RevMetrix Technical Report

Spring 2026

Matt Brown, Josh Byers, Charles Carroll, Zach Cox, Joseph Downey, Gabe Manero, Jakeb Nielsen, Andrew Olvera, Gavin Wentz, Hunter Wolfe

Table of Contents

Abstract

The RevMetrix project seeks to capture a bowling ball's motion as it travels down a lane. The project is based on two components developed initially by Professor Hake and Dr. Babcock. The first is the SmartDot module, developed by Professor Hake. The SmartDot module is a piece of hardware that sits inside the finger insert of a bowling ball and uses sensors to collect metrics of how the ball moves. The next component is the Ciclopes software, a motion detection software that can track and display the path of a bowling ball using an uploaded video of a bowling ball shot. The RevMetrix team is developing multiple tools: the Ball Spinner, the Mobile Bowling app that pairs with a Smart Watch app, and a new AI powered Ciclopes.

The Ball Spinner is a bowling analysis tool that recreates the movement and rotation of a bowling ball during a game. A SmartDot holder, which acts as an emulation of a bowling ball,

sits in an enclosure where it rotates on three orthogonal axes using three motors, each driving a single degree of freedom and being independently operable parts within the overall system. Users can simulate shots through an application interface by entering input parameters for the shot, where these parameters are then sent to the Ball Spinner, and the motors move the SmartDot holder, replicating the bowling ball traveling down the lane The input graphs provide the user the ability to define the motion of the motors over the length of the Ball Spinner shot. The Ball Spinner then collects this data and can store it for replay and/or analysis later.

The Mobile Application provides a tool for bowlers to use as they play, and is intended to be paired with the SmartDot and Smart Watch application. It allows the user to connect to a SmartDot, collect real bowling data, and record their bowling shot by entering pins they left standing which will store data just as a bowling game presents it.

The Smart Watch application connects to the Mobile App via Bluetooth and provides an additional interface to trigger bowling shot recording on the mobile app while capturing both pre and post shot information, such as pins knocked down. This minimizes user disruption between shots and offers an easier method for entering bowling data without needing to touch the phone, which can remain positioned to record video of each shot, all while maintaining accurate and consistent shot analytics.

The AI powered Ciclopes application will then receive video from bowling games recorded using the Mobile App, on which it will perform object detection and segmentation per frame. Segmentation refers to the process of determining which pixels in each frame belong to a detected object. This is then used to extract physical coordinates of the ball on the lane. These physical coordinates will then be rendered for the user, and used to calculate and display tabular data such as kinematics calculations.

In this paper, we will provide more context behind the RevMetrix project and any important concepts needed to understand our work, examine both the design and the implementation of each of the aspects of the RevMetrix project, and discuss any future work that needs to be completed.

Introduction

Overview

The RevMetrix project seeks to provide its users with a set of hardware and software to capture a bowling ball's internal and external metrics.

To collect real data from bowling games, we developed multiple tools, including a Mobile App, Smart Watch App, and an AI powered video processing tool. The following items allow us to collect and analyze real bowling shots and their results.

1. Mobile App: A Maui cross-platform application designed to track a wide range of bowling metrics. The core feature allows users to record a full game by selecting which pins remain standing after each shot. The app also supports additional data entry and management, including:
    a. Selecting which bowling ball was used for each shot.
    b. Viewing and managing a list of previous sessions.
    c. Viewing a list of games associated with a given session.
    d. Editing previously recorded shots.
    e. Loading past games from both the session list and the full games list.
    f. Syncing user data to the cloud.

   g. Connecting to a SmartDot Module via Bluetooth to record internal ball data.

2. Smart Watch App: A Flutter application designed for Samsung watches, working in conjunction with the mobile app over Bluetooth Low Energy, sending packets of session information between the two. The app supports session creation, game addition, frame navigation, and recording shot information in a pre and post shot format. It includes the ability to import sessions directly from the connected app. With a scaled down UI, it helps maintain a smoother rhythm during play and reduces interruptions between throws.
3. AI Ciclopes: Computer vision used to extract physical coordinates of the ball on the lane. These extracted coordinates are then processed to be rendered for the user as a visualization of the shot, as well as give them tabular data on the different metrics of the shot. These metrics include position data at important positions down the lane as well as kinematics data.
4. SmartDot Module: Device that contains an embedded accelerometer, gyroscope, magnetometer, and ambient light sensor. This device will be placed within the bowling ball, and will send data collected from the sensors back to the Ball Spinner Controller and/or mobile application, to then be sent to the application.
5. Cloud Application: Web server hosted on Digital Ocean that contains the ability to store data for the Mobile Application and Ball Spinner Application. These applications interface with the Cloud Application through API endpoints which handle the creation and retrieval of data from the Database. The Cloud

Application also handles user authentication for interacting with the Database in order to prevent unwanted traffic.

To test the metric algorithms, it is required to develop a device capable of emulating a bowling ball in three orthogonal axes to create known sets of data. Accompanying this device will be a controller module capable of driving the motors on the device and communicating with the database. This application provides users with an interface for creating simulated shots where the users can enter parameters representing shot dynamics. The controller module will create motor instructions from the application based on input parameters provided by the user. The controller will then drive the physical device and simultaneously display real-time data representing the device's real-time mechanical response on the physical device using sensor data returned from the SmartDot holder. The differences between the SmartDot data and input parameters must be accurately recorded to perform calibration for real bowling shots. Thus, a significant focus will be on ensuring the quality and reliability of this module through thorough testing and validation.

Each of the components that allow the data collection/testing system to fulfill these requirements are described below:

1. Ball Spinner Controller: The communication and control module responsible for facilitating communication between the Cloud database and the Ball Spinner device. Motor instructions are sent from the controller to the motors on the Ball Spinner, in addition to SmartDot feedback sent over bluetooth then to the Cloud.
2. Ball Spinner: The enclosed physical device that will simulate bowling ball movement based on user input from the application. The Ball Spinner will contain

motors that drive the ball, as specified by the motor instructions sent by the Ball Spinner Controller.

3. SmartDot Module: Device that contains an embedded accelerometer, gyroscope, magnetometer, and ambient light sensor. This device will be placed within the bowling ball, and will send data collected from the sensors back to the Ball Spinner Controller and/or mobile application, to then be sent to the application.

The Problem

The problem addressed by this project is the absence of a unified, user-friendly system for collecting, organizing, and analyzing bowling performance data. Users need a reliable way to capture detailed shot information, review historical sessions, and incorporate sensor-based measurements without navigating multiple platforms or recording data by hand. Our goal is to streamline data collection and provide meaningful insights that support performance improvement.

The Solution

Mobile

The Mobile Application is a cross-platform system designed to collect, organize, and analyze a wide range of bowling metrics. Users can create and manage events they are competing in, add the establishments where they bowl, and create sessions that link directly to both an event and an establishment. From each session, users can add games, which contain all associated shot and frame data. Selecting a specific game opens an interactive pin layout interface where the user records pins left for each shot. Once submitted, the application displays

the full sequence of shots and frames for immediate review. The application also includes a Bluetooth interface for optionally connecting to the SmartDot module, enabling real-time collection of accelerometer, gyroscope, magnetometer, and light sensor data. Additionally, the app supports integration with a Smart Watch companion app, allowing users to pair their watch for extended functionality. A cloud synchronization feature enables users to store and access their data across devices. The statistics page provides tools to search across all events, sessions, games, frames, and shots, giving users detailed performance insights to support skill development and improvement.

Watch

The Smart Watch Application is a lightweight data entry system designed to work alongside the mobile application through Bluetooth Low Energy. Users can import sessions from the connected mobile application or create a new session which would upload to the connected mobile application when ready. Within these sessions, the user can add or delete games, as well as navigate frames which includes entering pre-shot and post-shot details. This includes pins knocked down, ball used, lane number, ball speed, and board entries like stance, target, breakpoint and impact. Users can also trigger recording on the mobile app directly from within the shot entry pages, allowing for a hands free method when the phone is already positioned for Ciclopes video capture. The shot data is sent to the phone on submission with the use of a queue, allowing for accuracy and data loss prevention. The simplified interface allows bowlers to quickly input data and have minimal disruption between shots, while maintaining accurate shot data captured and saved in the database across a full session.

Cloud

The Cloud Application is an online web server that is hosted on a Digital Ocean Droplet that allows other facets of the project to have a central place to store and interact with data. Data is submitted and retrieved through HTTP request bodies through our API. These endpoints were created in order to give each team personalized functionality with how data is to be interacted with. The API endpoints then connect with data controllers which are also in the Cloud Application code. These data controllers contain SQL prepared statements that ensure that data is handled correctly and safely without the possibility of SQL injection or other common security vulnerabilities. These data controllers then interact with our Cloud Database in order to store and organize data. The Cloud Database is subdivided into multiple schemas for the purposes of organization . These schemas act as a smaller database inside the larger scope of the whole database, meaning each Database Controller can specify which schema it is going to use when interfacing with the database. The Cloud Application also has a system in place to automate the creation of new database tables and columns. This system automatically generates SQL script code from class files in the Cloud Application project.

Ciclopes

Ciclopes is a project that is directly integrated into the mobile application. The goal of Ciclopes is to allow bowlers to set up their phone on a tripod behind them to collect video of their shots. This video will then be processed using computer vision to extract the location of the lane and ball in the image. From this, more processing is done over each extracted position in each frame to gather actionable tabular data from the physical coordinates. This is then saved and integrated into the mobile application for the user to do further analysis. In parallel, the same video is used to produce a 3D pose estimation of the bowler. This is rendered in the application

and the 3D nature allows scrolling around the rendered pose and zooming to support user analysis workflows. These data pieces together give a complete picture of the shot dynamics and corresponding result, allowing the user to make real time decisions to improve their game shot to shot.

Ball Spinner Controller

The Ball Spinner Controller and its Ball Spinner Mechanical System counterpart serve a vital purpose in the analysis of bowling data. The purpose of this system is to offer researchers a tool that can simulate bowling shots in order to produce known, repeatable SmartDot data (Accelerometer, Gyroscope, Magnetometer, and Light). Developers can pass this data from the sensor and into signal processing algorithms in order to generate insights about how a bowling ball moves during a Ball Spinner Controller driven shot. In addition, the team can also test the accuracy and consistency of the SmartDot modules which we are using to collect bowling data. Users can describe the type of shot they wish to throw and the system will then simulate this shot. This description method is done through three input graphs. The first graph is for the main axis of rotation: the x-axis, also known as the spin axis. This axis is controlled in units of RPM. The other two graphs are for the two other axes: the rotation and tilt axes, or the y and z axes. These are controlled in units of degrees.

Ball Spinner Mechanical System

The Ball Spinner Mechanical System is the physical system that will spin a bowling ball equivalent in a way that emulates a real bowling shot. In order to do this, the SmartDot modules will need to be mounted so that their location relative to their center of rotation matches that of being inside of a real bowling ball. This will allow the SmartDot to collect and transmit real-time data to the Ball Spinner Controller while the system is in motion. Using a system similar to that

of a three-axis gimbal, all three degrees of rotation can be accommodated. Each degree of freedom will be influenced by subsequent degrees, and the SmartDot modules will be influenced by all three. The first degree will sit directly on top of the second, and these two will then be encapsulated by the third degree. The SmartDot holder is 8.5 inches in total length to mimic the diameter of a real bowling ball, and the SmartDots will be mounted 2 inches from the ends of the support as an analog for the depth of a finger hole. All three degrees will be driven independently by separate motors and will be housed in an acrylic box as a safety precaution. The SmartDot holder will be made with 3D printed Polycarbonate and the rest of the assembly will be fabricated with aluminum.

Background

Founding Technologies

The following describes the Ciclopes software and SmartDot device, the instruments that founded the RevMetrix project.

Ciclopes

Ciclopes is a software developed by Dr. Babcock. It is designed to detect and trace the path of a bowling ball that has traveled down a lane from an external view. The input is a video file that requires calibration based on the camera position but can generally be placed in any position that has visibility of the full lane. Figure 1.1.2.1 shows how the ball's movement is captured by the software from an inputted video. This software lets users easily view their bowling shots' external metrics along with statistics over multiple shots. Figure 1.1.2.2 shows an

example of how the software displays its captured data. The Ciclopes software is one of the two legacy innovations of the RevMetrix project. The limitations of this software include tedious manual camera calibration, tedious selection of the ball's position in the first few frames, a deprecated codebase developed in Microsoft Foundation Class Library, and a lack of support for modern video codecs.

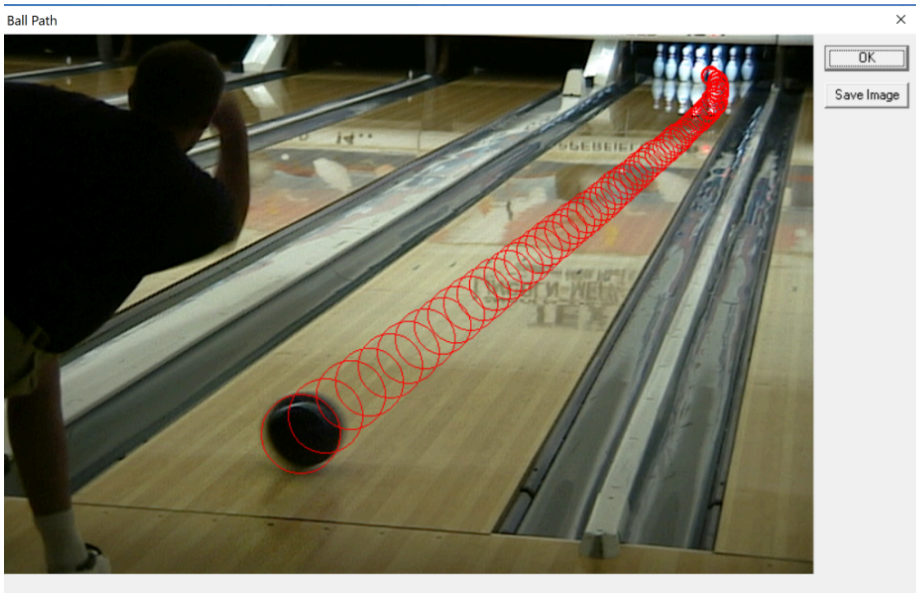


Figure 1.1.2.1: Bowling Ball being Detected by Ciclopes

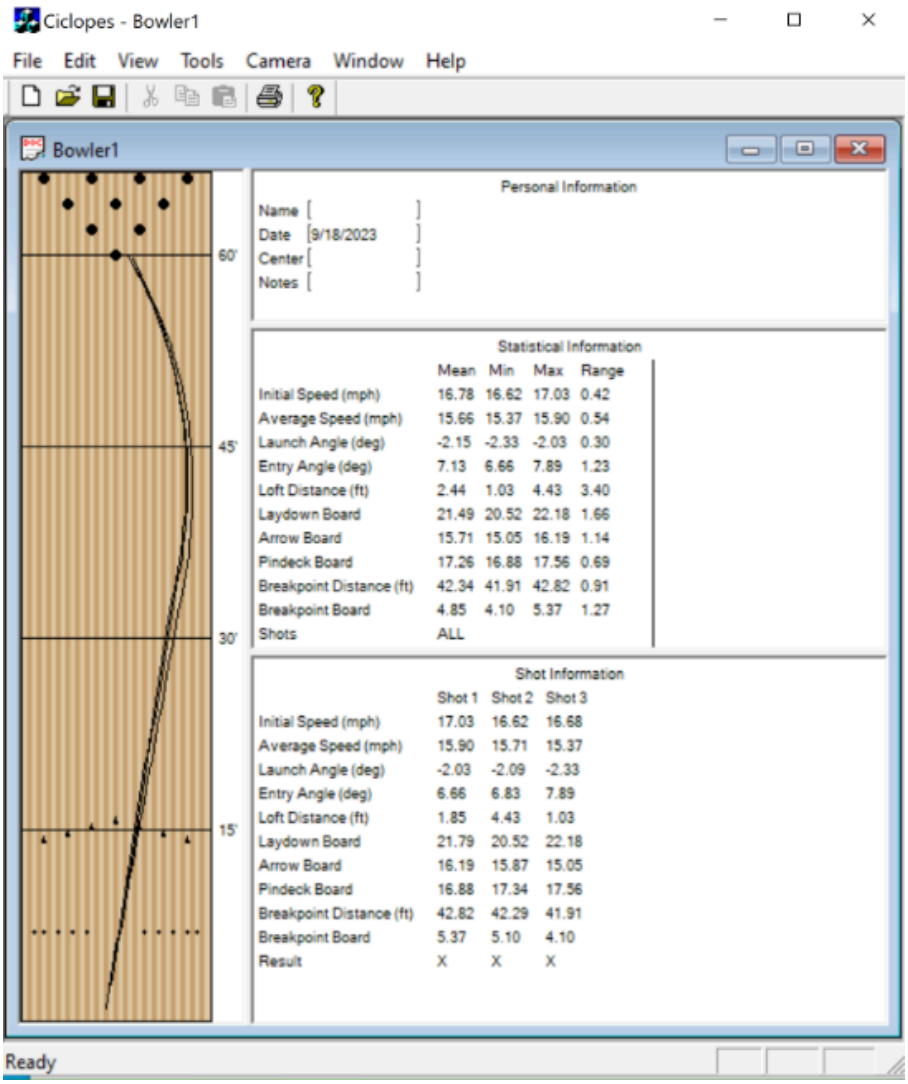


Figure 1.1.2.2: Bowling Ball Path traced by Ciclopes

SmartDot

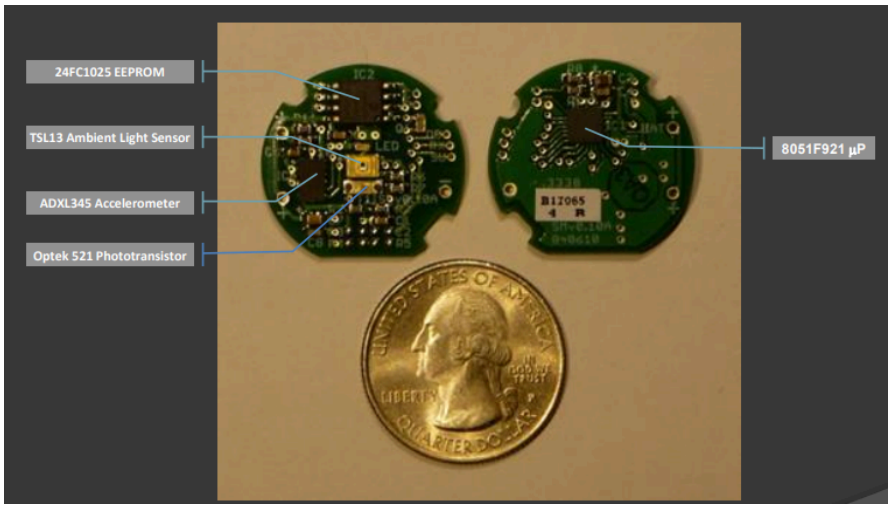


Figure 1.1.2.3 Professor Hake's SmartDot Compared to a Quarter

Designed as Professor Hake's thesis project (detailed in Professor Hake's 2014 thesis paper [1]), the SmartDot module, Figure 1.1.2.3, will be a 9DOF module with a light sensor primarily intended for bowling applications, yet it remains under development. The device will communicate over Bluetooth Low Energy (BLE). It is important to note that the current implementation of the SmartDot module does not yet have the full 9DOF implemented.

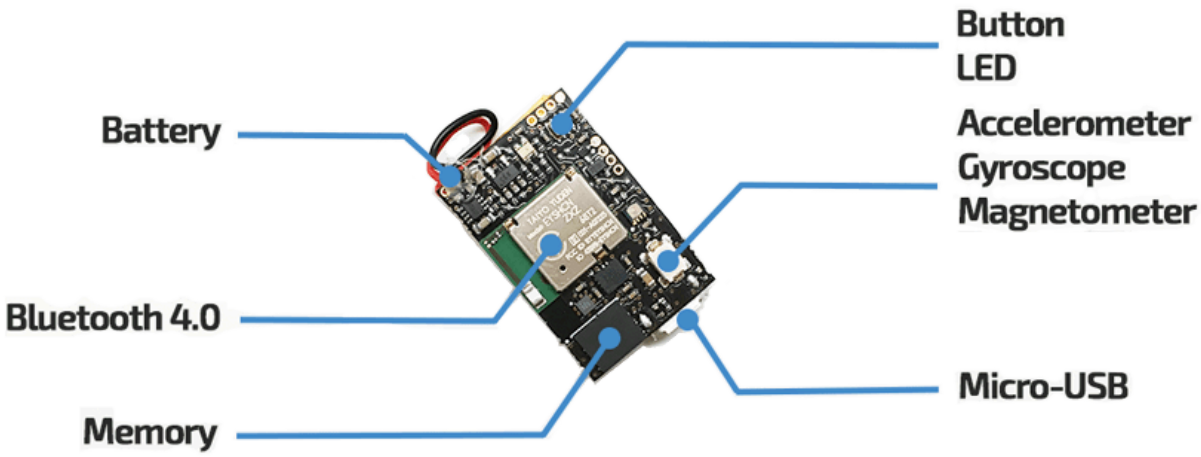


Figure 1.1.2.4 Diagram of the Components in the MetaMotionS module.

Out of the Accelerometer, Gyroscope, Magnetometer, and Light sensor, all present in a 9DOF system, the SmartDot only has the Accelerometer and Light sensor. In order to resolve this issue and collect the full 9DOF data, the RevMetrix team of developers will use the MetaMotionS module, Figure 1.1.2.4, a device that has Bluetooth and 9DOF already implemented, while being a similar size to the SmartDot. In Figure 1.1.2.5, an image of the data collected by the SmartDot module can be seen, and notice only accelerometer data is being collected. Additionally, the original SmartDot device is intended to sit below the finger insert of a bowling ball, Figure 1.1.2.6.

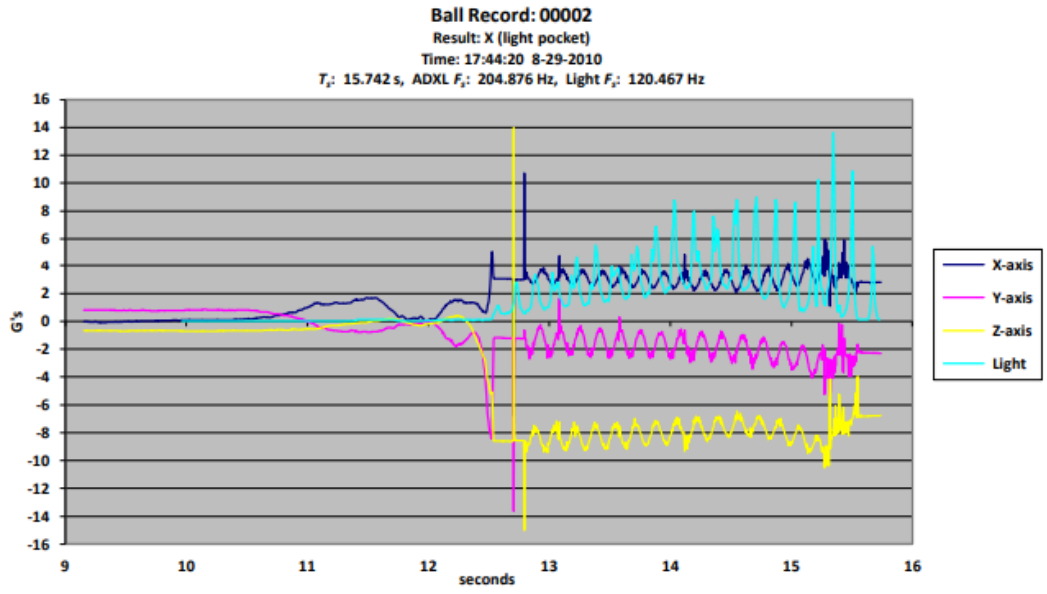


Figure 1.1.2.5 Real data collected by the SmartDot module.

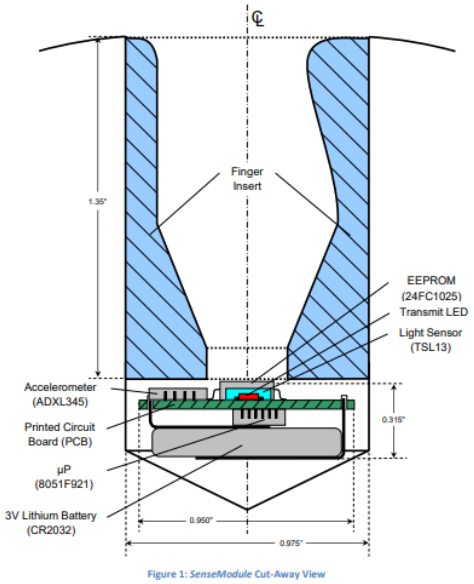


Figure 1.1.2.6 SmartDot below a Finger Insert

Technical Terms

First, Second, and Third Degree of Freedom

The 3 degrees of freedom (DOF) refer to the first, second, and third degree of freedom. As shown in Figure 1.2.1.1, the first degree of freedom refers to the spin about the x-axis, while the second and third degrees refer to the rotation and tilt about the y and z-axis respectively.

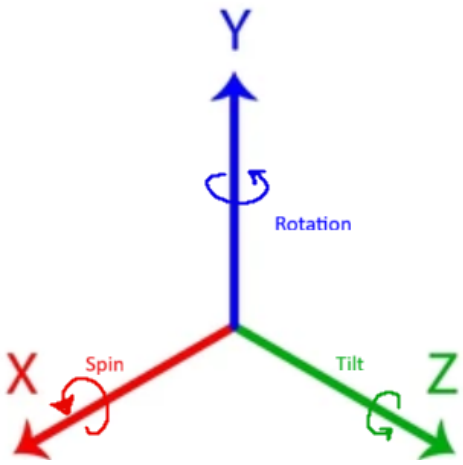


Figure 1.2.1.1: Axes of Rotation for Ball Spinner Mechanical System

Pulse Width Modulation (PWM)

PWM is a technique used to control the power delivered to electrical components. This is done by configuring the Duty Cycle of a periodic square signal sent to said components. Decreasing the duty cycle of high frequency signals decreases the average voltage of the signal. It is used to command the main motor, a brushless DC motor. This is because microcontrollers cannot output true analog voltages, while PWM provides an efficient, precise way to represent the necessary control level.

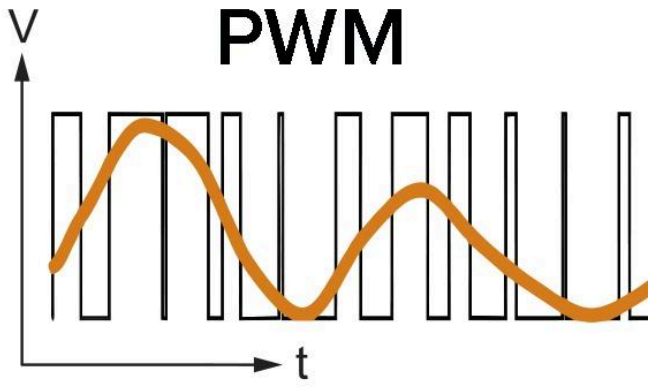


Figure 1.2.2.1: Timing Diagram of Pulse Width Modulation

9 Degrees of Freedom Modules (9DOF)

9 Degrees of Freedom Modules is a term for a module that supports an accelerometer, gyroscope, and magnetometer. The accelerometer measures linear acceleration along the X, Y, and Z axes, enabling detection of tilt, movement, and orientation relative to gravity; however, it cannot determine rotation about an axis. The gyroscope measures angular velocity and provides accurate short-term rotational data, though it is susceptible to long-term drift. The magnetometer measures the Earth's magnetic field to determine an absolute heading, but its readings are

sensitive to environmental interference. Combining all three sensors through sensor fusion produces a stable, drift-corrected, and comprehensive 3D orientation estimate by leveraging the strengths of each sensor to offset the limitations of the others. Each of these three sensors provides information along 3 axes, totaling to 9 degrees of motion.

Bluetooth Low Energy (BLE)

Bluetooth Low Energy (BLE) [2] is a subset of the Bluetooth specification designed for communication with low-power, resource-constrained devices. Introduced with Bluetooth 4.0 and refined in subsequent revisions, BLE emphasizes minimal energy consumption while maintaining sufficient data throughput for sensor-driven applications. This project uses BLE because it provides a lightweight, efficient method for streaming sensor data from the MetaMotionS without imposing significant power requirements on the device.

The primary advantage of BLE in this project is its ability to maintain a reliable wireless link while preserving battery life on the sensor module, making it suitable for continuous or long-term operation. However, BLE also introduces disadvantages, such as limited bandwidth, susceptibility to connection instability during initial pairing, and sensitivity to environmental interference, all of which can complicate real-time data acquisition.

HTTP Requests and API Server

HTTP Requests are how clients, like a web browser or an app, communicate with servers. These requests can include actions like getting data (GET), sending data (POST), updating data (PUT), or deleting data (DELETE). Each request follows a standard format, including a method, a URL, headers, and sometimes a body with additional information.

An API Server is a backend system that processes these requests. It acts as a bridge between the client and the server-side resources, like a database. The API server interprets the requests, performs the required actions, and sends a response, often in JSON format. This setup enables applications to interact with data and services efficiently over the web.

CI/CD Pipeline

A CI/CD pipeline automates the processes of continuous integration (CI) and continuous deployment/delivery (CD). It ensures that code changes are automatically built, tested, and deployed, streamlining development workflows and enabling faster and more reliable software updates. When issues do arise, the pipeline ensures to block the deployment of the codebase, allowing developers time to fix the issue before redeploying.

Homography

Homography is used in this application to perform camera calibration from pixel locations in video images to physical lane positions. These extracted coordinates are then used for data analysis.

Transfer Learning

Transfer learning is a general term meaning that a model is trained on one dataset/domain, and the learned aspects are then reused and further trained to not lose its learned capabilities, but simply apply them in a different way. This may require freezing certain layer's parameters to not overwrite them and lose generality (we are freezing the vision backbone, hopefully during transfer from synthetic to real data this keeps the understanding of real world visuals from being lost).

CNN (Convolutional Neural Network)

Convolutional neural networks extract an understanding of an image by finding spatial patterns between pixels. Early layers in the neural network find low level patterns like edges, corners, and lines. Further layers then begin to find shapes such as rectangles and circles. Then the final layers hold higher level representations such as ball and lane in purely numerical representations. This is the style of backbone used in the YOLOv11-seg models, which are further optimized for speed and parameter efficiency.

Pose Estimation

Pose estimation describes using an AI model to produce the predicted pose (the figure of the bowler) even if portions of the bowler are not in frame. We specifically are using SAM3D Body so the 3D positions are being predicted from the 2D view of the bowler.

Supervised Fine Tuning (SFT)

This is the technique to leverage transfer learning as an initial pretraining, and we do supervised learning on a much smaller dataset to teach the model to act in a different way than it has in the past, without losing any general understanding.

Inference

The forward pass of the model, means its predictions in a deterministic setting with no regularization or randomness that may have been introduced during training. This achieves the best possible performance of the model, and is used at test time / production.

Overfitting

Overfitting refers to the training loss and performance continuing to improve while the evaluation performance stays stagnant or regresses. This happens due to a lack of diversity in the train set, and is what causes the loss of general capability, and in our case the loss of its past real

world performance. It can be prevented by augmenting the training samples to make it seem as if there is more diverse data than there really is.

CUDA/cuDNN

Dependencies for using GPU acceleration on NVIDIA GPUs. Provided in the docker container, but does require either Game Ready or Studio drivers from the NVIDIA app to be installed.

Universal Asynchronous Reciever/Transmitter (UART)

UART is a hardware interface that sends and receives asynchronous serial data using start and stop bits instead of a shared clock. It uses TX (Transmit) and RX (Receive) lines and allows simple communication between devices such as the Raspberry Pi , hall sensors and motor drivers.

Duty Cycle

The Duty Cycle is the percentage of time the signal is in the “on” state during one cycle of the Pulse Modulated period as shown in the equation below.

$$Duty\ Cycle\ (\%)\ =\ (Signal\ "On"\ Time)/(Total\ Period\ Time)$$

Torque

Torque is a force which causes an object to rotate about an axis. The torque can be calculated using the equation below, where T is the torque, r is the perpendicular distance from the force, F is the applied force, α is the angular acceleration, and I is the moment of inertia.

$$T\ =\ r\ x\ F = \alpha\ x\ I$$

POCOs

POCOs (Plain Old Class Objects) often represent data models in applications, making them ideal for database mapping or transfer scenarios. Their simplicity makes them easy to use, test, and maintain, providing a straightforward way to structure data in an application. They can also be used for ORM (Object Relational Mapping) to map data from a database into an object. Within the RevMetrix project, POCOs are often used to serialize and deserialize request data between the client and server, ensuring consistent data sent/received on both ends.

Technologies Used

The following is a list of all technologies used by the RevMetrix project, with descriptions of each technology and how it applies to the project.

Raspberry Pi

The Raspberry Pi[2] is a computer used by the Ball Spinner Controller. The current implementation uses a Raspberry PI 5 as shown in Figure 1.3.1.1, but the Raspberry PI 4 is also compatible.

Figure 1.3.1.1: Photograph of the Raspberry PI 5

MetaMotionS

The MetaMotion S[3] (Figure 1.1.2.4) is an inertial measurement unit that contains the same sensors as the proposed future SmartDot module. The MetaMotion module contains a gyroscope, accelerometer, magnetometer, barometer, temperature, and ambient light sensor, and it is currently being used to emulate the SmartDot module, which has not yet been constructed.

CSVHelper

CSVHelper is a C# .NET library that simplifies reading and writing CSV files, especially when working with POCOs. It maps CSV rows to POCOs or lists, provided the structure aligns and can write lists of POCOs with a simple WriteRecords() command. While not limited to POCOs, CSVHelper is integral to the RevMetrix project, where it efficiently handles parsing and writing local rev files stored as CSVs.

Docker

Docker[4] is an open-source platform designed to simplify application creation, deployment, and management. It uses containerization to package applications and their dependencies into lightweight, portable units called containers. These containers ensure consistency across different environments, making developing, testing, and deploying applications easier. Docker is widely used to streamline DevOps workflows and enable scalability in cloud-based environments.

Digital Ocean

Digital Ocean[5] is a cloud service provider that allows developers to easily create and manage virtual servers, called droplets, to host websites, applications, and databases. These droplets can run various operating systems and applications, making it easy to deploy custom environments. Digital Ocean also offers features like load balancing, automatic scaling, and managed databases, which enable users to scale their infrastructure as needed while maintaining high availability. Digital Ocean is commonly used to host web servers, APIs, and other production services.

Nginx Proxy Manager

Nginx Proxy Manager[6] is a web-based interface for managing Nginx reverse proxy setups. It simplifies tasks like configuring proxy hosts, setting up SSL certificates (with Let's Encrypt), and managing multiple web applications on a single server without using the command line.

.NET

.NET[7] is a C# software development framework that supports cross-platform compilation. .NET will be used as the primary platform for application and cloud development due to its cross-platform nature.

.NET MAUI

.NET MAUI[8] is a multi-platform application framework that uses C# and XAML. As mentioned, it is compatible with Mac OS, Windows, iPhone, and Android. This technology will be used to create an application for Windows, Mac OS Android, and IOS.

xUnit

xUnit is a .NET testing framework with native integration with Github, allowing automated testing after any push. This testing framework also supports creating base classes to reduce code duplication, such as creating an instance of the application every time a test runs.

ASP.NET

ASP.NET[9] is a Microsoft framework for building web applications and APIs. It allows developers to create dynamic, scalable, and secure websites and services using .NET languages like C#. ASP.NET supports server-side and client-side development and includes routing, authentication, and data access features, making it a popular choice for enterprise-level web applications.

Python

Python is a high-level programming language designed for rapid development. In this project it will be used as the primary language for applications on the Raspberry Pi. Python is highly extensible and supports many libraries that enable the implementation of the Ball Spinner device. Our project currently utilizes Python version 3.13.7.

AutoCad

AutoCAD [41] is a powerful computer-aided design tool used to create detailed circuit schematics and custom electrical components. In this project, AutoCAD enables the development of accurate wiring diagrams, symbol libraries, and custom parts that support the system's unique design requirements. Its ability to generate precise, organized circuit layouts makes it a valuable resource, ensuring clarity in documentation and reducing errors during implementation.

TINACloud

TINACloud[10] is an online application that allows users to digitally create and simulate circuits. TINACloud will be used to develop potential circuit designs and simulate the power regulation to confirm the integrity of crucial components such as the Raspberry PIs, as we are currently limited in supply.

GitHub

GitHub[12] is a cloud-based hosting website that uses Git, a version control software that enables developers to track changes and resolve conflicts. GitHub was used as the code repository for the various software components of the project, along with tracking changes to all software developed for the RevMetrix project.

Draw.io

Draw.io[13] is a free online diagramming tool. Within RevMetrix, It was used primarily for creating UML diagrams.

SolidWorks

SolidWorks[11] is a computer-aided design software used to design, model, and perform analysis on 3D objects. The primary use for SolidWorks is to design the multiple physical prototypes and their corresponding parts. It also enables the use of Finite Element Analysis (FEA) to predict where mechanical failure may occur. Lastly, SolidWorks is capable of making drawings from the 3D parts with specific dimensioning to allow for accurate fabrication of the prototype.

EF Migration Tools

The Microsoft Entity Framework Migrations tools are a publicly available API that Microsoft provides for .NET C# projects, but more specifically .NET CORE server projects. These tools allow a developer to write C# class files for each table that is to be created, add properties to each class that correspond to the columns for each table, and then have the software automatically generate an SQL script file. This script file can then be used along with our Liquibase implementation to more securely add/remove/modify the Cloud database.

Liquibase

Liquibase is a database management tool that allows developers to automate database updates and migrations. The software lies within the project, and provides command line control over updating the database. When issues arise, the software has automatic rollbacks, so there is no risk of losing live data.

Postman

Postman is a software tool that allows developers to locally test API interactions without needing to use the browser. It provides the ability to easily create a request body in multiple data

formats (JSON, plain text, etc.) as well as request headers. This software expedites the process of creating and testing new API endpoints.

PyQt6

PyQt6 is a python library that is being used to create the Graphical User Interface (GUI)/Human Machine Interface (HMI) of the Ball Spinner Controller. It is a robust python framework that handles support for threading, subprocesses, and primarily building our user interface.

PyQtGraph

PyQtGraph is a python library built on top of PyQt that is being used to graph the collected data within the Ball Spinner Controller.

PyWavelets

PyWavelets is a python library for performing wavelet decomposition on data.

PyTorch

PyTorch is a python library maintained by Meta providing an API for defining a neural network and its function. It serves as an easy to use tensor operation framework for easy training setup, which automatically interfaces with NVIDIA/AMD GPU drivers, or can be used with the CPU (not recommended).

MLops

MLops refers to the operational process used for developing, testing, and deploying machine learning systems. The operations consist of data and model labeling, saving training/evaluation metrics, and data driven engineering decisions.

IsaacSim

IsaacSim is a 3D scene creation application built by NVIDIA on top of their Omniverse 3D graphics engine. It is made for synthetic data collection, and hyper realistic environment creation for robots. We are simply using it for frame collection, and leveraging its state of the art domain randomization, photorealistic rendering capabilities, and ground truth object detection and segmentation labels. It has a well developed ecosystem for assets as it utilizes .USD scene format which is a layered format that is easy to port and develop with.

Ultralytics - YOLOv26 Model Family

This is a python package from Ultralytics, an open source computer vision model developer, which allows easy interfacing with their prebuilt tooling, open source computer vision neural network definitions, and model weight downloads and caching for rapid initial implementation. It also provides model.train()*, model.predict()*, and model.eval()* API interfaces to leverage the training, inference, and evaluation setups they used when developing the models. This further makes implementation and experimentation easy and extremely fast.

The model from Ultralytics we are using is YOLOv26n, this is the nano version (~3M params) of the 26th version of their YOLO family. YOLO stands for You Only Look Once, meaning it is a single frame based object detection / segmentation model that is hyper optimized for parameter efficiency and inference speed. This allows real time object tracking on-device, with smaller compute capabilities. We are doing segmentation specifically which requires another output head on the vision backbone alongside the detection head, which yields per class bit masks of what pixels in the frame correspond to that class.

SAM3D Body - Meta

Meta’s FAIR research lab was released alongside the main SAM3 (Segment Anything Model) SAM3D Body which uses the same backbone, but adds a module which predicts the 3D pose and resulting mesh of the detected person. This release is open source under their SAM license allowing permissive research use. The model is trained at scale and can be used in many ways, we specifically are using it to produce 3D poses of the bowler per frame of the collected video to allow further biomechanical analysis than the 2D video would allow.

Primary Motor (E3665 BLDC Motor)

The E3665 is a high speed sensored Brushless DC 2500 KV inrunner motor, depicted in Figure 1.3.24.1, used for the primary axis of the Ball Spinner system [46]. BLDC motors use a rotor with permanent magnets and coils to create a rotating magnetic field. This is the force that drives the shaft. The E3665 includes integrated hall sensors and thermal/current monitoring which enables the controller to verify RPM and operating conditions in real time [46]. This motor has a no-load speed of 25,000-30,000 RPM at an operating voltage of 12 V, and delivers a stall torque in the range of 0.4-0.6 Nm [46]. To accurately emulate a bowling ball shot, the system requires the motor to reach approximately 600 RPM in 200 ms, and the torque capability of the E3665 provides enough margin to achieve this acceleration without exceeding the motor's limits.

Figure 1.3.24.1: E3665 2500 KV Motor used for Primary Spin Axis

VESC (Flipsky Mini FESC 4.20)

The VESC, depicted in Figure 1.3.25.1, is an advanced open source motor controller used to drive the BLDC motor [48]. It provides field oriented control, current limiting, thermal protection and high frequency PWM for smooth output. The included VESC software tool has the ability to perform automatic motor detection, sensor calibrations and current tuning [48]. The controller operates over a wide input range of 8-60 V, and supports a continuous motor current of approximately 50 A [49]. The controller provides more than sufficient current capability for the torque demands of the primary spin axis. The controller also includes several safety features such as over current protection and motor stall detection, ensuring reliable and safe operations during rapid spin ups [48].

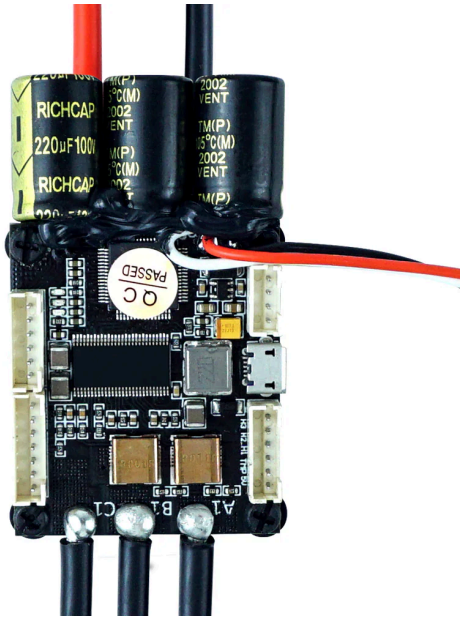


Figure 1.3.25.1: Flipsky Mini FSESC 4.20 used as BLDC Motor Controller

VESC Software

In addition to the ESC hardware, the system also utilizes a python-based VESC communication library to interface the Raspberry Pi with the Flipsky Mini FESC 4.20 [48], [49].This library enables structured bidirectional communication, allowing the controller to receive commands for duty cycle, current, or speed regulation while returning data such as electrical RPM, motor current, temperature, and input voltage [48]. These capabilities are essential for integrating the BLDC motor into the software architecture, as they allow the control logic to monitor motor behavior in real time and adjust commands accordingly. The library abstracts the VESC communication protocol into a reliable application programming interface (API), ensuring low latency and repeatable control of the primary spin axis during testing.

Nema 17 Stepper Motor

The Nema 17 stepper motor, depicted in Figure 1.3.26.1, is used for the secondary axis in the Ball Spinner system where precise angular positioning is required [45]. Stepper motors operate by energizing coil phases in sequence, producing discrete rotational steps. The Nema 17

provides a 1.8° step angle (200 steps per revolution), allowing a precise positional control when driven through microstepping [45]. The motor delivers a holding torque of approximately 0.40-0.45 Nm, which is necessary for maintaining a stable orientation of the axis under load [45]. The rated phase current of 1.5-1.7 A ensures a smooth motion and reliable torque output [45]. The motor's compact form and torque characteristics makes it suitable for repeatable angular positioning within the Ball Spinner system.

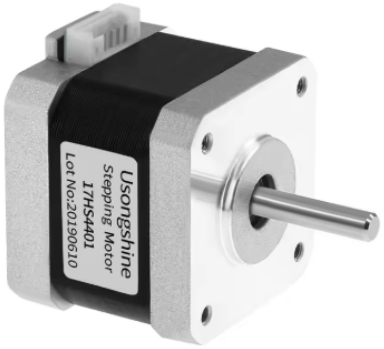


Figure 1.3.26.1: Nema 17 Stepper Motor used for Secondary Axis

DM542 Stepper Motor Driver

The DM542, depicted in Figure 1.3.27.1, is a stepper motor driver that is used to control the NEMA 17 using STEP and DIR signals from the Raspberry Pi microcontroller [47]. The driver operates from 20-50 VDC and supports an adjustable output current range of 1.0-4.2 A, set through onboard DIP switches [47]. For the Ball Spinner system, the output current is configured to match the NEMA 17's rated phase current of 1.7 A, ensuring stable torque while preventing overheating. The microstepping resolution is also DIP switch selectable, with the DM542 supporting up to 1/128 microstepping, corresponding to 25600 pulses per revolution for the NEMA 17's 1.8° step angle [47]. In the Ball Spinner system, the driver is set to 3200 pulses per revolution, which provides smooth motion.

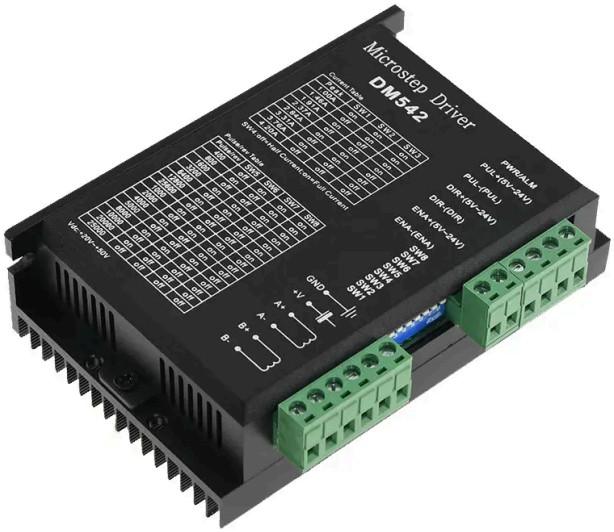


Figure 1.3.27.1: DM542 Driver used to Control the NEMA 17 Stepper Motor

Flutter

Flutter is an open source UI framework created by Google that allows developers to build cross platform applications from a single codebase. It uses the Dart programming language and a widget based system that gives developers a responsive environment with hot reload capabilities. Flutter simplifies the process for developing interfaces for different screen sizes and platforms. Flutter was chosen for its unified codebase between IOS and Android development, and its ability to create stand alone watch applications.

Dart

Dart is a client-optimized, open-source programming language developed by Google. Used by Flutter, it builds fast, reactive user interfaces across multiple platforms. Dart is fully object oriented and offers a clean syntax. With built in support for asynchronous programming, Dart can compile to both native machine code and Javascript, which is what allows for flexible use across different devices.

Kotlin

Kotlin is a statically typed programming language developed by Jetbrains. It offers concise syntax, powerful null-safety features, and seamless interoperability with existing Java codebases. Kotlin simplifies common programming tasks while improving readability. It integrates seamlessly with the Android ecosystem providing direct access to platform APIs, services and certain libraries. These things make it ideal for implementing Android specific features such as bluetooth permissions, foreground services, and background processes.

Android/IOS Launcher

The Android/iOS Launcher is a Visual Studio Code extension that acts as an interface to control and interact with pre-configured Android emulators via Android Studio and iOS simulators via Xcode on macOS. Allows for management and launch of these emulators directly from the Visual Studio development environment. This extension is being used to test our application without having to connect and test on a physical watch.

Swagger Documentation

Swagger UI Documentation is a system that allows for the generation of API usage documentation on runtime of a backend application. This documentation is useful for developers and users to have access to all of the information regarding how requests need to be sent and what sort of information will be returned to the client in a response.

ADB Logcat

Logcat is a command line tool that dumps a log of system messages into the terminal, including ones that you have written from an application with the Log class. This log was used to see certain messages being sent over BLE from the watch to the phone, and was used to track the validity of the messages being sent.

Design

System Overview

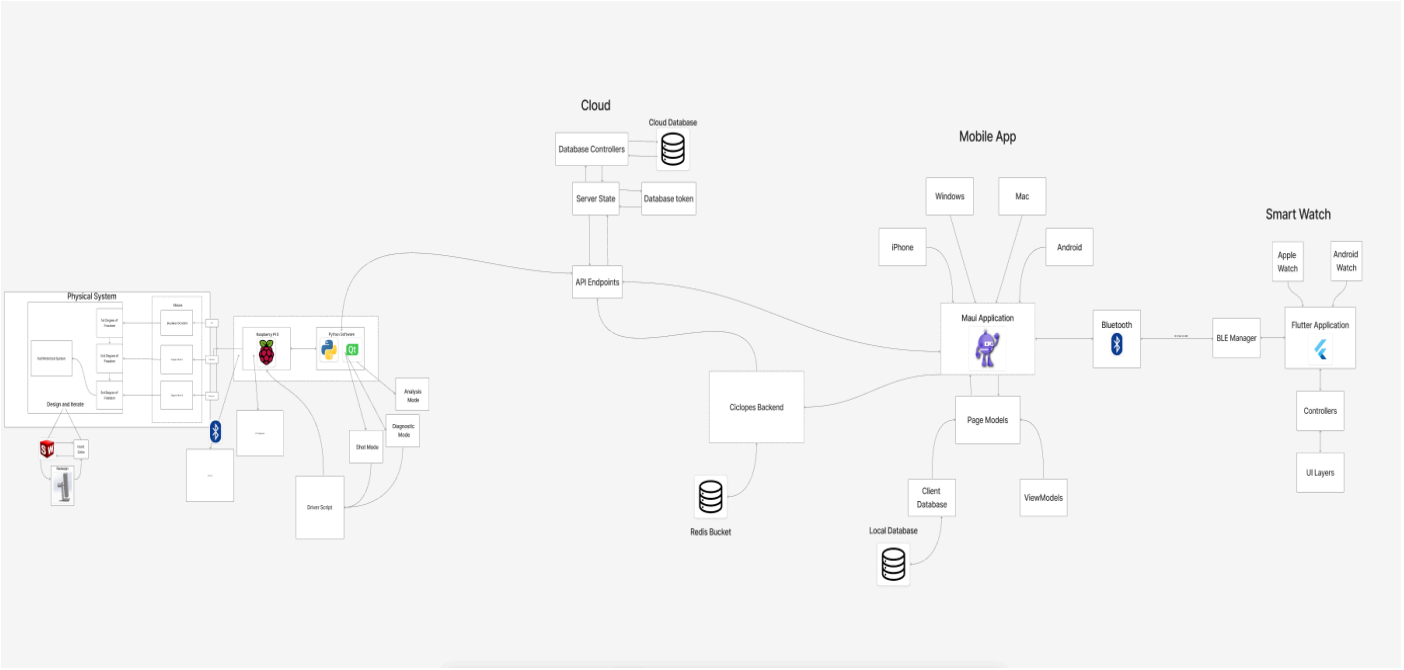


Figure 2.1.1: Screenshot of the High-Level Overview

The system centralizes around the Cloud Database, which stores all data collected from the Mobile App, Smart Watch, and Ball Spinner Controller (BSC). The BSC's purpose is to simulate a bowling shot while collecting data from the SmartDot, then the user can analyze with specific graphs, that in the future, will include but aren't limited to derivatives, Fast Fourier Transform (FFT), and Discrete Fourier Transform (DFT). The BSC directly controls the motors that drive the physical system which houses the SmartDot module used to collect data. A user can interact with the BSC software through an HMI, providing them with the ability to connect

to a Bluetooth SmartDot and create shots with the input graphs. These can then be submitted to drive the physical system with the motor instructions that were described in the graphs. The system also provides a diagnostic mode to individually control each of the motors. The system will then collect SmartDot data, motor encoder data, heat data (unimplemented), as well as storing the motor instruction set as a script. When a user is finished with a session, they can upload this data to the cloud and pull it down later for analysis.

The mobile application serves as the central platform that integrates all system components, including the Smart Watch, cloud services, SmartDot sensor, and the Ciclopes video analysis system. Its primary purpose is to enable users to input, manage, and analyze their bowling data while on the go. The application provides structured interfaces for recording locations, bowling balls, and events, and includes an interactive pin-selection interface that allows users to record each shot by selecting which pins remain standing. All collected data is stored locally on the device, enabling users to review previous games, examine performance trends, and view percentage-based statistics without requiring an internet connection. The application also includes a camera mode that interfaces with the Ciclopes system. When activated, the user positions the phone on a tripod and records their delivery and lane play while bowling. The captured video is uploaded to the cloud-based Ciclopes service, where it is processed to generate real-time feedback on the bowler's form and mechanics. This processed data is stored in the cloud and synchronized back to the mobile device for ongoing review. The Smart Watch application is designed to work alongside this mode, allowing the user to input pin leaves directly from the watch while the phone is recording. Finally, the mobile application includes a full data-synchronization feature that allows users to upload and back up their locally stored data to the cloud. Users can also pull data from the cloud in cases where local data

becomes incorrect or out of sync. This ensures that all statistics, sessions, and sensor information remain consistent across devices and accessible whenever needed.

Mobile

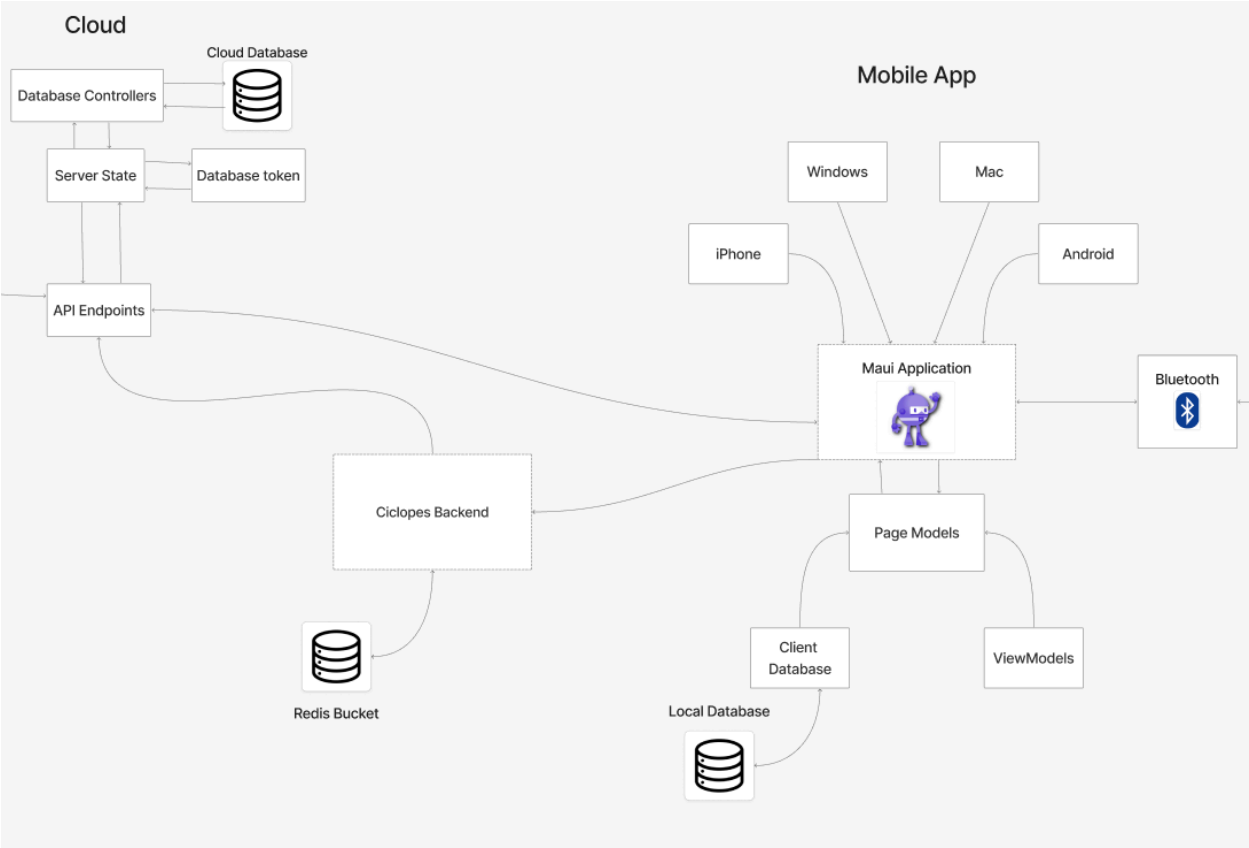


Figure 2.2.1: Screenshot of the high-level overview for the Mobile application

Overview

The Mobile Application is a cross-platform system built using .NET MAUI, enabling a single C# codebase to be compiled for Windows, macOS, Android, and iOS. This supports a unified development workflow while delivering native applications on all major user platforms. As illustrated in Figure 2.2.1 and modeled in the system's Component UML Diagram, the application consists of a MAUI-based frontend and a local persistence layer implemented using

SQLite. All user-generated data is stored locally by default, with an optional synchronization workflow that sends data to the cloud through API endpoints hosted on the DigitalOcean API Server. The Deployment UML Diagram further shows two external device interfaces: a MetaMotion S sensor module and a Smart Watch, both of which connect to the mobile application through its Bluetooth communication component.

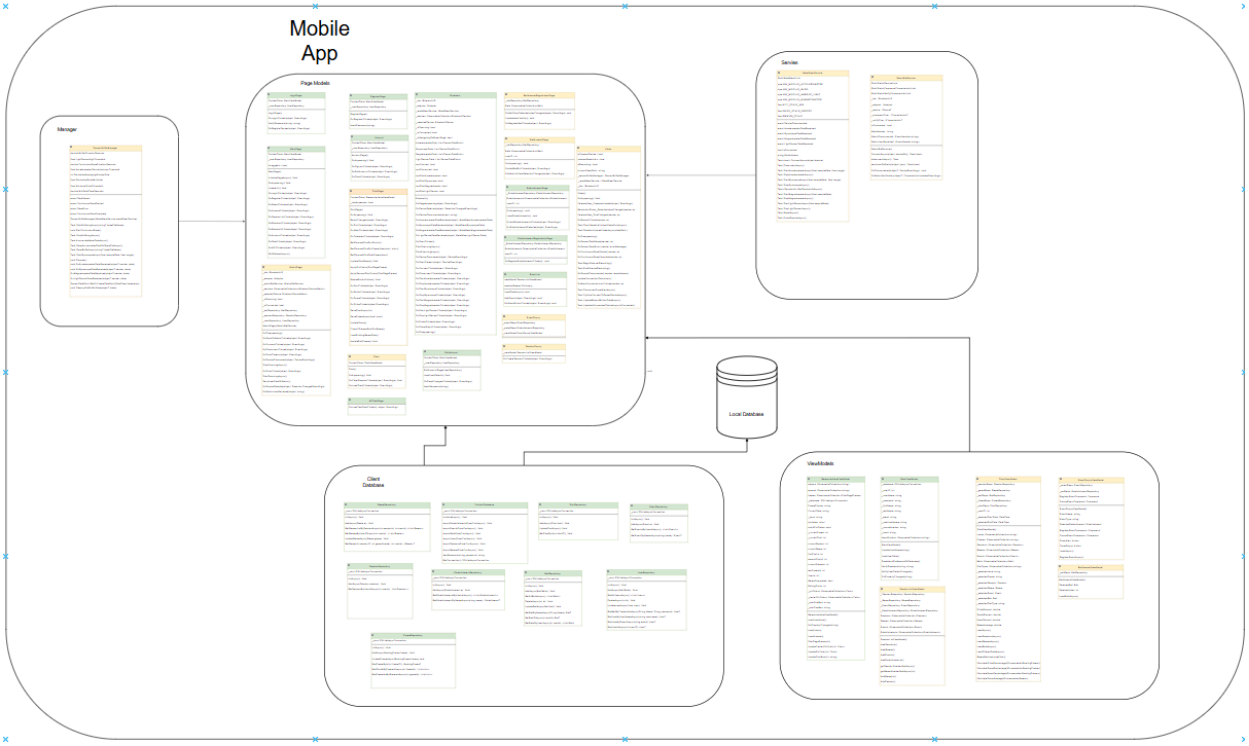


Figure 2.2.2.1 Mobile Application Backend UML diagram

Main Page

The main page is the first screen users see for the Mobile Application. It contains buttons to navigate to every other page in the application. Before the user is logged in, they can only see three buttons. The “Login” button sends users to the login page where they can enter their credentials. The “Registration” button sends users to the registration page where they can fill in

fields to create a new account. Finally, the “Guest” button bypasses the login page and signs the user into a guest account where they can currently access every page of the application. Once the user is logged in, the page updates to show additional buttons for navigation.

Login

The login page allows users to input their username and password to sign in to the Mobile Application. The password field contains a checkbox to mask and unmask the password. New users can click the blue text below the login button to navigate to the registration page to create a new account. When the User is creating or logging in, it will ask the Cloud to create an Authentication Token. This is used to authenticate the user when making cloud requests.

Registration

The registration page allows users to fill in a number of fields to create a new account. These fields are username, password, password confirmation, first name, last name, email, email confirmation, and phone number. Passwords can be unmasked using the checkbox to the right of the input field. Once all of these fields have been filled in, the user can click register to create the account and send them back to the home page to login.

Ball Arsenal

The Ball Arsenal page, provides users with a comprehensive overview of their available bowling balls. This interface displays key attributes for each ball, including the name, manufacturer, ball name from manufacturer, serial number, weight, coverstock, core, color, and comments. To register a new ball, users can select the "+" button located in the upper-right corner of the page, which navigates to the input form. Within this form, users must enter the following required information: name, manufacturer, manufacturer name, core type, and color.

Upon selecting the “Register Ball” button, the system stores the entry and updates the Ball Arsenal display to include the newly registered ball. Each ball has an “Edit” button in the bottom right. Clicking this brings the user back to the ball registration page where they can edit or disable a ball. If a ball is disabled, the name will appear red on the page. That ball will also become invisible to any other part of the app unless it is re-enabled.

Event List

The Event list page displays all of the users registered events. In the top right corner, users can tap the ‘Add Event’ button to bring up the event registration popup. This popup contains fields for name, nickname, type(practice, league, tournament, anonymous), location (linked to an establishment), day of the week played, start time, and number of games per session. Users can either cancel or click ‘Add Event’ to create the new event and go back to the main event list page. Each event item shows its name, nickname, type, location, day of the week played, and a start time. A user can click on any one of these event items to navigate to that events session list page. Finally, the orange pencil button next to each event triggers the registration popup to edit event information.

Session List

The Session list page displays all sessions and corresponding games associated with a user. In the top-right corner, users can tap the 'Add Session' button to create a new session, which brings up a popup to enter a session's date and time. Users can either click out of the popup to cancel or click ‘Create Session’ to create the new session. Sessions are displayed on the page as a combination of the session name, date, and week number. Selecting a session reveals its associated games. An ‘Add Game’ button will appear under the session if the maximum number

of games has not been reached yet. This number is determined by the event the session is associated with. Clicking on a specific game navigates to the Shot page, where individual frame and shot data can be entered. Additionally, a 'Test Interface' button is available to bypass session and game selection, taking the user directly to the Shot page for quick testing or development purposes.

Shot Page

The shot page is one of the core pages of the mobile application. This page allows users to input and save shot data. The page features a session date at the top of the page. Below that is a scrollable collection of frames. Each frame contains a frame number, boxes for the shot 1 and 2 pin count, a visual representation of the pinstates, and the game score. The current frame is highlighted in red. Users can click on a frame to edit it, and click on one of the two boxes at the top of the frame to edit a specific shot. On the left side of the screen below the frame collection there is text displaying the game number, frame number, and shot number. The center of the screen features 10 buttons representing the 10 pins as oriented the same way they would be seen in a bowling alley. These buttons are used to select which pins were left standing on each shot. To the right of the pin buttons are 4 shortcut buttons for foul, gutter, spare, and strike. Finally, at the bottom of the interface there are 3 buttons. The ‘Shot Info’ button opens up a popup where additional shot info can be selected. This info includes player (not currently used), strike ball, spare ball, stance, ball speed, position, and lane number. The “Comment” button displays a popup that allows users to input a comment for their current shot. If editing a shot, the comment can be edited and resaved. The “Next” button is used to submit the player's shot information and progress through the game.

The shot page does have some functionality restrictions based on bowling rules.

Shot 1 Restrictions

- Spare button becomes unresponsive

Shot 2 Restrictions

- Strike button becomes unresponsive
- Pin buttons that were down on shot 1 become unresponsive on shot 2

SmartDot Page

The SmartDot Page, as seen in Figure 3.1.8.1 is designed to provide a clear and streamlined workflow for discovering, selecting, and connecting to a Meta Motion S (MMS) or a Meta Motion C (MMC) device. The layout centers around simplicity so users can initiate sensor data collection with minimal steps while still supporting the more advanced capabilities of the MMS hardware. The page consists of three main design components: device discovery, device connection, and sensor control. The first component, device discovery, is initiated when the user presses the Scan button. The design intentionally uses a single-button interaction to reduce cognitive load and make the scanning process intuitive. Once scanning begins, the interface displays only MetaWear devices found in range. Each device is represented by its MAC address in a clean list format. This approach removes unnecessary device details while still giving the user enough information to identify the correct MMS module. The list updates dynamically as new devices are discovered, reinforcing the responsive nature of the Bluetooth scanning experience. The second design component is device connection. After selecting a MAC address, the user presses the Connect button, which moves the page into a “connected” state. The UI transitions visually by enabling the sensor control buttons only after a successful connection, as seen in Figure 3.1.7.2. This design choice prevents user confusion by ensuring that sensor interactions are not available until a stable BLE link is established. The page also retrieves and

stores device information upon connection, but this information remains in the background to avoid overwhelming the user with engineering-level details unless needed elsewhere in the app. The final component, sensor control, is organized around four sensor modules: accelerometer, gyroscope, magnetometer, and light sensor. Each module is represented as its own button group, making the page visually segmented and easy to navigate. When the MMS is connected, these modules become active, reflecting the system's readiness to stream data. Pressing a start button triggers the MMS to begin sending sensor data, while pressing stop suspends streaming. The interface is designed so that each sensor interaction is independent, allowing users to selectively enable only the sensors relevant to their activity. This modular design mirrors the MMS hardware layout, where each sensor corresponds to a distinct onboard module. Overall, the design of the SmartDot Page emphasizes clarity, accessibility, and real-time responsiveness. It abstracts away the complexity of BLE communication and bit-level hardware commands, presenting users with a simple, structured interface for connecting to the MMS and accessing its sensor capabilities. This design supports both novice users, who only need to connect and read data, and advanced users who rely on consistent sensor controls for data analysis, testing, or research workflows.

MetaWearBLE Service

The MetaWearBleService is responsible for managing all Bluetooth Low Energy (BLE) communication between the mobile application and MetaWear devices, specifically the MetaMotion S (MMS) and MetaMotion C (MMC). It provides a platform-agnostic interface that enables the application to connect to these devices, retrieve hardware information, control individual sensors, and stream real-time data. The service follows an event-driven architecture, exposing events for accelerometer, gyroscope, magnetometer, and light sensor data, as well as a

disconnection event that notifies the application when the device link is lost. Each event returns strongly typed data structures containing sensor values and timestamps, allowing other components of the application to process, store, or visualize the incoming data efficiently.

Communication with MMS and MMC devices is performed through encoded byte commands that correspond to specific sensor modules defined by the MetaWear protocol. Each onboard sensor is associated with a module ID, which may vary between device models. For both the MMS and MMC, the accelerometer uses module ID 0x03, the gyroscope uses 0x13, the light sensor uses 0x14, and the magnetometer uses 0x15. While the module IDs are consistent across these devices, the data packet formats they transmit differ slightly between the MMS and MMC. To begin streaming data from any sensor, commands must be issued in a specific sequence. First, a configuration command is sent to define parameters such as sampling frequency, data range, and reporting behavior. Next, the data producer is enabled, which prepares the sensor to generate data. This is followed by a subscription step, which establishes the data route from the device to the application. Finally, the sensor module is powered on, initiating data transmission. To stop data collection, this sequence is executed in reverse order. When the application starts a sensor (e.g., via StartAccelerometerAsync() or StartGyroscopeAsync()), the service constructs command packets containing the appropriate module ID, register ID, and configuration bits required for that sensor. These packets are written to the MMS/MMC using BLE characteristics, instructing the device to begin streaming data at the specified rate and configuration. The same low-level communication pattern is used to stop sensors, modify configurations, reset the device, or query available modules. The service also provides higher-level lifecycle management functions, including ConnectAsync(), DisconnectAsync(), ResetAsync(), and ProbeDeviceAsync(), which abstract away the complexity of BLE

communication. Once connected, the application can retrieve device metadata, such as model type, firmware version, and hardware revision, using GetDeviceInfoAsync().

Ciclopes

The Ciclopes page provides users with the ability to record video of a bowling lane to use with the Ciclopes software. This page is currently only compatible with Android devices. At the top right of the screen is an either red or green dot. Users can click on this to connect to a SmartDot module or view the current connection. A green dot means the phone is connected to a SmartDot. A red dot means the user is not connected to a SmartDot. Below the RevMetrix logo, users can select a resolution they wish to record in and press the record button at the bottom of the screen to start recording. The record button then becomes a stop button which is used to stop recording. After this, users can select where they want to save the video to on their phone. The default name is a date/time, but this can be changed by the user. The ‘Demo’ button below the resolution picker allows users to enter demo mode. This populates the camera view with a sample bowling video. Playing this video mimics real recording.

Establishment Page

The Establishment page provides users with a comprehensive overview of their available Establishments. This interface displays key attributes for each establishment including the name, nickname, number of lanes, address, and phone number. To register a new establishment, users can select the "+" button located in the upper-right corner of the page, which opens the popup shown in Figure 3.1.10.2. Within this form, users are prompted to enter the establishments identifying information: name, nickname, address, phone number, number of lanes, type, and the ability to specify if this is the user's home house. Upon selecting the “Register Establishment”

button, the system stores the entry and updates the Establishment display to include the newly registered establishment. Users also have the ability to edit establishment information by clicking on an establishment from the list. An ‘Edit Establishment’ popup is opened, allowing users to edit any of the information or disable an establishment. If an establishment is disabled, the name shows up in red on the main page.

API Test Page

The API Test Page functions as a method to sync the user’s local data from the mobile to the cloud database. This page is uniquely connected to a cloud-hosted Microsoft SQL database deployed on DigitalOcean. The page provides the ability for the user to select a request type and a data type, as well as provide data for the request. This functionality is for testing, and when the request arrives at the cloud server, the corresponding action is carried out from the request. When the user activates the designated ‘sync data’ button on the interface, an HTTP request with the corresponding type and data is sent to the cloud. This request will grab all of the relevant information from the local mobile database and send it to the cloud server. The cloud server then does a comparison between what data is contained within itself, and what data is contained in the request body and will add any necessary data to its own database.

Account Page

The account page displays the account information for the currently signed in user. Below this information is a drop down for “Hand”. This allows the user to specify whether they are left or right handed. This information is saved to the database and used on the shot page for the board slider as seen in Figure 3.1.12.1. The “Get Stats” button is not yet implemented. The “Edit Account Info” button sends users to the edit account page to change their information. Finally

the “Sign Out” button signs the user out of the app and sends them back to the main page seen in Figure 3.1.12.1.

The edit account page can only be accessed through the account page. This page displays the user’s account information in editable entry fields. The user can change the information in each field and then click “Save Changes” at the bottom of the screen. If the changes are valid, they will see them updated on the account page. For changing an email address or password, the user must enter the new one twice to prevent a mistake.

The “Sync with Cloud” button, shown in Figure 3.1.12.1, queries the cloud for any data associated with the user’s UserCloudID. If discrepancies are detected between the local and cloud-stored data, a popup prompt appears asking the user to choose between the two versions. If the user selects the local option, all existing cloud data is overwritten with the data currently stored on the device. Conversely, if the user selects the cloud option, all local data is deleted and replaced with the data retrieved from the cloud.

Watch Page

The watch page is used for discovering and communicating with RevMetrix enabled smart watches over Bluetooth Low Energy. A RevMetrix enabled smart watch must be running the RevMetrix software and advertising for connection. The page includes a scan button that triggers the BLE adapter to start a scan for peripherals broadcasting the RevMetrix service UUID. A list of available watches are shown and the user can then select one, thereafter connect and disconnect buttons become available. After connection, a “Set as default watch” pop-up appears, allowing the user to set the newly connected watch as their default watch. This connects the user's account information to a certain watch mac address. The user will now be able to skip the scanning process and connect directly to their saved watch.

WatchBLE Service

The WatchBleService is responsible for managing all Bluetooth Low Energy communication between the mobile application and the smartwatch device. It provides a platform-agnostic interface that allows the app to connect to the watch, send user data (sessions, games, balls, and player information), and receive shot packets captured during bowling games. The service operates using an event-driven architecture, exposing events for shot data received (WatchShotReceived), device disconnection, and control commands from the watch such as start/stop recording requests and synchronization commands. Communication with the watch is performed using a hybrid protocol. Binary packets encode shot data with header information like packet type, version, length followed by payload containing session ID, game/frame/shot numbers, pin states, stance, speed, and lane information, while JSON packets transmit control commands like startRec, stopRec, sync, and nextSession from the watch to the phone. Shot packets follow a fixed binary structure where a 16-byte payload encodes game/frame/shot identifiers, pin state and foul information, stance/target/break point/impact measurements, ball speed in mph, and lane number. When shot packets arrive, the service automatically parses them, resolves anonymous session IDs to unique database session IDs with collision detection and mapping caching, and persists the complete shot record to the database by creating or updating games, frames, and shot records as needed. The service maintains a sync context containing repository references (Game, Frame, Shot, Session, Ball, Event, Establishment) and user information that allows it to respond intelligently to watch requests, such as providing the next incomplete session when the watch completes one or rebuilding user data packets with current game progress. Through functions like ConnectAsync(), DisconnectAsync(), SendJsonToWatch(), and event handlers for incoming notifications, the service provides full

bidirectional control over the watch lifecycle and data flow, while internally managing database operations, anonymous session mapping, and session completion logic.

Stats Page

The stats page allows users to create custom queries to pull out different sets of data from the database. This can be done by selecting any combination of the following: “start date”, “end date”, “session”, “event”, “game”, “ball”, “lanes”, “frame”, “house”, and “stat type”. Each of these dropdowns are binded to the current user’s database. The “Load Stats” button submits the current combination of the dropdowns to query the database. A popup will appear showing what query data was selected and one of the 3 different types of stats: game, first ball, and second ball. If a stat type is not selected, an error message will appear telling the user they must select a stat type. Game stats displays the number of games returned from the query, the game score average, the highest score, lowest score, strike percent, spare percent, and open percent for those games. First ball stats show the number of games, game score average, attempts, count average, strike, strike percent, pocket percentages, carry percentages, and frame percentages. The second ball stats shows a spare count, open count, conversion rate, spare percent, open percent, split percent, and washout percent. The “Clear” button clears anything selected in the dropdowns.

Local Database

The local database, implemented using SQLite, stores all user-entered data directly on the device. It supports multiple records for key entities including Accounts, Establishments, Events, Sessions, Games, Frames, Shots, and Balls. As the data is stored locally, it will be permanently deleted if the application is uninstalled. To facilitate bulk data entry, the application supports importing CSV files. These files are processed through the MainViewModel, which parses

specific CSV formats to extract and load the corresponding data objects. All SQLite operations are organized within the application's data folder. Each data type has its own dedicated Repository class, providing a clean and modular way to access and manipulate records in the database.

Cloud Database

Cloud database integration with the mobile application's local SQLite database is implemented with data sync functionality. The former API Page now allows the user to connect to multiple endpoints in the cloud server. The user can build a request themselves and include specific data for the server to process. This feature was added as a test for our mobile connection to the cloud server. The user also can press the ‘data sync’ button. This button sends all the data from the mobile database to the cloud, and adds any data that is not present to the cloud database.

Watch

Overview

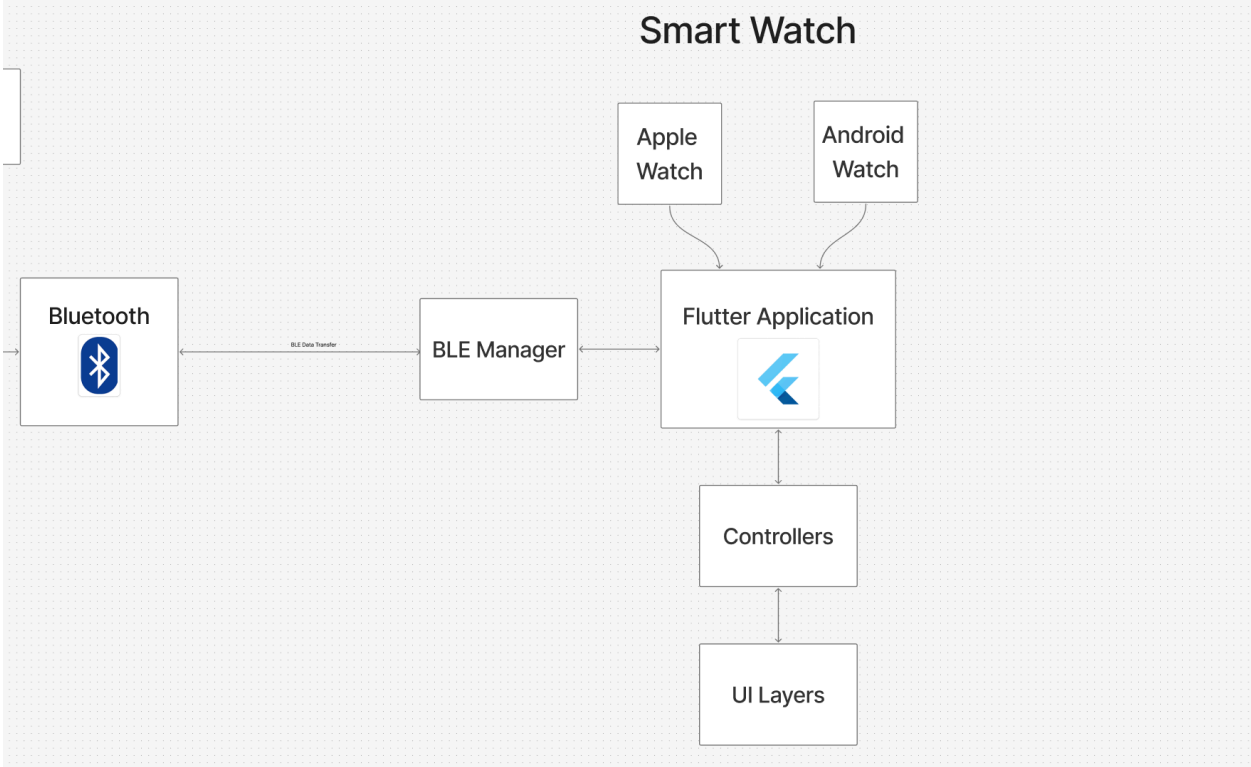


Figure 2.3.1.1: Screenshot of the high-level overview for the Smart Watch application

As shown in Figure 2.3.12 and Figure 2.3.1.1, the smartwatch application is designed to function exclusively as a data collection client within the RevMetrix system and does not act as a persistent data owner. Long-term data storage and authoritative session management are handled by the smartphone application and its associated relational database. The watch's responsibility is limited to capturing user input at the frame and shot level and forwarding that information upstream for processing, storage, and analytics.

Communication between the smartwatch and mobile application is handled entirely through Bluetooth Low Energy, which serves as the primary inter-device communication layer. As illustrated in Figure 2.3.1.1, BLE data flows through a native Bluetooth stack layer, bridged to the Flutter application through Method Channels. The BLEManager serves as the central mediator, coordinating packet transmission, managing connection state, and handling incoming data reassembly. In the architecture, data is structured using two packet formats: fixed-size BLEPackets (23 bytes with a 5-byte header and 18-byte payload) for streaming session data, and variable length AccountPackets for account information. The smartwatch operates as a BLE peripheral advertising the RevMetrix service, while the mobile device functions as the central client, responsible for initiating connections and reconstructing multi-packet messages through a reassembly mechanism.

The smartwatch application itself is implemented using Flutter and Dart, allowing a single shared codebase to support both Wear OS and WatchOS, as depicted in the platform abstraction layer of Figure 2.3.1.1. Flutter was selected to simplify cross-platform development while maintaining access to BLE functionality through supported libraries such as flutter_blue_plus and flutter_reactive_ble. As shown in Figure 2.3.1.2, the Flutter application layer communicates with controllers responsible for session state, frame navigation, and BLE operations, which then propagate data upward to the mobile application. This layered architecture ensures that the watch remains lightweight, platform-agnostic, and tightly integrated with the broader RevMetrix ecosystem.

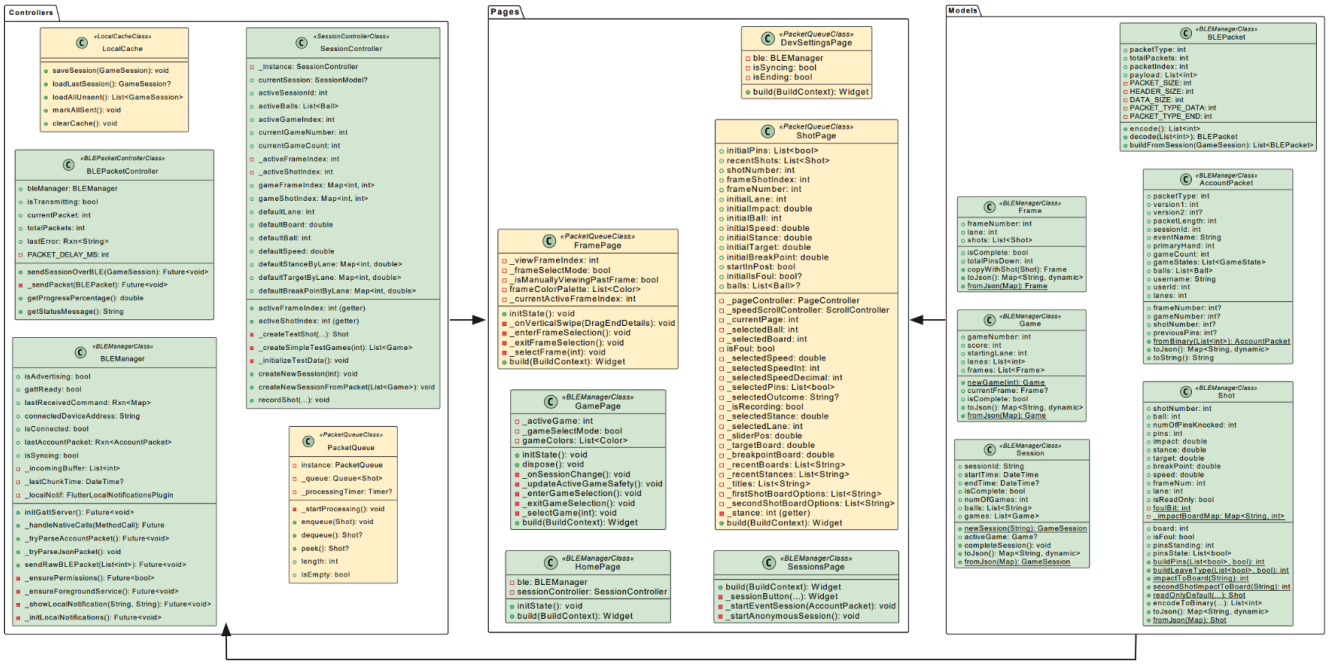


Figure 2.3.1.2: Smart Watch Application UML diagram

BLE Manager

The BLE Manager serves as the communication controller between the smartwatch and mobile application, confining all Bluetooth Low Energy operations into a single centralized manager. The manager is responsible for initializing the GATT server, advertising the RevMetrix service, managing device connections, and coordinating data transmission. The smartwatch operates as a BLE peripheral advertising a custom GATT server with two primary characteristics: a write characteristic for receiving commands from the mobile application, and a notify characteristic for transmitting session and account data. Once a connection is established, the BLE Manager bridges the native Bluetooth stack to the Flutter application layer through Method Channels, using platform specific implementation details. To ensure reliable data reception over wireless connections, the manager implements a chunk reassembly buffer.

Account Packet

The Account Packet is a variable-length binary message format received from the mobile application during session initialization, containing account and session metadata. It encodes user information like username, user ID, hand preference, session ID, event name, lane count, and game state information including an array of per-game frame/shot progress tracking, and a list of available balls. This format enables efficient transmission of session initialization data from the phone to the watch. The BLE Manager deserializes received Account Packets and passes them to the Session Controller for session state initialization.

Session Controller

The Session Controller is the primary state management controller for the smartwatch application, coordinating all session level data and game state throughout the application lifecycle. It follows the singleton pattern and uses ChangeNotifier to update things reactively. On startup, the controller initializes the session hierarchy from account packet data received via Bluetooth. The controller maintains the active session, game, frame, and shot while providing methods to create, edit, and record shot data. When a shot is recorded, the controller calls shot.encodeToBinary() to generate a binary packet and then directly transmits it via the BLE Manager. The controller tracks the active frame index, active game index, and provides reactive updates to the UI through NotifyListeners.

Session Model

The Session Model represents the root data structure for a complete bowling session and contains the session ID, event name, number of games, list of available balls, and a list of Game objects. The session ID is generated at session creation with a timestamp. The session receives

game count, event name, ball list, and game state information from the Account Packet transmitted by the mobile application. It serves as the entry point for all session data organization and hierarchical structure.

Game Model

The Game Model represents a single game within a session and contains the game number, list of Frame objects (up to 12 frames to handle the 10th frame's additional shots), incoming score reference from the phone, and starting frame index. The game number serves as an identifier within the session for database writes on the mobile application. The starting frame index indicates which frame the phone was on when the session was synchronized, allowing the watch to resume from that point. Each Game is dynamically created based on data received from the mobile application or as empty games for new sessions.

Frame Model

The Frame Model represents a single frame within a game and contains the frame number, lane assignment, and associated Shot objects. The frame number serves as an identifier for Bluetooth transmission and database writes. Each frame can contain one or two shot objects, representing the bowler's throws for that frame. Frames are generated either from phone data or as empty frames for new games.

Shot Model

The Shot Model represents the smallest unit of bowling data on the smartwatch, capturing all information associated with a single throw. Core fields include shot number, pins knocked down, ball used, stance, lane number, target, breakpoint, impact, ball speed, and recording state. When a shot is recorded and the user hits the submit button, the Session Controller calls

Shot.encodeToBinary() to serialize the shot into a compact binary format, then transmits it immediately via BLEManager.sendRawBLEPacket(). This custom binary encoding includes the session ID, game number, frame number, shot index, and all shot data, enabling incremental transmission of shots as they occur rather than bulk session transmission.

Settings Page

The Settings Page is an internal utility screen providing direct interaction with the smartwatch's Bluetooth system. It displays the current connection status and connected device address for debugging. Primary controls include a "Sync Session" button that requests the phone to send updated session data, an "End Session" button that completes the current session and returns to the session selection screen, and a "Log Out" button that disconnects from the phone and returns to the home page. The Sync button sends a sync command via sendSyncCommand(), the End button sends a nextSessionCommand() to the phone before navigating to the Sessions Page, and the Log Out button sends a disconnect command, clears the local session state, and navigates back to the home screen.

Shot Page

The Shot Input Page is the primary interface for recording per-shot information on the smartwatch, implemented as a page-based workflow optimized for the circular wearable display. The page uses a PageController to navigate through seven sequential screens: Recent Results displaying previous shots for reference, Select Ball for choosing the ball for this throw, Record for initiating video recording on the phone, Shot for entering pins left standing for that shot, Impact for selecting impact point, Boards for entering final lane and board information, and Speed for entering ball speed. Data is collected incrementally across these screens, allowing the

user to review recent throws before recording a new one. Once all screens are completed, the collected Shot object is passed to the active Frame, and the Session Controller transmits the shot via binary encoding to the phone.

Frame Page

The Frame Page serves as the navigation hub for shots within a frame, implemented as the FrameShell component. It displays the current frame information and provides access to individual shot input pages. The page supports swipe gesture navigation: vertical swipe down returns to the Game Shell, and horizontal swipes navigate between frames. It restricts frame navigation based on session state, preventing access to frames beyond the active frame. The page dynamically generates shot views based on the Frame Model.

Game Page

The Game Page serves as the home screen for the active gameplay, implemented as the GameShell component. It displays the current game number and provides users access to adding and deleting games within the session. The page supports horizontal swipe gestures to navigate between games within the session, and vertical swipe gestures to access the Frame Shell. The cog button provides access to the Dev Settings Page for system controls. The Game Shell acts as the primary navigation anchor for gameplay and does not permit direct data manipulation.

Home Page

The Home Page is the initial landing screen after application startup. It displays the RevMetrix branding and initiates BLE peripheral setup, including GATT server initialization and advertising. The page monitors the BLE Manager's connection status and automatically navigates to the Sessions Page upon successful connection or account packet reception. It also

listens for account packet events from connected phones and initializes the Session Controller with received session metadata.

Sessions Page

The Sessions Page displays available sessions and allows the user to select a session mode. It shows a personalized welcome message with the username from the received Account Packet (or cached username if disconnected), displays the event session name received from the phone, and provides buttons for starting the event session or an anonymous session. Upon session selection, it navigates to the Game Page.

Packet Queue

The Packet Queue is the buffer for outgoing BLE communications. It is a simple first come first serve queue that all of the shot packets are inserted into to then send to the phone. If there are multiple packets inserted into the queue, only the oldest one will try to send, then it will be removed from the queue and the next packet will be sent. If the queue is empty, the main send packet loop will stop until another packet is inserted to minimize overhead on the watch. A packet will be dequeued when a confirmation message is received.

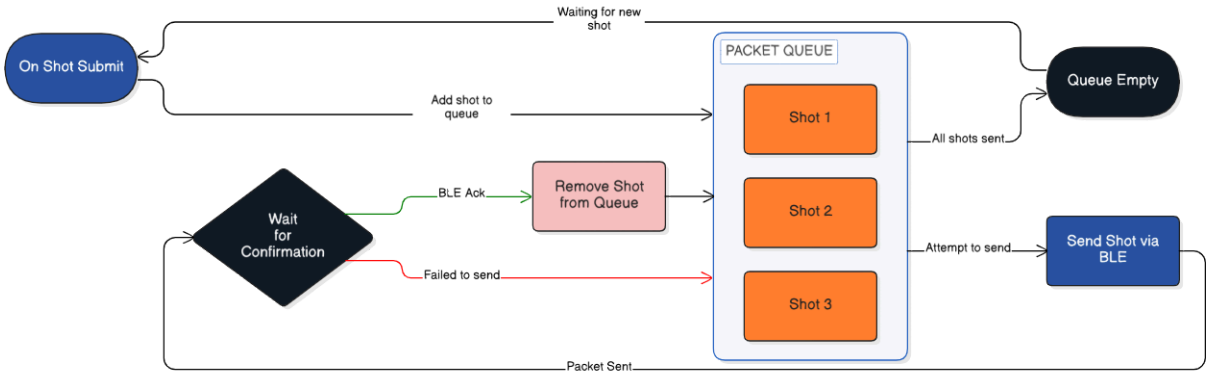


Figure 2.4.1.0: Packet Queue Diagram

Cloud

Overview

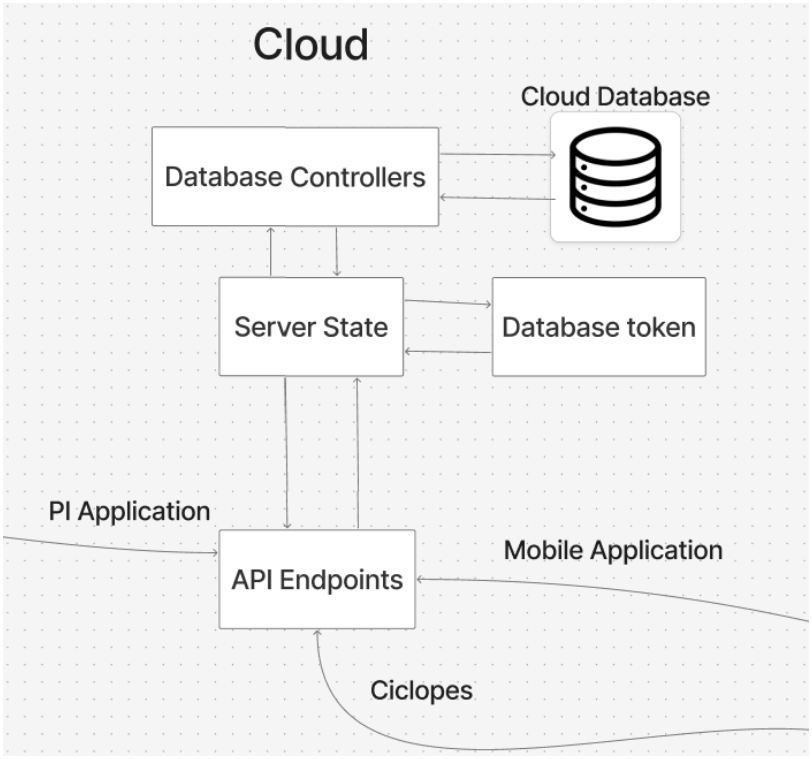


Figure 2.4.1.1: Cloud Application High Level UML

Figure 2.4.1.1 depicts a high-level diagram representing the internal interaction of components in the Cloud Application as well as the system's external connections to other facets of the project. The system is to serve as a mediator for data for the Pi, Mobile, and Ciclopes applications. These external systems communicate with the Cloud Application through our publicly available API endpoints. The API service used to create the endpoints comes as a part of the C# .NET Entity Framework CORE development environment. The endpoints then communicate with the state of the server to figure out which actions need to be performed based

on the data passed in the request. Once the appropriate action is determined, the server state will then pass the data from the request to the appropriate database controllers. The database controllers then perform any necessary validations on the data set passed in the request and then begin to store or retrieve data. The database controllers then communicate with the Microsoft SQL server in order to retrieve or create data. The Cloud Application also contains developer tools in order to improve the ability to modify, delete, and add to the database.

Entity Framework Core Developer Tools

The purpose of the Developer Tools was to simplify the process of managing the contents and version of the database. Beforehand, developers would write raw SQL code into a Liquibase SQL script file that would then be used to rebuild the database. The issue with this is that people can make mistakes when writing code, and SQL script will only create errors at runtime. The solution to this problem was to create some developer tools in order to abstract out the process of writing SQL code. The tool allows developers to write C# classes that contain all of the fields that they want a table to contain, specify special column types and parameters, and then have the system generate a SQL script file based on the class. This then allows errors to appear during the creation process of the table, rather than while the database is being updated which avoids database corruption. With this, the SQL script generator creates a robust SQL script file that ensures tables do not get created multiple times and has individual rollback procedures for each alteration or addition to the database. Additionally, the tools allow for database versioning. This gives developers an accurate timeline of when changes to the database were executed and also allows for the migration of the database back to an older version if necessary.

API

With the design of the new API implementation, a major factor was simplifying the complexity of requests to improve the usability for each application. To do this, the API expects that requests are sent with a list of objects, rather than just a single object. This is to eliminate the need for multiple requests to create multiple objects. On the server side, the API endpoints should have data validation and safe error handling to ensure that the whole server does not fail if a single error occurs. This was implemented by sending back an error code if errors occur, or if no entry was found or created simply sending back a blank object instead of throwing a server-side error. The Swagger Documentation also allows developers and users to easily see how to properly use each endpoint and what sort of requests can be made to the Cloud Application.

Database Controllers

The Database Controllers are the methods in the Cloud Application that get passed data from the API requests and then insert or retrieve data from the database accordingly. When the data hits these methods, firstly the data sets are validated to ensure that everything that needs to be present was sent in the request body. The controller then inserts the data into a prepared statement specific to each API endpoint, and then inserts or retrieves data as necessary. These controllers will also produce a response regardless of what the request is. When data is being created, if the creation was successful the controller will return a list of all the ids for the objects that were inserted into the database. On a successful retrieval, the controller will return a list of all of the objects found in the database matching the criteria sent to the API.

Ciclopes

Overview

Ciclopes from a technical perspective takes a video of the bowler from behind them, this is then processed using the computer vision model trained to specifically segment the lane and ball the bowler is using. The lane mask is then used to compute a homography transformation matrix, which is then used on the ball to get the ball and lane in a birds eye view representation. This allows retrieving the contact point of the ball on the ground in relation to the lane for each frame algorithmically. This can then be used to render the ball path, calculate kinematic statistics at different points down the lane, extract features like total break of the ball across the lane, and starting position. Which can be used by the user/bowler to make adjustments.

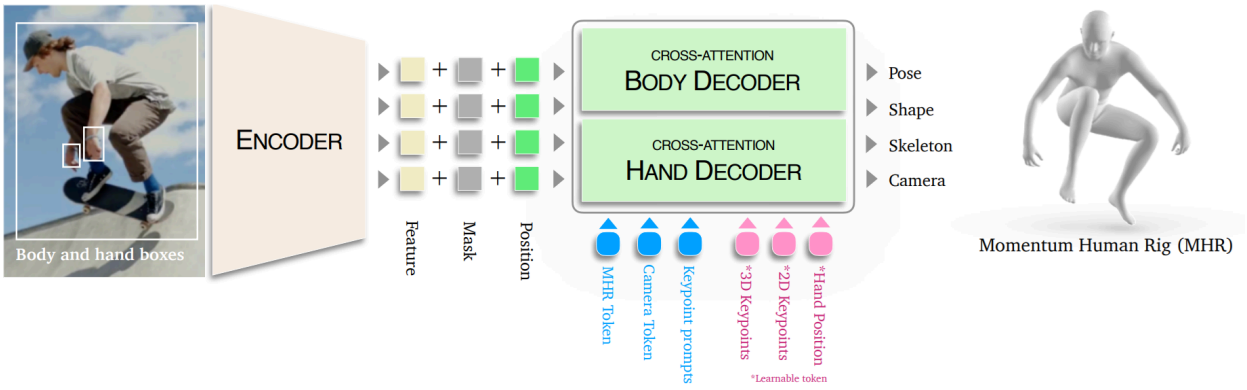


Figure 2.5.1.1: SAM3D Body Architecture

Alongside the ball trajectory feature we implement pose estimation using SAM3D Body. As seen in figure 2.5.1.1, an initial vision encoder produces features used as input for a larger cross attention decoder module which then produces the final pose, the camera position, and a mesh of the detected person to be rendered around the skeleton positions. For our use case we want the camera position and skeleton coordinates for client side rendering, the mesh increases

complexity of rendering and does not provide useful information for the use case of biomechanical analysis as the skeleton pose fully conveys the body position per frame. The video is batched and processed with SAM3D Body in parallel and the estimated poses are stitched together per frame recreating the video. Exponential moving average (EMA) is used across the produced pose video to reduce jitter while preserving the coordinates of the estimated pose.

The technical design of Ciclopes consists of the client side data collection (covered in depth in the Mobile sections), and a backend processing service called Ciclopes-API which runs inference of the AI models and further algorithms.

Model Flow

The system uses a modified version of the YOLOv26n-seg model from Ultralytics, which is pretrained on large general datasets of object detection and segmentation. The model is trained to detect/segment many classes such as a football or person. Originally, per run it detects every class it finds in the image. This differs from our required use case where it is only required to segment a ball and lane. We leverage transfer learning and do supervised fine tuning on this model on a dataset of labeled real images to teach it how to segment the ball and lane. We are leveraging the preexisting capabilities of the model to segment and detect objects extremely well, and then further teaching it to detect/segment the specific classes we want it to and only those classes.

MLops

The MLops of this project includes versioning the model used in our production Ciclopes-API service, and the dataset it was trained on. This allows our team to know which

version of the model is currently in use, and be able to better benchmark the performance of different architectures, learning techniques, and trained-on datasets.

Client Side Feature Design

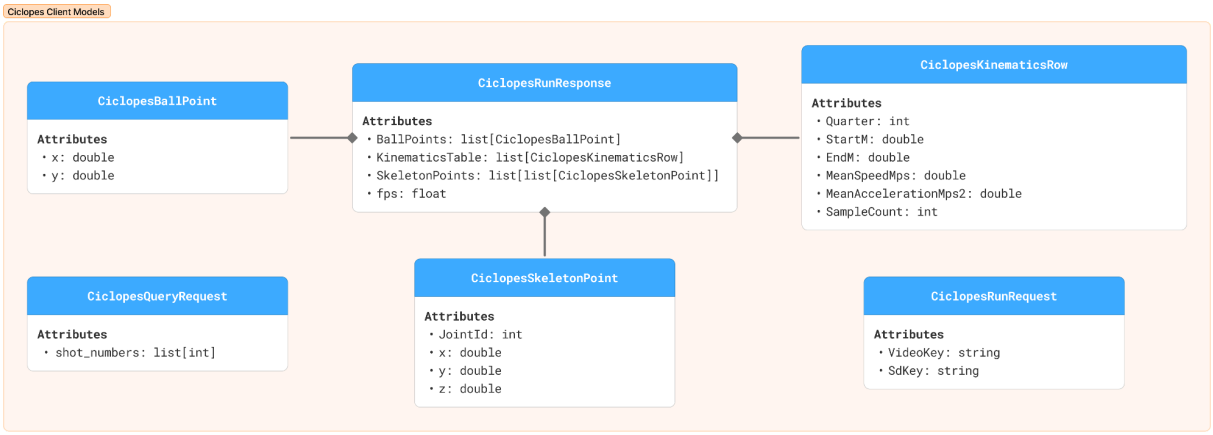


Figure 2.5.4.1: Client Side Model UML

The client side utilizes a RESTful API request and response format where video and smart dot data keys serve as the request format for all requests to the Ciclopes-API, and the response model forms the data that will be visualized. This allows the response model to require no serialization and serve as the view model for the UI components. This makes things clean and scalable to new response formats making visualization directly usable without complex refactoring.

Ciclopes-API Design

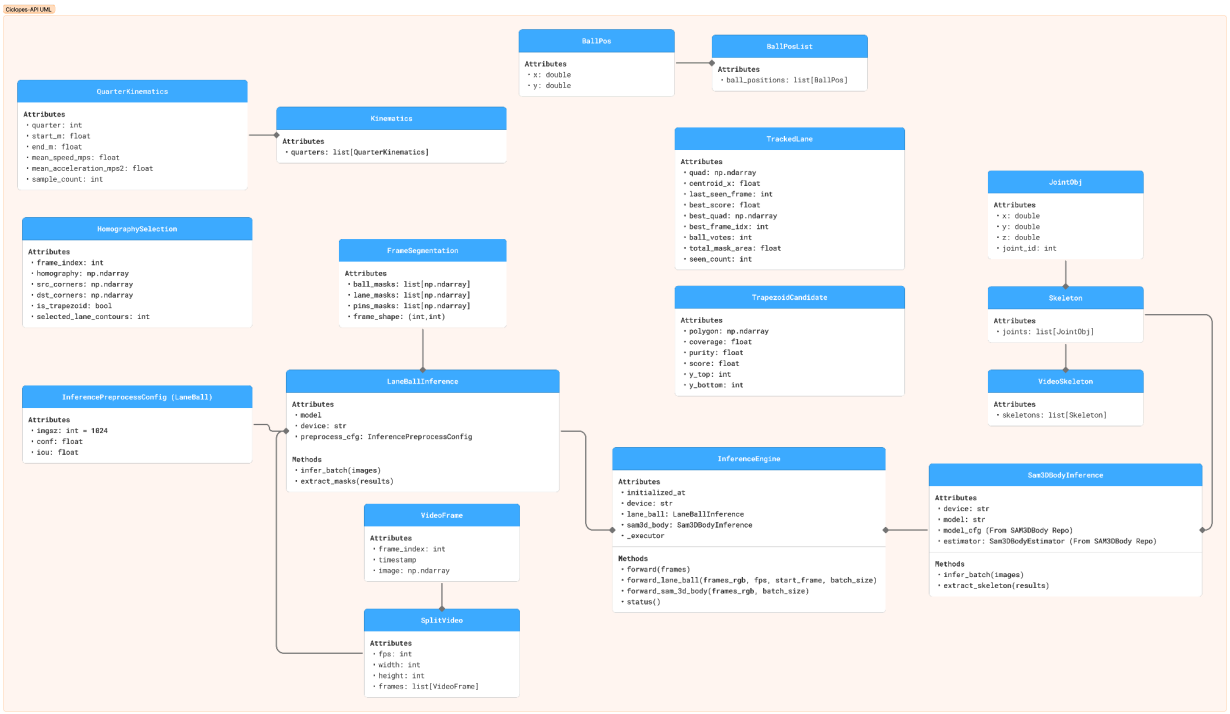


Figure 2.5.5.1: Ciclopes-API UML

Ciclopes-API is a FastAPI based backend service which requires at least one GPU/accelerator to run the AI models, and handles postprocessing of the inferred information to produce the final user facing data. We separate the inference concerns, and postprocessing into two distinct portions of the application. Inference is handled by the InferenceEngine singleton which handles initialization of GPU resource acquisition and initial weight uploading into memory of the AI models. Postprocessing is done functionally through pure functions allowing greater testing and use of dependency injection through multiple layers of an algorithm keeping the code readable. Ciclopes-API exposes routes which take in keys for the video and SmartDot data (explained in more detail in the Mobile and Cloud sections), there is an aggregated route which does runs both models and algorithms, and there are separate routes for the ball trajectory feature and pose estimation feature.

Inference Engine

As described above, the InferenceEngine singleton handles all GPU resource interfacing, and also handles batching / preprocessing for inference. Specifically, initialization goes through a process of first based on the config, either collecting one or two GPUs for the service. For development on our personal machines one is used, and two for our “production” environment for end-to-end testing and serving during demos. Both model weights are loaded on the single GPU setup, and when two are available, both models get their own GPU to maximize the parallelization for their individual inference workloads. Simple interfaces are exposed to be used in the route level workflows to run inference and get the results for postprocessing.

Postprocessing Algorithms

Postprocessing as defined earlier uses the data models produced from the inference engine and through multiple layered functions produces the response output models. Specifics are defined in detail in the implementation section. From a design standpoint, in order for the ball trajectory to be produced we must be able to detect the four corners of the actively used lane. This allows us to compute a transformation from the image coordinates to the known lane size rectangular coordinates. This transformation is the homography matrix, this homography matrix can then be used to transform the approximated ball contact point from the ball detection to lane coordinates. When done per frame we then have the trajectory the ball took over the video. Pose estimation in terms of postprocessing simply applies EMA smoothing to reduce jitter in the estimated poses while preserving the positions of the limbs.

Constraints

There are many constraints that needed to be considered in the implementation of this project. First is the issue of indoor bowling alley scenes. Due to the amount of noise in the scene

from screens and overhead lighting reflected on the glossy lane, the vision model has a difficult time finding the signal due to the reflection of other objects where it expects the lane to be. This was handled by training on real world labeled scenes that have specular reflections on the lane, specifically at the end of the lane to get good segmentation from the two end corners.

A main constraint which defined how the approach was formulated was the monocular nature of mobile device video recording. Depth cannot be algorithmically reproduced as if there were multiple cameras with known position differences. This specifically causes issues with finding when the ball lands on the lane from the bowl, as homography requires the points being transformed to be on the same plane. We did find in experimentation (discussed in depth in the Implementation section) that if you plot the processed ball positions for all frames of the video, there is a distinct uniform distribution of points when the ball is in contact with the lane and before the ball hits the pins, allowing for algorithmic truncation of the anomaly points which fall before the ball is in contact with the lane, and after the ball hits the pins leaving the true ball trajectory intact. However, accuracy cannot be promised in truncation, and overall accuracy of the produced ball positions adjacently is limited to the camera resolution, collected frames per second, and number of cameras alongside the actual detection model performance.

Ball Spinner Controller

Overview

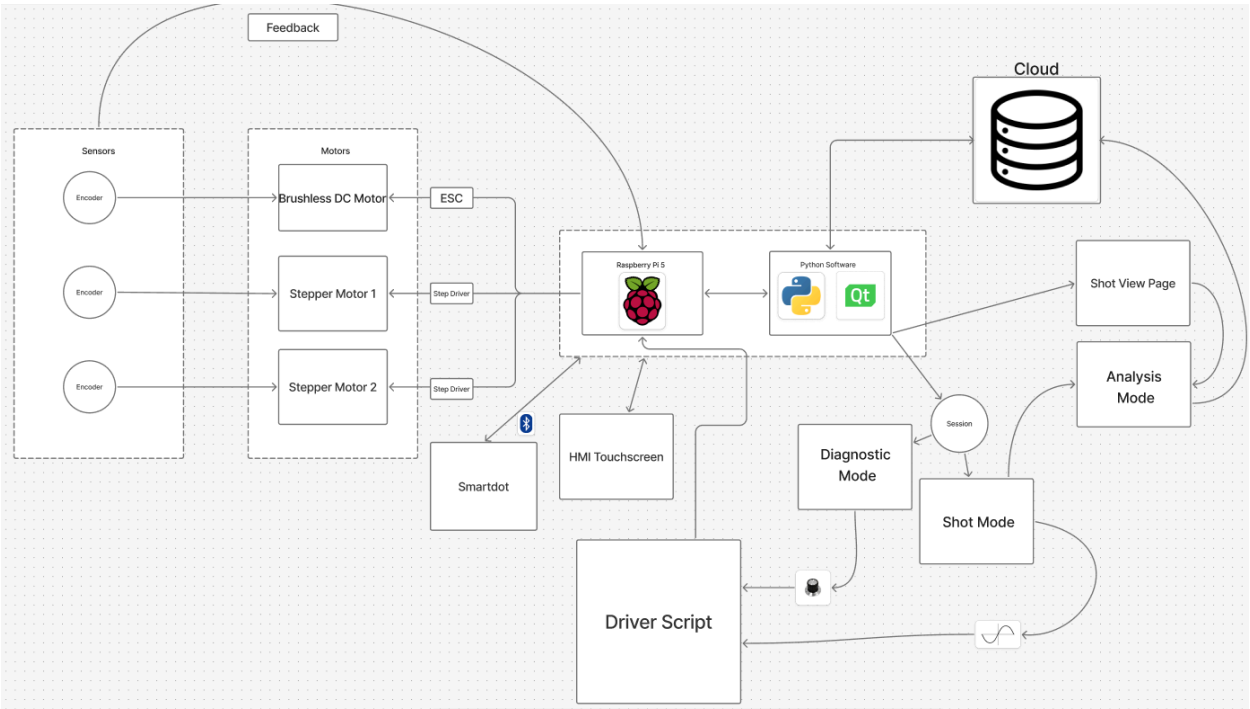


Figure 2.6.1.1: Ball Spinner Controller High Level UML

Figure 2.6.1.1 demonstrates the Raspberry Pi central system with its main I/O systems and software systems. The Ball Spinner Controller is designed to run on a Raspberry Pi 5 (Pi) The Pi's GUI is built through PyQt6, which is not Pi 4 or 5 dependent. The hardware that can be connected to the Ball Spinner Controller includes: 2 Stepper Motor Drivers which connect to their respective stepper motors, 1 ESC which controls the Brushless DC Motor, and a BLE connection to the MetaMotionS which acts as our SmartDot. Additionally, a 16 inch (1920x1080) touchscreen display is connected to the RPi5 where our GUI will be loaded. On this GUI, a user will initially be taken to the Home page where they can choose between three modes, Shot, Diagnostic, and Analysis. An interface will exist where users can connect to SmartDots as they wish.

Shot mode allows a user to construct three graphs, one for each axis of rotation. These graphs will be converted into motor instructions which will then be used by the Driver Script to send the motor instructions to each respective motor after clicking “Enter Shot”. The length of the shot can be set anywhere in the inclusive range of 1 to 3 seconds. While a shot runs, a live feedback screen will be displayed and the user will be prompted to go to the analysis page to further analyze the results of the data.

Diagnostic Mode allows a user to command an individual motor to move within the limits of the system. This system will use a secondary version of the Driver Script that is built to take in live inputs.

The Ball Spinner Controller will connect to the Cloud Application in order to store the data that it collects/creates. The data the controller will be collecting is the SmartDot data (accelerometer, gyroscope, magnetometer, and light), encoder data, and heat data. The data that the controller will create is that of the Session, Shot Script, and Diagnostic Script. The session will handle organizing the data collection serving as a foreign key for all data to connect to. The Shot Script is responsible for controlling the motors after entering a shot. The Diagnostic Script will control the motors in real time as inputs are received.

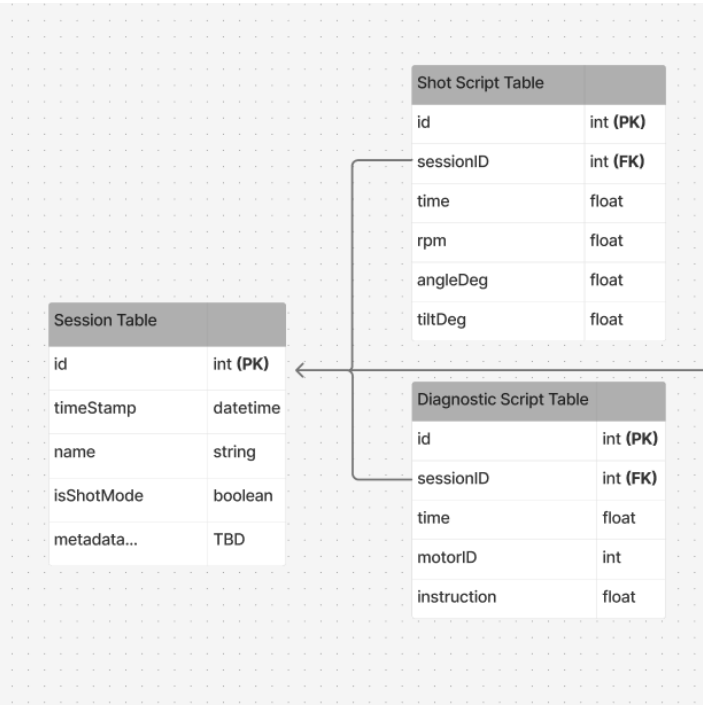


Figure 2.6.1.2: Database Schema for Session, Shot Script, and Diagnostic Script

In Figure 2.6.1.2 we can see that there are two different scripts that are used to drive the motors. These different scripts are used on the Shot mode and Diagnostic mode respectively. Their respective sections will go further into detail as to why this design decision was made.

When a user enters Shot or Diagnostic mode, a new Session is created. This session is where all of the collected data will be attached to the Data Controller as explained in the Cloud and Models section. The Shot View page provides the user with the ability to search for all sessions that have been saved to the Cloud. The user can then select a session, giving them access to the script that ran that session and all of the data that was collected during that session. A user can then choose to replay that session, creating a replay iteration and collecting new data while rerunning that identical Driver Script. The user can also choose to simply view the data

that was stored on the cloud inside of analysis mode. The user can additionally load the shot script into the editor to modify the script before running a new shot.

Lastly, Analysis mode will either appear after a shot has been taken or after a user selects a shot from the list of previous shots on the Shot View Page. Analysis mode will give the user the ability to view the collected data on a graph displaying motor data and a graph displaying SmartDot data.

Hardware Design

The hardware design for the Ball Spinner Controller is focused on developing a reliable system capable of driving three independent motors across three degrees of freedom. At this stage of the project, only the first two axes have active motor hardware. For the primary axis, we decided on using a brushless DC (BLDC) motor and for the secondary axis, we are using a stepper motor. The design is structured so a third motor can be added without significant redesign. Each motor requires a compatible driver, stable power distribution, and a wiring layout that keeps the system safe and adaptable as components evolve. The following sections describe the design decisions behind the motors, drivers, power system, and wiring approach that form the foundation for the controller.

Motors

The first design decision involved choosing the right motor for each axis. For the primary axis, which is responsible for the spinning of the SmartDot module at higher speeds, we quickly determined that a stepper motor would not suffice. This is because steppers struggle with continuous high speed operation and cannot provide the smooth rotation required. This led us to explore brushless DC motors. BLDC motors provide smoother continuous rotation at higher speeds and are capable of replicating the rapid spin of a bowling ball with enough torque to

maintain that speed under load. Testing started using an A2212/13T outrunner BLDC motor. This smaller motor helped us understand the BLDC operation and control, but was not powerful enough or consistent enough for the actual application.

After some work with the Mechanical Team, A required torque for the primary motor was calculated using the torque equation shown below.

$$\tau = \frac{\pi \rho r^2 \ell^3 w_f}{12\Delta t} \rightarrow \tau = \frac{\pi(2710(kg/m^3)(0.0127m)^2(0.2159m)^3(104.7\ rad/s)}{12(0.25s)}$$

$$\tau = 0.482\ Nm \approx 1\ Nm$$

Based on these specifications, the E3665 sensored BLDC motor was selected for the primary axis. It provides higher torque, smoother rotation and the ability to work with hall sensors for a more stable control. It aligned well with the performance requirements of the spin axis.

For the secondary axis, which requires precise positioning as opposed to rotational speed, a stepper motor became the more appropriate choice. A NEMA 17 stepper motor was selected due to its predictable movements and ability to hold positions without complex feedback. Its low speed and high precision makes it ideal for the second degree of freedom for the system.

Motor Drivers

Each motor requires a driver suited for its control. For the stepper motor, the DM542 driver was chosen. It accepts step and direction signals directly from the Raspberry Pi and

provides adjustable microstepping and current limits. The driver runs off of 24 V and provides enough torque and stability to prevent missed steps.

For the BLDC, we initially used a basic ESC for early testing with the A2212 motor. However, it became clear that sensorless control lacked the startup stability and the low speed performance that is needed by the first axis. This led to researching a new ESC and the adoption of the VESC motor controller. This ESC supports hall sensor inputs and closed loop control. The VESC also provides current, RPM and temperature data.

Power Distribution

The Ball Spinner System hardware design revolves around 24 V power supply. The system is being supplied by the P1-150-24 PSU, which converts the incoming 120 VAC main into a stable 24 VDC bus [50]. This 24 V rail feeds each of the motor drivers through their own fused branches, 3A fuses for the two DM542 step drivers and a 6A fuse for the Flipsky FSESC 4.20. Since the Raspberry Pi cannot operate on 24 V, a 24 to 5 V buck converter provides a clean 5 V supply that is dedicated to the Pi. A second buck converter is needed to generate 12 V for the HMI display. All components share a common ground to ensure reliability. In the architecture, depicted in Figure 2.6.2.3.1, 24 V paths are kept separated from the low voltage to avoid any complications. This distribution layout forms a basis for future PCB work as well which will consolidate the power.

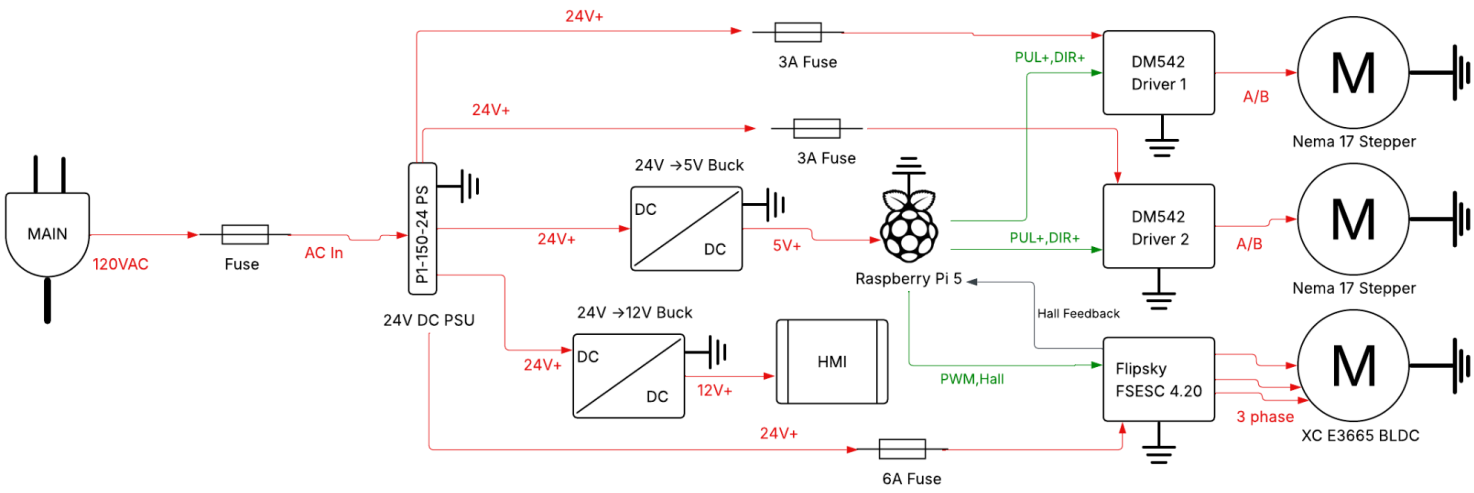


Figure 2.6.2.3.1: Initial power distribution architecture for the BSC

Logic Level Shifter

The Ball Spinner System's Stepper motors require a dependable way to bridge the voltage gap between the Raspberry Pi's 3.3 V logic and the 5 V control signals expected by the motor drivers. A pre-built BSS138-based 4-Channel Bidirectional Logic Level Converter Module is used to handle this translation cleanly, ensuring that the Pi can interface with higher voltage inputs without risking damage to its GPIO pins. This keeps the control path consistent with the rest of the hardware while avoiding missing any micro steps while the system is running.

SmartDot

The SmartDot is designed as an interface that provides the ability to connect, configure, collect, and disconnect from a module. An interface is used because there exists multiple different technologies that will implement this interface. Currently, the MetaMotionS, Figure 1.1.2.4, and the Simulated SmartDot are the only two that implement this interface. The SmartDot will collect data from its sensors, refer to Figure 2.6.3.1 to see the database schema

that will represent the SmartDot data on the Cloud, and then at the end of a session, it will upload this data to the cloud.

SmartDot Data Table	
id	int **(PK)**
sessionID	int **(FK)**
data_selector	int from 0-3
time	float
XL_X, Y, Z	3 different floats
GY_X, Y, Z	3 different floats
MG_X, Y, Z	3 different floats
LT	float
replay_iteration	int

Figure 2.6.3.1: Database Schema of SmartDot Data Table

Another class called the SmartDotConnectionManager manages storing and handling multiple SmartDot object connections concurrently. Since the physical SmartDot Holder holds two MetaMotionS modules for counterbalance, in the future we will use this to connect to both modules and compare and contrast data.

Models

The system must be able to record all of the necessary data that should be collected. In order to do so a Session is created. This Session serves as an identifier for which data belongs to which shot. A session is created when the Shot Mode or Diagnostic Mode page is opened. The following data models will then be updated and stored in real time by our DataController. This controller handles building out lists of our data and preparing it for the Cloud. The following

data models are created to be uploaded to the Cloud. The Diagnostic Script is used to track the inputs that are sent from Diagnostic Mode into the Driver Script.

Diagnostic Script Data - controls one motor at a time:

- Time of instruction (float)
- Motor ID (int) - 1: Spin (x-axis). 2: Angle (y-axis): 3:Tilt (z-axis)
- Instruction (float)

Shot Script Data: used to track the inputs that are sent from Shot Mode into the Driver Script.

Shot Script Data - controls all three motors at once:

- Time of instruction (float)
- Rotations Per Minute [RPM] (float) - x-axis
- Angle Degree (float) - y-axis
- Tilt Degree (float) - z-axis
- ShotPoints (string)

A Diagnostic Script and Shot Script data cannot exist inside the same Session. SmartDot data is stored with a timestamp, either a set of 3 corresponding x, y, and z values, or a light value, see Figure 2.6.4.1 for the list of all variables inside of the SmartDot data object. A data selector is used to choose which set of data will be valid. In the future work section there will be a note about optimizing this data setup to store the data in multiple tables in order to not waste fields. See Figure 2.6.4.1 of the different data models below.

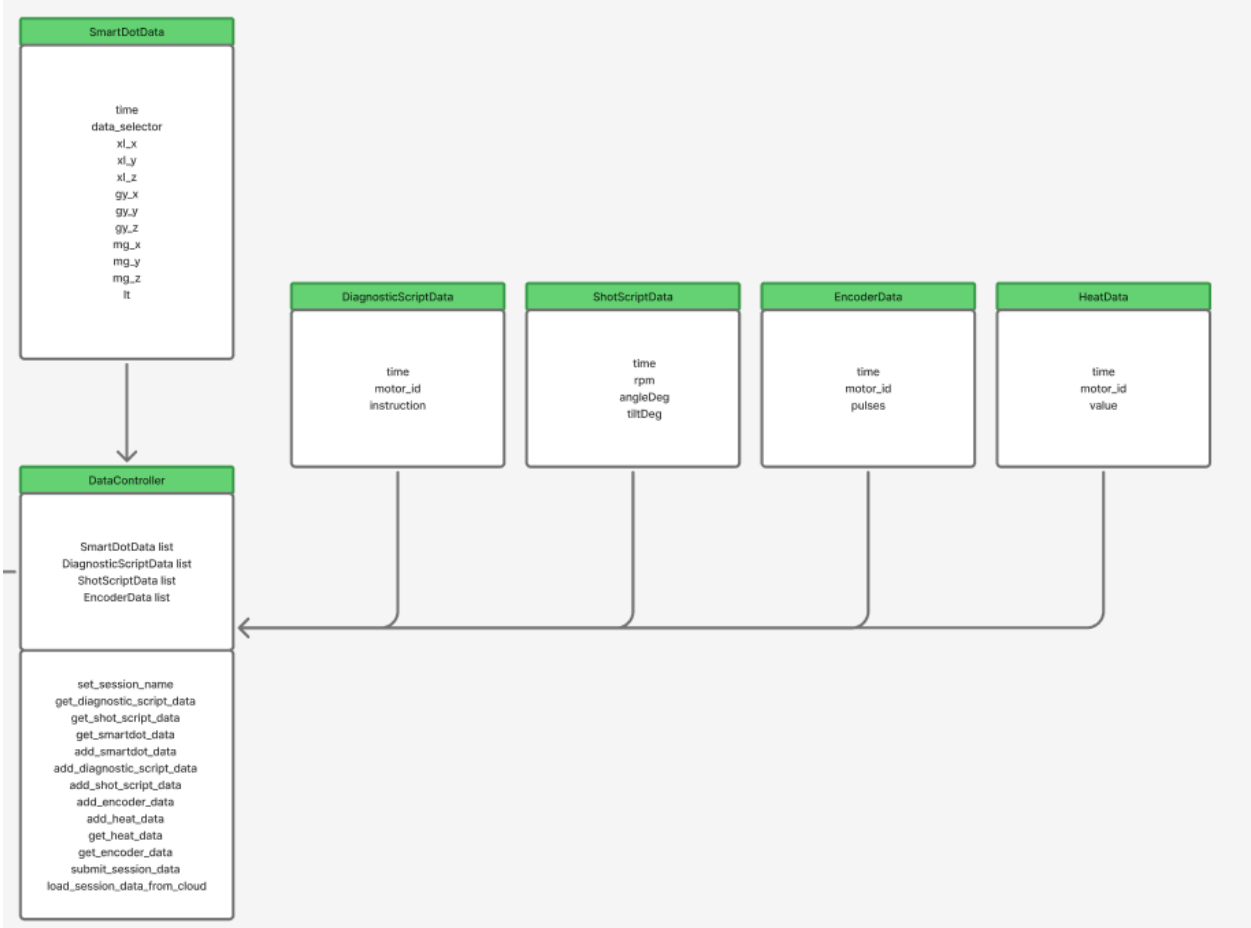


Figure 2.6.4.1: UML Diagram of Core Data Models

Lastly we also store Encoder and Heat data to track more information about how the motors actually performed.

Cloud

The Ball Spinner Controller is designed to locally store its data in objects. When a user finishes a Shot Mode or Diagnostic Mode session, they will be given the option to upload this local data to the Cloud. The system is designed into an APIUtils class and a CloudAPI class. The API utils class helps us abstract out our GET and POST requests and handles all the error

catching and timeouts. The CloudAPI class directly communicates with the Digital Ocean droplet through HTTP requests.

There will be the ability to POST and GET any of our data from the aforementioned data models. Additionally Sessions will be requestable by a range of their DateTime parameters. Below Figure 2.6.5.1 shows all of the functions that the CloudAPI will use.

Figure 2.6.5.1: UML Diagram of the CloudAPI

UI Design

UI

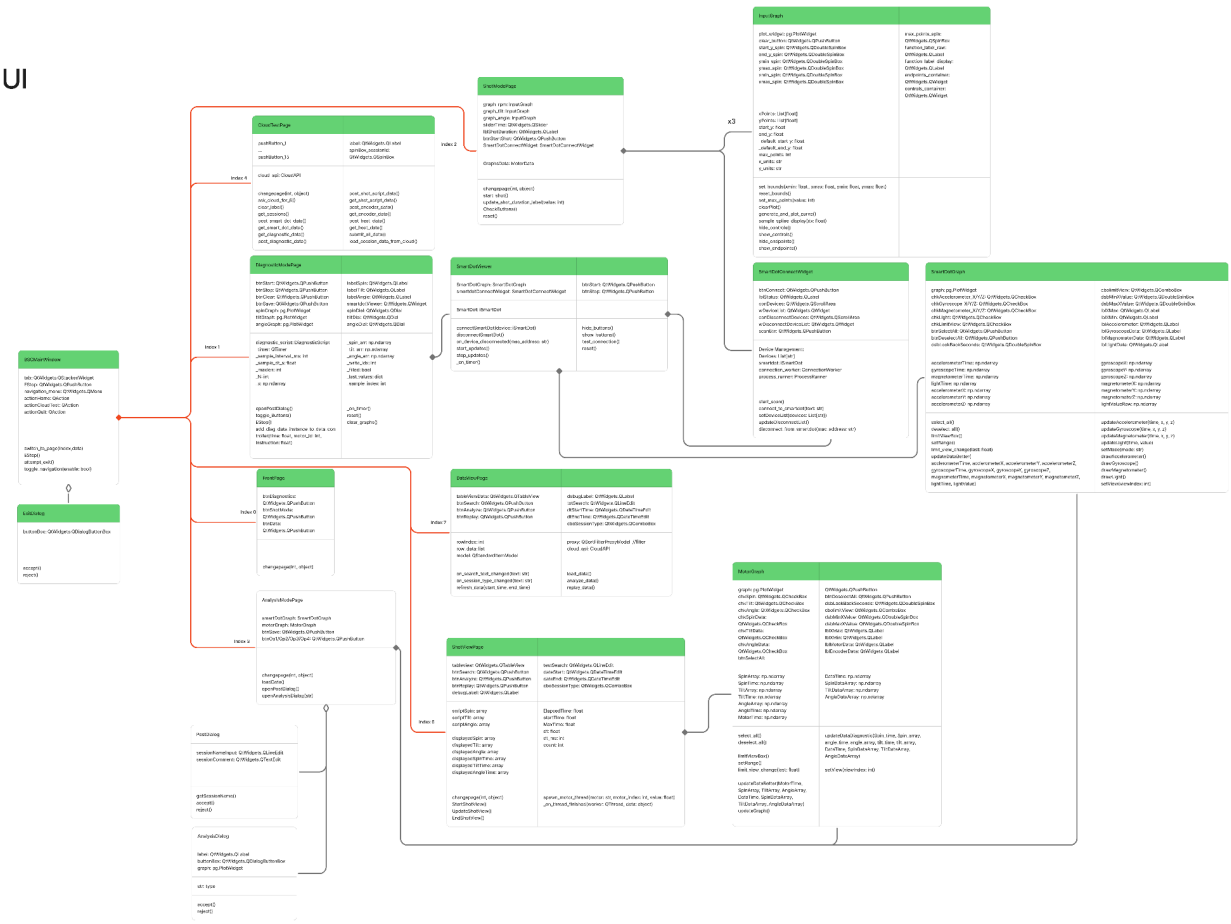


Figure 2.6.6.1: UML Diagram of the User Interface of the Ball Spinner Controller

BSCMainWindow

The BSCMainWindow is the base of the entire Ball Spinner Controller application. This window contains all of the other pages as well as an Emergency Stop button, which is designed to shut off the motors if portions of the UI become unresponsive.

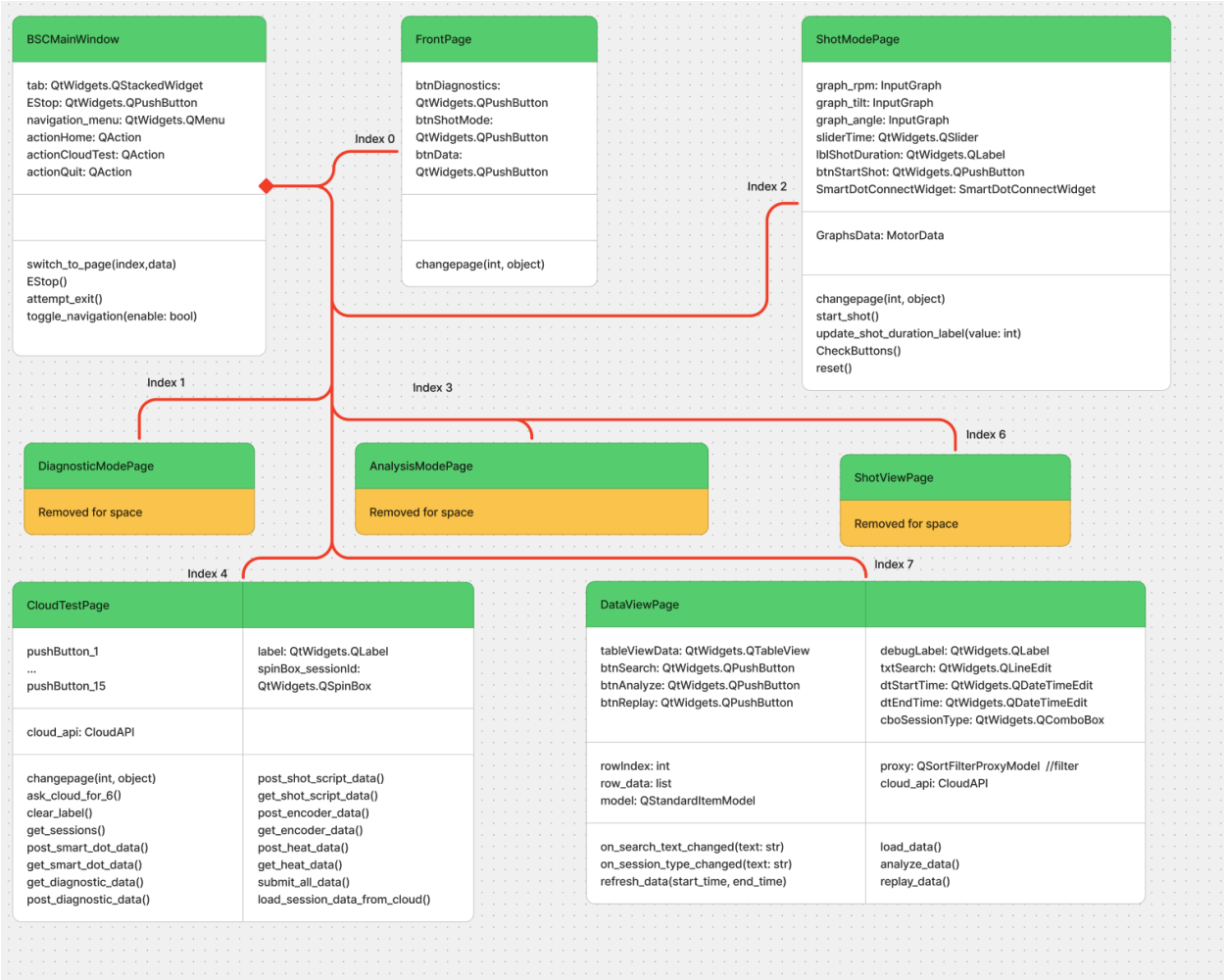


Figure 2.6.6.2: UML Diagram of the pages within BSCMainWindow

As shown in *Figure 2.6.6.2*, all of the pages are contained within the BSCMainWindow. As shown in *Figure 2.6.6.1.* many of the pages themselves contain other UI elements, those will be discussed within their respective sections.

Front Page

The Front Page is the initial page for the Ball Spinner Controller. Its main purpose is to provide the user with the ability to navigate through the application's primary modes: Diagnostic Mode, Shot Mode, and Analysis Mode. To access Analysis Mode, the user must either run a shot or navigate through the existing Data View Page.

Diagnostic Mode Page

The Diagnostic Mode Page is designed to allow the user to test the Ball Spinner Controller. Contained within this page are the controls for the three motors, graphs to view the motor instructions over the last few seconds, labels to display the motors current values, and a SmartDot Viewer to view SmartDot data while in Diagnostic Mode.

Shot Mode Page

The Shot Mode Page is designed to allow for the simulation of an actual bowling shot. The page contains three Input Graphs for controlling the controller's three motors. At the bottom of the page is a slider for controlling the shot length. A SmartDot Connect Widget is located on the side, allowing the user to connect to a SmartDot. When the user confirms the shot, the page uses the shot graphs to populate the BSC Shot Data Controller and switches to the Shot View Page to execute the shot on the motors.

Shot View Page

This page is designed to run the shot currently stored in the Ball Spinner Controller. It executes the motor instructions from the shot, plotting when the instructions are given and the encoder data on the left Motor Graph. Simultaneously, the page starts the SmartDot data collection and graphs the results in the left SmartDot Graph. When the shot is complete, the SmartDot data is loaded into the data controller, and the user is given the option to view that shot in Analysis Mode.

Data View Page

This Page is designed to allow the user to replay or analyse a previous shot stored in the Cloud. This page allows the user to obtain shots within a given date range and allows the user to filter the shots further by the name as well as the shot type. When a user selects a previous shot in the table they are able to navigate to either the Analysis mode to view the previous shot in its entirety or they can replay a shot using the same input parameters.

Analysis Mode Page

This page is designed to allow the user to analyze a shot, whether it was just created or retrieved from the database. The Motor Graph and SmartDot Graph display the data generated and stored from the executed shot. The buttons within the center allow the user to perform various data analysis operations, such as Wavelet analysis and Fast Fourier Transforms.

Input graph

The Input Graph is designed as the primary input method for controlling a motor when in Shot Mode. The graph includes controls for adjusting the following parameters of the input curve: the minimum and maximum values, the length, the starting and ending values, and the maximum number of points along the curve. The Input Graph contains accessor methods for all of the properties, as well as the ability to hide the controls from the user. In the implemented system, the controls for the minimum and maximum values and the shot length are hidden from the user. This is because the former is determined by the motor and the latter is determined by the slider on the Shot Mode page. When a user clicks inside the Input Graph, the point they clicked is plotted, and a cubic spline is used to connect all of the points. The spline is bounded by the graph's minimum and maximum values to ensure it never commands a motor to move

beyond its specified limits. The Input Graph also contains a utility function that exports the given input graph as a series of points separated by a given delta time variable.

SmartDot Connect Widget

The SmartDot Viewer page was designed to allow the user to first initiate a scan for nearby MetaWear devices, then connect to them. In addition to this, the SmartDot Viewer page will allow the user to disconnect any active connections as well. This page will use the user SmartDotConnectionManager tool to keep track of registered SmartDots.

SmartDot Graph

The SmartDot Graph provides a unified way to display data from the SmartDot. It contains stored curves for all of the SmartDot's various sensors. The page includes checkboxes to toggle the display of the corresponding curve. When the graph is clicked, a vertical cursor is placed, and the nearest points on the curve are marked on the graph and on the labels below it.

Motor Graph

The Motor Graph provides a unified way to display data from the motors. It contains stored curves for the three motors' instructions as well as the encoder values from the three motors. The page includes checkboxes to toggle the display of the corresponding curve. When the graph is clicked, a vertical cursor is placed, and the nearest points on the curve are marked on the graph and on the labels below it.

Ball Spinner Mechanical System

Overview

The Ball Spinner Mechanical system is used to simulate real bowling data by rotating a ball analog in all three degrees of freedom. Each degree of freedom will be controlled independently by the Ball Spinner Controller, as seen in Figure 2.6.1.1. The first degree of freedom about the x-axis direction will consist of a friction fit connection between the SmartDot holder and the motor shaft, along with a set screw to prevent slip. At both ends of the SmartDot holder, a SmartDot will be housed via friction fit. The SmartDots are inserted perpendicularly to the direction of rotation, minimizing the ability of rotational effects to dislodge them. Opposite the SmartDot slots are holes to allow the SmartDot module to be pushed out to allow for easy exchange. This SmartDot holder was designed with weight as a critical factor to reduce the required torque from all motors. This is especially true for the first degree motor as it will need to accelerate to 600 rpm in 150 milliseconds. The second degree of freedom system will control the rotation about the y axis, and will hold the first system on a rotating platform. A motor will drive the rotation of the top platform at a slower rate, rotating up to 45 degrees in each direction within one second. The third system will control the z-axis rotation of the previous systems. It will rotate up to 22.5 degrees in both directions in less than a second. This design places all three axes of rotation so that they intersect at the center of the simulated bowling ball to ensure that collected data is consistent with real-world motion. The final model of this system can be seen below in Figure 2.7.1.1 with the x, y, and z axes labeled.

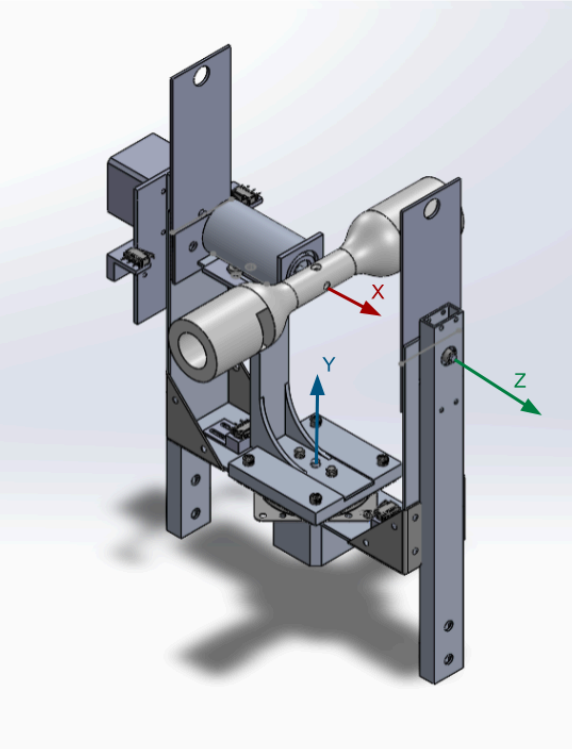


Figure 2.7.1.1: SolidWorks Model of Final Design for the Three Degrees of Motion on the Ball Spinner Mechanism

Implementation

Mobile

Main Page

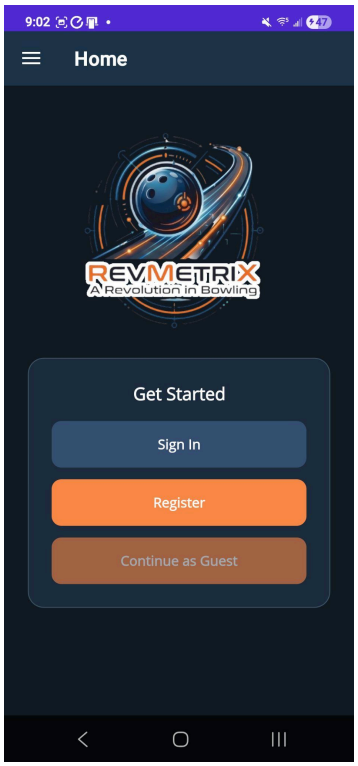


Figure 3.1.1.1: Main Page - Logged Out

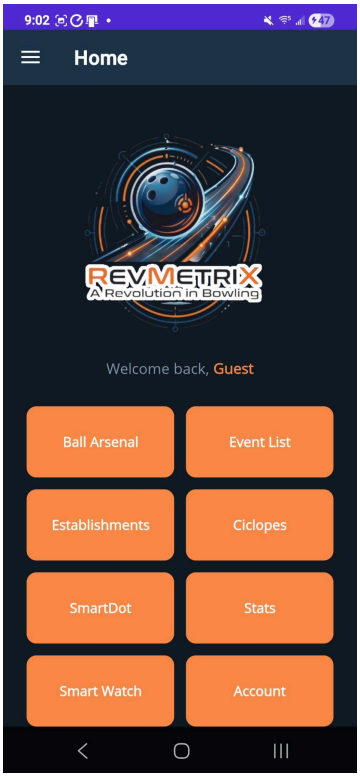


Figure 3.1.1.2: Main Page - Logged In

The main page, as seen in Figure 3.1.1.1 and Figure 3.1.1.2, connects the user to every page in the mobile application. The page contains XAML Button elements which each have their own OnButtonClicked() methods. These methods are used to create a new NavigationPage [20] and send the user to that page. When the user backs out of a page, it gets removed from the navigation stack and deleted.

Login

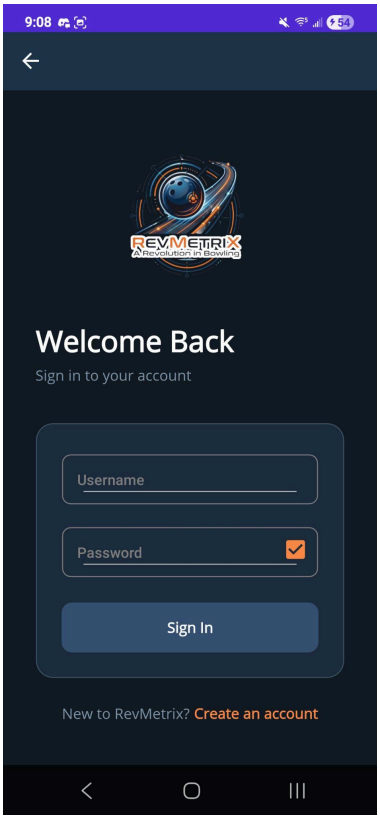


Figure 3.1.2.1: Login Page of the Mobile Application

The login page, as seen in Figure 3.1.2.1, allows the user to enter their credentials into XAML Entry boxes. Then they can submit these entries using the “Login” button which calls the OnLoginClicked() method. This queries the local SQLite database for the username and checks the password against the hashed password from the database using a verify method from BCrypt[34]. If the password doesn’t match the one in the database, or the username is invalid, a popup alert is displayed using the MAUI DisplayAlert()[18] function. Finally if the user clicks the link below the login button, this calls the OnRegisterTapped() method which sends the user to a new NavigationPage[33] of the registration page.

Registration

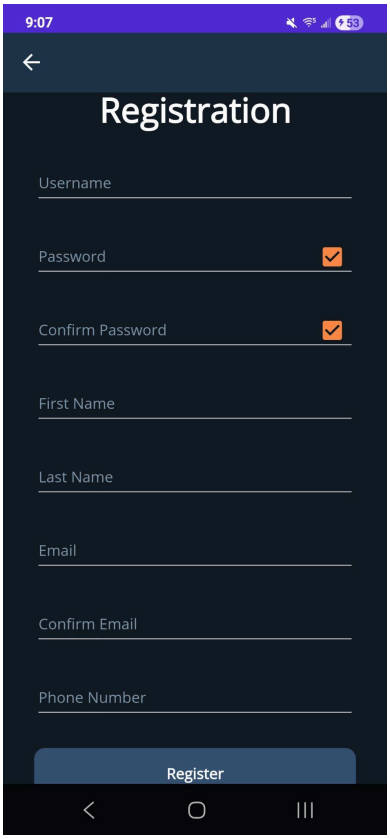
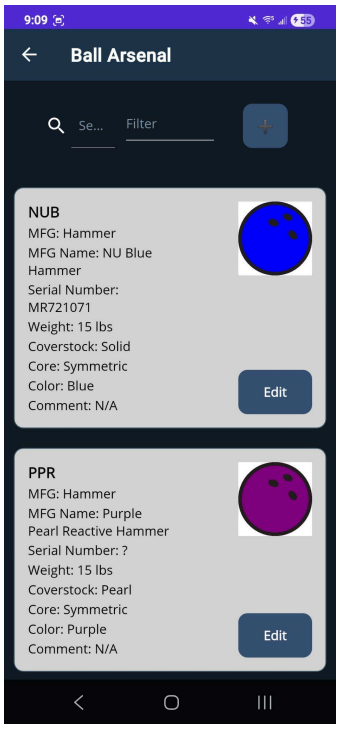


Figure 3.1.3.1: Registration Page of the Mobile Application

The registration page, as seen in 3.1.3.1, contains 8 XAML Entry[17] fields where the user can type in their information to create an account. This information as discussed earlier is username, password, confirm password, first name, last name, email, confirm email, and phone number. When the “Register” button at the bottom of the screen is clicked, it calls the OnRegisterClicked() method. This first checks if every field has been filled in before proceeding. Next, it queries the database to check if the username or email entered is already in use by another user. If the input information is new, then the password from the input field is hashed using the BCrypt.Hash() function and the user is added to the local database. If any input field is left empty, if the user or email already exists, or if a password or email doesn’t match, it will

display an alert using MAUI's DisplayAlert() function. Finally once an account is created, the user is sent back to the main page.



Ball Arsenal

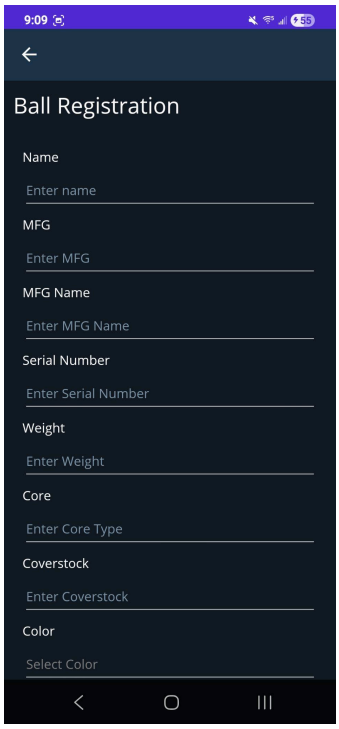


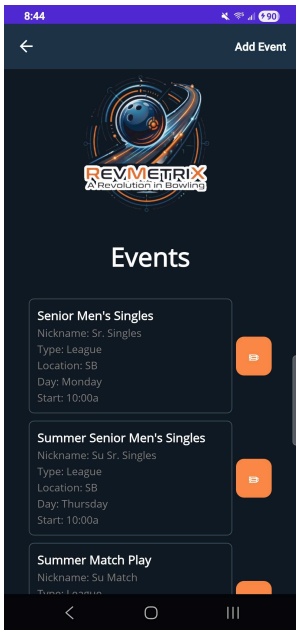
Figure 3.1.4.1: Ball Arsenal Page

Figure 3.1.4.2: Ball Registration Page

The Ball Arsenal page, as seen in Figure 3.1.4.1, displays all bowling balls that the user has registered in the application. Each ball is presented within a CollectionView that lists its information. At the top of the page, a search bar enables users to quickly find balls by name, while a filter menu allows sorting by name (A–Z or Z–A) or by registration date (newest to oldest, oldest to newest). A button on this page navigates to the Ball Registration page (Figure 3.1.4.2). This form includes nine XAML Entry fields: 'Ball Name', 'Ball MFG', 'MFG Name', 'Core Type', and 'comment' accept text input, while 'Serial Number' and 'Weight' are restricted to numeric input only, ensuring clean and validated data entry. The 'Color' picker contains nine

pre-set colors to choose from, as well as a ‘Custom’ option. Choosing this reveals an extra entry for a hex color value in the form ‘#FFFFFF’. Below the ‘Comment’ entry is a checkbox specifying if the ball is enabled or disabled.

Event Page



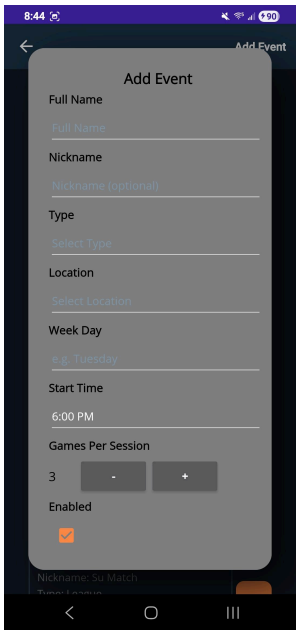
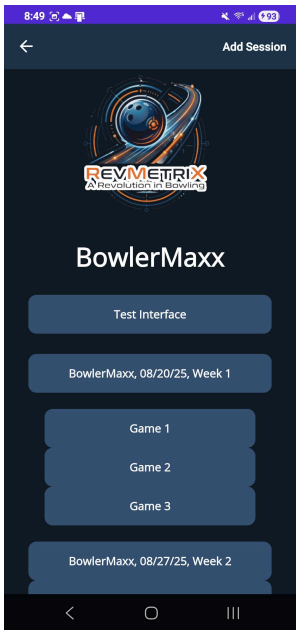


Figure 3.1.5.1: Event Page *Figure 3.1.5.2: Event Registration Page*

The Event page, as seen in Figure 3.1.5.1, displays all events entered by the user. Upon first login, the page will be empty except for a ‘Add Event’ button located in the top-right corner. Clicking this button opens the Event Registration popup(Figure 3.1.5.2). This popup includes 5 input fields, a time picker, two buttons for game number selection, and a checkbox: the Event Name, Nickname, and Week Day fields accept both text and numeric input, while the Event Type and Location fields utilize MAUI Picker elements. Event Type allows users to select from predefined categories, Practice, League, Tournament, or Anonymous, and Establishment lets users choose from their registered establishments. The time picker allows users to select a time from a clock wheel. Buttons labeled ‘-’ and ‘+’ are used to decrease and increase the game count,

and a checkbox is used to enable or disable an event. After completing the form, clicking the 'Add Event' button adds the new event to the main Event page. ‘Cancel’ can also be clicked to return without saving a new event. Each entry is displayed as a ListView item, showing the event’s name, nickname, type, location, day, and time. Next to each item is a button with a pencil icon. Clicking this button opens the same popup used for event registration, but with pre-populated data for the event that it corresponds to.

Session List



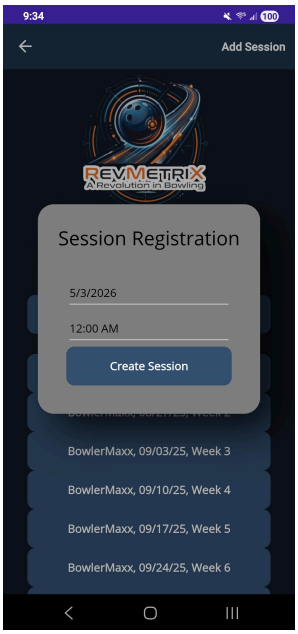


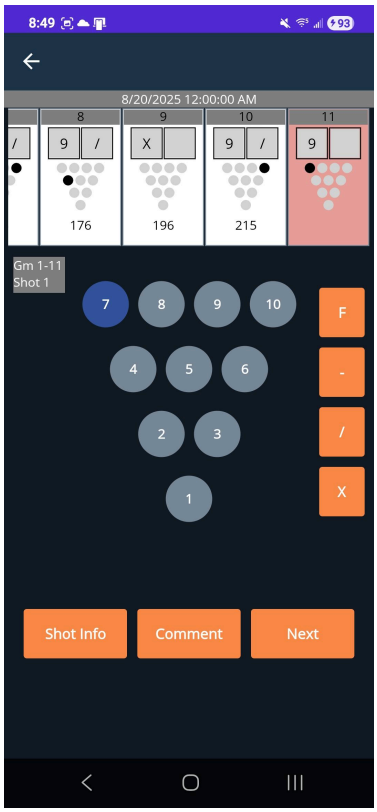
Figure 3.1.6.1: Session Page

Figure 3.1.6.2: Session Registration Popup

The Session page, as seen in Figure 3.1.6.1, displays all user-created sessions and their associated games for the event in a visual, interactive layout. When the application is first launched, no sessions or games are present. Users must initiate the process by clicking the 'Add Session' button in the top-right corner, which triggers the AddSession() function. This function

opens a session registration popup as seen in Figure 3.1.6.2. This popup contains a date and time picker. The popup also contains a button labeled ‘Create Session’ which saves the session and calls the LoadDataAsync() to populate the interface with the updated list of sessions and games. Sessions are displayed as a combination of the event nickname, session date, and week number (e.g., ‘BowlerMaxx, 8/20/25, Week 1’, ‘BowlerMaxx, 8/27/25, Week 2’, etc.), and clicking on a session reveals its associated games as individual buttons. Below the game list is an 'Add Game' button, which invokes the AddGame() function to create a new game with an incremented number. This button is only available for sessions that have not been completed. LoadDataAsync() is called again to reflect the changes. Selecting a game navigates the user to the Shot page, where frame and shot-level data can be recorded.

Shot Page



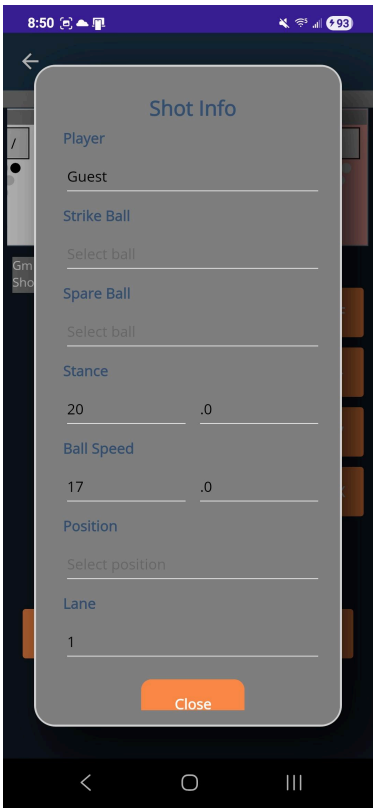
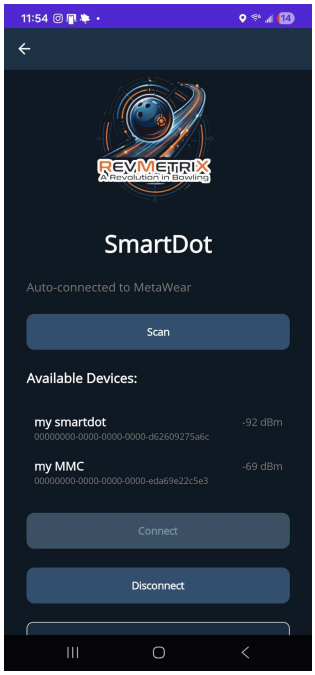


Figure 3.1.7.1: Shot Page of the Mobile Application *Figure 3.1.7.2: Shot Info Popup*

The Shot page, as seen in Figure 3.1.7.1, is one of the most complex pages of the mobile application. The page actively interacts with the GameInterfaceViewModel which keeps track of the local game variables and the frame view at the top of the page. The frame view is created using a MAUI CollectionView[22] which is a collection of items. Each of the frames is one item that contains 2 BoxView[23] elements for the shot boxes to display the number of pins knocked down on that shot (or strike, spare, gutter, and foul). All the text is binded to Label[24] elements. Each item also contains a mini representation of the pins which are just 10 BoxView elements with a corner radius of 5 to make them circular. The pin colors are set by the methods ApplyPinColors(), and ApplySecondShotColors(). Each frame and shot box contain GestureRecognizers so that if a previous frame is tapped, the user can edit that frame, select the first or second shot box, and resubmit the shots in case they input something wrong previously. The left side of the screen contains Label elements for the current game number, frame number, and shot number. The center of the screen contains 10 buttons which are each linked to a method called OnPinClicked() which toggles the pin's bit number between 0 (pin is down) and 1 (pin is up). It also changes the button's color to reflect the pin’s current state. The right side of the screen features 4 shortcut buttons for foul, gutter, spare, and strike. These are each linked to a method that sets the current frames shot boxes, changes the pinstates bits, and changes the pin buttons colors to reflect the pins that are either up or down. Finally, the bottom of the screen contains 3 more Button elements. The “Shot Info” button opens a popup for additional shot info. This popup, as seen in Figure 3.1.7.2, contains 9 Pickers[19] to select a player, strike ball, spare ball, stance, ball speed, position, and lane number. At the bottom of the popup is a button to return back to the shot page. The “Comment” button opens a popup for text input where the user

can either save, edit, or cancel changes to a comment for that shot. The “Next” button is linked to the method OnNextClicked(). This button is used to progress through a game of bowling. The user selects their pins left standing, clicks next, and the method calls upon other methods like SaveFrameAsync(), SaveShotAsync(), and UpdateScore() to update the UI accordingly.

SmartDot Page



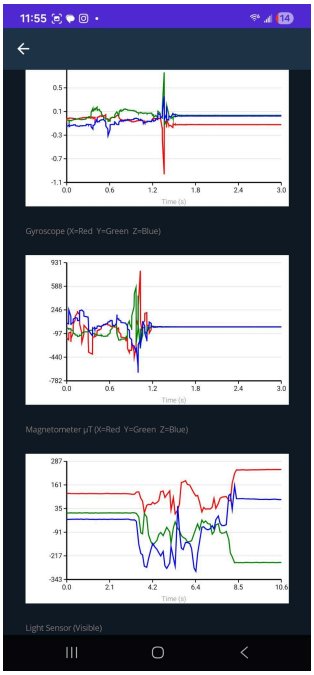


Figure 3.1.8.1: The SmartDot Page first half *Figure 3.1.8.2: Second half*

The SmartDot Page, as seen in Figure 3.1.8.1, enables the mobile application to connect to a Meta Motion S (MMS) or Meta Motion C (MMC) module and read data from its accelerometer, gyroscope, magnetometer, and light sensor. When the user opens this page, they can initiate device discovery by pressing the Scan button. This triggers the StartScanningAsync()

function, which first verifies that Bluetooth is enabled before calling StartScanningForDevicesAsync() from the BLE NuGet package. As devices are discovered, the scanner filters for MetaWear devices and displays all detected MMS modules along with their MAC addresses. To establish a connection, the user selects the desired device and clicks Connect, which triggers the OnConnectClicked() handler. This invokes _metaWearService.ConnectAsync() to establish the Bluetooth connection and _metaWearService.GetDeviceInfoAsync() to retrieve device metadata. Once connected, the interface exposes Start and Stop controls for each sensor type. When a Start button is pressed, the application calls the corresponding method:_metaWearService.StartAccelerometerAsync(), StartGyroscopeAsync(), StartMagnetometerAsync(), or StartLightSensorAsync(). Each of these functions transmits sensor-specific configuration bits to the MMS, instructing it to begin streaming the appropriate data to the phone. After data collection, the user can select Show Sensor Graph, which visualizes all captured readings on a graph for analysis.

Ciclopes

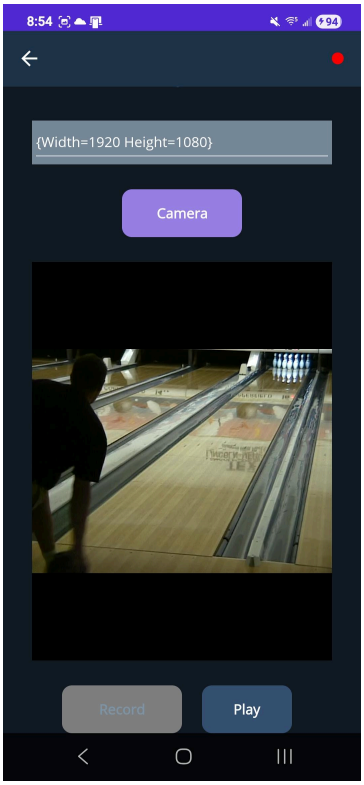


Figure 3.1.9.1: Ciclopes Page

The Ciclopes page, shown in Figure 3.1.9.1, utilizes the Microsoft Community Toolkit CameraView [26] to record video directly within the application. While CameraView is cross-platform, only the Android-specific controls have been implemented at this stage. The page includes a .NET MAUI Picker [19], which allows users to select from available camera resolutions. Beneath this control is a toggle button that switches between live camera mode and demo mode.

The main display area consists of a toolkit:CameraView, which renders the live camera feed, and a toolkit:MediaElement, which plays a prerecorded video when demo mode is enabled. Both elements are currently fixed at a width of 300 and a height of 400, and these dimensions are not user-adjustable.

At the bottom of the page, a “Record” button allows users to begin capturing video. When pressed, this button triggers the OnRecordClicked() method, sets the isRecording flag to true, and updates the button’s appearance by changing its color to red and its label to “Stop.” Pressing the button again stops the recording and prompts the user to select a save location within the device’s file system. After saving, a “Use Ciclopes” button appears, enabling the user to submit the recorded video for analysis.

For users who wish to incorporate SmartDot functionality, a red button is located in the top-right corner of the interface. Activating this button calls the mmsConnectionIconClicked() method, which opens a popup interface for device connection. Within this popup, users can select a previously saved SmartDot device. Upon selection, the TryAutoConnectToSavedDeviceAsync() method attempts to establish a connection automatically.

Once connected, the behavior of the video page changes. The system uses data streamed from the SmartDot, specifically accelerometer, magnetometer, gyroscope, and light sensor data, to detect key events such as ball release, ground impact, and pin contact. The application maintains a rolling three-second circular buffer of sensor data, which is preserved once a throw is detected. After the throw, an additional four seconds of data are recorded, at which point video capture is automatically stopped.

The moment the ball contacts the lane is identified and timestamped, and all associated sensor data are stored locally in JSON format. This data is then uploaded to the cloud alongside the recorded video for processing by the Ciclopes system. Once processing is complete, a “Ciclopes” button becomes available. Selecting this option displays the results generated by the machine learning model, including the ball trajectory and an analysis of the user’s bowling form.

Additional details can be found in the “Client-Side UI / Data Visualization” section of the Ciclopes documentation.

Establishment Page

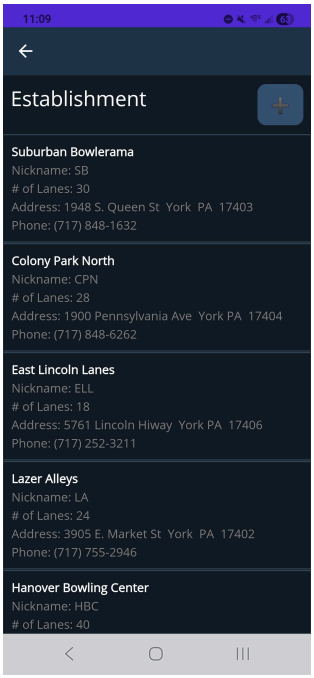


Figure 3.1.10.1: Establishment Page

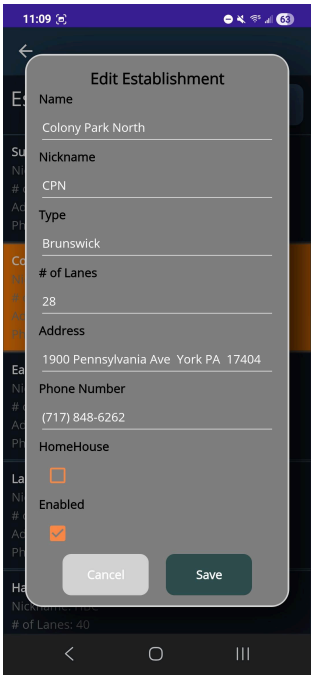


Figure 3.1.10.2: Establishment Registration Page

The Establishment page, as seen in Figure 3.1.10.1, displays all establishments entered by the user. Upon first login, the page will be empty except for an '+' button located in the top-right corner. When the user clicks this button, a popup is opened for Establishment Registration (Figure 3.1.10.2). This popup includes six input fields: Establishment Name, Nickname, Type, and Location accept both text and numeric input, while the Lanes field is restricted to numeric values. Two checkboxes are used to specify if the establishment is the user’s ‘HomeHouse’ and if it is enabled. After filling out the popup, clicking the 'Save' button adds the new establishment to the main Establishment page. The ‘Cancel’ button can be clicked to close the popup without saving. Each establishment entry appears as a ListView element, displaying the name, nickname,

number of lanes, address, and phone number of the establishment for easy reference. Each of these elements contain a GestureRecognizer which will open the registration popup with that event's information pre-populated for editing.

API Test Page

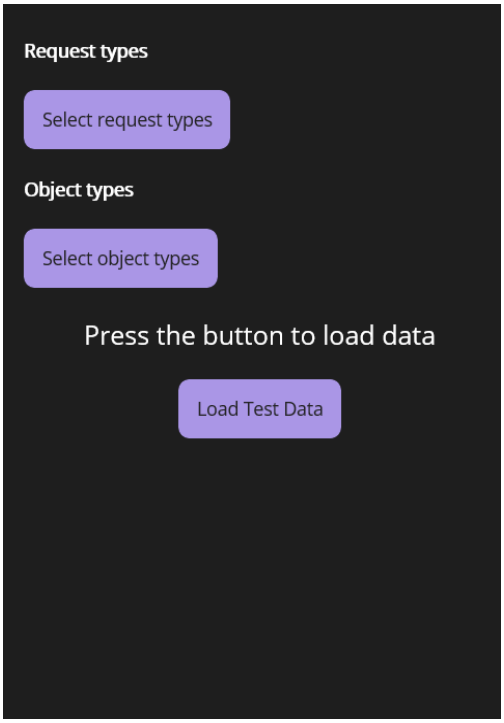


Figure 3.1.11.1: API Test Page

The API Test Page, as seen in Figure 3.1.11.1, contains two dropdowns containing a request type and data type as well as a field for data entry. These selectors are present for testing, and when the ‘Load Test Data’ button is pressed, the app will send a request to the Cloud Application based on what the user had selected in each of the dropdowns. The response is then displayed on the screen for the user. This page also contains a button ‘Sync Data’. When pressed, this button sends the contents of the local Cellular database to the Cloud Application, and

uploads any data that is not already present in the Cloud database. If successful, the server will then respond by saying how many tables were updated.

Account

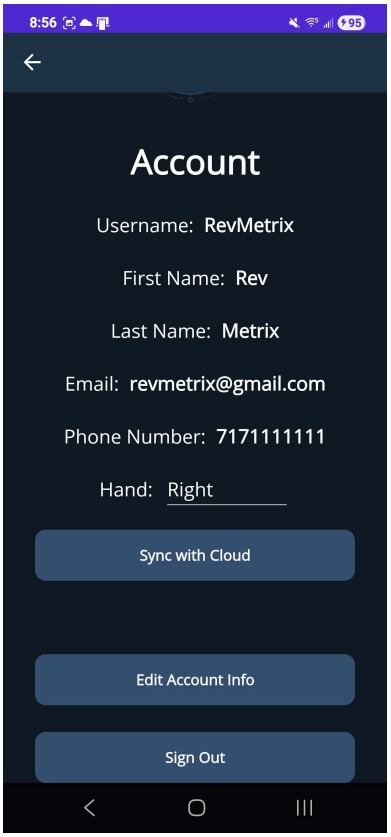


Figure 3.1.12.1: Account Page

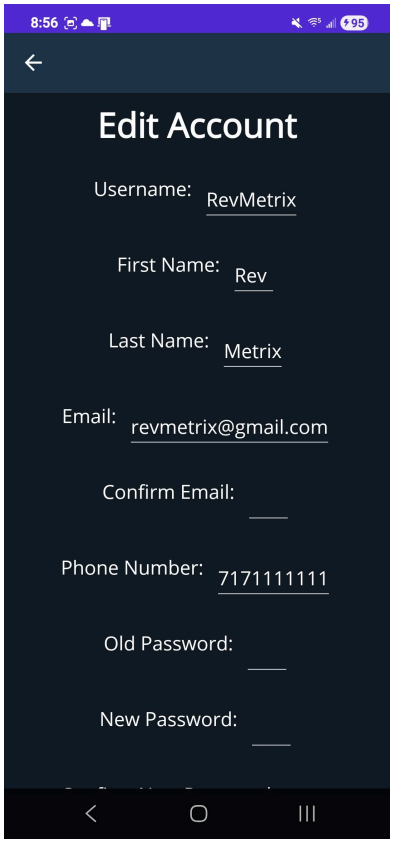


Figure 3.1.12.2: Edit Account Page

The account page, as seen in Figure 3.1.12.5, has 5 labels displaying the current user's username, first name, last name, email, and phone number. These are binded to the MainViewModel of the mobile application, which pulls that information from the database when logged in. Below these labels is a MAUI Picker[19] with the label “Hand”. This binds to the user's selection from the database for if they are right or left handed. If the user changes their selection, it automatically updates their profile in the database. Below that are 3 buttons. The first one is labeled “Get Stats” which calls OnStatsClicked(). This sends the user to the Stats page.

The “Edit Account Info” button is linked to an OnEditAccountClicked() method which sends the user to the edit account page (Figure 3.1.12.2). Finally, the “Sign Out” button calls the OnSignoutClicked() method which sets the preference for “IsLoggedIn” to false and redirects the user back to the main page.

The edit account page has 9 labels and 9 Entry fields. These correspond to username, first name, last name, email, confirm email, phone number, old password, new password, and confirm new password. When the page loads, it calls a LoadUserDetails() method to pull the user’s information from the database and display it in each entry field. Passwords are not displayed for security reasons. Each entry allows the user to edit and input new information for their account. To submit changes, the user can click the “Save Changes” button which calls OnSaveChangesClicked(). This grabs all of the current users information and compares it to the text currently in the entry fields. Next it will override the old information and show a popup using a MAUI DisplayAlert()[18]. This popup will either confirm that the account information was updated correctly, or that something is invalid.

Watch Page

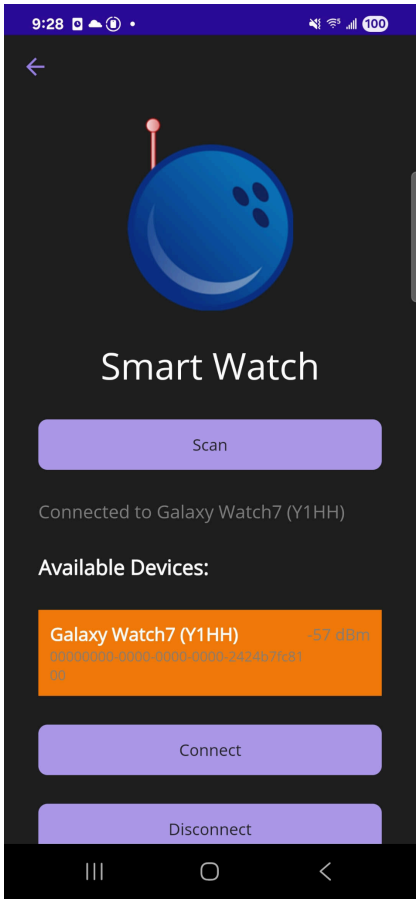


Figure 3.1.13.1: Smart Watch Page

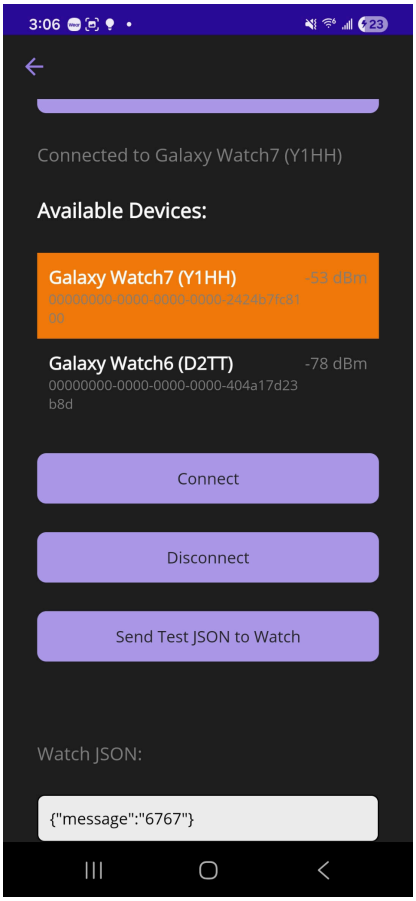


Figure 3.1.13.2: Smart Watch Page Ext.

The watch page, as seen in Figure 3.1.13.1, displays the list of available watches within the range that have the software running on them, that are advertising and ready to be connected to. When the user initially opens the page, they will be prompted with a “Scan” button, a “Connect” button and a “Disconnect" button. When the user clicks the “Scan” button, the list of available watches will be displayed and the user can select from the list. After selecting the correct watch, it will become highlighted orange and the user can select the “Connect” button. This will connect the phone to the selected watch, where the user can now send and receive data. In Figure 3.1.13.2, a “Send Json To Watch” button and a display box for the data is provided at the bottom of the page. This shows the data being sent from the watch to the phone for proof of

transmission. The “Send Json To Watch” button will send a piece of data to the watch. If the user does not want to have the watch connected anymore, they can select the “Disconnect” button which will remove the connection with that watch.

Stats

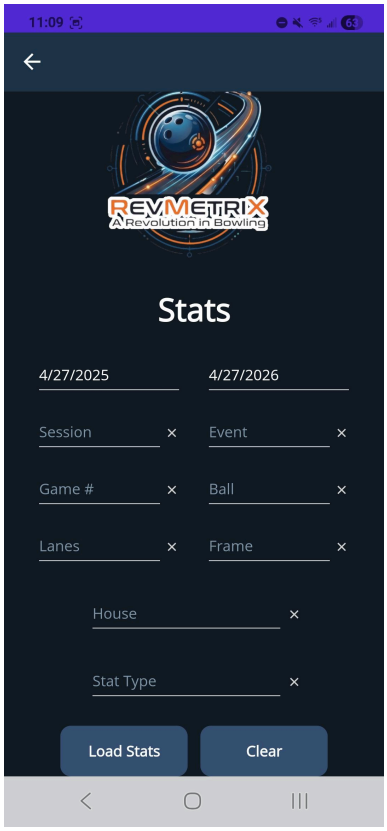


Figure 3.1.14.1: Stats Page Query Engine

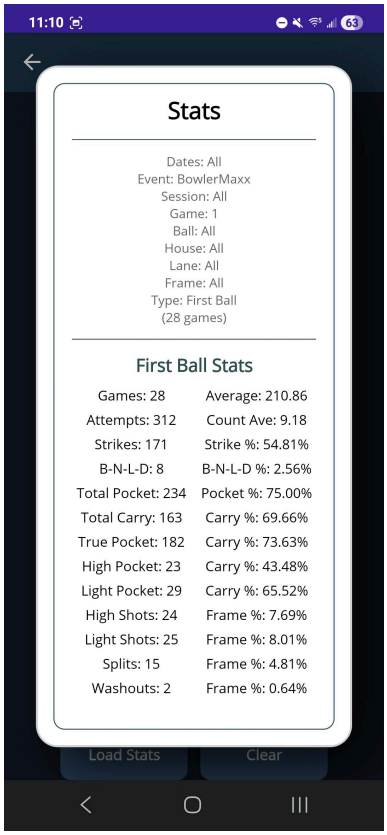


Figure 3.1.14.2: Stats Page Results

The stats page, as seen in Figure 3.1.14.1, has 10 Maui Picker[19] elements for selecting a startdate, enddate, session, event, game, ball, lanes, frame, house and stat type. These are used for creating custom queries to pull data from the database. The page interacts with the StatsViewModel to store the selected information and interact with the database. Below the pickers are 2 buttons. The “Load Stats” button submits a custom query based on the selections from the pickers. The ViewModel queries the database for all valid events, sessions, games,

frames, and shots based on the query criteria and calculates stats based on those shots. This button then triggers a popup to display the stats for the specific query as shown in Figure 3.1.14.2. The “Clear” button clears the user's selections so they can create new ones.

Watch

BLE Manager

The BLE Manager coordinates Bluetooth Low Energy (BLE) communication between the Flutter app and the paired phone. Built as a GetxController, it uses a MethodChannel ("ble_service_channel") to bridge Dart and Android native code. The BLE Manager initializes the GATT server by calling initGattServer(), which requests permissions, starts the Android foreground service, and sets up the GATT server with custom characteristics for writing and notifications. When a user records a shot, the BLE Manager serializes the Shot object, encodes it as bytes, and sends it to the phone using sendRawBLEPacket(). Each shot is transmitted individually as soon as it is recorded, rather than batching entire sessions. Incoming data from the phone, such as account packets, is received through the GATT server and routed to Dart via _handleNativeCalls(). The BLE Manager parses these packets with _tryParseAccountPacket(), updating session context and connection state for the rest of the app.

Session Controller

The Session Controller is the main state manager for the watch app, implemented as a ChangeNotifier singleton. It manages the session lifecycle, shot recording, and data synchronization with the phone. Key methods include createNewSessionFromPacket(), which initializes a new session using metadata from the phone, and recordShot(), which adds a new

shot to the current frame and triggers BLE transmission. The controller tracks the current session, active game and frame, and manages per-game/frame progress. Editing a shot is handled by editShot(), which updates the relevant shot in place. Default values for lane, ball, and stance are stored as properties, and test data generation is available for development. Every shot recorded in the UI is immediately serialized and sent to the phone, ensuring real-time data consistency.

Account Packet

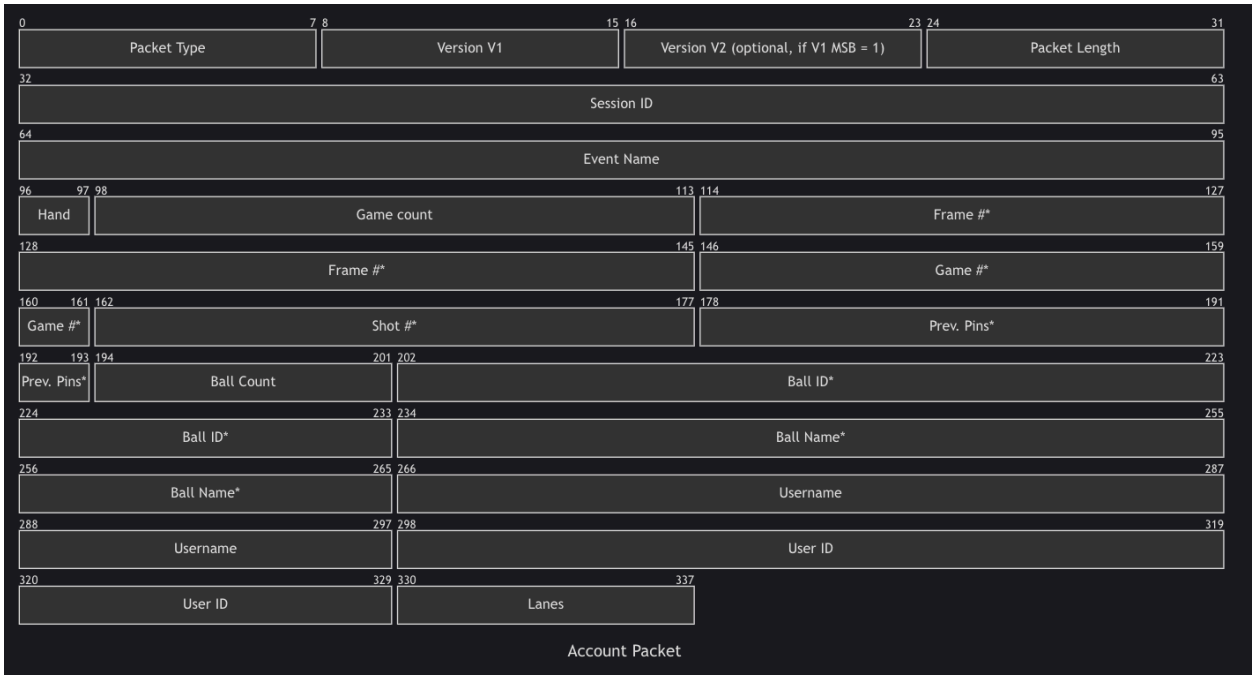


Figure 3.2.5.0: Account Packet Diagram

The Account Packet is a binary-encoded message received from the phone, parsed by the BLE Manager and used by the Session Controller to set up the session. The static method AccountPacket.fromBinary() extracts fields shown in Figure 3.5.2.0 like sessionId, eventName, primaryHand, gameCount, and available balls, as well as an array of GameState objects for multi-game sessions. When received, the Session Controller uses this data to create the session

and configure the UI. The Account Packet protocol ensures the watch is always in sync with the phone's session metadata, supporting seamless resuming of a session on connection to the watch. The asterisks in the Figure above show fields that have variable byte definitions, meaning they can increase or decrease based on the amount of data connected to the session or account.

Session Model

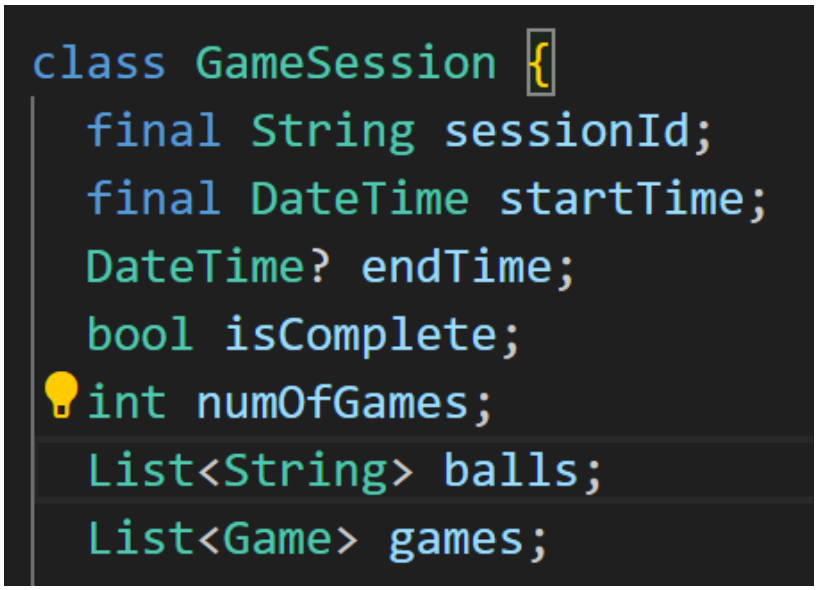


Figure 3.2.5.1: Session Model Definition

The Session Model encapsulates all data for an active session, including the session ID, start and end times, completion status, number of games, available balls, and a list of Game objects as seen in the definition in Figure 3.5.2.1. It provides methods for serialization (toJson, fromJson), enabling sessions to be saved locally. The model exposes helpers to find the active game, mark the session as complete, and update session metadata. It supports dynamic session structures, allowing for variable numbers of games and balls. The Session Model is designed for extensibility, supporting future features like session statistics, user profiles, and cloud

synchronization. It also integrates with the local cache to provide offline persistence and recovery.

Game Model

```
class Game {
  final int gameNumber;
  int score;
  int startingLane;
  List<int> lanes;
  List<Frame> frames;
```

Figure 3.2.6.1: Game Model Definition

The Game Model is responsible for constructing the game object which is the second highest level which is owned by the session class and contains frame objects. Its definition can be seen in Figure 3.2.6.1. The gameNumber field is for organizational purposes when converting data in and out of binary format from the Account Packet. The score field is for the calculation of the games score and will be updated per shot. The startingLane defines what lane the user will start on. The list of lanes contains all possible lanes the user can be playing on. Most of the time this will be two lanes and the user will alternate between. The list of frames will contain up to 12 frame objects. The game model also contains helper functions such as newGame which generates an empty game object. The Game Model integrates seamlessly with the Session Model and Frame Model, providing a consistent data structure for the entire session.

Frame Model

```
class Frame {
  final int frameNumber;
  int lane;
  final List<Shot> shots;
```

Figure 3.2.7.1: Frame Model Definition

The Frame Model is responsible for constructing a frame object which is the second lowest level which is owned by the game class and contains shot objects. Its definition can be seen in Figure 3.2.7.1. The frameNumber field is for organizational purposes when converting data in and out of binary format from the Account Packet. The lane field is the identifier that determines what lane that frame will be completed on. It enforces bowling rules by supporting up to two shots for frames 1–9, while the 10th frame can dynamically allow an 11th and even a 12th frame depending on the outcomes of each shot, specifically, if a strike or spare is achieved, bonus frames are appended to accommodate the extra shots awarded by standard bowling rules. The models isComplete property determines frame completion by evaluating the number of shots and the pinfall logic, handling all edge cases for strikes, spares, and bonus attempts. The totalPinsDown getter aggregates pinfall across all shots in the frame. For persistence and BLE communication, the Frame Model implements toJson() and fromJson() methods, ensuring the frame's state can be serialized and reconstructed. Integrated with the Game Model, which manages the sequence of frames, the Frame Model ensures that all frame-specific logic has accurate session management.

Shot Model

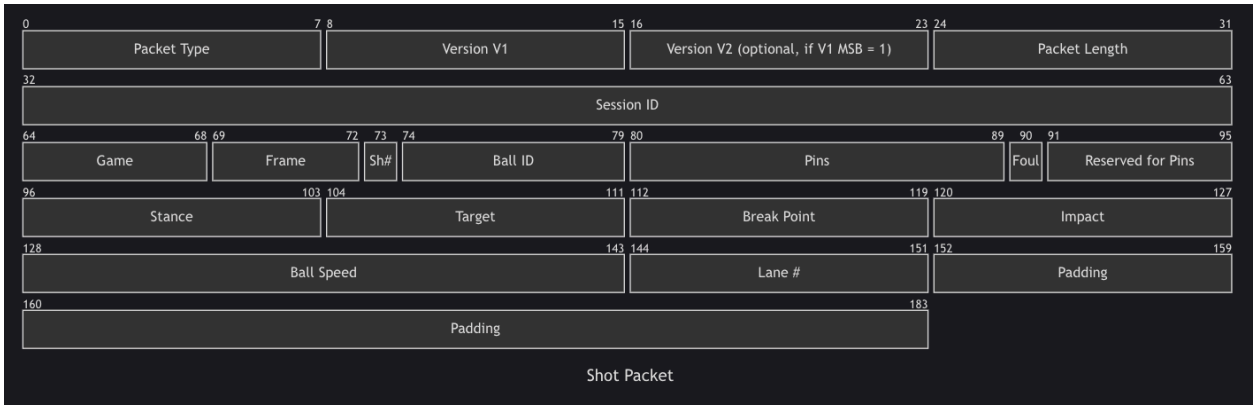


Figure 3.2.7.1: Shot Packet Diagram

The Shot Model is implemented as a class that encapsulates all the data and logic for a single bowling shot. Each Shot instance records the shot number, ball ID, number of pins knocked down, a bitmask encoding the pin state and foul status, impact (board), stance, target, break point, speed, frame number, lane, and a read-only flag. The model provides a constructor that allows for flexible initialization, including default values and support for both impact and board terminology. It features a static foulBit constant for bitmask operations and a mapping from impact descriptions to board numbers. The model includes computed properties such as pinsStanding, isFoul, and pinsState, which decode the bitmask to provide high-level information about the shot outcome. Utility methods like buildPins and buildLeaveType construct the bitmask from a boolean pin array and foul flag. The Shot Model supports serialization and deserialization via toJson and fromJson, ensuring compatibility with local storage. For BLE, it implements encodeToBinary, which packs all shot data into a 23-byte binary format for transmission, and decodeFromBinary, which reconstructs a Shot and its context from a received packet. The encodeToBinary method in the Shot model serializes a shot into a 23-byte array for BLE transmission. It packs the shot's fields such as session ID, game/frame/shot/ball numbers,

pins bitmask, stance, target, break point, impact, speed, and lane into specific byte offsets using bitwise operations. The decodeFromBinary method reverses this process, it parses a 23-byte array, extracting each field by reading and unpacking the corresponding bytes. It reconstructs the shot's properties including session context by applying bitwise masks, shifts, and scaling, returning a Shot object and its associated metadata. The packet makeup can be seen above in Figure 3.2.8.1.

Settings Page

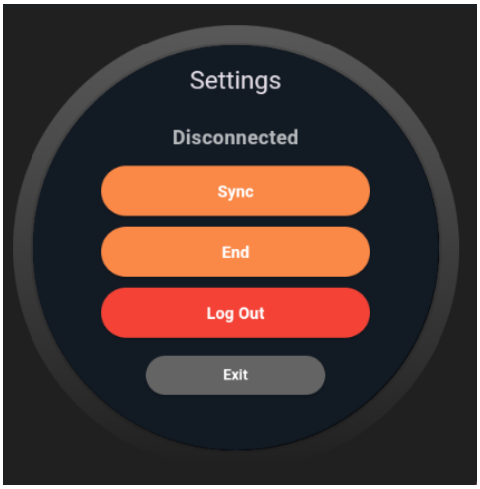


Figure 3.2.9.1: Settings Page

The Settings Page is a stateful widget providing advanced controls for BLE and session management, primarily for development and testing. It uses GetX's Obx to display real-time BLE connection status and device address, and provides buttons for session synchronization, session ending, and logout. Each action is guarded with local state flags to prevent duplicate operations, and all BLE commands are dispatched via the BLE manager. The session synchronization and ending logic includes asynchronous BLE command dispatch, error handling, and navigation to the Sessions Page upon completion. The logout logic disconnects BLE, clears

the session, and navigates back to the home page, ensuring a clean application state. As you can see in Figure 3.2.9.1, the page's UI is constructed with a column layout, using custom-styled buttons and conditional rendering based on BLE and session state. In Figure 3.2.9.2, you can see how each button interacts with the system.

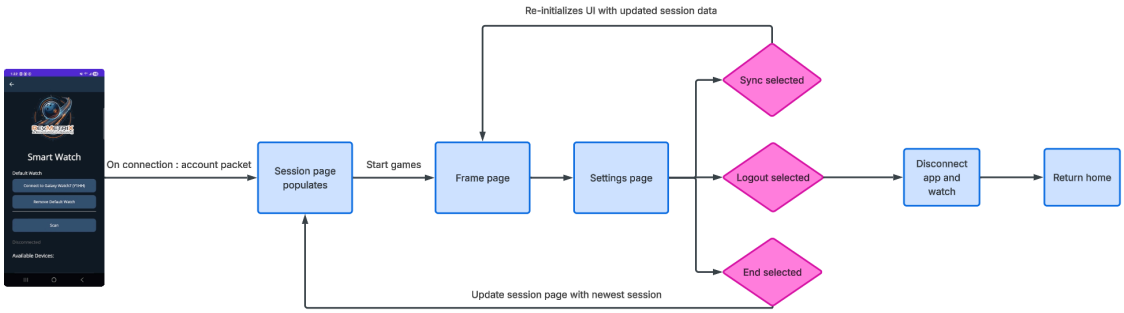


Figure 3.2.9.2: Settings Page Flowchart

Shot Page

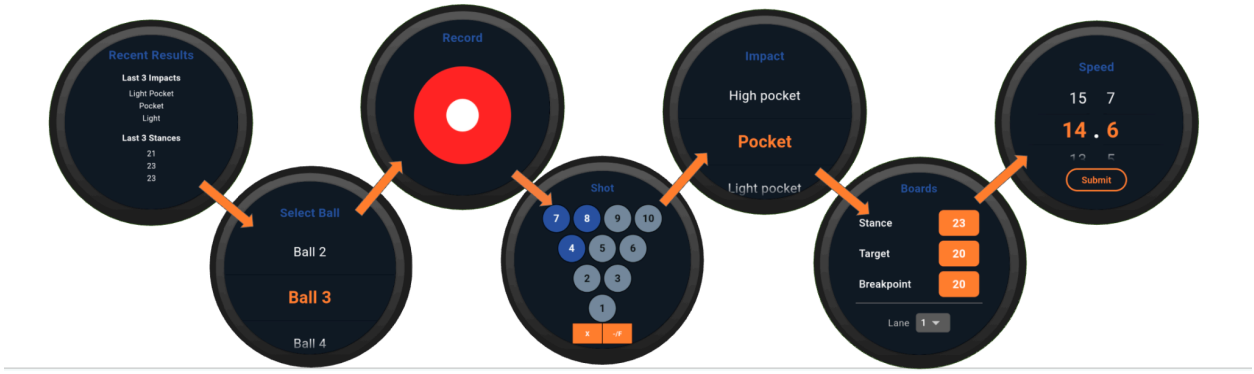


Figure 3.2.9.2: 7 Stage Shot Workflow

The Shot Input Page implements a structured, seven-stage workflow for capturing all relevant data for a bowling shot, using an interface managed by a PageController. Each stage corresponds to a distinct aspect of shot entry, as defined by the titles list: (1) Recent Results, (2) Select Ball, (3) Record, (4) Shot, (5) Impact, (6) Boards, and (7) Speed. The workflow, shown in Figure 3.2.8.2, is realized as a sequence of interactive pages, each with its own state and validation logic. The Recent Results stage displays a summary of the last three shots, leveraging helper methods to extract and format impact and stance labels from recent shot data. The Select Ball stage allows the user to choose from available balls, with logic to ensure the selected ball is valid and falls back to a default if necessary. The Record stage manages the actual shot recording process. The Shot stage provides a visual interface for selecting which pins remain standing, using a custom pin display widget, outcome toggling (strike, spare, foul), and foul handling, with pin toggling logic that respects the context of the shot (first or second shot in the frame).. The Impact and Boards stages use custom wheel pickers and dialogs to allow precise selection of impact board and stance, with dynamic label computation and recent value recall. The Speed stage employs a scrollable picker for ball speed, with snapping and real-time updates. Throughout the workflow, state is tightly managed and synchronized, with each stage initializing from the provided context and updating the shot model as the user progresses. This stage-based design ensures user-friendly shot data entry, supporting both manual and post-shot workflows and integrating seamlessly with the broader session and BLE transmission logic.

Frame Page

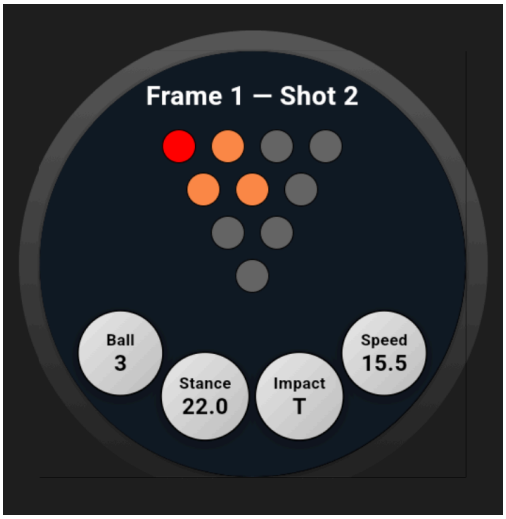


Figure 3.2.11.1: Frame Page

Figure 3.2.11.2: Frame Page Navigation

The Frame Page is architected around the FrameShell stateful widget, which orchestrates frame navigation, selection, and shot entry within a game context. It leverages the SessionController for global state synchronization, using a ListenableBuilder to trigger UI rebuilds on session changes. The page maintains local state for the currently viewed frame, selection mode, and manual navigation flags, enabling both real-time and retrospective frame inspection. Gesture detection is implemented for vertical swipes (to transition to the Game Page) and long-press (to activate frame selection mode) as seen in Figure 3.2.11.2, with haptic feedback for enhanced UX. The core UI is a stack containing the BowlingFrame widget, which itself manages a PageController for horizontal paging through shots within the frame. The shot slot logic is dynamically computed based on frame index and game state, with special handling for the 10th and 11th frames to accommodate strike/spare rules. The overlay for frame selection is conditionally rendered, and the page ensures that navigation, shot input, and state transitions are robustly synchronized with the session model, including clamping indices and handling edge

cases for out-of-bounds navigation. The Info Bubbles at the bottom as seen in Figure 3.2.11.1 are used as indicators for the bowlers of their stats for the shot that they just threw. These can be accessed by swiping left to the previous shot.

Game Page

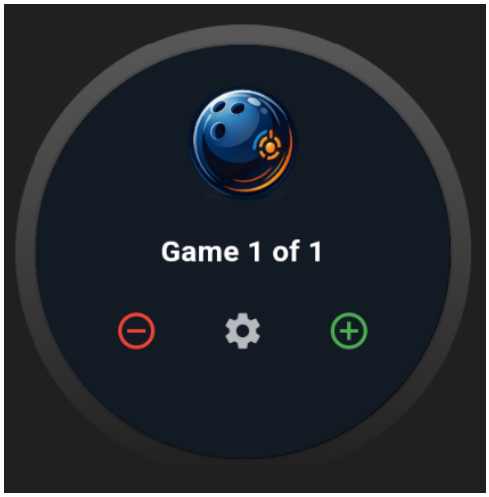


Figure 3.2.12.1: Game Page

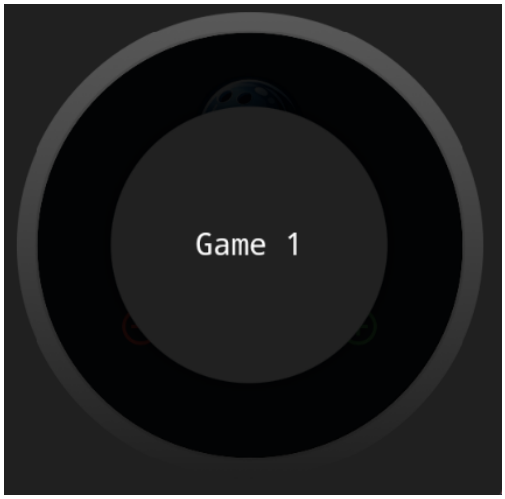


Figure 3.2.12.2: Game Page Navigation

The game page, as shown in Figure 3.2.12.1, is implemented as a GameShell stateful widget, encapsulating the logic for game selection, navigation, and dynamic game management within a session. It subscribes to the SessionController for reactive updates, ensuring the active game index is always valid relative to the current session state. The page supports gesture-based navigation, with vertical swipes mapped to transitions between the Game and Frame pages, and long-press to enter a carousel game selection overlay. The BowlingGame widget displays the current game's metadata and provides controls for incrementing or decrementing the number of games in the session, as well as accessing settings. The user can swipe up on this page to access the frame page.

Home Page

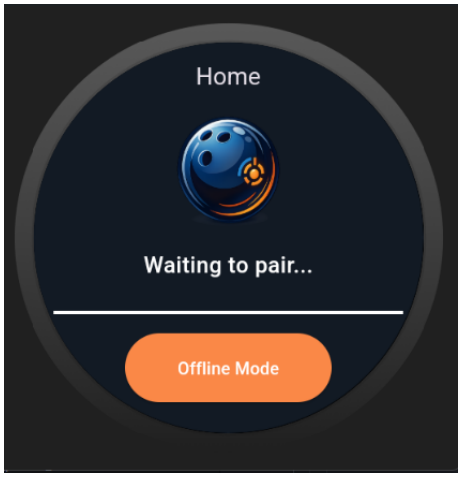


Figure 3.2.13.1: Home Page

The Home Page, as shown in Figure 3.2.13.1, acts as the application's entry point, orchestrating BLE connection management, session initialization, and navigation. It uses GetX's ever function to subscribe to BLE connection and account packet state, automatically navigating to the Sessions Page upon successful connection or packet reception. The page initializes the SessionController with session context from the account packet, supporting seamless handoff from BLE to session state. The UI includes branding, connection status, and an offline mode button, which initializes an anonymous session and navigates to the Frame Page for testing or disconnected workflows.

Sessions Page

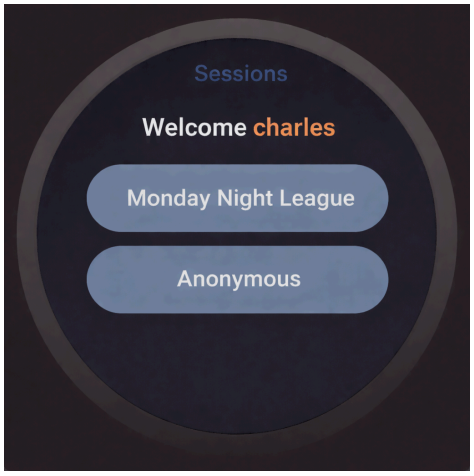


Figure 3.2.14.1: Sessions Page

The Sessions Page, as shown in Figure 3.2.14.1, is a stateful widget that serves as the entry point for session initialization, tightly integrated with BLE-driven workflows via GetX's reactive Obx widgets. It displays dynamic user and event information extracted from the latest BLE account packet, and provides two primary actions: starting an event session using account packet data or an anonymous session. The event session logic decodes bitmask-encoded pin states and initializes the SessionController with full session context, including session ID, game/frame/shot numbers, ball inventory, and game state. The anonymous session logic initializes the session with a default session ID value, supporting both continuing an active session from the phone or starting a new one from the watch.

Cloud

API

```
7
8   [ApiController]
9   [Tags("Gets")]
10  [Route( template: "api/gets/[controller]")]
    Emmet Larson *
11  public class GetAllPiDiagnosticScriptBySession : AbstractFeaturedController
12  {
13      [HttpGet(Name = "GetAllPiDiagnosticScriptBySession")]
14      [ProducesResponseType(typeof(List<PiDiagnosticScript>),  statusCode: StatusCodes.Status200OK)]
15      [ProducesResponseType( statusCode: StatusCodes.Status400BadRequest)]
        Emmet Larson *
16      public async Task<IActionResult> RetrieveAllPiShotsBySession(int sessionId)
17      {
18          var diagnosticScripts :List<PiDiagnosticScript>  = await ServerState.UserStore.GetAllPiDiagnosticScriptsBySession(sessionId);
19          return diagnosticScripts == null
20              ? Problem( detail: "unable to retrieve diagnostic scripts from the database") :
21              Ok(diagnosticScripts);
22      }
23
24  }
```

Figure 3.3.1.1: Code in the Cloud Application for a GET API endpoint

Figure 3.3.1.1 showcases how API endpoints are structured in the Cloud Application. The software here is using the C# .NET CORE server tools to setup the API endpoints. This library comes with all the necessary tools for creating API endpoints, adding request authentication, and handling asynchronous requests. The above endpoint is an example of a GET endpoint, which is retrieving data from our database and returning it to the user. The API does this by calling one of our database controller methods, and passes in a variable called ‘sessionId’ that was parsed from the request headers.

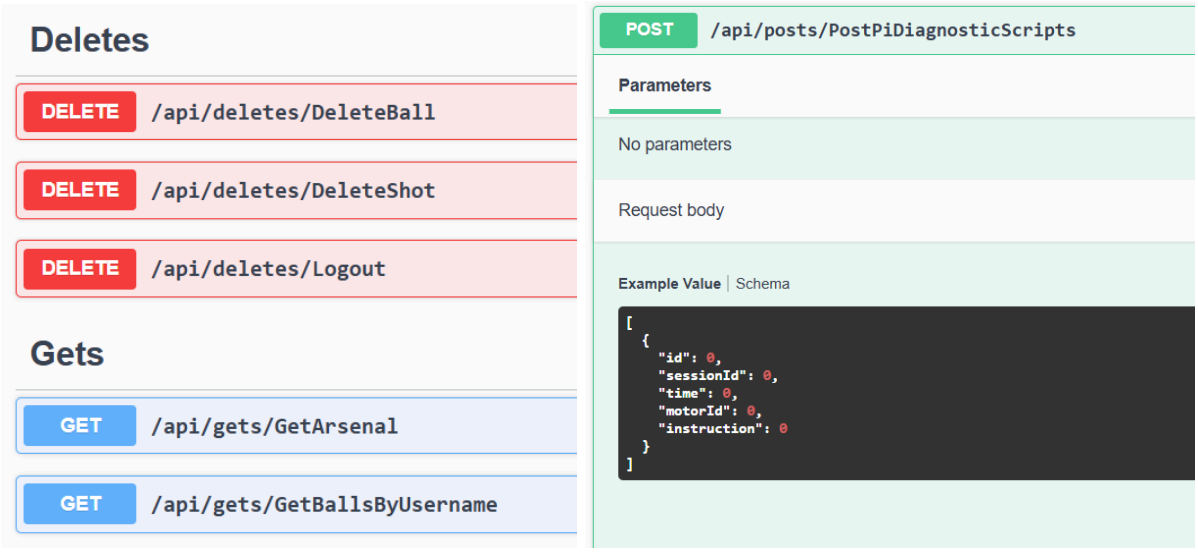


Figure 3.3.1.2: Swagger Documentation for Cloud Application

Figure 3.3.1.2 shows two screenshots from the Swagger Documentation of the Cloud Application's API. These documents are automatically generated at runtime, and give users the ability to see all of the request types and header structure in order to make a request. When one of the entries is clicked on, a panel opens below that shows a user what shape the request body needs to look like to submit a proper request. Below this, the documents also show a user what sort of responses the server can send back. This includes showing the user what shape data will return in once a successful request has been made.

Database Controllers

After a request has been made to the API, the system then calls the appropriate Database Controller in order to interface the request data with the database. At this layer, the system begins by validating the data set passed in from the request. Each controller may have slightly different implementations of data validation, but each controller at the very least verifies that every field of every object passed in the request contains at least some value, even if it's arbitrary. For example, if the user wants to create a ball, the request to the cloud server needs to

contain all of the fields present in the ball table or else the server will send back an error message. The server then plugs the data into a prepared statement for the corresponding table, and then sequentially attempts to insert each object from the request into the database. Our database controllers use transaction queries, so at any point in the process if an error occurs the database will roll-back any changes and the hot data will not corrupt. Once the data has been inserted, the controller then retrieves all of the primary keys for each entry and formats them into a response body. The API then takes this response body and returns it to the user. As mentioned previously, if any errors occur the database is rolled back and an appropriate error message is passed back to the user.

Entity Framework Core

Entity Framework Core is a Microsoft backed C# .NET server architecture used to build the Cloud Application. It provides all the tools necessary to create our API, Database Controllers, and connect to our Microsoft SQL database. There are other, more specific relevant tools that this framework provides including Table Classes, Database Application Context, and Migrations. All of these tools support the developer's ability to better manage the application’s database.

Application Context

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    base.OnModelCreating(modelBuilder);
    modelBuilder.Entity<BallTable>().ToTable( name: "Balls", schema:"combinedDB");
    modelBuilder.Entity<UserTable>().ToTable( name: "Users", schema:"combinedDB");
    modelBuilder.Entity<FrameTable>().ToTable( name: "Frames", schema:"combinedDB");
    modelBuilder.Entity<EventTable>().ToTable( name: "Events", schema:"combinedDB");
```

Figure 3.3.3.1.1: Example of Application Context

The Application Context is the core component of the new Database Migration system that was implemented. The Application Context defines an environment for the Database Migration system to construct an SQL script file from. It does this by collecting predefined table files which contain all the columns desired for the table and mapping them to a specific table and schema name in the database. The above Figure 3.3.3.1.1 outlines the process for how these table files are defined and added to the database under a specific schema.

Tables

```
public class BallTable
{
    [Key]
    public int Id { get; set; }

    [ForeignKey( name: "UserId")]
    public int UserId { get; set; }

    [MaxLength(50)]
    [Required]
    public string Name { get; set; }
```

Figure 3.3.3.2.1: DB Table Object

Here, Figure 3.3.3.2.1 represents what the table objects look like in the backend. When implementing a new table, all that is necessary is creating a file similar to in the figure, and defining all the desired properties. These files allow for just as much customization with tables as would be possible creating them with raw SQL. A developer can define primary and foreign keys, specify data type and length, and also specify whether a field is required when creating.

Migrations

Using both the Application Context and Table classes, the Entity Framework library provides an API for generating an SQL script file based on the contents of the Application Context in the project. This tool is especially useful, as Liquibase allows us to generate and regenerate our database using SQL script files. For the system, we have a command that generates the SQL script file after the system detects that changes have been made and then creates a file in our Liquibase SQL scripts folder that contains all of the changes to the database. As with the Database Controllers, these changes are also individually transactional (i.e. each change to the database occurs in a separate transaction) with rollback procedures that are generated along with the SQL script. This means we have no risk of corrupting our database when creating new changes. These migrations are also versioned as well, and the tools provided by the Entity Framework allow for the switching between individual database versions.

Ciclopes

Segmentation Model Performance

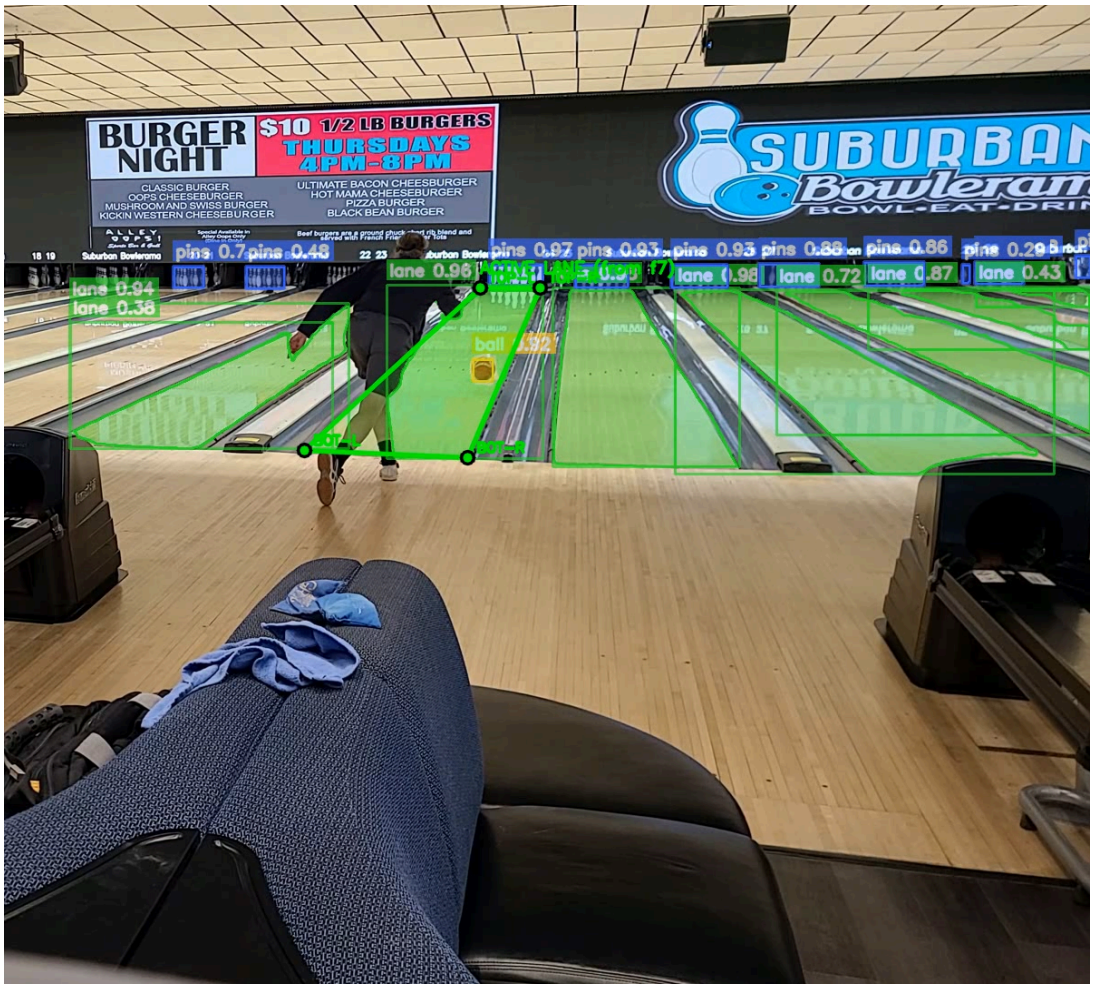


Figure 3.4.1.1: Segmentation and Lane Fit Visualization

In figure 3.4.1.1 the segmentation performance and lane fitting algorithm performance can be seen. The image is from a real world test video that was collected. The green segmentation overlays can be seen that the segmentation models perform well with the scene's noise of shadows and reflections properly giving a good candidate for corner fitting. The 4 labeled corner points on the active lane can also be seen as properly detecting the lane corners. Also, the model was trained on and properly detects the pin area in the background of the image,

this can be used if available to fit the end line of the lane as they should be in contact. The segmentation model which is able to be run on device due to its nano size performs extremely well, and is sufficient for real world use.

Postprocessing Algorithm

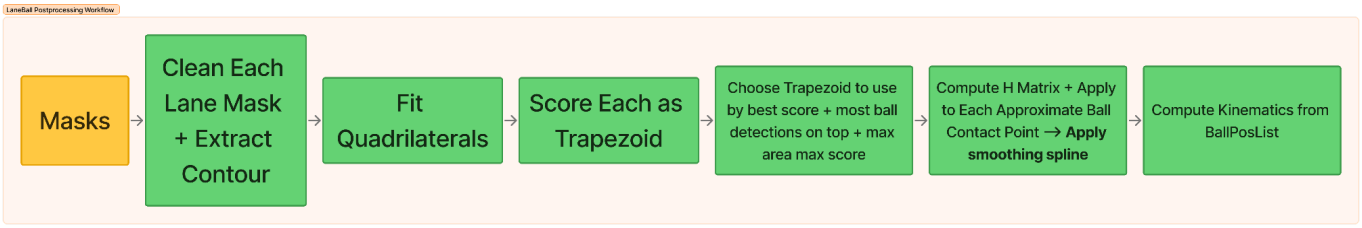


Figure 3.4.2.1: Postprocessing Workflow for Ball Trajectory Generation

For the ball trajectory producing flow seen in figure 3.4.2.1, the list of produced segmentation masks from each frame is walked through and using a scoring system to find the lane corner points to compute the homography transformation matrix. The detected lane with the highest score is used to compute the homography matrix, and this matrix is then used per frame to produce the ball positions per frame in the video by using the calculated transformation on the approximate contact point, the bottom point of the detected ball. These ball positions have a smoothing spline applied to them to interpolate points and produce a smoother trajectory, and this smoothed list of positions is used alongside the known frames per second of the video as the difference in time to calculate kinematics of the ball to give to the user. The pose estimation pipeline is much simpler and runs the batched video through the model and applies EMA smoothing to the skeleton coordinate points to reduce jitter for rendering client side. The results are either aggregated or kept separately for the result return from the route they were prompted from.

Lane Detection Algorithm

The lane detection algorithm was implemented due to the fact that segmentations of lanes may be imperfect, as well as due to multiple lanes being in the images. This forces us to both fit an equilateral to the segmentations, as well as use a scoring algorithm to find the in use lane. In terms of the “trapezoid fitting” as we refer to the process as, the initial segmentation of a lane may have holes in the center therefore this is filled in producing a shape which then based on expected geometry has a trapezoidal shape fit around the segmentation. This allows imperfect segmentations to properly find approximate lane corner positions, with this the better the segmentation the more accurate the corner point detections. The trapezoid fitting is done per frame, and each is compared against each other for their aspect ratios and shapes, finding the fit lane that fits the expected lane geometry best. Once the best lane is found based on a scoring system based on geometry, we must make sure this is the actively used lane by expecting that the in use lane will have the most detected balls on top of it through the video. Through testing on real world video we confirm that the active lane is accurately selected, and even with poor segmentation an approximate lane is fit with enough accuracy for the homography to be usable, the tradeoff is that downstream the trajectory and resulting calculations will have injected noise. The more accurate the segmentation, the more accurate the resulting trajectory.

Truncation Algorithm

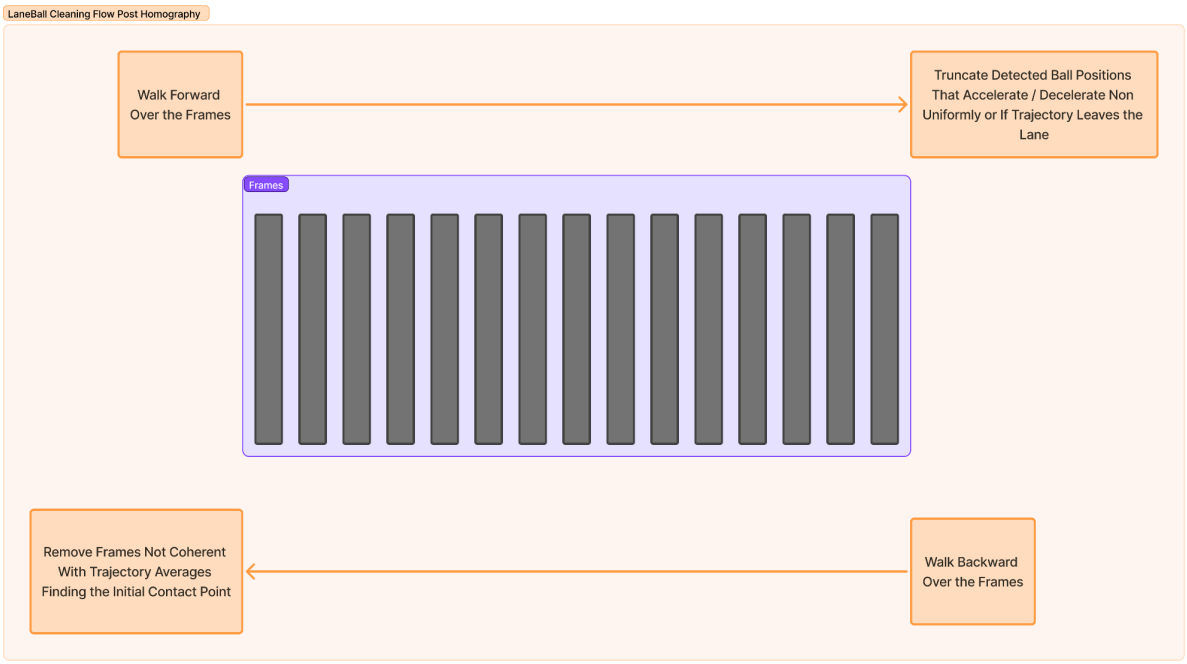


Figure 3.4.2.2: Truncation Algorithm Walk Visualization

To review, the algorithm itself does not strictly detect when the ball hits the lane, or when the ball hits the pins. We are able to use the data collected from the SmartDot to detect the actual timestep when the ball hits the lane and the frame this happens at can be interrelated from the frames per second of the collected video. In regimes when the SmartDot is not available we wanted the algorithm to be able to detect these points to not poison the trajectory due to plane differences, or including bad detections after the ball hits the pins. During our testing, when plotting the produced ball positions during the entire video, the uniform change in the ball position frame to frame distinctly changes when walking forward through the frames and the ball hits the pins, and when walking backward and we find the points when the ball is still in the air. This allows us to create a walking forward and backward algorithm to then truncate poor portions of the trajectory leaving the properly detected positions. After smoothing, the trajectory

is stretched to the end of the lane, and we are then able to calculate the full trajectory of the ball. This algorithm uses the moving expected difference of position in the x and y directions to find extreme anomalies, this is allowed due to the ball not being able to make distinct changes in direction or speed after the ball is thrown. This same principle and analysis can also be used to detect anomalies such as the ball going off of the lane, or bad detections which can then be truncated as well.

Trajectory Generation

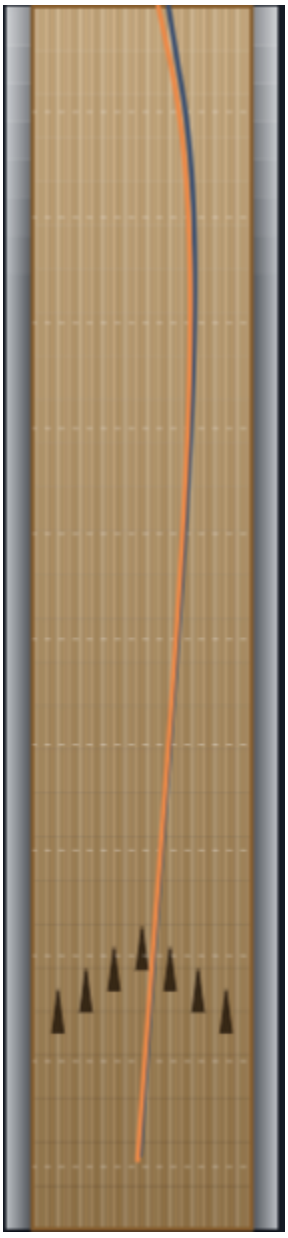


Figure 3.4.3.1: Rendered Trajectory

In figure 3.4.3.1 we show an overlay of rendered trajectories. In our previous report from Fall 2025 we were able to give error based statistics showing the accuracy from our algorithm, due to lack of real world reference data, we can only visually analyze accuracies. We can confirm that the trajectories in figure 3.4.3.1 follows the trajectories of the ball in the videos,

however without real world collected reference data with ground truth we cannot provide specific accuracy metrics.

Pose Estimation

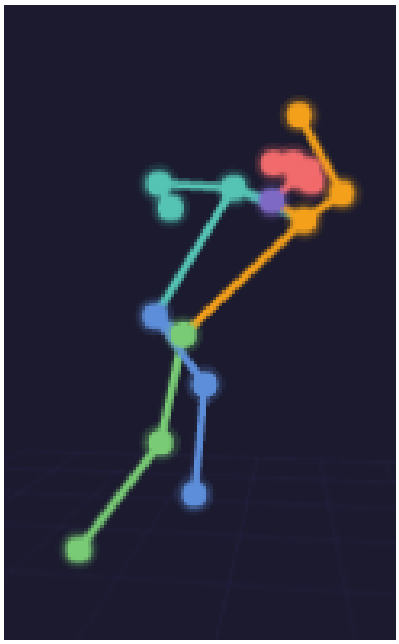


Figure 3.4.4.1: Estimated Pose

Figure 3.4.4.2: Reference Image

Figure 3.4.4.1 shows the estimated 3D pose from our reference image from a test video shown in figure 3.4.4.2. Exact accuracy statistics cannot be provided for the pose estimation performance, however specific portions of the pose such as shoulder, elbow, wrist, and hip accuracies matter much more for the end use case of biomechanical analysis than other joint positions, and we can visually confirm the accuracy of these joints. In this nature, only the right side of the body is shown in the image, and this portion of the estimated pose is the most accurate over multiple collected videos with this camera setup. Confirming that SAM3D Body can provide meaningful data for bowlers to analyze their mechanics and make meaningful improvements to their game from the provided visualizations.

Client Side UI / Data Visualization

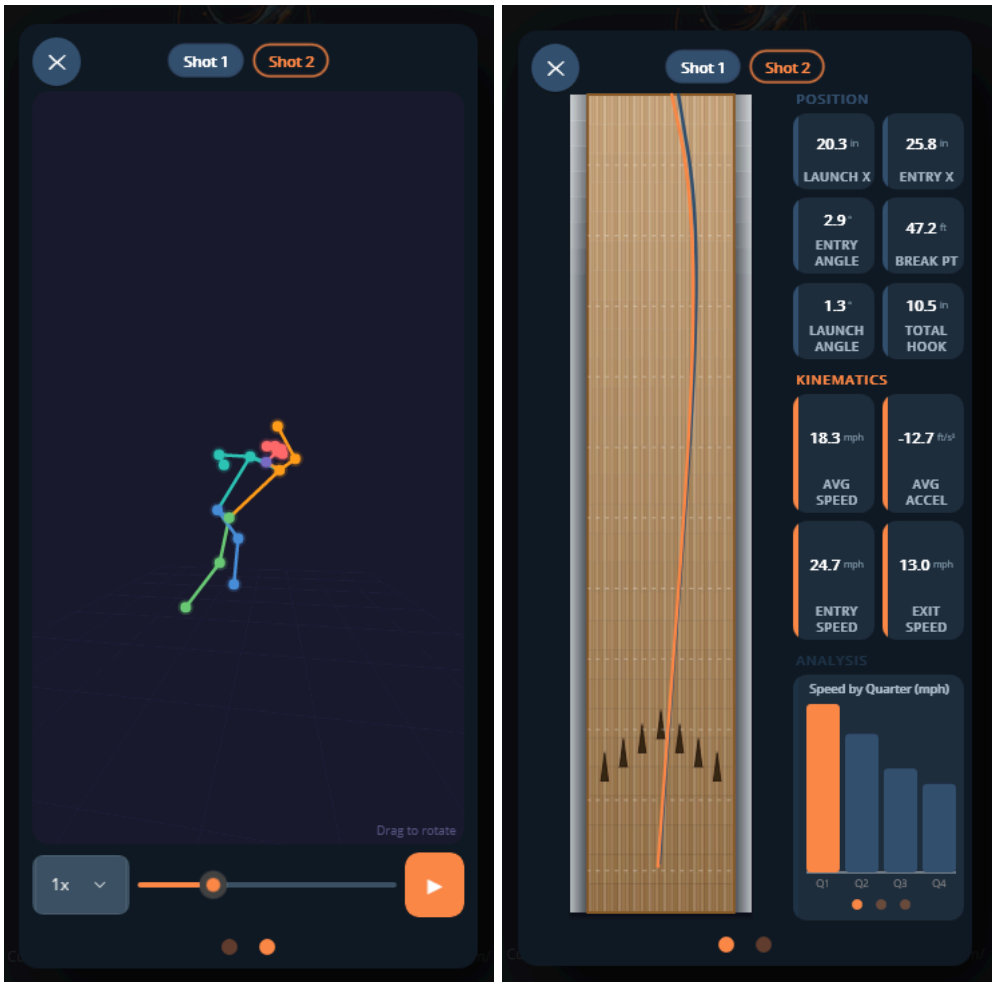


Figure 3.4.5.1: Data Visualization UI Components

The UI screenshots as seen in figure 3.4.5.1 show the visualizations provided to the end user for their analysis. The data collected from the algorithms and AI models are only as useful as the visualizations provided to the user. Our goal was to provide kinematics stats, as well as important position information from the trajectories. Also we provide the ability to overlay multiple shot trajectories for direct comparison while allowing easy toggling with the buttons at the top of the screen to look at the specific statistics and pose video of a specific shot.

Ball Spinner Controller

Ball Spinner Controller (BSC) Class

The BSC class, short for Ball Spinner Controller, is the core class. It is instantiated on application run and it handles creating the SmartDotConnectionManager singleton instance, the CloudAPI singleton instance, the Session object, DataController object, and motor objects. The BSC is created and can be accessed anywhere throughout the application, and as such, this is the main method of sharing data to our other classes.

Hardware Implementation

The hardware implementation consists of assembly, wiring, and validating the electrical subsystems that make up the Ball Spinner system. This includes mounting the motors, configuring the motor drivers, establishing the power distribution of the system and developing the Python-based control scripts that command and monitor the system. Removable adapters and connectors were added to support hardware swaps and all wiring has been sized according to current requirements.

Motors

The primary E3665 BLDC motor and the secondary axis NEMA 17 stepper motor were installed on the system frame with alignment and thought of accessibility. The BLDC motor was connected to the Flipsky Mini FSESC 4.20 through its three phase wires and Hall-sensor harness. To facilitate testing with alternative motors or replacement units, JST adapters and screw-terminal blocks were added between the motor leads and ESC wiring, eliminating the need for resoldering during component changes. Wiring for the BLDC phases used 18 AWG insulated conductors, appropriate for the current draw.

The BLDC motor software is written in Python using the VESC library, which sends control packets over USB to the ESC. These packets specify parameters such as target duty cycle, commanded current, or target RPM, depending on the selected control mode. Upon startup, the software performs an arming sequence holding the ESC at a defined neutral PWM equivalent state to ensure safe initialization before motor activation. After arming, the library allows the duty cycle commands to be transmitted to the ESC. The script also reads real time motor data including electrical RPM, current and temperature, which is used to validate correct motor behavior during hardware testing.

The secondary NEMA 17 stepper motor was electrically connected to the DM542 driver through its two-phase windings (A+, A–, B+, B–). Unlike the BLDC motor, the stepper does not rely on commutation feedback. Instead, the motor is actuated by precisely timed current pulses generated by the DM542 in response to STEP and DIR commands from the Raspberry Pi. The stepper wiring used 20–22 AWG, given the lower per-phase current relative to the BLDC motor.

Motor Drivers

The Flipsky Mini FSESC 4.20 was connected to the 24 V rail using 18 AWG wiring and interfaced with the Raspberry Pi over USB. Electrical configurations including motor current limits, voltage limits and hall sensor mapping were performed in the VESC software tool. During operations, the ESC generates high frequency PWM switching internally to modulate the motor's phase currents according to the commanded duty cycle. The software library communicates with the ESC at high speed and enables real time updates to duty cycle commands while monitoring motor data.

The DM542 stepper driver was powered with 18 AWG wire and configured via DIP switches to match the NEMA 17 motor's phase current requirement of 1.5–1.7 A and to operate

at 1/16 microstepping. The Raspberry Pi interfaces with the driver through STEP and DIR lines, where the frequency of STEP pulses determines motor speed and the polarity of DIR sets rotation direction. The driver internally regulates the current to each phase using bipolar control, ensuring that the magnetic field advances the rotor in controlled increments. A removable connector interface was added so that the driver could be reconfigured or replaced without modifying GPIO wiring.

Power Distribution

The power system is centered around the 24 V P1-150-24 supply, which provides regulated power to both motor drivers. Power was routed through a terminal block using 16 AWG wire for high current paths and 18 AWG for moderate current components. A buck converter stepper the 24 V to 5 V in order to supply the Raspberry Pi. A common electrical ground was established across the ESC, DM542 driver, and Raspberry Pi to ensure accurate signal referencing for STEP/DIR pulses and USB communication.

SmartDot

The SmartDot module, or in our current stage of development, the MetaMotionS, Figure 1.1.2.4, connects through bluetooth. The engineers at MetaWear created a python SDK [42]. This SDK only works with 32-bit ARM operating systems. However, we had chosen to develop our Graphical User Interface (GUI) with PyQt6 as this was a powerful choice for a Raspberry Pi 5 (RPi5) that ran optimally on 64-bit ARM architecture. The team had already begun development with the GUI once the dependency was discovered. In an effort to save refactoring and fully utilizing the technical capabilities of the RPi5, getting MetaWear to work on the 64-bit ARM OS was required. Inside of the Ball Spinner Controller repository, the instructions to compile the source code are provided [43]. This involves downloading five different project's source codes

and manually configuring them for 64-bit ARM instruction set architecture, compiling them in order of dependency, and setting linking variables. This method of compiling Metawear and its subdependencies from source results in an outcome equivalent to using `pip install metawear` in the terminal of the 32-bit OS.

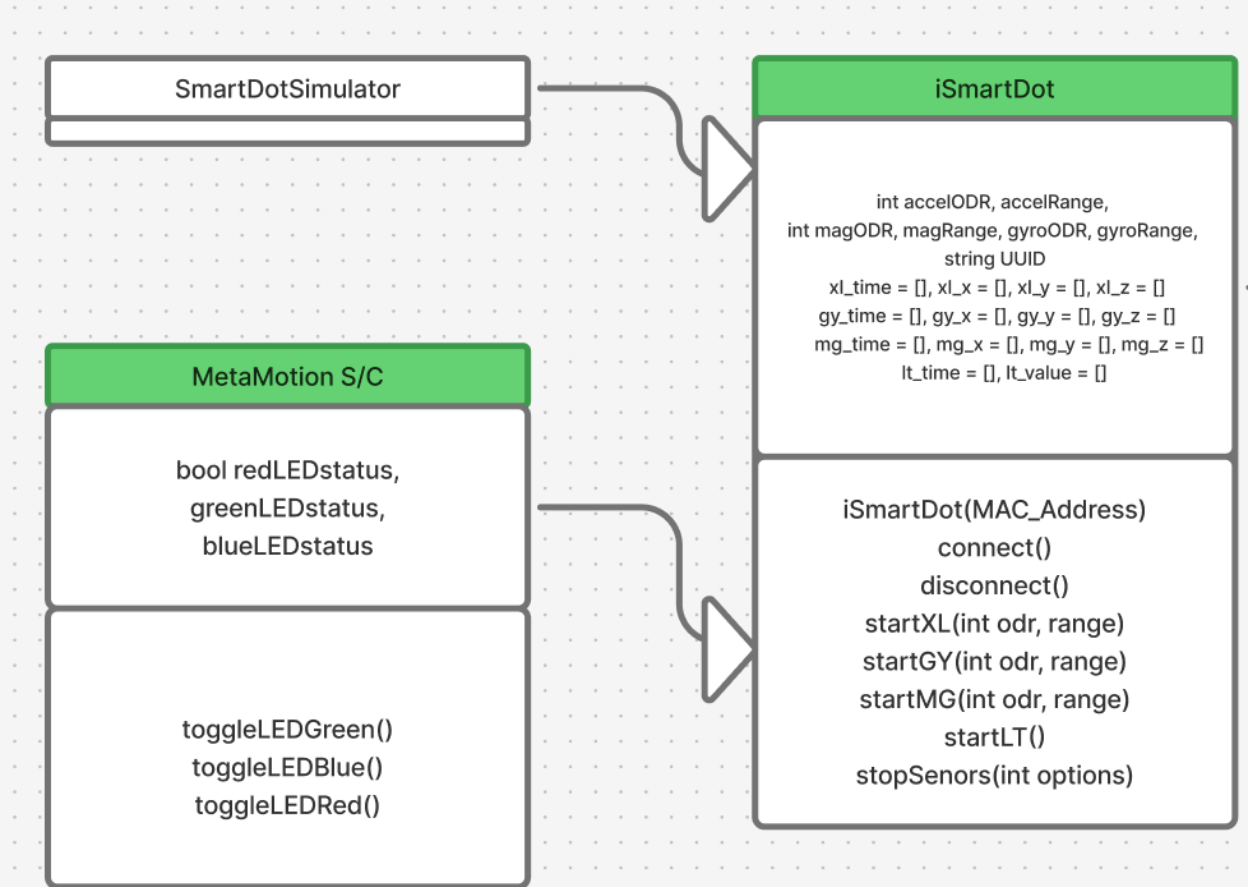


Figure 3.5.4.1: UML Diagram of SmartDot Design

Once a connection is established with the SmartDot, a MetaMotionS object is created, which implements the iSmartDot interface; see Figure 3.5.4.1 for the UML design of these interfaces and their implementations. The MetaMotionS class takes a MAC address as a constructor parameter. After construction, the connect function of the MetaMotionS class is

invoked. This function creates a MetaWear device using the installed MetaWear-Python-SDK, and toggles a blue LED once a connection is established.

After connection, the SmartDot object is stored in the SmartDotConnectionManager, which allows application pages to access it for data retrieval and display. Data handlers are configured to use callback functions that write live data to the accelerometer, gyroscope, magnetometer, and light sensor data arrays, which are then passed to the SmartDot data graphs.

Additionally, iSmartDot is implemented by the SimulatedSmartDot class. This class enables continued system operation when a physical SmartDot connection is unavailable or when development is performed off the Raspberry Pi 5. During such conditions, the user is assigned a SimulatedSmartDot instance instead of a MetaMotionS module, allowing full use and testing of the system's remaining components.

Models

Models exist for each type of data, SmartDotData, EncoderData, DiagnosticScriptData, ShotScriptData, and HeatData. These models were created as DataInstances. In addition a second class was created as a list controller of each of the data instance models. This controller allowed for the ability to add and modify the existing list of data for each data type.

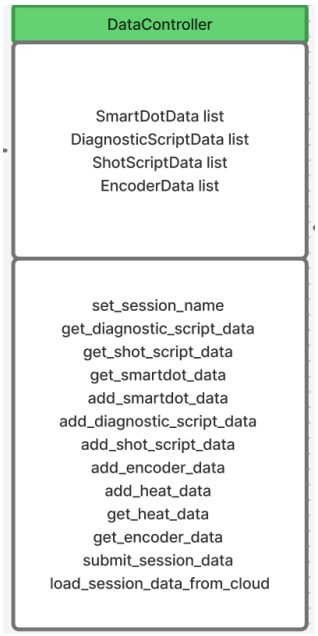


Figure 3.5.5.1: DataController UML Model

The DataController class is used to centralize data collection, see Figure 3.5.5.1 to view the DataController's methods and parameters. The DataController and Session classes both have new instances created when Shot Mode or Diagnostic Mode is entered. The DataController class provides the application with the ability to access the controllers for each and every data type that is being tracked. It also provides the ability to interact with the Cloud with functions to submit and load data to/from the Cloud.

In the current implementation, the Accelerometer, Magnetometer, Gyroscopic, and Light data are not sampled at the same intervals but the SmartDot's data model assumes they are. A temporary solution was implemented. A field was added to the SmartDot Data table called data_selector. The value of data_selector shows which value in the SmartDotDataInstance is actually valid.

```
for data in smartdot_data.data_entries:
    if data.data_selector == 0:  # Accelerometer
        time_accel.append(data.time)
        accel_x.append(data.accelerometer_x)
        accel_y.append(data.accelerometer_y)
        accel_z.append(data.accelerometer_z)
    elif data.data_selector == 1:  # Gyroscope
        time_gyro.append(data.time)
        gyro_x.append(data.gyroscope_x)
        gyro_y.append(data.gyroscope_y)
        gyro_z.append(data.gyroscope_z)
    elif data.data_selector == 2:  # Magnetometer
        time_mag.append(data.time)
        mag_x.append(data.magnetometer_x)
        mag_y.append(data.magnetometer_y)
        mag_z.append(data.magnetometer_z)
    elif data.data_selector == 3:  # Light
        time_light.append(data.time)
        light.append(data.light)
```

Figure 3.5.5.2 Code of the Data Selector Being Used To Choose Data

In Figure 3.5.5.2, one can see how the data_selector is used to assign data. The other values are simply filled with '-1'. In the future work section, new tables will be described that remove this overuse of empty fields.

Cloud

The connection with the Cloud was implemented with three classes. The APIUtils, CloudAPI, and DataController, see Figure 3.5.5.4 for the UML of the APIUtils and CloudAPI. The APIUtils class simply exists to abstract out the error handling and logging for API Requests. It handles the HTTP POST and GET requests that the CloudAPI class then calls with specific URLs and JSON data to establish communications. The CloudAPI class handles GET and POST requests for all of our database schema. It also handles a special GET by querying the Session table by time range, providing a request body inside of the GET request to obtain specific DateTime related information from the Session table.

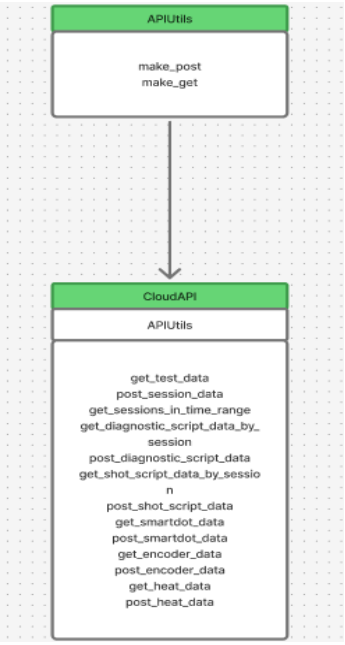


Figure 3.5.5.4 Cloud API and API Utils as defined in the UML

The DataController has two functions that interact with the Cloud, submit Session data and load Session data. These functions make use of all of the individual data controllers that the DataController object has as objects, and either calls their respective CloudAPI post function, or calls their respective get function by Session ID. The load all session data from Cloud will populate all of the data controllers with the received data from the Cloud which allows the data to then be accessed through the global DataController as necessary.

UI Implementation

BSCMainWindow

The BSCMainWindow is an extension of the QMainWindow class from PyQT6. This window loads ‘BSCMainWindow.ui’ to create the ui elements. Contained within ‘BSCMainWindow.ui’ is the top menu for navigation, an emergency stop button, and a stacked widget containing all of the pages within the app. All of the pages contained within

BSCMainWindow must contain a changePage() pyqtsignal. This signal contains both the index of the page they are trying to navigate to as well as any additional data that would need to be sent along with it. These pyqtsignals are connected to the switch_to_page() method in the BSCMainWindow. This method takes the index of the page as well as the data and changes the stacked widget to show that index. In addition a switch case is used on that index to change the title of the window and to determine if any other code needs to run on the page switch. For example, when the page is switched to the shot view page , its StartShotView() method is called. In addition to that pyqtsignal, any page that uses the motor also calls a navigationLock pyqtsignal. This signal prevents the page from being changed while the motor is in use. The list of indexes are as follows:

- 0 - FrontPage
- 1 - DiagnosticModePage
- 2 - ShotModePage
- 3 - AnalysisModePage
- 4 - Cloud Test
- 5 - Unused
- 6 - ShotViewPage
- 7 - DataViewPage

Figure 3.5.6.1.1: BSCMainWindow Navigation and E-Stop

The user is able to navigate back to the home page, to the cloud test page, or out of the application using the navigation menu shown in Figure 3.5.6.1.1. The E-Stop Button, as shown

in Figure 3.5.6.1.1 calls the motors disconnect function. Using the Simulation menu shown in 3.5.6.1.1 you can Enable and disable the physical motors.

Diagnostic Mode Page

The Diagnostic Mode Page is an extension of the Qwidget Class. This page loads the ‘DiagnosticModePage.ui’ file to load its UI. The page is structured with sliders and buttons to control the three motors, 3 pyqtgraphs that use a circular buffer for their data, labels to display the motors current values, and a SmartDot Viewer on the right side, as shown in Figure 3.5.6.2.1. Before the motors can be driven the user must press the connect motor button. Upon pressing the connect motors button the text changes to Disconnect motors, the Start Diagnostic button is unlocked, and the navigation is locked. Then the motor control timer is enabled allowing the 3 motors to be driven. The motor control timer is a QTimer that triggers every 30 milliseconds as defined in the BSC class. No data is collected until the start diagnostic button is pressed. When the start diagnostic button is pressed data collection begins, the text on the button is changed to

Stop Diagnostic, and the diagnostic session is created.

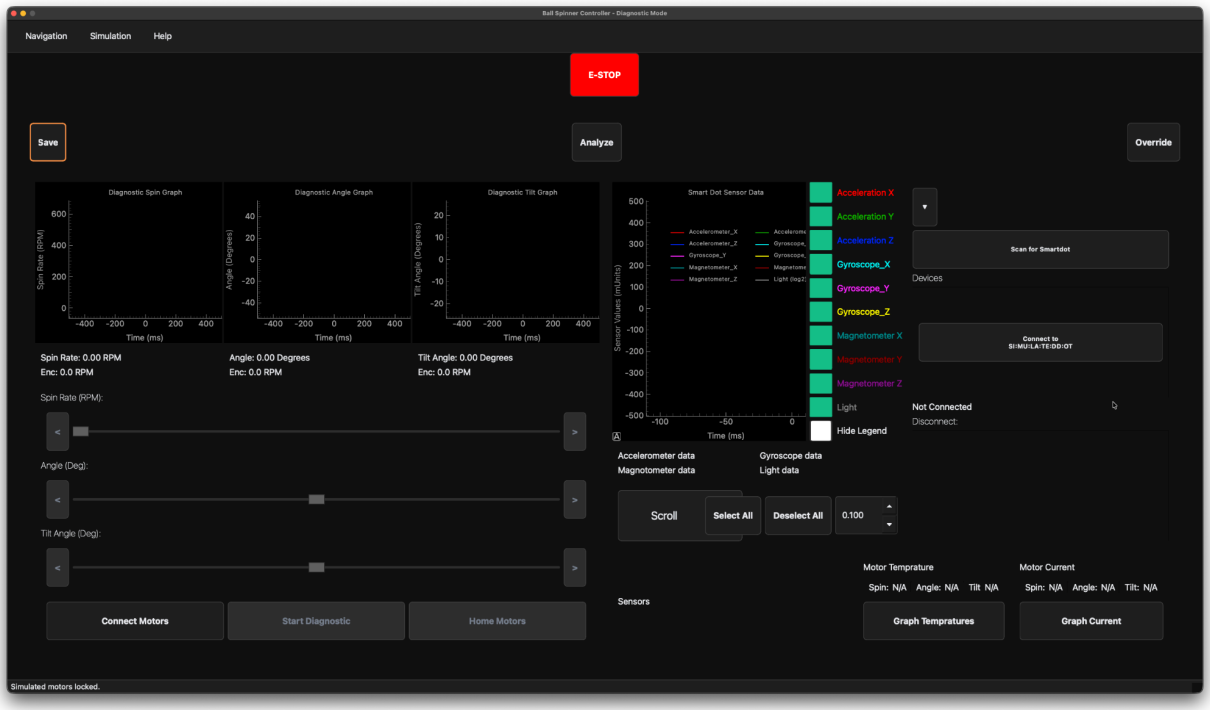


Figure 3.5.6.2.1: Diagnostic Mode Page

When that QTimer triggers, several events happen. First the current positions of the siders are fed into the motors. If the Angle and Tilt Motors haven't changed then sending their positions is skipped. The labels are updated with the motors positions and encoder values. If data collection is active then the positions of the 3 motors as well as their encoder values are passed into the data controller and the circular buffers for the pyqtgraphs. The SmartDot is then polled in order to receive its sensor data which is then passed into the SmartDot Graph on the left side of the Diagnostic Mode Page as shown in Figure 3.5.6.2.1.

When the Stop Diagnostic Button is pressed data collection is ended, but the timer continues until the Disconnect Motors button is pressed. When the Disconnect Motors Button is pressed Qtimer is disabled.When the save button is pressed a dialog is opened allowing the user

to enter a name for their session and submit their data to the database. When the analyze button is pressed the Diagnostic session is opened in the Analysis Mode Page.

Diagnostic Override Mode

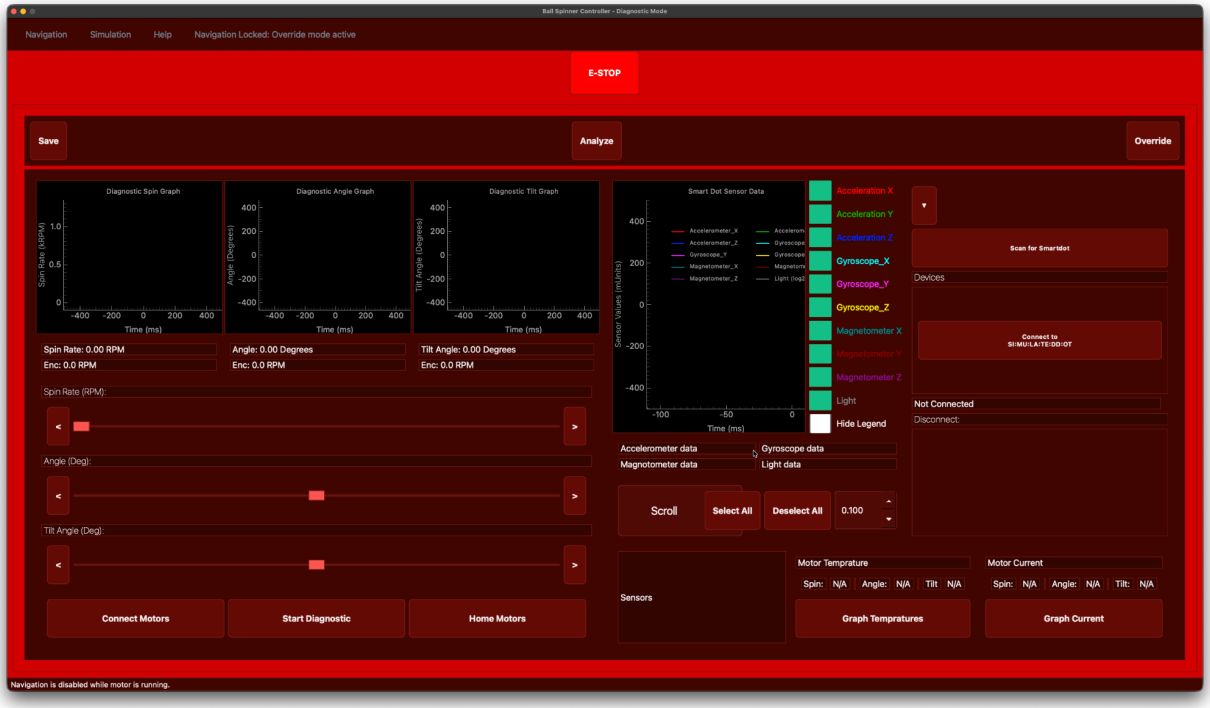


Figure 3.5.6.3.1: Diagnostic Mode Page in Override mode

When the Override Button is pressed in the Diagnostic mode the page enters Override mode as shown in Figure 3.5.6.3.1. In this mode the software limits of the motors are mostly ignored. The Tilt and Angle motors can be driven to $\pm$ 360° and the Spin Motor can be spun up to 1200 rpm.

Shot Mode Page

The Shot Mode Page is an extension of the Qwidget Class.This page loads the ‘ShotModePage.ui’ file to load its ui. The page contains three Input Graphs named for each motor. At the bottom of the page is a slider for controlling the shot length. The page is shown in Figure 3.5.6.4.1. When the slider is adjusted it calls the input graphs set_bounds() method.

Figure 3.5.6.4.1: Shot Mode Page

When Enter Shot is pressed the page calls the sample_spline_display() with a sample interval of 50ms and loads the results into the resulting instructions into the BSC’s data controller. The changePage pyqtsignal is then emitted to change the page to Shot View Page

Shot View Page

The Shot View Page is an extension of the Qwidget Class. This page loads the ‘ShotViewPage.ui’ file to load its UI.

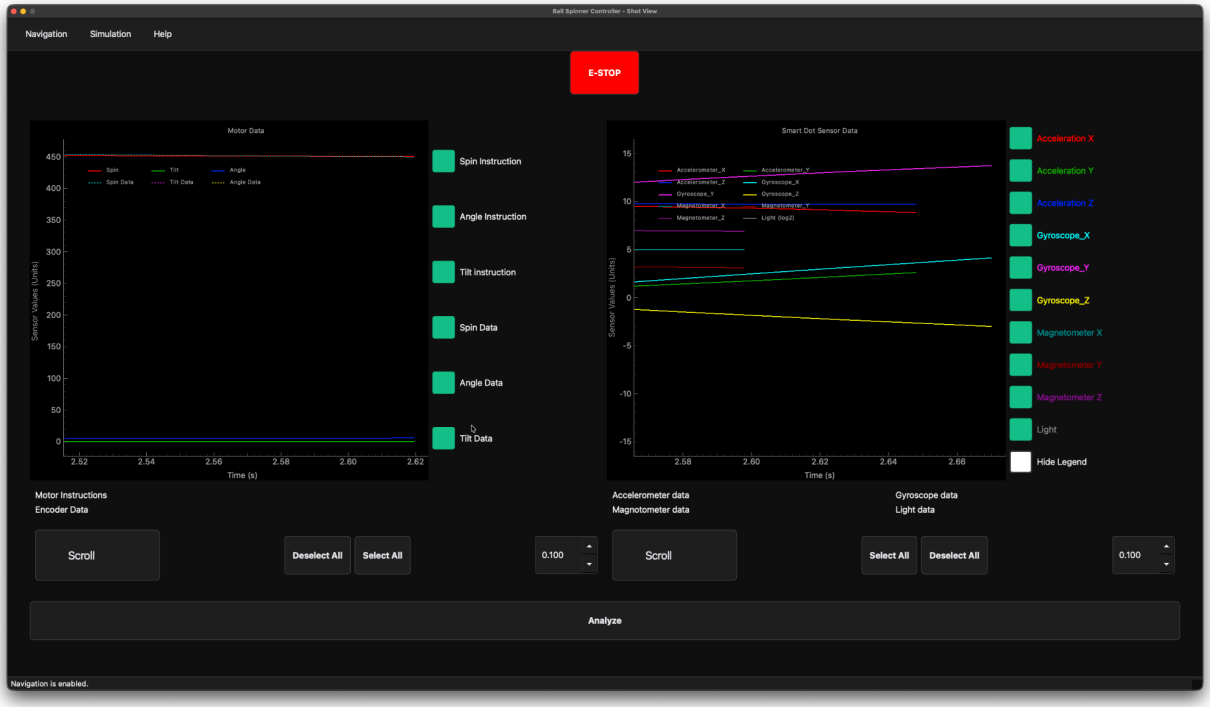


Figure 3.5.6.5.1: Shot View Page

When the page starts it unpacks the instructions from a controller using the PackageMotorData() function in our utility folder. The page contains arrays for both the motor instructions and the values when the motor completes. The page finds a delta time by subtracting the first and second time values from the motor instructions. Additionally note is taken of the start time for later use. The page also contains a pool of 9 threads to be used for calling the motors. Before starting the motors the page emits the lockNaigation pyqtsignal to prevent the user from navigating back to the home page. The analyze button, as shown in Figure 3.5.6.5.1, is also disabled until the shot is complete.

The Qtimer is started, and triggers on intervals of the calculated delta time. The total elapsed time is calculated and a thread is spawned from the pool to deliver the instruction to the motor asynchronously. Finally the timer checks to see if the time is greater than the final time or if the instruction is the last instruction then it ends the loop. These checks are redundant as they

should happen at the same time under normal conditions, however the redundant checks are kept in to avoid any issues of the instructions time not matching the given max time.

Upon a threads completion, the motor it was controlling and the time of its completion are returned and stored in the displayed array. This array is then graphed upon the next clock cycle. After the Qtimer stops itself all of the data from the Smartdot is parsed into the BSC's data controller and the page releases the navigation lock and enables the button to analyze the shot that was just created.

Data View Page

The Data View Page is an extension of the Qwidget Class. This page loads the 'DataviewPage.ui' file to load its ui. This Page lets one send a query to the database based on the time range that is selected. The default time range is 1 week prior to and 2 hours after the launch of the program. Once the data is retrieved, specific terms can be searched for by typing in the search box shown in Figure 3.5.6.6.1 and filtered for the shot type using the controls at the top of the page.

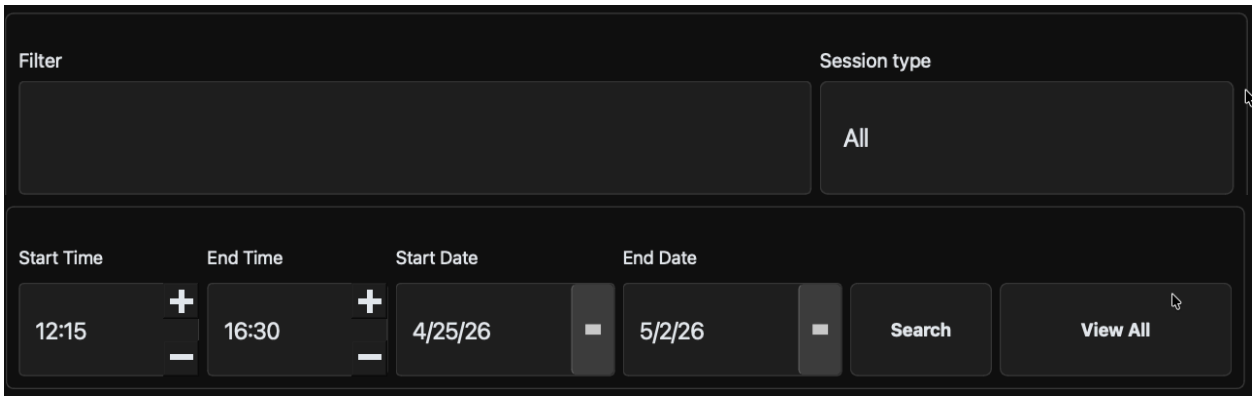


Figure 3.5.6.6.1: Filter Controls on Data View Page

When a shot in the table is selected, the buttons at the bottom of the page can be pressed to analyse the selected shot, edit the selected shot, or replay the selected shot. All buttons load the data from the selected row shown in Figure 3.5.6.6.2: into the data controller.

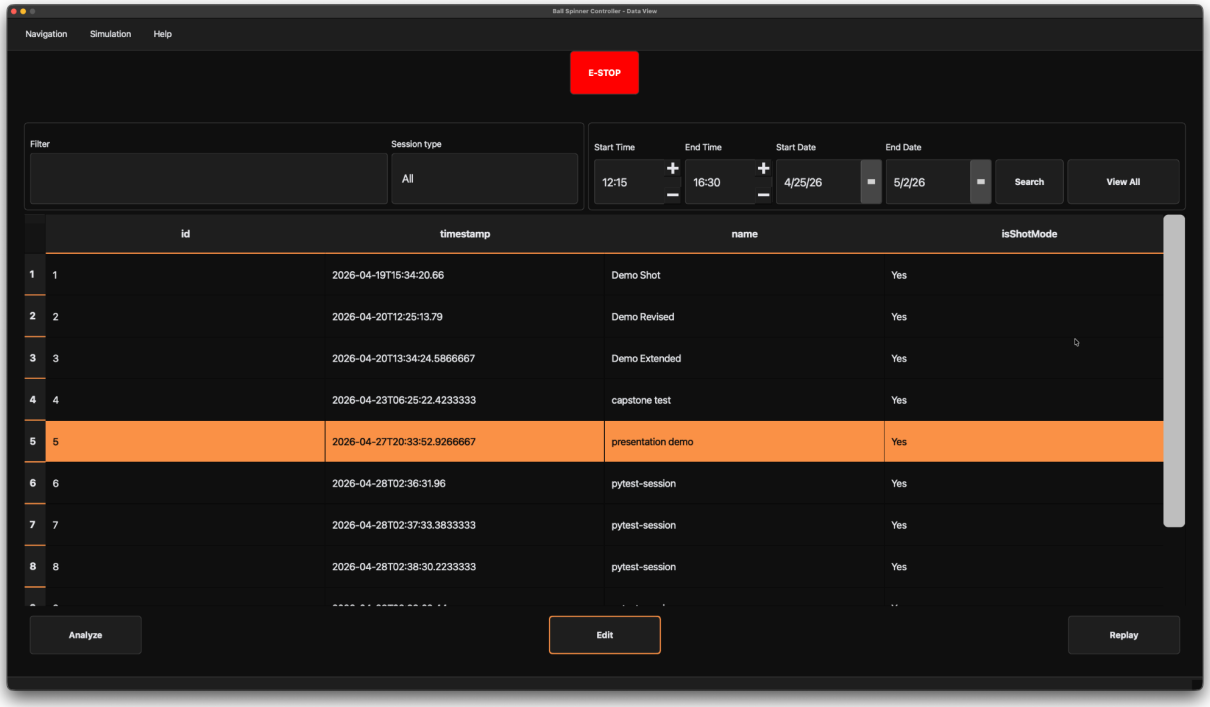


Figure 3.5.6.6.2: Data View Page with row selected.

Analysis Mode Page

The Analysis Mode Page is an extension of the Qwidget Class.This page loads the 'AnalysisModePage.ui' file to load its UI. The Analysis Mode Page loads the data from the DataController and displays it on the Motor Graph and Smart Dot Graph shown in Figure 3.5.6.7.1. The save button shown in the center of the page uploads the data from the DataController into the Cloud.

Figure 3.5.6.7.1: Analysis Mode Page with real Motor and SmartDot data

The central analysis options open an analysis dialog performing the operation stated on the button. The Motor FFT and the SmartDot FFT buttons perform a Fast Fast Fourier Transform on their respective data as shown in Figure 3.5.6.7.2. The Motor First Derivative, SmartDot First Derivative, Motor Second Derivative, SmartDot Second Derivative, Motor Bandpass and SmartDot Bandpass buttons all perform their stated operation on their relevant data as shown in Figure 3.5.6.7.3, Figure 3.5.6.7.4, and Figure 3.5.6.7.5 .

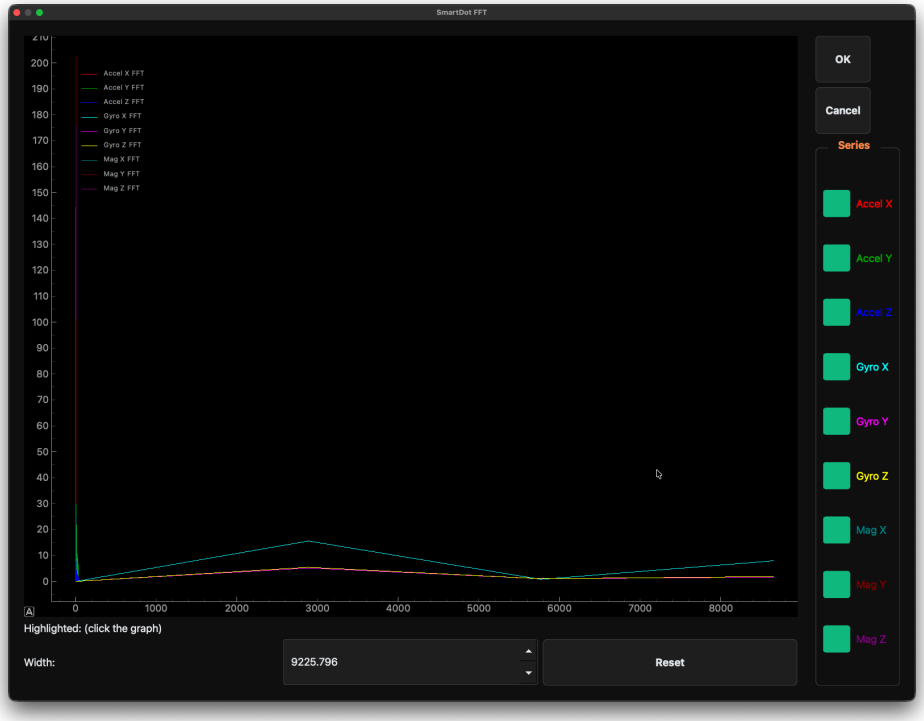


Figure 3.5.6.7.2: Analysis Dialog Box for the SmartDot Fast Fourier Transform

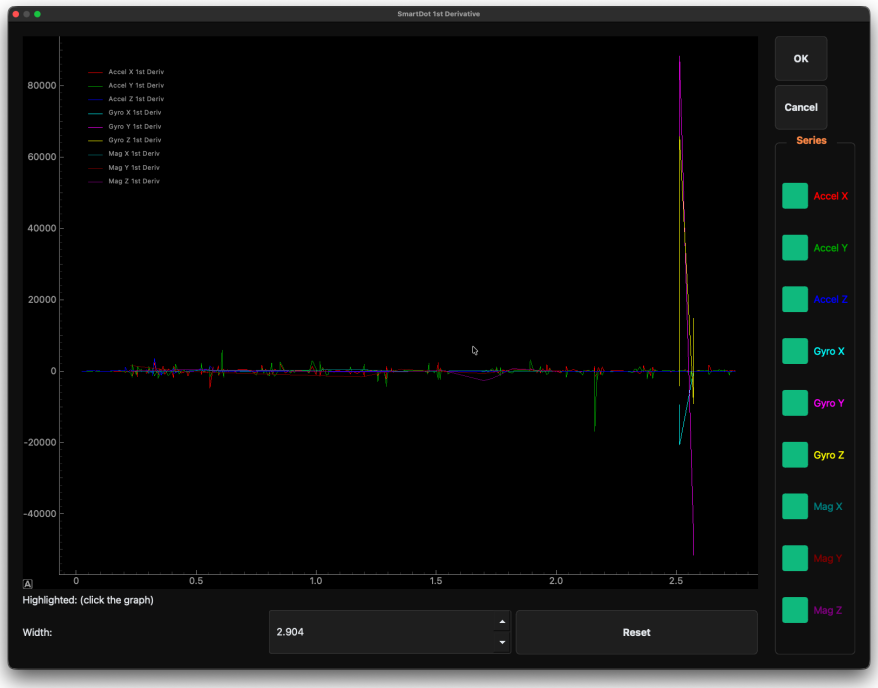


Figure 3.5.6.7.3: Analysis Dialog Box for the SmartDot First Devitive

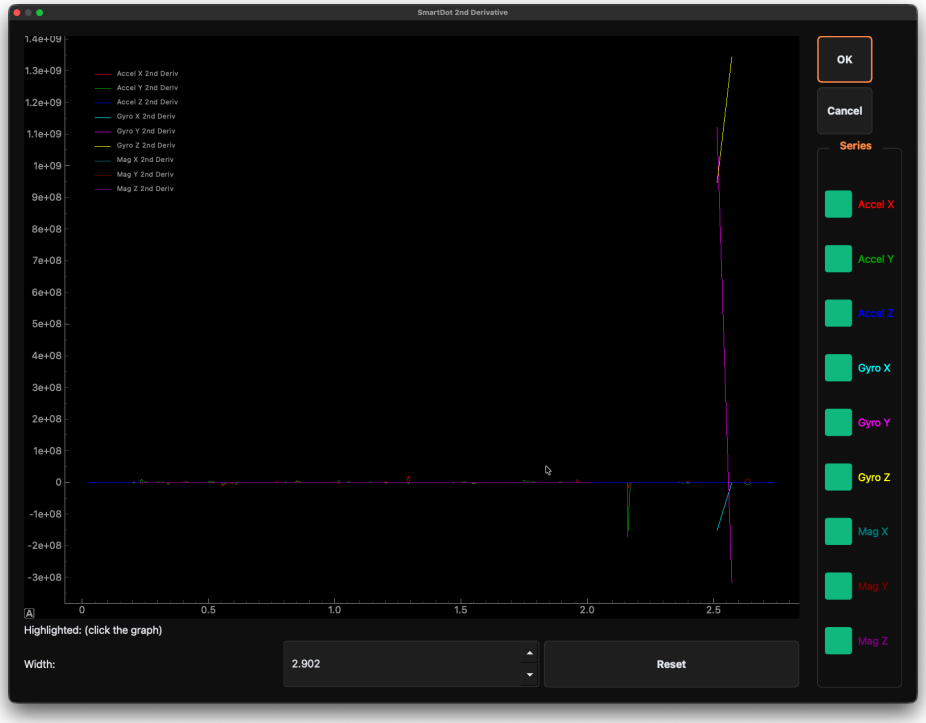


Figure 3.5.6.7.4: Analysis Dialog Box for the SmartDot Second Derivative

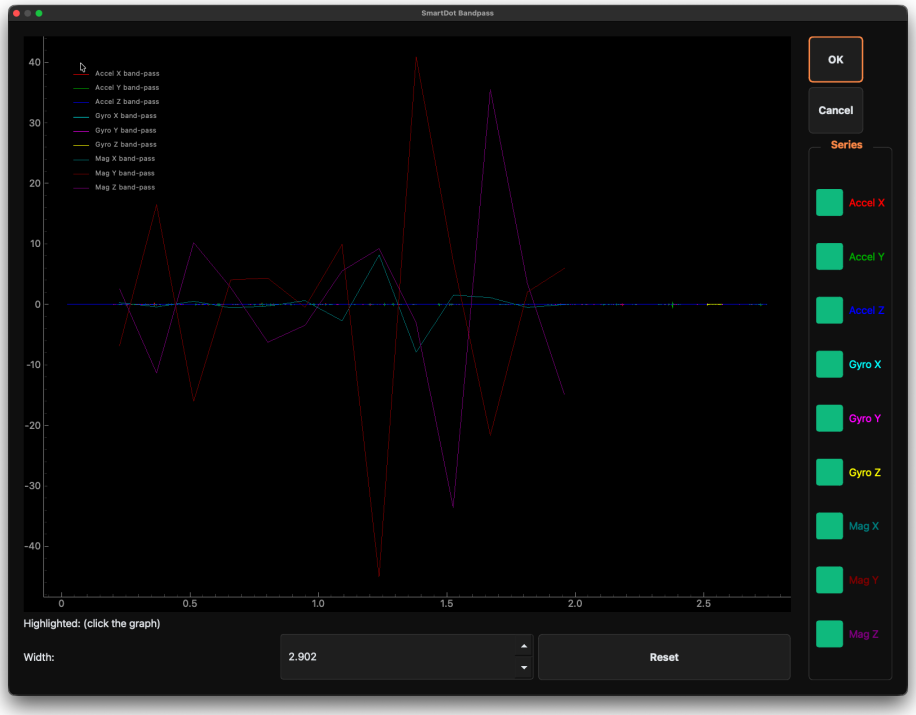


Figure 3.5.6.7.5: Analysis Dialog Box for the SmartDot Bandpass

When the SmartDot Wavelet and Motor Wavelet buttons are pressed this allows the user to perform wavelet decomposition using a Discrete Wavelet Transform from the PyWavelets library. Wavelet decomposition is the process of convolving your input signal against a known wavelet for the purpose of analyzing the signal in both the time and frequency domains. The pressed buttons open the menu as shown in Figure 3.5.6.7.6 which allows the user to select the wavelets to decompose and whether to isolate the High frequency or the details. The family and type fields are used to select the specific wavelet used for the decomposition. The order of the wavelet refers to the level or number of times the wavelet is convolved through the signal. Finally the Isolate High-Frequency and the Isolate Low frequency determine which portion of the results are not zeroed before performing the Discrete Inverse Wavelet Transform. The results of the full process can be seen in Figure 3.5.6.7.7. All of the results of the operations are graphed individually. Additionally on the Original page or the Wavelet Decomposition Dialog you can subtract the decompositions from the original signal to perform your analysis. More information on this process can be found in Professor Hake's Thesis.

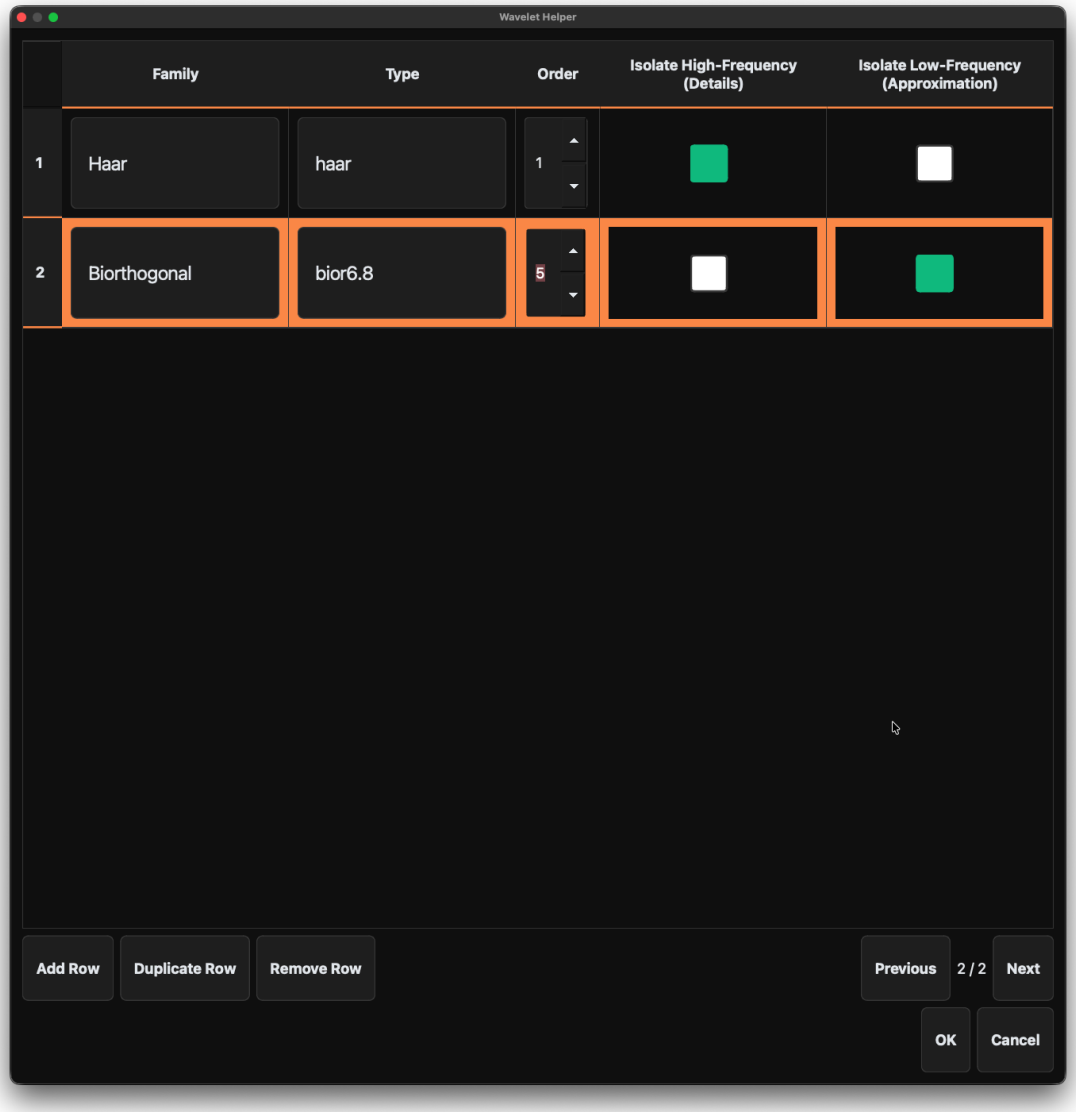


Figure 3.5.6.7.6: Wavelet helper dialog

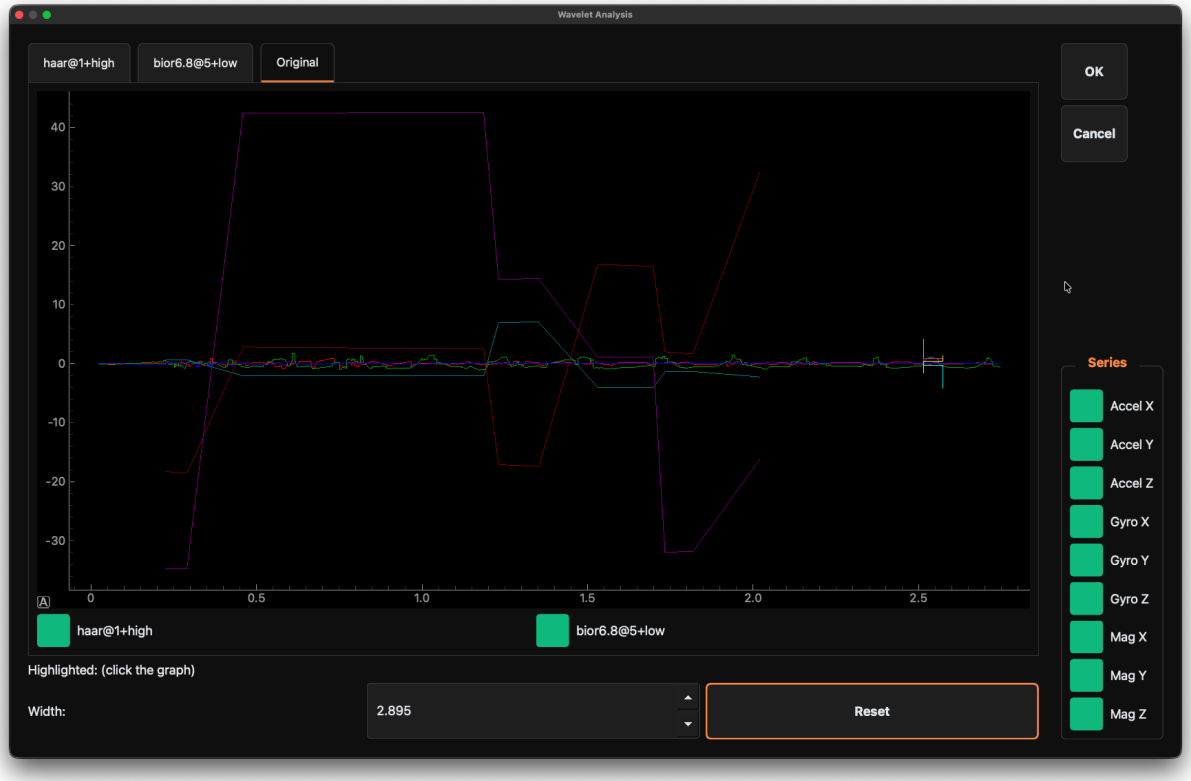


Figure 3.5.6.7.7: Wavelet Decomposition Dialog with SmartDot data and the wavelets analysis from Figure 3.5.6.7.6

Input Graph

The Input graph is an extension of the Qwidget Class. This page has all of its UI defined in the python rather than a separate .ui file. The value min, value max, time min, and time max fields of the graph can be modified to adjust the values given by the graph. There is a function that hides these fields so that they can be fixed for a given input.

The graph part of the Input Graph is a pyqtgraph. When the graph is clicked on, its coordinates are stored and a point is drawn on the graph. Clicking on an existing point deletes that point. A cubic spline is drawn to connect the drawn points as well as the start and end points, as shown in Figure 3.5.6.8.1. This spline is also bounded by the minimum and maximum values

set. The starting and ending values of the spline can be adjusted via the spinboxes at the bottom. The maximum number of points can similarly be adjusted.

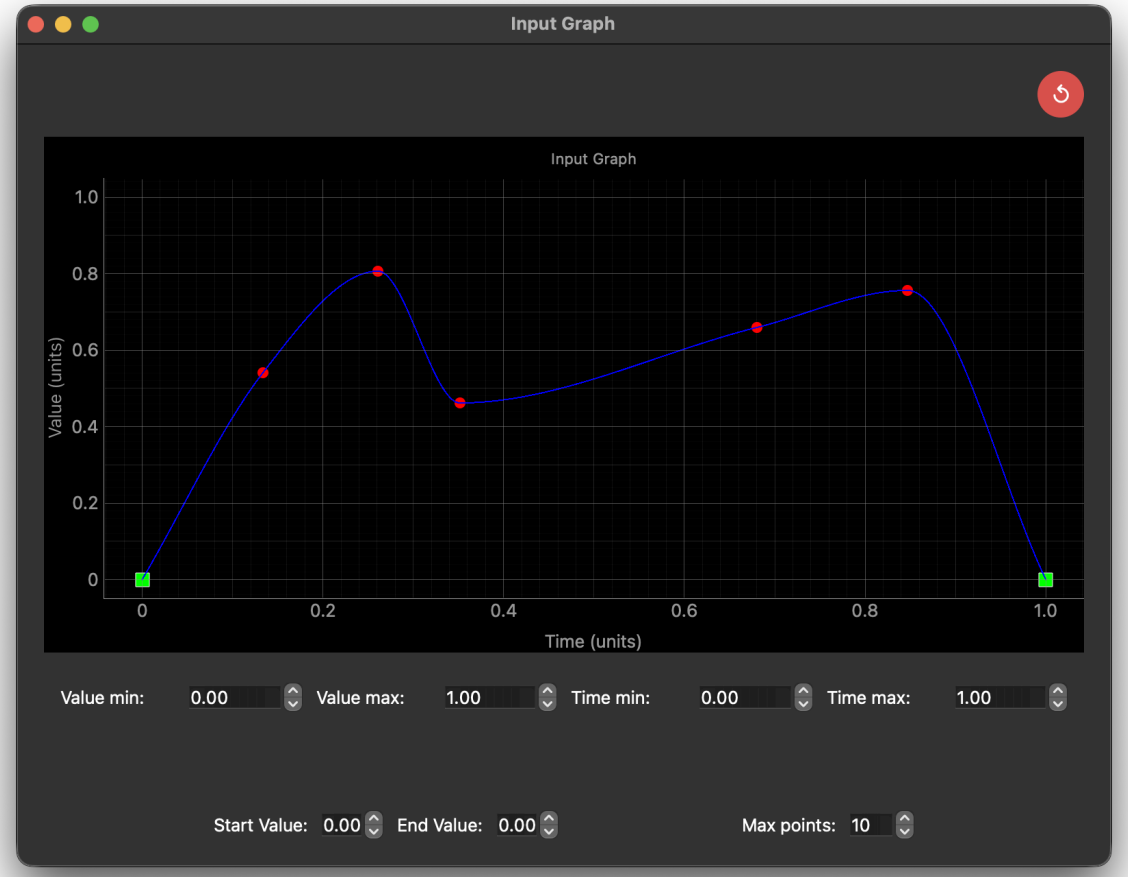


Figure 3.5.6.8.1: Input Graph

The Input Graph contains methods to control all of the given parameters. In addition the graph contains a method called sample_spline_display() that takes in a sample time variable that interpolates the curve at times separated by the sample time. Figure 3.5.6.8.2 shows a visual representation of what the values of the sample_spline_display() look like.

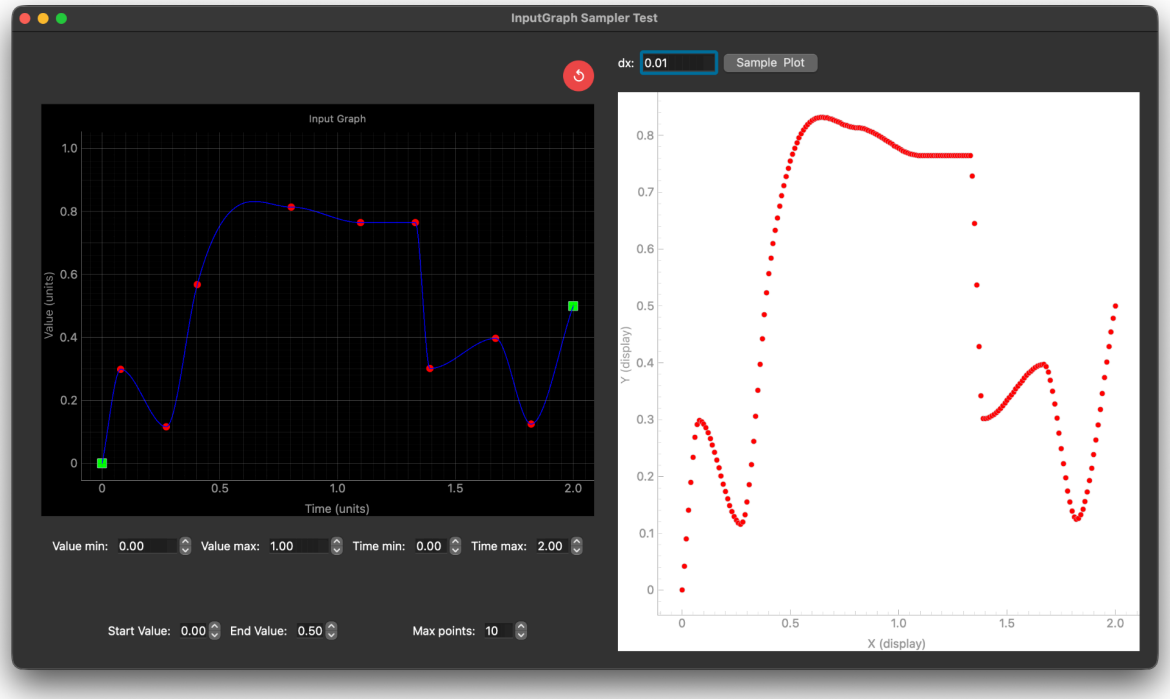


Figure 3.5.6.8.2: Sample Output from Input Graph

SmartDot Connect Widget

The SmartDot Connect Widget is an extension of the Qwidget Class. This widget loads the ‘SmartDotConnectWidget.ui’ file to load its UI. The widget is composed of three utilities. The widget imports a backend utility called SubProcess Scan. There was an issue with running a Bluetooth Scan thread in the same process as a PyQt6 process, where a segmentation fault would be generated. To solve this, the Bluetooth Scan is run in a separate subprocess, the SubProcess Scan runs the ScanSmartDot’s script which runs the Bluetooth Scan. Our PyQt6 process then captures the output from the subprocess and updates the label in real time.

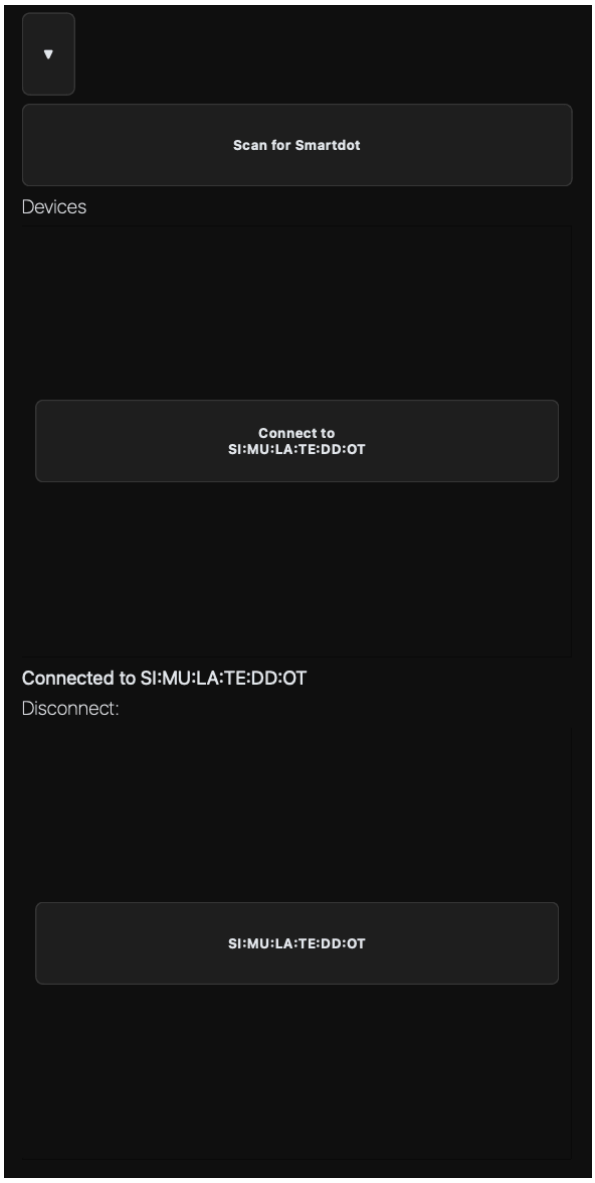


Figure 3.5.6.9.1: the SmartDotConnectWidget

Notice in Figure 3.5.6.9.1, the text reads “Scan complete”. This is a live status label that receives data from the subprocess. It also is updated with the latest action performed on the widget. Besides the Scan button, there are two other functions provided by this widget: connect and disconnect. The widget implements the SmartDotConnectionManager, MetaMotion, and Simulated SmartDot to handle connecting and disconnecting. There are two frame boxes on the widget. The top box includes SmartDots that appeared in the Scan. The Simulated SmartDot will

always appear as an option. A user can click on any of the SmartDot's in the above box to attempt to connect to them. If the connection succeeds or doesn't already exist the SmartDot will be added as a button to the bottom box which handles disconnections. The user can select a SmartDot in the bottom box in order to initiate a graceful disconnection.

SmartDot Graph

The SmartDot Graph is an extension of the Qwidget Class. This widget loads the 'SmartDotGraph.ui' file to load its UI. It contains stored curves for all of the SmartDot's various sensors. An additional curve for the log2 value of the light curve is stored to show being able to graph the y-axis values in a similar y-axis range. This is due to the maximum value of light being a few orders of magnitude higher than any other graph. The page includes checkboxes to toggle the display of the corresponding curve. When the graph is clicked, a vertical cursor is placed, and the nearest points on the curve are marked on the graph and on the labels below it. This is shown in Figure 3.5.6.10.1

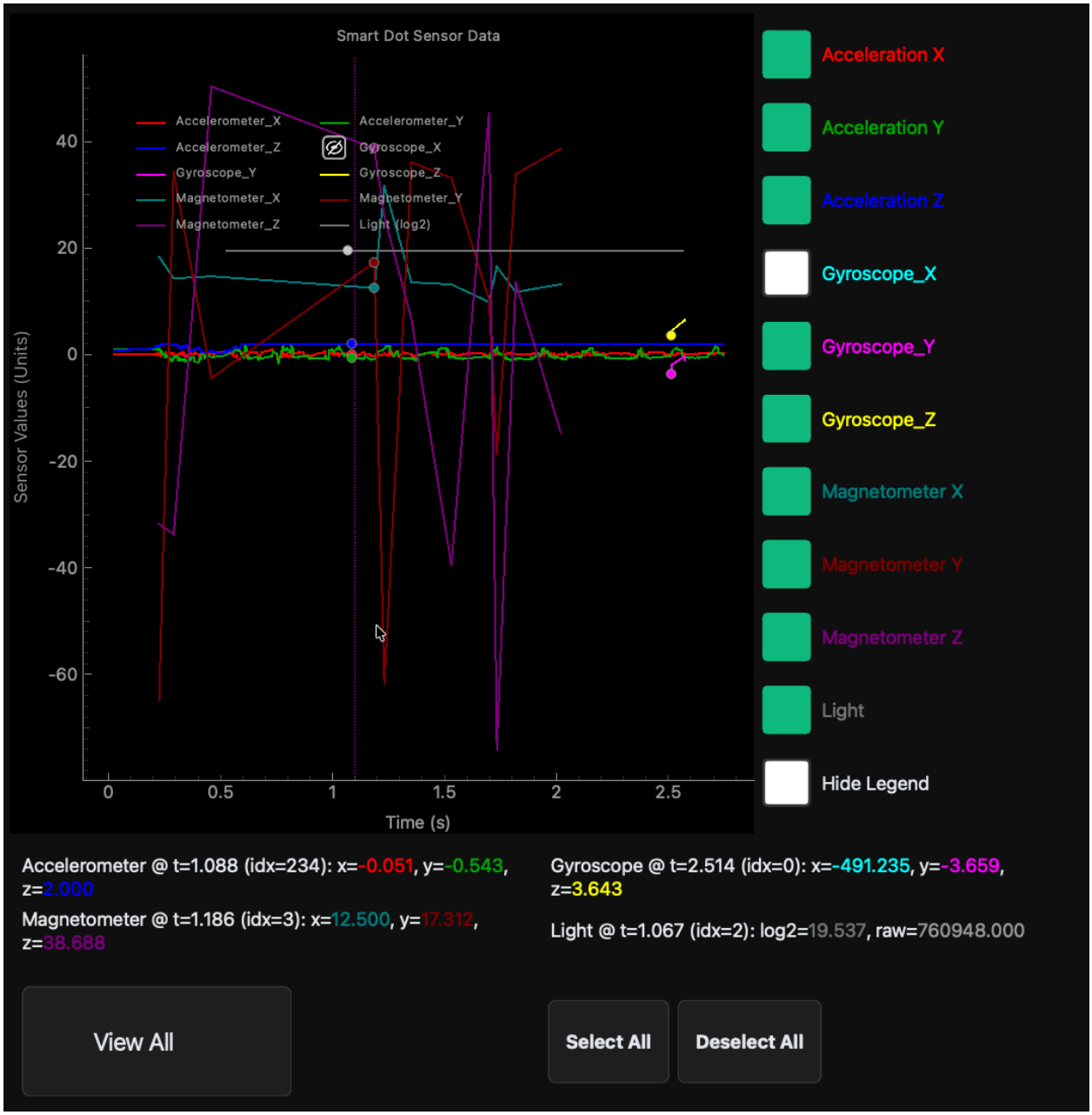


Figure 3.5.6.10.1: SmartDot Graph with vertical cursor and SmartDot data

The graph contains a control for the display mode of the graph. Scroll allows the user to select a number of seconds to see in the past, as shown in *Figure 3.5.6.10.3*. View All shows the shot's stored data, as shown in Figure 3.5.6.10.1 . Range allows the user to select a start and end value to view for the graph, as shown in Figure 3.5.6.10.2:

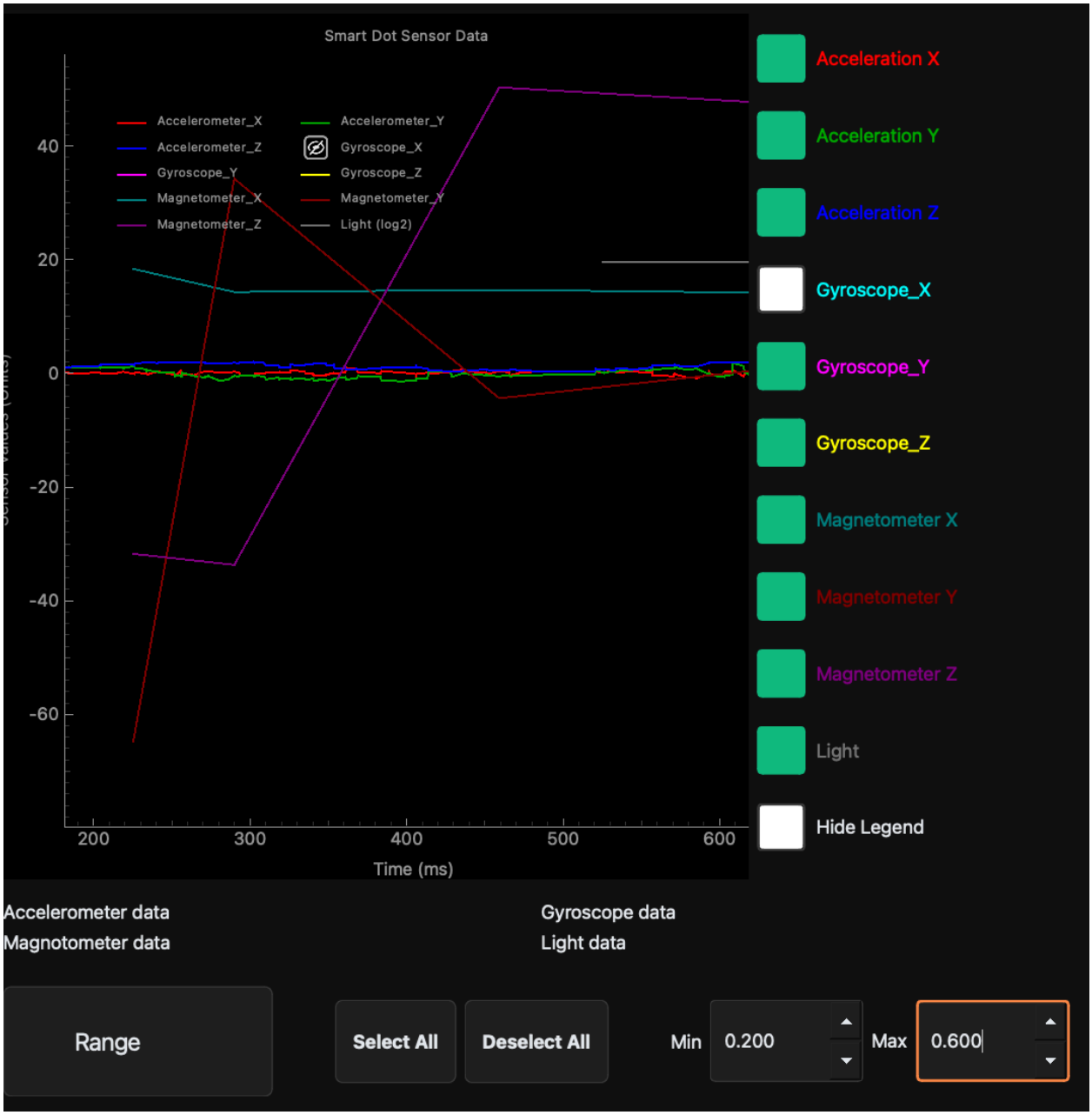


Figure 3.5.6.10.2: SmartDot Graph in Range Mode with simulated SmartDot data

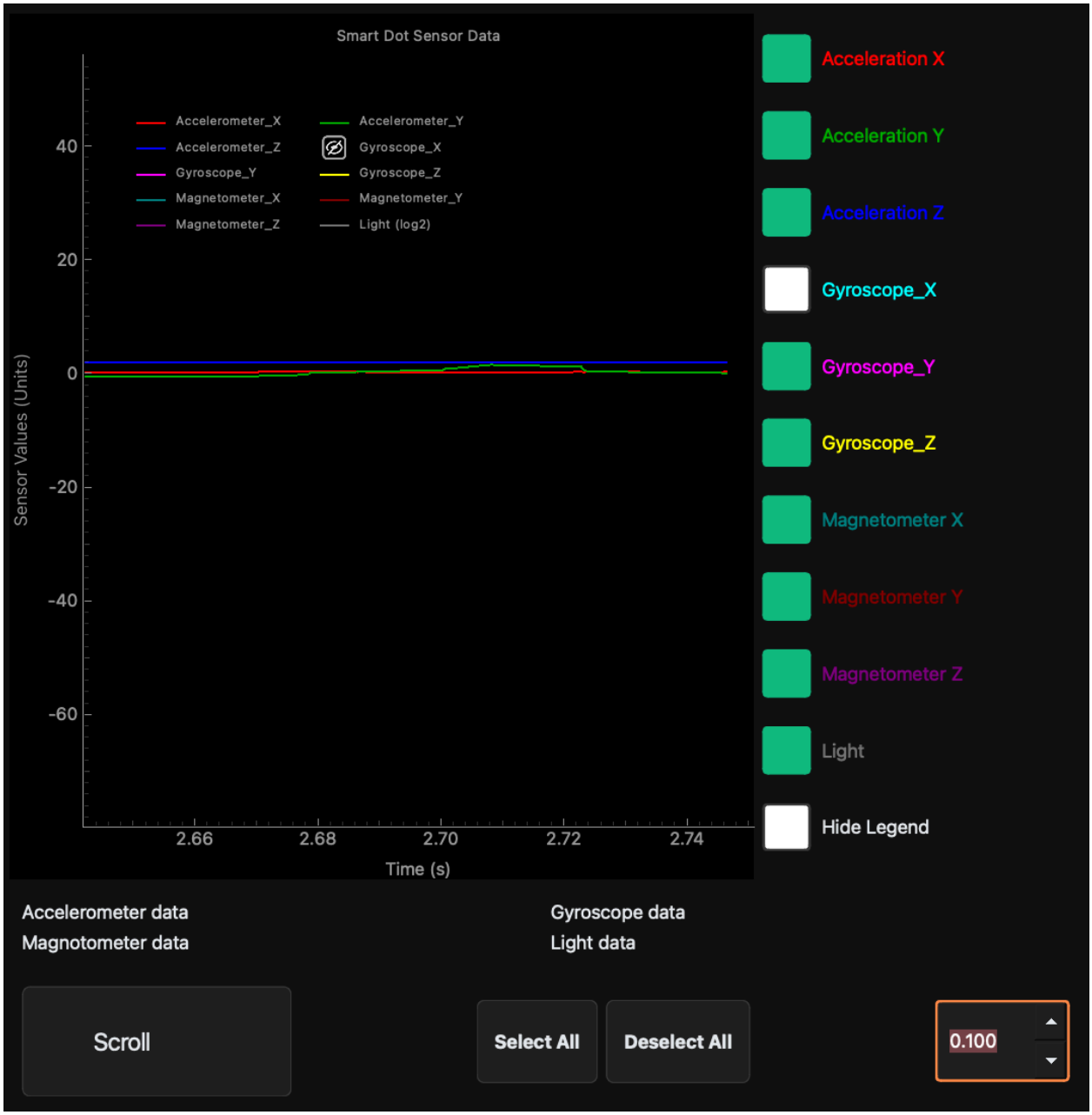


Figure 3.5.6.10.3: SmartDot Graph in Scroll Mode with simulated SmartDot data

Motor Graph

The Motor Graph is an extension of the Qwidget Class. This widget loads the ‘MotorGraph.ui’ file to load its UI. It contains stored curves for the three motors' instructions as

well as the encoder values from the three motors, as shown in Figure 3.5.6.11.1. The page includes checkboxes to toggle the display of the corresponding curve. When the graph is clicked, a vertical cursor is placed, and the nearest points on the curve are marked on the graph and on the labels below it. This graph contains the same controls and view modes as the SmartDot Graph.

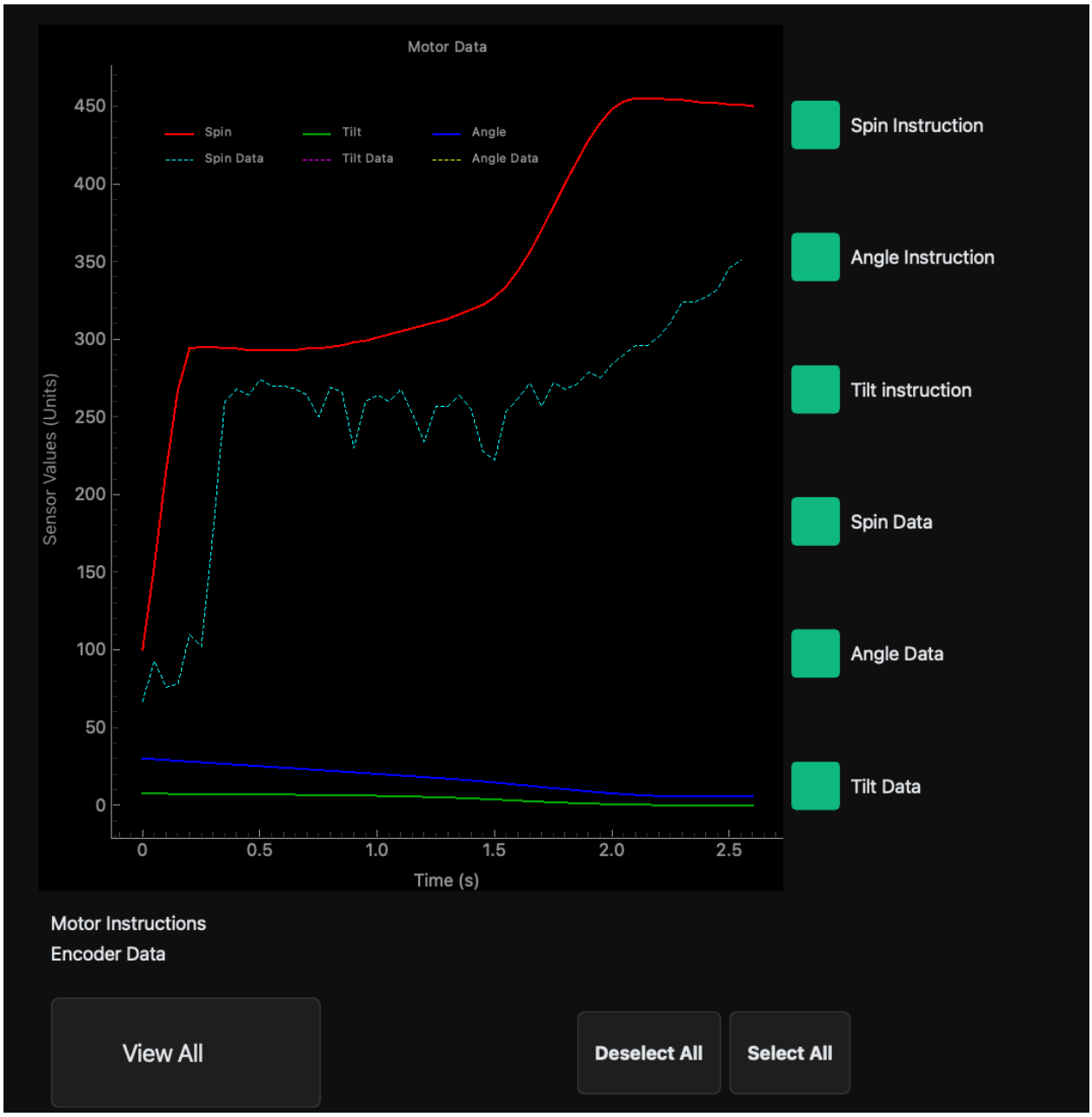


Figure 3.5.6.11.1: Motor Graph in View All Mode

Ball Spinner Mechanical System

Overview

The Ball Spinner system was chosen to be implemented as three independently-driven axes of rotation that would be divided into three degrees of motion with the first degree spinning the smartdot about the x axis, the second rotating the first about the y axis, and the third rotating the prior two about the z axis. Descriptions of these axes can be found in the Ball Spinner Mechanical System Design section and are shown in Figure 2.7.1.1. This was initially developed into a wooden prototype as a proof of concept, and was redesigned as an aluminum version after feedback was collected.

Wooden Prototype

The initial design of the Ball Spinner system was developed into a rough prototype made from materials that were available in the shop workspace along with components from the RevMetrix Spring 2025 team. This prototype was able to move the entire required range of motion, but was unpowered as it was not an accurate representation of the size or weight that would need to be rotated. The purpose of this prototype was to represent the initial design to the larger project team so that feedback could be collected and changes could be made to meet the requirements of the other teams. Additionally, the prototype was intended to further the physical team's understanding of issues or complications that might arise from the design. Pictures of the initial wooden prototype can be found below in Figure 3.6.2.1 and Figure 3.6.2.2.

Figure 3.6.2.1: Initial wooden prototype of the Ball Spinner Mechanism

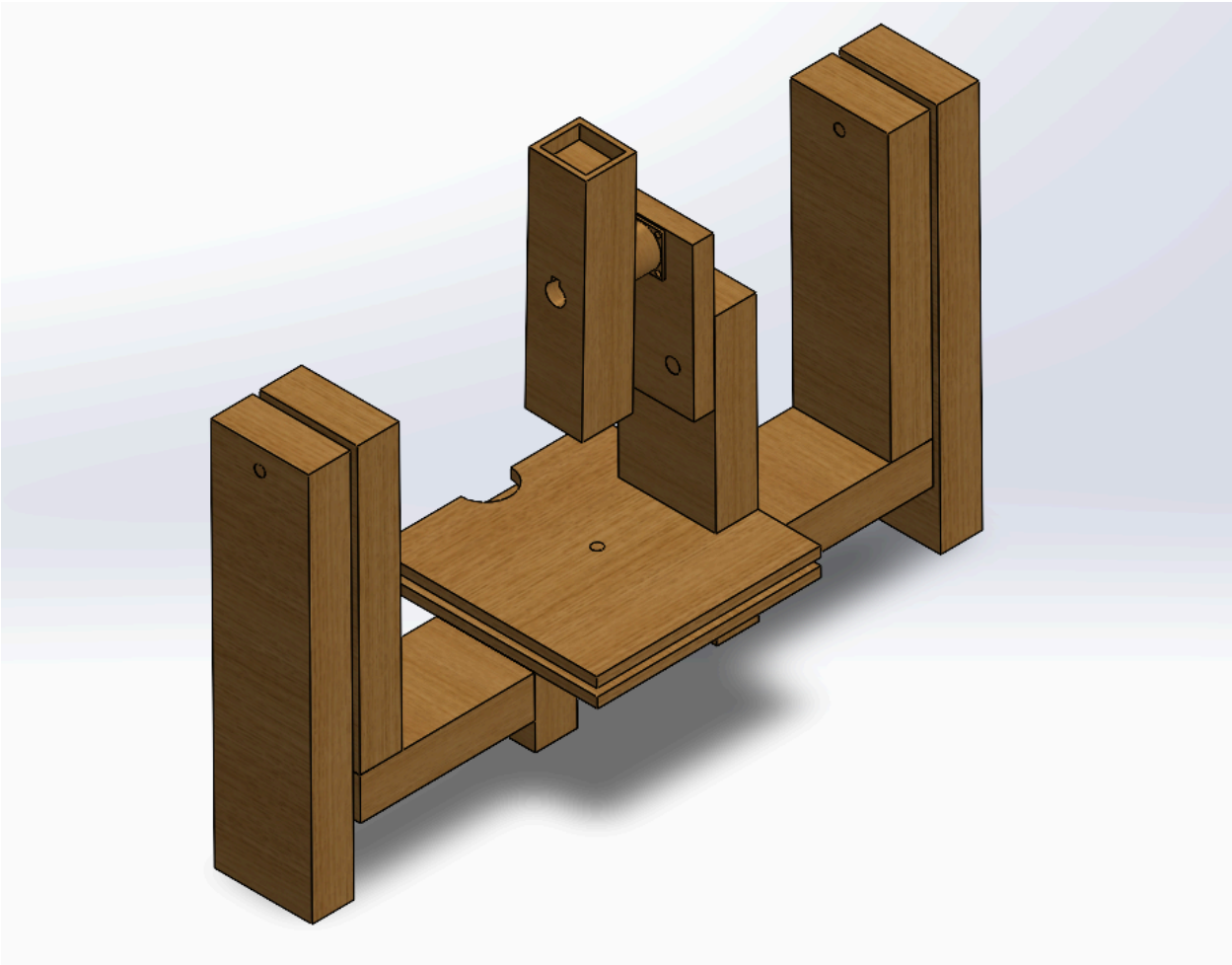


Figure 3.6.2.2: SolidWorks model of wooden prototype with first iteration of the SmartDot holder

Second Design

After feedback was received from the larger team, a design for a second version of the system was developed with the goal of minimizing the size and weight of the system to reduce the required torque and power consumption of the motors. This design is based around aluminum as the primary material because of its accessibility and ease of manufacturing, and is shown in Figure 3.6.3.1. Where necessary, some additional parts will be 3D printed, such as motor and limit switch mounts and the SmartDot Holder, to avoid excess time spent machining complex parts. A physical prototype that can be seen in Figure 3.6.3.2 was 3D printed for this model to visualize spacing within the acrylic encasing. Parts for the model were primarily selected from

the aluminum catalogue on McMaster-Carr. Links to these parts can be found in Appendix A [2][3]. The SmartDot Holder that makes up the majority of the 1st degree of motion is detailed more in the following section.

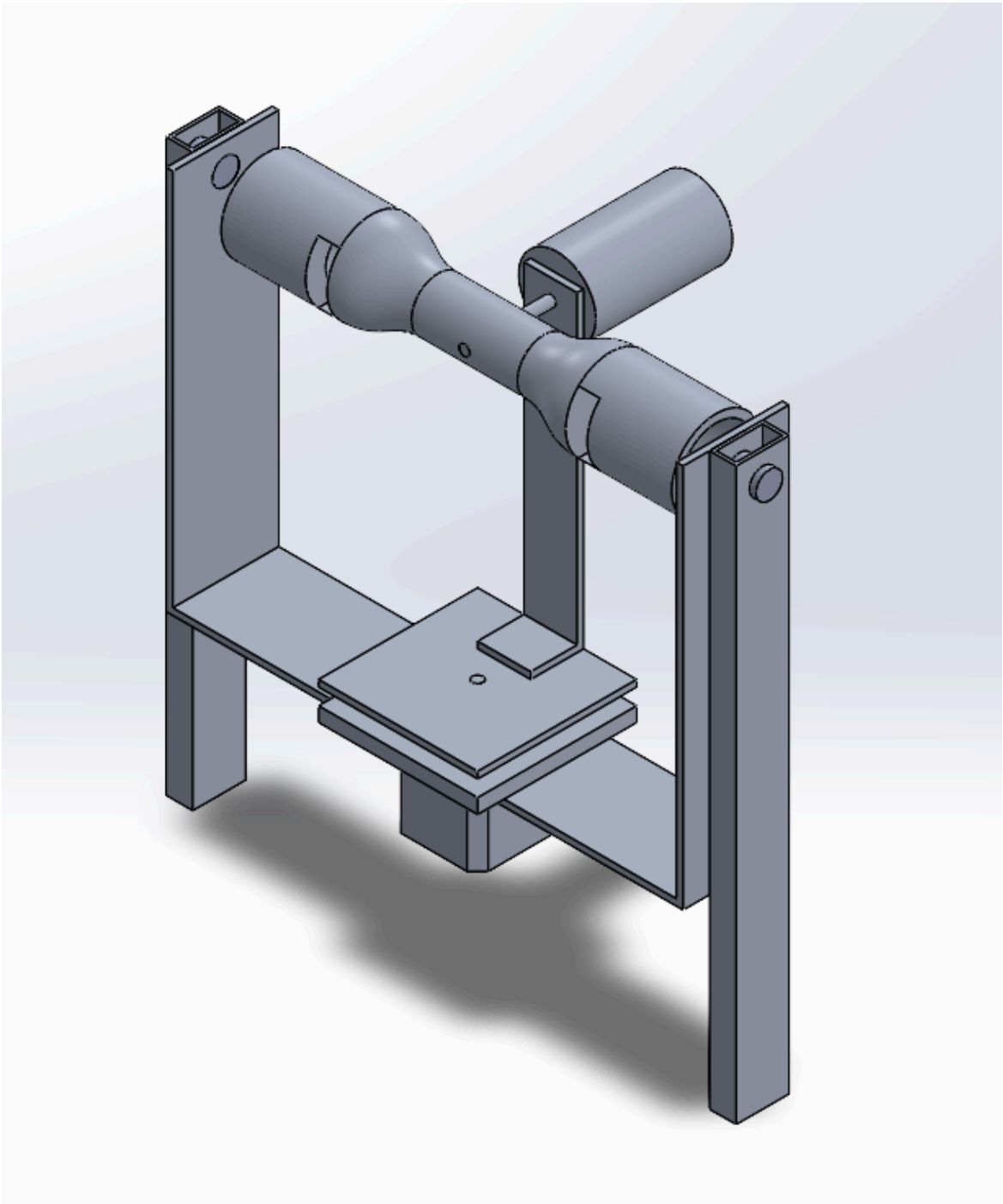


Figure 3.6.3.1: SolidWorks model for the updated second design of the Ball Spinner Mechanism

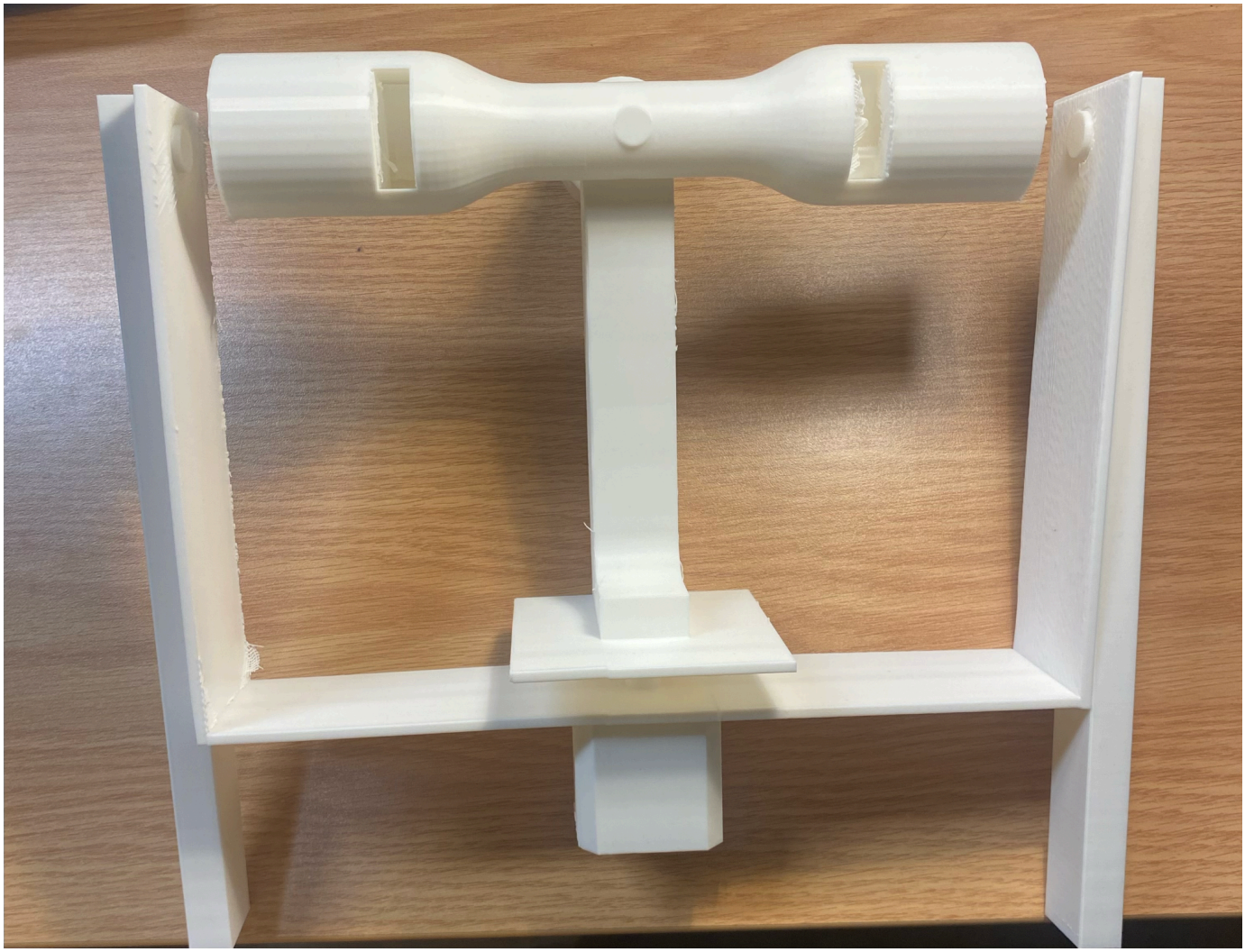


Figure 3.6.3.2: Static to scale 3D print of the second Ball Spinner Mechanism design

Final Design

Once beginning fabrication, further adjustments were made to the model to improve its mechanical robustness. The third degree motor is now mounted to the U-bracket via a shaft collar with face mounts. This ensures a much more stable connection between the motor and the center assembly. Limit switch mounts were also added for the 2nd and 3rd degrees of motion. Both mounts were made with 3D printed PLA and were tapped to allow for limit switch mounting, with the mounts themselves bolting into the U-bracket and 3rd degree motor bracket. The final design also features a redesigned L-bracket. This part was fabricated out of a 6” x 6” x

1 ¼ “ aluminum block. The original profile was cut in the Wire EDM. Following this, mounting holes and other surface finishes were completed in the CNC mill. The L-bracket features gussets that were included in the profile of the original EDM cut. The U-bracket also contains gussets which are bolted onto the plate. Reinforcements on both of these pieces help to reduce vibration. The last addition to the model was the incorporation of a counterweight system. These extensions mount directly onto the insides of the U-bracket and have holes located at the top of the plate to mount bolts with brass weights. This was done to help balance the third degree and shift the center of mass of the U-bracket closer to the 3rd degree motor shaft. The final design can be seen in Figure 2.7.1.1

SmartDot Holder

The SmartDot holder is the component that is mounted on the first degree of freedom motor and houses the SmartDot sensors. The design was made to be 8.5 inches long to accurately model the diameter of an actual bowling ball. This initial design, as seen in Figure 3.6.4.1, featured a rectangular design with a keyed shaft to attach to the motor and a small slot in the top to hold the SmartDot. This design was an adaptation of the project’s previous iteration. The second design features a cylindrical body with two slots within the widened ends to friction fit the SmartDots. Holes located directly behind these slots allow for easy SmartDot replacement. A model of the second design can be seen below in Figure 3.6.4.2. The third design, shown in Figure 3.6.4.3, improves upon the previous design by minimizing the width of the outer sections of the rotational arm to decrease the overall weight of the part. This design also improves the tolerances and dimensions of the slot to allow for the reduced size and weight. Furthermore, this design features press fit finger inserts holes above the SmartDot slots and a hole located 90 degrees from the shaft hole to allow for set screw placement. The fourth design, shown in Figure

3.6.4.4, decreases the diameter of the central portion of the rotational arm in order to minimize the mass of the part and to ensure no interference occurs between the L-bracket and the SmartDot holder.

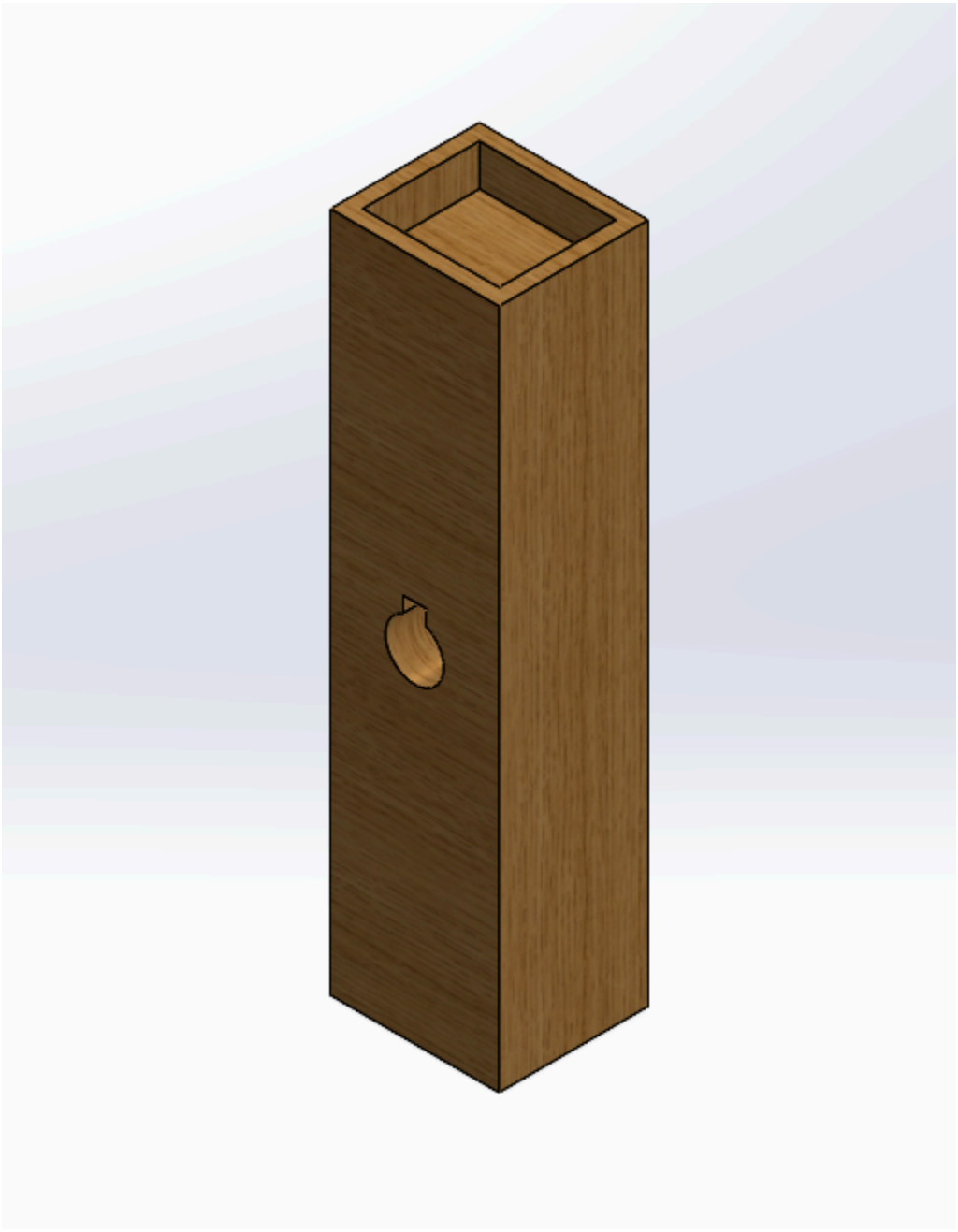


Figure 3.6.4.1: First Iteration of the SmartDot Holder Design

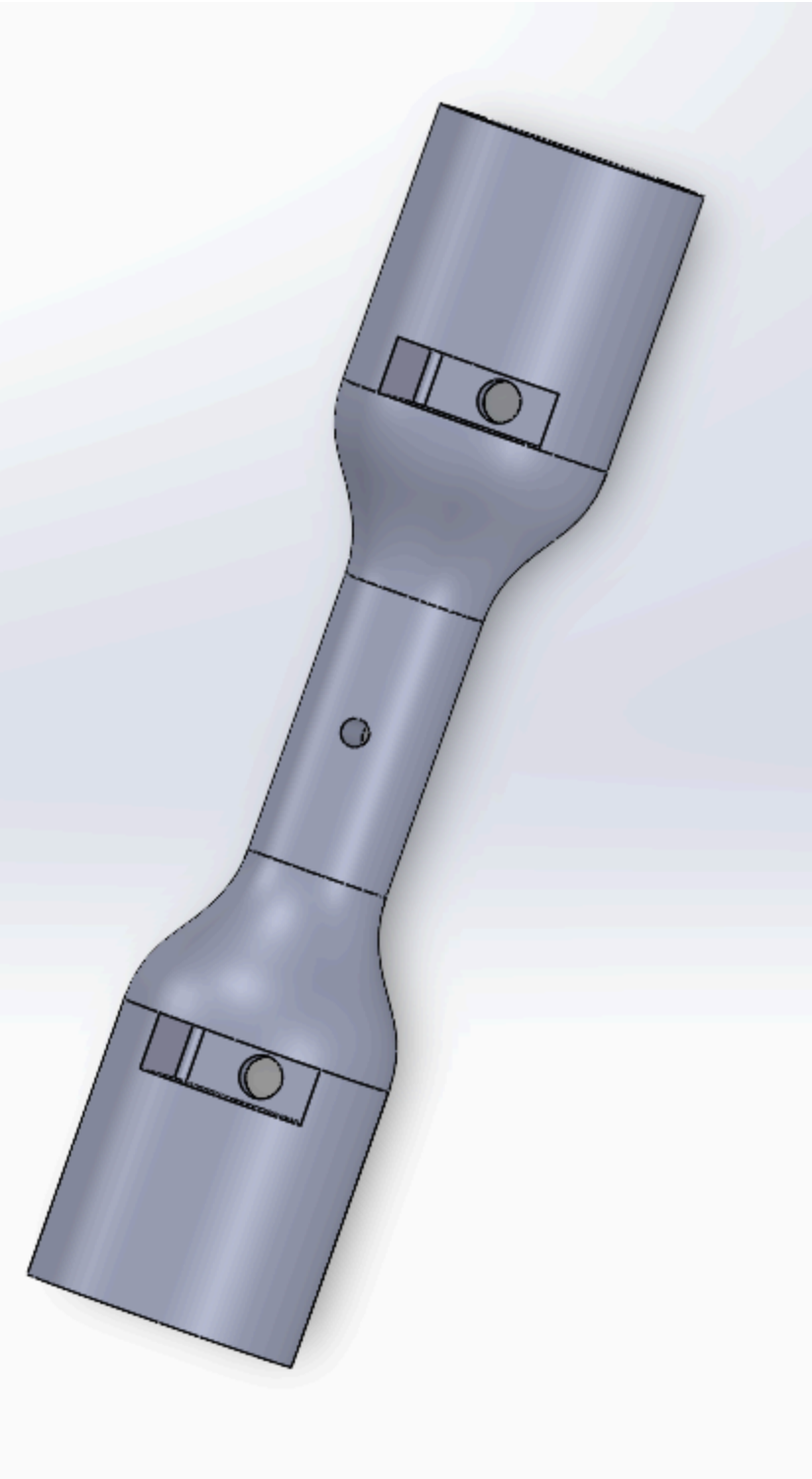


Figure 3.6.4.2: Second Design of the SmartDot Holder

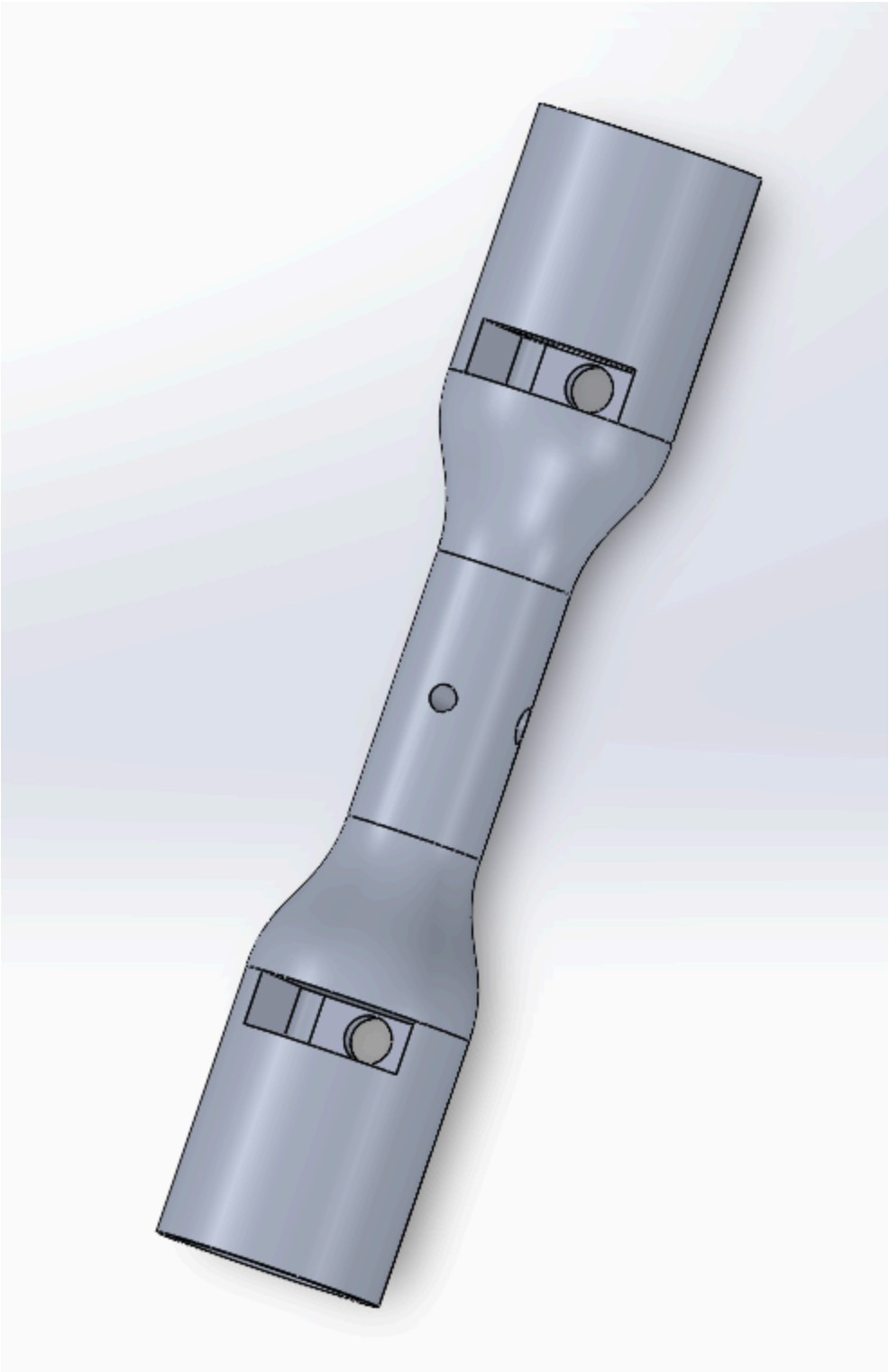


Figure 3.6.4.3: Third Design of the SmartDot Holder with Set Screw Hole

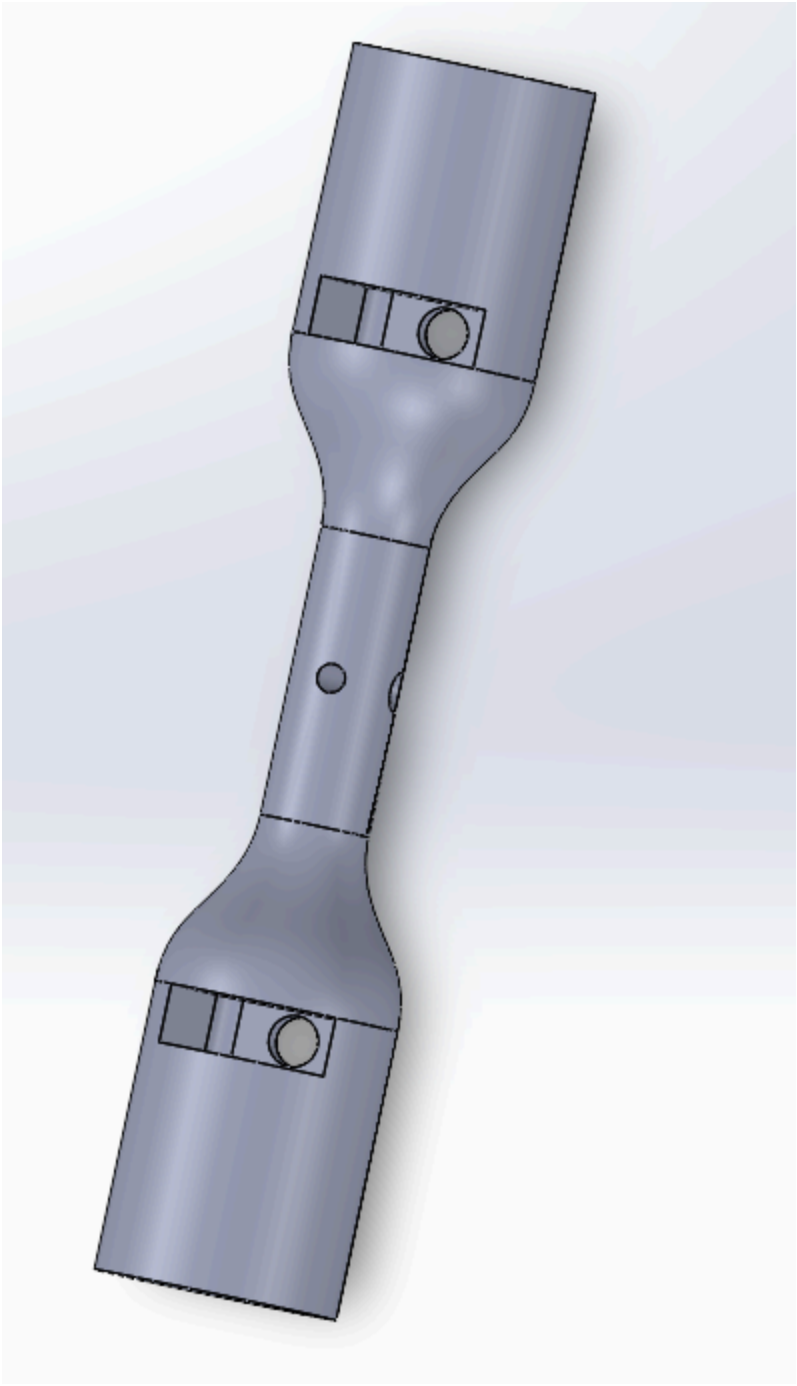


Figure 3.6.4.4: Fourth Design of the SmartDot Holder with Smaller Central Diameter

Safety Housing

To prevent injury, it was decided to use a clear acrylic cover to obstruct access to the system during operation. The acrylic cover is a 12 in by 12 in by 12 in box of 0.118 inch

thickness. The box was purchased from Amazon at the link in Appendix A [1]. The box is elevated using 3D printed supports at each corner to allow room for wires and electrical hardware. The acrylic case also contains holes to allow for T-handle spring plungers which keep the acrylic in place during a collision with moving components. In Figure 3.6.5.1 below, the acrylic case can be seen mounted on the three degree of freedom system.

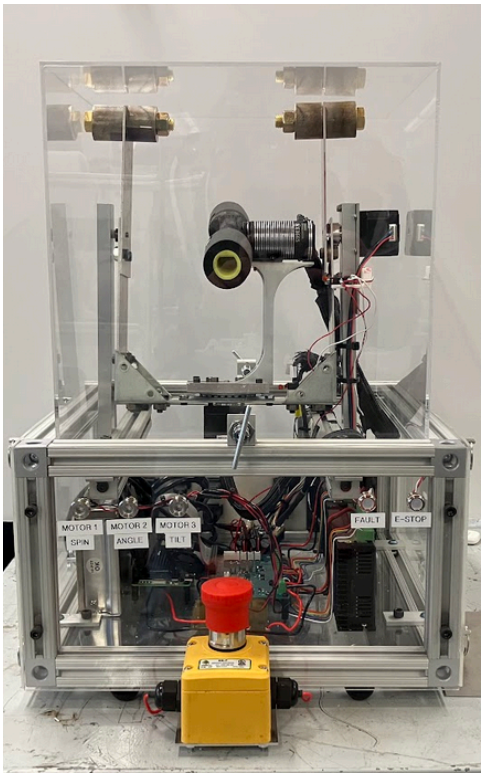


Figure 3.6.5.1: Enclosure Platform With Acrylic Enclosure Over Full Scale Design Model

BSC Enclosure and Platform

With the integration of the Ball Spinner Controller into the Ball Spinner Mechanical System, there was a need to house all of the electrical hardware which would spatially be close to the 3DOF mechanical system. The initial design shown in Figure 3.6.6.1, pictured below, was

made through collaboration with the Ball Spinner Controller Team and was to be a separate enclosure used to feed wires through the top of the acrylic safety enclosure to control the motors.

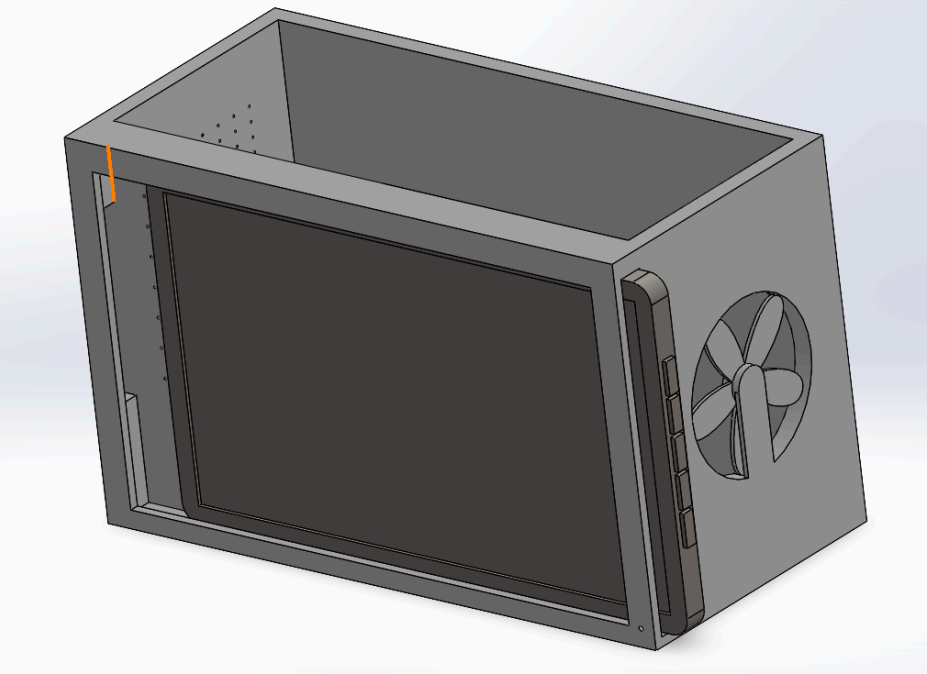


Figure 3.6.6.1: Preliminary Design of BSC Enclosure

This design was later altered to house the components below the acrylic safety enclosure with the touchscreen as a standalone component to the assembly. The platform enclosure is a 14”x14” box with removable top and side panels and interior space for the Ball Spinner Controller Team’s equipment, such as the power supply, motor drivers, and raspberry pi. Considerations have been made for wire routing, ease of access, and future adjustability.

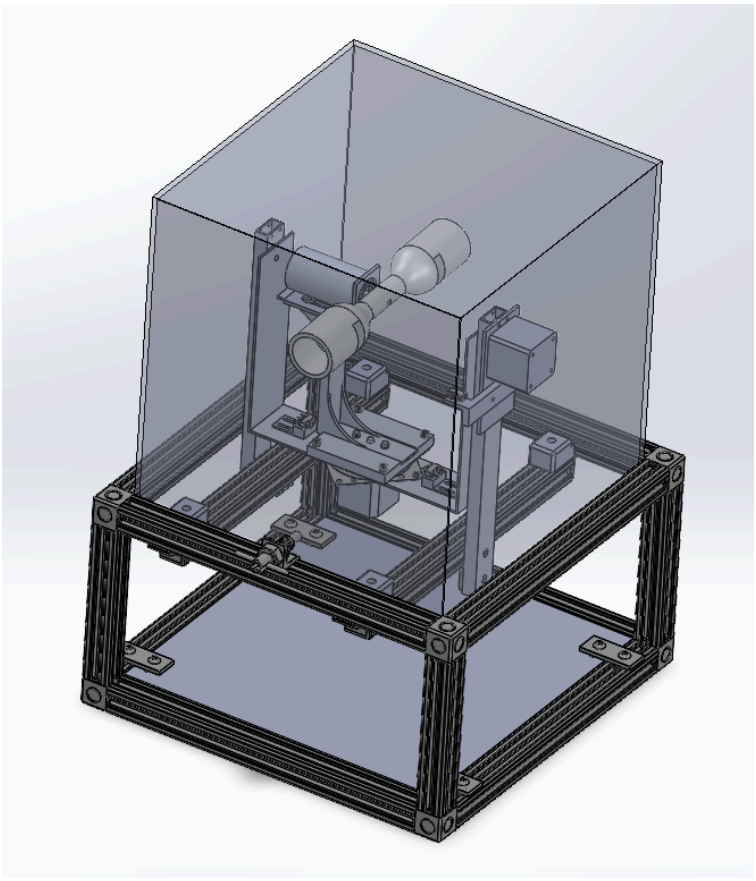


Figure 3.6.6.2: Enclosure Platform With Acrylic Enclosure Over Full Scale Design Model

Seen in Figure 3.6.6.2 is the final design of the enclosure. This design improves upon the former by being more modular. This iteration is made with aluminum T-slotted channels, bolts, and sliding nuts to allow for easy mounting and repositioning of parts. This design allows the third degree supports of the physical system to be mounted directly to the frame. The arms which the assembly is mounted on are easily adjusted by loosening four bolts. The enclosure is also equipped with acrylic side panels to allow visibility for all of the electrical components within. The side acrylic panels are pin mounted while the front and back are mounted through bolts and sliding nuts to allow for correct panel positioning. The back of the enclosure features a 3D printed back plate to allow for connecting various ports to the system. The back plate shown in

Figure 3.6.6.3 accommodates four USB-A connectors, two HDMI connectors, and one USB-C connector along with a slot for the AC power switch. This back plate slides into the T-slotted channel and is bolted to the acrylic plate on the back side of the enclosure. Located at the bottom of the enclosure is a polycarbonate sheet which holds all of the electrical components. Finally, the enclosure rests on top of eight total rubber feet to help prevent movement of the enclosure while the ball spinner is in motion. Along with this, a plate was mounted to the underside of the front lower channel to provide a support plate for the system's emergency stop button.

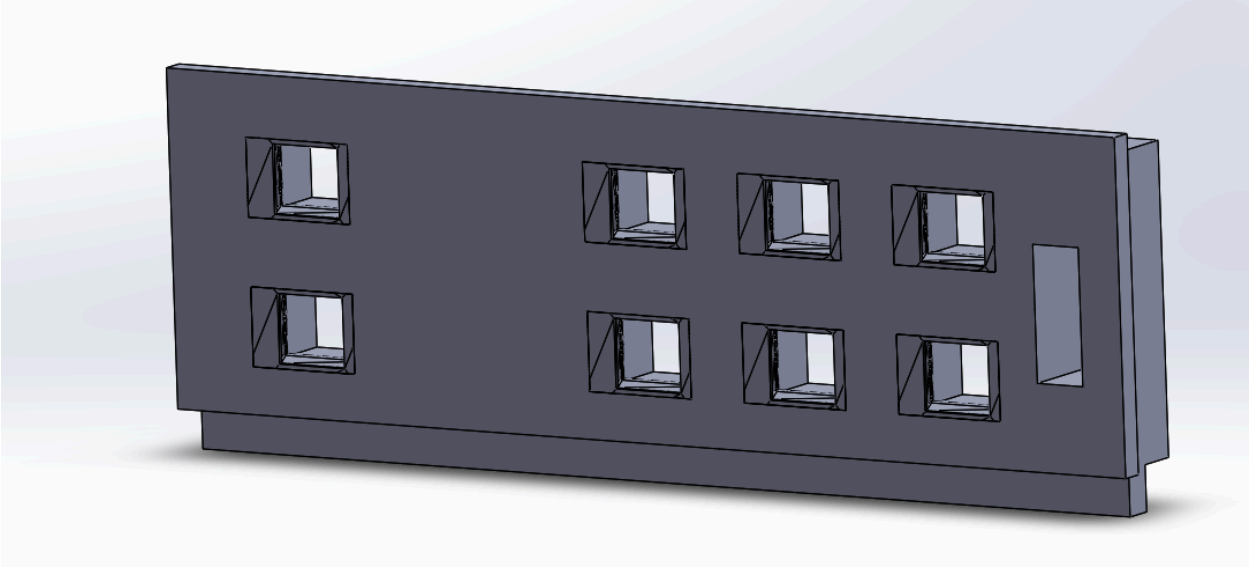


Figure 3.6.6.3: 3D printed back plate

Future Work

Mobile

Ball Spinner Controls

The Ball Spinner isn't currently connected to the mobile application. In the future it will be accessible at least through the desktop version of the app. This way users can control all aspects of the RevMetrix project from the same application.

Improved Stats Page

Currently stats are being computed on the users device. One thing we would like to do is make some of the computations on the cloud so that the phone is not bogged down with complex algorithms. We also need to calculate many more statistics. There are more stats that can be added to the game and second ball stat types specifically. We also plan to add graphical results in the form of charts and graphs. Finally, more testing needs to be done with the stat calculations against Prof. Hake's linescore to verify that the popup is showing accurate information.

Email and Phone Number Verification

In the future, we will implement a way to verify the user's email and phone number. This will send an actual email to the user for account verification and a text to the phone number. This will make the accounts more secure.

Improve MetaMotion S Connection

Because of our use of C# MAUI, there is no official MbientLab API available for the MMS, so we have been communicating with the device entirely through raw bits. Because of this, we must manually build every command, including module IDs, registers, and configuration bitfields, and we also have to decode all of the data packets ourselves. This requires creating our

own low-level communication layer to start sensor streams, parse responses, manage timing, and handle errors. While this makes development more complex and time-consuming, it also gives us full control over the MMS and provides valuable experience in working directly with binary protocols and embedded sensor communication. Moving forward, we want to improve the reliability of our implementations for the magnetometer, accelerometer, gyroscope, and light sensor, as these modules do not always behave as expected. This inconsistency suggests that some parts of our command structure or parsing logic are still incorrect, and refining these fundamentals will be a key part of our future work. This semester it was figured out that the MMC and the MMS have different chips on them. This means that they communicate differently which means we need to create more functions that are MMC or MMS specific, making code more complicated. This will be an ongoing issue.

Shot Page Improvements

The shot page currently can only be viewed in portrait mode. When viewed in landscape mode, the page does not adjust to fill the screen, and the user must scroll up and down to see everything. The frame view also does not indicate splits. There would need to be a circle added over the shot box that is invisible until a split is detected. Additionally, the back end code for the shot page could still be improved to reduce game reloading times and touch reception.

Implement Fixed Cloud Sync

At present, the cloud synchronization feature is in an early and unstable state. While a basic implementation exists, it is not yet fully reliable and contains known issues. Portions of the implementation were rapidly developed using AI-assisted tooling, with some intentional design layered on top, resulting in inconsistent behavior. This component was developed by a

backend/embedded systems engineer without prior experience in cloud architecture, which has contributed to gaps in robustness and scalability. Although the feature functions in certain scenarios, it requires significant refinement. Future improvements should include involvement from a developer with cloud expertise, along with the addition of a comprehensive testing suite to ensure reliability, data integrity, and proper handling of edge cases.

Improved Popups

Most popups have been adjusted to reflect the current app UI/UX, but many of the popups throughout the app still have different text and color schemes. Ideally popups would use a universal style.

Session List Improvements

Prof. Hake would like the session list page to display a series total, as well as a date on each session button. When a session button is selected, the games are shown as orange game buttons with the game number and score shown below the session button. A session summary stats section would also be shown below the game 3 game buttons. An example of this information is shown below:

Frames: 35	Leaves: 12
Strikes: 23/35 (65.7%)	Spares: 8/9 (88.9%)
Pocket: 28/35 (80.0%)	Splits: 0/1 (2)
Carry: 22/28 (78.6%)	Washouts: 1/1 (1)
High: 3	Opens: 3
Light: 2	Fill: 1
Count Ave: 9.125	

Settings Page

There are currently a lot of buttons on the main page of the mobile app. Prof. Hake requested that the buttons for SmartDot, SmartWatch, Account, API Test, Ciclopes Test, and SQL Database be put onto this new page.

Watch

Cross Platform Compatibility

The current build only supports use on watches running on wearOS. In the future, the codebase will be expanded to support watchOS builds. The codebase would include a shared Dart logic layer, and platform specific bluetooth modules, all within the flutter framework.

Integration with Ciclopes

Integrating with the current Ciclopes system could include features such as automated event detection. Instead of the user manually starting the recording from the pre shot phase, it would automatically detect when the bowler begins their approach, as well as autofilling values like position, or hitBoard. The watch could also receive analytics from Ciclopes the same as it receives session information from the phone.

Relative Sizing

In the current implementation, the 44m circular watch is the only size that fits the UI design we have implemented. In the future, adding global sizing that adjusts the UI based on what watch type you have, including square for Apple watch addition. This comes with the addition of cross platform capabilities.

Replay Recorded Video

The current workflow allows the user to start and stop recording on the cellular side via commands sent over BLE in the shot input page. If the video were to be corrupted or the framing was wrong, the user would have no way of knowing that until they went back to the phone after the shot was over. A way to overcome this would be adding a replay feature on the watch itself, allowing the user to make sure the video recorded the way they wanted before they start their next shot.

Cloud

API

In the future, additions and improvements to the API will be necessary, and as the project continues to be more complex, new API endpoints will always be needed. The goal this semester was to be able to set up an environment where adding new API endpoints is streamlined, and in the future developing new API endpoints will be more streamlined. With this new process, having full create/retrieve/update/delete support for all of the new database tables that were implemented is a goal.

CI/CD Pipeline Improvements

Previously, developers had created a CI/CD (continuous integration, continuous development) pipeline through GitHub actions for the backend code. Minor alterations were made to the pipeline this semester, as previously it would always fail and wasn’t deploying pushed code to our Droplet. In the future, there is a need for automated repository layer and API layer testing and having that integrated into the pipeline would be ideal. Additionally, having our database migrations automatically deployed onto our droplet would improve development time

as well as minimize user error when running the migrations from the command line. Along with this, having our pipeline automatically restart the server once the tests finish running successfully would be an improvement.

Migrations

Having automatic database migrations was a major goal for the backend cloud system this semester. The system that was implemented allows developers to create table objects, add them to a migration, push DB changes with automatic rollbacks, and have a version log of the database. This system is an improvement on the previous, but it could be expanded further. In the future, having a sub-system in the cloud server that acts as a pseudo-UI for developers to automatically version and post migration without needing to interface with the migration commands would allow even faster development of new database objects.

Ciclopes

Database Integration / Persistent Data

Currently saving in a “mock database” is toggleable to support the implementation of data visualization overlays, but no substance local or cloud database persistence is implemented. We have formulated that relationally each shot in the database will be the relational key for each further relational key for ball positions and pose coordinate vectors such as a ball_pos_frame_id or pose_frame_id. These would have a frame_idx to support the query logic to keep things temporally consistent, and these will be used for recalculations on query rather than saving end results as the data size for end resulting positions and kinematics/position statistics scales very quickly, and recomputation time is negligible in the end-to-end query. The infrastructure exists for implementation we just did not have time during the semester.

On-Device Inference

We have determined that on-device inference and implementation of the ball trajectory generation workflow allows a better user experience and removes the need for connection to a cloud API service, and further renting of a GPU-containing service. In terms of inference of the model, the nano size was chosen to allow this feature and on modern NPU containing devices inference and further computations will be minimal. The user experience can now implement pre recording setup checks and further functionalities to reduce user error which preludes many failure modes such as bad camera setup or others. Also this simplifies the system architecture to not require expensive GPU services for basic functionalities, in production this would allow for extensive free-tier functionalities, and pose estimation and further new functionalities would be paid only.

User Experience / Workflow Alignment

The current flow of record, and then using ciclopes is mainly for our demonstrations and ease of development, not optimized for use by the end user in the normal bowler workflow. This requires testing from real users and data collection of how bowlers actually go about their processes. There are many points where UX can be improved exponentially.

IoT Camera / Sensor Array For Bowling Alley Integration

We propose a bowling alley hardware product which looks very much like the Ciclopes mobile integration, but with an IoT array of camera modules above and around the bowling alley to support the ball trajectory feature. Specifically a main constraint is the monocular video collection, if the array were to be implemented correctly there could always be multiple cameras in use to provide more perspectives to improve the accuracy of the ball trajectory generation. This functionality is inspired by Specto, a product installed on specific bowling lanes using lidar

to provide within the inch measurements of the ball at high frequency giving the exact trajectory of the ball, and the IoT camera array alongside the Ciclopes algorithm formulation would allow the same functionality at a fraction of the price. Accuracy would need to be measured alongside Specto to give a rating, but the more cameras, the higher frame rate, and higher the resolution the higher accuracy Ciclopes can theoretically have. Currently this is a theoretical research formulation, and needs to be tested in practice to be validated.

Ball Spinner Controller

SmartDot Connection

When booting the RPi5 and establishing a connection to a MetaMotionS, the initial connection attempt consistently fails, while a second attempt typically succeeds. Our current workaround is to retry the connection once, which is reliable as long as BLE communication behaves normally. Implementing a method to cache or remember the MetaMotionS device within the application may reduce this latency and improve future connection performance [44].

Limit Switch Functionality Improvement

Currently the limit switches are polled rather than being interrupt driven. This leads to issues of them getting stuck and not being able to function properly if the Ball Spinner attempts to start moving while the limit switch is pressed. Switching to an interrupt A driven model should remedy these issues.

Motor Tuning

Currently the PID curve of the shot is improperly tuned to a previous motor configuration. A proper PID implementation should allow the Ball Spinner to better meet its desired specifications. In addition the Stepper motor has an issue of skipping steps when changing speeds rapidly.

Shot Replaying and Navigation Oversights.

The current implementation of the Ball Spinner Controller contains some navigation oversights. Currently there is no way to replay a shot without saving the shot to the database. In addition the process for starting a diagnostic session requires activating the motors and starting the session. The only indication for this is the help menu.

Ball Spinner Hardware System

Indicator Lights

Indicator lights have been added to the Ball Spinner enclosure so the operational state of the system can be monitored without opening the enclosure. These lights will provide clear visual feedback for conditions such as system armed, motors running, and motor faults. Including external indicators improves overall safety by ensuring that the operator can verify system status at a glance before interacting with the device or initiating motion. The exact placement and quantity of the indicators will be finalized once the internal layout is fully defined.

Figure 3.6.6.4: Indicator lights

Safety Features

Safety features will be incorporated into the system to ensure reliable and controlled operation. Fuses will be added to protect each major power path and prevent damage in the event of an electrical fault. Dedicated buttons will be installed to enable or disable the motors so the operator has direct control over when motion can occur. An emergency stop will also be included to immediately cut power to the motors if unsafe conditions arise. These safety components will create multiple layers of protection and will help maintain safe operation during testing and long term use.

Emergency Stop (E-Stop)

An emergency stop (E-stop) was implemented to immediately remove power from all high power and actuation components in the system. When activated, the E-stop disconnects power to the motors, motor drives, VESC, and indicator LEDs, effectively halting all physical system activity.

The Raspberry Pi and HMI remain powered during an E-stop condition. This design allows the controller to maintain operation for potential fault logging, system monitoring, and controlled recovery procedures after the fault condition is cleared. By isolating the power domains, the system ensures both immediate physical safety and retention of critical software functionality.

Figure 3.6.6.5: E-Stop

Printed Circuit Board (PCB)

A custom printed circuit board (PCB) will be developed in future iterations to replace the current point-to-point wiring and prototyping hardware. The PCB will centralize power distribution, logic-level signal routing, and connector interfaces for the Raspberry Pi, motor drivers, sensors and safety components. This will improve system reliability by reducing loose connections and wiring errors.

The PCB integrates voltage regulation (24 V to 5 V), logic-level shifting for motor control signals, indicator LEDs, and dedicated connectors for stepper drivers, the VESC, and sensors. Standardized connectors will also allow motors and drivers to be swapped or upgraded without redesigning the entire system. Overall, transitioning to a PCB will improve repeatability, safety, and maintainability while supporting long-term scalability of the Ball Spinner controller.

Communication expansion

Future versions of the system will expand communication capabilities beyond the current implementation to support more robust, real-time data exchange and multidevice coordination. We plan to transition additional motor and sensor communication from USB-based interface to UART or CAN bus communication. This would reduce latency, improve reliability and allow multiple devices to share a common communication backbone.

Expanded communication will also enable improved synchronization between axes, higher rate telemetry logging and tighter closed-loop control using sensor feedback. Support for CAN bus, in particular, would allow multiple sensors to operate on the same network with

standardized messaging and fault reporting. These upgrades will improve system performance, diagnostic capability, and future integration with additional motors or external data collection systems.

Ball Spinner Mechanical System

Physical Prototype of Aluminum System

The primary future goal for the Ball Spinner system is to improve and update the current prototype with the suggestions mentioned below. As mentioned in the current implementation section of this report, the current design uses stock parts available on McMaster-Carr in Appendix A[2][3].

Buffers

To prevent motors from moving past their maximum range of motion, physical buffers, or hard stops, should be improved. This will be an extra failsafe to go along with the digital limit switches that are a part of the Ball Spinner Controller.

Motor Mounting

To prevent motors from slipping and to allow for accurate motion, improved motor mounting should be considered in the future, especially for the second degree motor. Friction fitting onto motor shafts is not a feasible solution for a system that needs to run for any length of time. Shaft collars and set screws should be considered for more reliable motor mounting. A shaft collar that can be placed between the second degree rotational plate and the third degree u bracket, inside the bearing, would be sufficient.

3rd Degree Motor

While counterbalancing did allow for 3rd degree motion, a larger 3rd degree motor should be considered to more reliably reach the desired angular displacement. In order to accommodate this, the whole assembly should be rotated 45 degrees to utilize the extra space that would be provided by orienting the assembly corner-to-corner instead of side-to-side. This will also provide extra spacing for motor encoders on the second and third degree stepper motors.

3D Printing Methods

Polycarbonate is a difficult material to print accurately and consistently. The physical team ran into issues with warping and tolerances throughout the build phase. In order to save time and get more consistent prints, more research should be done on 3D printing polycarbonate. The best results were achieved with print head temperature at 260 C, print bed temperature at 110 C, fan speed at 0%, and speed at 1100 mm/min within an enclosed printer. For the Prusa mk3, crash detection must be turned off at a high bed temperature. Additionally, research into setting enclosure temperature to a specific value should be conducted.

BSC Enclosure Platform

Heat from the motors was not a major issue during operation of the system. However, ventilation should be improved to ensure electrical components do not overheat. Another issue was port connections to the back plate, as some of the inserts fit well into the back plate and others did not or were held loosely in place. The dimensions or design of the back plate should be updated to ensure a secure fit for all ports. Finally, motor wiring should be improved to ensure wires do not interfere with the motion of the ball spinner, which is especially true for the 1st degree motor wires.

Acrylic Replacement

As the system has been used, cracking and stress marks have been observed at various locations on the acrylic parts. Several cracks (~1 in) where the acrylic box rests on the stands and radial cracking around where bolts have been tightened against the acrylic side panels have both occurred. To prevent this in the future, acrylic parts should be replaced with polycarbonate or another similar material that is more flexible than acrylic. The upper enclosure could be manufactured from polycarbonate using solvent bonding of flat sheets.

References

[1] Donald Hake II. "A Roll Down The Lane" [Thesis Paper]. Available at: [https://ycpcs.github.io/cs400-fall2024/projects/RevMetrix-Project/Hake-MEngESci-Masters-Thesis.pdf].

[2] Raspberry PI 4. (2024). Raspberry PI. [Online]. Available: https://www.raspberrypi.com/

[3] MetaMotion S. Available at: [https://mbientlab.com/store/metamotions/].

[4] Docker. (2024). docker. [Online]. Available: https://www.docker.com/

[5] Digital Ocean. (2024). Digital Ocean. [Online]. Available: https://www.DigitalOcean.com/

[6] Nginx Proxy Manager. (2024). [Online]. Available: https://nginxproxymanager.com

[7] .NET. (9.0.0). Microsoft. [Online]. Available: https://dotnet.microsoft.com/en-us/

[8] .NET MAUI. (9.0.40). Microsoft. [Online]. Available: https://dotnet.microsoft.com/en-us/apps/maui

[9] ASP.NET. (2024) Microsoft [Online]. Available: https://dotnet.microsoft.com/en-us/apps/aspnet

[10] TINACloud. (2024). DesignSoft Tina. [Online]. Available: https://www.tina.com/tinacloud/

[11] SolidWorks. (2024). SolidWorks Corporation. [Online]. Available: https://www.solidworks.com/

[12] GitHub.(2024). GitHub. [Online]. Available: https://github.com

[13] Draw.io. (2024). Draw.io Diagrams. [Online]. Available: https://app.diagrams.net/

[14] Microsoft SQL Server. (2024). Microsoft. [Online]. Available https://www.microsoft.com/en-us/sql-server/

[15] RevMetrix. (2024). RevMetrix [Online]. Available: https://docs.RevMetrix.io/

[16] ListView. (2024). Microsoft. [Online]. Available: https://learn.microsoft.com/en-us/dotnet/maui/user-interface/controls/listview

[17] Entry. (2024). Microsoft. [Online]. Available: https://learn.microsoft.com/en-us/dotnet/maui/user-interface/controls/entry

[18] DisplayAlert. (2024). Microsoft. [Online]. Available: https://learn.microsoft.com/en-us/dotnet/api/microsoft.maui.controls.page.displayalert

[19] Picker. (2025). Microsoft. [Online]. Available: https://learn.microsoft.com/en-us/dotnet/maui/user-interface/controls/picker?view=net-maui-9.0

[20] NavigationPage. (2024). Microsoft. [Online]. Available: https://learn.microsoft.com/en-us/dotnet/maui/user-interface/pages/navigationpage?view=net-maui-9.0

[21] BCrypt.Net-Next. (2022). Nuget. [Online]. Available: https://www.nuget.org/packages/BCrypt.Net-Next

[22] CollectionView. (2024). Microsoft. [Online]. Available: https://learn.microsoft.com/en-us/dotnet/maui/user-interface/controls/collectionview/?view=net-maui-9.0

[23] BoxView. (2024). Microsoft. [Online]. Available:

https://learn.microsoft.com/en-us/dotnet/maui/user-interface/controls/boxview?view=net-maui-9.0

[24] Label. (2024). Microsoft. [Online]. Available:

https://learn.microsoft.com/en-us/dotnet/maui/user-interface/controls/label?view=net-maui-9.0

[25] Slider. (2024). Microsoft. [Online]. Available:

https://learn.microsoft.com/en-us/dotnet/maui/user-interface/controls/slider?view=net-maui-9.0

[26] CameraView. (2026). [Online]. Available:

https://learn.microsoft.com/en-us/dotnet/communitytoolkit/maui/views/camera-view?tabs=android

[27] Flutter Development Guide. (2023). [Online]. Available:

https://blog.flutter.wtf/wearable-app-development/

[28] WearOS Development Guide. (2025). [Online]. Available:

https://developer.android.com/training/wearables/get-started/creating

[29] Flutter App Development Guide for WearOS. (2025). [Online]. Available:

https://vibe-studio.ai/insights/building-apps-for-wear-os-and-watchos-using-flutter

[30] Meta AI Demos. (2025). Meta [Online]. Available:

https://aidemos.meta.com/segment-anything/gallery/

[31] Flutter Bluetooth Tutorial. (2025). [Online]. Available:

https://mobisoftinfotech.com/resources/blog/flutter-development/flutter-bluetooth-ble-integration-guide

[32] Bluetooth Protocol Stack. (2025). [Online]. Available:

https://www.mathworks.com/help/bluetooth/gs/bluetooth-protocol-stack.html

[33] Widget Fundamentals. (2025). [Online]. Available:

https://docs.flutter.dev/get-started/fundamentals/widgets

[34] Layout Fundamentals. (2025). [Online]. Available:

https://docs.flutter.dev/get-started/fundamentals/layout

[35] State Management Fundamentals. (2025). [Online]. Available:

https://docs.flutter.dev/get-started/fundamentals/state-management

[36] User Input Fundamentals. (2025). [Online]. Available:

https://docs.flutter.dev/get-started/fundamentals/user-input

[37] Local Caching Fundamentals. (2025). [Online]. Available:

https://docs.flutter.dev/get-started/fundamentals/local-caching

[38] Flutter Blue Plus. (2025). [Online]. Available:

https://pub.dev/packages/flutter_blue_plus

[39] Dart Tutorial. (2025). [Online]. Available:

https://www.geeksforgeeks.org/dart/dart-tutorial/

[40] AutoCad. (2025). [Online]. Available:

https://www.autodesk.com

[41] MetaWear-SDK-Python. (2025). [Online]. Available:

https://github.com/mbientlab/MetaWear-SDK-Python

[42] YCP-Rev-Metrix. (2025). [Online]. Available:

https://github.com/YCP-Rev-Metrix/BallSpinner-Controller-v2/blob/main/README.md

[43] MbientLab Documentation. (2025). [Online]. Available:

https://mbientlab.com/iosdocs/latest/metawearscanner.html

[45] Handson Technology. (2024). 17HS4401S NEMA 17 Stepper Motor Datasheet. [Online].

Available:

https://www.handsontec.com/dataspecs/17HS4401S.pdf

[46] XC ESC. (2025). E3665 Sensored Brushless DC Motor for RC Applications. [Online]. Available:

https://www.xc-esc.com/product/e3665-sensored-brushless-motor-for-rc-cars/

[47] Manuals+. (2025). DM542 Digital Stepper Driver User Manual. [Online]. Available:

https://manuals.plus/asin/B08GCFNQF2

[48] Vedder, B. (2025). VESC® Open Source Motor Controller Documentation. [Online]. Available:

https://vesc-project.com/node/309

[49] Flipsky. (2025). Mini FSESC 4.20 50A Brushless Motor Controller. [Online]. Available:

https://www.flipskyo.com/products/mini-fsesc4-20-50a-with-anodized-heat-sink

[50] TwinFly. (2025). P1-150-24 24V DC Power Supply. [Online]. Available:

https://www.radwell.com/Buy/TWINFLY/TWINFLY/P1-150-24

Appendix A: Parts Used for Design

[1] Amazon.com: 12 Inch Clear Acrylic Display Box Versatile Square No Lid Plexiglass Retail Product Storage Bin or Merchandise Riser with One Open Side No Assembly Required by Marketing Holders : Industrial & Scientific

[2] https://www.mcmaster.com/6546K51

[3]https://www.mcmaster.com/5679N35

Team RevMetrix (pictured left to right): Prof. Hake, Andrew Olvera, Jakeb Nielsen, Charles Carroll, Josh Byers, Dr. Moscola, Hunter Wolfe, Zach Cox, Dr. Babcock, Matthew Brown, Gabe Manero, Gavin Wentz, and Joseph Downey