Silesian
University
of Technology

FINAL PROJECT

Smart 9-Ball Assistant

Sara SOBSTYL

Student identification number: ⟨**306088**⟩

Programme: Informatics

Specialisation: ⟨CGS⟩

SUPERVISOR

⟨**Dr. hab. inż. Jakub Nalepa**⟩

DEPARTMENT ⟨**Katedra Algorytmiki i Oprogramowania**⟩

Faculty of Automatic Control, Electronics and Computer Science

Gliwice 2025

Thesis title

Smart 9-Ball Assistant

Abstract

A project that helps players during a game of 9-ball billiards by showing them how the white ball would likely move after a shot. The system has two main parts: 1. Phone App on the Cue Stick - the app that uses the phone's motion sensors (like gyroscope and accelerometer) to track how the cue stick is held and moved. The app sends this data in real time to a computer. 2. Camera Above the Table + Computer - A camera above the billiard table sends live video to a computer. There, a program using e.g., OpenCV detects where all the balls are based on their colors. It figures out where the cue ball is and updates the layout of the table.

Key words

Computer Vision, Billiards, Trajectory Prediction, Augmented Reality, Object Detection

Tytuł pracy

Inteligentny asystent do bilarda

Streszczenie

Projekt wspomagający graczy podczas gry w bilard (odmiana 9-bil) poprzez wizualizację przewidywanego toru ruchu białej bili po uderzeniu. System składa się z dwóch głównych części: 1. Aplikacja na kiju bilardowym – wykorzystuje czujniki ruchu telefonu (takie jak żyroskop i akcelerometr) do śledzenia sposobu trzymania i ruchu kija. Aplikacja przesyła te dane w czasie rzeczywistym do komputera. 2. Kamera nad stołem i komputer – kamera umieszczona nad stołem bilardowym przesyła obraz wideo na żywo do komputera. Następnie oprogramowanie (wykorzystujące m.in. OpenCV) wykrywa położenie wszystkich bil na podstawie ich kolorów, lokalizuje białą bilę i aktualizuje układ stołu.

Słowa kluczowe

Wizja komputerowa, Bilard, Predykcja trajektorii, Rzeczywistość rozszerzona, Wykrywanie obiektów

Contents

Chapter 1

Introduction

Billiards, especially 9-ball, is a sport that demands a high level of precision, spatial awareness, and consistent motor control. Unlike many other sports where physical part is the most important, billiards is fundamentally a game that uses geometry and physics executed through trained muscle memory.

1.1 Introduction to the problem domain

Game of billiards focuses mainly on two skill sets: the ability to see the correct path of the balls and the physical ability to make the stroke that is needed to shoot the cue ball along that path. New players often have problem with the idea of the "Ghost Ball" [1] — the imaginary position where the cue ball must hit the object ball to pocket it.

Additionally, the way a player executes the stroke — including stance, bridge stability, and wrist movement can have a strong effect on the shot result. Even if a player aims correctly, but player introduces unwanted lateral acceleration or English[2] (spin) can result in a miss. With the development of accessible Computer Vision (CV), we are now able to better digitize these physical interactions to provide live feedback. [9]

1.2 Settling of the Problem in the Domain

Training methods that we have right now are mostly manual, relying on observations of coaches or player's own intuition. We do have professional tracking systems, but they are often quite expensive and need fixed, industrial-grade hardware. When it comes to mobile applications, they often rely solely on 2D video analysis, which doesn't have the depth perception that is needed for accurate table mapping or the high-frequency sampling that would be needed for detailed stroke analysis.

1.3 Objective of the Thesis

The main goal of this thesis is to design, implement, and test a "Smart Pool Assistant and Stroke Analyzer" The system would aim to assist player in real-time by visualizing shot outcomes and showing the analysis of the their cue strokes.

The primary objectives are defined as follows:

- To develop a Computer Vision module that is able to detect billiard balls and the cue stick in a live video feed.
- To implement a physics engine that calculates and visualizes the predicted trajectory of the cue ball (Tangent and Normal lines) based on the "Ghost Ball" principle.
- To create a mobile telemetry system that captures high-frequency accelerometer and gyroscope data from the player's phone.
- To derive important biomechanical metrics, such as impact force ($F = ma$) and wrist stability, to provide insight on the player's technique.

1.4 Scope of the thesis

This project covers the software and hardware needed to build a working prototype for the game of 9-ball.

- **Vision System:** The visual analysis uses only a top-down camera view. The software is written with Python and OpenCV, and uses Roboflow machine learning models for object detection. The trajectory prediction assumes perfect elastic collisions and focuses on the immediate paths of the cue ball and the target ball.
- **Sensor System:** The biomechanical part uses an Android smartphone sensors (Linear Acceleration and Gyroscope). This part also includes creating a TCP/IP protocol to send the data to the computer for processing.
- **Hardware:** The phone is connected to the computer via USB tethering. The mobile app is developed in Java using Android Studio.

1.5 Short description of chapters

The thesis is organized as follows:

Chapter 2 (Problem analysis) shows the theoretical and mathematical background for the project. It shows the physics behind collision, the geometry of "Ghost Ball" aiming and

the mechanics od sensors. It also presents solutions that already exist in sports technology to set the context.

Chapter 3 (Requirements and tools) defines the system's funcitonal and non-functional requirements. It also presents the chosen technology stack. Explains the use of Python, OpenCV, and Android platform.

Chapter 4 (External specification) describes the system from the user's perspective. It shows Android App interface and the visualization of the desktop interface. Also explains how user interacts with the system.

Chapter 5 (Internal specification) details the technical implementation and backend logic. Explains the algorithms that were used to detect objects, predict trajectory, for TCP/IP communication protocol and sensor data processing.

Chapter 6 (Verification and validation) presents testing results. Evaluates the computer vision model accuracy, the latency of live data streaming and how reliable are stroke analysis parameters through unit tests and real-world examples.

Chapter 7 (Conclusions) summarizes the thesis, discussing the achieved objectives and suggesting future improvements.

Chapter 2

Problem analysis

This chapter presents the theoretical and mathematical basis of the system. It explains the geometric challenges of aiming that were implemented in the vision module and shows the biomechanical rules principles behind the sensor analysis. It also reviews solutions that already exist.

2.1 Problem analysis

The games of 9-ball billiards provides a singular challenge in which a player must convert a 3D intention into a 2D geometric action. The main problems that are addressed by this project are divided into visual perception and biomechanical execution.

2.1.1 The Visual Paradox (Ghost Ball)

According to Alciatore, the ”Ghost Ball” concept is fundamental to understanding how to aim in billiards [1]. The player must visualize an imaginary position where the cue ball must contact the object ball to send it into the pocket. It's often hard to visualize it, because the aiming line, which passes through the ghost ball's center, is not the same as the actual contact point on the object ball. This mismatch requires players to aim at an invisible target, which creates a ”visual paradox”, that is especially hard for new players without any visual feedback.

2.1.2 Biomechanical Inconsistency

Even if an aiming line is correctly visualized, if the stroke is poorly executed, it can cause the shot to miss. So mechanics of the cue stroke are also really important.

Ballistic vs. Tetanic Stroke: According to Moore [18], there are two main types of strokes in billiards. Amateur players usually use ”ballistic” strokes, which are marked by a sudden pull at the start of the movement. It makes the speed control at impact

unreliable. In contrast, professional players aim for a ”tetanic” stroke, which characterizes by maintaining a continous muscle contraction that produces smooth, steady accelarion throught the cue delivery. [18]

Lateral Deviation: Another important thing is wrist stability during the stroke. If the player is not able to maintain a vertical ”pendulum swing” by keeping the elbow fixed, the cue tip can move away from the intented aiming line [18]. The most common errors are sideways wrist movement and inconsistent impact force.

2.2 State of the art

The field of ”Smart Sports” has significantly advanced in the past decades, with the use of Convolutional Neural Networks (CNN) and Wearable Sensors. [13]

2.2.1 Existing Solutions

- **Hawk-Eye (Computer Vision):** Widely used in sports like snoorker or tennis, this system uses multiple calibrated high-speed cameras to determine 3D ball positions. While being highly accurate ($<$ 3mm error), it requires expensive, fixed infrastructure and it doesn't analyze player biomechanics, but only ball trajectories. [11]

- **Projection AR Systems:** Solutions such as *PoolLiveAid* use overhead projectors to overlay aiming lines directly onto the table. Even tho it offers a really nice user experience it requires expensive hardware and complex calibration. [3]

- **Wearable IMU Analyzers:** There are devices such as *Blast Motion*, that are used in golf or baseball. They use inertial sensors to measure swing parameters. However, they do not analyze anything beside the swing. They monitor the movement, but they do not care about the game environment. For example, they don't know the position of the balls. [19]

2.3 Mathematical Models

The implementation of the system is based on mathematical models derived from vector algebra and Newtonian mechanics.

2.3.1 Ghost Ball Vector Math

The vision module calculates where the aiming vector would likely intersect the target's vall collision zone.

Let P_{cue} and P_{target} denote the centers of the cue ball and the target ball, and let $\vec{v}_{aim}$ be the normalized direction vector of the cue stick. The projection length L_{proj} of the target vector onto the aiming direction is computed using the dot product:

$$L_{proj} = (P_{target} - P_{cue}) \cdot \vec{v}_{aim} \tag{2.1}$$

The point on the aiming line closest to the target ball, denoted as P_{close}, is given by:

$$P_{close} = P_{cue} + \vec{v}_{aim} \cdot L_{proj} \tag{2.2}$$

A collision is valid only if the perpendicular distance $d_{\perp}$ is smaller than the ball diameter D:

$$d_{\perp} = ||P_{target} - P_{close}|| \tag{2.3}$$

If $d_{\perp} < D$, the Ghost Ball position P_{ghost} is obtained by moving back from P_{close} by an offset computed using the Pythagorean theorem:

$$P_{ghost} = P_{close} - \vec{v}_{aim} \cdot \sqrt{D^2 - d_{\perp}^2} \tag{2.4}$$

Thanks to this formula system draws the "Ghost Ball" circle at the exact position that the cue ball will be at the moment of impact.

2.3.2 Physics of the Stroke (Sensor Fusion)

The sensor module processes raw data to estimate the force of the shot.

Linear Acceleration

The Android `TYPE_LINEAR_ACCELERATION` sensor is used instead of `TYPE_ACCELEROMETER` to avoid the influence of gravity on user's movement. The resulting acceleration vector $\vec{a} = [a_x, a_y, a_z]$ represents the cue stick's actual acceleration. Its magnitude is given by:

$$||\vec{a}|| = \sqrt{a_x^2 + a_y^2 + a_z^2} \tag{2.5}$$

Force Estimation

To provide the player with a metric of "Shot Power" in Newtons, the system applies Newton's Second Law. Instead of relying on a fixed constant for the arm mass, the system dynamically calculates the effective mass of the player's arm (m_{arm}) based on the user's body weight and gender defined in the `config.json` file.

The effective mass is derived by summing the segmental weights of the hand, forearm, and upper arm using anthropometric coefficients from Plagenhoef et al. [20]. Consequently,

the impact force F is calculated as:

$$F = (W_{body} \cdot \mu_{gender}) \cdot ||\vec{a}_{impact}|| \tag{2.6}$$

Where W_{body} is the player's weight, and μ_{gender} is the effective arm mass coefficient (5.77% for males, 4.97% for females).

$$F = m_{arm} \cdot ||\vec{a}_{impact}|| \tag{2.7}$$

2.4 Algorithms

Two primary algorithmic domains are utilized to interpret the raw data streams: Computer Vision for game state analysis and Signal Processing for biomechanical analysis.

2.4.1 Computer Vision Algorithms

The vision module utilizes the YOLO (You Only Look Once) architecture, deployed in two distinct configurations to handle different tracking requirements.

Object Detection (Balls)

To identify the game elements, a standard object detection model is employed. It classifies objects into categories such as `cue-ball` and `other` (object balls). The inference function f_{det} transforms the input image I into a set of bounding boxes B:

$$f_{det}(I) \rightarrow \{(x, y, w, h, class, conf)_i\} \tag{2.8}$$

where (x, y) is the center of the ball, used as the input for the "Ghost Ball" geometric calculation.

Pose Estimation (Cue Stick)

Determining the aiming line requires more precision than a bounding box can provide. Therefore, a **Pose Estimation** model (specifically `yolov8-pose`) is used to detect the cue stick. Instead of a box, this model detects exact semantic keypoints: the `tip` (cue tip) and the `handle` (grip point). The output is a set of coordinates K:

$$f_{pose}(I) \rightarrow \{(x_{tip}, y_{tip}), (x_{handle}, y_{handle})\} \tag{2.9}$$

The aiming vector $\vec{v}_{aim}$ is then derived directly from these keypoints: $\vec{v}_{aim} = (x_{tip} - x_{handle}, y_{tip} - y_{handle})$.

2.4.2 Peak Detection Algorithm

To automatically detect the moment of contact without external triggers, the system implements a real-time peak detection algorithm on the linear acceleration data stream. A hit is registered at time t if the acceleration magnitude $a(t)$ satisfies both a threshold condition and a local maximum condition:

$$a(t) > T_{peak} \quad \wedge \quad a(t) \geq a(t-1) \quad \wedge \quad a(t) \geq a(t+1) \tag{2.10}$$

where T_{peak} is the noise gate threshold (set to $15.0 m/s^2$ in the implementation). This makes sure that the system captures the exact moment of maximum force exertion (F_{max}) for the biomechanical analysis.

Chapter 3

Requirements and Tools

This chapter details the functional and non-functional requirements and presents the technological stack selected for implementation. It also describes the system's use cases and the method adopted for the design and development process.

3.1 Functional and Non-functional Requirements

The system requirements were defined to address the problems identified in the previous chapter, specifically the "ghost ball" visualization and the biomechanical analysis of the stroke.

3.1.1 Functional Requirements

The functional requirements define the specific behaviors and functions the system must support. They are categorized by module:

Vision Module (Desktop):

- **FR-01 Video acquisition:** The system must capture a real-time video stream from a webcam positioned above the billiard table.
- **FR-02 Object detection:** The system must detect and classify the cue ball and object balls using a convolutional neural network (yolo).
- **FR-03 Pose estimation:** The system must identify keypoints of the cue stick (tip and handle) to determine the aiming vector.
- **FR-04 Trajectory prediction:** The system must calculate the "ghost ball" position and predict the trajectories of the cue ball and target ball based on geometric rules.
- **FR-05 AR visualization:** The system must overlay the predicted paths and the ghost ball indicator onto the video feed in real-time.

Sensor Module (Mobile):

- **FR-06 Data acquisition:** The mobile application must read data from the accelerometer (linear acceleration) and gyroscope at a frequency of at least 50 Hz.
- **FR-07 Data transmission:** The mobile app must transmit sensor data to the desktop server via a TCP/IP socket connection (USB tethering).
- **FR-08 Stroke analysis:** The system must detect the moment of impact (peak acceleration) and calculate the impact force ($f = ma$) and wrist rotation stability.

3.1.2 Non-functional Requirements

The non-functional requirements define the quality attributes of the system:

- **NFR-01 Real-time performance:** The vision processing pipeline must maintain a frame rate of at least 20 fps (frames per second) to provide smooth visual feedback.
- **NFR-02 Latency:** The latency between the physical cue movement and the AR update should be less than 150 ms to ensure the user perceives the lines as responsive.
- **NFR-03 Accuracy:** The "ghost ball" projection error should be less than 5% of the ball diameter to be practically useful for aiming.
- **NFR-04 Sensor synchronization:** The time drift between the video impact detection and sensor peak detection must be handled to correctly associate the physical stroke with the visual event.
- **NFR-05 Usability:** The setup process (camera alignment, phone connection) should be performable by a single user within 5 minutes.

3.2 Use Cases

The interaction between the user (player) and the system is shown by a number of essential use cases. The primary actor is the player, who interacts with both the physical equipment (phone, cue) and the software (desktop app).

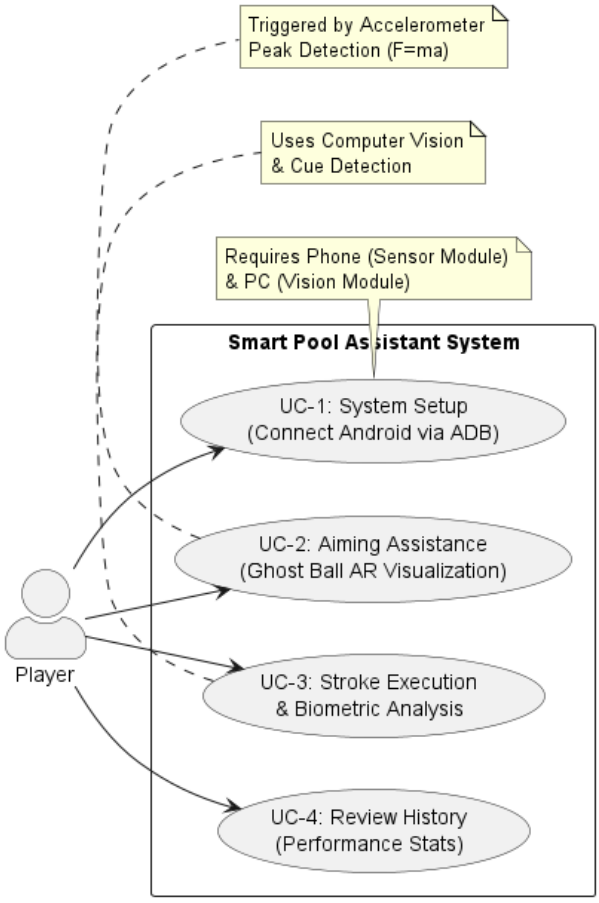


Figure 3.1: Use case diagram

The main use cases are defined as follows:

1. **UC-1 System setup:** The player connects the Android phone to the PC (via USB/ADB), starts the server script, and launches the mobile app. The connection is established if the ”connected” status appears.

2. **UC-2 Aiming assistance:** The player addresses the cue ball. The system detects the cue stick, calculates the trajectory, and projects the ”ghost ball” and aiming lines. The player adjusts their stance based on this visual feedback.

3. **UC-3 Stroke execution & analysis:** The player executes the shot. The system automatically detects the impact via the sensor stream, captures a snapshot of the metrics (force, rotation), and saves the data to a history log.

4. **UC-4 Review history:** The player views the saved ”hit history” (csv/images) to analyze their consistency over the training session.

3.3 Description of Tools

The project applies a modern technology stack combining computer vision, AI, and mobile development.

3.3.1 Hardware tools

- **Webcam:** A standard HD webcam is used for video input. It is mounted in a "top-down" configuration to minimize perspective distortion.
- **Android smartphone:** A device equipped with an inertial measurement unit (IMU). It works as the telemetry unit attached to the player's cue.
- **Workstation (PC):** A computer with access to the internet and with installed Python 3.x environment to run the vision server and process data.

3.3.2 Software tools

- **Python 3.x:** The main programming language for the backend server and vision processing. [10]
- **OpenCV (open source computer vision library):** Used for image pre-processing, drawing the AR visualization (lines, circles), and rendering the live telemetry graphs. [8]
- **Ultralytics YOLO (You Only Look Once):** A state-of-the-art framework for object detection.
  - *yolov12 (detection):* Used for robust detection of billiard balls under different lighting conditions. [23]
  - *yolov8-pose (keypoint estimation):* Specifically used to detect the `tip` and `handle` of the cue stick, allowing accurate vector calculation. [15]
- **Roboflow:** A platform used for dataset management, image annotation, and versioning. It facilitated the preparation of the training data for the custom billiard model. [12]
- **Android Studio & Java:** The development environment for the mobile sensor application. Java was chosen for its native support of Android sensor APIs. [21]
- **ADB (Android Debug Bridge):** Utilized for establishing a low-latency reverse TCP connection (`adb reverse`) between the Android device and the localhost server. [4]

3.4 Methodology of design and implementation

The development followed an iterative **prototyping methodology** [16], which is suitable for systems that include experimental algorithms (like computer vision) and hardware integration. The process was divided into four phases:

3.4.1 Phase 1: Data collection and model training

The initial phase focused on the preparation of the Computer Vision module. Two separate custom datasets were curated and annotated using the Roboflow platform [12]: one for billiard balls and one for the cue stick.

For the **Ball Detection model**, a dataset of overhead table images was collected. As shown in the class distribution analysis (see Figure 3.2), the dataset reflects the natural disproportion of the game: the ”other” class (object balls 1–15) is significantly more frequent than the ”cue ball” class. To reduce possible overfitting as weel as improve model dependability under different lighting conditions, image augmentation procedures (including rotation, brightness adjustment, and noise augmentation) were applied during the preprocessing stage.

A similar process was followed for the **Cue Detection model**, where the dataset was annotated with keypoints (skeleton) to define the tip and handle positions for the pose estimation task. The class distribution for this dataset (see Figure 3.3) focuses on a single class, ”cue”, with corresponding bounding box dimensions and instance counts showing the nature of cue stick detection.

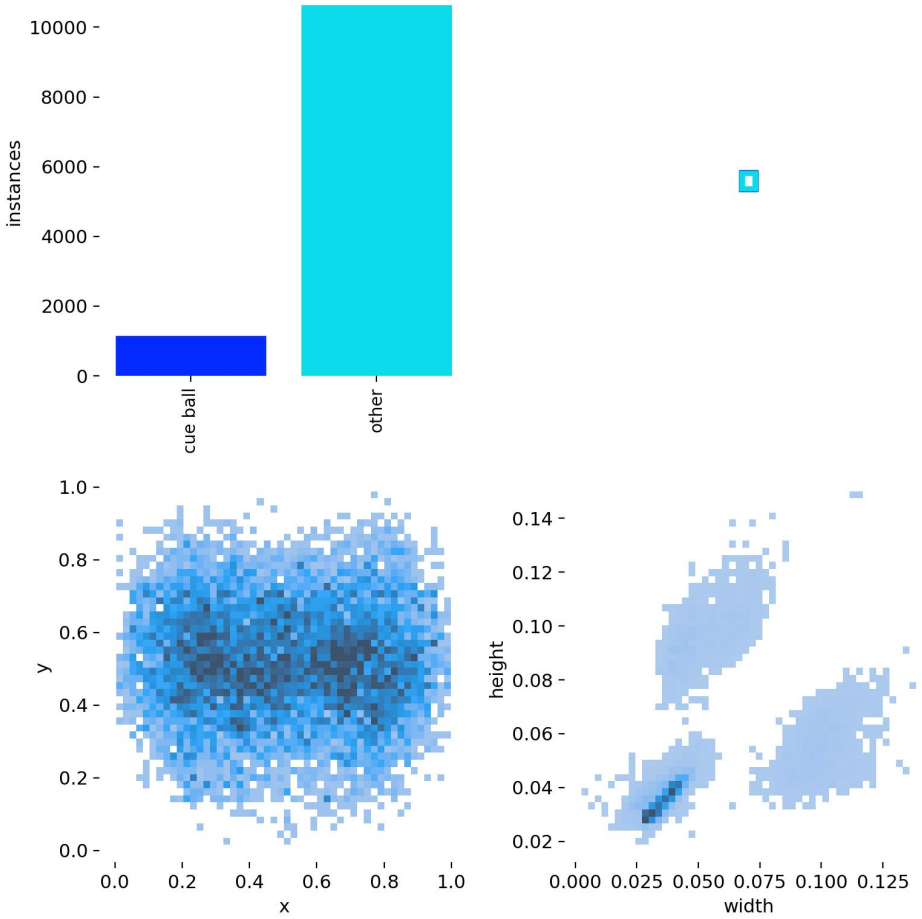


Figure 3.2: Class distribution in the ball detection training dataset, highlighting the imbalance between the single 'cue ball' and multiple 'other' balls.

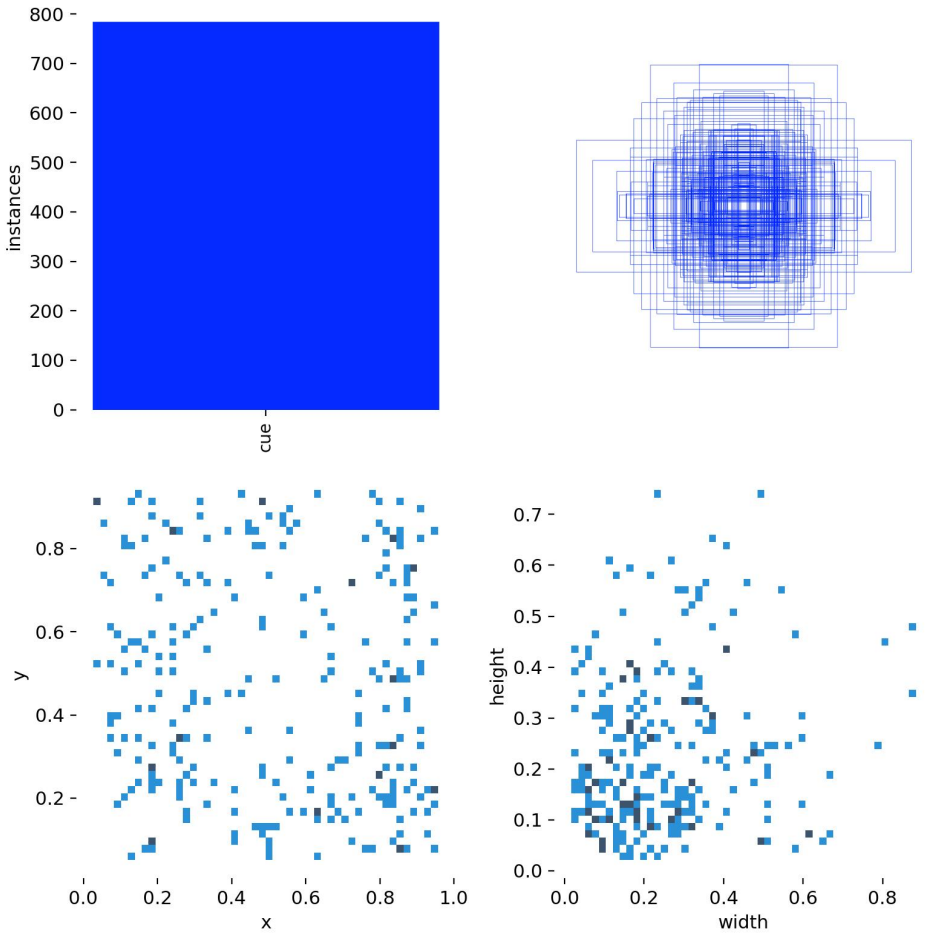


Figure 3.3: Class distribution and instance analysis in the cue detection training dataset.

3.4.2 Phase 2: Vision logic implementation

Once the models were trained, the Python logic was developed to translate raw detections into game state information. This involved implementing the vector algebra for the ”ghost ball” [1] algorithm and using OpenCV [8] to render the visual overlays.

3.4.3 Phase 3: Sensor module development

The Android application was developed to reliably extract linear acceleration and gyroscope data. The focus was on implementing the `TYPE_LINEAR_ACCELERATION` [5] sensor to filter out gravity and developing a stable TCP socket protocol for data transmission.

Chapter 4

External specification

This chapter provides a detailed description of the system from the user's perspective. It covers the installation requirements, setup procedures, and a walkthrough of typical usage scenarios, including the user interface (UI) and system output.

4.1 Hardware and software requirements

To ensure the correct operation of the "Smart Pool Assistant", the following minimum requirements must be met.

4.1.1 Hardware Requirements

- **Desktop Workstation (Server):**
  - CPU: Intel Core i5 (8th Gen) / AMD Ryzen 5 or better.
  - RAM: Minimum 8 GB (16 GB recommended for smooth video processing).
  - GPU: Dedicated NVIDIA GPU (GTX 1050 or better) recommended for YOLO inference, though CPU-only execution is supported.
  - Ports: At least one USB 3.0 port (the phone connection).
- **Video Input:**
  - HD Webcam (720p minimum resolution) mounted on a tripod or ceiling bracket directly above the pool table.
- **Mobile Device (Client):**
  - Smartphone running Android 9.0 (Pie) or higher.
  - Must contain hardware sensors: Accelerometer and Gyroscope.
  - USB cable for data tethering.

4.1.2 Software Requirements

- **PC Operating System:** Windows 10/11 or Linux.
- **Python Environment:** Python 3.8 or newer.
- **Android Environment:** Android device with "Developer Options" and "USB Debugging" enabled.
- **Dependencies:** The required Python libraries are listed in the `req.txt` file (e.g., `ultralytics`, `opencv-python`, `inference`, `supervision`).

4.2 Installation procedure

The installation process is twofold, involving the desktop server and the mobile client.

4.2.1 Desktop Installation

1. **Clone the Repository:** Download the project source code from the official GitHub repository. You can download the ZIP file directly:

   `https://github.com/ss19190/Smart-9-Ball-Assistant`

   Or via terminal:

   ```
   git clone https://github.com/ss19190/Smart-9-Ball-Assistant
   ```

2. **Install Dependencies:** Open a terminal in the project folder and run:

   ```
   pip install -r requirements.txt
   ```

3. **Install ADB:** Ensure the Android Debug Bridge (ADB) is installed and added to the system PATH variables.

4. **Configure API Keys:** You need a valid API Key to access the trained models.

   - Create a free account at `https://roboflow.com`.
   - Navigate to **Settings** → **API** (or your Workspace Settings).
   - Copy your `Private API Key`.

   Open a `.env` file in the root directory of the project and paste the key:

   ```
   API_KEY=your_copied_key_here
   ```

5. **Configure Player Metrics:** To ensure accurate impact force estimations ($F = ma$), the system requires the player's anthropometric data to calculate the effective arm mass. Open the `config.json` file in the root directory and update the values:

```
{
    "body_weight_kg": 54,
    "gender": "female"
}
```

Note: The `gender` *field accepts "male" or "female" (case-insensitive). This setting adjusts the arm mass coefficient used in the biomechanical calculations.*

4.2.2 Mobile Installation

1. **Enable Developer Mode:** On the Android phone, go to **Settings** → **About Phone**. Find the "Build Number" entry and tap it 7 times repeatedly until you see a message "You are now a developer!".

2. **Enable USB Debugging:** Go back to the main Settings menu (or System → Advanced), open **Developer Options**, and turn on the switch for **USB Debugging** to ON. Confirm the security prompt if requested.

3. **Install the App:** Locate the `SensorApp.apk` file in the project directory. You can install it manually by copying it to the phone, or simpler, by running this command in your PC terminal (while the phone is connected):

```
adb install SensorApp.apk
```

Ensure you allow installation from unknown sources if prompted on the device.

4.3 Activation procedure

To start a training session, the user must follow a specific sequence of steps to establish the client-server connection.

1. **Connect Hardware:** Plug the webcam into the PC and connect the Android phone via USB.

2. **Setup ADB Forwarding:** This step is important for the phone to communicate with the PC via localhost. Run the following command in the terminal:

```
adb reverse tcp:5555 tcp:5555
```

3. **Start the Sensor Server:** Run the script responsible for handling telemetry data:

```
python sensors.py
```

4. **Start the Vision Assistant:** Open a second terminal window and run the AR visualization script:

```
python bilard.py
```

5. **Connect the App:** Launch the app on the phone and tap the "CONNECT" button. The status should change to green "CONNECTED".

4.4 Types of users

The system is designed for a single type of user, although the roles can be conceptually divided:

- **The Player (Trainee):** The primary user who wears the sensor and performs the shots. They interact with the physical game and view the feedback on the screen.
- **The Operator (Optional):** A second person (e.g., a coach) who manages the software, starts/stops the scripts, and analyzes the history logs, though the Player can perform these tasks themselves.

4.5 User manual

4.5.1 Main Interface (Desktop)

The desktop window displays the live camera feed overlaid with augmented reality elements.

- **Ghost Ball Indicator:** A white circle appearing on the aiming line, showing where the cue ball will be at impact.
- **Trajectory Lines:**
  - *Gray Line:* The aiming line extending from the cue stick.
  - *Yellow Line:* The predicted path of the cue ball after impact (Tangent Line).
  - *Green Line:* The predicted path of the object ball.

4.5.2 Mobile Interface

The mobile app interface is minimal to avoid distraction.

- **Status Bar:** Shows connection state (Disconnected/Connected).
- **Real-time Values:** Displays raw linear acceleration (X, Y, Z) for debugging purposes.
- **Connect Button:** Toggles the TCP connection.

4.6 System administration

Since this is a standalone prototype, system administration tasks are limited to:

- **Log Management:** The system generates CSV files (`hit_history.csv`) and image snapshots in the `saved_graphs/` folder. The user should periodically archive or delete old files to save disk space.
- **Calibration:** If the "Ghost Ball" accuracy drifts, the user may need to verify the `BALL_DIAMETER_PX` constant in `bilard.py` to match the current camera height.

4.7 Security issues

- **Network Security:** The communication uses unencrypted TCP sockets. This is acceptable for a local USB connection (ADB forwarding), but if ever was implemented and used over public Wi-Fi, the data stream could be intercepted.
- **API Keys:** The Roboflow API key is stored in a `.env` file. This file should not be shared or committed to public version control repositories.

4.8 Example of usage

Scenario: Practicing the Cut Shot The player wants to practice a difficult cut shot into the corner pocket.

1. The player sets up the white ball and the target ball.
2. The system detects the balls and waits for the cue stick.
3. The player addresses the cue ball. The system detects the cue pose and renders the "Ghost Ball" slightly to the left of the target ball.

4. The AR lines show that the current aim will cause the target ball to miss the pocket (hit the rail).

5. The player adjusts their stance until the Green Line points directly into the pocket.

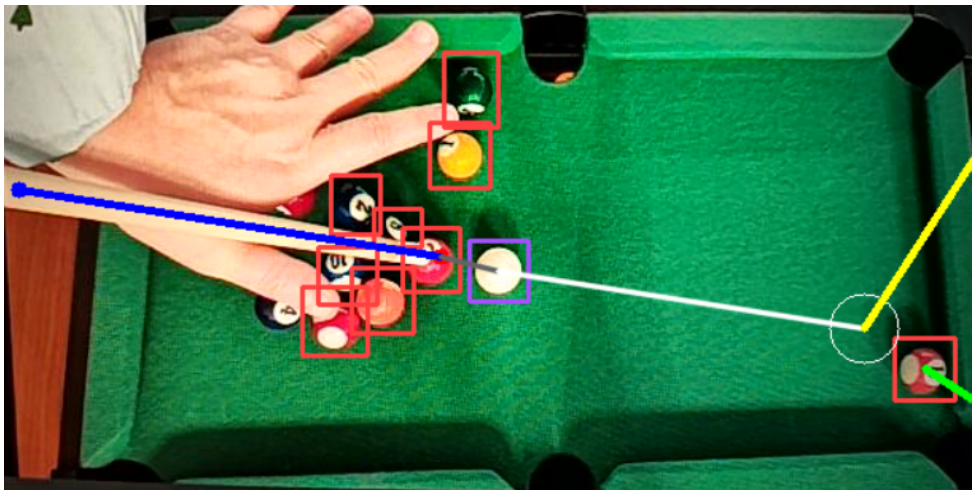
6. The player executes the stroke.

7. The `sensors.py` script detects a peak acceleration of $25 m/s^2$.

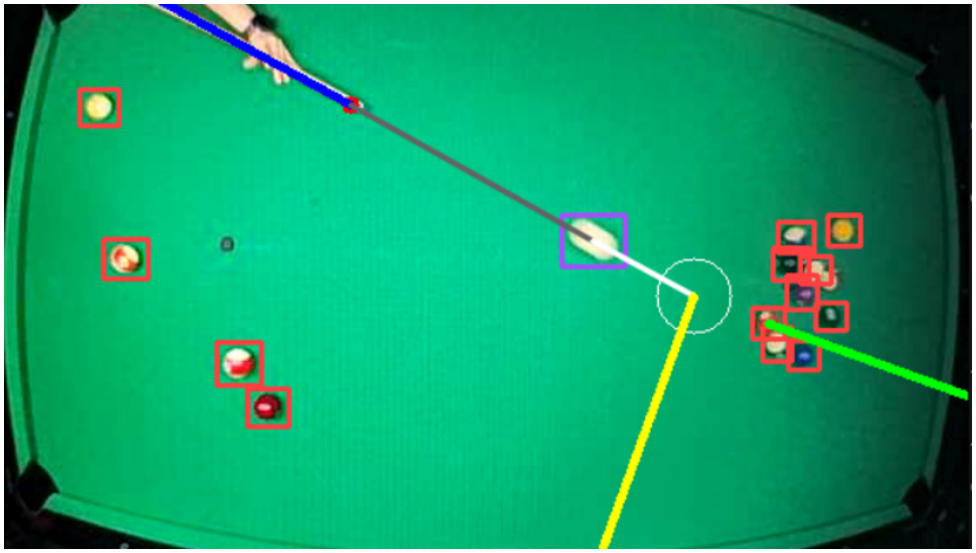
8. The system saves a snapshot of the stroke metrics, allowing the player to see if they maintained wrist steadiness during the shot.

4.9 Working scenarios

This section illustrates the system in action with screenshots from the prototype.



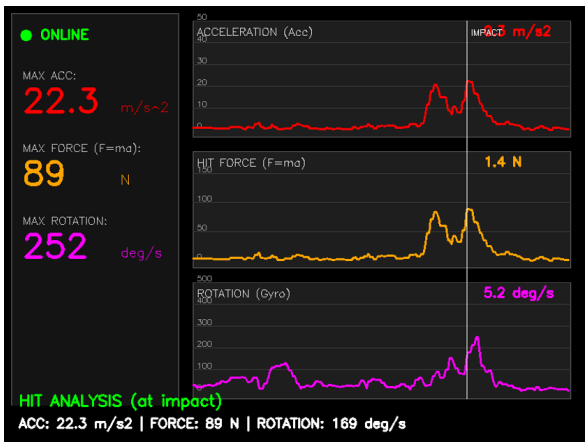
(a) Visualization on a small table.



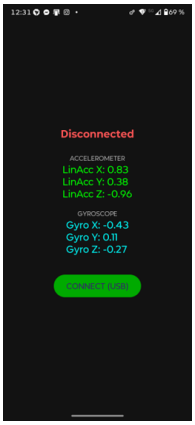
(b) Visualization on a standard pool table.

Figure 4.1: Scenario A: The Vision Assistant interface visualizing a valid collision. The system calculates and displays the predicted outcome in real-time.

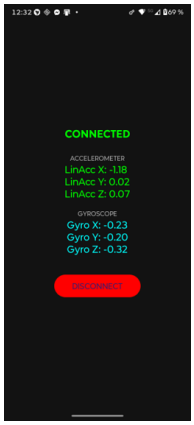
(a) Desktop interface in a state before a stroke.



(b) Desktop interface showing impact analysis showing max force of 89 N.



(c) Mobile app in disconnected state.



(d) Mobile app connected, streaming sensor data.

Figure 4.2: Scenario B: Stroke Analysis workflow. The top row shows the PC application analyzing the stroke data, where (a) is pre-hit and (b) is post-hit showing impact metrics. The bottom row shows two possible states of the mobile sensor app (c, d).

Chapter 5

Internal Specification

This chapter presents the technical details of the solution, including the system architecture, key data structures, and a description of the implementation of the most critical modules in Python and Java.

5.1 System Concept

The system concept is based on the fusion of sensory and visual data to assist in playing billiards. The system consists of:

1. **Vision Module (AI):** Analyzes the camera feed, detects billiard balls and the cue, and subsequently calculates the "Ghost Ball"—the predicted position of the cue ball at the moment of impact.

2. **Telemetry Module:** Analyzes the force of the impact and the smoothness of the player's movement based on IMU sensor data (accelerometer, gyroscope) transmitted via USB cable from a smartphone.

5.2 System Architecture

The system operates on a client-server architecture. The smartphone (Client) collects data and transmits it via TCP to the workstation (Server), which processes the camera image in parallel.

The main unit (PC) runs two independent processes (Python scripts):

- **Vision Process (`bilard.py`):** Utilizes the `inference` and `supervision` libraries for object detection and vector calculations.

- **Sensor Server (`sensors.py`):** A multi-threaded TCP server that receives JSON data, parses it, and visualizes graphs using the `OpenCV` library.

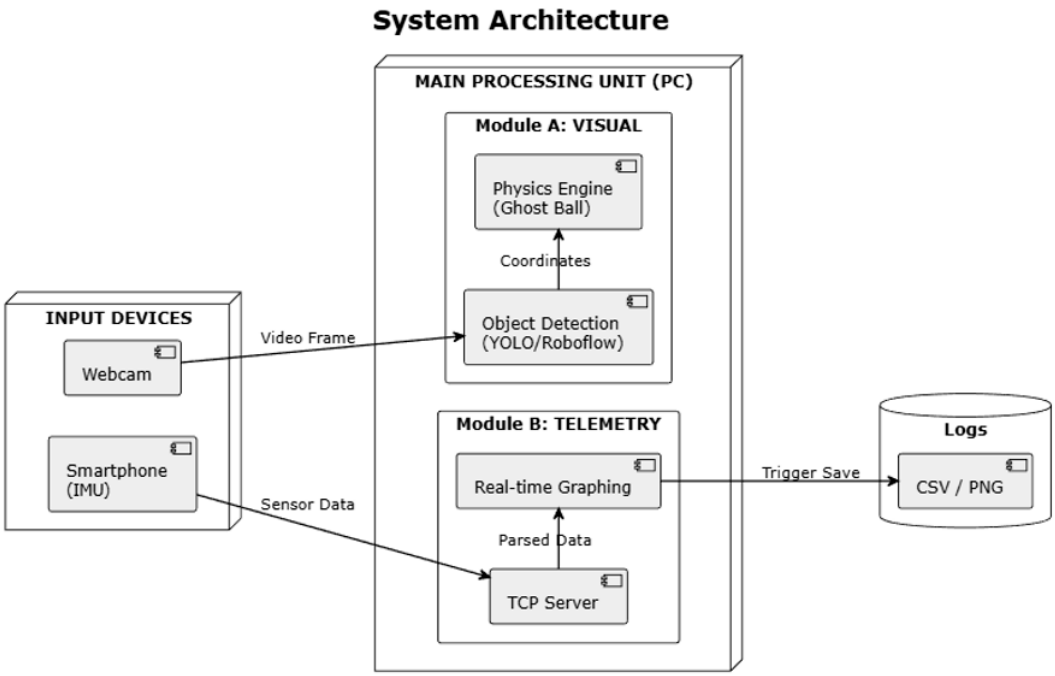


Figure 5.1: System Architecture Diagram and logic flow.

5.3 Description of Data Structures

5.3.1 Network Payload (JSON)

Communication between the Android device and the PC is handled using the JSON format. A main feature is the transmission of linear acceleration (gravity excluded) and angular velocity.

```
// JSON format generated in MainActivity.java
String json = String.format(Locale.US,
    "{\"acc_x\":␣%.4f,␣\"acc_y\":␣%.4f,␣\"acc_z\":␣%.4f,␣" +
    "\"gyro_x\":␣%.4f,␣\"gyro_y\":␣%.4f,␣\"gyro_z\":␣%.4f}",
    ax, ay, az, gx, gy, gz);
```

Figure 5.2: JSON structure creation in Java (Android).

5.3.2 Physics Vectors (Python)

The vision module operates on vectors from the `NumPy` library to compute trajectories.

```
# Position representation in bilard.py
cue_ball_center = np.array([x, y], dtype=np.float32)
aim_vector = normalize_vector(tip_pos - handle_pos)
```

5.4 Components and Modules

5.4.1 Mobile Module (Android)

The mobile application was developed in Java. The key component is the `SensorManager`. The `TYPE_LINEAR_ACCELERATION` sensor was utilized instead of the standard accelerometer to eliminate the influence of gravity on the impact force measurement.

```
// MainActivity.java - sensor initialization
linearAcceleration = sensorManager.getDefaultSensor(
    Sensor.TYPE_LINEAR_ACCELERATION
);
```

5.4.2 Vision Module (Computer Vision)

This module utilizes two separate neural network models hosted on the Roboflow platform, optimized for instant inference:

- **Ball Detection:** A generic object detection model (version `ball-detection-bzirz/3`) based on the state-of-the-art **YOLOv12** [23] architecture. It is trained to localize billiard balls and classifies objects into two distinct classes: `cue ball` and `other`. The model works on a standard input resolution (640×640) and outputs bounding box coordinates.

- **Cue Detection:** A pose estimation model (version `cue-detection-ciazj/3`) utilizing the **YOLOv8-pose** [15] architecture. Unlike the ball detector, this model outputs semantic keypoints representing the `tip` and `handle`, necessary for calculating the aiming vector.

5.5 Overview of Key Algorithms

5.5.1 Ghost Ball Algorithm

The implementation translates the vector mathematics and geometric rules defined in Section 2.3.1 into Python code using the `NumPy` library. The function `find_ghost_ball_position` calculates the projection vector and validates the perpendicular distance to render the visual aid.

```python
def find_ghost_ball_position(cue_ball_pos, aim_vector, target_ball_pos):
    vec_to_target = target_ball_pos - cue_ball_pos
    projection_length = np.dot(vec_to_target, aim_vector)

    # Calculating the closest point on the shot line
    closest_point = cue_ball_pos + aim_vector * projection_length
    perp_dist = np.linalg.norm(target_ball_pos - closest_point)

    # Correction for ball radius (backward offset)
    if perp_dist < BALL_DIAMETER_PX:
        back_offset = math.sqrt(BALL_DIAMETER_PX**2 - perp_dist**2)
        dist_impact = projection_length - back_offset
        return cue_ball_pos + aim_vector * dist_impact
```

Figure 5.3: Vector-based Ghost Ball calculation implementation.

5.5.2 Impact Force Calculation (Physics)

The `sensors.py` module implements the force estimation model derived in Section 2.3.2. To ensure flexibility, the system does not use hard-coded anthropometric constants in the source code. Instead, it dynamically loads the user's attributes (weight and gender) from a `config.json` file.

Based on these attributes, the system selects the appropriate effective mass coefficient (μ_{gender})—as detailed in the theoretical analysis (see Section 2.3.2)—and computes the personalized arm mass (m_{arm}). The final impact force is then derived by applying the calculated mass to the linear acceleration vector received from the mobile device.

Additionally, the algorithm implements a peak detection logic (refer to Section 2.4.2) that specifically identifies the *second* significant peak in the signal buffer. This implementation detail allows the system to distinguish the actual forward strike from the preparatory backswing movement.

5.6 Implementation Details

5.6.1 Thread Synchronization (Python)

The TCP server in `sensors.py` operates in a separate thread (`daemon=True`) to avoid blocking the interface drawing loop (GUI) in the OpenCV library, if someone would like to run both moduls at the same time.

```python
# sensors.py - Threading implementation
threading.Thread(target=tcp_server_thread, daemon=True).start()

while True:
```

```
5     # Main GUI Loop (OpenCV)
6     cv2.imshow("Hit Analysis", window)
7     if cv2.waitKey(20) & 0xFF == ord('q'): break
```

5.6.2 Network Handling (Android)

In the Android application, a separate thread was employed for network operations to avoid the `NetworkOnMainThreadException`. Data is transmitted at a frequency of approximately 100Hz (every 10ms), ensuring smooth physics representation.

5.7 Applied Design Patterns

- **Listener Pattern (Android):** The implementation if the `SensorEventListener` interface enables reactive handling of sensor value changes only when they occur. [17]
- **Producer-Consumer:** The TCP thread (Producer) receives data and updates the global `sensor_data` structure, which is subsequently consumed by the main visualization loop.[17]

Chapter 6

Verification and Validation

This chapter describes the testing methods adopted to maintain the consistency and accuracy of the developed system. It covers the testing paradigm, the scope of test cases, a detailed analysis of encountered software defects, and the final experimental results.

6.1 Testing Paradigm

To ensure an organized approach to verification, the **V-Model** (Verification and Validation Model) was adopted. This model focuses on the relationship between the design phase and the testing phase. The testing process was divided into three distinct levels:

1. **Unit Testing:** Individual components were tested in isolation. For the Python server, this involved testing the vector calculation functions (e.g., `find_ghost_ball_position`) and the impact force formula ($F = ma$) with known static values to ensure mathematical correctness.

2. **Integration Testing:** This phase focused on the communication between the Android Client and the PC Server. Key tests included verifying the TCP handshake, the serialization of sensor data into JSON format, and the handling of network latency.

3. **System Testing:** The full system was validated in a real-world environment. This involved a player performing actual shots while the system tracked the game state and visualized the analytics in real-time.

6.2 Testing Scope and Test Cases

The testing scope covered both the functional requirements (correct physics calculations, correct object detection) and non-functional requirements (performance, latency). Table 6.1 summarizes the key test scenarios.

Table 6.1: Summary of Key Test Cases.

ID	Test Scenario	Expected Result	Status
TC-01	JSON Packet Transmission	Server receives valid JSON structure without corruption.	Passed
TC-02	Ghost Ball Projection	Projector line intersects the target ball center.	Passed
TC-03	High-Velocity Shot Detection	Accelerometer detects peak > 20 m/s^2.	Passed
TC-04	Multi-ball Occlusion	System maintains ID of balls when partially obscured.	Partial

6.3 Detected and Fixed Bugs

During the implementation and testing phases, multile significant software defects were identified. Below is a description of the most significant bugs and the solutions applied.

6.3.1 Android UI Thread Blocking (UI Lag)

Problem: The initial version of the Android application exhibited severe interface lag and unresponsiveness during data recording. The accelerometer sensor was configured to `SENSOR_DELAY_FASTEST`, generating events at approximately 100Hz. The application attempted to update the on-screen `TextView` elements inside every sensor callback. Since the UI rendering thread operates at a lower frequency (approx. 60Hz), the event queue became overwhelmed, causing the displayed values to ”trail” behind reality (e.g., the value would continue rising long after the phone had stopped moving).

Solution: The implementation was modified to decouple data collection from UI updates. A throttling mechanism was introduced to update the UI elements only once every 100ms (10Hz), while the data transmission to the server continued at the full 100Hz rate using a background thread.

6.3.2 TCP Stream Fragmentation

Problem: Occasionally, the Python server would crash with a `JSONDecodeError`. This occurred because TCP is a stream-oriented protocol, not a message-oriented one. At high transmission rates, multiple JSON objects were merged into a single buffer read, or a single JSON object was split across two reads.

Solution: A delimiter (newline character `\n`) was appended to every message sent from Android. On the server side, a buffer was implemented to accumulate incoming bytes until a full delimiter was found, making sure that only complete JSON strings were passed to the parser.

6.3.3 Ghost Ball Jitter

Problem: The detected position of the cue ball and cue tip fluctuated slightly (by 1-2 pixels) due to noise in the camera, even when the objects were stationary. This caused the projected "Ghost Ball" vector to shake rapidly, making it difficult for the user to aim.

Solution: A moving average filter (size $N = 5$) was applied to the coordinates of the detected objects. This smoothed out the high-rate noise without losing acceptable responsiveness to intentional movements.

6.4 Computer Vision Model Evaluation

To ensure the reliability of the system, the trained neural networks were subjected to rigorous assessment a hold-out test set [7]. This section presents the evaluation metrics for both the Ball Detection (**YOLOv12**) and Cue Detection (**YOLOv8-pose**) models.

6.4.1 Ball Detection Model Results

The Ball Detection model, built on the **YOLOv12** architecture, showed outstanding performance, obtaining a **mean Average Precision (mAP@0.5)** of **0.991**, indicating nearly perfect detection abilities at a standard Intersection over Union (IoU) threshold (see Figure 6.1).

- **Training Performance:** As illustrated in the training plots, the box loss and classification loss converged rapidly within the first 50 epochs, stabilizing with no signs of overfitting. The precision and recall metrics consistently improved, plateauing near 1.0.

- **Confusion Matrix:** The confusion matrix (Figure 6.2) confirms the model's accuracy on the test set. The model correctly identified **110** Cue Ball instances and **1043** Object Balls ("other"). Misclassifications were negligible, with only minimal instances of background falsely detected as balls (False Positives).

- **F1-Score:** The F1-Confidence curve (Figure 6.3) shows an optimal confidence threshold of **0.660**, at which the model reaches an F1 score of **0.99**. This threshold was subsequently used in the production code to filter weak predictions.

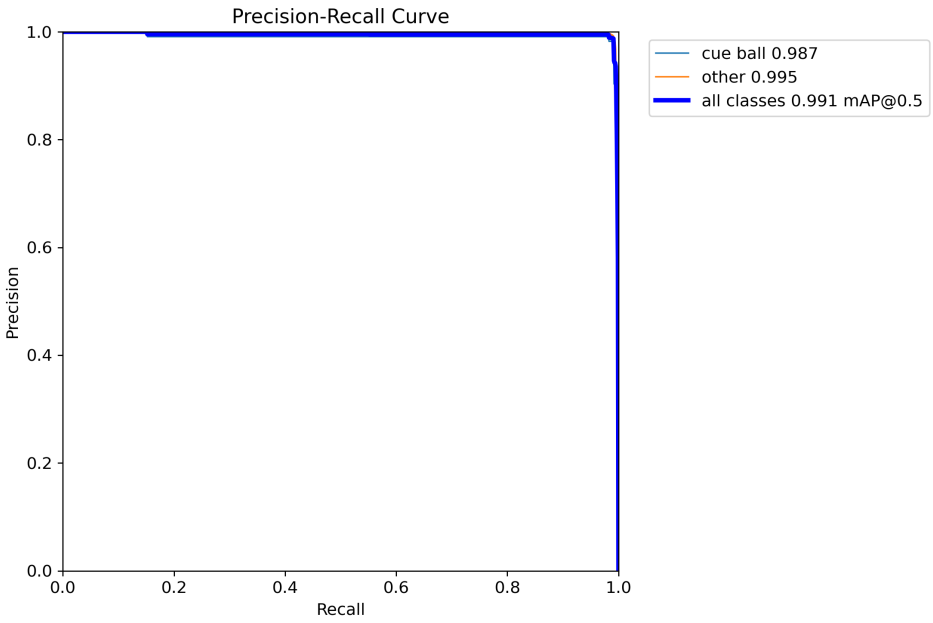


Figure 6.1: Precision-Recall Curve showing mAP@0.5 of 0.991 for the Ball Detection model.

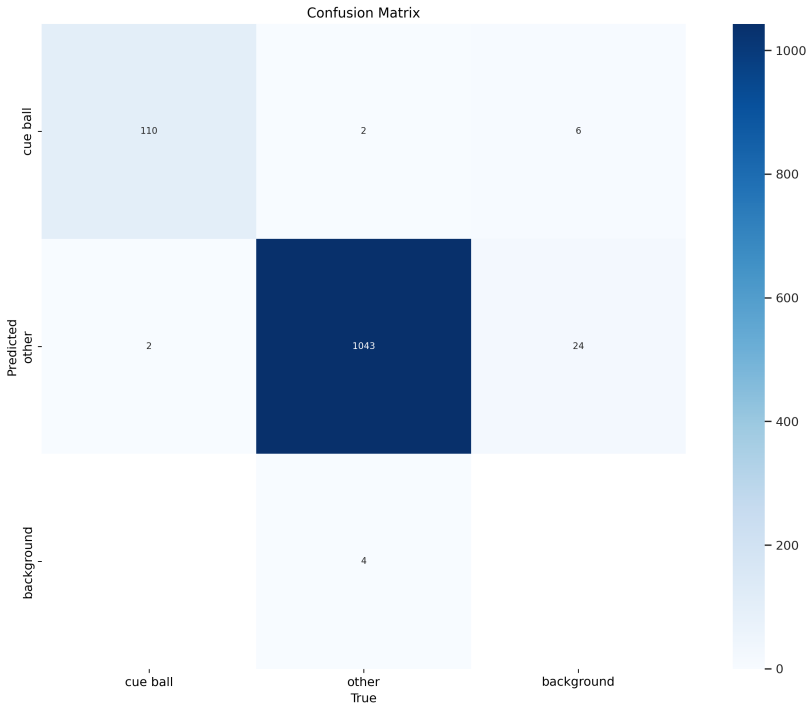


Figure 6.2: Confusion Matrix demonstrating raw detection counts for Cue Ball, Other, and Background classes.

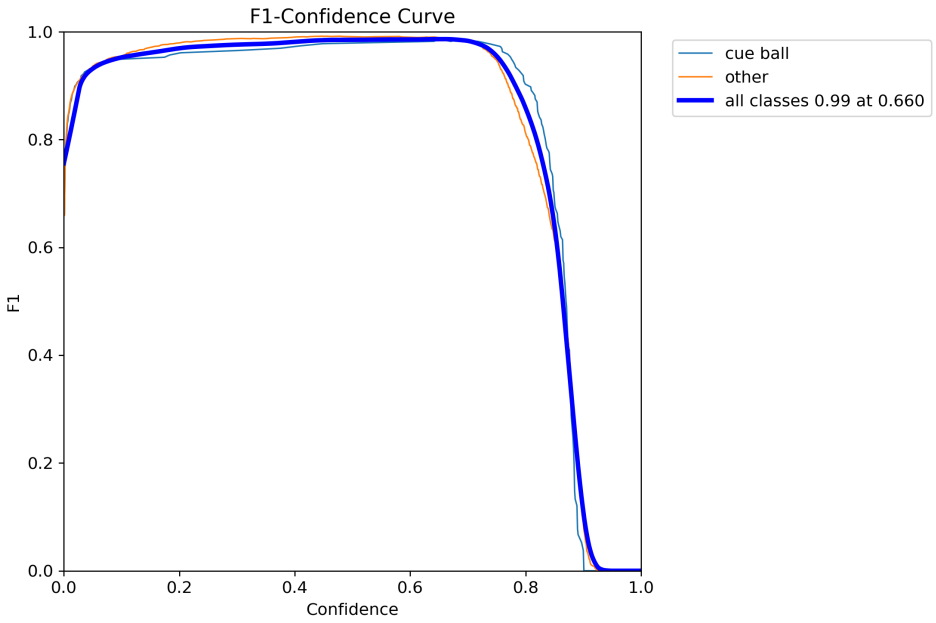


Figure 6.3: F1-Confidence Curve indicating the optimal confidence threshold of 0.660.

6.4.2 Cue Detection Model Results

The Cue Detection model, utilizing pose estimation (YOLOv8-pose), was evaluated based on both bounding box detection and, more importantly, the accuracy of keypoint localization (Object Keypoint Similarity). Accurate keypoint detection is critical for determining the vector of the cue stick.

- **Pose Estimation Accuracy:** The model reached a **Pose mAP@0.5** of **0.970** (see Figure 6.4), indicating high precision in locating the cue tip and handle keypoints. The bounding box detection also performed well with a mAP@0.5 of 0.954.

- **F1-Score Analysis:** The F1-Confidence curve for pose estimation (Figure 6.5) identifies an optimal confidence threshold of **0.591**, where the model obtains a peak F1 score of **0.93**.

- **Classification Performance:** The confusion matrix (Figure 6.6) displays the model's ability to distinguish the cue from the table background. Out of 76 actual cue instances in the validation set, the model correctly identified 70, with only a minor number of background false positives.

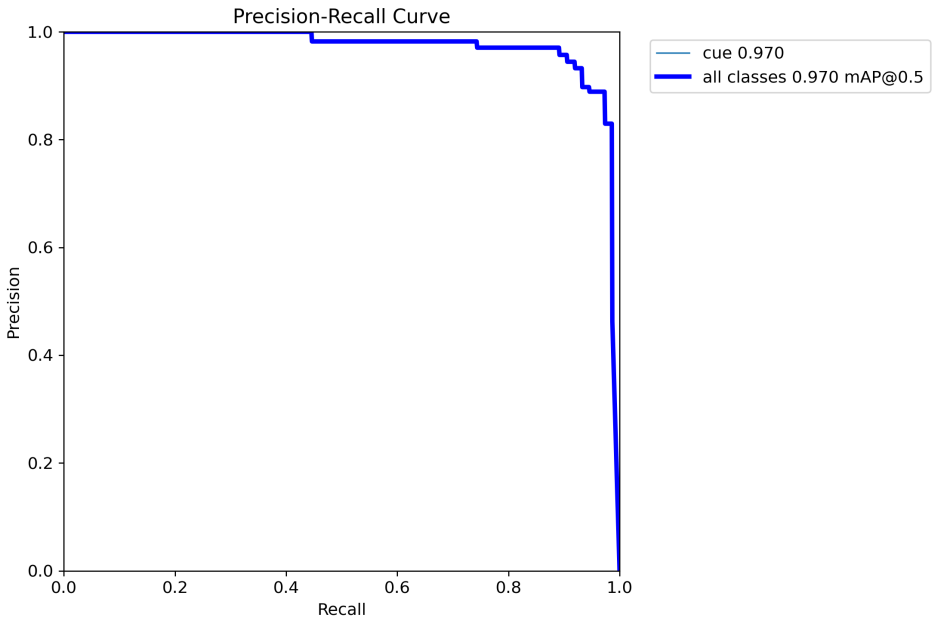


Figure 6.4: Precision-Recall Curve for Pose Estimation showing a high mAP@0.5 of 0.970.

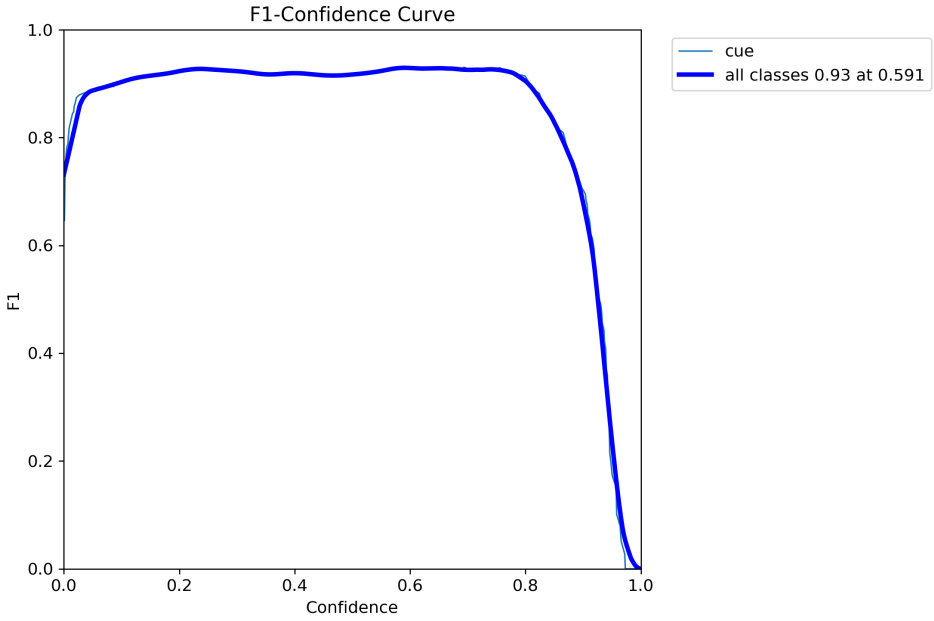


Figure 6.5: F1-Confidence Curve for the cue stick keypoints, peaking at 0.93.

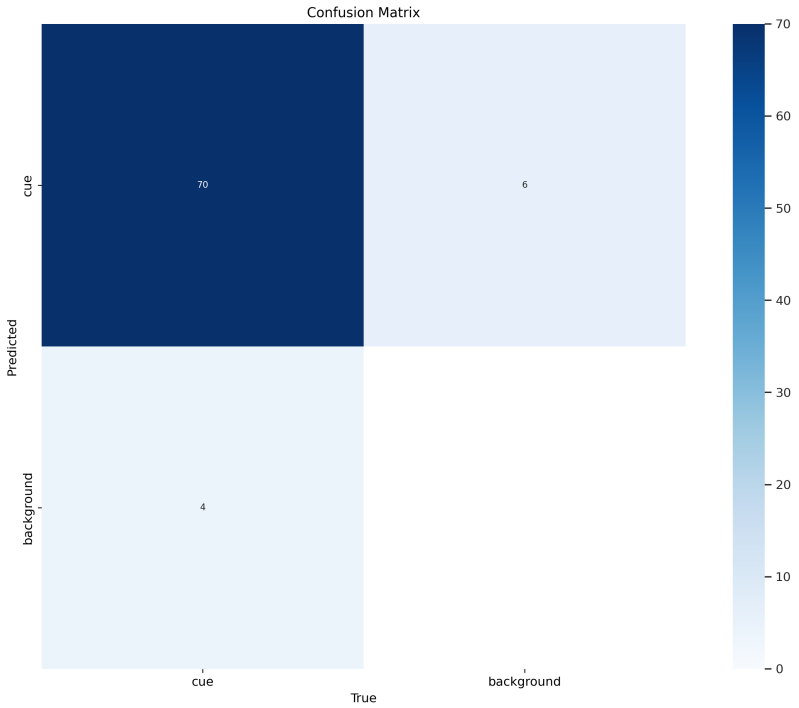


Figure 6.6: Confusion Matrix for Cue Detection, showing 70 correct detections against 6 false negatives (background).

6.5 Experimental Results

To validate the real-word applicability of the system, an experiment was conducted to verify the consistency and distinctiveness of the impact force estimation algorithm. The test was performed by a female subject with a body weight of 54 kg. Based on the anthropometric configuration described in Chapter 5, the system calculated the effective arm mass as:

$$m_{arm} = 54 \text{ kg} \times 4.97\% \approx 2.68 \text{ kg}$$

6.5.1 Force Estimation Consistency

The test procedure involved performing series of shots categorized by subjective intensity. Initially, three categories were planned: Soft, Medium, and Hard.

However, during the experiment, it was observed that shots categorized as ”Soft” (gentle taps for exact positioning) consistently generated acceleration values below the system’s triggering threshold ($20m/s^2$). Consequently, these shots were filtered out by the noise reduction algorithm. This thresholding method corresponds with kinematic analysis methodologies found in clinical literature, where velocity or acceleration thresholds are fundamental for defining discrete movement segments and assessing smoothness [14].This confirms that the threshold effectively separates between active gameplay strikes and minor refinements or accidental movements.

Therefore, the assessment was targeted on two distinct categories ($N = 20$ total):

1. **Medium shots:** Standard shots used during regular gameplay.
2. **Hard shots:** Break-type shots or power shots.

Table 6.2 presents the raw force values recorded for each attempt.

Table 6.2: Raw recorded force values (in Newtons) for the experimental session ($m_{arm} = 2.68$ kg).

Attempt #	Medium Shot (N)	Hard Shot (N)
1	63.4	109.7
2	65.7	115.4
3	74.4	107.4
4	83.1	126.6
5	54.9	114.9
6	56.5	128.0
7	76.9	124.9
8	68.9	110.2
9	63.2	92.8
10	60.4	118.1

Based on the raw data, a statistical analysis was performed to calculate the Mean Force (F_{avg}) and Standard Deviation (σ) for each category[22]. The results are summarized in Table 6.3.

Table 6.3: Statistical analysis of impact force measurements ($N = 20$).

Shot Category	Min (N)	Max (N)	Mean ± Std Dev (N)
Medium Shot	60.4	83.1	**66.74 ± 8.6**
Hard Shot	92.8	128.0	**114.8 ± 10.1**

Interpretation: The outcomes indicate a clear separation between the "Medium" and "Hard" categories. As shown in the statistical summary, the maximum force recorded for a medium shot (83.1 N) is significantly lower than the minimum force for a hard shot (92.8 N). This lack of overlap confirms that the system reliably distinguishes between different levels of player intent.

The calculated forces are proportional to the subject's arm mass (2.68 kg). For example, a hard shot of 178 N corresponds to an acceleration of approximately 66 m/s^2 ($\approx 6.7g$), which is a realistic value for a dynamic arm movement [6].

6.5.2 System Latency and Performance

A secondary test was conducted to evaluate the real-time capabilities of the system. The total latency was measured as the time difference between the sensor event timestamp

(on Android) and the visualization update on the PC.

- **Network Latency:** Average 15ms (local Wi-Fi network).
- **Processing Time:** Average 25ms (Vision Module inference per frame).
- **Frame Rate:** The vision module maintained a stable 30 FPS on the test workstation.

These efficiency measures confirm that the system acts in ”near real-time,” providing feedback instant enough for the user to relate the data to the just-performed action.

Chapter 7

Conclusions

The main goal of this thesis was to design and implement the "Smart 9-Ball Assistant," a hybrid system capable of augmenting the game of billiards with real-time visual guidance and biomechanical feedback. The resulting prototype successfully integrates Computer Vision and Telemetry data to bridge the gap between physical execution and geometric visualization.

7.1 Achievement of Objectives

The project goals, as defined in the introduction, have been met with the following outcomes:

- **Computer Vision Module:** A effective detection pipeline was developed using a custom-trained **YOLOv12** model for billiard balls and **YOLOv8-pose** for the cue stick. The validation results confirmed exceptional accuracy, with the ball detection model attaining a mean Average Precision (mAP@0.5) of **0.991**, making sure that the system can reliably track the game state under different lighting conditions.

- **Trajectory Prediction:** The "Ghost Ball" algorithm was successfully implemented. By calculating the intersection of the cue stick vector with the target ball's geometry, the system provides immediate visual feedback (Augmented Reality overlays) that helps users visualize the aim line and the tangent line.

- **Telemetry and Biomechanics:** The mobile sensor module, developed for the Android platform, reliably obtains high-frequency inertial data. By implementing the physics-based force model ($F = ma$) and utilizing anthropometric data (Plagenhoef et al.), the system can estimate the impact force in Newtons. Experimental results confirmed that the system allows for the objective differentiation of shot intensity (e.g., distinguishing "Medium" vs. "Hard" shots with clear statistical separation).

- **Real-time Performance:** The system satisfies the non-functional requirement for low latency. The vision pipeline operates at approximately 30 FPS, and the network transmission delay (approx. 15ms) is negligible for the purpose of static aiming adjustment.

7.2 Difficulties Experienced

During the development lifecycle, multiple technical challenges were encountered and addressed:

- **Network Fragmentation:** A significant issue arose with the TCP communication where high-frequency JSON packets were being coalesced or split, causing parsing errors. This was resolved by implementing a stream buffering mechanism with strict delimiter handling.
- **Android UI Performance:** The initial mobile application suffered from severe "UI Lag" due to processing sensor events on the main thread. Decoupling the data transmission thread from the UI rendering thread (via the producer-consumer pattern) was necessary to maintain stable performance.
- **Visual Jitter:** Raw detection data from the camera resulted in "shaking" projection lines, which made aiming difficult. A temporal smoothing algorithm (Moving Average Filter) was implemented to stabilize the vector calculations without introducing perceptible lag.
- **Dataset Imbalance:** Training the neural network required addressing the significant class imbalance between the single "cue ball" and the fifteen "object balls." This was mitigated via focused data augmentation during the pre-processing phase.

7.3 Future Development

While the current prototype is fully functional, several paths for additional development have been identified to enhance the system's commercial viability and user experience:

- **Perspective Correction (Homography):** Currently, the system requires a strictly top-down camera view. Future iterations could implement a homography transformation matrix to allow the camera to be placed at an angle (e.g., on a tripod at the side of the table), making the setup more flexible.

- **Wireless Sensor Integration:** Replacing the USB tethering with Bluetooth Low Energy (BLE) or Wi-Fi Direct would significantly improve player comfort, removing the physical cable constraint during the stroke.
- **Advanced Physics Engine:** The current model assumes elastic collisions. Further development might incorporate the effects of "English" (spin) and cloth friction into the trajectory prediction to simulate curve shots and bank shots more accurately.
- **Gamification:** The system could be expanded with a "Training Mode" that creates a challenge for the player to hit specific force targets or execute pre-defined shots, tracking their progress over time in a cloud-based database.

Bibliography

[1] David Alciatore. *Ghost-ball Aiming*. 2025. URL: https://drdavepoolinfo.com/resource_files/ghost_ball_aiming.pdf (visited on 14/02/2025).

[2] David Alciatore. *Sidespin and English Terminology and Uses*. 2012. URL: https://drdavepoolinfo.com/faq/sidespin/terminology-and-uses/ (visited on 14/02/2025).

[3] Ricardo Alves, Luís Sousa and Joao Rodrigues. 'PoolLiveAid: Augmented reality pool table to assist inexperienced players'. In: *21st International Conference on Computer Graphics, Visualization and Computer Vision*. 2013, pp. 184 –193.

[4] Android Developers. *Android Debug Bridge (adb)*. Official Documentation. Google. 2025. URL: https://developer.android.com/tools/adb (visited on 19/02/2025).

[5] Android Developers. *Motion sensors*. Official Documentation. Google. 2025. URL: https://developer.android.com/develop/sensors-and-location/sensors/sensors_motion (visited on 19/02/2025).

[6] Driveline Baseball. *Momentum and Arm Action: Myths and Misunderstandings*. 2014. URL: https://www.drivelinebaseball.com/2014/04/momentum-arm-action-myths-misunderstandings/ (visited on 28/04/2014).

[7] Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. New York, NY, USA: Springer, 2006. ISBN: 978-1493938438.

[8] G. Bradski. 'The OpenCV Library'. In: *Dr. Dobb's Journal of Software Tools* (2000).

[9] Cheng Chen, Jiaxin Xue, Wenling Gou, Mengning Xie and Xiaolin Yao. 'Quantitative analysis and evaluation of research on the application of computer vision in sports since the 21st century'. In: *frontiers* (2025), pp. 1 –8.

[10] Abhinav Dadhich. *Practical Computer Vision: Extract insightful information from images using TensorFlow, Keras, and OpenCV*. Packt Publishing, 2018. ISBN: 978-17-882-9476-8.

[11] Manish Duggal. 'Hawk Eye Technology'. In: *Journal of Global Research Computer Science & Technology* 1.2 (2024), pp. 1 –7.

[12] Brad Dwyer and James Gallagher. *Getting Started with Roboflow*. 2023. URL: https://blog.roboflow.com/getting-started-with-roboflow/ (visited on 16/03/2023).

[13] Bernhard Hollaus, Sebastian Stabinger, Andreas Mehrle and Christian Raschner. 'Using Wearable Sensors and a Convolutional Neural Network for Catch Detection in American Football'. In: *Sensors* 20.23 (2020), p. 6722.

[14] Netha Hussain, Katharina S Sunnerhagen and Margit Alt Murphy. 'Recovery of arm function during acute to chronic stage of stroke quantified by kinematics.' In: *Journal of Rehabilitation Medicine* 53.3 (2021), pp. 2771 –2778.

[15] Glenn Jocher, Ayush Chaurasia and Jing Qiu. *Ultralytics YOLOv8*. Version 8.0.0. 2023. URL: https://github.com/ultralytics/ultralytics.

[16] Lumitex. *Prototyping Methodology Steps on How to Use It Correctly*. 2017. URL: https://www.lumitex.com/blog/prototyping-methodology (visited on 01/06/2017).

[17] Zigurd Mednieks, Laird Dornin, G. Blake Meike and Masumi Nakamura. *Programming Android, 2nd Edition: Java Programming for the New Generation of Mobile Devices*. Sebastopol, CA, USA: O'Reilly Media, Inc., 2012. ISBN: 978-14-493-5847-1.

[18] A. D. Moore. 'Mechanics of Billiards, and Analysis of Willie Hoppe's Stroke'. In: University of Michigan, College of Engineering, 1947.

[19] Sean Ogle. *Breaking Eighty*. 2022. URL: https://breakingeighty.com/blast-motion-golf-review (visited on 13/10/2022).

[20] Stanley Plagenhoef, F. Gaynor Evans and Thomas Abdelnour. 'Anatomical Data for Analyzing Human Motion'. In: *RESEARCH QUARTERLY FOR EXERCISE AND SPORT* 54.2 (1983), pp. 169 –178.

[21] Neil Smyth. *Android Studio 3.3 Development Essentials - Android 9 Edition: Developing Android 9 Apps Using Android Studio 3.3, Java and Android Jetpack*. Payload Media, Inc., 2019. ISBN: 978-1795654760.

[22] Katarzyna Stąpor. *Wykłady z metod statystycznych dla informatyków*. Gliwice: Wydawnictwo Politechniki Śląskiej, 2008. ISBN: 978-83-7335-543-9.

[23] Yunjie Tian, Qixiang Ye and David Doermann. *YOLO12: Attention-Centric Real-Time Object Detectors*. 2025. URL: https://github.com/sunsmarterjie/yolov12.

Appendices

Index of abbreviations and symbols

Abbreviations

ADB	Android Debug Bridge
AI	Artificial Intelligence
API	Application Programming Interface
AR	Augmented Reality
CNN	Convolutional Neural Network
CV	Computer Vision
FPS	Frames Per Second
IMU	Inertial Measurement Unit
IoU	Intersection over Union
JSON	JavaScript Object Notation
mAP	mean Average Precision
TCP	Transmission Control Protocol
UI	User Interface
USB	Universal Serial Bus
YOLO	You Only Look Once

Symbols

F	impact force [N]
m_{arm}	effective mass of the arm [kg]

μ_{gender} anthropometric arm mass coefficient [%]

$\vec{a}$ linear acceleration vector

$\vec{v}_{aim}$ normalized aiming direction vector

P_{cue} center position of the cue ball

P_{ghost} calculated position of the ghost ball

L_{proj} length of the projection vector

$d_{\perp}$ perpendicular distance to the aiming line

Listings

This chapter includes the complete source code for the configuration file and the two main backend modules (Vision and Telemetry) developed for the system.

.1 Configuration File (config.json)

The user-specific configuration file utilized to define anthropometric data for the physics engine.

Listing 1: Content of config.json

```
{
    "body_weight_kg": 54,
    "gender": "female"
}
```

.2 Vision Module (bilard.py)

The main computer vision script responsible for object detection (YOLO), cue keypoint tracking, and the "Ghost Ball" trajectory calculation.

Listing 2: Vision Module implementation

```
import cv2
import supervision as sv
from inference import get_model
import numpy as np
import math
from dotenv import load_dotenv
import os

load_dotenv()

# --- CONFIGURATION ---

API_KEY = os.getenv("API_KEY")
DETECTION_MODEL = "ball-detection-bzirz/3"
KEYPOINTS_MODEL = "cue-detection-ciazj/3"

BALL_DIAMETER_PX = 45
```

```
18
19 def normalize_vector(v):
20     norm = np.linalg.norm(v)
21     if norm == 0: return v
22     return v / norm
23
24 def find_ghost_ball_position(cue_ball_pos, aim_vector, target_ball_pos):
25     vec_to_target = target_ball_pos - cue_ball_pos
26     projection_length = np.dot(vec_to_target, aim_vector)
27
28     if projection_length <= 0: return None, float('inf')
29
30     closest_point_on_line = cue_ball_pos + aim_vector * projection_length
31     perpendicular_dist = np.linalg.norm(target_ball_pos - closest_point_on_line)
32
33     if perpendicular_dist < BALL_DIAMETER_PX:
34         back_offset = math.sqrt(BALL_DIAMETER_PX**2 - perpendicular_dist**2)
35         distance_to_impact = projection_length - back_offset
36         ghost_ball_pos = cue_ball_pos + aim_vector * distance_to_impact
37         return ghost_ball_pos, distance_to_impact
38
39     return None, float('inf')
40
41 def main():
42     model_obj = get_model(model_id=DETECTION_MODEL, api_key=API_KEY)
43     model_kp = get_model(model_id=KEYPOINTS_MODEL, api_key=API_KEY)
44
45     cap = cv2.VideoCapture(0)
46
47     box_annotator = sv.BoxAnnotator(thickness=2)
48
49     print("Start!␣Press␣'q'␣to␣exit.")
50
51     while True:
52         ret, frame = cap.read()
53         if not ret: break
54
55         annotated_frame = frame.copy()
56
57         # 1. OBJECT DETECTION
58         results_obj = model_obj.infer(frame)[0]
59         detections = sv.Detections.from_inference(results_obj)
60
61         annotated_frame = box_annotator.annotate(scene=annotated_frame, detections=detections)
62
63         cue_ball_center = None
64         other_balls_centers = []
65
66         for i in range(len(detections)):
67             class_name = detections.data['class_name'][i]
68             box = detections.xyxy[i]
69             center_x = int((box[0] + box[2]) / 2)
70             center_y = int((box[1] + box[3]) / 2)
71             center_point = np.array([center_x, center_y], dtype=np.float32)
72
73             if class_name == "cue␣ball" or class_name == "cue-ball":
74                 cue_ball_center = center_point
75             elif class_name == "other": # or other name for colored balls
76                 other_balls_centers.append(center_point)
```

```
77
78          # 2. CUE KEYPOINT DETECTION
79          results_kp = model_kp.infer(frame)[0]
80          tip_pos = None
81          handle_pos = None
82
83          if hasattr(results_kp, "predictions"):
84              for pred in results_kp.predictions:
85                  if hasattr(pred, "keypoints"):
86                      for kp in pred.keypoints:
87                          if kp.confidence > 0.4:
88                              pos = np.array([kp.x, kp.y], dtype=np.float32)
89                              if kp.class_name == "tip":
90                                  tip_pos = pos
91                                  cv2.circle(annotated_frame, (int(pos[0]), int(pos[1])), 5, (0,
                                        0, 255), -1)
92                              elif kp.class_name == "handle" or kp.class_name == "grip":
93                                  handle_pos = pos
94                                  cv2.circle(annotated_frame, (int(pos[0]), int(pos[1])), 5,
                                        (255, 0, 0), -1)
95
96          # 3. DRAWING LINES AND PREDICTION
97          if cue_ball_center is not None and tip_pos is not None and handle_pos is not None:
98
99              # --- DRAWING CUE LINE (Handle -> Tip) ---
100             handle_int = tuple(handle_pos.astype(int))
101             tip_int = tuple(tip_pos.astype(int))
102             # Thick line representing the cue
103             cv2.line(annotated_frame, handle_int, tip_int, (255, 0, 0), 4)
104
105             # Calculate aiming vector based on the cue
106             aim_vector_raw = tip_pos - handle_pos
107             aim_direction = normalize_vector(aim_vector_raw)
108
109             # Looking for collision
110             closest_ghost_ball = None
111             closest_target_ball = None
112             min_distance = float('inf')
113
114             for target_ball in other_balls_centers:
115                 ghost_pos, distance = find_ghost_ball_position(cue_ball_center, aim_direction,
                        target_ball)
116                 if ghost_pos is not None and distance < min_distance:
117                     min_distance = distance
118                     closest_ghost_ball = ghost_pos
119                     closest_target_ball = target_ball
120
121             # --- DRAWING PREDICTION LINES ---
122             cue_start_int = tuple(cue_ball_center.astype(int))
123
124             # Line connecting Tip to Cue Ball (shows "aiming")
125             cv2.line(annotated_frame, tip_int, cue_start_int, (100, 100, 100), 2, cv2.LINE_AA)
126
127             if closest_ghost_ball is not None:
128                 ghost_int = tuple(closest_ghost_ball.astype(int))
129                 target_int = tuple(closest_target_ball.astype(int))
130
131                 # 1. Line Cue Ball -> Ghost Ball (continuation of cue line)
132                 cv2.line(annotated_frame, cue_start_int, ghost_int, (255, 255, 255), 2, cv2.
```

```
                    LINE_AA)
133
134                 # Circle at ghost ball location
135                 cv2.circle(annotated_frame, ghost_int, int(BALL_DIAMETER_PX/2), (255, 255,
                        255), 1)
136
137                 # Rebound physics
138                 impact_vector = closest_target_ball - closest_ghost_ball
139                 impact_direction = normalize_vector(impact_vector)
140
141                 tangent_direction = np.array([-impact_direction[1], impact_direction[0]])
142                 if np.dot(tangent_direction, aim_direction) < 0:
143                     tangent_direction = -tangent_direction
144
145                 # 2. Target Ball Path (Green)
146                 target_end_pos = closest_target_ball + impact_direction * 200
147                 cv2.arrowedLine(annotated_frame, target_int, tuple(target_end_pos.astype(int))
                        , (0, 255, 0), 4, tipLength=0.2)
148
149                 # 3. Cue Ball Path (Yellow)
150                 cue_end_pos = closest_ghost_ball + tangent_direction * 200
151                 cv2.arrowedLine(annotated_frame, ghost_int, tuple(cue_end_pos.astype(int)),
                        (0, 255, 255), 4, tipLength=0.2)
152             else:
153                 # If no collision, draw line "to infinity" from the cue ball
154                 infinity_point = cue_ball_center + aim_direction * 1000
155                 cv2.line(annotated_frame, cue_start_int, tuple(infinity_point.astype(int)),
                        (200, 200, 200), 1, cv2.LINE_AA)
156
157         cv2.imshow("Billiards␣AI␣-␣Full␣Path", annotated_frame)
158         if cv2.waitKey(1) & 0xFF == ord('q'):
159             break
160
161     cap.release()
162     cv2.destroyAllWindows()
163
164 if __name__ == "__main__":
165     main()
```

.3 Telemetry Module (sensors.py)

The backend server that handles TCP communication with the mobile device, calculates impact force ($F = ma$) using anthropometric data, and visualizes the results on a dashboard.

Listing 3: Telemetry Module implementation

```
1 import cv2
2 import numpy as np
3 import math
4 import socket
5 import json
6 import threading
7 import time
8 import csv
9 import os
```

```
from datetime import datetime
from collections import deque
import itertools

# --- CONFIGURATION ---
PORT = 5555
HISTORY_SIZE = 200
CONFIG_FILE = "config.json"

# Files and Folders
CSV_FILE = "hit_history.csv"
IMAGE_FOLDER = "saved_graphs"

# Delays
DELAY_SNAPSHOT_1 = 0.8
DELAY_SNAPSHOT_2 = 3.0

# --- DATA FROM PLAGENHOEF ET AL. (1983) ---
# Sum of Hand + Forearm + Upper Arm percentages
# Source: Table 4, Segment Weights as Percentages of Total Body Weight
ARM_PERCENTAGE_MALE = 0.0577    # 0.65% + 1.87% + 3.25%
ARM_PERCENTAGE_FEMALE = 0.0497  # 0.5% + 1.57% + 2.9%

def load_arm_mass():
    """
    Calculates arm mass based on Plagenhoef et al. (1983) data.
    """
    default_mass = 4.0

    if not os.path.exists(CONFIG_FILE):
        print(f"Config file not found. Using default arm mass: {default_mass} kg")
        return default_mass

    try:
        with open(CONFIG_FILE, 'r') as f:
            config = json.load(f)

        weight = float(config.get("body_weight_kg", 75.0))
        gender = config.get("gender", "male").lower()

        # Calculate biological arm mass based on research percentages
        if gender == "female":
            bio_arm_mass = weight * ARM_PERCENTAGE_FEMALE
            print(f"Loaded settings for FEMALE: {weight}kg -> Arm mass: {bio_arm_mass:.2f} kg (4.97%)")
        else:
            bio_arm_mass = weight * ARM_PERCENTAGE_MALE
            print(f"Loaded settings for MALE: {weight}kg -> Arm mass: {bio_arm_mass:.2f} kg (5.77%)")

        total_mass = bio_arm_mass
        print(f"Total Effective Mass (Bio): {total_mass:.2f} kg")
        return total_mass

    except Exception as e:
        print(f"Error reading config: {e}. Using default: {default_mass} kg")
        return default_mass

# PHYSICAL CONSTANTS - CALCULATED DYNAMICALLY
```

```
67 ESTIMATED_ARM_MASS = load_arm_mass()
68
69 # Thresholds
70 THRESHOLD_HIT = 20.0  # m/s^2 (Hit detection threshold)
71 THRESHOLD_PEAK = 15.0 # m/s^2 (Threshold to recognize a peak in acceleration)
72
73 # --- GLOBAL VARIABLES ---
74 sensor_data = {
75     "acc_x": 0.0, "acc_y": 0.0, "acc_z": 0.0,
76     "gyro_x": 0.0, "gyro_y": 0.0, "gyro_z": 0.0,
77     "last_update": 0
78 }
79
80 # Session Peaks
81 peak_acc_current = 0.0    # m/s^2
82 peak_force_n_current = 0.0 # N (Newtons)
83 peak_gyro_current = 0.0   # deg/s
84
85 # History Deques
86 history_acc = deque([0]*HISTORY_SIZE, maxlen=HISTORY_SIZE)    # Acceleration
87 history_force_n = deque([0]*HISTORY_SIZE, maxlen=HISTORY_SIZE) # Force (N)
88 history_gyro = deque([0]*HISTORY_SIZE, maxlen=HISTORY_SIZE)   # Rotation
89
90 shot_trigger_time = 0
91 waiting_for_snapshot_1 = False
92 waiting_for_snapshot_2 = False
93
94 # Variables to save (Values from impact moment)
95 saved_acc = 0.0
96 saved_force_n = 0.0
97 saved_gyro = 0.0
98
99 # Function to save file, updated with Newtons
100 def save_file(val_acc, val_force_n, val_gyro, image_snapshot):
101     if not os.path.exists(IMAGE_FOLDER):
102         os.makedirs(IMAGE_FOLDER)
103
104     now = datetime.now()
105     timestamp = now.strftime("%Y%m%d_%H%M%S")
106     # Save only one type of image (MOMENT)
107     png_filename = f"{IMAGE_FOLDER}/hit_{timestamp}_MOMENT.png"
108
109     # Background for text
110     cv2.rectangle(image_snapshot, (0, 550), (800, 600), (0, 0, 0), -1)
111
112     desc_mode = "(at␣impact)"
113
114     # INFO ON IMAGE (Added Newtons)
115     info_text = f"ACC:␣{val_acc:.1f}␣m/s2␣|␣FORCE:␣{val_force_n:.0f}␣N␣|␣ROTATION:␣{val_gyro
        :.0f}␣deg/s"
116     cv2.putText(image_snapshot, info_text, (20, 580), cv2.FONT_HERSHEY_SIMPLEX, 0.6, (255,
        255, 255), 2)
117
118     cv2.putText(image_snapshot, f"HIT␣ANALYSIS␣{desc_mode}", (20, 550), cv2.
        FONT_HERSHEY_SIMPLEX, 0.7, (0, 255, 0), 2)
119
120     cv2.imwrite(png_filename, image_snapshot)
121
122     # SAVE TO CSV (Added Force_N column)
```

```
123     file_exists = os.path.isfile(CSV_FILE)
124     with open(CSV_FILE, mode='a', newline='') as file:
125         writer = csv.writer(file, delimiter=';')
126         if not file_exists:
127             writer.writerow(["Date", "Time", "Acc_ms2", "Force_N", "Rotation_deg_s", "File"])
128 
129         date_str = now.strftime("%Y-%m-%d")
130         time_str = now.strftime("%H:%M:%S")
131         writer.writerow([date_str, time_str, f"{val_acc:.2f}", f"{val_force_n:.1f}", f"{
                val_gyro:.2f}", png_filename])
132 
133     print(f"SAVED: Acc: {val_acc:.1f}, Force: {val_force_n:.0f} N, Gyro: {val_gyro:.0f}")
134 
135 def tcp_server_thread():
136     global sensor_data
137     server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
138     server_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
139     last_packet_time = time.time()
140 
141     try:
142         server_socket.bind(('0.0.0.0', PORT))
143         server_socket.listen(1)
144         print(f"Server listening on port {PORT}...")
145 
146         while True:
147             conn, addr = server_socket.accept()
148             with conn:
149                 buffer = ""
150                 last_packet_time = time.time()
151 
152                 while True:
153                     try:
154                         data_chunk = conn.recv(4096)
155                         if not data_chunk: break
156                         buffer += data_chunk.decode('utf-8')
157                         if len(buffer) > 8192: buffer = buffer[-4096:]
158 
159                         while "\n" in buffer:
160                             line, buffer = buffer.split("\n", 1)
161                             if not line.strip(): continue
162                             try:
163                                 js = json.loads(line)
164                                 ax, ay, az = float(js.get("acc_x", 0)), float(js.get("acc_y",
                                        0)), float(js.get("acc_z", 0))
165                                 gx, gy, gz = float(js.get("gyro_x", 0)), float(js.get("gyro_y"
                                        , 0)), float(js.get("gyro_z", 0))
166 
167                                 gx, gy, gz = gx * 57.2958, gy * 57.2958, gz * 57.2958
168 
169                                 now = time.time()
170                                 last_packet_time = now
171 
172                                 # 1. Total Acceleration (a)
173                                 total_acc = math.sqrt(ax**2 + ay**2 + az**2)
174 
175                                 # 2. Force (F = m*a)
176                                 force_n = total_acc * ESTIMATED_ARM_MASS
177 
178                                 # 3. Total Rotation
```

```
179                                 total_gyro = math.sqrt(gx**2 + gy**2 + gz**2)
180
181                                 sensor_data["acc_x"] = ax; sensor_data["acc_y"] = ay;
                                        sensor_data["acc_z"] = az
182                                 sensor_data["gyro_x"] = gx; sensor_data["gyro_y"] = gy;
                                        sensor_data["gyro_z"] = gz
183                                 sensor_data["last_update"] = now
184
185                                 history_acc.append(total_acc)
186                                 history_force_n.append(force_n) # Add to history
187                                 history_gyro.append(total_gyro)
188
189                             except json.JSONDecodeError: pass
190                     except Exception: break
191     except Exception as e:
192         print(f"Server error: {e}")
193
194 def draw_graph(img, data, x_start, y_start, width, height, color, scale_factor=2.0, title="",
        unit="", grid_step=10):
195     cv2.rectangle(img, (x_start, y_start), (x_start + width, y_start + height), (30, 30, 30),
            -1)
196     cv2.rectangle(img, (x_start, y_start), (x_start + width, y_start + height), (100, 100,
            100), 1)
197     base_line_y = y_start + height - 10
198
199     max_val_visible = int(height / scale_factor)
200     for val in range(0, max_val_visible, grid_step):
201         y_pos = base_line_y - int(val * scale_factor)
202         if y_pos < y_start: break
203         if val > 0:
204             cv2.line(img, (x_start, y_pos), (x_start + width, y_pos), (60, 60, 60), 1)
205         cv2.putText(img, str(val), (x_start + 5, y_pos - 2), cv2.FONT_HERSHEY_SIMPLEX, 0.35,
                (150, 150, 150), 1)
206
207     points = []
208     if len(data) > 0:
209         step = width / len(data)
210         for i, value in enumerate(data):
211             x = x_start + int(i * step)
212             y = base_line_y - int(value * scale_factor)
213             y = max(y_start, min(y, base_line_y))
214             points.append((x, y))
215
216     if len(points) > 1:
217         cv2.polylines(img, [np.array(points)], isClosed=False, color=color, thickness=2)
218
219     cv2.putText(img, title, (x_start + 5, y_start + 20), cv2.FONT_HERSHEY_SIMPLEX, 0.5, (220,
            220, 220), 1)
220     if data:
221         cv2.putText(img, f"{data[-1]:.1f} {unit}", (x_start + width - 120, y_start + 20), cv2.
                FONT_HERSHEY_SIMPLEX, 0.6, color, 2)
222
223 def main():
224     global shot_trigger_time, waiting_for_snapshot_1, waiting_for_snapshot_2
225     global saved_acc, saved_force_n, saved_gyro
226     global peak_acc_current, peak_force_n_current, peak_gyro_current
227
228     threading.Thread(target=tcp_server_thread, daemon=True).start()
229
```

```
230     cv2.namedWindow("Hit␣Analysis")
231     cv2.moveWindow("Hit␣Analysis", 400, 50)
232     print("System␣ready.")
233
234     while True:
235         window = np.zeros((600, 800, 3), dtype=np.uint8)
236         current_time = time.time()
237         is_connected = (current_time - sensor_data["last_update"]) < 1.0
238
239         if is_connected:
240             # HUD
241             cv2.rectangle(window, (10, 10), (240, 590), (20, 20, 20), -1)
242             cv2.rectangle(window, (10, 10), (240, 590), (100, 100, 100), 1)
243             cv2.circle(window, (30, 40), 8, (0, 255, 0), -1)
244             cv2.putText(window, "ONLINE", (50, 45), cv2.FONT_HERSHEY_SIMPLEX, 0.6, (0, 255, 0)
                    , 2)
245
246             if waiting_for_snapshot_1:
247                 curr_acc = history_acc[-1] if history_acc else 0
248                 curr_force_n = history_force_n[-1] if history_force_n else 0
249                 curr_gyro = history_gyro[-1] if history_gyro else 0
250
251                 if curr_acc > peak_acc_current: peak_acc_current = curr_acc
252                 if curr_force_n > peak_force_n_current: peak_force_n_current = curr_force_n
253                 if curr_gyro > peak_gyro_current: peak_gyro_current = curr_gyro
254
255             # HUD Display (Session Peak Values)
256             disp_acc = peak_acc_current if waiting_for_snapshot_1 else saved_acc
257             disp_force_n = peak_force_n_current if waiting_for_snapshot_1 else saved_force_n
258             disp_gyro = peak_gyro_current if waiting_for_snapshot_1 else saved_gyro
259
260             # 1. ACCELERATION
261             cv2.putText(window, "MAX␣ACC:", (25, 100), cv2.FONT_HERSHEY_SIMPLEX, 0.5, (200,
                    200, 200), 1)
262             cv2.putText(window, f"{disp_acc:.1f}", (25, 145), cv2.FONT_HERSHEY_SIMPLEX, 1.5,
                    (0, 0, 255), 3)
263             cv2.putText(window, "m/s^2", (160, 145), cv2.FONT_HERSHEY_SIMPLEX, 0.6, (0, 0,
                    255), 1)
264
265             # 2. FORCE (NEWTONS) - NEW
266             cv2.putText(window, "MAX␣FORCE␣(F=ma):", (25, 200), cv2.FONT_HERSHEY_SIMPLEX, 0.5,
                     (200, 200, 200), 1)
267             cv2.putText(window, f"{disp_force_n:.0f}", (25, 245), cv2.FONT_HERSHEY_SIMPLEX,
                    1.5, (0, 165, 255), 3) # Orange color
268             cv2.putText(window, "N", (160, 245), cv2.FONT_HERSHEY_SIMPLEX, 0.6, (0, 165, 255),
                     1)
269
270             # 3. ROTATION
271             cv2.putText(window, "MAX␣ROTATION:", (25, 300), cv2.FONT_HERSHEY_SIMPLEX, 0.5,
                    (200, 200, 200), 1)
272             cv2.putText(window, f"{disp_gyro:.0f}", (25, 345), cv2.FONT_HERSHEY_SIMPLEX, 1.5,
                    (255, 0, 255), 3)
273             cv2.putText(window, "deg/s", (160, 345), cv2.FONT_HERSHEY_SIMPLEX, 0.6, (255, 0,
                    255), 1)
274
275             # GRAPHS (Now 3 pieces)
276             # 1. ACCELERATION (Top, Red)
277             draw_graph(window, list(history_acc), 260, 20, 520, 160, (0, 0, 255),
278                        scale_factor=3.0, title="ACCELERATION␣(Acc)", unit="m/s2", grid_step
```

```
                    =10)
279
280             # 2. FORCE N (Middle, Orange) - Smaller scale because values are large (e.g. 40g *
                    4kg = 160N)
281             draw_graph(window, list(history_force_n), 260, 200, 520, 160, (0, 165, 255),
282                        scale_factor=0.8, title="HIT␣FORCE␣(F=ma)", unit="N", grid_step=50)
283
284             # 3. ROTATION (Bottom, Purple)
285             draw_graph(window, list(history_gyro), 260, 380, 520, 160, (255, 0, 255),
286                        scale_factor=0.3, title="ROTATION␣(Gyro)", unit="deg/s", grid_step=100)
287
288             # --- HIT DETECTION ---
289             acc_now = history_acc[-1] if history_acc else 0
290
291             if acc_now > THRESHOLD_HIT and not waiting_for_snapshot_1 and not
                    waiting_for_snapshot_2:
292                 shot_trigger_time = current_time
293                 # Reset peaks at start of new session
294                 peak_acc_current = acc_now
295                 peak_force_n_current = acc_now * ESTIMATED_ARM_MASS
296                 peak_gyro_current = 0.0
297
298                 waiting_for_snapshot_1 = True
299                 waiting_for_snapshot_2 = True
300
301             # Snapshot 1 (MOMENT OF IMPACT)
302             if waiting_for_snapshot_1 and (current_time - shot_trigger_time > DELAY_SNAPSHOT_1
                    ):
303
304                 hist_a = list(history_acc)
305                 hist_f = list(history_force_n)
306                 hist_g = list(history_gyro)
307
308                 if hist_a:
309                     # INTELLIGENT SEARCH FOR 2ND PEAK (Skipping swing)
310                     peaks_indices = []
311                     for i in range(1, len(hist_a) - 1):
312                         if hist_a[i] > THRESHOLD_PEAK:
313                             if hist_a[i] >= hist_a[i-1] and hist_a[i] >= hist_a[i+1]:
314                                 peaks_indices.append(i)
315
316                     impact_idx = 0
317                     if peaks_indices:
318                         if len(peaks_indices) > 1:
319                             candidates = peaks_indices[1:] # Discard 1st peak (swing)
320                             impact_idx = max(candidates, key=lambda i: hist_a[i]) # Highest of
                                    the rest
321                         else:
322                             impact_idx = peaks_indices[0]
323                     else:
324                         impact_idx = np.argmax(hist_a)
325
326                     # --- GET VALUES FROM IMPACT POINT ---
327                     saved_acc = hist_a[impact_idx]
328
329                     if impact_idx < len(hist_f):
330                         saved_force_n = hist_f[impact_idx]
331                     else:
332                         saved_force_n = peak_force_n_current
```

```
333
334                     if impact_idx < len(hist_g):
335                         saved_gyro = hist_g[impact_idx]
336                     else:
337                         saved_gyro = peak_gyro_current
338
339                     # Draw Line
340                     graph_width = 520
341                     x_start = 260
342                     step = graph_width / len(hist_a)
343                     impact_x = x_start + int(impact_idx * step)
344                     cv2.line(window, (impact_x, 20), (impact_x, 540), (255, 255, 255), 1)
345                     cv2.putText(window, "IMPACT", (impact_x + 5, 40), cv2.FONT_HERSHEY_SIMPLEX
                            , 0.4, (255, 255, 255), 1)
346
347                 snapshot = window.copy()
348                 # Saving only this one image
349                 threading.Thread(target=save_file, args=(saved_acc, saved_force_n, saved_gyro,
                        snapshot), daemon=True).start()
350                 waiting_for_snapshot_1 = False
351
352             # End of session (without saving second image)
353             if waiting_for_snapshot_2 and (current_time - shot_trigger_time > DELAY_SNAPSHOT_2
                    ):
354                 waiting_for_snapshot_2 = False
355
356             if waiting_for_snapshot_1:
357                 cv2.putText(window, "ANALYZING...", (30, 550), cv2.FONT_HERSHEY_SIMPLEX, 0.8,
                        (0, 255, 255), 2)
358
359         else:
360             cv2.putText(window, "WAITING...", (200, 300), cv2.FONT_HERSHEY_SIMPLEX, 0.8, (0,
                    0, 255), 2)
361
362         cv2.imshow("Hit␣Analysis", window)
363         if cv2.waitKey(20) & 0xFF == ord('q'):
364             break
365
366     cv2.destroyAllWindows()
367
368 if __name__ == "__main__":
369     main()
```

List of additional files in electronic submission

The electronic submission attached to this thesis contains a compressed archive (`.zip`) organized as follows:

- **Source Code:** The complete source code of the system, divided into:
  - `/pc_server` – Python scripts for the Vision and Telemetry modules (`bilard.py`, `sensors.py`) including the `req.txt` file.
  - `/android_app` – The Android Studio project files for the mobile sensor application.
- **Executables:**
  - `SensorApp.apk` – The compiled installation package for the Android mobile device.
- **Configuration:**
  - `config.json` – Default configuration file with calibration parameters and anthropometric data structure.
- **Documentation:**
  - `README.md` – Short instruction on how to run the environment and install dependencies.

List of Figures

List of Tables