.bspc.2007.09.001.

[102] Y. Zhou, M. Habermann, W. Xu, I. Habibie, C. Theobalt, and F. Xu, "Monocular real-time hand shape and motion capture using multi-modal data," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, Jun. 2020, pp. 5345–5354. DOI: 10.1109/CVPR42600.2020.00539.

A Digital Attachment

The following folder structure can be found on the data carrier:

- **documentation:** Contains this document as Latex files and PDF.
- **data_analysis:** Contains raw data and analysis files generated during development and evaluation.
- **images:** Contains all self-created graphics/diagrams of this work.
- **code:** Contains the complete source code of the developed software.

B Profiling Data

This appendix presents the detailed profiling data collected using Tracy [81] for the C++ components of the motion capture pipeline.

B.1 Timing Statistics

Table B.1 presents the raw timing statistics for all significant C++ pipeline components.

B.2 Performance Distributions

Figures B.1–B.16 show the timing distributions for each C++ pipeline component.

Component	Count	Mean (ns)	Median (ns)	Min (ns)	Max (ns)
Create3DPose	251	2.14e2	1.51e2	9.00e1	1.11e3
ExtractValidPoints	251	1.19e6	6.81e5	2.41e5	1.06e7
FindSyncedPoses	28 743	3.20e3	1.08e3	9.00e1	4.78e6
Found synced frames	891	3.58e6	1.57e3	3.90e2	3.71e7
Handle Frames	858	2.50e4	1.21e4	3.63e3	4.03e6
NotifyHandlers	891	7.46e6	6.10e6	1.46e6	3.44e7
ProcessAndQueueSample	2158	2.81e6	1.32e6	6.47e5	3.33e7
ProcessPosePair	251	5.31e3	2.31e3	1.19e3	4.51e5
Processing Pose	251	2.01e5	1.48e5	1.11e5	2.91e6
ProcessingLoop	32 692	1.28e6	1.06e6	1.60e2	1.64e7
TriangulatePoints	251	3.43e5	2.85e5	2.03e5	3.05e6
TriangulationLoop	28 743	1.28e6	1.06e6	2.17e4	1.60e7
TryGetOrderedPair	28 743	1.83e3	4.21e2	5.00e1	2.77e6
TryReadPoseResult	32 692	1.27e3	2.20e2	9.00	3.16e6
TryWriteFrame	1716	3.89e6	1.96e6	6.72e5	3.30e7
Frame processing	65 245	4.54e3	3.00e1	1.00	1.20e7

Table B.1: C++ Component Timing Statistics

B.2.1 FrameGrabber Threads

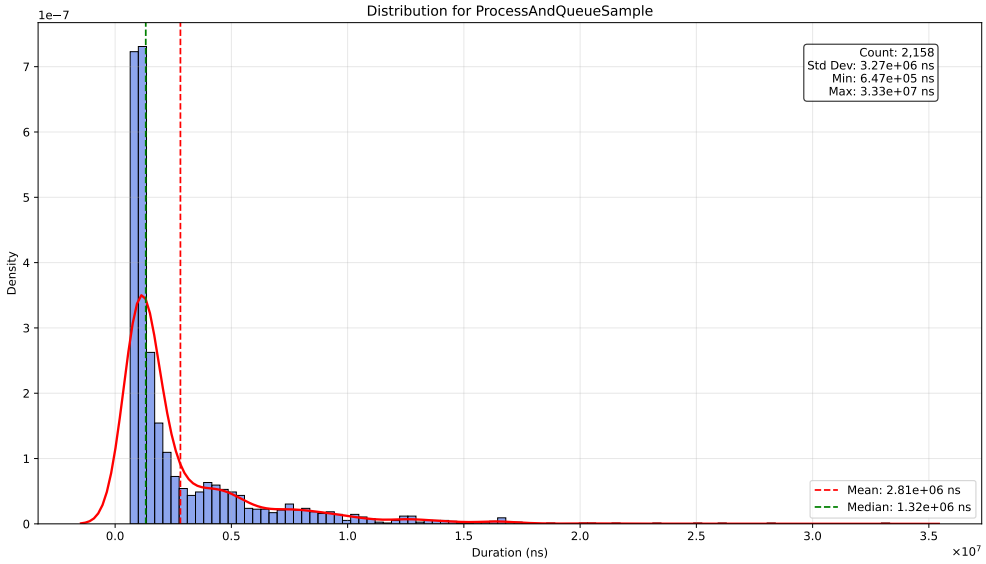


Figure B.1: Timing distribution: ProcessAndQueueSample threads unified

B.2.2 FrameSynchronizer Thread

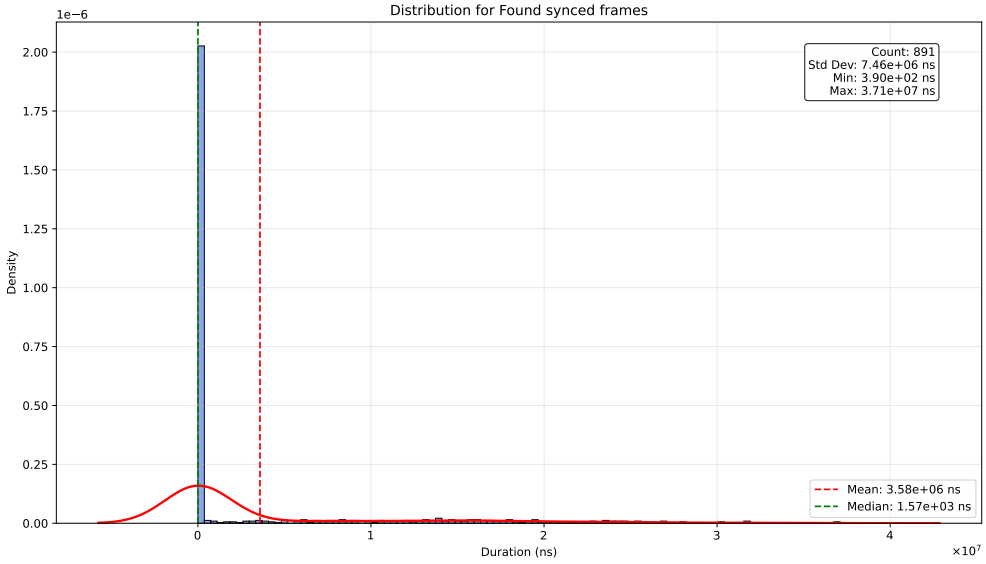


Figure B.2: Timing distribution: FoundSyncedFrames

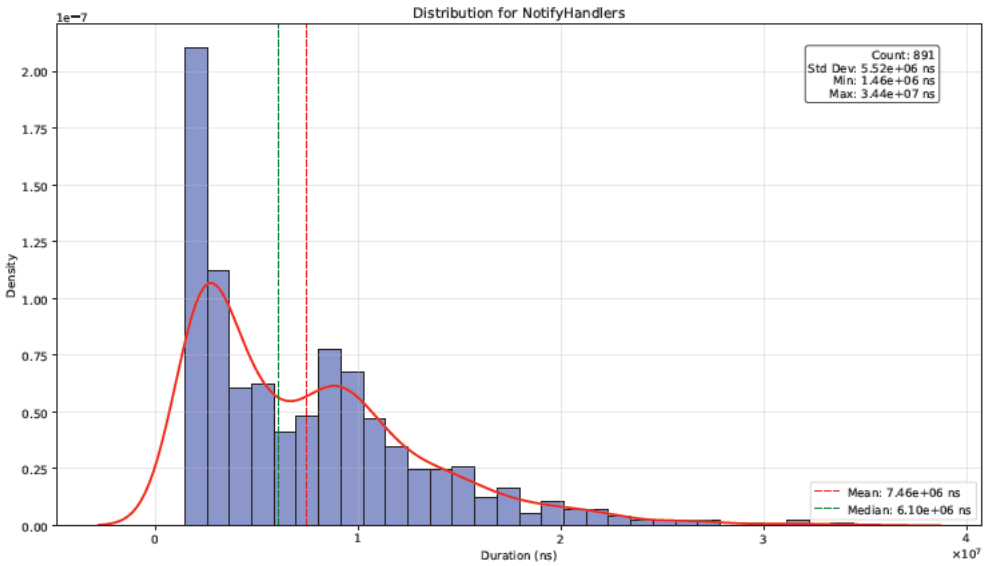


Figure B.3: Timing distribution: NotifyHandlers

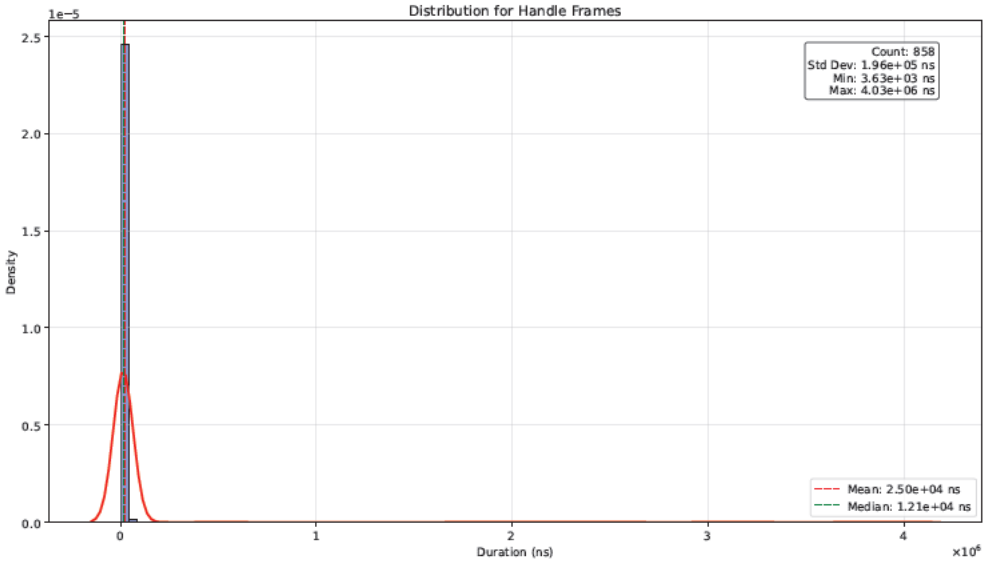


Figure B.4: Timing distribution: HandleFrames

B.2.3 MediaPipe Thread

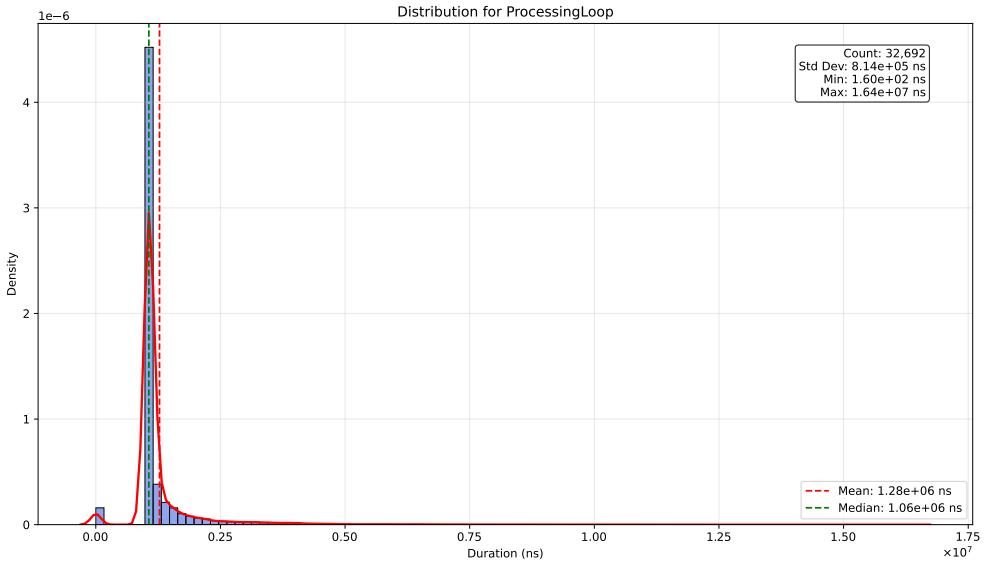


Figure B.5: Timing distribution: ProcessingLoop

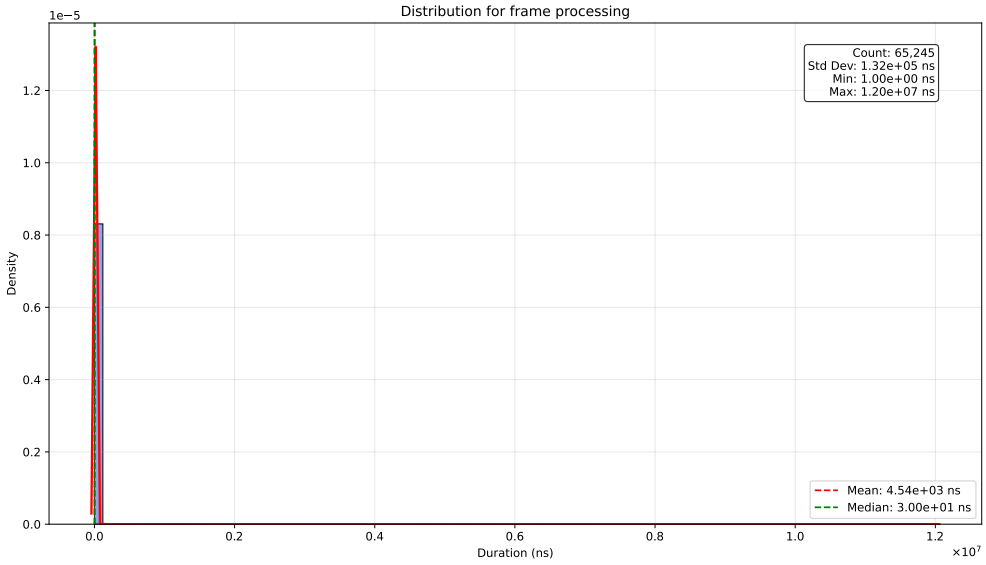


Figure B.6: Timing distribution: FrameProcessing

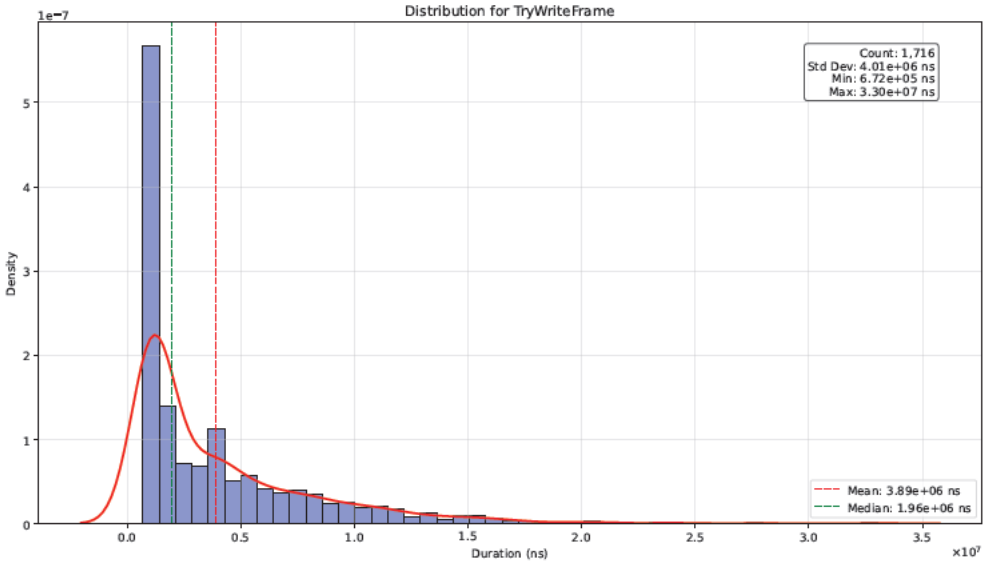


Figure B.7: Timing distribution: TryWriteFrame

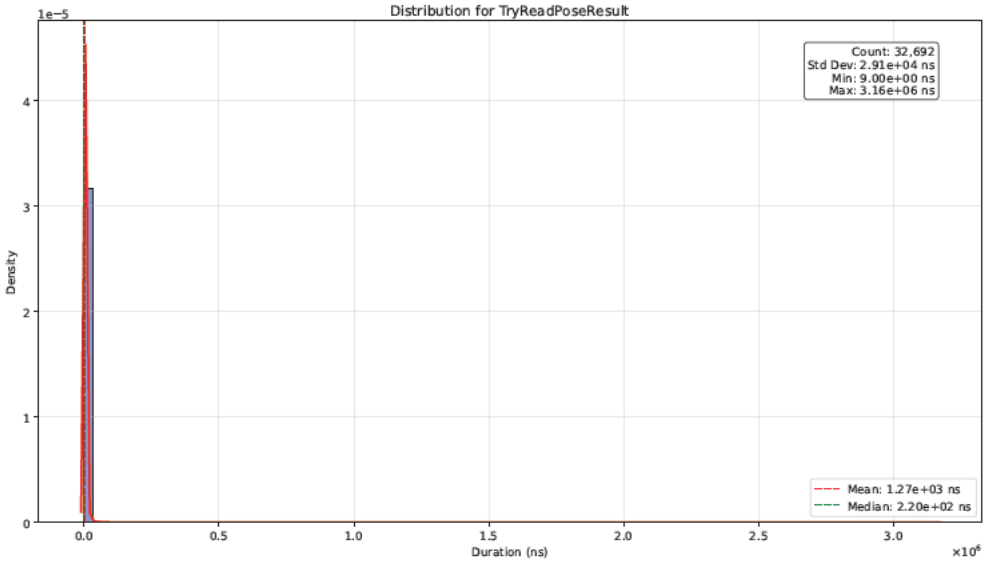


Figure B.8: Timing distribution: TryReadPoseResult

B.2.4 Triangulation Thread

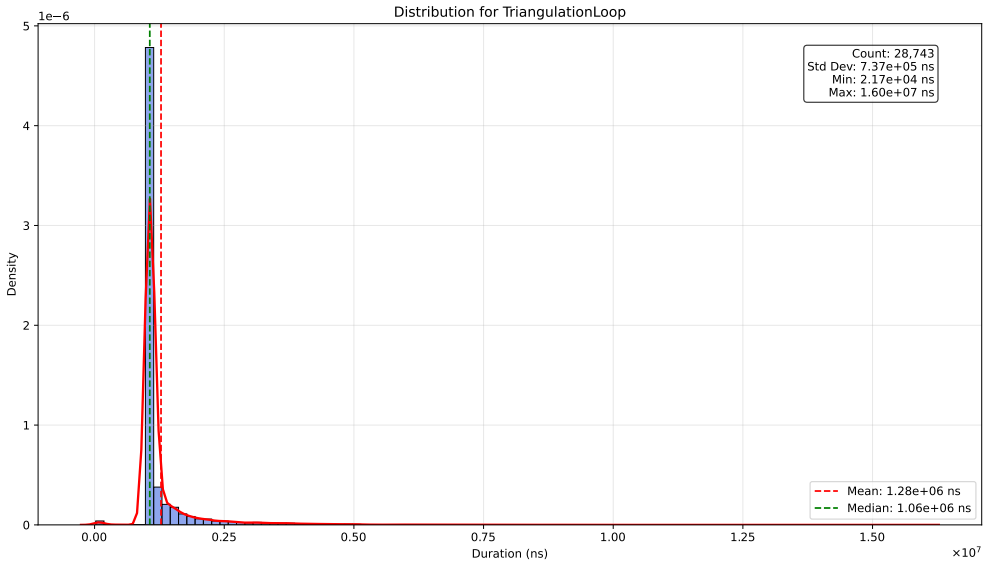


Figure B.9: Timing distribution: TriangulationLoop

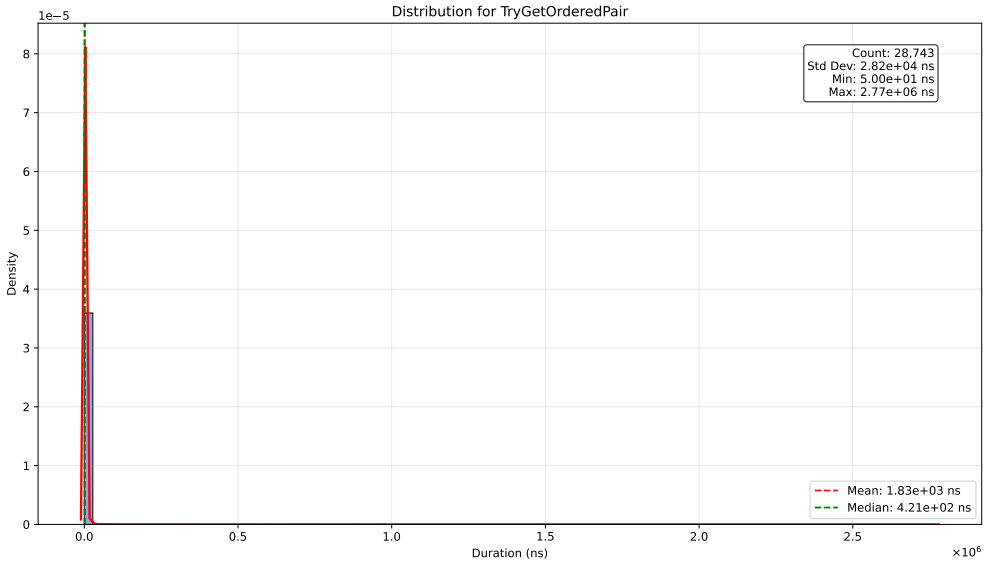


Figure B.10: Timing distribution: TryGetOrderedPair

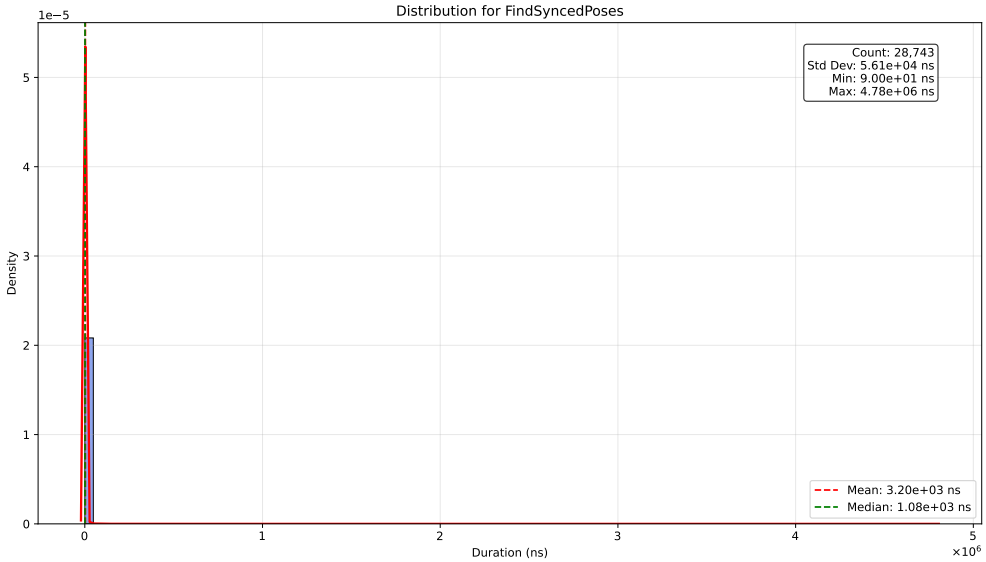


Figure B.11: Timing distribution: FindSyncedPoses

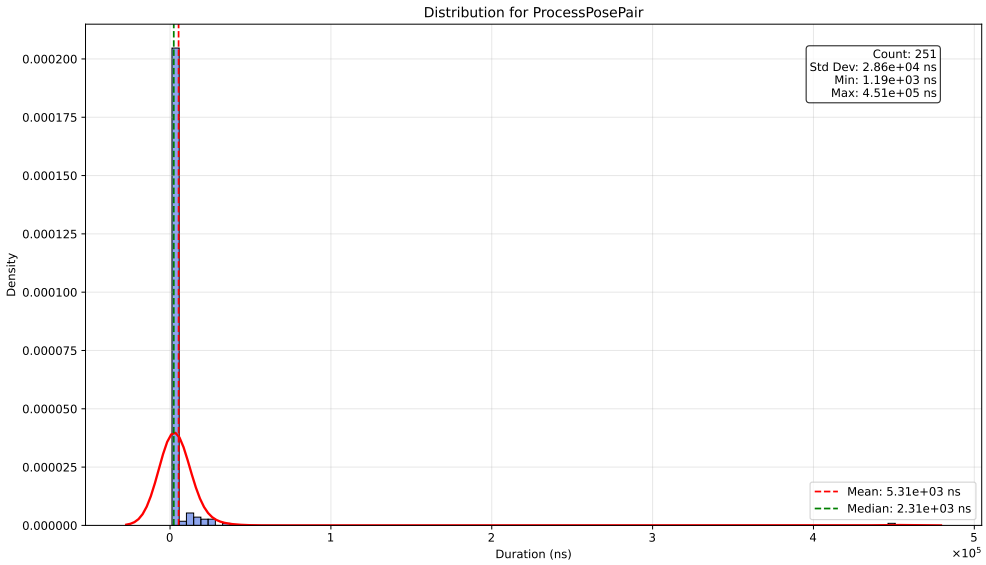


Figure B.12: Timing distribution: ProcessingPosePair

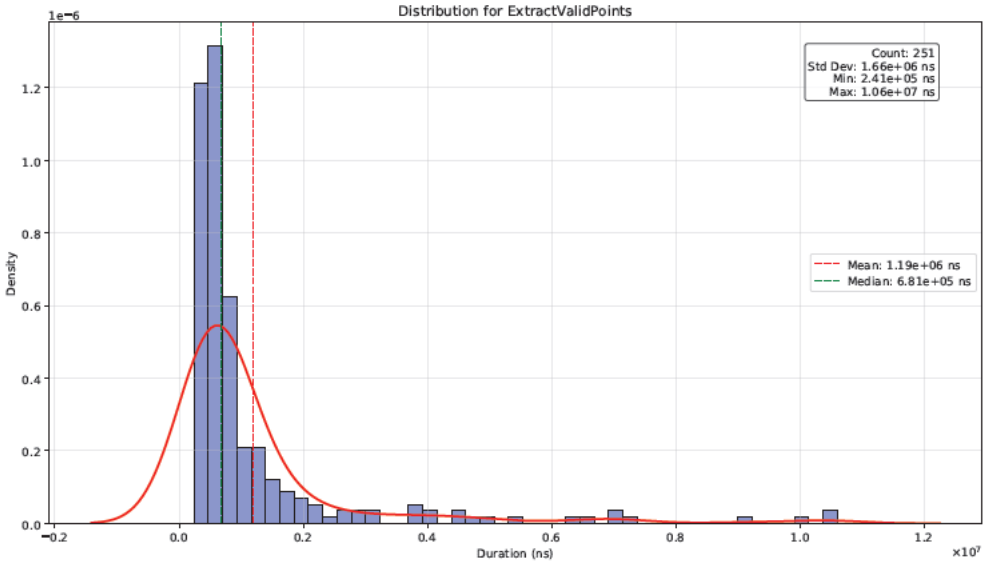


Figure B.13: Timing distribution: ExtractValidPoints

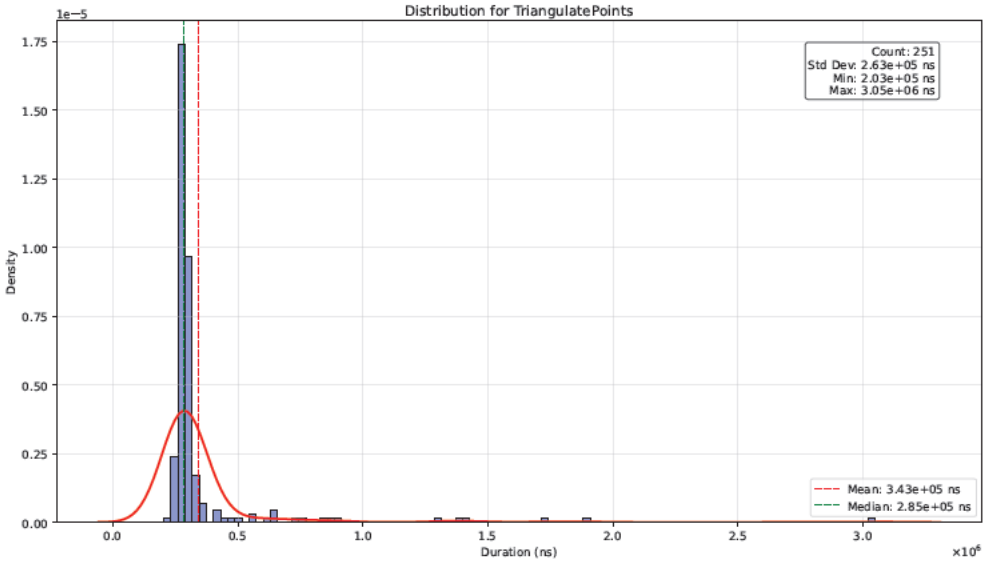


Figure B.14: Timing distribution: TriangulatePoints

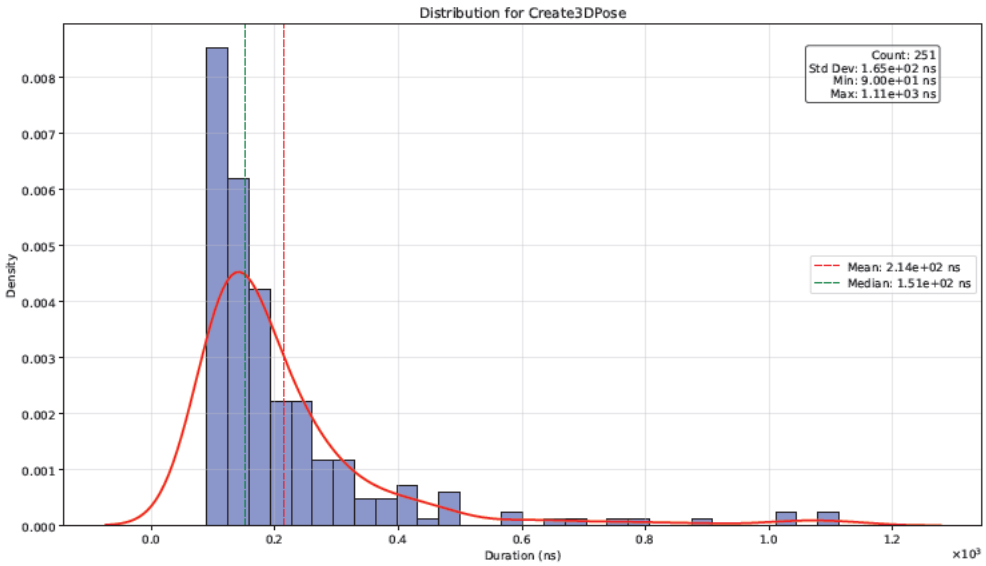


Figure B.15: Timing distribution: Create3DPose

B.2.5 Data export

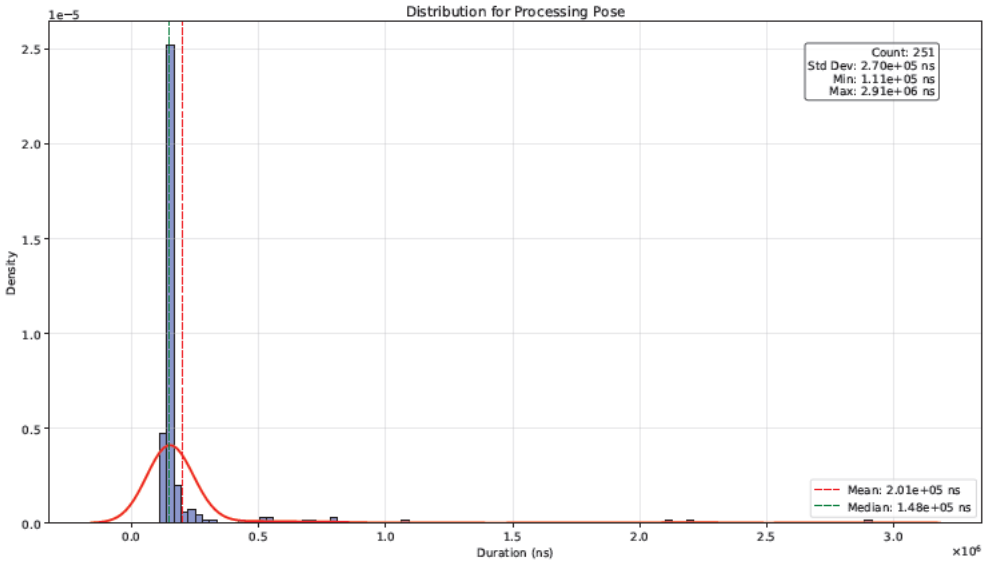


Figure B.16: Timing distribution: ProcessingPose

B.2.6 Heatmap

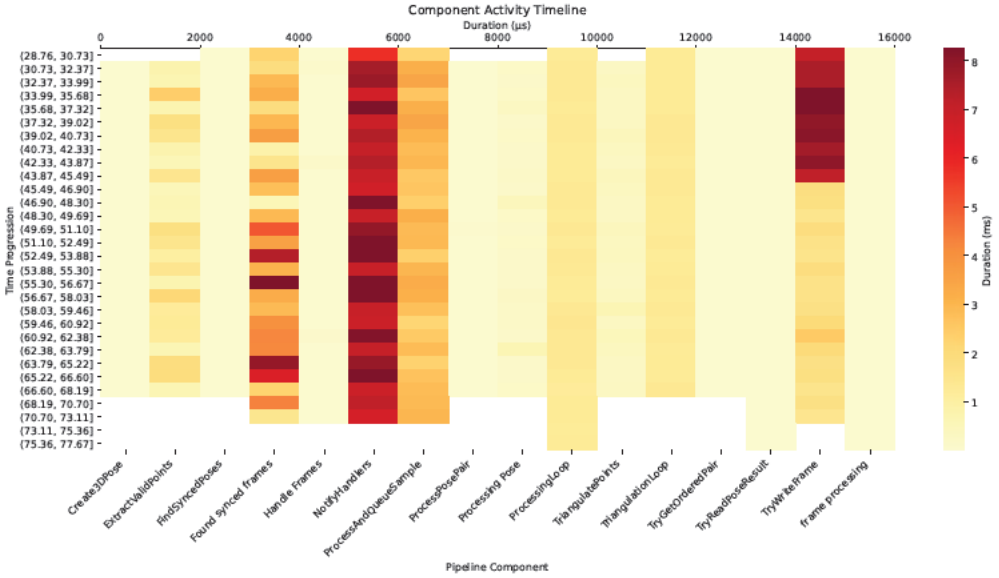


Figure B.17: Component activity timeline

B.3 Used tools

The table B.2 lists the tools and aids used in the context of working on the topic of Bachelor thesis.

Table B.2: Used aids and tools

Tool	Usage
OpenCV	Camera calibration and 3D reconstruction operations
MediaPipe	Framework for pose estimation and tracking
Boost Libraries	C++ utility libraries for implementation
ChatGPT	Code suggestions, debugging support and concept discussion
LanguageTool	Grammar and style checking

Glossary

accelerometer A sensor that measures proper acceleration (g-force) and can detect magnitude and direction of acceleration as a vector quantity, used to detect orientation and movement.

activity recognition The automated identification and classification of human activities from sensor data, typically using pattern recognition and machine learning techniques.

Azure Kinect A developer kit with advanced AI sensors for sophisticated computer vision and speech models, released as the successor to the original Microsoft Kinect.

buffer overflow A condition where a program attempts to write data beyond the boundaries of a pre-allocated memory buffer, potentially causing system crashes or security vulnerabilities.

bundle adjustment A technique that simultaneously refines the 3D structure and camera parameters by minimizing reprojection errors.

callback A function that is passed as an argument to another function and is executed after its parent function has finished execution.

camera calibration The process of determining a camera's internal parameters and its position and orientation in the world.

camera coordinates A three-dimensional coordinate system with its origin at the camera's optical center, used to describe points in the camera's frame of reference.

camera synchronization The process of coordinating multiple cameras to capture images at precisely the same moment, enabling accurate multi-view analysis.

computer vision A field of artificial intelligence that enables computers to understand and process visual information from the digital world, such as images and videos.

confidence score A numerical value indicating the reliability or certainty of a detection or estimation, typically used in computer vision and machine learning systems.

container A standalone, executable package that includes everything needed to run a piece of software, including code, runtime, system tools, system libraries, and settings.

context switching The process of storing and restoring the state of a process or thread so that execution can be resumed from the same point at a later time.

deep learning A subset of machine learning that uses artificial neural networks with multiple layers (deep neural networks) to progressively learn hierarchical representations of data, where each layer transforms its input data into a more abstract and composite representation, enabling the learning of complex patterns and features without manual feature engineering.

deployment configuration The arrangement and distribution of software components across hardware infrastructure in a distributed system.

depth sensor A device that measures the distance between the sensor and objects in a scene, creating a three-dimensional representation of the environment.

epipolar geometry The intrinsic projective geometry between two views, independent of scene structure.

eventual consistency A consistency model where distributed system replicas may temporarily differ but will converge to the same state over time.

extrinsic parameters The parameters that define a camera's position and orientation in world coordinates, typically consisting of a rotation matrix and translation vector.

fall detection An automated system capability to identify when a person has fallen by analyzing changes in their position, movement, or posture.

false sharing A performance degradation that occurs in multi-threaded systems when data structures from independent threads are placed in the same CPU cache line, causing unnecessary cache invalidations and memory synchronization overhead despite the threads not actually sharing data.

focal length The distance between a camera's lens and its image sensor when focused on infinity, determining the angle of view and magnification.

frame buffering The temporary storage of video frames in memory to manage timing differences between capture, processing, and display operations.

frame skipping A technique in real-time video processing where frames are deliberately omitted to maintain processing speed when the system cannot keep up with the input rate.

gyroscope A sensor that measures angular velocity and maintains orientation, using the principles of angular momentum to detect and measure rotational movement.

hardware acceleration The use of specialized hardware to perform certain computations more efficiently than in software running on a general-purpose CPU.

homogeneous coordinates A coordinate system used in computer vision and graphics where 3D points are represented with four components, enabling projective geometry operations and representing points at infinity.

homography A projective transformation that maps points from one plane to another.

horizontal scaling A method of increasing system capacity by adding more processing nodes, rather than increasing the capacity of existing nodes.

image coordinates A two-dimensional coordinate system where points in a camera's field of view are projected onto the image sensor plane, typically measured in pixels from the image origin (usually top-left corner) with the x-axis pointing right and y-axis pointing down.

image plane The two-dimensional surface in a camera where the three-dimensional scene is projected and recorded, typically the camera's sensor or film.

interprocess communication Methods and mechanisms used by processes to share data and communicate with each other in a computing system.

intrinsic parameters The internal camera parameters that define how 3D points are projected onto the image plane, including focal length, principal point, and lens distortion coefficients.

layered architecture An architectural pattern that organizes components into horizontal layers, where each layer provides services to the layer above and uses services of the layer below.

lock-free A programming approach where thread synchronization is achieved without using mutual exclusion locks, typically using atomic operations to ensure thread safety.

magnetometer A sensor that measures the strength and direction of magnetic fields, often used in conjunction with other sensors to determine absolute orientation relative to the Earth's magnetic field.

markerless motion capture A technique for recording movement of objects or individuals without the use of physical markers, typically using computer vision algorithms.

MediaPipe An open-source framework for building multimodal machine learning pipelines, particularly focused on computer vision tasks.

memory management The process of controlling and coordinating computer memory, allocating portions to various programs and processes while ensuring efficient utilization.

Microsoft Kinect A line of motion sensing input devices developed by Microsoft for Xbox gaming consoles and Windows PCs. The device features RGB camera, depth sensor and multi-array microphone.

monitoring system A technological solution that continuously observes, records, and analyzes specific conditions or behaviors within an environment, capable of detecting changes and potentially triggering responses to specific events.

multicast A network communication pattern where data is sent simultaneously to a group of recipients, optimizing bandwidth usage for group communication.

node A discrete processing unit within a distributed system that performs specific tasks and communicates with other nodes through defined network protocols.

noise Random variations or disturbances in data that can affect measurement accuracy and signal quality, occurring from various sources such as sensor limitations, environmental factors, or processing artifacts.

normalized coordinates A coordinate system where values are scaled to a fixed range (typically [0,1]) regardless of the original image dimensions, enabling resolution-independent processing.

occlusion The blocking or hiding of one object or part by another in a visual scene, affecting the ability to detect or track objects accurately.

optical axis The straight line that passes through the center of all optical elements in a camera system, typically perpendicular to the image plane.

pinhole camera model A simple camera model where light rays pass through a single point to form an image on the image plane.

pipeline A set of data processing elements connected in series, where the output of one element is the input of the next one, enabling parallel processing of different stages.

pixel coordinates A two-dimensional coordinate system where points are measured in pixel units relative to the image origin, typically the top-left corner.

pose estimation A computer vision technique that determines the position and orientation of a human body or object by identifying and tracking key points or joints.

principal point The point where the optical axis intersects the image plane in a camera system.

radial distortion A type of lens distortion where straight lines appear curved in the image, more pronounced at the edges.

real-time Processing and responding to input immediately, typically within a guaranteed time constraint, making the computed results available virtually immediately.

reprojection error The distance between the observed image points and the projected points using the estimated camera parameters.

service-oriented architecture An architectural pattern where services communicate with each other through defined protocols across a network to fulfill a software system's requirements.

shared memory A memory region that can be simultaneously accessed by multiple programs or processes, enabling efficient inter-process communication without data copying.

skeletal data A representation of human body movement through a simplified skeleton model, consisting of joint positions and their connections, without preserving visual appearance or personal identifiable features.

smart home A residence equipped with technology that allows automated control of various home systems and devices, including lighting, climate, entertainment systems, and security, often with capabilities for remote monitoring and control.

smart pointer A programming construct that automatically manages memory allocation and deallocation of dynamically allocated objects, helping prevent memory leaks.

star configuration A network topology where all nodes connect to a central hub or switch, providing direct communication paths between any two nodes through the central point.

state management The handling of data that represents the condition of a system or its components, including storage, updates, and synchronization.

stereo setup A configuration of two cameras positioned to capture the same scene from different viewpoints, enabling depth perception and 3D reconstruction.

strong consistency A consistency model where all system replicas immediately reflect updates, ensuring all readers see the same data at the same time, regardless of which replica they access.

structural pattern A reusable solution to a commonly occurring problem in software architecture, defining how components should be organized and interact.

system latency The time delay between an input to a system and its corresponding output, encompassing all processing and communication delays.

task system A programming pattern that organizes work into discrete tasks that can be scheduled and executed independently, often used for parallel processing.

template specialization A C++ feature that allows providing a specific implementation of a template for particular data types or values.

temporal consistency The property of maintaining coherent and smooth transitions between successive frames or time steps in a sequence of measurements or observations, ensuring that changes over time are physically plausible.

threading model A design pattern that defines how multiple threads are organized, managed, and coordinated within a software system.

time-of-flight camera An imaging system that measures distance based on the time it takes for light pulses to travel from the camera to an object and back to the sensor, enabling three-dimensional scene capture.

tracking algorithm A computational method that follows and predicts the position and movement of objects or features across a sequence of observations.

triangulation A process in computer vision that determines the three-dimensional location of a point by using its projections in two or more two-dimensional images.

undistortion The process of correcting lens distortion in images by applying the inverse of the estimated distortion model, producing images that follow the ideal pinhole camera model.

Video4Linux2 (V4L2) A Linux kernel interface for video capture and output devices, providing a standardized API for accessing video hardware.

worker pool A software design pattern where a collection of threads process tasks from a shared queue, enabling parallel execution of similar tasks.

zero-latency A system configuration or operational mode that minimizes processing and response delays to achieve near-instantaneous data processing and output, typically achieved through optimized algorithms, minimal buffering, and direct processing paths.

Erklärung zur selbständigen Bearbeitung

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

Ort	Datum	Unterschrift im Original